VAX MACRO and Instruction Set Reference Manual

Order Number: AA-LA89A-TE

April 1988

This document describes the features of the VAX MACRO instruction set and assembler. It includes a detailed description of MACRO directives and instructions, as well as information about MACRO source program syntax.

Revision/Update Information: This manual supersedes the *VAX MACRO and Instruction Set Reference Manual*, Version 4.0.

Software Version: VMS Version 5.0

digital equipment corporation
maynard, massachusetts

April 1988

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

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	DIBOL	UNIBUS
DEC/CMS	EduSystem	VAX
DEC/MMS	IAS	VAXcluster
DECnet	MASSBUS	VMS
DECsystem-10	PDP	VT
DECSYSTEM-20	PDT	
DECUS	RSTS	
DECwriter	RSX	digital™

ZK4515

HOW TO ORDER ADDITIONAL DOCUMENTATION
DIRECT MAIL ORDERS

USA & PUERTO RICO*	CANADA	INTERNATIONAL
Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061	Digital Equipment of Canada Ltd. 100 Herzberg Road Kanata, Ontario K2K 2A6 Attn: Direct Order Desk	Digital Equipment Corporation PSG Business Manager c/o Digital's local subsidiary or approved distributor

In Continental USA and Puerto Rico call 800-258-1710.

In New Hampshire, Alaska, and Hawaii call 603-884-6660.

In Canada call 800-267-6215.

*Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575).

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Westminster, Massachusetts 01473.

Production Note

This book was produced with the VAX DOCUMENT electronic publishing system, a software tool developed and sold by DIGITAL. In this system, writers use an ASCII text editor to create source files containing text and English-like code; this code labels the structural elements of the document, such as chapters, paragraphs, and tables. The VAX DOCUMENT software, which runs on the VMS operating system, interprets the code to format the text, generate a table of contents and index, and paginate the entire document. Writers can print the document on the terminal or line printer, or they can use DIGITAL-supported devices, such as the LN03 laser printer and PostScript™ printers (PrintServer 40 or LN03R ScriptPrinter), to produce a typeset-quality copy containing integrated graphics.

™ PostScript is a trademark of Adobe Systems, Inc.

Contents

PREFACE xix

NEW AND CHANGED FEATURES xxiii

VAX MACRO LANGUAGE

CHAPTER 1 INTRODUCTION 1–1

CHAPTER 2 MACRO SOURCE STATEMENT FORMAT 2–1

2.1 LABEL FIELD 2–2

2.2 OPERATOR FIELD 2–3

2.3 OPERAND FIELD 2–3

2.4 COMMENT FIELD 2–3

CHAPTER 3 COMPONENTS OF MACRO SOURCE STATEMENTS 3–1

3.1 CHARACTER SET 3–1

3.2 NUMBERS 3–2
3.2.1 Integers 3–3
3.2.2 Floating-Point Numbers 3–3
3.2.3 Packed Decimal Strings 3–4

3.3 SYMBOLS 3–4
3.3.1 Permanent Symbols 3–4
3.3.2 User-Defined Symbols and Macro Names 3–5
3.3.3 Determining Symbol Values 3–6

3.4	LOCAL LABELS	3–7
3.5	TERMS AND EXPRESSIONS	3–9
3.6	UNARY OPERATORS	3–10
3.6.1	Radix Control Operators	3–11
3.6.2	Textual Operators	3–12
3.6.2.1	ASCII Operator • 3–13	
3.6.2.2	Register Mask Operator • 3–13	
3.6.3	Numeric Control Operators	3–14
3.6.3.1	Floating-Point Operator • 3–14	
3.6.3.2	Complement Operator • 3–15	
3.7	BINARY OPERATORS	3–15
3.7.1	Arithmetic Shift Operator	3–16
3.7.2	Logical AND Operator	3–16
3.7.3	Logical Inclusive OR Operator	3–16
3.7.4	Logical Exclusive OR Operator	3–16
3.8	DIRECT ASSIGNMENT STATEMENTS	3–17
3.9	CURRENT LOCATION COUNTER	3–17
CHAPTER 4	MACRO ARGUMENTS AND STRING OPERATORS	4–1
4.1	ARGUMENTS IN MACROS	4–1
4.2	DEFAULT VALUES	4–2
4.3	KEYWORD ARGUMENTS	4–3
4.4	STRING ARGUMENTS	4–3
4.5	ARGUMENT CONCATENATION	4–5
4.6	PASSING NUMERIC VALUES OF SYMBOLS	4–6
4.7	CREATED LOCAL LABELS	4–7

4.8	MACRO STRING OPERATORS	4–8
4.8.1	%LENGTH Operator	4–8
4.8.2	%LOCATE Operator	4–9
4.8.3	%EXTRACT Operator	4–10

CHAPTER 5 MACRO ADDRESSING MODES 5–1

5.1	GENERAL REGISTER MODES	5–1
5.1.1	Register Mode	5–4
5.1.2	Register Deferred Mode	5–5
5.1.3	Autoincrement Mode	5–5
5.1.4	Autoincrement Deferred Mode	5–6
5.1.5	Autodecrement Mode	5–7
5.1.6	Displacement Mode	5–8
5.1.7	Displacement Deferred Mode	5–9
5.1.8	Literal Mode	5–10
5.2	PROGRAM COUNTER MODES	5–12
5.2.1	Relative Mode	5–12
5.2.2	Relative Deferred Mode	5–13
5.2.3	Absolute Mode	5–14
5.2.4	Immediate Mode	5–14
5.2.5	General Mode	5–15
5.3	INDEX MODE	5–16
5.4	BRANCH MODE	5–18

CHAPTER 6 MACRO ASSEMBLER DIRECTIVES 6–1

.ADDRESS	6–4
.ALIGN	6–5
.ASCI*X*	6–7
.ASCIC	6–8
.ASCID	6–9
.ASCII	6–10
.ASCIZ	6–11
.BLK*X*	6–12
.BYTE	6–14
.CROSS	6–16

.DEBUG	6–18
.DEFAULT	6–19
.D_FLOATING	6–20
.DISABLE	6–21
.ENABLE	6–22
.END	6–25
.ENDC	6–26
.ENDM	6–27
.ENDR	6–28
.ENTRY	6–29
.ERROR	6–31
.EVEN	6–32
.EXTERNAL	6–33
.F_FLOATING	6–34
.G_FLOATING	6–35
.GLOBAL	6–36
.H_FLOATING	6–37
.IDENT	6–38
.IF	6–39
.IF_X	6–42
.IIF	6–45
.IRP	6–46
.IRPC	6–48
.LIBRARY	6–50
.LINK	6–51
.LIST	6–54
.LONG	6–55
.MACRO	6–56
.MASK	6–58
.MCALL	6–59
.MDELETE	6–60
.MEXIT	6–61
.NARG	6–62
.NCHR	6–63
.NLIST	6–64
.NOCROSS	6–65
.NOSHOW	6–66
.NTYPE	6–67
.OCTA	6–69
.ODD	6–70
.OPDEF	6–71
.PACKED	6–73

.PAGE	6–74
.PRINT	6–75
.PSECT	6–76
.QUAD	6–80
.REFN	6–81
.REPEAT	6–82
.RESTORE_PSECT	6–84
.SAVE_PSECT	6–85
.SHOW	6–87
.SIGNED_BYTE	6–89
.SIGNED_WORD	6–90
.SUBTITLE	6–92
.TITLE	6–93
.TRANSFER	6–94
.WARN	6–97
.WEAK	6–98
.WORD	6–99

VAX DATA TYPES AND INSTRUCTION SET

CHAPTER 7 TERMINOLOGY AND CONVENTIONS 7–1

7.1 NUMBERING 7–1

7.2 UNPREDICTABLE AND UNDEFINED 7–1

7.3 RANGES AND EXTENTS 7–1

7.4 MBZ 7–1

7.5 RESERVED 7–2

7.6 FIGURE DRAWING CONVENTIONS 7–2

CHAPTER 8	BASIC ARCHITECTURE	8–1
8.1	VAX ADDRESSING	8–1
8.2	DATA TYPES	8–1
8.2.1	Byte	8–1
8.2.2	Word	8–1
8.2.3	Longword	8–2
8.2.4	Quadword	8–2
8.2.5	Octaword	8–3
8.2.6	F_floating	8–3
8.2.7	D_floating	8–4
8.2.8	G_floating	8–4
8.2.9	H_floating	8–5
8.2.10	Variable-Length Bit Field	8–5
8.2.11	Character String	8–7
8.2.12	Trailing Numeric String	8–7
8.2.13	Leading Separate Numeric String	8–11
8.2.14	Packed Decimal String	8–12
8.3	PROCESSOR STATUS LONGWORD (PSL)	8–13
8.3.1	C Bit	8–14
8.3.2	V Bit	8–14
8.3.3	Z Bit	8–14
8.3.4	N Bit	8–14
8.3.5	T Bit	8–14
8.3.6	IV Bit	8–14
8.3.7	FU Bit	8–14
8.3.8	DV Bit	8–15
8.4	PERMANENT EXCEPTION ENABLES	8–15
8.4.1	Divide by Zero	8–15
8.4.2	Floating Overflow	8–15
8.5	INSTRUCTION AND ADDRESSING MODE FORMATS	8–15
8.5.1	Opcode Formats	8–15
8.5.2	Operand Specifiers	8–16
8.6	GENERAL ADDRESSING MODE FORMATS	8–17
8.6.1	Register Mode	8–17
8.6.2	Register Deferred Mode	8–18
8.6.3	Autoincrement Mode	8–18

8.6.4 Autoincrement Deferred Mode 8–19
8.6.5 Autodecrement Mode 8–19
8.6.6 Displacement Mode 8–20
8.6.7 Displacement Deferred Mode 8–20
8.6.8 Literal Mode 8–21
8.6.9 Index Mode 8–23

8.7 SUMMARY OF GENERAL MODE ADDRESSING 8–25

8.8 BRANCH MODE ADDRESSING FORMATS 8–26

CHAPTER 9 VAX INSTRUCTION SET 9–1

9.1 INTRODUCTION 9–1

9.2 INSTRUCTION DESCRIPTIONS 9–1
9.2.1 Operand Specifier Notation 9–2
9.2.2 Operation Description Notation 9–3

9.3 INTEGER ARITHMETIC AND LOGICAL INSTRUCTIONS 9–5
ADAWI 9–7
ADD 9–8
ADWC 9–9
ASH 9–10
BIC 9–11
BIS 9–12
BIT 9–13
CLR 9–14
CMP 9–15
CVT 9–16
DEC 9–17
DIV 9–18
EDIV 9–19
EMUL 9–20
INC 9–21
MCOM 9–22
MNEG 9–23
MOV 9–24
MOVZ 9–25
MUL 9–26

	PUSHL	9–27	
	ROTL	9–28	
	SBWC	9–29	
	SUB	9–30	
	TST	9–31	
	XOR	9–32	
9.4	ADDRESS INSTRUCTIONS		9–33
	MOVA	9–34	
	PUSHA	9–35	
9.5	VARIABLE-LENGTH BIT FIELD INSTRUCTIONS		9–36
	CMP	9–38	
	EXT	9–39	
	FF	9–40	
	INSV	9–41	
9.6	CONTROL INSTRUCTIONS		9–42
	ACB	9–44	
	AOBLEQ	9–46	
	AOBLSS	9–47	
	B	9–48	
	BB	9–50	
	BB	9–51	
	BB	9–52	
	BLB	9–53	
	BR	9–54	
	BSB	9–55	
	CASE	9–56	
	JMP	9–58	
	JSB	9–59	
	RSB	9–60	
	SOBGEQ	9–61	
	SOBGTR	9–62	
9.7	PROCEDURE CALL INSTRUCTIONS		9–63
	CALLG	9–65	
	CALLS	9–67	
	RET	9–69	
9.8	MISCELLANEOUS INSTRUCTIONS		9–70

	BICPSW	9–71	
	BISPSW	9–72	
	BPT	9–73	
	HALT	9–74	
	INDEX	9–75	
	MOVPSL	9–77	
	NOP	9–78	
	POPR	9–79	
	PUSHR	9–80	
	XFC	9–81	
9.9	QUEUE INSTRUCTIONS		9–82
9.9.1	Absolute Queues		9–82
9.9.2	Self-Relative Queues		9–85
9.9.3	Instruction Descriptions		9–88
	INSQHI	9–89	
	INSQTI	9–91	
	INSQUE	9–93	
	REMQHI	9–95	
	REMQTI	9–97	
	REMQUE	9–99	
9.10	FLOATING POINT INSTRUCTIONS		9–101
9.10.1	Introduction		9–101
9.10.2	Overview of the Instruction Set		9–102
9.10.3	Accuracy		9–103
9.10.4	Instruction Descriptions		9–104
	ADD	9–106	
	CLR	9–107	
	CMP	9–108	
	CVT	9–109	
	DIV	9–112	
	EMOD	9–114	
	MNEG	9–116	
	MOV	9–117	
	MUL	9–118	
	POLY	9–119	
	SUB	9–122	
	TST	9–123	
9.11	CHARACTER STRING INSTRUCTIONS		9–124
	CMPC	9–126	

	LOCC	9–128	
	MATCHC	9–129	
	MOVC	9–130	
	MOVTC	9–132	
	MOVTUC	9–133	
	SCANC	9–135	
	SKPC	9–136	
	SPANC	9–137	
9.12	CYCLIC REDUNDANCY CHECK INSTRUCTION		9–138
	CRC	9–139	
9.13	DECIMAL STRING INSTRUCTIONS		9–141
9.13.1	Decimal Overflow		9–142
9.13.2	Zero Numbers		9–142
9.13.3	Reserved Operand Exception		9–142
9.13.4	UNPREDICTABLE Results		9–142
9.13.5	Packed Decimal Operations		9–143
9.13.6	Zero-Length Decimal Strings		9–143
9.13.7	Instruction Descriptions		9–143
	ADDP	9–145	
	ASHP	9–147	
	CMPP	9–149	
	CVTLP	9–150	
	CVTPL	9–151	
	CVTPS	9–152	
	CVTPT	9–154	
	CVTSP	9–156	
	CVTTP	9–157	
	DIVP	9–159	
	MOVP	9–161	
	MULP	9–162	
	SUBP	9–163	
9.14	THE EDITPC INSTRUCTION AND ITS PATTERN OPERATORS		9–165
	EDITPC	9–166	
	EO$ADJUST_INPUT	9–171	
	EO$BLANK_ZERO	9–172	
	EO$END	9–173	
	EO$END_FLOAT	9–174	
	EO$FILL	9–175	

	EO$FLOAT	9–176	
	EO$INSERT	9–177	
	EO$LOAD_	9–178	
	EO$MOVE	9–179	
	EO$REPLACE_SIGN	9–180	
	EO$_SIGNIF	9–181	
	EO$STORE_SIGN	9–182	
9.15	OTHER VAX INSTRUCTIONS		9–183
	PROBE*X*	9–184	
	CHM	9–186	
	REI	9–188	
	LDPCTX	9–189	
	SVPCTX	9–190	
	MTPR	9–191	
	MFPR	9–192	
	BUG	9–193	

APPENDIX A ASCII CHARACTER SET A–1

APPENDIX B HEXADECIMAL/DECIMAL CONVERSION B–1

B.1	HEXADECIMAL TO DECIMAL	B–1
B.2	DECIMAL TO HEXADECIMAL	B–1
B.3	POWERS OF 2 AND 16	B–2

APPENDIX C VAX MACRO ASSEMBLER DIRECTIVES AND LANGUAGE SUMMARY C–1

C.1	ASSEMBLER DIRECTIVES	C–1
C.2	SPECIAL CHARACTERS	C–6
C.3	OPERATORS	C–7
C.3.1	Unary Operators	C–7

C.3.2	Binary Operators	C–8
C.3.3	Macro String Operators	C–8
C.4	ADDRESSING MODES	C–9
APPENDIX D	PERMANENT SYMBOL TABLE	D–1
D.1	OPCODES (ALPHABETIC ORDER)	D–1
D.2	OPCODES (NUMERIC ORDER)	D–10
APPENDIX E	EXCEPTIONS	E–1
E.1	ARITHMETIC TRAPS AND FAULTS	E–1
E.1.1	Integer Overflow Trap	E–2
E.1.2	Integer Divide-by-Zero Trap	E–2
E.1.3	Floating Overflow Trap	E–2
E.1.4	Divide-by-Zero Trap	E–2
E.1.5	Floating Underflow Trap	E–2
E.1.6	Decimal String Overflow Trap	E–3
E.1.7	Subscript-Range Trap	E–3
E.1.8	Floating Overflow Fault	E–3
E.1.9	Divide-by-Zero Floating Fault	E–3
E.1.10	Floating Underflow Fault	E–3
E.2	MEMORY MANAGEMENT EXCEPTIONS	E–3
E.2.1	Access Control Violation Fault	E–4
E.2.2	Translation Not Valid Fault	E–4
E.3	EXCEPTIONS DETECTED DURING OPERAND REFERENCE	E–4
E.3.1	Reserved Addressing Mode Fault	E–4
E.3.2	Reserved Operand Exception	E–4
E.4	EXCEPTIONS OCCURRING AS THE CONSEQUENCE OF AN INSTRUCTION	E–5
E.4.1	Reserved or Privileged Instruction Fault	E–5
E.4.2	Operand Reserved to Customers Fault	E–6
E.4.3	Instruction Emulation Exceptions	E–6
E.4.4	Compatibility Mode Exception	E–6

E.4.5	Change Mode Trap	E–7
E.4.6	Breakpoint Fault	E–7
E.5	TRACE FAULT	E–8
E.5.1	Trace Operation When Entering a Change Mode Instruction	E–9
E.5.2	Trace Operation Upon Return From Interrupt	E–9
E.5.3	Trace Operation After a BISPSW Instruction	E–9
E.5.4	Trace Operation After a CALLS or CALLG Instruction	E–9
E.6	SERIOUS SYSTEM FAILURES	E–9
E.6.1	Kernel Stack Not Valid Abort	E–10
E.6.2	Interrupt Stack Not Valid Halt	E–10
E.6.3	Machine Check Exception	E–10

INDEX

FIGURES

6–1	Using Transfer Vectors	6–95
E–1	Compatibility Mode Exception Stack Frame	E–6

TABLES

3–1	Special Characters Used in VAX MACRO Statements	3–1
3–2	Separating Characters in VAX MACRO Statements	3–2
3–3	Unary Operators	3–11
3–4	Binary Operators	3–15
5–1	Addressing Modes	5–2
5–2	Floating-Point Literals Expressed as Decimal Numbers	5–11
5–3	Floating-Point Literals Expressed as Rational Numbers	5–11
5–4	Index Mode Addressing	5–18
6–1	Summary of General Assembler Directives	6–1
6–2	Summary of Macro Directives	6–3
6–3	.ENABLE and .DISABLE Symbolic Arguments	6–22
6–4	Condition Tests for Conditional Assembly Directives	6–40
6–5	Operand Descriptors	6–71
6–6	Program Section Attributes	6–76
6–7	Default Program Section Attributes	6–78
6–8	.SHOW and .NOSHOW Symbolic Arguments	6–87

8–1	Representation of Least-Significant Digit and Sign in Zoned Numeric Format	8–9
8–2	Representation of Least-Significant Digit and Sign in Overpunch Format	8–10
8–3	Floating-Point Literals Expressed as Decimal Numbers	8–23
8–4	Floating-Point Literals Expressed as Rational Numbers	8–23
8–5	General Register Addressing	8–25
8–6	Program Counter Addressing	8–26
9–1	Summary of EDITPC Pattern Operators	9–168
9–2	EDITPC Pattern Operator Encoding	9–169
A–1	Decimal, Hexadecimal, and ASCII Conversion	A–1
B–1	Hexadecimal to Decimal Conversion	B–2
C–1	Assembler Directives	C–1
C–2	Special Characters Used in VAX MACRO Statements	C–6
C–3	Unary Operators	C–7
C–4	Binary Operators	C–8
C–5	Macro String Operators	C–8
C–6	Addressing Modes	C–9
D–1	Opcodes and Functions	D–1
D–2	One-Byte Opcodes	D–10
D–3	Two-Byte Opcodes	D–14
E–1	Arithmetic Exception Type Codes	E–1
E–2	Compatibility Mode Exception Type Codes	E–7

Preface

This manual describes the VAX MACRO language and the VAX instruction set. It includes the format and function of each feature of the language. The *VAX Architecture Reference Manual* describes the instruction set in greater detail.

Intended Audience

This manual is intended for all programmers writing VAX MACRO programs. You should be familiar with assembly language programming, the VAX instruction set, and the VMS operating system before reading this manual.

Document Structure

This manual is divided into two parts, each of which is subdivided into several chapters.

Part I describes the VAX MACRO language.

- Chapter 1 introduces the features of the VAX MACRO language.
- Chapter 2 describes the format used in VAX MACRO source statements.
- Chapter 3 describes the following components of VAX MACRO source statements:
  - Character set
  - Numbers
  - Symbols
  - Local labels
  - Terms and expressions
  - Unary and binary operators
  - Direct assignment statements
  - Current location counter
- Chapter 4 describes the arguments and string operators used with macros.
- Chapter 5 summarizes and gives examples of using the VAX MACRO addressing modes.
- Chapter 6 describes the VAX MACRO general assembler directives and the directives used in defining and expanding macros.

Part II describes the VAX data types, the instruction and addressing mode formats, and the instruction set.

- Chapter 7 summarizes the terminology and conventions used in the descriptions in Part II.

- Chapter 8 describes the basic VAX architecture, including the following:
  - Address space
  - Data types
  - Processor state
  - Processor status longword
  - Permanent exception enables
  - Instruction and addressing mode formats
- Chapter 9 describes the native-mode instruction set. The instructions are divided into groups according to their function and are listed alphabetically within each group.

VAX MACRO and Instruction Set Reference Manual also contains the following five appendixes:

- Appendix A lists the ASCII character set used in VAX MACRO programs.
- Appendix B gives rules for hexadecimal/decimal conversion.
- Appendix C summarizes the general assembler and macro directives (in alphabetical order), special characters, unary operators, binary operators, and addressing modes.
- Appendix D lists the permanent symbols (instruction set) defined for use with VAX MACRO.
- Appendix E describes the exceptions (traps and faults) that may occur during instruction execution.

Associated Documents

The following documents are relevant to VAX MACRO programming:

- *VAX Architecture Reference Manual*
- *VMS DCL Dictionary*
- The descriptions of the VMS Linker and Symbolic Debugger in:
  - *VMS Linker Utility Manual*
  - *VMS Debugger Manual*
- *Introduction to VMS System Routines*
- *VMS Run-Time Library Routines Volume*

Conventions

Convention	Meaning
RET	In examples, a key name (usually abbreviated) shown within a box indicates that you press a key on the keyboard; in text, a key name is not enclosed in a box. In this example, the key is the RETURN key. (Note that the RETURN key is not usually shown in syntax statements or in all examples; however, assume that you must press the RETURN key after entering a command or responding to a prompt.)
CTRL/C	A key combination, shown in uppercase with a slash separating two key names, indicates that you hold down the first key while you press the second key. For example, the key combination CTRL/C indicates that you hold down the key labeled CTRL while you press the key labeled C. In examples, a key combination is enclosed in a box.
UPPERCASE WORDS AND LETTERS	Uppercase words and letters used in format examples indicate that you should type the word or letter exactly as shown.
lowercase words and letters	Lowercase words and letters used in format examples indicate that you are to substitute a word or value of your choice.
$ TYPE MYFILE.DAT . . .	In examples, a vertical series of periods, or ellipsis, means either that not all the data that the system would display in response to a command is shown or that not all the data a user would enter is shown.
input-file, . . .	In examples, a horizontal ellipsis indicates that additional parameters, values, or other information can be entered, that preceding items can be repeated one or more times, or that optional arguments in a statement have been omitted.
[logical-name]	Brackets indicate that the enclosed item is optional. (Brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)
quotation marks apostrophes	The term quotation marks is used to refer to double quotation marks ("). The term apostrophe (') is used to refer to a single quotation mark.

New and Changed Features

The following technical changes have been made since Version 4.0:

- A list of instructions that manipulate self-relative queues was added to the end of Section 9.9.2.
- The use of POPL was added to the description of the PUSHL instruction in Section 9.3.

VAX MACRO Language

Part I provides an overview of the features of the VAX MACRO language. It includes an introduction to the structure and components of VAX MACRO source statements. Part I also contains a detailed discussion of the VAX MACRO addressing modes, general assembler directives, and macro directives.

1 Introduction

VAX MACRO is an assembly language for programming VAX computers using the VMS operating system. Source programs written in VAX MACRO are translated into object (or binary) code by the VAX MACRO assembler, which produces an object module and, optionally, a listing file. The features of the language are introduced in this chapter.

VAX MACRO source programs consist of a sequence of source statements. These source statements may be any of the following:

- VAX native-mode instructions
- Direct assignment statements
- Assembler directives

Instructions manipulate data. They perform such functions as addition, data conversion, and transfer of control. Instructions are usually followed in the source statement by operands, which can be any kind of data needed for the operation of the instruction. The VAX instruction set is summarized in Appendix D of this volume and is described in detail in Chapter 9.

Direct assignment statements equate symbols to values.

Assembler directives guide the assembly process and provide tools for using the instructions. There are two classes of assembler directives: general assembler directives and macro directives.

General assembler directives can be used to perform the following operations:

- Store data or reserve memory for data storage
- Control the alignment of parts of the program in memory
- Specify the methods of accessing the sections of memory in which the program will be stored
- Specify the entry point of the program or a part of the program
- Specify the way in which symbols will be referenced
- Specify that a part of the program is to be assembled only under certain conditions
- Control the format and content of the listing file
- Display informational messages
- Control the assembler options that are used to interpret the source program
- Define new opcodes

Macro directives are used to define macros and repeat blocks. They allow you to perform the following operations:

- Repeat identical or similar sequences of source statements throughout a program without rewriting those sequences

- Use string operators to manipulate and test the contents of source statements

Use of macros and repeat blocks helps minimize programmer errors and speeds the debugging process.

2 MACRO Source Statement Format

A source program consists of a sequence of source statements that the assembler interprets and processes, one at a time, generating object code or performing a specific assembly-time process. A source statement can occupy one source line or can extend onto several source lines. Each source line can be up to 132 characters long; however, to ensure that the source line fits (with its binary expansion) on one line in the listing file, no line should exceed 80 characters.

MACRO statements can consist of up to four fields, as follows:

- Label field—symbolically defines a location in a program.
- Operator field—specifies the action to be performed by the statement; can be an instruction, an assembler directive, or a macro call.
- Operand field—contains the instruction operand(s) or the assembler directive argument(s) or the macro argument(s).
- Comment field—contains a comment that explains the meaning of the statement; does not affect program execution.

The label field and the comment field are optional. The label field ends with a colon (:) and the comment field begins with a semicolon (;). The operand field must conform to the format of the instruction, directive, or macro specified in the operator field.

Although statement fields can be separated by either a space or a tab (see Table 3–2), formatting statements with the tab character is recommended for consistency and clarity and is a DIGITAL convention.

Field	Begins in Column	Tab Characters to Reach Column
Label	1	0
Operator	9	1
Operand	17	2
Comment	41	5

For example:

```
        .TITLE  ROUT1
        .ENTRY  START,^M<>              ; Beginning of routine
        CLRL    R0                      ; Clear register
LABT:   SUBL3   #10,4(AP),R2            ; Subtract 10
LAB2:   BRB     CONT                    ; Branch to another routine
```

Continue a single statement on several lines by using a hyphen (-) as the last nonblank character before the comment field, or at the end of line (when there is no comment). For example:

```
LAB1:   MOVAL   W^BOO$AL_VECTOR,-       ; Save boot driver
                RPB$L_IOVEC(R7)
```

VAX MACRO treats the preceding statement as equivalent to the following statement:

```
LAB1:   MOVAL   W^BOO$AL_VECTOR,RPB$L_IOVEC(R7)  ; Save boot driver
```

A statement can be continued at any point. Do not continue permanent and user-defined symbol names on two lines. If a symbol name is continued and the first character on the second line is a tab or a blank, the symbol name is terminated at that character. Section 3.3 describes symbols in detail.

Note that when a statement occurs in a macro definition (see Chapter 4 and Chapter 6), the statement cannot contain more than 1000 characters.

Blank lines are legal, but they have no significance in the source program except that they terminate a continued line.

The following sections describe each of the statement fields in detail.

2.1 Label Field

A label is a user-defined symbol that identifies a location in the program. The symbol is assigned a value equal to the location counter where the label occurs. The user-defined symbol name can be up to 31 characters long and can contain any alphanumeric character and the underscore (_), dollar sign ($), and period (.) characters. Section 3.3.2 describes the rules for forming user-defined symbol names in more detail.

If a statement contains a label, the label must be in the first field on the line.

A label is terminated by a colon (:) or a double colon (::). A single colon indicates that the label is defined only for the current module (an internal symbol). A double colon indicates that the label is globally defined; that is, the label can be referenced by other object modules.

Once a label is defined, it cannot be redefined during the source program. If a label is defined more than once, VAX MACRO displays an error message when the label is defined and again when it is referenced.

If a label extends past column 7, place it on a line by itself so that the following operator field can start in column 9 of the next line.

The following example illustrates some of the ways you can define labels:

```
EXP:    .BLKL   50      ; Table stores expected values
DATA::  .BLKW   25      ; Data table accessed by store
                        ;   routine in another module
EVAL:   CLRL    R0      ; Routine evaluates expressions
ERROR_IN_ARG:           ; The arg-list contains an error
        INCL    R0      ;   increment error count
TEST::  MOVO    EXP,R1  ; This tests routine
                        ;   referenced externally
TEST1:  BRW     EXIT    ; Go to exit routine
```

The label field is also used for the symbol in a direct assignment statement (see Section 3.8).

2.2 Operator Field

The operator field specifies the action to be performed by the statement. This field can contain an instruction, an assembler directive, or a macro call.

When the operator is an instruction, VAX MACRO generates the binary code for that instruction in the object module. The binary codes are listed in Appendix D; the instruction set is described in Chapter 9. When the operator is a directive, VAX MACRO performs certain control actions or processing operations during source program assembly. The assembler directives are described in Chapter 6. When the operator is a macro call, VAX MACRO expands the macro. Macro calls are described in Chapter 4 and in Chapter 6 (.MACRO directive).

Use either a space or a tab character to terminate the operator field; however, the tab is the recommended termination character.

2.3 Operand Field

The operand field can contain operands for instructions or arguments for either assembler directives or macro calls.

Operands for instructions identify the memory locations or the registers that are used by the machine operation. These operands specify the addressing mode for the instruction, as described in Chapter 5. The operand field for a specific instruction must contain the number of operands required by that instruction. See Chapter 9 for descriptions of the instructions and their operands.

Arguments for a directive must meet the format requirements of that directive. Chapter 6 describes the directives and the format of their arguments.

Operands for a macro must meet the requirements specified in the macro definition. See the description of the .MACRO directive in Chapter 6.

If two or more operands are specified, they must be separated by commas. VAX MACRO also allows a space or tab to be used as a separator for arguments to any directive that does not accept expressions (see Section 3.5 for a discussion of expressions). However, a comma is required to separate operands for instructions and for directives that accept expressions as arguments.

The semicolon that starts the comment field terminates the operand field. If a line does not have a comment field, the operand field is terminated by the end of the line.

2.4 Comment Field

The comment field contains text that explains the function of the statement. Every line of code should have a comment. Comments do not affect assembly processing or program execution. You can cause user-written messages to be displayed during assembly by the .ERROR, .PRINT, and .WARN directives (see descriptions in Chapter 6).

The comment field must be preceded by a semicolon; it is terminated by the end of the line. The comment field can contain any printable ASCII character (see Appendix A).

To continue a lengthy comment to the next line, write the comment on the next line and precede it with another semicolon. If a comment does not fit on one line, it can be continued on the next, but the continuation must be preceded by another semicolon. A comment can appear on a line by itself.

Write the text of a comment to convey the meaning rather than the action of the statement. The instruction MOVAL BUF_PTR_1,R7, for example, should have a comment such as "Get pointer to first buffer," not "Move address of BUF_PTR_1 to R7."

For example:

```
MOVAL   STRING_DES_1,R0 ; Get address of string
                        ;   descriptor
MOVZWL  (R0),R1         ; Get length of string
MOVL    4(R0),R0        ; Get address of string
```

3 Components of MACRO Source Statements

This chapter describes the following components of VAX MACRO source statements:

- Character set
- Numbers
- Symbols
- Local labels
- Terms and expressions
- Unary and binary operators
- Direct assignment statements
- Current location counter

3.1 Character Set

The following characters can be used in VAX MACRO source statements:

- The letters of the alphabet, A through Z, uppercase and lowercase. Note that the assembler considers lowercase letters equivalent to uppercase letters except when they appear in ASCII strings.
- The digits 0 through 9.
- The special characters listed in Table 3–1.

Table 3–1 Special Characters Used in VAX MACRO Statements

Character	Character Name	Function
_	Underline	Character in symbol names
$	Dollar sign	Character in symbol names
.	Period	Character in symbol names, current location counter, and decimal point
:	Colon	Label terminator
=	Equal sign	Direct assignment operator and macro keyword argument terminator
	Tab	Field terminator
	Space	Field terminator
#	Number sign	Immediate addressing mode indicator
@	At sign	Deferred addressing mode indicator and arithmetic shift operator

Table 3–1 (Cont.) Special Characters Used in VAX MACRO Statements

Character	Character Name	Function
,	Comma	Field, operand, and item separator
;	Semicolon	Comment field indicator
+	Plus sign	Autoincrement addressing mode indicator, unary plus operator, and arithmetic addition operator
–	Minus sign or hyphen	Autodecrement addressing mode indicator, unary minus operator, arithmetic subtraction operator, and line continuation indicator
*	Asterisk	Arithmetic multiplication operator
/	Slash	Arithmetic division operator
&	Ampersand	Logical AND operator
!	Exclamation	Logical inclusive OR operator point
\	Backslash	Logical exclusive OR and numeric conversion indicator in macro arguments
^	Circumflex	Unary operators and macro argument delimiter
[ ]	Square brackets	Index addressing mode and repeat count indicators
( )	Parentheses	Register deferred addressing mode indicators
<>	Angle brackets	Argument or expression grouping delimiters
?	Question mark	Created local label indicator in macro arguments
'	Apostrophe	Macro argument concatenation indicator
%	Percent sign	Macro string operators

Table 3–2 defines the separating characters used in VAX MACRO.

Table 3–2 Separating Characters in VAX MACRO Statements

Character	Character Name	Usage
(space) *(tab)*	Space or tab	Separator between statement fields. Spaces within expressions are ignored.
,	Comma	Separator between symbolic arguments within the operand field. Multiple expressions in the operand field must be separated by commas.

3.2 Numbers

Numbers can be integers, floating-point numbers, or packed decimal strings.

3.2.1 Integers

Integers can be used in any expression including expressions in operands and in direct assignment statements (Section 3.5 describes expressions).

Format

snn

s
An optional sign: plus sign (+) for positive numbers (the default) or minus sign (−) for negative numbers.

nn
A string of numeric characters that is legal for the current radix.

VAX MACRO interprets all integers in the source program as decimal unless the number is preceded by a radix control operator (see Section 3.6.1).

Integers must be in the range of −2,147,483,648 through +2,147,483,647 for signed data or in the range of 0 through 4,294,967,295 for unsigned data.

Negative numbers must be preceded by a minus sign; VAX MACRO translates such numbers into two's complement form. In positive numbers, the plus sign is optional.

3.2.2 Floating-Point Numbers

A floating-point number can be used in the .F_FLOATING (.FLOAT), .D_FLOATING (.DOUBLE), .G_FLOATING, and .H_FLOATING directives (described in Chapter 6) or as an operand in a floating-point instruction. A floating-point number cannot be used in an expression or with a unary or binary operator except the unary plus, unary minus, and unary floating-point operator, ^F (F_FLOATING). Sections 3.6 and 3.7 describe unary and binary operators.

A floating-point number can be specified with or without an exponent.

Formats

Floating-point number without exponent:

snn
snn.nn
snn.

Floating-point number with exponent:

snnEsnn
snn.nnEsnn
snn.Esnn

s
An optional sign.

nn
A string of decimal digits in the range of 0 through 9.

The decimal point can appear anywhere to the right of the first digit. Note that a floating-point number cannot start with a decimal point because VAX MACRO will treat the number as a user-defined symbol (see Section 3.3.2).

Floating-point numbers can be single-precision (32-bit), double-precision (64-bit), or extended-precision (128-bit) quantities. The degree of precision is 7 digits for single-precision numbers, 16 digits for double-precision numbers, and 33 digits for extended-precision numbers.

The magnitude of a nonzero floating-point number cannot be smaller than approximately 0.29E-38 or greater than approximately 1.7E38.

Single-precision floating-point numbers can be rounded (by default) or truncated. The .ENABLE and .DISABLE directives (described in Chapter 6) control whether single-precision floating-point numbers are rounded or truncated. Double-precision and extended-precision floating-point numbers are always rounded.

Sections 8.2.6, 8.2.7, 8.2.8, and 8.2.9 describe the internal format of floating-point numbers.

3.2.3 Packed Decimal Strings

A packed decimal string can be used only in the .PACKED directive (described in Chapter 6).

Format

snn

s
An optional sign.

nn
A string containing up to 31 decimal digits in the range of 0 through 9.

A packed decimal string cannot have a decimal point or an exponent.

Section 8.2.14 describes the internal format of packed decimal strings.

3.3 Symbols

Three types of symbols can be used in VAX MACRO source programs: permanent symbols, user-defined symbols, and macro names.

3.3.1 Permanent Symbols

Permanent symbols consist of instruction mnemonics (see Appendix D), VAX MACRO directives (see Chapter 6), and register names. You need not define instruction mnemonics and directives before you use them in the operator field of a VAX MACRO source statement. Also, you need not define register names before using them in the addressing modes (see Chapter 5).

Register names cannot be redefined; that is, a symbol that you define cannot be one of the register names contained in the following list. You can express the 16 general registers of the VAX processor in a source program only as follows:

Register Name	Processor Register
R0	General register 0
R1	General register 1
R2	General register 2
.	.
.	.
.	.
R11	General register 11
R12 or AP	General register 12 or argument pointer. If you use R12 as an argument pointer, the name AP is recommended; if you use R12 as a general register, the name R12 is recommended.
FP	Frame pointer
SP	Stack pointer
PC	Program counter

Note that the symbols IV and DV are also permanent symbols and cannot be redefined. These symbols are used in the register mask to set the integer overflow trap (IV) and the decimal string overflow trap (DV). See Section 3.6.2.2 for an explanation of their uses.

3.3.2 User-Defined Symbols and Macro Names

You can use symbols that you define as labels or you can equate them to a specific value by a direct assignment statement (see Section 3.8). These symbols can also be used in any expression (see Section 3.5).

The following rules govern the creation of user-defined symbols:

- User-defined symbols can be composed of alphanumeric characters, underlines (_), dollar signs ($), and periods (.). Any other character terminates the symbol.
- The first character of a symbol must not be a number.
- The symbol must be no more than 31 characters long and must be unique.

In addition, by DIGITAL convention:

- The dollar sign ($) is reserved for names defined by DIGITAL. This convention ensures that a user-defined name (which does not have a dollar sign) will not conflict with a DIGITAL-defined name (which does have a dollar sign).

- Do not use the period (.) in any global symbol name (see Section 3.3.3) because languages, such as FORTRAN, do not allow periods in symbol names.

Macro names follow the same rules and conventions as user-defined symbols. (See the description of the .MACRO directive in Chapter 6 for more information on macro names.) User-defined symbols and macro names do not conflict; that is, the same name can be used for a user-defined symbol and a macro. To avoid confusion, give the symbols and macros that you define different names.

3.3.3 Determining Symbol Values

The value of a symbol depends on its use in the program. VAX MACRO uses a different method to determine the values of symbols in the operator field than it uses to determine the values of symbols in the operand field.

A symbol in the operator field can be either a permanent symbol or a macro name. VAX MACRO searches for a symbol definition in the following order:

1. Previously defined macro names
2. User-defined opcode (see the .OPDEF description in Chapter 6)
3. Permanent symbols (instructions and directives)
4. Macro libraries

This search order allows permanent symbols to be redefined as macro names. If a symbol in the operator field is not defined as a macro or a permanent symbol, the assembler displays an error message.

A symbol in the operand field must be either a user-defined symbol or a register name.

User-defined symbols can be either local (internal) symbols or global (external) symbols. Whether symbols are local or global depends on their use in the source program.

A local symbol can be referenced only in the module in which it is defined. If local symbols with the same names are defined in different modules, the symbols are completely independent. The definition of a global symbol, however, can be referenced from any module in the program.

VAX MACRO treats all symbols that you define as local unless you explicitly declared them to be global by doing any one of the following:

- Use the double colon (::) in defining a label (see Section 2.1).
- Use the double equal sign (==) in a direct assignment statement (see Section 3.8).
- Use the .GLOBAL, .ENTRY, or .WEAK directive (see Chapter 6).

When your code references a symbol within the module in which it is defined, VAX MACRO considers the reference internal. When your code references a symbol within a module in which it is not defined, VAX MACRO considers the reference external (that is, the symbol is defined externally in another module). You can use the .DISABLE directive to make references to symbols not defined in the current module illegal. In this case, you must use the

.EXTERNAL directive to specify that the reference is an external reference. See Chapter 6 for descriptions of the .DISABLE and .EXTERNAL directives.

3.4 Local Labels

Use local labels to identify addresses within a block of source code.

Format

nn$

nn
A decimal integer in the range of 1 through 65535.

Use local labels in the same way as you use the symbol labels that you define, with the following differences:

- Local labels cannot be referenced outside the block of source code in which they appear.
- Local labels can be reused in another block of source code.
- Local labels do not appear in the symbol tables and thus cannot be accessed by the VAX Symbolic Debugger.
- Local labels cannot be used in the .END directive (see Chapter 6).

By convention, local labels are positioned like statement labels: left-justified in the source text. Although local labels can appear in the program in any order, by convention, the local labels in any block of source code should be in numeric order.

Local labels are useful as branch addresses when you use the address only within the block. You can use local labels to distinguish between addresses that are referenced only in a small block of code and addresses that are referenced elsewhere in the module. A disadvantage of local labels is that their numeric names cannot provide any indication of their purpose. Consequently, you should not use local labels to label sequences of statements that are logically unrelated; user-defined symbols should be used instead.

DIGITAL recommends that users create local labels only in the range of 1$ to 29999$ because the assembler automatically creates local labels in the range of 30000$ to 65535$ for use in macros (see Section 4.7).

The local label block in which a local label is valid is delimited by the following statements:

- A user-defined label
- A .PSECT directive (see Chapter 6)
- The .ENABLE and .DISABLE directives (see Chapter 6), which can extend a local label block beyond user-defined labels and .PSECT directives

A local label block is usually delimited by two user-defined labels. However, the .ENABLE LOCAL_BLOCK directive starts a local block that is terminated only by one of the following:

- A second .ENABLE LOCAL_BLOCK directive

- A .DISABLE LOCAL_BLOCK directive followed by a user-defined label or a .PSECT directive

Although local label blocks can extend from one program section to another, DIGITAL recommends that local labels in one program section not be referenced from another program section. User-defined symbols should be used instead.

Local labels can be preserved for future reference with the context of the program section in which they are defined; see the descriptions of the .SAVE_PSECT [LOCAL_BLOCK] directive and the .RESTORE_PSECT directive in Chapter 6.

An example showing the use of local labels follows:

```
RPSUB:  MOVL    AMOUNT,R0         ; Start local label block
10$:    SUBL2   DELTA,R0          ; Define local label 10$
        BGTR    10$               ; Conditional branch to
                                  ;   local label
        ADDL2   DELTA,R0          ; Executed when R0 not > 0
COMP:   MOVL    MAX,R1            ; End previous local label
        CLRL    R2                ;   block and start new one
10$:    CMPL    R0,R1             ; Define new local label 10$
        BGTR    20$               ; Conditional branch to
                                  ;   local label
        SUBL    INCR,R0           ; Executed when R0 not > R1
        INCL    R2                ; . . .
        BRB     10$               ; Unconditional branch to
                                  ;   local label
20$:    MOVL    R2,COUNT          ; Define local label
        BRW     TEST              ; Unconditional branch to
                                  ;   user-defined label

        .ENABLE LOCAL_BLOCK       ; Start local label block that
ENTR1:  POPR    #^M<R0,R1,R2>     ;   will not be terminated
        ADDL3   R0,R1,R3          ;   by a user-defined label
        BRB     10$               ; Branch to local label that appears
                                  ;   after a user-defined label
ENTR2:  SUBL2    R2,R3            ; Does not start a new
                                  ;   local label block
10$:    SUBL2   R2,R3             ; Define local label
        BGTR    20$               ; Conditional branch to
                                  ;   local label
        INCL    R0                ; Executed when R2 not > R3
        BRB     NEXT              ; Unconditional branch to
                                  ;   user-defined label
20$:    DECL    R0                ; Define local label
        .DISABLE LOCAL_BLOCK      ; Directive followed by user-
NEXT:   CLRL    R4                ;   defined label terminates
                                  ;   local label block
```

3.5 Terms and Expressions

A term can be any of the following:

- A number
- A symbol
- The current location counter (see Section 3.9)
- A textual operator followed by text (see Section 3.6.2)
- Any of the previously noted items preceded by a unary operator (see Section 3.6)

VAX MACRO evaluates terms as longword (4-byte) values. If you use an undefined symbol as a term, the linker determines the value of the term. The current location counter (.) has the value of the location counter at the start of the current operand.

Expressions are combinations of terms joined by binary operators (see Section 3.7) and evaluated as longword (4-byte) values. VAX MACRO evaluates expressions from left to right with no operator precedence rules. However, angle brackets (<>) can be used to change the order of evaluation. Any part of an expression that is enclosed in angle brackets is first evaluated to a single value, which is then used in evaluating the complete expression. For example, the expressions A*B+C and A* <B+C> are different. In the first case, A and B are multiplied and then C added to the product. In the second case, B and C are added and the sum is multiplied by A. Angle brackets can also be used to apply a unary operator to an entire expression, such as – <A+B> .

If an arithmetic expression is continued on another line, the listing file will not show the continued line. For example:

```
.WORD <DATA1'$^XFF@8+-
     89>
```

You must use /LIST/SHOW=EXPANSION to show the continuation line.

VAX MACRO considers unary operators part of a term and thus, performs the action indicated by a unary operator before it performs the action indicated by any binary operator.

Expressions fall into three categories: relocatable, absolute, and external (global), as follows:

- An expression is relocatable if its value is fixed relative to the start of the program section in which it appears. The current location counter is relocatable in a relocatable program section.
- An expression is absolute if its value is an assembly-time constant. An expression whose terms are all numbers is absolute. An expression that consists of a relocatable term minus another relocatable term from the same program section is absolute, since such an expression reduces to an assembly-time constant.
- An expression is external if it contains one or more symbols that are not defined in the current module.

Any type of expression can be used in most MACRO statements, but restrictions are placed on expressions used in the following:

- .ALIGN alignment directives
- .BLK*x* storage allocation directives
- .IF and .IIF conditional assembly block directives
- .REPEAT repeat block directives
- .OPDEF opcode definition directives
- .ENTRY entry point directives
- .BYTE, .LONG, .WORD, .SIGNED_BYTE, and .SIGNED_WORD directive repetition factors
- Direct assignment statements (see Section 3.8)

See Chapter 6 for descriptions of the directives listed in the preceding list.

Expressions used in these directives and in direct assignment statements can contain only symbols that have been previously defined in the current module. They cannot contain either external symbols or symbols defined later in the current module. In addition, the expressions in these directives must be absolute. Expressions in direct assignment statements can be relocatable.

An example showing the use of expressions follows.

```
A = 2*100                  ; 2*100 is an absolute expression
        .BLKB   A+50       ; A+50 is an absolute expression and
                           ;   contains no undefined symbols
LAB:    .BLKW   A          ; LAB is relocatable
HALF = LAB+<A/2>           ; LAB+<A/2> is a relocatable
                           ;   expression and contains no
                           ;   undefined symbols
LAB2:   .BLKB   LAB2-LAB   ; LAB2-LAB is an absolute expression
                           ;   and contains no undefined symbols
                           ;   but contains the symbol LAB3
                           ;   that is defined later in this module
LAB3:   .WORD   TST+LAB+2  ; TST+LAB+2 is an external expression
                           ;   because TST is an external symbol
```

3.6 Unary Operators

A unary operator modifies a term or an expression and indicates an action to be performed on that term or expression. Expressions modified by unary operators must be enclosed in angle brackets. You can use unary operators to indicate whether a term or expression is positive or negative. If unary plus or minus is not specified, the default value is assumed to be plus. In addition, unary operators perform radix conversion, textual conversion (including ASCII conversion), and numeric control operations, as described in the following sections. Table 3–3 summarizes the unary operators.

Table 3–3 Unary Operators

Unary Operator	Operator Name	Example	Operation
+	Plus sign	+A	Results in the positive value of A
–	Minus sign	–A	Results in the negative (two's complement) value of A
^B	Binary	^B11000111	Specifies that 11000111 is a binary number
^D	Decimal	^D127	Specifies that 127 is a decimal number
^O	Octal	^O34	Specifies that 34 is an octal number
^X	Hexadecimal	^XFCF9	Specifies that FCF9 is a hexadecimal number
^A	ASCII	^A/ABC/	Produces an ASCII string; the characters between the matching delimiters are converted to ASCII representation
^M	Register mask	#^M <R3,R4,R5>	Specifies the registers R3, R4, and R5 in the register mask
^F	Floating-point	^F3.0	Specifies that 3.0 is a floating-point number
^C	Complement	^C24	Produces the one's complement value of 24 (decimal)

More than one unary operator can be applied to a single term or to an expression enclosed in angle brackets. For example:

```
-+-A
```

This construct is equivalent to:

```
-<+<-A>>
```

3.6.1 Radix Control Operators

VAX MACRO accepts terms or expressions in four different radixes: binary, decimal, octal, and hexadecimal. The default radix is decimal. Expressions modified by radix control operators must be enclosed in angle brackets.

Formats

^Bnn
^Dnn
^Onn
^Xnn

nn
A string of characters that is legal in the specified radix. The following are the legal characters for each radix:

Format	Radix Name	Legal Characters
^Bnn	Binary	0 and 1
^Dnn	Decimal	0 through 9
^Onn	Octal	0 through 7
^Xnn	Hexadecimal	0 through 9 and A through F

Radix control operators can be included in the source program anywhere a numeric value is legal. A radix control operator affects only the term or expression immediately following it, causing that term or expression to be evaluated in the specified radix.

For example:

```
.WORD   ^B00001101              ; Binary radix
.WORD   ^D123                   ; Decimal radix (default)
.WORD   ^047                    ; Octal radix
.WORD   <A+^013>                ; 13 is in octal radix
.LONG   ^X<F1C3+FFFFF-20>       ; All numbers in expression
                                ;   are in hexadecimal radix
```

The circumflex cannot be separated from the B, D, O, or X that follows it, but the entire radix control operator can be separated by spaces and tabs from the term or expression that is to be evaluated in that radix.

The default decimal operator is needed only within an expression that has another radix control operator. In the following example, "16" is interpreted as a decimal number because it is preceded by the decimal operator ^D even though the "16" is in an expression prefixed by the octal radix control operator.

```
.LONG    ^O<10000 + 100 + ^D16>
```

3.6.2 Textual Operators

The textual operators are the ASCII operator (^A) and the register mask operator (^M).

3.6.2.1 ASCII Operator

The ASCII operator converts a string of printable characters to their 8-bit ASCII values and stores them one character to a byte. The string of characters must be enclosed in a pair of matching delimiters.

The delimiters can be any printable character except the space, tab, or semicolon (;). Use nonalphanumeric characters to avoid confusion.

Format

^Astring

string

A delimited ASCII string from 1 through 16 characters long.

The delimited ASCII string must not be larger than the data type of the operand. For example, if the ^A operator occurs in an operand in a MOVW instruction (the data type is a word), the delimited string cannot be more than two characters.

For example:

```
.QUAD   ^A%1234/678%    ; Generates 8 bytes of ASCII data
MOVL    #^A/ABCD/,R0    ; Moves characters ABCD
                        ;   into R0 right justified with
                        ;   "A" in low-order byte and "D"
                        ;   in high-order byte
CMPW    #^A/XY/,R0      ; Compares X and Y as ASCII
                        ;   characters with contents of low
                        ;   order 2 bytes of R0
MOVL    #^A/AB/,R0      ; Moves ASCII characters AB into
                        ;   R0; "A" in low-order byte; "B" in
                        ;   next; and zero the 2 high-order bytes
```

3.6.2.2 Register Mask Operator

The register mask operator converts a register name or a list of register names enclosed in angle brackets into a 1- or 2-byte register mask. The register mask is used by the PUSHR and POPR instructions and the .ENTRY and .MASK directives (see Chapter 6).

Formats

^Mreg-name
^M <reg-name-list>

reg-name

One of the register names or the DV or IV arithmetic trap-enable specifiers.

reg-name-list

A list of register names and the DV and IV arithmetic trap-enable specifiers, separated by commas.

The register mask operator sets a bit in the register mask for every register name or arithmetic trap enable specified in the list. The bits corresponding to each register name and arithmetic trap-enable specifier are listed below.

Register Name	Arithmetic Trap	Bits
R0 through R11		0 through 11
R12 or AP		12
FP		13
SP	IV	14
	DV	15

When the POPR or PUSHR instruction uses the register mask operator, R0 through R11, R12 or AP, FP, and SP can be specified. You cannot specify the PC register name and the IV and DV arithmetic trap-enable specifiers.

When the .ENTRY or .MASK directives use the register mask operator, you can specify R2 through R11 and the IV and DV arithmetic trap-enable specifiers. However, you cannot specify R0, R1, FP, SP, and PC. IV sets the integer overflow trap, and DV sets the decimal string overflow trap.

The arithmetic trap-enable specifiers are described in Chapter 8.

For example:

```
 ENTRY RT1,^M<R3,R4,R5,R6,IV>   ; Save registers R3, R4,
                                ;   R5, and R6 and set the
                                ;   integer overflow trap

PUSHR   #^M<R0,R1,R2,R3>        ; Save registers R0, R1,
                                ;   R2, and R3

POPR    #^M<R0,R1,R2,R3>        ; Restore registers R0, R1,
                                ;   R2, and R3
```

3.6.3 Numeric Control Operators

The numeric control operators are the floating-point operator (^F) and the complement operator (^C). The use of the numeric control operators is explained in the following two sections.

3.6.3.1 Floating-Point Operator

The floating-point operator accepts a floating-point number and converts it to its internal representation (a 4-byte value). This value can be used in any expression. VAX MACRO does not perform floating-point expression evaluation.

Format

^Fliteral

literal
A floating-point number (see Section 3.2.2).

The floating-point operator is useful because it allows a floating-point number in an instruction that accepts integers.

For example:

```
MOVL     #^F3.7,R0      ; NOTE: the recommended instruction
                        ;   to move this floating-point
MOVF     #3.7,R0        ;   number is the MOVF instruction
```

3.6.3.2 Complement Operator

The complement operator produces the one's complement of the specified value.

Format

^Cterm

term

Any term or expression. If an expression is specified, it must be enclosed in angle brackets.

VAX MACRO evaluates the term or expression as a 4-byte value before complementing it.

For example:

```
.LONG       ^C^XFF       ; Produces FFFFFF00 (hex)
.LONG       ^C25         ; Produces complement of
                         ;   25 (dec) which is
                         ;   FFFFFFE6 (hex)
```

3.7 Binary Operators

In contrast to unary operators, binary operators specify actions to be performed on two terms or expressions. Expressions must be enclosed in angle brackets. Table 3–4 summarizes the binary operators.

Table 3–4 Binary Operators

Binary Operator	Operator Name	Example	Operation
+	Plus sign	A+B	Addition
–	Minus sign	A–B	Subtraction
*	Asterisk	A*B	Multiplication
/	Slash	A/B	Division
@	At sign	A@B	Arithmetic shift
&	Ampersand	A&B	Logical AND
!	Exclamation point	A!B	Logical inclusive OR
\	Backslash	A\B	Logical exclusive OR

All binary operators have equal priority. Terms or expressions can be grouped for evaluation by enclosing them in angle brackets. The enclosed terms and expressions are evaluated first, and remaining operations are performed from left to right. For example:

```
.LONG       1+2*3        ; Equals 9
.LONG       1+<2*3>      ; Equals 7
```

Note that a 4-byte result is returned from all binary operations. If you use a 1-byte or 2-byte operand, the result is the low-order byte(s) of the 4-byte result. VAX MACRO displays an error message if the truncation causes a loss of significance.

The following sections describe the arithmetic shift, logical AND, logical inclusive OR, and logical exclusive OR operators.

3.7.1 Arithmetic Shift Operator

You use the arithmetic shift operator (@) to perform left and right arithmetic shifts of arithmetic quantities. The first argument is shifted left or right by the number of bit positions that you specify in the second argument. If the second argument is positive, the first argument is shifted left; if the second argument is negative, the first argument is shifted right. When the first argument is shifted left, the low-order bits are set to 0. When the first argument is shifted right, the high-order bits are set to the value of the original high-order bit (the sign bit).

For example:

```
        .LONG   ^B101@4             ; Yields 1010000 (binary)
        .LONG   1@2                 ; Yields 100 (binary)
A = 4
        .LONG   1@A                 ; Yields 10000 (binary)
        .LONG   ^X1234@-A           ; Yields 123(hex)
        MOVL    #<^B1100000@-5>,R0  ; Yields 11 (binary)
```

3.7.2 Logical AND Operator

The logical AND operator (&) takes the logical AND of two operands.

For example:

```
A = ^B1010
B = ^B1100
        .LONG   A&B             ; Yields 1000 (binary)
```

3.7.3 Logical Inclusive OR Operator

The logical inclusive OR operator (!) takes the logical inclusive OR of two operands.

For example:

```
A = ^B1010
B = ^B1100
        .LONG   A!B             ; Yields 1110 (binary)
```

3.7.4 Logical Exclusive OR Operator

The logical exclusive OR operator (\) takes the logical exclusive OR of two arguments.

For example:

```
A = ^B1010
B = ^B1100
        .LONG   A\B             ; Yields 0110 (binary)
```

3.8 Direct Assignment Statements

A direct assignment statement equates a symbol to a specific value. Unlike a symbol that you use as a label, you can redefine a symbol defined with a direct assignment statement as many times as you want.

Formats

symbol=expression
symbol==expression

symbol
A user-defined symbol.

expression
An expression that does not contain any undefined symbols (see Section 3.5).

The format with a single equal sign (=) defines a local symbol and the format with a double equal sign (==) defines a global symbol. See Section 3.3.3 for more information about local and global symbols.

The following three syntactic rules apply to direct assignment statements:

- An equal sign (=) or double equal sign (==) must separate the symbol from the expression which defines its value. Spaces preceding or following the direct assignment operators have no significance in the resulting value.
- Only one symbol can be defined in a single direct assignment statement.
- A direct assignment statement can be followed only by a comment field.

By DIGITAL convention, the symbol in a direct assignment statement is placed in the label field.

For example:

```
A == 1                  ; The symbol 'A' is globally
                        ;   equated to the value 1

B = A@5                 ; The symbol 'B' is equated
                        ;   to 1@5 or 20(hex)

C = 127*10              ; The symbol 'C' is equated
                        ;   to 1270(dec)

D = ^X100/^X10          ; The symbol 'D' is equated
                        ;   to 10(hex)
```

3.9 Current Location Counter

The symbol for the current location counter, the period (.), always has the value of the address of the current byte. VAX MACRO sets the current location counter to 0 at the beginning of the assembly and at the beginning of each new program section.

Every VAX MACRO source statement that allocates memory in the object module increments the value of the current location counter by the number of bytes allocated. For example, the directive .LONG 0 increments the current location counter by 4. However, with the exception of the special form described below, a direct assignment statement does not increase the current location counter because no memory is allocated.

The current location counter can be explicitly set by a special form of the direct assignment statement. The location counter can be either incremented or decremented. This method of setting the location counter is often useful when defining data structures. Data storage areas should not be reserved by explicitly setting the location counter; use the .BLK*x* directives (see Chapter 6).

Format

.=expression

expression
An expression that does not contain any undefined symbols (see Section 3.5).

In a relocatable program section, the expression must be relocatable; that is, the expression must be relative to an address in the current program section. It may be relative to the current location counter.

For example:

```
. = .+40        ; Moves location counter forward
```

When a program section that you defined in the current module is continued, the current location counter is set to the last value of the current location counter in that program section.

When you use the current location counter in the operand field of an instruction, the current location counter has the value of the address of that operand; it does not have the value of the address of the beginning of the instruction. For this reason, you would not normally use the current location counter as a part of the operand specifier.

4 Macro Arguments and String Operators

By using macros, you can use a single line to insert a sequence of source lines into a program.

A macro definition contains the source lines of the macro. The macro definition can optionally have formal arguments. These formal arguments can be used throughout the sequence of source lines. Later, the formal arguments are replaced by the actual arguments in the macro call.

The macro call consists of the macro name optionally followed by actual arguments. The assembler replaces the line containing the macro call with the source lines in the macro definition. It replaces any occurrences of formal arguments in the macro definition with the actual arguments specified in the macro call. This process is called the macro expansion.

The macro directives (described in Chapter 6) provide facilities for performing eight categories of functions. Table 6–2 lists these categories and the directives that fall under them.

By default, macro expansions are not printed in the assembly listing. They are printed only when the .SHOW directive (see description in Chapter 6) or the /SHOW qualifier (described in the *VMS DCL Dictionary*) specifies the EXPANSIONS argument. In the examples in this chapter, the macro expansions are listed as they would appear if .SHOW EXPANSIONS was specified in the source file or /SHOW=EXPANSIONS was specified in the MACRO command string.

The remainder of this chapter describes macro arguments, created local labels, and the macro string operators.

4.1 Arguments in Macros

Macros have two types of arguments: actual and formal. Actual arguments are the strings given in the macro call after the name of the macro. Formal arguments are specified by name in the macro definition; that is, after the macro name in the .MACRO directive. Actual arguments in macro calls and formal arguments in macro definitions can be separated by commas, tabs, or spaces.

The number of actual arguments in the macro call can be less than or equal to the number of formal arguments in the macro definition. If the number of actual arguments is greater than the number of formal arguments, the assembler displays an error message.

Formal and actual arguments normally maintain a strict positional relationship. That is, the first actual argument in a macro call replaces all occurrences of the first formal argument in the macro definition. This strict positional relationship can be overridden by the use of keyword arguments (see Section 4.3).

An example of a macro definition using formal arguments follows:

```
.MACRO  STORE  ARG1,ARG2,ARG3
.LONG   ARG1                    ; ARG1 is first argument
.WORD   ARG3                    ; ARG3 is third argument
.BYTE   ARG2                    ; ARG2 is second argument
.ENDM   STORE
```

The following two examples show possible calls and expansions of the macro defined previously:

```
STORE   3,2,1                   ; Macro call
.LONG   3                       ; 3 is first argument
.WORD   1                       ; 1 is third argument
.BYTE   2                       ; 2 is second argument

STORE   X,X-Y,Z                 ; Macro call
#.LONG   X                       ; X is first argument
#.WORD   Z                       ; Z is third argument
#.BYTE   X-Y                     ; X-Y is second argument
```

4.2 Default Values

Default values are values that are defined in the macro definition. They are used when no value for a formal argument is specified in the macro call.

Default values are specified in the .MACRO directive as follows:

formal-argument-name = default-value

An example of a macro definition specifying default values follows:

```
.MACRO  STORE   ARG1=12,ARG2=0,ARG3=1000
.LONG   ARG1
.WORD   ARG3
.BYTE   ARG2
.ENDM   STORE
```

The following three examples show possible calls and expansions of the macro defined previously:

```
STORE                   ; No arguments supplied
.LONG   12
.WORD   1000
.BYTE   0

STORE   ,5,X            ; Last two arguments supplied
.LONG   12
.WORD   X
.BYTE   5

STORE   1               ; First argument supplied
.LONG   1
.WORD   1000
.BYTE   0
```

4.3 Keyword Arguments

Keyword arguments allow a macro call to specify the arguments in any order. The macro call must specify the same formal argument names that appear in the macro definition. Keyword arguments are useful when a macro definition has more formal arguments than need to be specified in the call.

In any one macro call, the arguments should be either all positional arguments or all keyword arguments. When positional and keyword arguments are combined in a macro, only the positional arguments correspond by position to the formal arguments; the keyword arguments are not used. If a formal argument corresponds to both a positional argument and a keyword argument, the argument that appears last in the macro call overrides any other argument definition for the same argument.

For example, the following macro definition specifies three arguments:

```
.MACRO  STORE   ARG1,ARG2,ARG3
.LONG   ARG1
.WORD   ARG3
.BYTE   ARG2
.ENDM   STORE
```

The following macro call specifies keyword arguments:

```
STORE   ARG3=27+5/4,ARG2=5,ARG1=SYMBL
.LONG   SYMBL
.WORD   27+5/4
.BYTE   5
```

Because the keywords are specified in the macro call, the arguments in the macro call need not be given in the order they were listed in the macro definition.

4.4 String Arguments

If an actual argument is a string containing characters that the assembler interprets as separators (such as a tab, space, or comma), the string must be enclosed by delimiters. String delimiters are usually paired angle brackets (<>).

The assembler also interprets any character after an initial circumflex (^) as a delimiter. To pass an angle bracket as part of a string, you can use the circumflex form of the delimiter.

The following are examples of delimited macro arguments:

```
<HAVE THE SUPPLIES RUN OUT?>
<LAST NAME, FIRST NAME>
<LAB:    CLRL    R4>
^%ARGUMENT IS <LAST,FIRST> FOR CALL%
^?EXPRESSION IS <5+3>*<4+2>?
```

In the last two examples, the initial circumflex indicates that the percent sign (%) and question mark (?) are the delimiters. Note that only the left hand delimiter is preceded by a circumflex.

The assembler interprets a string argument enclosed by delimiters as one actual argument and associates it with one formal argument. If a string argument that contains separator characters is not enclosed by delimiters, the assembler interprets it as successive actual arguments and associates it with successive formal arguments.

For example, the following macro call has one formal argument:

```
.MACRO  REPEAT STRNG
.ASCII  /STRNG/
.ASCII  /STRNG/
.ENDM   REPEAT
```

The following two macro calls demonstrate actual arguments with and without delimiters:

```
REPEAT  <A B C D E>
.ASCII  /A B C D E/
.ASCII  /A B C D E/

REPEAT  A B C D E
%MACRO-E-TOOMNYARGS, Too many arguments in MACRO call
```

Note that the assembler interpreted the second macro call as having five actual arguments instead of one actual argument with spaces.

When a macro is called, the assembler removes any delimiters around a string before associating it with the formal arguments.

If a string contains a semicolon, the string must be enclosed by delimiters, or the semicolon will mark the start of the comment field.

Strings enclosed by delimiters cannot be continued on a new line.

To pass a number containing a radix or unary operator (for example, ^XF19), the entire argument must be enclosed by delimiters, or the assembler will interpret the radix operator as a delimiter.

The following are macro arguments that are enclosed in delimiters because they contain radix operators:

```
<^XF19>
<^B01100011>
<^F1.5>
```

Macros can be nested; that is, a macro definition can contain a call to another macro. If, within a macro definition, another macro is called and is passed a string argument, you must delimit the argument so that the entire string is passed to the second macro as one argument.

The following macro definition contains a call to the REPEAT macro defined in an earlier example:

```
        .MACRO   CNTRPT LAB1,LAB2,STR_ARG
LAB1:   .BYTE    LAB2-LAB1-1            ; Length of 2*string
        REPEAT   <STR_ARG>              ; Call REPEAT macro
LAB2:
        .ENDM    CNTRPT
```

Note that the argument in the call to REPEAT is enclosed in angle brackets even though it does not contain any separator characters. The argument is thus delimited because it is a formal argument in the definition of the macro CNTRPT and will be replaced with an actual argument that may contain separator characters.

The following example calls the macro CNTRPT, which in turn calls the macro REPEAT:

```
        CNTRPT  ST,FIN,<LEARN YOUR ABC'S>
ST:     .BYTE   FIN-ST-1                ; Length of 2*string
        REPEAT  <LEARN YOUR ABC'S>      ; Call REPEAT macro
        .ASCII  /LEARN YOUR ABC'S/
        .ASCII  /LEARN YOUR ABC'S/
FIN:
```

An alternative method to pass string arguments in nested macros is to enclose the macro argument in nested delimiters. Do not use delimiters around the macro calls in the macro definitions. Each time you use the delimited argument in a macro call, the assembler removes the outermost pair of delimiters before associating it with the formal argument. This method is not recommended because it requires that you know how deeply a macro is nested.

The following macro definition also contains a call to the REPEAT macro:

```
        .MACRO  CNTRPT2 LAB1,LAB2,STR_ARG
LAB1:   .BYTE   LAB2-LAB1-1             ; Length of 2*string
        REPEAT  STR_ARG                 ; Call REPEAT macro
LAB2:
        .ENDM   CNTRPT2
```

Note that the argument in the call to REPEAT is not enclosed in angle brackets.

The following example calls the macro CNTRPT2:

```
        CNTRPT2 BEG,TERM,<<MIND YOUR P'S AND Q'S>>
BEG:    .BYTE   TERM-BEG-1              ; Length of 2*string
        REPEAT  <MIND YOUR P'S AND Q'S> ; Call REPEAT macro
        .ASCII  /MIND YOUR P'S AND Q'S/
        .ASCII  /MIND YOUR P'S AND Q'S/
TERM:
```

Note that even though the call to REPEAT in the macro definition is not enclosed in delimiters, the call in the expansion is enclosed because the call to CNTRPT2 contains nested delimiters around the string argument.

4.5 Argument Concatenation

The argument concatenation operator, the apostrophe ('), concatenates a macro argument with some constant text. Apostrophes can either precede or follow a formal argument name in the macro source.

If an apostrophe precedes the argument name, the text before the apostrophe is concatenated with the actual argument when the macro is expanded. For example, if ARG1 is a formal argument associated with the actual argument TEST, ABCDE'ARG1 is expanded to ABCDETEST.

If an apostrophe follows the formal argument name, the actual argument is concatenated with the text that follows the apostrophe when the macro is expanded. For example, if ARG2 is a formal argument associated with the actual argument MOV, ARG2'L is expanded to MOVL.

Note that the apostrophe itself does not appear in the macro expansion.

To concatenate two arguments, separate the two formal arguments with two successive apostrophes. Two apostrophes are needed because each concatenation operation discards an apostrophe from the expansion.

An example of a macro definition that uses concatenation follows:

```
        .MACRO CONCAT    INST,SIZE,NUM
TEST'NUM':
        INST''SIZE       R0,R'NUM
TEST'NUM'X:
        .ENDM   CONCAT
```

Note that two successive apostrophes are used when concatenating the two formal arguments INST and SIZE.

An example of a macro call and expansion follows:

```
        CONCAT  MOV,L,5
TEST5:
        MOVL    R0,R5
TEST5X:
```

4.6 Passing Numeric Values of Symbols

When a symbol is specified as an actual argument, the name of the symbol, not the numeric value of the symbol, is passed to the macro. The value of the symbol can be passed by inserting a backslash (\) before the symbol in the macro call. The assembler passes the characters representing the decimal value of the symbol to the macro. For example, if the symbol COUNT has a value of 2 and the actual argument specified is \COUNT, the assembler passes the string "2" to the macro; it does not pass the name of the symbol, "COUNT".

Passing numeric values of symbols is especially useful with the apostrophe (') concatenation operator for creating new symbols.

An example of a macro definition for passing numeric values of symbols follows:

```
.MACRO  TESTDEF,TESTNO,ENTRYMASK=^?^M<>?
.ENTRY  TEST'TESTNO,ENTRYMASK      ; Uses arg concatenation
.ENDM   TESTDEF
```

The following example shows a possible call and expansion of the macro defined previously:

```
COUNT = 2
        TESTDEF \COUNT
        .ENTRY  TEST2,^M<>           ; Uses arg concatenation
COUNT = COUNT + 1
        TESTDEF \COUNT,^?^M<R3,R4>?
        .ENTRY  TEST3,^M<R3,R4>      ; Uses arg concatenation
```

4.7 Created Local Labels

Local labels are often very useful in macros. Although you can create a macro definition that specifies local labels within it, these local labels might be duplicated elsewhere in the local label block possibly causing errors. However, the assembler can create local labels in the macro expansion which will not conflict with other local labels. These labels are called created local labels.

Created local labels range from 30000$ through 65535$. Each time the assembler creates a new local label, it increments the numeric part of the label name by 1. Consequently, no user-defined local labels should be in the range of 30000$ through 65535$.

A created local label is specified by a question mark (?) in front of the formal argument name. When the macro is expanded, the assembler creates a new local label if the corresponding actual argument is blank. If the corresponding actual argument is specified, the assembler substitutes the actual argument for the formal argument. Created local symbols can be used only in the first 31 formal arguments specified in the .MACRO directive.

Created local labels can be associated only with positional actual arguments; created local labels cannot be associated with keyword actual arguments.

The following example is a macro definition specifying a created local label:

```
        .MACRO  POSITIVE        ARG1,?L1
        TSTL    ARG1
        BGEQ    L1
        MNEGL   ARG1,ARG1
L1:     .ENDM   POSITIVE
```

The following three calls and expansions of the macro defined previously show both created local labels and a user-defined local label:

```
        POSITIVE  R0
        TSTL    R0
        BGEQ    30000$
        MNEGL   R0,R0
30000$:
        POSITIVE  COUNT
        TSTL    COUNT
        BGEQ    30001$
        MNEGL   COUNT,COUNT
30001$:
        POSITIVE  VALUE,10$
        TSTL    VALUE
        BGEQ    10$
        MNEGL   VALUE,VALUE
10$:
```

4.8 Macro String Operators

Following are the three macro string operators:

- %LENGTH
- %LOCATE
- %EXTRACT

These operators perform string manipulations on macro arguments and ASCII strings. They can be used only in macros and repeat blocks. The following sections describe these operators and give their formats and examples of their use.

4.8.1 %LENGTH Operator

Format

%LENGTH(string)

string
A macro argument or a delimited string. The string can be delimited by angle brackets or a character preceded by a circumflex (see Section 4.4).

DESCRIPTION

The %LENGTH operator returns the length of a string. For example, the value of %LENGTH(<ABCDE>) is 5.

EXAMPLES

Macro definition:

1

```
        .MACRO  CHK_SIZE     ARG1                ; Macro checks if ARG1
        .IF GREATER_EQUAL    %LENGTH(ARG1)-3     ;   is between 3 and
        .IF LESS_THAN        6-%LENGTH(ARG1)     ;   6 characters long
        .ERROR  ; Argument ARG1 is greater than 6 characters
        .ENDC                                    ; If more than 6
        .IF_FALSE                                ; If less than 3
        .ERROR  ; Argument ARG1 is less than 3 characters
        .ENDC                                    ; Otherwise do nothing
        .ENDM   CHK_SIZE
```

Macro calls and expansions of the macro defined previously:

2

```
        CHK_SIZE        A                        ; Macro checks if A
        .IF GREATER_EQUAL   1-3                  ;   is between 3 and
        .IF LESS_THAN       6-1                  ;   6 characters long.
                                                 ;   Should be too short.
        .ERROR  ; Argument A is greater than 6 characters
        .ENDC                                    ; If more than 6
        .IF_FALSE                                ; If less than 3
%MACRO-E-GENERR, Generated ERROR: Argument A is less than 3 characters

        .ENDC                                    ; Otherwise do nothing
```

3

```
CHK_SIZE        ABC                     ; Macro checks if ABC
.IF GREATER_EQUAL   3-3                 ;   is between 3 and
.IF LESS_THAN       6-3                 ;   6 characters long.
                                        ;   Should be ok.
.ERROR  ; Argument ABC is greater than 6 characters
.ENDC                                   ; If more than 6
.IF_FALSE                               ; If less than 3
.ERROR  ; Argument ABC is less than 3 characters
.ENDC                                   ; Otherwise do nothing
```

4.8.2 %LOCATE Operator

Format

%LOCATE(string1,string2 [,symbol])

Parameters

string1
A substring. The substring can be written either as a macro argument or as a delimited string. The delimiters can be either angle brackets or a character preceded by a circumflex.

string2
The string to be searched for the substring. The string can be written either as a macro argument or as a delimited string. The delimiters can be either angle brackets or a character preceded by a circumflex.

symbol
An optional symbol or decimal number that specifies the position in string2 at which the assembler should start the search. If this argument is omitted, the assembler starts the search at position 0 (the beginning of the string). The symbol must be an absolute symbol that has been previously defined; the number must be an unsigned decimal number. Expressions and radix operators are not allowed.

DESCRIPTION

The %LOCATE operator locates a substring within a string. If %LOCATE finds a match of the substring, it returns the character position of the first character of the match in the string. For example, the value of %LOCATE(<D> , <ABCDEF>) is 3. Note that the first character position of a string is 0. If %LOCATE does not find a match, it returns a value equal to the length of the string. For example, the value of %LOCATE(<Z> , <ABCDEF>) is 6.

The %LOCATE operator returns a numeric value that can be used in any expression.

EXAMPLES

Macro definition:

1

```
.MACRO  BIT_NAME ARG1    ; Checks if ARG1 is in list
 .IF EQUAL  %LOCATE(ARG1,<DELDFWDLTDMOESC>)-15
                         ; If it is not, print error
 .ERROR  ; ARG1 is an invalid bit name
 .ENDC                   ; If it is, do nothing
 .ENDM  BIT_NAME
```

Macro calls and expansions of the macro defined previously:

2

```
        BIT_NAME  ESC           ; Is ESC in list
        .IF EQUAL  12-15        ; If it is not, print error
        .ERROR  ; ESC is an invalid bit name
        .ENDC                   ; If it is, do nothing

        BIT_NAME  FOO           ; Not in list
        .IF EQUAL  15-15
                                ; If it is not, print error
%MACRO-E-GENERR, Generated ERROR:  FOO is an invalid bit name

        .ENDC                   ; If it is, do nothing
```

Note: If the optional symbol is specified, the search begins at the character position of string2 specified by the symbol. For example, the value of %LOCATE(<ACE> , <SPACE_HOLDER> ,5) is 12 because there is no match after the 5th character position.

4.8.3 %EXTRACT Operator

Format

%EXTRACT(symbol1,symbol2,string)

Parameters

symbol1

A symbol or decimal number that specifies the starting position of the substring to be extracted. The symbol must be an absolute symbol that has been previously defined; the number must be an unsigned decimal number. Expressions and radix operators are not allowed.

symbol2

A symbol or decimal number that specifies the length of the substring to be extracted. The symbol must be an absolute symbol that has been previously defined; the number must be an unsigned decimal number. Expressions and radix operators are not allowed.

string

A macro argument or a delimited string. The string can be delimited by angle brackets or a character preceded by a circumflex.

DESCRIPTION

The %EXTRACT operator extracts a substring from a string. It returns the substring that begins at the specified position and is of the specified length. For example, the value of %EXTRACT(2,3, <ABCDEF>) is CDE. Note that the first character in a string is in position 0.

EXAMPLES

Macro definition:

1
```
        .MACRO  RESERVE ARG1
XX = %LOCATE(<=>,ARG1)
        .IF EQUAL   XX-%LENGTH(ARG1)
        .WARN   ; Incorrect format for macro call - ARG1
        .MEXIT
        .ENDC
%EXTRACT(0,XX,ARG1)::
XX = XX+1
        .BLKB   %EXTRACT(XX,3,ARG1)
        .ENDM   RESERVE
```

Macro calls and expansions of the macro defined previously:

2
```
        RESERVE FOOBAR
XX = 6
        .IF EQUAL  XX-6
%MACRO-W-GENWRN, Generated WARNING:  Incorrect format for macro call - FOOBAR

        .MEXIT
```

3
```
        RESERVE LOCATION=12
XX = 8
        .IF EQUAL  XX-11
        .WARN   ; Incorrect format for macro call - LOCATION=12
        .MEXIT
        .ENDC
LOCATION::
XX = XX+1
        .BLKB   12
```

Note: If the starting position specified is equal to or greater than the length of the string, or if the length specified is 0, %EXTRACT returns a null string (a string of 0 characters).

5 MACRO Addressing Modes

This section summarizes the VAX addressing modes and contains examples of VAX MACRO statements that use these addressing modes. Table 5–1 summarizes the addressing modes. Chapter 8 describes the addressing mode formats in detail.

The following are the four types of addressing modes:

- General Register
- Program Counter
- Index
- Branch

Although index mode is a general register mode, it is considered separate because it can be used only in combination with another type of mode.

5.1 General Register Modes

The general register modes use registers R0 through R12, AP (the same as R12), FP, and SP.

The following are the eight general register modes:

- Register
- Register deferred
- Autoincrement
- Autoincrement deferred
- Autodecrement
- Displacement
- Displacement deferred
- Literal

MACRO Addressing Modes

5.1 General Register Modes

Table 5–1 Addressing Modes

Type	Addressing Mode	Format	Hex Value	Description	Indexable?
General Register	Register	Rn	5	Register contains the operand.	No
	Register Deferred	(Rn)	6	Register contains the address of the operand.	Yes
	Autoincrement	(Rn)+	8	Register contains the address of the operand; the processor increments the register contents by the size of the operand data type.	Yes
	Autoincrement Deferred	@(Rn)+	9	Register contains the address of the operand address; the processor increments the register contents by 4.	Yes
	Autodecrement	-(Rn)	7	The processor decrements the register contents by the size of the operand data type; the register then contains the address of the operand.	Yes
	Displacement	dis(Rn) B^dis(Rn) W^dis(Rn) L^dis(Rn)	 A C E	The sum of the contents of the register and the displacement is the address of the operand; B^, W^, and L^ respectively indicate byte, word, and longword displacement.	Yes
	Displacement Deferred	@dis(Rn) @B^dis(Rn) @W^dis(Rn) @L^dis(Rn)	 B D F	The sum of the contents of the register and the displacement is the address of the operand address; B^, W^, and L^ respectively indicate, byte, word, and longword displacement.	Yes
	Literal	#literal S^#literal	 0-3	The literal specified is the operand; the literal is stored as a short literal.	No

Key:

Rn — Any general register R0 through R12. Note that the AP, FP, or SP register can be used in place of Rn.
Rx — Any general register R0 through R12. Note that the AP, FP, or SP register can be used in place of Rx. Rx cannot be the same as the Rn specified in the base-mode for certain base modes (see Section 5.3).
dis — An expression specifying a displacement.
address — An expression specifying an address.
literal — An expression, an integer constant, or a floating-point constant.

Table 5–1 (Cont.) Addressing Modes

Type	Addressing Mode	Format	Hex Value	Description	Indexable?
Program Counter	Relative	address B^address W^address L^address	 A C E	The address specified is the address of the operand; the address is stored as a displacement from the PC; B^, W^, and L^ respectively indicate byte, word, and longword displacement.	Yes
	Relative Deferred	@address @B^address @W^address @L^address	 B D F	The address specified is the address of the operand address; the address specified is stored as a displacement from the PC; B^, W^, and L^ indicate byte, word, and longword displacement respectively.	Yes
	Absolute	@#address	9	The address specified is the address of the operand; the address specified is stored as an absolute virtual address, not as a displacement.	Yes
	Immediate	#literal I^#literal	 8	The literal specified is the operand; the literal is stored as a byte, word, longword, or quadword.	No
	General	G^address	—	The address specified is the address of the operand; if the address is defined as relocatable, the Linker stores the address as a displacement from the PC; if the address is defined as an absolute virtual address, the Linker stores the address as an absolute value.	Yes

Key:

Rn — Any general register R0 through R12. Note that the AP, FP, or SP register can be used in place of Rn.
Rx — Any general register R0 through R12. Note that the AP, FP, or SP register can be used in place of Rx. Rx cannot be the same as the Rn specified in the base-mode for certain base modes (see Section 5.3).
dis — An expression specifying a displacement.
address — An expression specifying an address.
literal — An expression, an integer constant, or a floating-point constant.

Table 5–1 (Cont.) Addressing Modes

Type	Addressing Mode	Format	Hex Value	Description	Indexable?
Index	Index	base-mode[Rx]	4	The base-mode specifies the base address and the register specifies the index; the sum of the base address and the product of the contents of Rx and the size of the operand data type is the address of the operand; base mode can be any addressing mode except register, immediate, literal, index, or branch.	No
Branch	Branch	address	—	The address specified is the operand; this address is stored as a displacement from the PC; branch mode can only be used with the branch instructions.	No

Key:

Rn — Any general register R0 through R12. Note that the AP, FP, or SP register can be used in place of Rn.
Rx — Any general register R0 through R12. Note that the AP, FP, or SP register can be used in place of Rx. Rx cannot be the same as the Rn specified in the base-mode for certain base modes (see Section 5.3).
dis — An expression specifying a displacement.
address — An expression specifying an address.
literal — An expression, an integer constant, or a floating-point constant.

5.1.1 Register Mode

In register mode, the operand is the contents of the specified register, except in the following cases:

- For quadword, D_floating, G_floating, or variable-bit field operands, the operand is the contents of register n concatenated with the contents of register n+1.
- For octaword and H_floating operands, the operand is the contents of register n concatenated with the contents of registers n+1, n+2, and n+3.

In each of these cases, the least significant bytes of the operand are in register n and the most significant bytes are in the highest register used, either n+1 or n+3.

The results of the operation are unpredictable if you use the PC in register mode or if you use a large data type that extends the operand into the PC.

Formats

Rn
AP
FP
SP

n
A number in the range of 0 through 12.

EXAMPLE

```
CLRB    R0      ; Clear lowest byte of R0
CLRQ    R1      ; Clear R1 and R2
TSTW    R10     ; Test lower word of R10
INCL    R4      ; Add 1 to R4
```

5.1.2 Register Deferred Mode

In register deferred mode, the register contains the address of the operand. Register deferred mode can be used with index mode (see Section 5.3).

Formats

(Rn)
(AP)
(FP)
(SP)

Parameters

n
A number in the range of 0 through 12.

EXAMPLE

```
        MOVAL   LDATA,R3        ; Move address of LDATA to R3
        CMPL    (R3),R0         ; Compare value at LDATA to R0
        BEQL    10$             ; If they are the same, ignore
        CLRL    (R3)            ; Clear longword at LDATA
10$:    MOVL    (SP),R1         ; Copy top item of stack into R1
        MOVZBL  (AP),R4         ; Get number of arguments in call
```

5.1.3 Autoincrement Mode

In autoincrement mode, the register contains the address of the operand. After evaluating the operand address contained in the register, the processor increments that address by the size of the operand data type. The processor increments the contents of the register by 1, 2, 4, 8, or 16 for a byte, word, longword, quadword, or octaword operand, respectively.

Autoincrement mode can be used with index mode (see Section 5.3), but the index register cannot be the same as the register specified in autoincrement mode.

Formats

(Rn)+
(AP)+
(FP)+
(SP)+

Parameters

n
A number in the range of 0 through 12.

EXAMPLE

```
MOVAL   TABLE,R1                ; Get address of TABLE.
CLRQ    (R1)+                   ; Clear first and second longwords
CLRL    (R1)+                   ;   and third longword in TABLE;
                                ;   leave R1 pointing to TABLE+12.

MOVAB   BYTARR,R2               ; Get address of BYTARR.
INCB    (R2)+                   ; Increment first byte of BYTARR
INCB    (R2)+                   ;   and second.

XORL3   (R3)+,(R4)+,(R5)+       ; Exclusive-OR the two longwords
                                ;   whose addresses are stored in
                                ;   R3 and R4 and store result in
                                ;   address contained in R5; then
                                ;   add 4 to R3, R4, and R5.
```

5.1.4 Autoincrement Deferred Mode

In autoincrement deferred mode, the register contains an address that is the address of the operand address (a pointer to the operand). After evaluating the operand address, the processor increments the contents of the register by 4 (the size in bytes of an address).

Autoincrement deferred mode can be used with index mode (see Section 5.3), but the index register cannot be the same as the register specified in autoincrement deferred mode.

Formats

@(Rn)+
@(AP)+
@(FP)+
@(SP)+

Parameters

n
A number in the range of 0 through 12.

EXAMPLE

```
MOVAL   PNTLIS,R2       ; Get address of pointer list.
CLRQ    @(R2)+          ; Clear quadword pointed to by
                        ;   first absolute address in PNTLIS;
                        ;   then add 4 to R2.
CLRB    @(R2)+          ; Clear byte pointed to by second
                        ;   absolute address in PNTLIS
                        ;   then add 4 to R2.
MOVL    R10,@(R0)+      ; Move R10 to location whose address
                        ;   is pointed to by R0; then add 4
                        ;   to R0.
```

5.1.5 Autodecrement Mode

In autodecrement mode, the processor decrements the contents of the register by the size of the operand data type; the register contains the address of the operand. The processor decrements the register by 1, 2, 4, 8, or 16 for byte, word, longword, quadword, or octaword operands, respectively.

Autodecrement mode can be used with index mode (see Section 5.3), but the index register cannot be the same as the register specified in autodecrement mode.

Formats

–(Rn)
–(AP)
–(FP)
–(SP)

Parameters

n
A number in the range of 0 through 12.

EXAMPLE

```
CLRO    -(R1)           ; Subtract 8 from R1 and zero
                        ;   the octaword whose address
                        ;   is in R1.
MOVZBL  R3,-(SP)        ; Push the zero-extended low byte
                        ;   of R3 onto the stack as a
                        ;   longword.
CMPB    R1,-(R0)        ; Subtract 1 from R0 and compare
                        ;   low byte of R1 with byte whose
                        ;   address is now in R0.
```

5.1.6 Displacement Mode

In displacement mode, the contents of the register plus the displacement (sign-extended to a longword) produce the address of the operand.

Displacement mode can be used with index mode (see Section 5.3). If used in displacement mode, the index register can be the same as the base register.

Formats

dis(Rn)
dis(AP)
dis(FP)
dis(SP)

Parameters

n
A number in the range of 0 through 12.

dis
An expression specifying a displacement; the expression can be preceded by one of the following displacement length specifiers, which indicate the number of bytes needed to store the displacement.

Displacement Length Specifier	Meaning
B^	Displacement requires one byte.
W^	Displacement requires one word (two bytes).
L^	Displacement requires one longword (four bytes).

If no displacement length specifier precedes the expression, and the value of the expression is known, the assembler chooses the smallest number of bytes (one, two, or four) needed to store the displacement. If no length specifier precedes the expression, and the value of the expression is unknown, the assembler reserves one word (two bytes) for the displacement. Note that if the displacement is either relocatable or defined later in the source program, the assembler considers it unknown. If the actual displacement does not fit in the memory reserved, the linker displays an error message.

EXAMPLE

```
MOVAB   KEYWORDS,R3          ; Get address of KEYWORDS.

MOVB    B^IO(R3),R4          ; Get byte whose address is IO
                             ;   plus address of KEYWORDS;
                             ;   the displacement is stored
                             ;   as a byte.

MOVB    B^ACCOUNT(R3),R5     ; Get byte whose address is
                             ;   ACCOUNT plus address of
                             ;   KEYWORDS; the displacement
                             ;   is stored as a byte.
```

```
CLRW    L^STA(R1)           ; Clear word whose address
                            ;   is STA plus contents of R1;
                            ;   the displacement is stored
                            ;   as a longword.

MOVL    R0,-2(R2)           ; Move R0 to address that is -2
                            ;   plus the contents of R2; the
                            ;   displacement is stored as a byte.

TSTB    EXTRN(R3)           ; Test the byte whose address
                            ;   is EXTRN plus the address
                            ;   of KEYWORDS; the displacement
                            ;   is stored as a word, since
                            ;   EXTRN is undefined.

MOVAB   2(R5),R0            ; Move <contents of R5> + 2
                            ;   to R0.
```

Note: If the value of the displacement is 0, and no displacement length is specified, the assembler uses register deferred mode rather than displacement mode.

5.1.7 Displacement Deferred Mode

In displacement deferred mode, the contents of the register plus the displacement (sign-extended to a longword) produce the address of the operand address (a pointer to the operand).

Displacement deferred mode can be used with index mode (see Section 5.3). If used in displacement deferred mode, the index register can be the same as the base register.

Formats

@dis(Rn)
@dis(AP)
@dis(FP)
@dis(SP)

Parameters

n
A number in the range of 0 through 12.

dis
An expression specifying a displacement; the expression can be preceded by one of the following displacement length specifiers, which indicate the number of bytes needed to store the displacement.

Displacement Length Specifier	Meaning
B^	Displacement requires one byte.
W^	Displacement requires one word (two bytes).
L^	Displacement requires one longword. (four bytes)

If no displacement length specifier precedes the expression, and the value of the expression is known, the assembler chooses the smallest number of bytes (one, two, or four) needed to store the displacement. If no length specifier precedes the expression, and the value of the expression is unknown, the assembler reserves one word (two bytes) for the displacement. Note that if the displacement is either relocatable or defined later in the source program, the assembler considers it unknown. If the actual displacement does not fit in the memory the assembler has reserved, the linker displays an error message.

EXAMPLE

```
MOVAL   ARRPOINT,R6              ; Get address of array of pointers.
CLRL    @16(R6)                  ; Clear longword pointed to by
                                 ;   longword whose address is
                                 ;   <16 + address of ARRPOINT>; the
                                 ;   displacement is stored as a byte.

MOVL    @B^OFFS(R6),@RSOFF(R6)   ; Move the longword pointed to
                                 ;   by longword whose address is
                                 ;   <OFFS + address of ARRPOINT>
                                 ;   to the address pointed to by
                                 ;   longword whose address is
                                 ;   <RSOFFS + address of ARRPOINT>;
                                 ;   the first displacement is
                                 ;   stored as a byte; the second
                                 ;   displacement is stored as a word.

CLRW    @84(R2)                  ; Clear word pointed to by
                                 ;   <longword at 84 + contents of R2>;
                                 ;   the assembler uses byte
                                 ;   displacement automatically.
```

5.1.8 Literal Mode

In literal mode, the value of the literal is stored in the addressing mode byte.

Formats

#literal
S^#literal

Parameters

literal
An expression, an integer constant, or a floating-point constant. The literal must fit in the short literal form. That is, integers must be in the range of 0 through 63 and floating-point constants must be one of the 64 values listed in Table 5–2 and Table 5–3. Floating-point short literals are stored with a 3-bit exponent and a 3-bit fraction. Table 5–2 and Table 5–3 also show the value of the exponent and the fraction for each literal. See Section 8.6.8 for information on the format of short literals.

Table 5–2 Floating-Point Literals Expressed as Decimal Numbers

Exponent	0	1	2	3	4	5	6	7
0	0.5	0.5625	0.625	0.6875	0.75	0.8125	0.875	0.9375
1	1.0	1.125	1.25	1.37	1.5	1.625	1.75	1.875
2	2.0	2.25	2.5	2.75	3.0	3.25	3.5	3.75
3	4.0	4.5	5.0	5.5	6.0	6.5	7.0	7.5
4	8.0	9.0	10.0	11.0	12.0	13.0	14.0	15.0
5	16.0	18.0	20.0	22.0	24.0	26.0	28.0	30.0
6	32.0	36.0	40.0	44.0	48.0	52.0	56.0	60.0
7	64.0	72.0	80.0	88.0	96.0	104.0	112.0	120.0

Table 5–3 Floating-Point Literals Expressed as Rational Numbers

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

EXAMPLE

```
MOVL    #1,R0           ; R0 is set to 1; the 1 is stored
                        ;   in the instruction as a short
                        ;   literal.

MOVB    S^#CR,R1        ; The low byte of R1 is set
                        ;   to the value CR.
                        ;   CR is stored in the instruction
                        ;   as a short literal.
                        ;   If CR is not in range 0-63,
                        ;   the linker produces a
                        ;   truncation error.

MOVF    #0.625,R6       ; R6 is set to the floating-point
                        ;   value 0.625; it is stored
                        ;   in the floating-point short
                        ;   literal form.
```

Notes

1 When you use the #literal format, the assembler chooses whether to use literal mode or immediate mode (see Section 5.2.4). The assembler uses immediate mode if any of the following conditions is satisfied:

- The value of the literal does not fit in the short literal form.
- The literal is a relocatable or external expression (see Section 3.5).

- The literal is an expression that contains undefined symbols.

The difference between immediate mode and literal mode is the amount of storage that it takes to store the literal in the instruction.

2 The S^#literal format forces the assembler to use literal mode.

5.2 Program Counter Modes

The program counter modes use the PC for a general register. Following are the five program counter modes:

- Relative
- Relative Deferred
- Absolute
- Immediate
- General

5.2.1 Relative Mode

In relative mode, the address specified is the address of the operand. The assembler stores the address as a displacement from the PC.

Relative mode can be used with index mode (see Section 5.3).

Format

address

Parameters

address
An expression specifying an address; the expression can be preceded by one of the following displacement length specifiers, which indicate the number of bytes needed to store the displacement.

Displacement Length Specifier	Meaning
B^	Displacement requires one byte.
W^	Displacement requires one word (two bytes).
L^	Displacement requires one longword. (four bytes)

If no displacement length specifier precedes the address expression, and the value of the expression is known, the assembler chooses the smallest number of bytes (one, two, or four) needed to store the displacement. If no length specifier precedes the address expression, and the value of the expression is unknown, the assembler uses the default displacement length (see the description of .DEFAULT in Chapter 6). If the address expression is either defined later in the program or defined in another program section, the assembler considers the value unknown.

EXAMPLE

```
MOVL    LABEL,R1          ; Get longword at LABEL; the
                          ;   assembler uses default
                          ;   displacement unless LABEL was
                          ;   previously defined in this
                          ;   section

CMPL    W^<DATA+4>,R10    ; Compare R10 with longword at
                          ;   address DATA+4; CMPL
                          ;   uses a word displacement
```

5.2.2 Relative Deferred Mode

In relative deferred mode, the address specified is the address of the operand address (a pointer to the operand). The assembler stores the address specified as a displacement from the PC.

Relative deferred mode can be used with index mode (see Section 5.3).

Format

@address

Parameters

address
An expression specifying an address; the expression can be preceded by one of the following displacement length specifiers, which indicate the number of bytes needed to store the displacement:

Displacement Length Specifier	Meaning
B^	Displacement requires one byte.
W^	Displacement requires one word (two bytes).
L^	Displacement requires one longword. (four bytes)

If no displacement length specifier precedes the address expression, and the value of the expression is known, the assembler chooses the smallest number of bytes (one, two, or four) needed to store the displacement. If no length specifier precedes the address expression, and the value of the expression is unknown, the assembler uses the default displacement length (see the description of .DEFAULT in Chapter 6). If the address expression is either defined later in the program or defined in another program section, the assembler considers the value unknown.

EXAMPLE

```
CLRL    @W^PNTR           ; Clear longword pointed to by
                          ;   longword at PNTR; the assembler
                          ;   uses a word displacement

INCB    @L^COUNTS+4       ; Increment byte pointed to by
                          ;   longword at COUNTS+4; assembler
                          ;   uses a longword displacement
```

5.2.3 Absolute Mode

In absolute mode, the address specified is the address of the operand. The address is stored as an absolute virtual address (compare relative mode, where the address is stored as a displacement from the PC).

Absolute mode can be used with index mode (see Section 5.3).

Format

@#address

Parameters

address
An expression specifying an address.

EXAMPLE

```
CLRL    @#^X1100        ; Clear the contents of location 1100(hex)

CLRB    @#ACCOUNT       ; Clear the contents of location
                        ;   ACCOUNT; the address is stored
                        ;   absolutely, not as a displacement

CALLS   #3,@#SYS$FAO    ; Call the procedure SYS$FAO with
                        ;   three arguments on the stack
```

5.2.4 Immediate Mode

In immediate mode, the literal specified is the operand.

Formats

#literal
I^#literal

Parameters

literal
An expression, an integer constant, or a floating-point constant.

EXAMPLE

```
MOVL        #1000,R0        ; R0 is set to 1000; the operand 1000
                            ;   is stored in a longword

MOVB        #BAR,R1         ; The low byte of R1 is set
                            ;   to the value of BAR

MOVF        #0.1,R6         ; R6 is set to the floating-point
                            ;   value 0.1; it is stored
                            ;   as a 4-byte floating-point
                            ;   value (it cannot be
                            ;   represented as a short literal)

ADDL2        I^#5,R0        ; The 5 is stored in a longword
                            ;   because the I^ forces the
                            ;   assembler to use immediate mode
```

```
MOVG      #0.2,R6        ; The value 0.2 is converted
                         ;   to its G_FLOATING representation

MOVG      #PI,R6         ; The value contained in PI is
                         ;   moved to R6; no conversion is
                         ;   performed
```

Notes

1 When you use the #literal format, the assembler chooses whether to use literal mode (Section 5.1.8) or immediate mode. If the literal is an integer from 0 through 63 or a floating-point constant that fits in the short literal form, the assembler uses literal mode. If the literal is an expression, the assembler uses literal mode if all the following conditions are met:

- The expression is absolute.
- The expression contains no undefined symbols.
- The value of the expression fits in the short literal form.

In all other cases, the assembler uses immediate mode.

The difference between immediate mode and literal mode is the amount of storage required to store the literal in the instruction. The assembler stores an immediate mode literal in a byte, word, or longword depending on the operand data type.

2 The I^#literal format forces the assembler to use immediate mode.

3 You can specify floating-point numbers two ways: as a numeric value or as a symbol name. The assembler handles these values in different ways:

- Numeric values are converted to the appropriate internal floating-point representation.
- Symbols are not converted. The assembler assumes that the values have already been converted to internal floating-point representation.

Once the assembler obtains the value, it tries to convert the internal representation of the value to a short floating literal. If conversion fails, the assembler uses immediate mode; if conversion succeeds, the assembler uses short floating literal mode.

5.2.5 General Mode

In general mode, the address you specify is the address of the operand. The linker converts the addressing mode to either relative or absolute mode. If the address is relocatable, the linker converts general mode to relative mode. If the address is absolute, the linker converts general mode to absolute mode. You should use general mode to write position-independent code when you do not know whether the address is relocatable or absolute. A general addressing mode operand requires five bytes of storage.

You can use general mode with index mode (see Section 5.3).

Format

G^address

Parameters

address
An expression specifying an address.

EXAMPLE

```
CLRL    G^LABEL_1            ; Clears the longword at LABEL_1
                             ;   If LABEL_1 is defined as
                             ;   absolute then general mode is
                             ;   converted to absolute
                             ;   mode; if it is defined as
                             ;   relocatable, then general mode is
                             ;   converted to relative mode

CALLS   #5,G^SYS$SERVICE     ; Calls procedure SYS$SERVICE
                             ;   with 5 arguments on stack
```

5.3 Index Mode

Index mode is a general register mode that can be used only in combination with another mode (the base mode). The base mode can be any addressing mode except register, immediate, literal, index, or branch. The assembler first evaluates the base mode to get the base address. To get the operand address, the assembler multiplies the contents of the index register by the number of bytes of the operand data type, then adds the result to the base address.

Combining index mode with the other addressing modes produces the following addressing modes:

- Register deferred index
- Autoincrement index
- Autoincrement deferred index
- Autodecrement index
- Displacement index
- Displacement deferred index
- Relative index
- Relative deferred index
- Absolute index
- General index

The process of first evaluating the base mode and then adding the index register is the same for each of these modes.

Formats

base-mode[Rx]
base-mode[AP]
base-mode[FP]
base-mode[SP]

Parameters

base-mode

Any addressing mode except register, immediate, literal, index, or branch, specifying the base address.

x

A number in the range 0 through 12, specifying the index register.

Table 5–4 lists the formats of index mode addressing.

EXAMPLE

```
;
; Register deferred index mode
;
OFFS=20                                 ; Define OFFS
        MOVAB   BLIST,R9                ; Get address of BLIST
        MOVL    #OFFS,R1                ; Set up index register
        CLRB    (R9)[R1]                ; Clear byte whose address
                                        ;   is the address of BLIST
                                        ;   plus 20*1

        CLRQ    (R9)[R1]                ; Clear quadword whose
                                        ;   address is the address
                                        ;   of BLIST plus 20*8

        CLRO    (R9)[R1]                ; Clear octaword whose
                                        ;   address is the address
                                        ;   of BLIST plus 20*16
;
; Autoincrement index mode
;
        CLRW    (R9)+[R1]               ; Clear word whose address
                                        ;   is address of BLIST plus
                                        ;   20*2; R9 now contains
                                        ;   address of BLIST+2
;
; Autoincrement deferred index mode
;
        MOVAL   POINT,R8                ; Get address of POINT
        MOVL    #30,R2                  ; Set up index register
        CLRW    @(R8)+[R2]              ; Clear word whose address
                                        ;   is 30*2 plus the address
                                        ;   stored in POINT; R8 now
                                        ;   contains 4 plus address of
                                        ;   POINT
;
; Displacement deferred index mode
;
        MOVAL   ADDARR,R9               ; Get address of address array
        MOVL    #100,R1                 ; Set up index register
        TSTF    @40(R9)[R1]             ; Test floating-point value
                                        ;   whose address is 100*4 plus
                                        ;   the address stored at
                                        ;   (ADDARR+40)
```

Table 5–4 Index Mode Addressing

Mode	Format
Register Deferred Index[1 2]	(Rn)[Rx]
Autoincrement Index[1 2]	(Rn)+[Rx]
Autoincrement Deferred Index[1 2]	@(Rn)+[Rx]
Autodecrement Index[1 2]	–(Rn)[Rx]
Displacement Index[1 2 3]	dis(Rn)[Rx]
Displacement Deferred Index[1 2 3]	@dis(Rn)[Rx]
Relative Index[2]	address[Rx]
Relative Deferred Index[2]	@address[Rx]
Absolute Index[2]	@#address[Rx]
General Index[2]	G^address[Rx]

[1] Rn — Any general register R0 through R12 or the AP, FP, or SP register.

[2] Rx — Any general register R0 through R12 or the AP, FP, or SP register. Rx cannot be the same register as Rn in the autoincrement index, autoincrement deferred index, and decrement index addressing modes.

[3] dis — An expression specifying a displacement.

Notes

1. If the base mode alters the contents of its register (autoincrement, autoincrement deferred, and autodecrement), the index mode cannot specify the same register.
2. The index register is added to the address after the base mode is completely evaluated. For example, in autoincrement deferred index mode, the base register contains the address of the operand address. The index register (times the length of the operand data type) is added to the operand address rather than to the address stored in the base register.

5.4 Branch Mode

In branch mode, the address is stored as an implied displacement from the PC. This mode can be used only in branch instructions. The displacement for conditional branch instructions and the BRB instruction is stored in a byte. The displacement for the BRW instruction is stored in a word (two bytes). A byte displacement allows a range of 127 bytes forward and 128 bytes backward. A word displacement allows a range of 32767 bytes forward and 32768 bytes backward. The displacement is relative to the updated PC, the byte past the byte or word where the displacement is stored. See Chapter 9 for more information on the branch instructions.

Format

address

Parameters

address
An expression that represents an address.

EXAMPLE

```
ADDL3   (R1)+,R0,TOTAL        ; Total values and set condition
                              ;   codes
BLEQ    LABEL1                ; Branch to LABEL1 if result is
                              ;   less than or equal to 0
BRW     LABEL                 ; Branch unconditionally to LABEL
```

6 MACRO Assembler Directives

The general assembler directives provide facilities for performing 11 types of functions. Table 6–1 lists these types of functions and their directives.

The macro directives provide facilities for performing eight categories of functions. Table 6–2 lists these categories and their associated directives. Chapter 4 describes macro arguments and string operators.

The remainder of this chapter describes both the general assembler directives and the macro directives, showing their formats and giving examples of their use. For ease of reference, the directives are presented in alphabetical order. Appendix C contains a summary of all assembler directives.

Table 6–1 Summary of General Assembler Directives

Category	Directives[1]
Listing Control Directives	.SHOW (.LIST) .NOSHOW(.NLIST) .TITLE .SUBTITLE (.SBTTL) .IDENT .PAGE
Message Display Directives	.PRINT .WARN .ERROR
Assembler Option Directives	.ENABLE (.ENABL) .DISABLE(.DSABL) .DEFAULT
Data Storage Directives	.BYTE .WORD .LONG .ADDRESS .QUAD .OCTA .PACKED .ASCII .ASCIC .ASCID .ASCIZ .F_FLOATING (.FLOAT) .D_FLOATING (.DOUBLE) .G_FLOATING .H_FLOATING .SIGNED_BYTE .SIGNED_WORD

[1]The alternate form, if any, is given in parentheses.

Table 6–1 (Cont.) Summary of General Assembler Directives

Category	Directives[1]
Location Control Directives	.ALIGN .EVEN .ODD .BLKA .BLKB .BLKD .BLKF .BLKG .BLKH .BLKL .BLKO .BLKQ .BLKW .END
Program Sectioning Directives	.PSECT .SAVE_PSECT (.SAVE) .RESTORE_PSECT (.RESTORE)
Symbol Control Directives	.GLOBAL (.GLOBL) .EXTERNAL (.EXTRN) .DEBUG .WEAK
Routine Entry Point Definition Directives	.ENTRY .TRANSFER .MASK
Conditional and Subconditional Assembly Block Directives	.IF .ENDC .IF_FALSE (.IFF) .IF_TRUE (.IFT) .IF_TRUE_FALSE (.IFTF) .IIF
Cross-Reference Directives	.CROSS .NOCROSS
Instruction Generation Directives	.OPDEF .REF1 .REF2 .REF4 .REF8 .REF16
Linker Option Record Directive	.LINK

[1]The alternate form, if any, is given in parentheses.

Table 6–2 Summary of Macro Directives

Category	Directives[1]
Macro Definition Directives	.MACRO .ENDM
Macro Library Directives	.LIBRARY .MCALL
Macro Deletion Directive	.MDELETE
Macro Exit Directive	.MEXIT
Argument Attribute Directives	.NARG .NCHR .NTYPE
Indefinite Repeat Block Directives	.IRP .IRPC
Repeat Block Directives	.REPEAT (.REPT)
End Range Directive	.ENDR

[1]The alternate form, if any, is given in parentheses.

.ADDRESS

Address storage directive

FORMAT

.ADDRESS *address-list*

PARAMETER

address-list

A list of symbols or expressions, separated by commas, which VAX MACRO interprets as addresses. Repetition factors are not allowed.

DESCRIPTION

.ADDRESS stores successive longwords containing addresses in the object module. DIGITAL recommends that you use .ADDRESS rather than .LONG for storing address data to provide additional information to the linker. In shareable images, addresses that you specify with .ADDRESS produce position-independent code.

EXAMPLE

```
TABLE:  .ADDRESS  LAB_4, LAB_3, ROUTTERM        ; Reference table
```

.ALIGN

Location counter alignment directive

FORMAT

.ALIGN *integer[,expression]*
.ALIGN *keyword[,expression]*

PARAMETERS

integer

An integer in the range of 0 through 9. The location counter is aligned at an address that is the value of 2 raised to the power of the integer.

keyword

One of five keywords that specify the alignment boundary. The location counter is aligned to an address that is the next multiple of the values listed below.

Keyword	Size (in Bytes)
BYTE	2^0 = 1
WORD	2^1 = 2
LONG	2^2 = 4
QUAD	2^3 = 8
PAGE	2^9 = 512

expression

Specifies the fill value to be stored in each byte. The expression must not contain any undefined symbols and must be an absolute expression (see Section 3.5).

DESCRIPTION

.ALIGN aligns the location counter to the boundary specified by either an integer or a keyword.

Notes

1 The alignment that you specify in .ALIGN cannot exceed the alignment of the program section in which the alignment is attempted (see the description of .PSECT). For example, if you are using the default program section alignment (BYTE) and you specify .ALIGN with a WORD or larger alignment, the assembler displays an error message.

2 If the optional expression is supplied, the assembler fills the bytes skipped by the location counter (if any) with the value of that expression. Otherwise, the assembler fills the bytes with zeros.

3 Although most instructions can use byte alignment of data, execution speed is improved by the following alignments:

Data Length	Alignment
Word	Word
Longword	Longword
Quadword	Quadword

EXAMPLE

```
.ALIGN  BYTE,0        ; Byte alignment--fill with null
.ALIGN  WORD          ; Word alignment
.ALIGN  3,^A/ /       ; Quad alignment--fill with blanks
.ALIGN  PAGE          ; Page alignment
```

.ASCIx

ASCII character storage directives

DESCRIPTION

VAX MACRO has four ASCII character storage directives:

Directive	Function
ASCIC	Counted ASCII string storage
ASCID	String-descriptor ASCII string storage
ASCII	ASCII string storage
ASCIZ	Zero-terminated ASCII string storage

Each directive is followed by a string of characters enclosed in a pair of matching delimiters. The delimiters can be any printable character except the space or tab character, equal sign (=), semicolon (;), or left angle bracket (<). The character that you use as the delimiter cannot appear in the string itself. Although you can use alphanumeric characters as delimiters, use nonalphanumeric characters to avoid confusion.

Any character except the null, carriage return, and form feed characters can appear within the string. The assembler does not convert lowercase alphabetic characters to uppercase.

ASCII character storage directives convert the characters to their 8-bit ASCII value (see Appendix A) and store them one character to a byte.

Any character, including the null, carriage return, and form feed characters, can be represented by an expression enclosed in angle brackets outside of the delimiters. You must define the ASCII values of null, carriage return, and form feed with a direct assignment statement. The ASCII character storage directives store the 8-bit binary value specified by the expression.

ASCII strings can be continued over several lines. Use the hyphen as the line continuation character and delimit the string on each line at both ends. Note that you can use a different pair of delimiters for each line. For example:

```
CR=13
LF=10

        .ASCII      /ABC DEFG/
        .ASCIZ      @Any character can be a delimiter@
        .ASCIC      ? lowercase is not converted to UPPER?
        .ASCII      ? this is a test!?<CR><KEY>(LF\TEXT)!Isn't it?!
        .ASCII      \ Angle Brackets <are part <of> this> string \
        .ASCII      / This string is continued / -
                    \ on the next line \
        .ASCII      <CR><KEY>(LF\TEXT)! this string includes an expression!
                    <128+CR>? whose value is a 13 plus 128?
```

.ASCIC

Counted ASCII string storage directive

FORMAT

.ASCIC *string*

PARAMETER

string
A delimited ASCII string.

DESCRIPTION

.ASCIC performs the same function as .ASCII, except that .ASCIC inserts a count byte before the string data. The count byte contains the length of the string in bytes. The length given includes any bytes of nonprintable characters outside the delimited string but excludes the count byte.

.ASCIC is useful in copying text because the count indicates the length of the text to be copied.

EXAMPLE

```
CR=13                               ; Direct assignment statement
                                    ;   defines CR
       .ASCIC     #HELLO#<CR>       ; This counted ASCII string
                                    ;   is equivalent to the
       .BYTE      6                 ;   count followed by
       .ASCII     #HELLO#<CR>       ;   the ASCII string
```

.ASCID

String-descriptor ASCII string storage directive

FORMAT

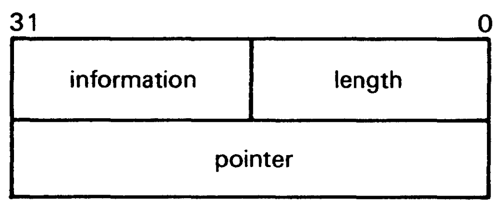
.ASCID *string*

PARAMETER

string
A delimited ASCII string.

DESCRIPTION

.ASCID performs the same function as ASCII, except that .ASCID inserts a string descriptor before the string data. The string descriptor has the following format:



ZK-370-81

Parameters

length
The length of the string (two bytes).

information
Descriptor information (two bytes) is always set to 010E.

pointer
Position independent pointer to the string (four bytes).

String descriptors are used in calling procedures (see the *VMS RTL String Manipulation (STR$) Manual*).

EXAMPLE

```
DESCR1:  .ASCID  /ARGUMENT FOR CALL/      ; String descriptor
DESCR2:  .ASCID  /SECOND ARGUMENT/        ; Another string
                                          ;   descriptor
          .
          .
          .
         PUSHAL  DESCR1                   ; Put address of descriptors
         PUSHAL  DESCR2                   ;   on the stack
         CALLS   #2,STRNG_PROC            ; Call procedure
```

.ASCII

ASCII string storage directive

FORMAT

.ASCII *string*

PARAMETER

string
A delimited ASCII string.

DESCRIPTION

.ASCII stores the ASCII value of each character in the ASCII string or the value of each byte expression in the next available byte.

EXAMPLE

```
CR=13                                 ; Assignment statements
LF=10                                 ;   define CR and LF

       .ASCII   "DATE: 17-NOV-1988"   ; Delimiter is "
       .ASCII   /EOF/<CR><LF>       ; Delimiter is /
```

.ASCIZ

Zero-terminated ASCII string storage directive

FORMAT

.ASCIZ *string*

PARAMETER

string
A delimited ASCII string.

DESCRIPTION

.ASCIZ performs the same function as .ASCII, except that .ASCIZ appends a null byte as the final character of the string. When a list or text string is created with an .ASCIZ directive, you need only perform a search for the null character in the last byte to determine the end of the string.

EXAMPLE

```
FF=12                               ; Define FF
        .ASCIZ  /ABCDEF/            ; 6 characters in string,
                                    ;   7 bytes of data
        .ASCIZ  /A/<KEY>(FF\TEXT)/B/        ; 3 characters in strings,
                                    ;   4 bytes of data
```

.BLK*x*

Block storage allocation directives

FORMAT

.BLKA *expression*
.BLKB *expression*
.BLKD *expression*
.BLKF *expression*
.BLKG *expression*
.BLKH *expression*
.BLKL *expression*
.BLKO *expression*
.BLKQ *expression*
.BLKW *expression*

PARAMETER

expression
An expression specifying the amount of storage to be allocated. All the symbols in the expression must be defined and the expression must be an absolute expression (see Section 3.5). If the expression is omitted, a default value of 1 is assumed.

DESCRIPTION

VAX MACRO has 10 block storage directives.

Directive	Function
.BLKA	Reserves storage for addresses (longwords)
.BLKB	Reserves storage for byte data
.BLKD	Reserves storage for double-precision floating-point data (quadwords)
.BLKF	Reserves storage for single-precision floating-point data (longwords)
.BLKG	Reserves storage for G_floating data (quadwords)
.BLKH	Reserves storage for H_floating data (octawords)
.BLKL	Reserves storage for longword data
.BLKO	Reserves storage for octaword data
.BLKQ	Reserves storage for quadword data
.BLKW	Reserves storage for word data

Each directive reserves storage for a different data type. The value of the expression determines the number of data items for which VAX MACRO reserves storage. For example, .BLKL 4 reserves storage for four longwords of data and .BLKB 2 reserves storage for two bytes of data.

The total number of bytes reserved is equal to the length of the data type times the value of the expression as follows:

Directive	Number of Bytes Allocated
.BLKB	Value of expression
.BLKW	2 * value of expression
.BLKA	
.BLKF	4 * value of expression
.BLKL	
.BLKD	8 * value of expression
.BLKG	
.BLKQ	
.BLKH	16 * value of expression
.BLKO	

EXAMPLE

```
.BLKB   15          ; Space for 15 bytes
.BLKO   3           ; Space for 3 octawords (48 bytes)
.BLKL   1           ; Space for 1 longword (4 bytes)
.BLKF   <3*4>       ; Space for 12 single-precision
                    ;   floating-point values (48 bytes)
```

.BYTE

Byte storage directive

FORMAT

.BYTE *expression-list*

PARAMETER

expression-list
One or more expressions separated by commas. Each expression is first evaluated as a longword expression; then the value of the expression is truncated to one byte. The value of each expression should be in the range of 0 through 255 for unsigned data or in the range of –128 through +127 for signed data.

Optionally, each expression can be followed by a repetition factor delimited by square brackets. An expression followed by a repetition factor has the format:

expression1[expression2]

expression1
An expression that specifies the value to be stored.

[expression2]
An expression that specifies the number of times the value will be repeated. The expression must not contain any undefined symbols and it must be absolute (see Section 3.5). The square brackets are required.

DESCRIPTION

.BYTE generates successive bytes of binary data in the object module.

Notes

1 The assembler displays an error message if the high-order three bytes of the longword expression have a value other than 0 or ^XFFFFFF.

2 At link time, a relocatable expression can result in a value that exceeds one byte in length. In this case, the VAX linker issues a truncation diagnostic message for the object module in question. For example:

```
A:      .BYTE   A       ; Relocatable value 'A' will
                        ;   cause VAX linker truncation
                        ;   diagnostic if the statement
                        ;   has a virtual address of 256
                        ;   or above
```

3 The .SIGNED_BYTE directive is the same as .BYTE except that the assembler displays a diagnostic message if a value in the range 128 through 255 is specified. See the description of .SIGNED_BYTE for more information.

EXAMPLE

```
.BYTE   <1024-1000>*2            ; Stores a value of 48
.BYTE   ^XA,FIF,10,65-<21*3>     ; Stores 4 bytes of data
.BYTE   0                        ; Stores 1 byte of data
.BYTE   X,X+3[5*4],Z             ; Stores 22 bytes of data
```

.CROSS
.NOCROSS

Cross-reference directives

FORMAT

.CROSS *[symbol-list]*
.NOCROSS *[symbol-list]*

PARAMETER

symbol-list

A list of legal symbol names separated by commas.

DESCRIPTION

When you specify the /CROSS_REFERENCE qualifier in the MACRO command, VAX MACRO produces a cross-reference listing. The .CROSS and .NOCROSS directives control which symbols are included in the cross-reference listing. The .CROSS and .NOCROSS directives have an effect only if /CROSS_REFERENCE was specified in the MACRO command (see the *VMS DCL Dictionary*).

By default, the cross-reference listing includes the definition and all the references to every symbol in the module.

You can disable the cross-reference listing for all symbols or for a specified list of symbols by using .NOCROSS. Using .NOCROSS without a symbol list disables the cross-reference listing of all symbols. Any symbol definition or reference that appears in the code after .NOCROSS used without a symbol list and before the next .CROSS used without a symbol list is excluded from the cross-reference listing. You reenable the cross-reference listing by using .CROSS without a symbol list.

.NOCROSS with a symbol list disables the cross-reference listing for the listed symbols only. .CROSS with a symbol list enables or reenables the cross-reference listing of the listed symbols.

Notes

1. .CROSS without a symbol list will not reenable the cross-reference listing of a symbol specified in .NOCROSS with a symbol list.
2. If the cross-reference listing of all symbols is disabled, .CROSS with a symbol list will have no effect until the cross-reference listing is reenabled by .CROSS without a symbol list.

EXAMPLES

1
```
        .NOCROSS                ; Stop cross-reference
LAB1:   MOVL      LOC1, LOC2    ; Copy data
        .CROSS                  ; Reenable cross-reference
```

In this example, the definition of LAB1 and the references to LOC1 and LOC2 are not included in the cross-reference listing.

2
```
        .NOCROSS  LOC1          ; Do not cross-reference LOC1
LAB2:   MOVL      LOC1,LOC2     ; Copy data
        .CROSS    LOC1          ; Reenable cross-reference
                                ;   of LOC1
```

In this example, the definition of LAB2 and the reference to LOC2 are included in the cross-reference, but the reference to LOC1 is not included in the cross-reference.

.DEBUG

Debug symbol attribute directive

FORMAT

.DEBUG *symbol-list*

PARAMETER

symbol-list
A list of legal symbols separated by commas.

DESCRIPTION

.DEBUG specifies that the symbols in the list are made known to the VAX Symbolic Debugger. During an interactive debugging session, you can use these symbols to refer to memory locations or to examine the values assigned to the symbols.

Note

The assembler adds the symbols in the symbol list to the symbol table in the object module. You need not specify global symbols in the .DEBUG directive because global symbols are automatically put in the object module's symbol table. (See the description of .ENABLE for a discussion of how to make information about local symbols available to the debugger.)

EXAMPLE

```
.DEBUG  INPUT,OUTPUT,-     ; Make these symbols known
        LAB_30,LAB_40      ;   to the debugger
```

.DEFAULT

Default control directive

FORMAT

.DEFAULT *DISPLACEMENT, keyword*

PARAMETER

keyword
One of three keywords—BYTE, WORD, or LONG—indicating the default displacement length.

DESCRIPTION

.DEFAULT determines the default displacement length for the relative and relative deferred addressing modes (see Sections 5.2.1 and 5.2.2).

Notes

1. .DEFAULT has no effect on the default displacement for displacement and displacement deferred addressing modes (see Sections 5.1.6 and 5.1.7).
2. If there is no .DEFAULT in a source module, the default displacement length for the relative and relative deferred addressing modes is a longword.

EXAMPLE

```
.DEFAULT  DISPLACEMENT,WORD    ; WORD is default
MOVL      LABEL,R1             ; Assembler uses word
                               ;   displacement unless
                               ;   label has been defined
.DEFAULT  DISPLACEMENT,LONG    ; LONG is default
INCB    @COUNTS+4              ; Assembler uses longword
                               ;   displacement unless
                               ;   COUNTS has been defined
```

.D_FLOATING
.DOUBLE

Floating-point storage directive

FORMAT

.D_FLOATING *literal-list*
.DOUBLE *literal-list*

PARAMETER

literal-list
A list of floating-point constants (see Section 3.2.2). The constants cannot contain any unary or binary operators except unary plus or unary minus.

DESCRIPTION

.D_FLOATING evaluates the specified floating-point constants and stores the results in the object module. .D_FLOATING generates 64-bit, double-precision, floating-point data (1 bit of sign, 8 bits of exponent, and 55 bits of fraction). See the description of .F_FLOATING for information on storing single-precision floating-point numbers and the descriptions of .G_FLOATING and .H_FLOATING for descriptions of other floating-point numbers.

Notes

1. Double-precision floating-point numbers are always rounded. They are not affected by .ENABLE TRUNCATION.
2. The floating-point constants in the literal list must not be preceded by the floating-point operator (^F).

EXAMPLE

```
.D_FLOATING  1000,1.0E3,1.0000000E-9   ; Constant
.DOUBLE      3.1415928, 1.107153423828 ; List
.D_FLOATING  5, 10, 15, 0, 0.5
```

.DISABLE

Function control directive

FORMAT

.DISABLE *argument-list*

PARAMETER

argument-list
One or more of the symbolic arguments listed in Table 6–3 in the description of .ENABLE. You can use either the long or the short form of the symbolic arguments. If you specify multiple arguments, separate them by commas, spaces, or tabs.

DESCRIPTION

.DISABLE disables the specified assembler functions. See the description of .ENABLE for more information.

Note

The alternate form of .DISABLE is .DSABL.

.ENABLE

Function control directive

FORMAT

.ENABLE *argument-list*

PARAMETER

argument-list

One or more of the symbolic arguments listed in Table 6–3. You can use either the long form or the short form of the symbolic arguments.

If you specify multiple arguments, separate them with commas, spaces, or tabs.

Table 6–3 .ENABLE and .DISABLE Symbolic Arguments

Long Form	Short Form	Default Condition	Function
ABSOLUTE	AMA	Disabled	When ABSOLUTE is enabled, all the PC relative addressing modes are assembled as absolute addressing modes.
DEBUG	DBG	Disabled	When DEBUG is enabled, all local symbols are included in the object module's symbol table for use by the debugger.
GLOBAL	GBL	Enabled	When GLOBAL is enabled, all undefined symbols are considered external symbols. When GLOBAL is disabled, any undefined symbol that is not listed in an .EXTERNAL directive causes an assembly error.
LOCAL_BLOCK	LSB	Disabled	When LOCAL_BLOCK is enabled, the current local label block is ended and a new one is started. When LOCAL_BLOCK is disabled, the current local label block is ended. See Section 3.4 for a complete description of local label blocks.

Table 6–3 (Cont.) .ENABLE and .DISABLE Symbolic Arguments

Long Form	Short Form	Default Condition	Function
SUPPRESSION	SUP	Disabled	When SUPPRESSION is enabled, all symbols that are defined but not referred to are not listed in the symbol table. When SUPPRESSION is disabled, all symbols that are defined are listed in the symbol table.
TRACEBACK	TBK	Enabled	When TRACEBACK is enabled, the program section names and lengths, module names, and routine names are included in the object module for use by the debugger. When TRACEBACK is disabled, VAX MACRO excludes this information and, in addition, does not make any local symbol information available to the debugger.
TRUNCATION	FPT	Disabled	When TRUNCATION is enabled, single-precision floating-point numbers are truncated. When TRUNCATION is disabled, single-precision floating-point numbers are rounded. D_floating, G_floating, and H_floating numbers are not affected by .ENABLE TRUNCATION; they are always rounded.

DESCRIPTION

.ENABLE enables the specified assembly function. .ENABLE and its negative form, .DISABLE, control the following assembler functions:

- Creating local label blocks
- Making all local symbols available to the debugger and enabling the traceback feature
- Specifying that undefined symbol references are external references
- Truncating or rounding single-precision floating-point numbers
- Suppressing the listing of symbols that are defined but not referenced
- Specifying that all the PC references are absolute, not relative

Note

The alternate form of .ENABLE is .ENABL.

EXAMPLE

```
.ENABLE ABSOLUTE, GLOBAL        ; Assemble relative address mode
                                ;   as absolute address mode, and consider
                                ;   undefined references as global

.DISABLE TRUNCATION,TRACEBACK   ; Round floating-point numbers, and
                                ;   omit debugging information from
                                ;   the object module
```

.END

Assembly termination directive

FORMAT

.END *[symbol]*

PARAMETER

symbol

The address (called the transfer address) at which program execution is to begin.

DESCRIPTION

.END terminates the source program. No additional text should occur beyond this point in the current source file or in any additional source files specified in the command line for this assembly. If any additional text does occur, the assembler ignores it. The additional text does not appear in either the listing file or the object file.

Notes

1. The transfer address must be in a program section that has the EXE attribute (see the description of .PSECT).
2. When an executable image consisting of several object modules is linked, only one object module should be terminated by an .END directive that specifies a transfer address. All other object modules should be terminated by .END directives that do not specify a transfer address. If an executable image contains either no transfer address or more than one transfer address, the VAX linker displays an error message.
3. If the source program contains an unterminated conditional code block when the .END directive is specified, the assembler displays an error message.

EXAMPLE

```
.ENTRY  START,0              ; Entry mask
   .
   .                         ; Main program
   .
.END    START                ; Transfer address
```

.ENDC

End conditional directive

FORMAT

.ENDC

DESCRIPTION

.ENDC terminates the conditional range started by the .IF directive. See the description of .IF for more information and examples.

.ENDM

End definition directive

FORMAT

.ENDM *[macro-name]*

PARAMETERS

macro-name

The name of the macro whose definition is to be terminated. The macro name is optional; if specified, it must match the name defined in the matching .MACRO directive. The macro name should be specified so that the assembler can detect any improperly nested macro definitions.

DESCRIPTION

.ENDM terminates the macro definition. See the description of .MACRO for an example of the use of .ENDM.

Note

If .ENDM is encountered outside a macro definition, the assembler displays an error message.

.ENDR

End range directive

FORMAT

.ENDR

DESCRIPTION

.ENDR indicates the end of a repeat range. It must be the final statement of every indefinite repeat block directive (.IRP and .IRPC) and every repeat block directive (.REPEAT). See the description of these directives for examples of the use of .ENDR.

.ENTRY

Entry directive

FORMAT

.ENTRY *symbol,expression*

PARAMETERS

symbol

The symbolic name for the entry point.

expression

The register save mask for the entry point. The expression must be an absolute expression and must not contain any undefined symbols.

DESCRIPTION

.ENTRY defines a symbolic name for an entry point and stores a register save mask (two bytes) at that location. The symbol is defined as a global symbol with a value equal to the value of the location counter at the .ENTRY directive. You can use the entry point as the transfer address of the program. Use the register save mask to determine which registers are saved before the procedure is called. These saved registers are automatically restored when the procedure returns control to the calling program. See the description of the procedure call instructions in Chapter 9.

Notes

1. The register mask operator (^M) is convenient to use for setting the bits in the register save mask (see Section 3.6.2.2).
2. An assembly error occurs if the expression has bits 0, 1, 12, or 13 set. These bits correspond to the registers R0, R1, AP, and FP and are reserved for the CALL interface.
3. DIGITAL recommends that you use .ENTRY to define all callable entry points including the transfer address of the program. Although the following construct also defines an entry point, DIGITAL discourages its use:

   ```
   symbol:: .WORD    expression
   ```

   Although your program can call a procedure starting with this construct, the entry mask is not checked for any illegal registers, and the symbol cannot be used in a .MASK directive.
4. You should use .ENTRY only for procedures that are called by the CALLS or CALLG instruction. A routine that is entered by the BSB or JSB instruction should not use .ENTRY because these instructions do not expect a register save mask. Begin these routines using the following format:

   ```
   symbol:: first instruction
   ```

   The first instruction of the routine immediately follows the symbol.

EXAMPLE

```
.ENTRY  CALC,^M<R2,R3,R7>       ; Procedure starts here.
                                ; Registers R2, R3, and R7
                                ;   are preserved by CALL
                                ;   and RET instructions
```

.ERROR

Error directive

FORMAT

.ERROR *[expression] ;comment*

PARAMETERS

expression

An expression whose value is displayed when .ERROR is encountered during assembly.

;comment

A comment that is displayed when .ERROR is encountered during assembly. The comment must be preceded by a semicolon.

DESCRIPTION

.ERROR causes the assembler to display an error message on the terminal or batch log file and in the listing file (if there is one).

Notes

1. .ERROR, .WARN, and .PRINT are message display directives. Use them to display information indicating that a macro call contains an error or an illegal set of conditions.
2. When the assembly is finished, the assembler displays the total number of errors, warnings, information messages, and the sequence numbers of the lines causing the errors or warnings.
3. If .ERROR is included in a macro library, end the comment with a semicolon. Otherwise, the librarian will strip the comment from the directive and it will not be displayed when the macro is called.
4. The line containing the .ERROR directive is not included in the listing file.
5. If the expression has a value of 0, it is not displayed in the error message.

EXAMPLE

```
.IF DEFINED     LONG_MESS
.IF GREATER     1000-WORK_AREA
.ERROR 25                       ; Need larger WORK_AREA;
.ENDC
.ENDC
```

In this example, if the symbol LONG_MESS is defined and if the symbol WORK_AREA has a value of 1000 or less, the following error message is displayed:

```
%MACRO-E-GENERR, Generated ERROR: 25 Need larger WORK_AREA
```

.EVEN

Even location counter alignment directive

FORMAT

.EVEN

DESCRIPTION

.EVEN ensures that the current value of the location counter is even by adding 1 if the current value is odd. If the current value is already even, no action is taken.

.EXTERNAL

External symbol attribute directive

FORMAT

.EXTERNAL *symbol-list*

PARAMETER

symbol-list
A list of legal symbols, separated by commas.

DESCRIPTION

.EXTERNAL indicates that the specified symbols are external; that is, the symbols are defined in another object module and cannot be defined until link time (see Section 3.3.3 for a discussion of external references).

Notes

1 If the GLOBAL argument is enabled (see Table 6–3), all unresolved references will be marked as global and external. If GLOBAL is enabled, you need not specify .EXTERNAL. If GLOBAL is disabled, you must explicitly specify .EXTERNAL to declare any symbols that are defined externally but are referred to in the current module.

2 If GLOBAL is disabled and the assembler finds symbols that are neither defined in the current module nor listed in a .EXTERNAL directive, the assembler displays an error message.

3 Note that if your program does not reference, in a relocatable program section, symbols that are declared in the absolute program section (ABS), the unreferenced symbols are filtered out by the assembler and will not be included in the object file. This filtering out will occur even if the symbols are declared global or external.

If you want to be sure that a symbol will be included even if it is not referenced, declare it in a relocatable program section. If you want to make sure that a symbol you define in an absolute program section is included, reference it in a relocatable program section.

4 The alternate form of .EXTERNAL is .EXTRN.

EXAMPLE

```
.EXTERNAL      SIN,TAN,COS      ; These symbols are defined in
.EXTERNAL      SINH,COSH,TANH   ;   externally assembled modules
```

.F_FLOATING
.FLOAT

Floating-point storage directive

FORMAT

.F_FLOATING *literal-list*
.FLOAT *literal-list*

PARAMETER

literal-list
A list of floating-point constants (see Section 3.2.2). The constants cannot contain any unary or binary operators except unary plus and unary minus.

DESCRIPTION

.F_FLOATING evaluates the specified floating-point constants and stores the results in the object module. .F_FLOATING generates 32-bit, single-precision, floating-point data (1 bit of sign, 8 bits of exponent, and 23 bits of fractional significance). See the description of .D_FLOATING for information on storing double-precision floating-point numbers and the descriptions of .G_FLOATING and .H_FLOATING for descriptions of other floating-point numbers.

Notes

1. See the description of .ENABLE for information on specifying floating-point rounding or truncation.
2. The floating-point constants in the literal list must not be preceded by the floating-point unary operator (^F).

EXAMPLE

```
.F_FLOATING  134.5782,74218.34E20     ; Constant list
.F_FLOATING  134.2,0.1342E3,1342E-1   ; These all generate 134.2
.F_FLOATING  -0.75,1E38,-1.0E-37      ; Constant list
.FLOAT       0,25,50
```

.G_FLOATING

G_floating-point storage directive

FORMAT

.G_FLOATING *literal-list*

PARAMETERS

literal-list
A list of floating-point constants (see Section 3.2.2). The constants cannot contain any unary or binary operators except unary plus or unary minus.

DESCRIPTION

.G_FLOATING evaluates the specified floating-point constants and stores the results in the object module. .G_FLOATING generates 64-bit data (1 bit of sign, 11 bits of exponent, and 52 bits of fraction).

Notes

1. G_floating-point numbers are always rounded. They are not affected by the .ENABLE TRUNCATION directive.
2. The floating-point constants in the literal list must not be preceded by the floating-point operator (^F).

EXAMPLE

```
.G_FLOATING     1000,  1.0E3,  1.0000000E-9     ; Constant list
```

.GLOBAL

Global symbol attribute directive

FORMAT

.GLOBAL *symbol-list*

PARAMETER

symbol-list
A list of legal symbol names, separated by commas.

DESCRIPTION

.GLOBAL indicates that specified symbol names are either globally defined in the current module or externally defined in another module (see Section 3.3.3).

Notes

1. .GLOBAL is provided for MACRO–11 compatibility only. DIGITAL recommends that global definitions be specified by a double colon or double equals sign (see Sections 2.1 and 3.8) and that external references be specified by .EXTERNAL when necessary.
2. The alternate form of .GLOBAL is .GLOBL.

EXAMPLE

```
.GLOBAL LAB_40,LAB_30           ; Make these symbol names
                                ;   globally known
.GLOBAL UKN_13                  ;   to all linked modules
```

.H_FLOATING

H_floating-point storage directive

FORMAT

.H_FLOATING *literal-list*

PARAMETER

literal-list

A list of floating-point constants (see Section 3.2.2). The constants cannot contain any unary or binary operators except unary plus or unary minus.

DESCRIPTION

.H_FLOATING evaluates the specified floating-point constants and stores the results in the object module. .H_FLOATING generates 128-bit data (1 bit of sign, 15 bits of exponent, and 112 bits of fraction).

Notes

1. H_floating-point numbers are always rounded. They are not affected by the .ENABLE TRUNCATION directive.
2. The floating-point constants in the literal list must not be preceded by the floating-point operator (^F).

EXAMPLE

```
.H_FLOATING     36912,  15.0E18,  1.0000000E-9     ; Constant list
```

.IDENT

Identification directive

FORMAT

.IDENT *string*

PARAMETER

string

A 1- to 31-character string that identifies the module, such as a string that specifies a version number. The string must be delimited. The delimiters can be any paired printing characters other than the left angle bracket (<) or the semicolon (;), as long as the delimiting character is not contained within the text string.

DESCRIPTION

.IDENT provides a means of identifying the object module. This identification is in addition to the name assigned to the object module with .TITLE. A character string can be specified in .IDENT to label the object module. This string is printed in the header of the listing file and also appears in the object module.

Notes

1 If a source module contains more than one .IDENT, the last directive given establishes the character string that forms part of the object module identification.

2 If the delimiting characters do not match, or if you use an illegal delimiting character, the assembler displays an error message.

EXAMPLE

```
.IDENT  /3-47/                  ; Version and edit numbers
```

The character string "3-47" is included in the object module.

.IF

Conditional assembly block directives

FORMAT

.IF *condition argument(s)*

.
.
.

range

.
.
.

.ENDC

PARAMETERS

condition
A specified condition that must be met if the block is to be included in the assembly. The condition must be separated from the argument by a comma, space, or tab. Table 6–4 lists the conditions that can be tested by the conditional assembly directives.

argument(s)
One or more symbolic arguments or expressions of the specified conditional test. If the argument is an expression, it cannot contain any undefined symbols and must be an absolute expression (see Section 3.5).

range
The block of source code that is conditionally included in the assembly.

Table 6–4 Condition Tests for Conditional Assembly Directives

Condition Test Long Form	Short Form	Complement Condition Test Long Form	Short Form	Argument Type	Number of Arguments	Condition that Assembles Block
EQUAL	EQ	NOT_EQUAL	NE	Expression	1	Expression is equal to 0/not equal to 0.
GREATER	GT	LESS_EQUAL	LE	Expression	1	Expression is greater than 0/ less than or equal to 0.
LESS_THAN	LT	GREATER_EQUAL	GE	Expression	1	Expression is less than 0/greater than or equal to 0.
DEFINED	DF	NOT_DEFINED	NDF	Symbolic	1	Symbol is defined /not defined.
BLANK[1]	B	NOT_BLANK[1]	NB	Macro	1	Argument is blank / nonblank.
IDENTICAL[1]	IDN	DIFFERENT[1]	DIF	Macro	2	Arguments are identical/different.

[1]The BLANK, NOT_BLANK, IDENTICAL, and DIFFERENT conditions are only useful in macro definitions.

DESCRIPTION

A conditional assembly block is a series of source statements that is assembled only if a certain condition is met. .IF starts the conditional block and .ENDC ends the conditional block; each .IF must have a corresponding .ENDC. The .IF directive contains a condition test and one or two arguments. The condition test specified is applied to the argument(s). If the test is met, all VAX MACRO statements between .IF and .ENDC are assembled. If the test is not met, the statements are not assembled. An exception to this rule occurs when you use subconditional directives (see the description of the .IF_x directive).

Conditional blocks can be nested; that is, a conditional block can be inside another conditional block. In this case the statements in the inner conditional block are assembled only if the condition is met for both the outer and inner block.

Notes

1 If .ENDC occurs outside a conditional assembly block, the assembler displays an error message.

2 VAX MACRO permits a nesting depth of 31 conditional assembly levels. If a statement attempts to exceed this nesting level depth, the assembler displays an error message.

3 Lowercase string arguments are converted to uppercase before being compared, unless the string is surrounded by delimiters. For information on string arguments and delimiters, see Chapter 4.

4 The assembler displays an error message if .IF specifies any of the following: a condition test other than those in Table 6–4, an illegal argument, or a null argument specified in an .IF directive.

5 The .SHOW and .NOSHOW directives control whether condition blocks that are not assembled are included in the listing file.

EXAMPLES

1 An example of a conditional assembly directive is:

```
.IF EQUAL  ALPHA+1        ; Assemble block if ALPHA+1=0. Do
  .                       ;   not assemble if ALPHA+1 not=0
  .
  .
.ENDC
```

2 Nested conditional directives take the form:

```
.IF   condition,argument(s)
.IF   condition,argument(s)
  .
  .
  .
.ENDC
.ENDC
```

3 The following conditional directives can govern whether assembly is to occur:

```
.IF DEFINED  SYM1
.IF DEFINED  SYM2
  .
  .
  .
.ENDC
.ENDC
```

In this example, if the outermost condition is not satisfied, no deeper level of evaluation of nested conditional statements within the program occurs. Therefore, both SYM1 and SYM2 must be defined for the code to be assembled.

.IF_x

Subconditional assembly block directives

FORMAT

.IF_FALSE
.IF_TRUE
.IF_TRUE_FALSE

DESCRIPTION

VAX MACRO has the following three subconditional assembly block directives:

Directive	Function
.IF_FALSE	If the condition of the assembly block tests false, the program includes the source code following the .IF_FALSE directive and continuing up to the next subconditional directive or to the end of the conditional assembly block.
.IF_TRUE	If the condition of the assembly block tests true, the program includes the source code following the .IF_TRUE directive and continuing up to the next subconditional directive or to the end of the conditional assembly block.
.IF_TRUE_FALSE	Regardless of whether the condition of the assembly block tests true or false, the source code following the .IF TRUE_FALSE directive (and continuing up to the next subconditional directive or to the end of the assembly block) is always included.

The implied argument of a subconditional directive is the condition test specified when the conditional assembly block was entered. A conditional or subconditional directive in a nested conditional assembly block is not evaluated if the preceding (or outer) condition in the block is not satisfied (see Examples 3 and 4).

A conditional block with a subconditional directive is different from a nested conditional block. If the condition in the .IF is not met, the inner conditional block(s) are not assembled, but a subconditional directive can cause a block to be assembled.

Notes

1 If a subconditional directive appears outside a conditional assembly block, the assembler displays an error message.

2 The alternate forms of .IF_FALSE, .IF_TRUE, and .IF_TRUE_FALSE are .IFF, .IFT, and .IFTF.

EXAMPLES

1 Assume that symbol SYM is defined:

```
.IF DEFINED   SYM           ; Tests TRUE since SYM is defined.
  .                         ;   Assembles the following code.
  .
  .
.IF_FALSE                   ; Tests FALSE since previous
  .                         ;   .IF was TRUE.  Does not
  .                         ;   assemble the following code.
  .
.IF_TRUE                    ; Tests TRUE since SYM is defined.
  .                         ;   Assembles the following code.
  .
  .
.IF_TRUE_FALSE              ; Assembles following code
  .                         ;   unconditionally.
  .
  .
.IF_TRUE                    ; Tests TRUE since SYM is defined.
  .                         ;   Assembles remainder of
  .                         ;   conditional assembly block.
  .
.ENDC
```

2 Assume that symbol X is defined and that symbol Y is not defined:

```
.IF DEFINED  X              ; Tests TRUE since X is defined.
.IF DEFINED  Y              ; Tests FALSE since Y is not defined.
.IF_FALSE                   ; Tests TRUE since Y is not defined.
  .                         ;   Assembles the following code.
  .
  .
.IF_TRUE                    ; Tests FALSE since Y is not defined.
  .                         ;   Does not assemble the following
  .                         ;   code.
  .
.ENDC
.ENDC
```

3 Assume that symbol A is defined and that symbol B is not defined:

```
.IF DEFINED  A              ; Tests TRUE since A is defined.
  .                         ;   Assembles the following code.
  .
  .
.IF_FALSE                   ; Tests FALSE since A is defined.
  .                         ;   Does not assemble the following
  .                         ;   code.
  .
.IF NOT_DEFINED B           ; Nested conditional directive
  .                         ;   is not evaluated.
  .
  .
.ENDC
.ENDC
```

4 Assume that symbol X is not defined but symbol Y is defined:

```
.IF DEFINED  X              ; Tests FALSE since X is not defined.
  .                         ;   Does not assemble the following
  .                         ;   code.
  .
.IF DEFINED  Y              ; Nested conditional directive
  .                         ;   is not evaluated.
  .
  .
.IF_FALSE                   ; Nested subconditional
  .                         ;   directive is not evaluated.
  .
  .
.IF_TRUE                    ; Nested subconditional
  .                         ;   directive is not evaluated.
  .
  .
.ENDC
.ENDC
```

.IIF

Immediate conditional assembly block directive

FORMAT

.IIF *condition [,]argument(s), statement*

PARAMETERS

condition

One of the legal condition tests defined for conditional assembly blocks in Table 6–4 (see the description of .IF). The condition must be separated from the arguments by a comma, space, or tab. If the first argument can be a blank, the condition must be separated from the arguments with a comma.

argument(s)

An expression or symbolic argument (described in Table 6–4) associated with the immediate conditional assembly block directive. If the argument is an expression, it cannot contain any undefined symbols and must be an absolute expression (see Section 3.5). The arguments must be separated from the statement by a comma.

statement

The statement to be assembled if the condition is satisfied.

DESCRIPTION

.IIF provides a means of writing a one-line conditional assembly block. The condition to be tested and the conditional assembly block are expressed completely within the line containing the .IIF directive. No terminating .ENDC statement is required.

Note

The assembler displays an error message if .IIF specifies a condition test other than those listed in Table 6–4, an illegal argument, or a null argument.

EXAMPLE

```
.IIF DEFINED EXAM, BEQL ALPHA
```

This directive generates the following code if the symbol EXAM is defined within the source program:

```
BEQL    ALPHA
```

.IRP

Indefinite repeat argument directive

FORMAT

.IRP *symbol, <argument list>*

.
.
.

range

.
.
.

.ENDR

PARAMETERS

symbol

A formal argument that is successively replaced with the specified actual arguments enclosed in angle brackets. If no formal argument is specified, the assembler displays an error message.

<argument list>

A list of actual arguments enclosed in angle brackets and used in expanding the indefinite repeat range. An actual argument can consist of one or more characters. Multiple arguments must be separated by a legal separator (comma, space, or tab). If no actual arguments are specified, no action is taken.

range

The block of source text to be repeated once for each occurrence of an actual argument in the list. The range can contain macro definitions and repeat ranges. .MEXIT is legal within the range.

DESCRIPTION

.IRP replaces a formal argument with successive actual arguments specified in an argument list. This replacement process occurs during the expansion of the indefinite repeat block range. The .ENDR directive specifies the end of the range.

.IRP is analogous to a macro definition with only one formal argument. At each expansion of the repeat block, this formal argument is replaced with successive elements from the argument list. The directive and its range are coded in line within the source program. This type of macro definition and its range do not require calling the macro by name, as do other macros described in this section.

.IRP can appear either inside or outside another macro definition, indefinite repeat block, or repeat block (see the description of .REPEAT). The rules for specifying .IRP arguments are the same as those for specifying macro arguments.

EXAMPLE

Macro definition:

```
.MACRO  CALL_SUB        SUBR,A1,A2,A3,A4,A5,A6,A7,A8,A9,A10
.NARG   COUNT
.IRP    ARG,<A10,A9,A8,A7,A6,A5,A4,A3,A2,A1>
.IIF    NOT_BLANK ,     ARG,    PUSHL ARG
.ENDR
CALLS   #<COUNT-1>,SUBR         ; Note SUBR is counted
.ENDM   CALL_SUB
```

Macro call and expansion of the macro defined previously:

```
CALL_SUB        TEST,INRES,INTES,UNLIS,OUTCON,#205
.NARG   COUNT
.IRP    ARG,<,,,,,#205,OUTCON,UNLIS,INTES,INRES>
.IIF    NOT_BLANK ,     ARG,      PUSHL ARG
.ENDR
.IIF    NOT_BLANK ,     ,         PUSHL
.IIF    NOT_BLANK ,     ,         PUSHL
.IIF    NOT_BLANK ,     ,         PUSHL
.IIF    NOT_BLANK ,     ,         PUSHL
.IIF    NOT_BLANK ,     ,         PUSHL
.IIF    NOT_BLANK ,     #205,     PUSHL #205
.IIF    NOT_BLANK ,     OUTCON,   PUSHL OUTCON
.IIF    NOT_BLANK ,     UNLIS,    PUSHL UNLIS
.IIF    NOT_BLANK ,     INTES,    PUSHL INTES
.IIF    NOT_BLANK ,     INRES,    PUSHL INRES
CALLS   #<COUNT-1>,TEST         ; Note TEST is counted
```

This example uses the .NARG directive to count the arguments and the .IIF NOT_BLANK directive (see descriptions of .IF and .IIF in this section) to determine whether the actual argument is blank. If the argument is blank, no binary code is generated.

.IRPC

Indefinite repeat character directive

FORMAT

.IRPC *symbol, <STRING>*

.
.
.

range

.
.
.

.ENDR

PARAMETERS

symbol

A formal argument that is successively replaced with the specified characters enclosed in angle brackets. If no formal argument is specified, the assembler displays an error message.

<STRING>

A sequence of characters enclosed in angle brackets and used in the expansion of the indefinite repeat range. Although the angle brackets are required only when the string contains separating characters, their use is recommended for legibility.

range

The block of source text to be repeated once for each occurrence of a character in the list. The range can contain macro definitions and repeat ranges. .MEXIT is legal within the range.

DESCRIPTION

.IRPC is similar to .IRP except that .IRPC permits single-character substitution rather than argument substitution. On each iteration of the indefinite repeat range, the formal argument is replaced with each successive character in the specified string. The .ENDR directive specifies the end of the range.

.IRPC is analogous to a macro definition with only one formal argument. At each expansion of the repeat block, this formal argument is replaced with successive characters from the actual argument string. The directive and its range are coded in line within the source program and do not require calling the macro by name.

.IRPC can appear either inside or outside another macro definition, indefinite repeat block, or repeat block (see description of .REPEAT).

EXAMPLE

Macro Definition:

```
      .MACRO  HASH_SYM        SYMBOL
      .NCHR   HV,<SYMBOL>
      .IRPC   CHR,<SYMBOL>
HV = HV+^A?CHR?
      .ENDR
      .ENDM   HASH_SYM
```

Macro call and expansion of the macro defined previously:

```
      HASH_SYM        <MOVC5>
      .NCHR   HV,<MOVC5>
      .IRPC   CHR,<MOVC5>
HV = HV+^A?CHR?
      .ENDR
HV = HV+^A?M?
HV = HV+^A?O?
HV = HV+^A?V?
HV = HV+^A?C?
HV = HV+^A?5?
```

This example uses the .NCHR directive to count the number of characters in an actual argument.

.LIBRARY

Macro library directive

FORMAT

.LIBRARY *macro-library-name*

PARAMETERS

macro-library-name
A delimited string that is the file specification of a macro library.

DESCRIPTION

.LIBRARY adds a name to the macro library list that is searched whenever a .MCALL or an undefined opcode is encountered. The libraries are searched in the reverse order in which they were specified to the assembler.

If you omit any information from the macro-library-name argument, default values are assumed. The device defaults to your current default disk; the directory defaults to your current default directory; the file type defaults to MLB.

DIGITAL recommends that libraries be specified in the MACRO command line with the /LIBRARY qualifier rather than with the .LIBRARY directive. The .LIBRARY directive makes moving files cumbersome.

EXAMPLE

```
.LIBRARY   /DISK:[TEST]USERM/     ; DISK:[TEST]USERM.MLB
.LIBRARY   ?DISK:SYSDEF.MLB?      ; DISK:SYSDEF.MLB
.LIBRARY   \CURRENT.MLB\          ; Uses default disk and directory
```

.LINK

Linker option record directive

FORMAT

.LINK *"file-spec" [/qualifier[=(module-name[,...])],...]*

PARAMETERS

file-spec[,...]

A delimited string that specifies one or more input files. The delimiters can be any matching pair of printable characters except the space, tab, equal sign (=), semicolon (;), or left angle bracket (<). The character that you use as the delimiter cannot appear in the string itself. Although you can use alphanumeric characters as delimiters, use nonalphanumeric characters to avoid confusion.

The input files can be object modules to be linked, or shareable images to be included in the output image. Input files can also be libraries containing external references or specific modules for inclusion in the output image. The linker will search the libraries for the external references. If you specify multiple input files, separate the file specifications with commas (,).

If you do not specify a file type in an input file specification, the linker supplies default file types, based on the nature of the file. All object modules are assumed to have file types of OBJ.

Note that the input file specifications must be correct at *link* time. Make your references explicit, so that if the object module created by VAX MACRO is linked in a directory other than the one in which it was created, the linker will still be able to find the files referenced in the .LINK directive.

No wildcard characters are allowed in the file specification.

FILE QUALIFIERS

/INCLUDE=(module-name[,...])

Indicates that the associated input file is an object library or shareable image library, and that only the module names specified are to be unconditionally included as input to the linker.

At least one module name must be specified. If you specify more than one module name, separate the names with commas and enclose the list in parentheses.

No wildcard characters are allowed in the module name specifications. Module names may not be longer than 31 characters, the maximum length of a VAX MACRO symbol.

/LIBRARY

Indicates that the associated input file is a library to be searched for modules to resolve any undefined symbols in the input files.

If the associated input file specification does not include a file type, the linker assumes the default file type of OLB. You can use both /INCLUDE and /LIBRARY to qualify a file specification. If you specify both /INCLUDE and /LIBRARY, the library is subsequently searched for unresolved references.

In this case, the explicit inclusion of modules occurs first; then the linker searches the library for unresolved references.

/SELECTIVE_SEARCH

Directs the linker to add to its symbol table only those global symbols that are defined in the specified file and are currently unresolved. If /SELECTIVE_SEARCH is not specified, the linker includes all symbols from that file in its global symbol table.

/SHAREABLE

Requests that the linker include a shareable image file. No wildcard characters are allowed in the file specification.

The following table contains the abbreviations of the qualifiers for the .LINK directive. Note that to ensure readability, as well as compatibility with future releases, it is recommended that you use the full names of the qualifiers.

Abbreviation	Qualifier
/I	/INCLUDE
/L	/LIBRARY
/SE	/SELECTIVE_SEARCH
/SH	/SHAREABLE

DESCRIPTION

The .LINK directive allows you to include linker option records in an object module produced by VAX MACRO. The qualifiers for the .LINK directive perform functions similar to the functions performed by the same qualifiers for the DCL command LINK.

You should use the .LINK directive for references that are not linker defaults, but that you always want to include in a particular image. Using the .LINK directive enables you to avoid having to explicitly name these references in the DCL command LINK.

For detailed information on the qualifiers to the DCL command LINK, see the *VMS DCL Dictionary*. For a complete discussion of the operation of the linker itself, see the *VMS Linker Utility Manual*.

EXAMPLES

1

```
.LINK "SYS$LIBRARY:MYLIB" /INCLUDE=(MOD1, MOD2, MOD6)
```

This statement, when included in the file MYPROG.MAR, causes the assembler to request that MYPROG.OBJ be linked with modules MOD1, MOD2, and MOD6 in the library SYS$LIBRARY:MYLIB.OLB (where SYS$LIBRARY is a logical name for the disk and directory in which MYLIB.OLB is listed). The library is not searched for other unresolved references. The statement is equivalent to linking the file with the DCL command:

2

```
$ LINK MYPROG, SYS$LIBRARY:MYLIB /INCLUDE=(MOD1, MOD2, MOD6)
```

3
```
.LINK \SYS$LIBRARY:MYOBJ\                         ; Link with object module
                                                  ;   SYS$LIBRARY:MYOBJ.OBJ

.LINK 'SYS$LIBRARY:YOURLIB' /LIBRARY              ; Search object library
                                                  ;   SYS$LIBRARY:YOURLIB.OLB
                                                  ;   for unresolved references

.LINK *SYS$LIBRARY:MYSTB.STB* /SELECTIVE_SEARCH   ; Search symbol table
                                                  ;   SYS$LIBRARY:MYSTB.STB
                                                  ;   for unresolved references

.LINK "SYS$LIBRARY:MYSHR.EXE" /SHAREABLE          ; Link with shareable image
                                                  ;   SYS$LIBRARY:MYSHR.EXE
```

To increase efficiency and performance, include several related input files in a single .LINK directive. The following example shows how the five options illustrated previously can be included in one statement:

4
```
.LINK   'SYS$LIBRARY:MYOBJ',-
        'SYS$LIBRARY:YOURLIB' /LIBRARY,-
        'SYS$LIBRARY:MYLIB' /INCLUDE=(MOD1, MOD2, MOD6),-
        'SYS$LIBRARY:MYSTB.STB' /SELECTIVE_SEARCH,-
        'SYS$LIBRARY:MYSHR.EXE' /SHAREABLE
```

.LIST

Listing directive

FORMAT

.LIST *[argument-list]*

PARAMETER

argument-list

One or more of the symbolic arguments defined in Table 6–8 in the description of .SHOW. You can use either the long form or the short form of the arguments. If multiple arguments are specified, separate them with commas, spaces, or tabs.

DESCRIPTION

.LIST is equivalent to .SHOW. See the description of .SHOW for more information.

.LONG

Longword storage directive

FORMAT

.LONG *expression-list*

PARAMETERS

expression-list

One or more expressions separated by commas. You have the option of following each expression with a repetition factor delimited by square brackets.

An expression followed by a repetition factor has the format:

expression1[expression2]

expression1

An expression that specifies the value to be stored.

[expression2]

An expression that specifies the number of times the value is repeated. The expression must not contain any undefined symbols and must be an absolute expression (see Section 3.5). The square brackets are required.

DESCRIPTION

.LONG generates successive longwords (four bytes) of data in the object module.

EXAMPLE

```
LAB_3:  .LONG   LAB_3,^X7FFFFFFF,^A'ABCD' ; 3 longwords of data
        .LONG   ^XF@4                     ; 1 longword of data
        .LONG   0[22]                     ; 22 longwords of data
```

Note

Each expression in the list must have a value that can be represented in 32 bits.

.MACRO

Macro definition directive

FORMAT

.MACRO *macro-name [formal-argument-list]*

.
.
.

range

.
.
.

.ENDM *[macro name]*

PARAMETERS

macro-name
The name of the macro to be defined; this name can be any legal symbol up to 31 characters long.

formal-argument-list
The symbols, separated by commas, to be replaced by the actual arguments in the macro call.

range
The source text to be included in the macro expansion.

DESCRIPTION

.MACRO begins the definition of a macro. It gives the macro name and a list of formal arguments (see Chapter 4). If the name specified is the same as the name of a previously defined macro, the previous definition is deleted and replaced with the new one. The .MACRO directive is followed by the source text to be included in the macro expansion. The .ENDM directive specifies the end of the range.

Macro names do not conflict with user-defined symbols. Both a macro and a user-defined symbol can have the same name.

When the assembler encounters a .MACRO directive, it adds the macro name to its macro name table and stores the source text of the macro (up to the matching .ENDM directive). No other processing occurs until the macro is expanded.

The symbols in the formal argument list are associated with the macro name and are limited to the scope of the definition of that macro. For this reason, the symbols that appear in the formal argument list can also appear elsewhere in the program.

Notes

1. If a macro has the same name as a VAX opcode, the macro is used instead of the instruction. This feature allows you to temporarily redefine an opcode.

2. If a macro has the same name as a VAX opcode and is in a macro library, you must use the .MCALL directive to define the macro. Otherwise, because the symbol is already defined (as the opcode), the assembler will not search the macro libraries.

3. You can redefine a macro with new source text during assembly by specifying a second .MACRO directive with the same name. Including a second .MACRO directive within the original macro definition causes the first macro call to redefine the macro. This is useful when a macro performs initialization or defines symbols, when an operation is performed only once. The macro redefinition can eliminate unneeded source text in a macro or it can delete the entire macro. The .MDELETE directive provides another way to delete macros.

EXAMPLE

Macro definition:

```
        .MACRO  USERDEF
        .PSECT  DEFIES,ABS
MYSYM=  5
HIVAL=  ^XFFF123
LOWVAL= 0
        .PSECT  RWDATA,NOEXE,LONG
TABLE:  .BLKL   100
LIST:   .BLKB   10
        .MACRO  USERDEF                 ; Redefine it to null
        .ENDM   USERDEF
        .ENDM   USERDEF
```

Macro calls and expansions of the macro defined previously:

```
        USERDEF                         ; Should expand data
        .PSECT  DEFIES,ABS
MYSYM=  5
HIVAL=  ^XFFF123
LOWVAL= 0
        .PSECT  RWDATA,NOEXE,LONG
TABLE:  .BLKL   100
LIST:   .BLKB   10
        .MACRO  USERDEF                 ; Redefine it to null
        .ENDM   USERDEF

        USERDEF                         ; Should expand nothing
```

In this example, when the macro is called the first time, it defines some symbols and data storage areas and then redefines itself. When the macro is called a second time, the macro expansion contains no source text.

.MASK

Mask directive

FORMAT

.MASK *symbol[,expression]*

PARAMETERS

symbol

A symbol defined in an .ENTRY directive.

expression

A register save mask.

DESCRIPTION

.MASK reserves a word for a register save mask for a transfer vector. See the description of .TRANSFER for more information and for an example of .MASK.

Notes

1 If .MASK does not contain an expression, the assembler directs the linker to copy the register save mask specified in .ENTRY to the word reserved by .MASK.

2 If .MASK contains an expression, the assembler directs the linker to combine this expression with the register save mask specified in .ENTRY and store the result in the word reserved by .MASK. The linker performs an inclusive OR operation to combine the mask in the entry point and the value of the expression. Consequently, a register specified in either .ENTRY or .MASK will be included in the combined mask. See the description of .ENTRY for more information on entry masks.

.MCALL

Macro call directive

FORMAT

.MCALL *macro-name-list*

PARAMETERS

macro-name-list

A list of macros to be defined for this assembly. Separate the macro names with commas.

DESCRIPTION

.MCALL specifies the names of the system and user-defined macros that are required to assemble the source program but are not defined in the source file.

If any named macro is not found upon completion of the search (that is, if the macro is not defined in any of the macro libraries), the assembler displays an error message.

Note: **.MCALL is provided for compatibility with MACRO-11; with one exception, DIGITAL recommends that you not use it. When VAX MACRO finds an unknown symbol in the opcode field, it automatically searches all macro libraries. If it finds the symbol in a library, it uses the macro definition and expands the macro reference. If VAX MACRO does not find the symbol in the library, it displays an error message.**

You must use .MCALL when a macro has the same name as an opcode (see description of .MACRO).

EXAMPLE

```
.MCALL  INSQUE          ; Substitute macro in
                        ;   library for INSQUE
                        ;   instruction
```

.MDELETE

Macro deletion directive

FORMAT

.MDELETE *macro-name-list*

PARAMETERS

macro-name-list

A list of macros whose definitions are to be deleted. Separate the names with commas.

DESCRIPTION

.MDELETE deletes the definitions of specified macros. The number of macros actually deleted is printed in the assembly listing on the same line as the .MDELETE directive.

.MDELETE completely deletes the macro, freeing memory as necessary. Macro redefinition with .MACRO merely redefines the macro.

EXAMPLE

```
.MDELETE      USERDEF,$SSDEF,ALTR
```

.MEXIT

Macro exit directive

FORMAT

.MEXIT

DESCRIPTION

.MEXIT terminates a macro expansion before the end of the macro. Termination is the same as if .ENDM were encountered. You can use the directive within repeat blocks. .MEXIT is useful in conditional expansion of macros because it bypasses the complexities of nested conditional directives and alternate assembly paths.

Notes

1. When .MEXIT occurs in a repeat block, the assembler terminates the current repetition of the range and suppresses further expansion of the repeat range.
2. When macros or repeat blocks are nested, .MEXIT exits to the next higher level of expansion.
3. If .MEXIT occurs outside a macro definition or a repeat block, the assembler displays an error message.

EXAMPLE

```
.MACRO   POLO    N,A,B
   .
   .
   .
.IF EQ  N                     ; Start conditional assembly block
   .
   .
   .
.MEXIT                        ; Terminate macro expansion
.ENDC                         ; End conditional assembly block
   .
   .
   .
.ENDM   POLO                  ; Normal end of macro
```

In this example, if the actual argument for the formal argument N equals 0, the conditional block is assembled, and the macro expansion is terminated by .MEXIT.

.NARG

Number of arguments directive

FORMAT

.NARG *symbol*

PARAMETERS

symbol
A symbol that is assigned a value equal to the number of arguments in the macro call.

DESCRIPTION

.NARG determines the number of arguments in the current macro call.

.NARG counts all the positional arguments specified in the macro call, including null arguments (specified by adjacent commas). The value assigned to the specified symbol does not include either any keyword arguments or any formal arguments that have default values.

Note

If .NARG appears outside a macro, the assembler displays an error message.

EXAMPLE

Macro definition:

```
.MACRO  CNT_ARG A1,A2,A3,A4,A5,A6,A7,A8,A9=DEF9,A10=DEF10
.NARG   COUNTER         ; COUNTER is set to no. of ARGS
.WORD   COUNTER         ; Store value of COUNTER
.ENDM   CNT_ARG
```

Macro calls and expansions of the macro defined previously:

```
CNT_ARG TEST,FIND,ANS   ; COUNTER will = 3
.NARG   COUNTER         ; COUNTER is set to no. of ARGS
.WORD   COUNTER         ; Store value of COUNTER

CNT_ARG                 ; COUNTER will = 0
.NARG   COUNTER         ; COUNTER is set to no. of ARGS
.WORD   COUNTER         ; Store value of COUNTER

CNT_ARG TEST,A2=SYMB2,A3=SY3      ; COUNTER will = 1
.NARG   COUNTER         ; COUNTER is set to no. of ARGS
.WORD   COUNTER         ; Store value of COUNTER
                        ; Keyword arguments are not counted

CNT_ARG ,SYMBL,,        ; COUNTER will = 3
.NARG   COUNTER         ; COUNTER is set to no. of ARGS
.WORD   COUNTER         ; Store value of COUNTER
                        ; Null arguments are counted
```

.NCHR

Number of characters directive

FORMAT

.NCHR *symbol, <string>*

PARAMETERS

symbol

A symbol that is assigned a value equal to the number of characters in the specified character string.

<string>

A sequence of printable characters. Delimit the character string with angle brackets (or a character preceded by a circumflex) only if the specified character string contains a legal separator (comma, space, and/or tab) or a semicolon.

DESCRIPTION

.NCHR determines the number of characters in a specified character string. It can appear anywhere in a VAX MACRO program and is useful in calculating the length of macro arguments.

EXAMPLE

Macro definition:

```
.MACRO   CHAR    MESS                  ; Define MACRO
.NCHR    CHRCNT,<MESS>                 ; Assign value to CHRCNT
.WORD    CHRCNT                        ; Store value
.ASCII   /MESS/                        ; Store characters
.ENDM    CHAR                          ; Finish
```

Macro calls and expansions of the macro defined previously:

```
CHAR     <HELLO>                      ; CHRCNT will = 5
.NCHR    CHRCNT,<HELLO>               ; Assign value to CHRCNT
.WORD    CHRCNT                        ; Store value
.ASCII   /HELLO/                       ; Store characters

CHAR     <14, 75.39  4>               ; CHRCNT will = 12(dec)
.NCHR    CHRCNT,<14, 75.39  4>        ; Assign value to CHRCNT
.WORD    CHRCNT                        ; Store value
.ASCII   /14, 75.39  4/                ; Store characters
```

.NLIST

Listing directive

FORMAT

.NLIST *[argument-list]*

PARAMETER

argument-list

One or more of the symbolic arguments listed in Table 6–8 in the description of .SHOW. Use either the long form or the short form of the arguments. If you specify multiple arguments, separate them with commas, spaces, or tabs.

DESCRIPTION

.NLIST is equivalent to .NOSHOW. See the description of .SHOW for more information.

.NOCROSS

Cross-reference directive

FORMAT

.NOCROSS *[symbol-list]*

PARAMETER

symbol-list
A list of legal symbol names separated by commas.

DESCRIPTION

VAX MACRO produces a cross-reference listing when the /CROSS_REFERENCE qualifier is specified in the MACRO command. The .CROSS and .NOCROSS directives control which symbols are included in the cross-reference listing. The description of .NOCROSS is included with the description of .CROSS.

.NOSHOW

Listing directive

FORMAT

.NOSHOW *[argument-list]*

PARAMETER

argument-list
One or more of the symbolic arguments listed in Table 6–8 in the description of .SHOW. Use either the long form or the short form of the arguments. If you specify multiple arguments, separate them with commas, spaces, or tabs.

DESCRIPTION

.NOSHOW specifies listing control options. See the description of .SHOW for more information.

.NTYPE

Operand type directive

FORMAT

.NTYPE *symbol,operand*

PARAMETERS

symbol

Any legal VAX MACRO symbol. This symbol is assigned a value equal to the 8- or 16-bit addressing mode of the operand argument that follows.

operand

Any legal address expression, as you use it with an opcode. If no argument is specified, 0 is assumed.

DESCRIPTION

.NTYPE determines the addressing mode of the specified operand.

The value of the symbol is set to the specified addressing mode. In most cases, an 8-bit (1-byte) value is returned. Bits 0 through 3 specify the register associated with the mode, and bits 4 through 7 specify the addressing mode. To provide concise addressing information, the mode bits 4 through 7 are not exactly the same as the numeric value of the addressing mode described in Table C–6. Literal mode is indicated by a 0 in bits 4 through 7, instead of the values 0 through 3. Mode 1 indicates an immediate mode operand, mode 2 indicates an absolute mode operand, and mode 3 indicates a general mode operand.

For indexed addressing mode, a 16-bit (2-byte) value is returned. The high-order byte contains the addressing mode of the base operand specifier and the low-order byte contains the addressing mode of the primary operand (the index register).

See Chapter 5 of this volume for more information on addressing modes.

EXAMPLE

```
; The following macro is used to push an address on the stack.  It checks
; the operand type (by using .NTYPE) to determine if the operand is an
; address and, if not, the macro simply pushes the argument on the stack
; and generates a warning message.
;
        .MACRO  PUSHADR #ADDR
        .NTYPE  A,ADDR                  ; Assign operand type to 'A'
A = A@-4&^XF                            ; Isolate addressing mode
        .IF IDENTICAL 0,<ADDR>          ; Is argument exactly 0?
        PUSHL   #0                      ; Stack zero
        .MEXIT                          ; Exit from macro
        .ENDC
ERR = 0                                 ; ERR tells if mode is address
                                        ; ERR = 0 if address, 1 if not
        .IIF LESS_EQUAL A-1,   ERR=1    ; Is mode not literal or immediate?
        .IIF EQUAL A-5,   ERR=1         ; Is mode not register?
        .IF EQUAL  ERR                  ; Is mode address?
        PUSHAL  ADDR                    ; Yes, stack address
        .IFF                            ; No
        PUSHL   ADDR                    ; Then stack operand & warn
        .WARN   ; ADDR is not an address;
        .ENDC
        .ENDM   PUSHADR
```

Macro calls and expansions of the macro defined previously:

```
        PUSHADR (R0)                    ; Valid argument
        PUSHAL  (R0)                    ; Yes, stack address

        PUSHADR (R1)[R4]                ; Valid argument
        PUSHAL  (R1)[R4]                ; Yes, stack address

        PUSHADR 0                       ; Is zero
        PUSHL   #0                      ; Stack zero

        PUSHADR #1                      ; Not an address
        PUSHL   #1                      ; Then stack operand & warn
%MACRO-W-GENWRN, Generated WARNING: #1 is not an address

        PUSHADR R0                      ; Not an address
        PUSHL   R0                      ; Then stack operand & warn
%MACRO-W-GENWRN, Generated WARNING: R0 is not an address
```

Note that to save space, this example is listed as it would appear if .SHOW BINARY, not .SHOW EXPANSIONS, were specified in the source program.

.OCTA

Octaword storage directive

FORMAT

.OCTA *literal*
.OCTA *symbol*

PARAMETERS

literal

Any constant value. This value can be preceded by ^O, ^B, ^X, or ^D to specify the radix as octal, binary, hexadecimal, or decimal, respectively; or it can be preceded by ^A to specify ASCII text. Decimal is the default radix.

symbol

A symbol defined elsewhere in the program. This symbol results in a sign-extended, 32-bit value being stored in an octaword.

DESCRIPTION

.OCTA generates 128 bits (16 bytes) of binary data.

Note

.OCTA is like .QUAD and unlike other data storage directives (.BYTE, .WORD, and .LONG), in that it does not evaluate expressions and that it accepts only one value. It does not accept a list.

EXAMPLE

```
.OCTA   ^A"FEDCBA987654321"         ; Each ASCII character
                                    ;   is stored in a byte
.OCTA   0                           ; OCTA 0
.OCTA   ^X01234ABCD5678F9           ; OCTA hex value specified
.OCTA   VINTERVAL                   ; VINTERVAL has 32-bit value,
                                    ;   sign-extended
```

.ODD

Odd location counter alignment directive

FORMAT

.ODD

DESCRIPTION

.ODD ensures that the current value of the location counter is odd by adding 1 if the current value is even. If the current value is already odd, no action is taken.

.OPDEF

Opcode definition directive

FORMAT

.OPDEF *opcode value,operand-descriptor-list*

PARAMETERS

opcode

An ASCII string specifying the name of the opcode. The string can be up to 31 characters long and can contain the letters A through Z, the digits 0 through 9, and the special characters underline (_), dollar sign ($), and period (.). The string should not start with a digit and should not be surrounded by delimiters.

value

An expression that specifies the value of the opcode. The expression must be absolute and must not contain any undefined values (see Section 3.5). The value of the expression must be in the range of 0 through decimal 65,535 (hexadecimal FFFF), but you cannot use the values 252 through 255 because the architecture specifies these as the start of a 2-byte opcode. The expression is represented as follows:

if 0 < expression < 251	expression is a 1-byte opcode.
if expression > 255	expression bits 7:0 are the first byte of the opcode and expression bits 15:8 are the second byte of the opcode.

operand-descriptor-list

A list of operand descriptors that specifies the number of operands and the type of each. Up to 16 operand descriptors are allowed in the list. Table 6–5 lists the operand descriptors.

Table 6–5 Operand Descriptors

Access Type	Data Type								
	Byte	Word	Longword	Floating Point	Double Floating Point	G_ Floating Point	H_ Floating Point	Quadword	Octaword
Address	AB	AW	AL	AF	AD	AG	AH	AQ	AO
Read-only	RB	RW	RL	RF	RD	RG	RH	RQ	RO
Modify	MB	MW	ML	MF	MD	MG	MH	MQ	MO
Write-only	WB	WW	WL	WF	WD	WG	WH	WQ	WO
Field	VB	VW	VL	VF	VD	VG	VH	VQ	VO
Branch	BB	BW	—	—	—	—	—	—	—

DESCRIPTION

.OPDEF defines an opcode, which is inserted into a user-defined opcode table. The assembler searches this table before it searches the permanent symbol table. This directive can redefine an existing opcode name or create a new one.

Notes

1 You can also use a macro to redefine an opcode (see the description of .MACRO in this section). Note that the macro name table is searched before the user-defined opcode table.

2 .OPDEF is useful in creating "custom" instructions that execute user-written microcode. This directive is supplied to allow you to execute your microcode in a MACRO program.

3 The operand descriptors are specified in a format similar to the operand specifier notation described in Chapter 8. The first character specifies the operand access type, and the second character specifies the operand data type.

EXAMPLE

```
.OPDEF  MOVL3   ^XA9FF,RL,ML,WL        ; Defines an instruction
                                       ;   MOVL3, which uses
                                       ;   the reserved opcode FF
.OPDEF  DIVF2   ^X46,RF,MF             ; Redefines the DIVF2 and
.OPDEF  MOVC5   ^X2C,RW,AB,AB,RW,AB    ;   MOVC5 instructions

.OPDEF  CALL    ^X10,BB                ; Equivalent to a BSBB
```

.PACKED

Packed decimal string storage directive

FORMAT

.PACKED *decimal-string[,symbol]*

PARAMETERS

decimal-string

A decimal number from 0 through 31 digits long with an optional sign. Digits can be in the range of 0 through 9 (see Section 8.2.14).

symbol

An optional symbol that is assigned a value equivalent to the number of decimal digits in the string. The sign is not counted as a digit.

DESCRIPTION

.PACKED generates packed decimal data, two digits per byte. Packed decimal data is useful in calculations requiring exact accuracy. Packed decimal data is operated on by the decimal string instructions. See Section 8.2.14 for more information on the format of packed decimal data.

EXAMPLE

```
.PACKED -12,PACK_SIZE          ; PACK_SIZE gets value of 2
.PACKED +500
.PACKED 0
.PACKED -0,SUM_SIZE            ; SUM_SIZE gets value of 1
```

.PAGE

Page ejection directive

FORMAT

.PAGE

DESCRIPTION

.PAGE forces a new page in the listing. The directive itself is not printed in the listing.

VAX MACRO ignores .PAGE in a macro definition. The paging operation is performed only during macro expansion.

.PRINT

Assembly message directive

FORMAT

.PRINT *[expression] ;comment*

PARAMETERS

expression
An expression whose value is displayed when .PRINT is encountered during assembly.

;comment
A comment that is displayed when .PRINT is encountered during assembly. The comment must be preceded by a semicolon.

DESCRIPTION

.PRINT causes the assembler to display an informational message. The message consists of the value of the expression and the comment specified in the .PRINT directive. The message is displayed on the terminal for interactive jobs and in the log file for batch jobs. The message produced by .PRINT is not considered an error or warning message.

Notes

1. .PRINT, .ERROR, and .WARN are called the message display directives. You can use these to display information indicating that a macro call contains an error or an illegal set of conditions.
2. If .PRINT is included in a macro library, end the comment with an additional semicolon. If you omit the semicolon, the comment will be stripped from the directive and will not be displayed when the macro is called.
3. If the expression has a value of 0, it is not displayed with the message.

EXAMPLE

```
.PRINT  2       ; The sine routine has been changed
```

.PSECT

Program sectioning directive

FORMAT

.PSECT *[program-section-name[,argument-list]]*

PARAMETERS

program-section-name

The name of the program section. This name can be up to 31 characters long and can contain any alphanumeric character and the special characters underline (_), dollar sign ($), and period (.). The first character must not be a digit.

argument-list

A list containing the program section attributes and the program section alignment. Table 6–6 lists the attributes and their functions. Table 6–7 lists the default attributes and their opposites. Program sections are aligned when you specify an integer in the range of 0 through 9 or one of the five keywords listed below. If you specify an integer, the program section is linked to begin at the next virtual address, which is a multiple of 2 raised to the power of the integer. If you specify a keyword, the program section is linked to begin at the next virtual address (a multiple of the values listed below):

Keyword	Size (in Bytes)
BYTE	2^0 = 1
WORD	2^1 = 2
LONG	2^2 = 4
QUAD	2^3 = 8
PAGE	2^9 = 512

BYTE is the default.

Table 6–6 Program Section Attributes

Attribute	Function
ABS	Absolute—The linker assigns the program section an absolute address. The contents of the program section can be only symbol definitions (usually definitions of symbolic offsets to data structures that are used by the routines being assembled). No data allocations can be made. An absolute program section contributes no binary code to the image, so its byte allocation request to the linker is 0. The size of the data structure being defined is the size of the absolute program section printed in the "program section synopsis" at the end of the listing. Compare this attribute with its opposite, REL.

Table 6–6 (Cont.) Program Section Attributes

Attribute	Function
CON	Concatenate—Program sections with the same name and attributes (including CON) are merged into one program section. Their contents are merged in the order in which the linker acquires them. The allocated virtual address space is the sum of the individual requested allocations.
EXE	Executable—The program section contains instructions. This attribute provides the capability of separating instructions from read-only and read/write data. The linker uses this attribute in gathering program sections and in verifying that the transfer address is in an executable program section.
GBL	Global—Program sections that have the same name and attributes, including GBL and OVR, will have the same relocatable address in memory even when the program sections are in different clusters (see the *VMS Linker Utility Manual* for more information on clusters). This attribute is specified for FORTRAN COMMON block program sections (see the *VAX FORTRAN User's Guide*). Compare this attribute with its opposite, LCL.
LCL	Local—The program section is restricted to its cluster. Compare this attribute with its opposite, GBL.
LIB	Library Segment—Reserved for future use.
NOEXE	Not Executable—The program section contains data only; it does not contain instructions.
NOPIC	Non-Position-Independent Content—The program section is assigned to a fixed location in virtual memory (when it is in a shareable image).
NORD	Nonreadable—Reserved for future use.
NOSHR	No Share—The program section is reserved for private use at execution time by the initiating process.
NOWRT	Nonwriteable—The contents of the program section cannot be altered (written into) at execution time.
OVR	Overlay—Program sections with the same name and attributes, including OVR, have the same relocatable base address in memory. The allocated virtual address space is the requested allocation of the largest overlaying program section. Compare this attribute with its opposite, CON.
PIC	Position-Independent Content—The program section can be relocated; that is, it can be assigned to any memory area (when it is in a shareable image).
RD	Readable—Reserved for future use.
REL	Relocatable—The linker assigns the program section a relocatable base address. The contents of the program section can be code or data. Compare this attribute with its opposite, ABS.
SHR	Share—The program section can be shared at execution time by multiple processes. This attribute is assigned to a program section that can be linked into a shareable image.

Table 6–6 (Cont.) Program Section Attributes

Attribute	Function
USR	User Segment—Reserved for future use.
VEC	Vector-Containing—The program section contains a change mode vector indicating a privileged shareable image. You must use the SHR attribute with VEC.
WRT	Write—The contents of the program section can be altered (written into) at execution time.

Table 6–7 Default Program Section Attributes

Default	Opposite
CON	OVR
EXE	NOEXE
LCL	GBL
NOPIC	PIC
NOSHR	SHR
RD	NORD
REL	ABS
WRT	NOWRT
NOVEC	VEC

DESCRIPTION

.PSECT defines a program section and its attributes and refers to a program section once it is defined. Use program sections to do the following:

- Develop modular programs
- Separate instructions from data
- Allow different modules to access the same data
- Protect read-only data and instructions from being modified
- Identify sections of the object module to the debugger
- Control the order in which program sections are stored in virtual memory

The assembler automatically defines two program sections: the absolute program section and the unnamed (or blank) program section. Any symbol definitions that appear before any instruction, data, or .PSECT directive are placed in the absolute program section. Any instructions or data that appear before the first named program section is defined are placed in the unnamed program section. Any .PSECT directive that does not include a program section name specifies the unnamed program section.

A maximum of 254 user-defined, named program sections can be defined.

When the assembler encounters a .PSECT directive that specifies a new program section name, it creates a new program section and stores the name, attributes, and alignment of the program section. The assembler includes all data and instructions that follow the .PSECT directive in that program section until it encounters another .PSECT directive. The assembler starts all program sections at a location counter of 0, which is relocatable.

If the assembler encounters a .PSECT directive that specifies the name of a previously defined program section, it stores the new data or instructions after the last entry in the previously defined program section. The location counter is set to the value of the location counter at the end of the previously defined program section. You need not list the attributes when continuing a program section but any attributes that are listed must be the same as those previously in effect for the program section. A continuation of a program section cannot contain attributes conflicting with those specified in the original .PSECT directive.

The attributes listed in the .PSECT directive only describe the contents of the program section. The assembler does not check to ensure that the contents of the program section actually include the attributes listed. However, the assembler and the linker do check that all program sections with the same name have exactly the same attributes. The assembler and linker display an error message if the program section attributes are not consistent.

Program section names are independent of local symbol, global symbol, and macro names. You can use the same symbolic name for a program section and for a local symbol, global symbol, or macro name.

Notes

1. The .ALIGN directive cannot specify an alignment greater than that of the current program section; consequently, .PSECT should specify the largest alignment needed in the program section. For efficiency of execution, an alignment of longword or larger is recommended for all program sections that have longword data.
2. The attributes of the default absolute and the default unnamed program sections are listed below. Note that the program section names include the periods and enclosed spaces.

Program Section Name	Attributes and Alignment
. ABS .	NOPIC,USR,CON,ABS,LCL,NOSHR,NOEXE, NORD,NOWRT,NOVEC,BYTE
. BLANK .	NOPIC,USR,CON,REL,LCL,NOSHR,EXE, RD,WRT,NOVEC,BYTE

EXAMPLE

```
.PSECT  CODE,NOWRT,EXE,LONG      ; Program section to contain
                                 ;   executable code
.PSECT  RWDATA,WRT,NOEXE,QUAD
                                 ; Program section to contain
                                 ;   modifiable data
```

.QUAD

Quadword storage directive

FORMAT

.QUAD *literal*
.QUAD *symbol*

PARAMETERS

literal

Any constant value. This value can be preceded by ^O, ^B, ^X, or ^D to specify the radix as octal, binary, hexadecimal, or decimal, respectively; or it can be preceded by ^A to specify the ASCII text operator. Decimal is the default radix.

symbol

A symbol defined elsewhere in the program. This symbol results in a sign-extended, 32-bit value being stored in a quadword.

DESCRIPTION

.QUAD generates 64 bits (eight bytes) of binary data.

Note

.QUAD is like .OCTA and different from other data storage directives (.BYTE, .WORD, and .LONG) in that it does not evaluate expressions and that it accepts only one value. It does not accept a list.

EXAMPLE

```
.QUAD   ^A'..ASK?..'            ; Each ASCII character is stored
                                ;   in a byte
.QUAD   0                       ; QUAD 0
.QUAD   ^X0123456789ABCDEF      ; QUAD hex value specified
.QUAD   ^B1111000111001101      ; QUAD binary value specified
.QUAD   LABEL                   ; LABEL has a 32-bit,
                                ;   zero-extended value.
```

.REFn

Operand generation directives

FORMAT

.REF1 *operand*
.REF2 *operand*
.REF4 *operand*
.REF8 *operand*
.REF16 *operand*

PARAMETER

operand
An operand of byte, word, longword, quadword, or octaword context, respectively.

DESCRIPTION

VAX MACRO has the following five operand generation directives that you can use in macros to define new opcodes:

Directive	Function
.REF1	Generates a byte operand
.REF2	Generates a word operand
.REF4	Generates a longword operand
.REF8	Generates a quadword operand
.REF16	Generates an octaword operand

The .REFn directives are provided for compatibility with VAX MACRO V1.0. Because the .OPDEF directive provides greater functionality and is easier to use than .REFn, you should use .OPDEF instead of .REFn.

EXAMPLE

```
.MACRO  MOVL3 A,B,C
.BYTE   ^XFF,^XA9
.REF4   A                 ; This operand has longword context
.REF4   B                 ; This operand has longword context
.REF4   C                 ; This operand has longword context
.ENDM   MOVL3

MOVL3   R0,@LAB-1,(R7)+[R10]
```

This example uses .REF4 to create a new instruction, MOVL3, which uses the reserved opcode FF. See the example in .OPDEF for a preferred method to create a new instruction.

.REPEAT

Repeat block directive

FORMAT

.REPEAT expression

.
.
.

range

.
.
.

.ENDR

PARAMETERS

expression
An expression whose value controls the number of times the range is to be assembled within the program. When the expression is less than or equal to 0, the repeat block is not assembled. The expression must be absolute and must not contain any undefined symbols (see Section 3.5).

range
The source text to be repeated the number of times specified by the value of the expression. The repeat block can contain macro definitions, indefinite repeat blocks, or other repeat blocks. .MEXIT is legal within the range.

DESCRIPTION

.REPEAT repeats a block of code a specified number of times, in line with other source code. The .ENDR directive specifies the end of the range.

Note

The alternate form of .REPEAT is .REPT.

EXAMPLE

Macro definition:

```
.MACRO  COPIES  STRING,NUM
.REPEAT NUM
.ASCII  /STRING/
.ENDR
.BYTE   0
.ENDM   COPIES
```

Macro calls and expansions of the macro defined previously:

```
        COPIES  <ABCDEF>,5
        .REPEAT 5
        .ASCII  /ABCDEF/
        .ENDR
        .ASCII  /ABCDEF/
        .ASCII  /ABCDEF/
        .ASCII  /ABCDEF/
        .ASCII  /ABCDEF/
        .ASCII  /ABCDEF/
        .BYTE   0
VARB = 3
        COPIES  <HOW MANY TIMES>,VARB
        .REPEAT 3
        .ASCII  /HOW MANY TIMES/
        .ENDR
        .ASCII  /HOW MANY TIMES/
        .ASCII  /HOW MANY TIMES/
        .ASCII  /HOW MANY TIMES/
        .BYTE   0
```

.RESTORE_PSECT

Restore previous program section context directive

FORMAT

.RESTORE_PSECT

DESCRIPTION

.RESTORE_PSECT retrieves the program section from the top of the program section context stack, an internal stack in the assembler. If the stack is empty when .RESTORE_PSECT is issued, the assembler displays an error message. When .RESTORE_PSECT retrieves a program section, it restores the current location counter to the value it had when the program section was saved. The local label block is also restored if it was saved when the program section was saved. See the description of .SAVE_PSECT for more information.

Note

The alternate form of .RESTORE_PSECT is .RESTORE.

EXAMPLE

.RESTORE_PSECT and .SAVE_PSECT are especially useful in macros that define program sections. The macro definition below saves the current program section context and defines new program sections. Then, it restores the saved program section. If the macro did not save and restore the program section context each time the macro was invoked, the program section would change.

```
        .MACRO  INITD                  ; Initialize symbols
                                       ;   and data areas
        .SAVE_PSECT                    ; Save the current PSECT
        .PSECT  SYMBOLS,ABS            ; Define new PSECT
HELP_LEV=2                             ; Define symbol
MAXNUM=100                             ; Define symbol
RATE1=16                               ; Define symbol
RATE2=4                                ; Define symbol
        .PSECT  DATA,NOEXE,LONG        ; Define another PSECT
TABL:   .BLKL   100                    ; 100 longwords in TABL
TEMP:   .BLKB   16                     ; More storage
        .RESTORE_PSECT                 ; Restore the PSECT
                                       ;   in effect when
                                       ;   MACRO is invoked
        .ENDM
```

.SAVE_PSECT

Save current program section context directive

FORMAT

.SAVE_PSECT [LOCAL_BLOCK]

PARAMETER

LOCAL_BLOCK
An optional keyword that specifies that the current local label is to be saved with the program section context.

DESCRIPTION

.SAVE_PSECT stores the current program section context on the top of the program section context stack, an internal assembler stack. It leaves the current program section context in effect. The program section context stack can hold 31 entries. Each entry includes the value of the current location counter and the maximum value assigned to the location counter in the current program section. If the stack is full when .SAVE_PSECT is encountered, an error occurs.

.SAVE_PSECT and .RESTORE_PSECT are especially useful in macros that define program sections. See the description of .RESTORE_PSECT for another example using .SAVE_PSECT.

Note

The alternate form of .SAVE_PSECT is .SAVE.

EXAMPLE

```
Macro definition:

        .MACRO  ERR_MESSAGE,TEXT          ; Set up lists of messages
                                          ;   and pointers
                                          ;
        .IIF    NOT_DEFINED      MESSAGE_INDEX, MESSAGE_INDEX=0
        .SAVE_PSECT -
                LOCAL_BLOCK               ; Keep local labels
        .PSECT  MESSAGE_TEXT              ; List of error messages
MESSAGE::
        .ASCIC  /TEXT/
        .PSECT  MESSAGE_POINTERS          ; Addresses of error
        .ADDRESS -                        ;   messages
                MESSAGE                   ; Store one pointer
        .RESTORE_PSECT                    ; Get back local labels
        PUSHL   #MESSAGE_INDEX            ;
        CALLS   #1,PRINT_MESS             ; Print message
MESSAGE_INDEX=MESSAGE_INDEX+1
        .ENDM   ERR_MESSAGE
```

Macro call:

```
RESETS: CLRL    R4
        BLBC    R0,30$
        ERR_MESSAGE     <STRING TOO SHORT> ; Add "STRING TOO SHORT"
                                           ;   to list of error
30$:    RSB                                ;   messages
```

By using .SAVE_PSECT LOCAL_BLOCK, the local label 30$ is defined in the same local label block as the reference to 30$. If a local label is not defined in the block in which it is referenced, the assembler produces the following error message:

```
%MACRO-E-UNDEFSYM, Undefined Symbol
```

.SHOW
.NOSHOW

Listing directives

FORMAT

.SHOW *[argument-list]*
.NOSHOW *[argument-list]*

PARAMETER

argument-list

One or more of the optional symbolic arguments defined in Table 6–8. You can use either the long form or the short form of the arguments. You can use each argument alone or in combination with other arguments. If you specify multiple arguments, you must separate them by commas, tabs, or spaces. If any argument is not specifically included in a listing control statement, the assembler assumes its default value (show or noshow) throughout the source program.

Table 6–8 .SHOW and .NOSHOW Symbolic Arguments

Long Form	Short Form	Default	Function
BINARY	MEB	Noshow	Lists macro and repeat block expansions that generate binary code. BINARY is a subset of EXPANSIONS.
CALLS	MC	Show	Lists macro calls and repeat block specifiers.
CONDITIONALS	CND	Show	Lists unsatisfied conditional code associated with the conditional assembly directives.
DEFINITIONS	MD	Show	Lists macro and repeat range definitions that appear in an input source file.
EXPANSIONS	ME	Noshow	Lists macro and repeat range expansions.

DESCRIPTION

.SHOW and .NOSHOW specify listing control options in the source text of a program. You can use .SHOW and .NOSHOW with or without an argument list.

When you use them with an argument list, .SHOW includes and .NOSHOW excludes the lines specified in Table 6–8. .SHOW and .NOSHOW control the listing of the source lines that are in conditional assembly blocks (see the description of .IF), macros, and repeat blocks.

When you use them without arguments, these directives alter the listing level count. The listing level count is initialized to 0. Each time .SHOW appears in a program, the listing level count is incremented; each time .NOSHOW appears in a program, the listing level count is decremented.

When the listing level count is negative, the listing is suppressed (unless the line contains an error). Conversely, when the listing level count is positive, the listing is generated. When the count is 0, the line is either listed or suppressed, depending on the value of the listing control symbolic arguments.

Notes

1 The listing level count allows macros to be listed selectively; a macro definition can specify .NOSHOW at the beginning to decrement the listing count and can specify .SHOW at the end to restore the listing count to its original value.

2 The alternate forms of .SHOW and .NOSHOW are .LIST and .NLIST.

EXAMPLE

```
        .MACRO  XX
          .
          .
          .
        .SHOW                   ; List next line
X=.
        .NOSHOW                 ; Do not list remainder
          .                     ;   of macro expansion
          .
          .
        .ENDM
        .NOSHOW EXPANSIONS      ; Do not list macro
                                ;   expansions
        XX
X=.
```

.SIGNED_BYTE

Signed byte data directive

FORMAT

.SIGNED_BYTE *expression-list*

PARAMETERS

expression-list

An expression or list of expressions separated by commas. You have the option of following each expression with a repetition factor delimited by square brackets.

An expression followed by a repetition factor has the format:

expression1[expression2]

expression1

An expression that specifies the value to be stored. The value must be in the range −128 through +127.

[expression2]

An expression that specifies the number of times the value will be repeated. The expression must not contain any undefined symbols and must be an absolute expression (see Section 3.5). The square brackets are required.

DESCRIPTION

.SIGNED_BYTE is equivalent to .BYTE, except that VAX MACRO indicates that the data is signed in the object module. The linker uses this information to test for overflow conditions.

Note

Specifying .SIGNED_BYTE allows the linker to detect overflow conditions when the value of the expression is in the range of 128 through 255. Values in this range can be stored as unsigned data but cannot be stored as signed data in a byte.

EXAMPLE

```
.SIGNED_BYTE   LABEL1-LABEL2   ;  Data must fit
.SIGNED_BYTE   ALPHA[20]       ;    in byte
```

.SIGNED_WORD

Signed word storage directive

FORMAT

.SIGNED_WORD *expression-list*

PARAMETERS

expression-list

An expression or list of expressions separated by commas. You have the option of following each expression with a repetition factor delimited by square brackets.

An expression followed by a repetition factor has the format:

expression1[expression2]

expression1

An expression that specifies the value to be stored. The value must be in the range –32,768 through +32,767.

[expression2]

An expression that specifies the number of times the value will be repeated. The expression must not contain any undefined symbols and must be an absolute expression (see Section 3.5). The square brackets are required.

DESCRIPTION

.SIGNED_WORD is equivalent to .WORD except that the assembler indicates that the data is signed in the object module. The linker uses this information to test for overflow conditions. .SIGNED_WORD is useful after the case instruction to ensure that the displacement fits in a word.

Note

Specifying .SIGNED_WORD allows the linker to detect overflow conditions when the value of the expression is in the range of 32,768 through 65,535. Values in this range can be stored as unsigned data but cannot be stored as signed data in a word.

EXAMPLE

```
        .MACRO   CASE,SRC,DISPLIST,TYPE=W,LIMIT=#0,NMODE=S^#,?BASE,?MAX
                                        ; MACRO to use CASE instruction,
                                        ;   SRC is selector, DISPLIST
                                        ;   is list of displacements, TYPE
                                        ;   is B (byte) W (word) L (long),
                                        ;   LIMIT is base value of selector
        CASE'TYPE       SRC,LIMIT,NMODE'<<MAX-BASE>/2>-1
                                        ; Case instruction
BASE:                                   ; Local label specifying base
        .IRP    EP,<DISPLIST>           ;   to set up offset list
        .SIGNED_WORD    EP-BASE         ; Offset list
        .ENDR                           ;
MAX:                                    ; Local label used to count
        .ENDM   CASE                    ;   args

        CASE    IVAR    <ERR_PROC,SORT,REV_SORT>        ; If IVAR=0, error
        CASEW   IVAR,#0,S^#<<30001$-30000$>/2>-1

30000$:                                 ; Local label specifying base
        .SIGNED_WORD    ERR_PROC-30000$ ; Offset list
        .SIGNED_WORD    SORT-30000$     ; Offset list
        .SIGNED_WORD    REV_SORT-30000$ ; Offset list
30001$:                                 ; Local label used to count args
                                        ; =1, forward sort;  =2, backward
                                        ;   sort

        CASE    TEST    <TEST1,TEST2,TEST3>,L,#1
        CASEL   TEST,#1,S^#<<30003$-30002$>/2>-1
30002$:                                 ; Local label specifying base
        .SIGNED_WORD    TEST1-30002$    ; Offset list
        .SIGNED_WORD    TEST2-30002$    ; Offset list
        .SIGNED_WORD    TEST3-30002$    ; Offset list
30003$:                                 ; Local label used to count args
                                        ; Value of TEST can be 1, 2, or 3
```

In this example, the CASE macro uses .SIGNED_WORD to create a CASEB, CASEW, or CASEL instruction.

.SUBTITLE

Subtitle directive

FORMAT

.SUBTITLE *comment-string*

PARAMETER

comment-string
An ASCII string from 1 to 40 characters long; excess characters are truncated.

DESCRIPTION

.SUBTITLE causes the assembler to print the line of text, represented by the comment-string, in the table of contents (which the assembler produces immediately before the assembly listing). The assembler also prints the line of text as the subtitle on the second line of each assembly listing page. This subtitle text is printed on each page until altered by a subsequent .SUBTITLE directive in the program.

Note

The alternate form of .SUBTITLE is .SBTTL.

EXAMPLES

1
```
.SUBTITLE CONDITIONAL ASSEMBLY
```

This directive causes the assembler to print the following text as the subtitle of the assembly listing:

```
CONDITIONAL ASSEMBLY
```

It also causes the text to be printed out in the listing's table of contents, along with the source page number and the line sequence number of the source statement where .SUBTITLE was specified. The table of contents would have the following format:

2
```
TABLE OF CONTENTS
```

(1)	5000 ASSEMBLER DIRECTIVES
(2)	300 MACRO DEFINITIONS
(2)	2300 DATA TABLES AND INITIALIZATION
(3)	4800 MAIN ROUTINES
(4)	2800 CALCULATIONS
(4)	5000 I/O ROUTINES
(5)	1300 CONDITIONAL ASSEMBLY

.TITLE

Title directive

FORMAT

.TITLE *module-name comment-string*

PARAMETERS

module-name

An identifier from 1 to 31 characters long.

comment-string

An ASCII string from 1 to 40 characters long; excess characters are truncated.

DESCRIPTION

.TITLE assigns a name to the object module. This name is the first 31 or fewer nonblank characters following the directive.

Notes

1. The module name specified with .TITLE bears no relationship to the file specification of the object module, as specified in the VAX MACRO command line. The object module name appears in the linker load map and is also the module name that the debugger and librarian recognize.
2. If .TITLE is not specified, VAX MACRO assigns the default name .MAIN to the object module. If more than one .TITLE directive is specified in the source program, the last .TITLE directive encountered establishes the name for the entire object module.
3. When evaluating the module name, VAX MACRO ignores all spaces and/or tabs up to the first nonspace/nontab character after .TITLE.

EXAMPLE

```
.TITLE  EVAL    Evaluates Expressions
```

.TRANSFER

Transfer directive

FORMAT

.TRANSFER *symbol*

PARAMETER

symbol
A global symbol that is an entry point in a procedure or routine.

DESCRIPTION

.TRANSFER redefines a global symbol for use in a shareable image. The linker redefines the symbol as the value of the location counter at the .TRANSFER directive after a shareable image is linked.

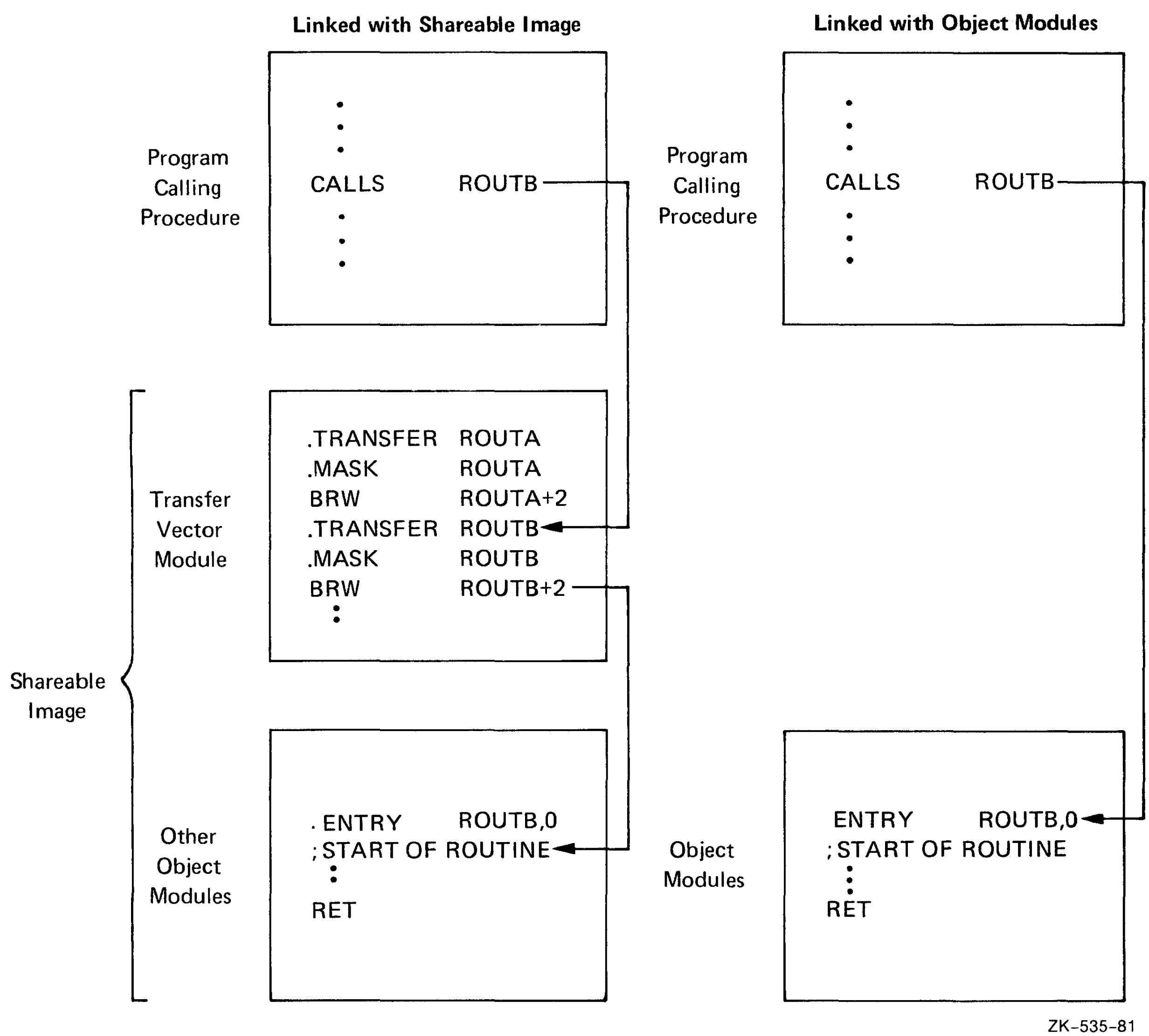
To make program maintenance easier, programs should not need to be relinked when the shareable images to which they are linked change. To avoid relinking whole programs when their linked shareable images change, keep the entry points in the changed shareable image at their original addresses. To do this, create an object module that contains a transfer vector for each entry point. Do not change the order of the transfer vectors. Link this object module at the beginning of the shareable image. The addresses of the entry points remain fixed even if the source code for a routine is changed. After each .TRANSFER directive, create a register save mask (for procedures only) and a branch to the first instruction of the routine.

The .TRANSFER directive does not cause any memory to be allocated and does not generate any binary code. It merely generates instructions to the linker to redefine the symbol when a shareable image is being created.

Use .TRANSFER with procedures entered by the CALLS or CALLG instruction. In this case, use .TRANSFER with the .ENTRY and .MASK directives. The branch to the actual routine must be a branch to the entry point plus 2 to bypass the 2-byte register save mask.

Figure 6–1 illustrates the use of transfer vectors.

Figure 6–1 Using Transfer Vectors



EXAMPLE

```
.TRANSFER ROUTINE_A
.MASK     ROUTINE_A,^M<R4,R5>   ; Copy entry mask
                                ;   and add registers
                                ;   R4 and R5
BRW       ROUTINE_A+2           ; Branch to routine
                                ;   (past entry mask)
   .
   .
   .
.ENTRY    ROUTINE_A,^M<R2,R3>   ; ENTRY point, save
                                ;   registers R2 and R3
   .
   .
   .
RET
```

In this example, .MASK copies the entry mask of a routine to the new entry address specified by .TRANSFER. If the routine is placed in a shareable image and then called, registers R2, R3, R4, and R5 will be saved.

.WARN

Warning directive

FORMAT

.WARN *[expression] ;comment*

PARAMETERS

expression
An expression whose value is displayed when .WARN is encountered during assembly.

;comment
A comment that is displayed when .WARN is encountered during assembly. The comment must be preceded by a semicolon.

DESCRIPTION

.WARN causes the assembler to display a warning message on the terminal or in the batch log file, and in the listing file (if there is one).

Notes

1. .WARN, .ERROR, and .PRINT are called the message display directives. Use them to display information indicating that a macro call contains an error or an illegal set of conditions.
2. When the assembly is finished, the assembler displays on the terminal or in the batch log file, the total number of errors, warnings, and information messages, and the page numbers and line numbers of the lines causing the errors or warnings.
3. If .WARN is included in a macro library, end the comment with an additional semicolon. If you omit the semicolon, the comment will be stripped from the directive and will not be displayed when the macro is called.
4. The line containing the .WARN directive is not included in the listing file.
5. If the expression has a value of 0, it is not displayed in the warning message.

EXAMPLE

```
.IF DEFINED   FULL
.IF DEFINED   DOUBLE_PREC
.WARN         ; This combination not tested
.ENDC
.ENDC
```

If the symbols FULL and DOUBLE_PREC are both defined, the following warning message is displayed:

```
%MACRO-W-GENWRN, Generated WARNING: This combination not tested
```

.WEAK

Weak symbol attribute directive

FORMAT

.WEAK *symbol-list*

PARAMETER

symbol-list
A list of legal symbols separated by commas.

DESCRIPTION

.WEAK specifies symbols that are either defined externally in another module or defined globally in the current module. .WEAK suppresses any object library search for the symbol.

When .WEAK specifies a symbol that is not defined in the current module, the symbol is externally defined. If the linker finds the symbol's definition in another module, it uses that definition. If the linker does not find an external definition, the symbol has a value of 0 and the linker does not report an error. The linker does not search a library for the symbol, but if a module brought in from a library for another reason contains the symbol definition, the linker uses it.

When .WEAK specifies a symbol that is defined in the current module, the symbol is considered to be globally defined. However, if this module is inserted in an object library, this symbol is not inserted in the library's symbol table. Consequently, searching the library at link time to resolve this symbol does not cause the module to be included.

EXAMPLE

```
.WEAK     IOCAR,LAB_3
```

.WORD

Word storage directive

FORMAT

.WORD *expression-list*

PARAMETERS

expression-list

One or more expressions separated by commas. You have the option of following each expression by a repetition factor delimited with square brackets.

An expression followed by a repetition factor has the format:

expression1[expression2]

expression1

An expression that specifies the value to be stored.

[expression2]

An expression that specifies the number of times the value will be repeated. The expression must not contain any undefined symbols and must be an absolute expression (see Section 3.5). The square brackets are required.

DESCRIPTION

.WORD generates successive words (two bytes) of data in the object module.

Notes

1. The expression is first evaluated as a longword, then truncated to a word. The value of the expression should be in the range of –32,768 through +32,767 for signed data or 0 through 65,535 for unsigned data. The assembler displays an error if the high-order two bytes of the longword expression have a value other than 0 or ^XFFFF.
2. The .SIGNED_WORD directive is the same as .WORD except that the assembler displays a diagnostic message if a value is in the range from 32,768 to 65,535.

EXAMPLE

```
.WORD     ^X3F,FIVE[3],32
```

VAX Data Types and Instruction Set

Part II describes the VAX data types, addressing mode formats, instruction formats, and the instructions themselves.

7 Terminology and Conventions

The following sections describe terminology and conventions used in Part II of this volume.

7.1 Numbering

All numbers, unless otherwise indicated, are decimal. Where there is ambiguity, numbers other than decimal are indicated with the base in English following the number in parentheses. For example:

```
FF (hex)
```

7.2 UNPREDICTABLE and UNDEFINED

Results specified as UNPREDICTABLE may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. Software can never depend on results specified as UNPREDICTABLE. Operations specified as UNDEFINED may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. The operation might vary from causing no effect to stopping system operation. UNDEFINED operations must not cause the processor to hang—to reach an unhalted state from which there is no transition to a normal state in which the machine executes instructions. Note the distinction between result and operation. Nonprivileged software cannot invoke UNDEFINED operations.

7.3 Ranges and Extents

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 (that is, bits 7:3 specifies an extent of bits including bits 7, 6, 5, 4, and 3).

7.4 MBZ

Fields specified as MBZ (Must Be Zero) must never be filled by software with a nonzero value. If the processor encounters a nonzero value in a field specified as MBZ, a reserved operand fault or abort occurs if that field is accessible to nonprivileged software. MBZ fields that are accessible only to privileged software (kernel mode) cannot be checked for nonzero value by some or all VAX implementations. Nonzero values in MBZ fields accessible only to privileged software may produce UNDEFINED operation.

7.5 Reserved

Unassigned values of fields are reserved for future use. In many cases, some values are indicated as reserved to CSS and customers. Only these values should be used for nonstar.dard applications. The values indicated as reserved to DIGITAL and all MBZ (Must Be Zero) fields are to be used only to extend future standard architecture.

7.6 Figure Drawing Conventions

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

8 Basic Architecture

8.1 VAX Addressing

The basic addressable unit in VAX MACRO is the 8-bit byte. Virtual addresses are 32 bits long. Therefore, 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.

8.2 Data Types

The following sections describe the VAX data types.

8.2.1 Byte

A byte is eight contiguous bits starting on an addressable byte boundary. The bits are numbered from right to left 0 through 7.

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

A byte is specified by its address A. When interpreted arithmetically, a byte is a two's complement integer with bits of increasing significance ranging from bit 0 through bit 6, with 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 ranging from bit 0 through bit 7. The value of the unsigned integer is in the range 0 through 255.

8.2.2 Word

A word is two contiguous bytes starting on an arbitrary byte boundary. The 16 bits are numbered from right to left 0 through 15.

```
 1
 5                              0
+-------------------------------+
|                               | :A
+-------------------------------+
```

A word is specified by its address, A, which is the address of the byte containing bit 0. When interpreted arithmetically, a word is a two's complement integer with bits of increasing significance ranging from bit 0 through bit 14, with 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 ranging from bit 0 through bit 15. The value of the unsigned integer is in the range 0 through 65,535.

8.2.3 Longword

A longword is four contiguous bytes starting on an arbitrary byte boundary. The 32 bits are numbered from right to left 0 through 31.

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

A longword is specified by its address, A, which is the address of the byte containing bit 0. When interpreted arithmetically, a longword is a two's complement integer with bits of increasing significance ranging from bit 0 through bit 30, with 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 ranging from bit 0 through bit 31. The value of the unsigned integer is in the range 0 through 4,294,967,295.

8.2.4 Quadword

A quadword is eight contiguous bytes starting on an arbitrary byte boundary. The 64 bits are numbered from right to left 0 through 63.

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

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

8.2.5 Octaword

An octaword is 16 contiguous bytes starting on an arbitrary byte boundary. The 128 bits are numbered from right to left 0 through 127.

```
 3
 1                                                            0
+-------------------------------------------------------------+
|                                                             | :A
+-------------------------------------------------------------+
|                                                             | :A+4
+-------------------------------------------------------------+
|                                                             | :A+8
+-------------------------------------------------------------+
|                                                             | :A+12
+-------------------------------------------------------------+
 1                                                            9
 2                                                            6
 7
```

An octaword is specified by its address, A, which is the address of the byte containing bit 0. When interpreted arithmetically, an octaword is a two's complement integer with bits of increasing significance ranging from bit 0 through bit 126, with bit 127 the sign bit. The value of the integer is in the range −2**127 to +2**127−1. The octaword data type is not fully supported by VAX instructions.

8.2.6 F_floating

An F_floating datum is four contiguous bytes starting on an arbitrary byte boundary. The 32 bits are labeled from right to left 0 through 31.

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

An F_floating datum is specified by its address, A, which is the address of the byte containing bit 0. The form of an F_floating datum is sign magnitude with bit 15 as the sign bit, bits 14:7 as an excess 128 binary exponent, and bits 6:0 and 31:16 as a normalized 24-bit fraction with the redundant most-significant fraction bit not represented. Within the fraction, bits of increasing significance range from bits 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 Appendix E). 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; that is, typically seven decimal digits.

8.2.7 D_floating

A D_floating datum is eight contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from right to left 0 through 63.

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

A D_floating datum is specified by its address, A, which is the address of the byte containing bit 0. The form of a D_floating datum is identical to an F_floating datum except for additional 32 low-significance fraction bits. Within the fraction, bits of increasing significance range from bits 48 through 63, 32 through 47, 16 through 31, and 0 through 6. The exponent conventions and the approximate range of values are the same for D_floating as they are for F_floating. The precision of a D_floating datum is approximately one part in 2**55, typically, 16 decimal digits.

8.2.8 G_floating

A G_floating datum is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from right to left 0 through 63.

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

A G_floating datum is specified by its address, A, which is the address of the byte containing bit 0. The form of a G_floating datum is sign magnitude, with bit 15 as the sign bit, bits 14:4 as an excess 1024 binary exponent, and bits 3:0 and 63:16 as a normalized 53-bit fraction with the redundant most-significant fraction bit not represented. Within the fraction, bits of increasing significance range from bits 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 Appendix E). 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; that is, typically 15 decimal digits.

8.2.9 H_floating

An H_floating datum is 16 contiguous bytes starting on an arbitrary byte boundary. The 128 bits are labeled from right to left 0 through 127.

```
 1 1
 5 4                           0
+-+-----------------------------+
|S|          exponent           | :A
+-+-----------------------------+
|            fraction           | :A+2
+-------------------------------+
|            fraction           | :A+4
+-------------------------------+
|            fraction           | :A+6
+-------------------------------+
|            fraction           | :A+8
+-------------------------------+
|            fraction           | :A+10
+-------------------------------+
|            fraction           | :A+12
+-------------------------------+
|            fraction           | :A+14
+-------------------------------+
```

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 as the sign bit, bits 14:0 as an excess 16,384 binary exponent, and bits 127:16 as a normalized 113-bit fraction with the redundant most-significant fraction bit not represented. Within the fraction, bits of increasing significance range from bits 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 32,767. 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 32,767 indicate true binary exponents of −16,383 through +16,383. 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 Appendix E). 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.

8.2.10 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:

- Address A of a byte
- Bit position P, which is the starting location of the field with respect to bit 0 of the byte at A
- Size S of the field

The specification of a bit field is indicated by the following figure, where the field is the shaded area.

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

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

```
3
1                                                  3 2     0
+--------------------------------------------------+-------+
|                    byte offset                   | bwb   |
+--------------------------------------------------+-------+
```

The sign-extended 29-bit byte offset is added to the address A; 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, it is two's complement with bits of increasing significance ranging from bits 0 through S-2; bit S-1 is the sign bit. When interpreted as an unsigned integer, bits of increasing significance range from bits 0 to S-1. A field of size 0 has a value identically equal to 0.

A variable-length bit field may be contained in one to five bytes. From a memory management point of view, only the minimum number of aligned longwords necessary to contain the field may be actually referenced.

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

```
3
1       P P-1                                               0
+---------+-------------------------------------------------+
|/////////|                                                 | Rn
+---------+-------------------------------------------------+
|                                              |////////////| R[n+1]
+----------------------------------------------+------------+
                                             P+S P+S-1
```

For further details on the specification of variable-length bit fields, see the descriptions of the variable-length bit field instructions in Section 9.5.

8.2.11 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. Thus, the format of a character string is represented as follows:

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

         .
         .
         .

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

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

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

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

8.2.12 Trailing Numeric String

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 through 9).

The representation for the high-order digits is as follows:

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

The highest-addressed byte of a trailing numeric string represents an encoding of both the least-significant digit and the sign of the numeric string. The VAX numeric string instructions support any encoding; however, DIGITAL software uses three encodings. These are:

- Unsigned numeric encoding, in which there is no sign and the least-significant digit contains an ASCII decimal digit character
- Zoned numeric encoding
- Overpunched numeric encoding

Because compilers of many manufacturers over the years have used the overpunch format and various card encodings, several variations in overpunch format have evolved. Typically, these alternate forms are accepted on input; the normal form is generated as the output for all operations. The valid representations of the digit and sign in each of the latter two formats is indicated in Table 8–1 and Table 8–2.

Table 8–1 Representation of Least-Significant Digit and Sign in Zoned Numeric Format

Digit	Decimal	Hex	ASCII Character
0	48	30	0
1	49	31	1
2	50	32	2
3	51	33	3
4	52	34	4
5	53	35	5
6	54	36	6
7	55	37	7
8	56	38	8
9	57	39	9
-0	112	70	p
-1	113	71	q
-2	114	72	r
-3	115	73	s
-4	116	74	t
-5	117	75	u
-6	118	76	v
-7	119	77	w
-8	120	78	x
-9	121	79	y

Table 8–2 Representation of Least-Significant Digit and Sign in Overpunch Format

			ASCII Character	
Digit	**Decimal**	**Hex**	**norm**	**alt.**
0	123	7B	{	0[?
1	65	41	A	1
2	66	42	B	2
3	67	43	C	3
4	68	44	D	4
5	69	45	E	5
6	70	46	F	6
7	71	47	G	7
8	72	48	H	8
9	73	49	I	9
-0	125	7D	}	]:!
-1	74	4A	J	
-2	75	4B	K	
-3	76	4C	L	
-4	77	4D	M	
-5	78	4E	N	
-6	79	4F	O	
-7	80	50	P	
-8	81	51	Q	
-9	82	52	R	

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

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

```
Zoned Format or Unsigned        Overpunch Format

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

The trailing numeric string with a value of "–123" is represented as follows.

```
Zoned Format                          Overpunch Format

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

8.2.13 Leading Separate Numeric String

A leading separate numeric string is a contiguous sequence of bytes in memory. A leading separate numeric string is specified by 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 separate leading numeric string is stored in a separate byte. Valid sign bytes are indicated in the following table:

Sign	Decimal	Hex	ASCII character
+	43	2B	+
+	32	20	{blank}
–	45	2D	–

The preferred representation for "+" is ASCII "+". All subsequent bytes contain an ASCII digit character, as indicated in the following table:

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

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

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

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

The leading separate numeric string with a value of "–123" is represented as follows:

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

8.2.14 Packed Decimal String

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 (nibbles). Each nibble except the low nibble (bits 3:0) of the last (highest-addressed) byte must contain a decimal digit. The low nibble of the highest-addressed byte must contain a sign. The representation for the digits and sign is indicated as follows:

Digit or Sign	Decimal	Hexadecimal
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9
+	10,12,14, or 15	A,C,E, or F
–	11 or 13	B or D

The preferred sign representation is 12 for "+" and 13 for "–". The length L is the number of digits in the packed decimal string (not counting the sign); L must be in the range 0 through 31. When the number of digits is odd, the digits and the sign fit into a string of bytes whose length is defined by the following equation: $L/2(integerpartonly) + 1$. When the number of digits is even, it is required that an extra "0" digit appear in the high nibble (bits 7:4) of the first byte of the string. Again, the length in bytes of the string is $L/2 + 1$.

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

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

The packed decimal number "−12" has a length of 2 and is represented as follows:

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

8.3 Processor Status Longword (PSL)

The processor status longword (PSL) consists of a set of processor state variables associated with each process. Bits 31:16 of the PSL have privileged status. For information on this part of the PSL, refer to the *VAX Architecture Reference Manual*. Bits 15:0 of the PSL are referred to separately as the processor status word (PSW).

The format of the PSL is:

```
 3 3 2 2 2 2 2 2 2 2 2 2 2   1 1
 1 0 9 8 7 6 5 4 3 2 1 0   6 5               8 7 6 5 4 3 2 1 0
+-+-+---+-+-+---+---+-+------+--------------+-+-+-+-+-+-+-+-+
|C|T|   |F|I|CUR|PRV|M|      |              |D|F|I| | | | | |
|M|P|MBZ|P|S|MOD|MOD|B| IPL  |     MBZ      |V|U|V|T|N|Z|V|C|
| | |   |D| |   |   |Z|      |              | | | | | | | | |
+-+-+---+-+-+---+---+-+------+--------------+-+-+-+-+-+-+-+-+
```

The processor status word (PSW), bits 0 to 15 of the processor status longword, contains:

- The condition codes, which give information on the results produced by previous instructions.
- The exception enables, which control the processor action on certain exception conditions (see Appendix E).

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

8.3.1 C Bit

The C (carry) condition code bit, 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, no carry or borrow occurred.

8.3.2 V Bit

The V (overflow) condition code bit, 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 that there was a conversion error. When V is clear, no overflow or conversion error occurred.

8.3.3 Z Bit

The Z (zero) condition code, when set, indicates that the last instruction that affected Z produced a result that was 0. When Z is clear, the result was nonzero.

8.3.4 N Bit

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

8.3.5 T Bit

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

8.3.6 IV Bit

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

8.3.7 FU Bit

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

8.3.8 DV Bit

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

8.4 Permanent Exception Enables

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

8.4.1 Divide by Zero

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

8.4.2 Floating Overflow

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

8.5 Instruction and Addressing Mode Formats

The following sections describe the formats for instruction opcodes and for the operand specifiers used with the various addressing modes.

8.5.1 Opcode Formats

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

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

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

```
 1
 5             8 7              0
+---------------+---------------+
|    opcode     |    FC - FF    | :A
+---------------+---------------+
```

8.5.2 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 the following:

1 Read—The specified operand is read only.

2 Write—The specified operand is written only.

3 Modify—The specified operand is read, potentially modified, and written. This operation is not performed under a memory interlock.

4 Address—The address of the specified operand in the form of a longword is the actual instruction operand. The specified operand is not accessed directly, although the instruction may subsequently use the address to access that operand.

5 Variable bit field base address—This access type is a special variant of the address access type. Variable bit field base address type 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).

6 Branch—No operand is accessed. The operand specifier itself is a branch displacement.

Access types 1 through 5 are general mode addressing. Type 6 is branch mode addressing.

The data types include the following:

- Byte
- Word
- Longword and F_floating (equivalent for addressing mode considerations)
- Quadword, D_floating, and G_floating (equivalent for addressing mode considerations)
- Octaword and H_floating (equivalent for addressing mode considerations)

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

- Address—the operand size to be used in the address calculation in autoincrement, autodecrement, and index modes
- Branch—the size of the branch displacement

8.6 General Addressing Mode Formats

The following sections describe the operand specifier formats for the general addressing modes. For descriptions and examples of the use of the general addressing modes, see Chapter 5.

Notation for Describing Addressing Modes

The following notation describes the addressing modes:

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

Note: **In the formal descriptions of the addressing modes, the symbol for a register (for example, Rn or PC) always means the contents of the register (for example, the contents of register n or the contents of register 15). However, in text, when there is no ambiguity, the symbol for a register is often used as the name of a register (for example, Rn may be used for the name of register n, and PC may be used for the name of register 15).**

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.

8.6.1 Register Mode

The operand specifier format is:

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

No specifier extension follows.

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

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

The assembler notation for register mode is Rn.

8.6.2 Register Deferred Mode

The operand specifier format is:

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

No specifier extension follows.

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

OA = Rn

operand = (OA)

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

8.6.3 Autoincrement Mode

The operand specifier format is:

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

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

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

OA = Rn

Rn <- Rn + size

operand = (OA)

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

8.6.4 Autoincrement Deferred Mode

The operand specifier format is:

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

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

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

OA = (Rn)

Rn <- Rn + 4

operand = (OA)

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

8.6.5 Autodecrement Mode

The operand specifier format is:

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

No specifier extension follows.

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

Rn <- Rn - size

OA = Rn

operand = (OA)

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

8.6.6 Displacement Mode

There are three operand specifier formats:

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

The specifier extension is a signed byte displacement that follows the operand specifier. This is the byte displacement mode.

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

The specifier extension is a signed word displacement that follows the operand specifier. This is the word displacement mode.

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

The specifier extension is a longword displacement that follows the operand specifier. This is the longword displacement mode.

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

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

operand = (OA)
```

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

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

8.6.7 Displacement Deferred Mode

There are three operand specifier formats:

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

The specifier extension is a signed byte displacement that follows the operand specifier. This is the byte displacement deferred mode.

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

The specifier extension is a signed word displacement that follows the operand specifier. This is the word displacement deferred mode.

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

The specifier extension is a longword displacement that follows the operand specifier. This is the longword displacement deferred mode.

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

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

operand = (OA)

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

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

8.6.8 Literal Mode

The operand specifier format is:

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

No specifier extension follows.

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

operand = ZEXT(literal)

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

For operands of data type F_floating, G_floating, D_floating, and H_floating, the 6-bit literal field is composed of two 3-bit fields. These fields are illustrated in the following diagram, where *exp* is exponent and *fra* is fraction:

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

You use the exponent and fraction fields to form an F_floating or D_floating operand as follows:

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

Note that bits 63:32 are not present in an F_floating operand.

You use the exponent and fraction fields to form a G_floating operand as follows:

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

You use the exponent and fraction fields to form an H_floating operand as follows:

```
 1 1
 5 4                          0
+-+----------------------------+
|0|   16,384 + exp             |
+-+---+------------------------+
| fra |         0              | :A+2
+-----+------------------------+
|               0              | :A+4
+------------------------------+
|               0              | :A+6
+------------------------------+
|               0              | :A+8
+------------------------------+
|               0              | :A+10
+------------------------------+
|               0              | :A+12
+------------------------------+
|               0              | :A+14
+------------------------------+
```

The range of values available is given in Table 8–3 and Table 8–4 in both decimal and rational number notation.

Table 8–3 Floating-Point Literals Expressed as Decimal Numbers

Exponent	0	1	2	3	4	5	6	7
0	0.5	0.5625	0.625	0.6875	0.75	0.8125	0.875	0.9375
1	1.0	1.125	1.25	1.37	1.5	1.625	1.75	1.875
2	2.0	2.25	2.5	2.75	3.0	3.25	3.5	3.75
3	4.0	4.5	5.0	5.5	6.0	6.5	7.0	7.5
4	8.0	9.0	10.0	11.0	12.0	13.0	14.0	15.0
5	16.0	18.0	20.0	22.0	24.0	26.0	28.0	30.0
6	32.0	36.0	40.0	44.0	48.0	52.0	56.0	60.0
7	64.0	72.0	80.0	88.0	96.0	104.0	112.0	120.0

Table 8–4 Floating-Point Literals Expressed as Rational Numbers

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

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

8.6.9 Index Mode

The operand specifier format is:

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

Bits 15:8 contain a second operand specifier (termed the base operand specifier) for any of the addressing modes except register, literal, or index. The specification of register, literal, or index addressing mode results in an illegal addressing mode fault (see Appendix E). If the base operand specifier requires it, a specifier extension 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 (that is, 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. You normally use the base operand specifier to determine an operand address. This address is termed the base operand address (BOA). The address of the primary operand specified is determined by multiplying the contents of the index register x by the size of the primary operand in bytes (1 for byte; 2 for word; 4 for longword and F_floating; 8 for quadword, D_floating, and G_floating; and 16 for octaword and H_floating), adding BOA, and taking the result.

OA = BOA + {size * (Rx)}

operand = (OA)

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

Certain restrictions are placed on the index register x. You cannot use the PC as an index register. If you use it, a reserved addressing mode fault occurs (see Appendix E). If the base operand specifier is for an addressing mode that results in register modification (that is, autoincrement mode, autodecrement mode, or autoincrement deferred mode), the same register cannot be the index register. If it is, the primary operand address is UNPREDICTABLE.

The names of the addressing modes resulting from index mode addressing are formed by adding the suffix "indexed" to the addressing mode of the base operand specifier. The following list gives 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]

  or

  Immediate indexed— I^#constant[Rx] (Immediate indexed is recognized by the assembler, but is not generally useful. Note that the operand address is independent of the value of the constant.)
- 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]

8.7 Summary of General Mode Addressing

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

Table 8–5 General Register Addressing

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

Key:

D — Displacement
i — Any indexable addressing mode
— — Logically impossible
f — Reserved addressing mode fault
p — Program Counter addressing
u — UNPREDICTABLE
uq — UNPREDICTABLE for quadword, octaword, D_floating, H_floating, and G_floating, (and field if position and 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

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

Table 8–6 Program Counter Addressing

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

Key:

u — UNPREDICTABLE
y — Yes, always valid addressing mode
r — Read access
m — Modify access
w — Write access
a — Address access
v — Field access

8.8 Branch Mode Addressing Formats

There are two operand specifier formats:

```
         7       0
         +---------------+
1.       |     displ     |
         +---------------+
```

The operand specifier is a signed byte displacement.

```
         1
         5                            0
         +------------------------------+
2.       |            displ             |
         +------------------------------+
```

The operand specifier is a signed word displacement.

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

A = PC + SEXT(displ)

The assembler notation for byte and word branch displacement addressing is A, where A is the branch address. Note that you must use the branch address, and not the displacement.

9 VAX Instruction Set

9.1 Introduction

This section describes the instructions generally used by all software across all implementations of the VAX architecture.

You can find a more complete description of the instruction set in the *VAX Architecture Reference Manual.* The *VAX Architecture Reference Manual* also contains information on instructions that are generally used by privileged software and are specific to specialized portions of the VAX architecture, such as memory management, interrupts and exceptions, process dispatching, and processor registers.

A list of instructions and opcode assignments appears in Appendix D.

9.2 Instruction Descriptions

The instruction set is divided into the following 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, instructions that are closely related are combined into groups and described together. The instruction group description is composed of the following:

- The group name.
- The format of each instruction in the group, including the name and type of each instruction operand specifier and the order in which it appears in memory. Operand specifiers from left to right appear in increasing memory addresses.
- The effect on condition codes.

- Exceptions specific to the instruction. Exceptions that are generally possible for all instructions (for example, illegal or reserved addressing mode, T-bit, and memory management violations) are not listed.
- The opcodes, mnemonics, and names of each instruction in the group. The opcodes are given in hexadecimal.
- A description, in English, of the instruction.
- Optional notes on the instruction and programming examples.

9.2.1 Operand Specifier Notation

Operand specifiers are described as follows:

name . access-type data-type

name
A mnemonic name for the operand in the context of the instruction. The name is often abbreviated.

access-type
A letter denoting the operand specifier access type:

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 is given by **data-type**; that is, size to be used in autoincrement, autodecrement, and indexing.
b	No operand reference. Operand specifier is a branch displacement. Size of branch displacement is given by **data-type**.
m	Operand is read, potentially modified, and written. Note that this is *not* an indivisible memory operation. Also note that if the operand is not actually modified, it may not be written back. However, modify type operands are always checked for both read and write accessibility.
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. Context of address calculation is given by **data-type**. If the effective address is Rn, the operand is in Rn or R[n+1]'Rn.
w	Operand is written only.

data-type
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

9.2.2 Operation Description Notation

The operation of an instruction is given as a sequence of control and assignment statements in an ALGOL-like syntax. No attempt is made to formally define the syntax; it is assumed to be familiar to the reader. The notation used is an extension of the notation introduced in Section 8.6.

+	addition
–	subtraction, unary minus
*	multiplication
/	division (quotient only)
**	exponentiation
'	concatenation
<-	is replaced by
=	is defined as
Rn or R[n]	contents of register Rn
PC, SP, FP, or AP	the contents of register R15, R14, R13, or R12, respectively
PSW	the contents of the processor status word
PSL	the contents of the processor status long word
(x)	contents of memory location whose address is x
(x)+	contents of memory location whose address is x; x incremented by the size of operand referenced at x
–(x)	x decremented by size of operand to be referenced at x; contents of memory location whose address is x
<x:y>	a modifier 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

Use the following conventions:

- Other than alterations caused by (x)+, or –(x), and the advancement of the PC, only operands or portions of operands appearing on the left side of assignment statements are affected.
- No operator precedence is assumed, except that replacement (<-) has the lowest precedence. Precedence is indicated explicitly by { }.
- All arithmetic, logical, and relational operators are defined in the context of their operands. For example, "+" applied to floating operands means a floating add, while "+" applied to byte operands is an integer byte add. Similarly, "LSS" is a floating comparison when applied to floating operands, while "LSS" is an integer byte comparison when applied to byte operands.
- Instruction operands are evaluated according to the operand specifier conventions (see Chapter 8). The order in which operands appear in the instruction description has no effect on the order of evaluation.
- Condition codes generally indicate the effect of an operation on the value of actual stored results, not on "true" results (which might be generated internally to greater precision). For example, two positive integers can be added together and the sum stored as a negative value because of overflow. The condition codes indicate a negative value even though the "true" result is clearly positive.

9.3 Integer Arithmetic and Logical Instructions

The following instructions are described in this section:

	Description and Opcode	Number of Instructions
1.	Add Aligned Word ADAWI add.rw, sum.mw	1
2.	Add 2 Operand ADD{B,W,L}2 add.rx, sum.mx	3
3.	Add 3 Operand ADD{B,W,L}3 add1.rx, add2.rx, sum.wx	3
4.	Add with Carry ADWC add.rl, sum.ml	1
5.	Arithmetic Shift ASH{L,Q} cnt.rb, src.rx, dst.wx	2
6.	Bit Clear 2 Operand BIC{B,W,L}2 mask.rx, dst.mx	3
7.	Bit Clear 3 Operand BIC{B,W,L}3 mask.rx, src.rx, dst.wx	3
8.	Bit Set 2 Operand BIS{B,W,L}2 mask.rx, dst.mx	3
9.	Bit Set 3 Operand BIS{B,W,L}3 mask.rx, src.rx, dst.wx	3
10.	Bit Test BIT{B,W,L} mask.rx, src.rx	3
11.	Clear CLR{B,W,L,Q,O} dst.wx	5
12.	Compare CMP{B,W,L} src1.rx, src2.rx	3
13.	Convert CVT{B,W,L}{B,W,L} src.rx, dst.wy All pairs except BB,WW,LL	6
14.	Decrement DEC{B,W,L} dif.mx	3
15.	Divide 2 Operand DIV{B,W,L}2 divr.rx, quo.mx	3
16.	Divide 3 Operand DIV{B,W,L}3 divr.rx, divd.rx, quo.wx	3
17.	Extended Divide EDIV divr.rl, divd.rq, quo.wl, rem.wl	1
18.	Extended Multiply EMUL mulr.rl, muld.rl, add.rl, prod.wq	1
19.	Increment INC{B,W,L} sum.mx	3
20.	Move Complemented MCOM{B,W,L} src.rx, dst.wx	3

	Description and Opcode	Number of Instructions
21.	Move Negated MNEG{B,W,L} src.rx, dst.wx	3
22.	Move OV{B,W,L,Q} src.rx, dst.wx	4
23.	Move Zero-Extended MOVZ{BW,BL,WL} src.rx, dst.wy	3
24.	Multiply 2 Operand MUL{B,W,L}2 mulr.rx, prod.mx	3
25.	Multiply 3 Operand MUL{B,W,L}3 mulr.rx, muld.rx, prod.wx	3
26.	Push Long PUSHL src.rl, {-(SP).wl}	1
27.	Rotate Long ROTL cnt.rb, src.rl, dst.wl	1
28.	Subtract with Carry SBWC sub.rl, dif.ml	1
29.	Subtract 2 Operand SUB{B,W,L}2 sub.rx, dif.mx	3
30.	Subtract 3 Operand SUB{B,W,L}3 sub.rx, min.rx, dif.wx	3
31.	Test TST{B,W,L} src.rx	3
32.	Exclusive OR 2 Operand XOR{B,W,L}2 mask.rx, dst.mx	3
33.	Exclusive OR 3 Operand XOR{B,W,L}3 mask.rx, src.rx, dst.wx	3

ADAWI

Add Aligned Word Interlocked

FORMAT

opcode   add.rw, sum.mw

condition codes

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

exceptions

reserved operand fault
integer overflow

opcodes

58	ADAWI	Add Aligned Word Interlocked

DESCRIPTION

The addend operand is added to the sum operand, and the sum operand is replaced by the result. The operation is interlocked against similar operations on other processors in a multiprocessor system. The destination must be aligned on a word boundary; that is, bit 0 of the address of the sum operand must be 0. 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

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

condition codes

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

exceptions

integer overflow

opcodes

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

DESCRIPTION

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

Note

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*

condition codes

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

exceptions

integer overflow

opcodes

D8	ADWC	Add with Carry

DESCRIPTION

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

Notes

1. On overflow, the sum operand is replaced by the low-order bits of the true result.
2. The two additions in the operation are performed simultaneously.

ASH

Arithmetic Shift

FORMAT

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

condition codes

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

exceptions

integer overflow

opcodes

78	ASHL	Arithmetic Shift Long
79	ASHQ	Arithmetic Shift Quad

DESCRIPTION

The source operand is arithmetically shifted by the number of bits specified by the count operand and the destination operand is replaced by the result. The source operand is unaffected. A positive count operand shifts to the left, bringing zeros into the least significant bit. A negative count operand shifts to the right, bringing in copies of the most significant (sign) bit into the most significant bit. A 0 count operand replaces the destination operand with the unshifted source operand.

Notes

1. Integer overflow occurs on a left shift if any bit shifted into the sign bit position differs from the sign bit of the source operand.
2. If **cnt** GTR 32 (ASHL) or **cnt** GTR 64 (ASHQ), the destination operand is replaced by 0.
3. If **cnt** LEQ –31 (ASHL) or **cnt** LEQ –63 (ASHQ), all the bits of the destination operand are copies of the sign bit of the source operand.

BIC

Bit Clear

FORMAT

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

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 result of the logical AND on the destination operand and the one's complement of the mask operand replaces the destination operand. In 3 operand format, the result of the logical AND on the source operand and the one's complement of the mask operand replaces the destination operand.

BIS

Bit Set

FORMAT

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

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 result of the logical OR on the mask operand and the destination operand replaces the destination operand. In 3 operand format, the result of the logical OR on the mask operand and the source operand replaces the destination operand.

BIT

Bit Test

FORMAT

opcode *mask.rx, src.rx*

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 logical AND is performed on the mask operand and the source operand. Both operands are unaffected. The only action is to modify condition codes.

CLR

Clear

FORMAT

opcode *dst.wx*

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.

Note

CLR*x* **dst** is equivalent to MOV*x* S^#0, **dst**, but is one byte shorter.

CMP

Compare

FORMAT

opcode *src1.rx, src2.rx*

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 modify the condition codes.

CVT

Convert

FORMAT

opcode *src.rx, dst.wy*

condition codes

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

exceptions

integer overflow

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 of longer data type to a shorter one is done by truncation of the higher-numbered (most significant) bits.

Note

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*

condition codes

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

exceptions

integer overflow

opcodes

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

DESCRIPTION

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

Notes

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

2 DEC*x* **dif** is equivalent to SUB*x* S^#1, **dif**, but is one byte shorter.

DIV

Divide

FORMAT

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

condition codes

N	← quo LSS 0;
Z	← quo EQL 0;
V	← {integer overflow} OR {divr EQL 0};
C	← 0;

exceptions

integer overflow
divide by 0

opcodes

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

DESCRIPTION

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

Notes

1. Division is performed so that the remainder has the same sign as the dividend; that is, the result is truncated toward 0. (Note that a remainder of 0 is not saved.)
2. Integer overflow occurs only if the largest negative integer is divided by −1. On overflow, operands are affected as in note 3 following.
3. If the divisor operand is 0, then in 2 operand format the quotient operand is not affected; in 3 operand format the quotient operand is replaced by the dividend operand.

EDIV

Extended Divide

FORMAT

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

condition codes

N ⟵ quo LSS 0;
Z ⟵ quo EQL 0;
V ⟵ {integer overflow} OR {divr EQL 0};
C ⟵ 0;

exceptions

integer overflow
divide by 0

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 note 3, following.
3. If the divisor operand is 0, then the quotient operand is replaced by bits 31:0 of the dividend operand, and the remainder operand is replaced by 0.

EMUL

Extended Multiply

FORMAT

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

condition codes

N ← prod LSS 0;
Z ← prod EQL 0;
V ← 0;
C ← 0;

exceptions

None.

opcodes

7A	EMUL	Extended Multiply

DESCRIPTION

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

INC

Increment

FORMAT

opcode *sum.mx*

condition codes

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

exceptions

integer overflow

opcodes

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

DESCRIPTION

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

Notes

1. Arithmetic overflow occurs if the largest positive integer is incremented. On overflow, the sum operand is replaced by the largest negative integer.
2. INC*x* **sum** is equivalent to ADD*x* S^#1, **sum**, but is one byte shorter.

MCOM

Move Complemented

FORMAT

opcode *src.rx, dst.wx*

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*

condition codes

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

exceptions

integer overflow

opcodes

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

DESCRIPTION

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

Note

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*

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.

MOVZ

Move Zero-Extended

FORMAT

opcode *src.rx, dst.wy*

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

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

condition codes

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

exceptions

integer overflow

opcodes

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

DESCRIPTION

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

Note

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

PUSHL

Push Long

FORMAT

opcode *src.rl*

condition codes

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

exceptions

None.

opcodes

DD	PUSHL	Push Long

DESCRIPTION

The longword source operand is pushed on the stack.

Notes

1. PUSHL is equivalent to MOVL **src**, –(SP), but is one byte shorter.
2. POPL is not a VAX instruction. However, the assembler recognizes the inclusion of *POPL destination* in a program, for which it generates the code for *MOVL (SP)+,destination*.

ROTL

Rotate Long

FORMAT

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

condition codes

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

exceptions

None.

opcodes

9C	ROTL	Rotate Long

DESCRIPTION

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

SBWC

Subtract with Carry

FORMAT

opcode *sub.rl, dif.ml*

condition codes

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

exceptions

integer overflow

opcodes

D9	SBWC	Subtract With Carry

DESCRIPTION

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

Notes

1. On overflow, the difference operand is replaced by the low-order bits of the true result.
2. The two subtractions in the operation are performed simultaneously.

SUB

Subtract

FORMAT

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

condition codes

N ⟵ dif LSS 0;
Z ⟵ dif EQL 0;
V ⟵ {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.

Note

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*

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 modified according to the value of the source operand.

Note

The operand **src** is equivalent to CMP*x* **src**, S^#0, but is one byte shorter.

XOR

Exclusive OR

FORMAT

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

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 result of the logical XOR on the mask operand and the destination operand replaces the destination operand. In 3 operand format, the result of the logical XOR on the mask operand and the source operand replaces the destination operand.

9.4 Address Instructions

The following instructions are described in this section.

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

MOVA

Move Address

FORMAT

opcode src.ax, dst.wl

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.

Note

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

PUSHA

Push Address

FORMAT

opcode *src.ax*

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. PUSHA*x* **src** is equivalent to MOVA*x* **src**, –(SP), but is one byte shorter.
2. The source operand is of address access type, which causes the address of the specified operand to be pushed.

9.5 Variable-Length Bit Field Instructions

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

1. A longword position operand.
2. A byte field size operand in the range 0 through 32; if out of this range, a reserved operand fault occurs.
3. A base address. Use the position operand to locate the bit field relative to this base address. The address is obtained from an operand of address access type. However, unlike other instances of operand specifiers of address access type, register mode can 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 8.) If the field is contained in a register and the size operand is not 0, the position operand must have a value in the range 0 through 31, or a reserved operand fault occurs.

Zero bytes are referenced if the field size is 0.

The following instructions are described in this section.

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

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

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

CMP

Compare Field

FORMAT

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

condition codes

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

exceptions

reserved operand

opcodes

EC	CMPV	Compare Field
ED	CMPZV	Compare Zero-Extended Field

DESCRIPTION

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

Notes

1 A reserved operand fault occurs if:

- **size** GTRU 32
- **pos** GTRU 31, **size** NEQ 0, and the field is contained in the registers

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

EXT

Extract Field

FORMAT

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

condition codes

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

exceptions

reserved operand

opcodes

EE	EXTV	Extract Field
EF	EXTZV	Extract Zero-Extended Field

DESCRIPTION

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

Notes

1 A reserved operand fault occurs if:

- **size** GTRU 32
- **pos** GTRU 31, **size** NEQ 0, and the field is contained in the registers

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

FF

Find First

FORMAT

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

condition codes

N ⟵ 0;
Z ⟵ {bit not found};
V ⟵ 0;
C ⟵ 0;

exceptions

reserved operand

opcodes

EB	FFC	Find First Clear
EA	FFS	Find First Set

DESCRIPTION

A field specified by the start position, size, and base operands is extracted. Starting at bit 0 and extending to the highest bit in the field, the field is tested for a bit in the state indicated by the instruction. If a bit in the indicated state is found, the find position operand is replaced by the position of the bit, and the Z condition code bit is cleared. If no bit in the indicated state is found, the find position operand is replaced by the position (relative to the base) of a bit one position to the left of the specified field, and the Z condition code bit is set. If the size operand is 0, the find position operand is replaced by the start position operand, and the Z condition code bit is set.

Notes

1. A reserved operand fault occurs if:
   - **size** GTRU 32
   - **startpos** GTRU 31, **size** NEQ 0, and the field is contained in the registers
2. On a reserved operand fault, the find position operand is unaffected, and the condition codes are UNPREDICTABLE.

INSV

Insert Field

FORMAT

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

condition codes

N ← N;
Z ← Z;
V ← V;
C ← C;

exceptions

reserved operand

opcodes

F0	INSV	Insert Field

DESCRIPTION

The field specified by the position, size, and base operands is replaced by bits *size* – 1:0 of the source operand. If the size operand is 0, the instruction has no effect.

Notes

1 A reserved operand fault occurs if:
- **size** GTRU 32
- **pos** GTRU 31, **size** NEQ 0, and the field is contained in the registers

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

9.6 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.

The following instructions are described in this section.

	Description and Opcode	Number of Instructions
1.	Add Compare and Branch ACB{B,W,L,F,D,G,H} limit.rx, add.rx, index.mx, displ.bw Compare is LE on positive add, GE on negative add.	7
2.	Add One and Branch Less Than or Equal AOBLEQ limit.rl, index.ml, displ.bb	1
3.	Add One and Branch Less Than AOBLSS limit.rl, index.ml, displ.bb	1
4.	Conditional Branch	12

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

	Description and Opcode	Number of Instructions
5.	Branch on Bit BB{S,C} pos.rl, base.vb, displ.bb, {field.rv}	2
6.	Branch on Bit (and modify without interlock) BB{S,C}{S,C} pos.rl, base.vb, displ.bb, {field.mv}	4
7.	Branch on Bit (and modify) Interlocked BB{SS,CC}I pos.rl, base.vb, displ.bb, {field.mv}	2

	Description and Opcode	Number of Instructions
8.	Branch on Low Bit BLB{S,C} src.rl, displ.bb	2
9.	Branch with {Byte, Word} Displacement BR{B,W} displ.bx	2
10.	Branch to Subroutine with {Byte, Word} Displacement BSB{B,W} displ.bx, {–(SP).wl}	2
11.	Case CASE{B,W,L} selector.rx, base.rx, limit.rx, displ.bw-list	3
12.	Jump JMP dst.ab	1
13.	Jump to Subroutine JSB dst.ab, {–(SP).wl}	1
14.	Return from Subroutine RSB {(SP)+.rl}	1
15.	Subtract One and Branch Greater Than or Equal SOBGEQ index.ml, displ.bb	1
16.	Subtract One and Branch Greater Than SOBGTR index.ml, displ.bb	1

ACB

Add Compare and Branch

FORMAT

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

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
4FFD	ACBG	Add Compare and Branch G_floating
6F	ACBD	Add Compare and Branch D_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 to 0, or if the addend is negative and the comparison is greater than or equal to 0, the sign-extended branch displacement is added to the PC, and the PC is replaced by the result.

Notes

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

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

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

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

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

6 Except for the circumstance described in note 5, the C-bit is unaffected.

AOBLEQ

Add One and Branch Less Than or Equal

FORMAT

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

condition codes

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

exceptions

integer overflow

opcodes

F3	AOBLEQ	Add One and Branch Less Than or Equal

DESCRIPTION

One is added to the index operand, and the index operand is replaced by the result. The index operand is compared with the limit operand. If the comparison is less than or equal to 0, the sign-extended branch displacement is added to the PC, and the PC is replaced by the result.

Notes

1. Integer overflow occurs if the index operand before addition is the largest positive integer. On overflow, the index operand is replaced by the largest negative integer, and the branch is taken.
2. The C-bit is unaffected.

AOBLSS

Add One and Branch Less Than

FORMAT

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

condition codes

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

exceptions

integer overflow

opcodes

F2	AOBLSS	Add One and Branch Less Than

DESCRIPTION

One is added to the index operand and the index operand is replaced by the result. The index operand is compared with the limit operand. If the comparison result is less than 0, the sign-extended branch displacement is added to the PC, and the PC is replaced by the result.

Notes

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

B

Branch on (condition)

FORMAT

opcode *displ.bb*

condition codes

N ⟵ N;
Z ⟵ Z;
V ⟵ V;
C ⟵ C;

exceptions

None.

opcodes

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,	Branch on Not Equal (signed)
		BNEQU	Branch on Not Equal Unsigned
13	Z EQL 1	BEQL,	Branch on Equal (signed)
		BEQLU	Branch on Equal Unsigned
18	N EQL 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,	Branch on Greater Than or Equal Unsigned
		BCC	Branch on Carry Clear
1F	C EQL 1	BLSSU,	Branch on Less Than Unsigned
		BCS	Branch on Carry Set

DESCRIPTION

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

Typically, you would use these instructions 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 floating-point instructions, decimal string instructions, and integer and field instructions where the operands are being treated as signed integers.

BB

Branch on Bit

FORMAT

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

condition codes

N ⟵ N;
Z ⟵ Z;
V ⟵ V;
C ⟵ C;

exceptions

reserved operand

opcodes

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

DESCRIPTION

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

Notes

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

2 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*

condition codes

N ⟵ N;
Z ⟵ Z;
V ⟵ V;
C ⟵ C;

exceptions

reserved operand

opcodes

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

DESCRIPTION

The single bit field specified by the position and base operands is tested. If it is in the test state indicated by the instruction, the sign-extended branch displacement is added to the PC, and the 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. A reserved operand fault occurs if **pos** GTRU 31 and the bit is contained in a register.
2. On a reserved operand fault, the field is unaffected, and the condition codes are UNPREDICTABLE.
3. 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*

condition codes

N ← N;
Z ← Z;
V ← V;
C ← C;

exceptions

reserved operand

opcodes

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

DESCRIPTION

The single bit field specified by the position and base operands is tested. If it is in the test state indicated by the instruction, the sign-extended branch displacement is added to the PC, and the 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. If the bit is contained in memory, the reading of the state of the bit and the setting of the bit to the new state is an interlocked operation. No other processor or I/O device can do an interlocked access on this bit during the interlocked operation.

Notes

1. A reserved operand fault occurs if **pos** GTRU 31 and the specified bit is contained in a register.
2. On a reserved operand fault, the field is unaffected, and the condition codes are UNPREDICTABLE.
3. Except for memory interlocking, BBSSI is equivalent to BBSS, and BBCCI is equivalent to BBCC.
4. This instruction is designed to modify interlocks with other processors or devices. For example, to implement "busy waiting":

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

BLB

Branch on Low Bit

FORMAT

opcode *src.rl, displ.bb*

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. If it is equal to the test state indicated by the instruction, the sign-extended branch displacement is added to the PC, and the PC is replaced by the result.

BR

Branch

FORMAT

opcode *displ.bx*

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 the PC, and the PC is replaced by the result.

BSB

Branch to Subroutine

FORMAT

opcode *displ.bx*

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

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

CASE

Case

FORMAT

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

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 the result replaces a temporary operand. The temporary operand 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 the PC, and the PC is replaced by the result. Otherwise, twice the sum of the limit operand and 1 is added to the PC, and the PC is replaced by the result. This operation causes the PC to be moved past the array of branch displacements. Regardless of the branch taken, the condition codes are modified as a result of the comparison of the temporary operand with the limit operand.

Notes

1. After operand evaluation, the PC points at **displ**[0], not to the next instruction. The branch displacements are relative to the address of **displ**[0].
2. The selector and base operands can both be considered as either signed or unsigned integers.

In the following example, the CASEB instruction selects one of eight displacements immediately following the instruction. The example is for illustration only. An actual instruction would use run-time variables instead of the assembly-time static values shown. Also, in an actual instruction, the displacements selected by the CASEB instruction would be branches to various routines.

```
        .PSECT    CODE, PIC, SHR, WRT, EXE, LONG
TABIND: .WORD 4
        .ENTRY    START,^M<>
        CLRW      R4
        CLRW      R5
        MOVW      #0,R4
        MOVW      #7,R5
        CASEB     TABIND,R4,R5
TAB:    .WORD     1$-TAB
        .WORD     2$-TAB
        .WORD     3$-TAB
        .WORD     4$-TAB
        .WORD     5$-TAB
        .WORD     6$-TAB
        .WORD     7$-TAB
        BRB       9$
1$:     .ASCII    /AT 1/
2$:     .ASCII    /AT 2/
3$:     .ASCII    /AT 3/
4$:     .ASCII    /AT 4/
5$:     .ASCII    /AT 5/
6$:     .ASCII    /AT 6/
7$:     .ASCII    /AT 7/
8$:     .ASCII    /AT 8/
9$:     $EXIT_S
        .END START
```

The objective of the CASE instruction is to transfer control to one of many possible locations depending on the value of "selector," or TABIND, as shown in the example. These locations are labeled in the example from 1$: to 8$:.

In the example, the table contains eight branch displacements. In all cases, the limit operand (here shown as R5, which contains a 7) is one less than the number of displacements (8) in the table. The base operand (here shown as R4, which contains a 0) is the lowest permissible value for TABIND.

The CASE instruction subtracts base (contents of R4, a 0) from the value of TABIND to produce a zero-origin index into the table. The limit (contents of R5, a 7) is compared with this index to ensure that the table limit is not exceeded.

After operand evaluation, the PC points to TAB:. The locations to which branching occurs are represented in the table as displacements. The displacement in the table selected by TABIND is added to the PC to form a destination address. The destination selected in the example is at location 5$:. In practical usage, this location would contain a branch to a specific routine.

JMP

Jump

FORMAT

opcode *dst.ab*

condition codes

N ← N;
Z ← Z;
V ← V;
C ← C;

exceptions

None.

opcodes

17	JMP	Jump

DESCRIPTION

The PC is replaced by the destination operand.

JSB

Jump to Subroutine

FORMAT

opcode *dst.ab*

condition codes

N ← N;
Z ← Z;
V ← V;
C ← C;

exceptions

None.

opcodes

16	JSB	Jump to Subroutine

DESCRIPTION

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

Note

Because the operand specifier conventions cause the evaluation of the destination operand before saving the PC, you can use JSB for coroutine calls with the stack used for linkage. The form of this call is:

```
JSB @(SP)+
```

RSB

Return from Subroutine

FORMAT

opcode

condition codes

N $\leftarrow$ N;
Z $\leftarrow$ Z;
V $\leftarrow$ V;
C $\leftarrow$ C;

exceptions

None.

opcodes

05	RSB	Return From Subroutine

DESCRIPTION

The PC is replaced by a longword popped from the stack.

Notes

1. Use RSB to return from subroutines called by the BSBB, BSBW, and JSB instructions.
2. RSB is equivalent to JMP @(SP)+, but is one byte shorter.

SOBGEQ

Subtract One and Branch Greater Than or Equal

FORMAT

opcode *index.ml, displ.bb*

condition codes

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

exceptions

integer overflow

opcodes

F4	SOBGEQ	Subtract One and Branch Greater Than or Equal

DESCRIPTION

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

Notes

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

2 The C-bit is unaffected.

SOBGTR

Subtract One and Branch Greater Than

FORMAT

opcode *index.ml, displ.bb*

condition codes

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

exceptions

integer overflow

opcodes

F5	SOBGTR	Subtract One and Branch Greater Than

DESCRIPTION

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

Notes

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

9.7 Procedure Call Instructions

The following three instructions implement a standard procedure calling interface:

- CALLG
- CALLS
- RET

CALLG and CALLS call the procedure. The RETURN instruction returns from the procedure. Refer to the *Introduction to VMS System Routines* for the procedure calling standard.

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 instruction, 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 called the *entry mask* followed by the procedure's instructions. The procedure terminates by executing a RET instruction.

The entry mask specifies the register use and overflow enables of the subprocedure.

```
1 1 1 1 1
5 4 3 2 1                   0
+-+-+---+-----------------------+
|D|I|MBZ|      REGISTERS        |
|V|V|   |                       |
+-+-+---+-----------------------+
```

At the occurrence of one of the call instructions, 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. 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, the 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 (see the VAX Procedure Calling and Condition Handling Standard in the *Introduction to VMS System Routines*). 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.

To preserve the state, the CALL instructions form a structure on the stack termed a *call frame* or *stack frame*. The call frame 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. The system uses this longword to implement the VMS condition handling facility (see the VAX Procedure Calling and Condition Handling Standard in the *Introduction to VMS System Routines*). 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 the restore state. The condition handling facility assumes that FP always points to the stack frame.

The stack frame has the following format:

```
+-------------------------------------------------------------+
|              condition handler (initially 0)                | :(FP)
+---+-+-+-----------------------+-------------------+---------+
|SPA|S|0|        mask<11:0>     |  saved PSW<15:5>  |    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.
```

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

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

The following instructions are described in this section.

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

CALLG

Call Procedure With General Argument List

FORMAT

opcode *arglist.ab, dst.ab*

condition codes

N ⟵ 0;
Z ⟵ 0;
V ⟵ 0;
C ⟵ 0;

exceptions

reserved operand

opcodes

FA	CALLG	Call Procedure with General Argument List

DESCRIPTION

The SP is saved in a temporary register. Bits 1:0 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 numbers correspond to set bits in the mask are pushed on the stack as longwords. The PC, FP, and AP are pushed on the stack as longwords. The condition codes are cleared. A longword containing the saved two low bits of the SP in bits 31:30, a 0 in bits 29 and 28, the low 12 bits of the procedure entry mask in bits 27:16, and the PSW in bits 15:0 with T cleared are pushed on the stack. A longword 0 is pushed on the stack. The FP is replaced by the SP. The AP is replaced by the **arglist** operand. The trap enables in the PSW are set to a known state. Integer overflow and decimal overflow are affected according to bits 14 and 15 of the entry mask, respectively; floating underflow is cleared. The T-bit is unaffected. The PC is replaced by the sum of destination operand plus 2, which transfers control to the called procedure at the byte beyond the entry mask.

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

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

Notes

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

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

3 The procedure calling standard and the condition handling facility require the following register saving conventions:

- R0 and R1 are always available for function return values and are never saved in the entry mask.
- All registers R2 through R11 that are modified in the called procedure must be preserved in the mask.

Refer to the VAX Procedure Calling and Condition Handling Standard in the *Introduction to VMS System Routines*.

CALLS

Call Procedure with Stack Argument List

FORMAT

opcode *numarg.rl, dst.ab*

condition codes

N ← 0;
Z ← 0;
V ← 0;
C ← 0;

exceptions

reserved operand

opcodes

FB	CALLS	Call Procedure with Stack Argument List

DESCRIPTION

The **numarg** operand is pushed on the stack as a longword (byte 0 contains the number of arguments; DIGITAL software uses the high-order 24 bits). The SP is saved in a temporary register, and then bits 1:0 of the 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 numbers correspond to set bits in the mask are pushed on the stack. The PC, FP, and AP are pushed on the stack as longwords. The condition codes are cleared. A longword containing the saved two low bits of the SP in bits 31:30, a 1 in bit 29, a 0 in bit 28, the low 12 bits of the procedure entry mask in bits 27:16, and the PSW in bits 15:0 with T cleared is pushed on the stack. A longword 0 is pushed on the stack. The FP is replaced by the SP. The AP is set to the value of the stack pointer after the **numarg** operand was pushed on the stack. The trap enables in the PSW are set to a known state. Integer overflow and decimal overflow are affected according to bits 14 and 15 of the entry mask, respectively. Floating underflow is cleared. T-bit is unaffected. The PC is replaced by the sum of destination operand plus 2, which transfers control to the called procedure at the byte beyond the entry mask. The appearance of the stack after CALLS is executed is:

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

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

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

Notes

1. If bits 13:12 of the entry mask are not 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.

   Refer to the VAX Procedure Calling and Condition Handling Standard in the *Introduction to VMS System Routines.*

RET

Return from Procedure

FORMAT

opcode

condition codes

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

exceptions

reserved operand

opcodes

04	RET	Return from Procedure

DESCRIPTION

The SP is replaced by the FP plus 4. A longword containing stack alignment bits in bits 31:30, a CALLS/CALLG flag in bit 29, the low 12 bits of the procedure entry mask in bits 27:16, and a saved PSW in bits 15:0 is popped from the stack and saved in a temporary. The 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 numbers are indicated by set bits in the mask are replaced by longwords popped from the stack. The SP is incremented by 31:30 of the temporary. The 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 the SP, and the 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 which return a function value or a status code do so in R0, or R0 and R1. Refer to the VAX Procedure Calling and Condition Handling Standard in the *Introduction to VMS System Routines.*

9.8 Miscellaneous Instructions

The following instructions are described in this section.

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

BICPSW

Bit Clear PSW

FORMAT

opcode *mask.rw*

condition codes

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

exceptions

reserved operand

opcodes

B9	BICPSW	Bit Clear PSW

DESCRIPTION

The result of the logical AND on PSW and the one's complement of the mask operand replaces PSW.

Note

A reserved operand fault occurs if **mask** <15:8> is not 0. On a reserved operand fault, the PSW is not affected.

BISPSW

Bit Set PSW

FORMAT

opcode *mask.rw*

condition codes

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

exceptions

reserved operand

opcodes

B8	BISPSW	Bit Set PSW

DESCRIPTION

The result of the logical OR on PSW and the mask operand replaces PSW.

Note

A reserved operand fault occurs if **mask** <15:8> is not 0. On a reserved operand fault, the PSW is not affected.

BPT

Breakpoint Fault

FORMAT

opcode

condition codes

N ⟵ 0; ! Condition codes cleared after BPT fault
Z ⟵ 0;
V ⟵ 0;
C ⟵ 0;

exceptions

None.

opcodes

03	BPT	Breakpoint Fault

DESCRIPTION

To understand the operation of this instruction, refer to Appendix E. This instruction, together with the T-bit, is used to implement debugging facilities.

HALT

Halt

FORMAT

opcode

condition codes

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

N ← N; ! If processor halt
Z ← Z;
V ← V;
C ← C;

exceptions

privileged instruction

opcodes

00	HALT	Halt

DESCRIPTION

If the process is running in kernel mode, the processor is halted. Otherwise, a privileged instruction fault occurs. For information about privileged instruction faults, refer to Appendix E.

Note

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*

condition codes

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

exceptions

subscript range

opcodes

0A	INDEX	index

DESCRIPTION

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

Notes

1. No arithmetic exception other than subscript range can result from this instruction. Therefore, no indication is given if overflow occurs in either the add or the 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 that is not readily absorbed by address arithmetic. The following notes show some of the uses of INDEX.

3 The following example shows a sequence of COBOL statements and the VAX MACRO code their compilation might generate:

```
COBOL:

01  A-ARRAY.
    02  A PIC X(10) OCCURS 15 TIMES.

01  B PIC X(10).
    MOVE A(I) TO B.

MACRO:

INDEX I, #1, #15, #10, #0, RO

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

4 The following example shows a sequence of PL/I statements and the VAX MACRO code their compilation might generate:

```
PL/I:

DCL A(-3:10) BIT (5);
A(I) = 1;

MACRO:

INDEX I, #-3, #10, #5, #3, RO

INSV  #1, RO, #5, A ; Assumes A is byte aligned
```

5 The following example shows a sequence of FORTRAN statements and the VAX MACRO code their compilation might generate:

```
FORTRAN:

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

MACRO:

INDEX J, #L2, #U2, #M1, #0, RO; M1=U1-L1+1
INDEX I, #L1, #U1, #1, RO, RO;
MOVL     #1, A-a[RO]; a = {{L2*M1} + L1} *4
```

MOVPSL

Move from PSL

FORMAT

opcode *dst.wl*

condition codes

N ← N;
Z ← Z;
V ← V;
C ← C;

exceptions

None.

opcodes

DC	MOVPSL	Move from PSL

DESCRIPTION

The destination operand is replaced by PSL.

NOP

No Operation

FORMAT

opcode

condition codes

$N \leftarrow N;$
$Z \leftarrow Z;$
$V \leftarrow V;$
$C \leftarrow C;$

exceptions

None.

opcodes

01	NOP	No Operation

DESCRIPTION

No operation is performed.

POPR

Pop Registers

FORMAT

opcode *mask.rw*

condition codes

N ⟵ N;
Z ⟵ Z;
V ⟵ V;
C ⟵ C;

exceptions

None.

opcodes

BA	POPR	Pop Registers

DESCRIPTION

The contents of registers whose numbers correspond 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*

condition codes

N ⟵ N;
Z ⟵ Z;
V ⟵ V;
C ⟵ C;

exceptions

None.

opcodes

BB	PUSHR	Push Registers

DESCRIPTION

The contents of registers whose numbers correspond 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.

Note

The order of pushing is specified so that the contents of higher-numbered registers are stored at higher memory addresses. An example of a result of this would be a double-floating datum stored in adjacent registers being stored by PUSHR in memory in the correct order.

XFC

Extended Function Call

FORMAT

opcode

condition codes

N ← 0;
Z ← 0;
V ← 0;
C ← 0;

exceptions

None.

opcodes

FC	XFC	Extended Function Call

DESCRIPTION

To understand the operation of this instruction, refer to Appendix E and the *VAX Architecture Reference Manual*. This instruction provides for customer-defined extensions to the instruction set.

9.9 Queue Instructions

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

A queue is classified by the type of link that it uses. The VAX supports two distinct types of links: absolute and self-relative.

9.9.1 Absolute Queues

Absolute queues use absolute addresses as links. Queue entries are linked by a pair of longwords. The first (lowest-addressed) longword is the forward link; it is the address of the succeeding queue entry. The second (highest-addressed) longword is the backward link; it is 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 called the *head* of the queue. The backward link of the header is the address of the entry termed the *tail* of the queue. The forward link of the tail points to the header.

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

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

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

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

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

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

If an entry at address A is inserted at the head of the queue, the queue appears as follows:

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

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

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

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

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

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

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

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

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

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. In the preceding example with the queue containing entries A,B, and C, the entry at address B can be removed, giving the following:

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

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

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

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

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

9.9.2 Self-Relative Queues

Self-relative queues use displacements from queue entries as links. Queue entries are linked by a pair of longwords. The first (lowest addressed) longword is the forward link; it is the displacement of the succeeding queue entry from the present entry. The second (highest-addressed) longword 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. The forward link of the header is the address of the entry called the *head* of the queue. The backward link of the header is the address of the entry called the *tail* of the queue. The forward link of the tail points to the header.

The following text contains examples of queue operations. An empty queue is specified by its header at address H. Because the queue is empty, the self-relative links must be 0, as shown.

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

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

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

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

If an entry at address A is inserted at the head of the queue, the queue appears as follows:

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

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

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

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

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

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

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

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

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

The following four instructions manipulate self-relative queues:

1. INSQHI - Insert entry into queue at head, interlocked.
2. INSQTI - Insert entry into queue at tail, interlocked.
3. REMQHI - Remove entry from queue at head, interlocked.
4. REMQTI - Remove entry from queue at tail, interlocked.

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; it is set when the queue is being accessed. If an interlocked queue instruction encounters the secondary interlock set, it terminates after setting the condition codes to indicate failure to gain access to the queue. If the secondary interlock bit is not set, then the interlocked queue instruction sets it during its operation and clears it at instruction completion. In this way, other interlocked queue instructions are prevented from operating on the same queue.

9.9.3 Instruction Descriptions

The following instructions are described in this section:

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

INSQHI

Insert Entry into Queue at Head, Interlocked

FORMAT

opcode *entry.ab, header.aq*

condition codes

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

exceptions

reserved operand

opcodes

5C	INSQHI	Insert Entry into Queue at Head, Interlocked

DESCRIPTION

The entry specified by the entry operand is inserted into the queue following the header. If the entry inserted was the first one in the queue, the condition code Z-bit is set; otherwise it is cleared. The insertion is a noninterruptible 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 method ensures that if a memory management exception occurs (see Appendix E), the queue is left in a consistent state. If the instruction fails to acquire the secondary interlock, the instruction sets condition codes and terminates.

Notes

1. Because the insertion is noninterruptible, processes running in kernel mode can share queues with interrupt service routines.
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, you can use the following:

```
INSERT:
        INSQHI  ...             ; Was queue empty?
        BEQL    1$              ; Yes
        BCS     INSERT          ; Try inserting again
        CALL    WAIT(...)       ; No, 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. A reserved operand fault occurs if **entry** or **header** is an address that is not quadword aligned (that is, <2:0> NEQU 0) or if **header** <2:1> is not 0. A reserved operand fault also occurs if **header** equals **entry**. In this case, the queue is not altered.

INSQTI

Insert Entry into Queue at Tail, Interlocked

FORMAT

opcode entry.ab, header.aq

condition codes

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

exceptions

reserved operand

opcodes

5D	INSQTI	Insert Entry into Queue at Tail, Interlocked

DESCRIPTION

The entry specified by the entry operand is inserted into the queue preceding the header. If the entry inserted was the first one in the queue, the condition code Z-bit is set; otherwise it is cleared. The insertion is a noninterruptible 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 method ensures that if a memory management exception occurs (see Appendix E), queue is left in a consistent state. If the instruction fails to acquire the secondary interlock, the instruction sets condition codes and terminates.

Notes

1. Because the insertion is noninterruptible, processes running in kernel mode can share queues with interrupt service routines.
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, you can use the following:

```
INSERT:
        INSQHI  ...             ; Was queue empty?
        BEQL    1$              ; Yes
        BCS     INSERT          ; Try inserting again
        CALL    WAIT(...)       ; No, 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. A reserved operand fault occurs if **entry**, **header**, or (**header**+4) is an address that is not quadword aligned (that is, <2:0> NEQU 0) or if **header** <2:1> is not 0. A reserved operand fault also occurs if **header** equals **entry**. In this case, the queue is not altered.

INSQUE

Insert Entry in Queue

FORMAT

opcode entry.ab, pred.ab

condition codes

N ← (entry) LSS (entry+4);
Z ← (entry) EQL (entry+4); ! First entry in queue
V ← 0;
C ← (entry) LSSU (entry+4);

exceptions

None.

opcodes

0E	INSQUE	Insert Entry in Queue

DESCRIPTION

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

Notes

1 The following three types of insertion can be performed by appropriate choice of the 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 noninterruptible, processes running in kernel mode can share queues with interrupt service routines.

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, you can use the following:

```
    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

condition codes

```
if {removal succeeded} then
    begin
    N ⟵ 0;
    Z ⟵ (header) EQL 0; ! Queue empty after removal
    V ⟵ {queue empty before this instruction};
    C ⟵ 0;
    end;
else
    begin
    N ⟵ 0;
    Z ⟵ 0;
    V ⟵ 1; ! Did not remove anything
    C ⟵ 1; ! Secondary interlock failed
    end;
```

exceptions

reserved operand

opcodes

5E	REMQHI	Remove Entry from Queue at Head, Interlocked

DESCRIPTION

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 it is cleared. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process even in a multiprocessor environment. The removal is a noninterruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs (see Appendix E), the queue is left in a consistent state. If the instruction fails to acquire the secondary interlock, the instruction sets condition codes and terminates without altering the queue.

Notes

1 Because the removal is noninterruptible, processes running in kernel mode can share queues with interrupt service routines.

2 The INSQHI, INSQTI, REMQHI, and REMQTI instructions are implemented so 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, you can use the following:

```
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, you can use the following:

```
1$:     REMQHI  ...             ; Anything removed?
        BVS     2$              ; Nc
         .
        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 A reserved operand fault occurs if **header** or (**header** + (**header**)) is an address that is not quadword aligned (that is, <2:0> NEQU 0) or if (**header**) <2:1> is not 0. A reserved operand fault also occurs if the header address operand equals the address of the **addr** operand. In this case, the queue is not altered.

REMQTI

Remove Entry from Queue at Tail, Interlocked

FORMAT

opcode header.aq, addr.wl

condition codes

```
if {removal succeeded} then
    begin
    N <— 0;
    Z <— (header + 4) EQL 0; ! Queue empty after removal
    V <— {queue empty before this instruction};
    C <— 0;
    end;
else
    begin
    N <— 0;
    Z <— 0;
    V <— 1; ! Did not remove anything
    C <— 1; ! Secondary interlock failed
    end;
```

exceptions

reserved operand

opcodes

5F	REMQTI	Remove Entry from Queue at Tail, Interlocked

DESCRIPTION

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 it is cleared. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, even in a multiprocessor environment. The removal is a noninterruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs (see Appendix E), the queue is left in a consistent state. If the instruction fails to acquire the secondary interlock, the instruction sets condition codes and terminates without altering the queue.

Notes

1 Because the removal is noninterruptible, processes running in kernel mode can share queues with interrupt service routines.

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

3 To release a software interlock realized with a queue, you can use the following:

```
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, you can use the following:

```
1$:     REMQTI  ...             ; Anything removed?
        BVS     2$              ; No
         .
        process removed entry
         .
        BR      1$              ;
         .
2$:     BCS     1$              ; Try removing again
        queue empty
```

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

6 A reserved operand fault occurs if **header**, (**header** + 4), or (**header** + (**header** + 4)+4) is an address that is not quadword aligned (that is, <2:0> NEQU 0), or if (**header**) <2:1> is not 0. A reserved operand fault also occurs if the header address operand equals the address of the **addr** operand. In this case, the queue is not altered.

REMQUE

Remove Entry From Queue

FORMAT

opcode *entry.ab,addr.wl*

condition codes

N ⟵ (entry) LSS (entry+4);
Z ⟵ (entry) EQL (entry+4); ! Queue empty
V ⟵ (entry) EQL (entry+4); ! No entry to remove
C ⟵ (entry) LSSU (entry+4);

exceptions

None.

opcodes

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 it is cleared. The removal is a noninterruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs (see Appendix E), the queue is left in a consistent state.

Notes

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

- Remove at head:

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

- Remove at tail:

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

- Remove arbitrary entry:

```
REMQUE  entry,addr
```

2 Because the removal is noninterruptible, processes running in kernel mode can share queues with interrupt service routines.

3 The INSQUE and REMQUE instructions are implemented so 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, you can use the following:

```
        REMQUE  ...             ; Queue empty?
        BEQL    1$              ; Yes
        CALL    ACTIVATE(...)   ; Activate other waiters
1$:
```

5 To remove entries until the queue is empty, you can use the following:

```
1$:     REMQUE  ...             ; Anything removed?
        BVS     EMPTY           ; No
          .
          .
          .
        BR      1$
```

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

9.10 Floating Point Instructions

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

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

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

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

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

9.10.1 Introduction

Mathematically, a floating-point number may be defined as having the following form: $(+or-)(2**K)*f$

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

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

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

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

1. Nonzero floating-point numbers:

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

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

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

In the G_floating data type, 11 bits are reserved for the storage of the exponent in excess 1024 notation. In the H_floating data type, 15 bits are reserved for the storage of the exponent in excess 16,384 notation. A biased exponent of 0 is reserved for floating-point 0. Thus, exponents are restricted to −1023 to +1023 inclusive (in excess notation, 1 to 2047), and −16,383 to +16,383 inclusive (in excess notation, 1 to 32,767) for G_floating and H_floating data types, respectively.

2 Floating-point 0:

Because of the hidden bit, the fractional factor is not available to distinguish between zero and nonzero numbers whose fractional factor is exactly 1/2. Therefore, the VAX reserves a sign-exponent field of 0 for this purpose. Any positive floating-point number with a 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 0, and this is the format generated by all floating-point instructions for which the result is 0.

3 The reserved operands:

A reserved operand is defined to be any bit pattern with a sign bit of 1 and a biased exponent of 0. On the VAX, all floating-point instructions generate a fault if a reserved operand is encountered. A reserved operand is never generated as a result of a floating-point instruction.

9.10.2 Overview of the Instruction Set

The VAX has the standard arithmetic operations ADD, SUB, MUL, and DIV implemented for all four floating-point data types. The results of these operations are always rounded, as described in 9.10.3. In addition, VAX has two composite operations, EMOD and POLY, also implemented for all four floating-point data types. EMOD generates a product of two operands and then separates the product into its integer and fractional terms. POLY evaluates a polynomial, given the degree, the argument, and a pointer to a table of coefficients. Details on the operation of EMOD and POLY are given in their respective descriptions. All of these instructions are subject to the rounding errors associated with floating-point operations, as well as to exponent overflow and underflow. Accuracy is discussed in Section 9.10.3. Exceptions are discussed in Appendix E.

The VAX architecture also has a complete set of instructions for conversion from integer arithmetic types (byte, word, longword) to all floating types (F_floating, D_floating, G_floating, H_floating), and vice versa. The VAX 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 9.10.3. However, a few may generate rounding error, floating overflow, or floating underflow, or induce integer overflow. Details are given in the description of the CVT instructions.

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

All of the floating-point instructions on the VAX fault if they encounter a reserved operand. Floating-point instructions also fault on the occurrence of floating overflow or divide by 0, and the condition codes are UNPREDICTABLE. The FU bit in the PSW is available to enable or disable an exception on underflow. If the FU bit is clear, no exception occurs on underflow and 0 is returned as the result. If the FU bit is set, a fault occurs on underflow. Further details on the actions taken if any of these exceptions occurs are included in the descriptions of the instructions and discussed in Appendix E.

9.10.3 Accuracy

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

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

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

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

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

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

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

- If a stored result is exact:
  - *roundedvalue = choppedvalue = truevalue*
- If a stored result is not exact:
  - Its magnitude is always less than that of the true result for chopping.
  - Its magnitude is always less than that of the true result for rounding if the rounding bit is 0.
  - Its magnitude is greater than that of the true result for rounding if the rounding bit is 1.

9.10.4 Instruction Descriptions

The following instructions are described in this section:

	Description and Opcode	Number of Instructions
1.	Add 2 Operand ADD{F,D,G,H}2 add.rx, sum.mx	4
2.	Add 3 Operand ADD{F,D,G,H}3 add1.rx, add2.rx, sum.wx	4
3.	Clear CLR{L=F,Q=D=G,O=H} dst.wx	3
4.	Compare CMP{F,D,G,H} src1.rx, src2.rx	4
5.	Convert CVT{F,D,G,H}{B,W,L,F,D,G,H} src.rx, dst.wy CVT{B,W,L}{F,D,G,H} src.rx, dst.wy All pairs except FF,DD,GG,HH,DG, and GD	34
6.	Convert Rounded CVTR{F,D,G,H}L src.rx, dst.wl	4
7.	Divide 2 Operand DIV{F,D,G,H}2 divr.rx, quo.mx	4
8.	Divide 3 Operand DIV{F,D,G,H}3 divr.rx, divd.rx, quo.wx	4
9.	Extended Modulus EMOD{F,D} mulr.rx, mulrx.rb, muld.rx, int.wl, fract.wx EMOD{G,H} mulr.rx, mulrx.rw, muld.rx, int.wl, fract.wx	4

	Description and Opcode	Number of Instructions
10.	Move Negated MNEG{F,D,G,H} src.rx, dst.wx	4
11.	Move MOV{F,D,G,H} src.rx, dst.wx	4
12.	Multiply 2 Operand MUL{F,D,G,H}2 mulr.rx, prod.mx	4
13.	Multiply 3 Operand MUL{F,D,G,H}3 mulr.rx, muld.rx, prod.wx	4
14.	Polynomial Evaluation F_floating POLYF arg.rf, degree.rw, tbladdr.ab, {R0-3.wl}	1
15.	Polynomial Evaluation D_floating POLYD arg.rd, degree.rw, tbladdr.ab, {R0-5.wl}	1
16.	Polynomial Evaluation G_floating POLYG arg.rg, degree.rw, tbladdr.ab, {R0-5.wl}	1
17.	Polynomial Evaluation H_floating POLYH arg.rh, degree.rw, tbladdr.ab, {R0-5.wl,-16(SP):-1(SP).wb}	1
18.	Subtract 2 Operand SUB{F,D,G,H}2 sub.rx, dif.mx	4
19.	Subtract 3 Operand SUB{F,D,G,H}3 sub.rx, min.rx, dif.wx	4
20.	Test TST{F,D,G,H} src.rx	4

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

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

ADD

Add

FORMAT

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

condition codes

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

exceptions

floating overflow
floating underflow
reserved operand

opcodes

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

DESCRIPTION

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

Notes

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

CLR

Clear

FORMAT

opcode *dst.wx*

condition codes

N ⟵ 0;
Z ⟵ 1;
V ⟵ 0;
C ⟵ C;

exceptions

None.

opcodes

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

DESCRIPTION

The destination operand is replaced by 0.

Note

CLR*x* **dst** is equivalent to MOV*x* S^#0, **dst,** but is one byte shorter.

CMP

Compare

FORMAT

opcode *src1.rx, src2.rx*

condition codes

N	⟵	src1 LSS src2;
Z	⟵	src1 EQL src2;
V	⟵	0;
C	⟵	0;

exceptions

reserved operand

opcodes

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

DESCRIPTION

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

CVT

Convert

FORMAT

opcode *src.rx, dst.wy*

condition codes

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

exceptions

integer overflow
floating overflow
floating underflow
reserved operand

opcodes

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

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

DESCRIPTION

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

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 a floating overflow fault; the destination operand is unaffected, and the condition codes are UNPREDICTABLE.
2. Only converts with a floating-point source operand can result in a reserved operand fault. On a reserved operand fault, the destination operand is unaffected, and the condition codes are UNPREDICTABLE.
3. Only converts with an integer destination operand can result in integer overflow. On integer overflow, the destination operand is replaced by the low-order bits of the true result.
4. Only CVTGF, CVTHF, CVTHD, and CVTHG can result in floating underflow. If FU is set, a fault occurs. On a floating underflow fault, the destination operand is unaffected. If FU is clear, the destination operand is replaced by 0, and no exception occurs.

DIV

Divide

FORMAT

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

condition codes

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

exceptions

floating overflow
floating underflow
divide by 0
reserved operand

opcodes

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

DESCRIPTION

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

Notes

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

2 On floating underflow, if FU is set, a fault occurs. On a floating underflow fault, the quotient operand is unaffected. If FU is clear, the quotient operand is replaced by 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 0, the quotient operand, and condition codes are affected as in note 3.

EMOD

Extended Multiply and Integerize

FORMAT

EMODF and EMODD:

opcode mulr.rx, mulrx.rb, muld.rx, int.wl, fract.wx

EMODG and EMODH:

opcode mulr.rx, mulrx.rw, muld.rx, int.wl, fract.wx

condition codes

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

exceptions

integer overflow
floating underflow
reserved operand

opcodes

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

DESCRIPTION

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

Notes

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

2 On floating underflow, if FU is set, a fault occurs. On a floating underflow fault, the integer and fraction parts are unaffected. If FU is clear, the integer and fraction parts are replaced by 0, and no exception occurs.

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

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

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

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

MNEG

Move Negated

FORMAT

opcode src.rx, dst.wx

condition codes

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

exceptions

reserved operand

opcodes

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

DESCRIPTION

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

MOV

Move

FORMAT

opcode *src.rx, dst.wx*

condition codes

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

exceptions

reserved operand

opcodes

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

DESCRIPTION

The destination operand is replaced by the source operand.

Note

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

MUL

Multiply

FORMAT

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

condition codes

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

exceptions

floating overflow
floating underflow
reserved operand

opcodes

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

DESCRIPTION

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

Notes

1. On a reserved operand fault, the product operand is unaffected, and the condition codes are UNPREDICTABLE.
2. On floating underflow, if FU is set, a fault occurs. On a floating underflow fault, the product operand is unaffected. If FU is clear, the product operand is replaced by 0, and no exception occurs.
3. On floating overflow, the instruction faults, the product operand is unaffected, and the condition codes are UNPREDICTABLE.

POLY

Polynomial Evaluation

FORMAT

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

condition codes

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

exceptions

floating overflow
floating underflow
reserved operand

opcodes

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

DESCRIPTION

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

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

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

Notes

1 After execution:

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

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

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

2 On a floating fault:

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

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

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

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

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

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

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

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

5 On a reserved operand fault:

- 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 FU is set. If FU is clear, the temporary result (**tmp3**) is replaced by 0 and the operation continues. In this case, the final result may be nonzero if underflow occurred before the last iteration.

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

8 If the argument is 0 and one of the coefficients in the table is the reserved operand, whether a reserved operand fault occurs is UNPREDICTABLE.

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

EXAMPLE

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

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

SUB

Subtract

FORMAT

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

condition codes

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

exceptions

floating overflow
floating underflow
reserved operand

opcodes

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

DESCRIPTION

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

Notes

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

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

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

TST

Test

FORMAT

opcode *src.rx*

condition codes

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

exceptions

reserved operand

opcodes

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

DESCRIPTION

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

Notes

1. TST*x* **src** is equivalent to CMP*x* **src,** #0, but is 5 (F_floating) or 9 (D_floating or G_floating) or 17 (H_floating) bytes shorter.
2. On a reserved operand fault, the condition codes are UNPREDICTABLE.

9.11 Character String Instructions

A character string is specified by two operands:

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

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

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

```
+------------------------------+------------------------------+
|                              |           LENGTH 1           | : R0
+------------------------------+------------------------------+
|                          ADDRESS 1                          | : R1
+------------------------------+------------------------------+
|                              |           LENGTH 2           | : R2
+------------------------------+------------------------------+
|                          ADDRESS 2                          | : R3
+------------------------------+------------------------------+
|                              |           LENGTH 3           | : R4
+------------------------------+------------------------------+
|                          ADDRESS 3                          | : R5
+-------------------------------------------------------------+
```

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

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

The following instructions are described in this section.

	Description and Opcode	Number of Instructions
1.	Compare Characters 3 Operand CMPC3 len.rw, src1addr.ab, src2addr.ab, {R0-3.wl}	1
2.	Compare Characters 5 Operand CMPC5 src1len.rw, src1addr.ab, fill.rb, src2len.rw, src2addr.ab, {R0-3.wl}	1
3.	Locate Character LOCC char.rb, len.rw, addr.ab, {R0-1.wl}	1
4.	Match Characters MATCHC len1.rw, addr1.ab, len2.rw, addr2.ab, {R0-3.wl}	1
5.	Move Character 3 Operand MOVC3 len.rw, srcaddr.ab, dstaddr.ab, {R0-5.wl}	1
6.	Move Character 5 Operand MOVC5 srclen.rw, srcaddr.ab, fill.rb, dstlen.rw, dstaddr.ab, {R0-5.wl}	1
7.	Move Translated Characters MOVTC srclen.rw, srcaddr.ab, fill.rb, tbladdr.ab, dstlen.rw, dstaddr.ab, {R0-5.wl}	1
8.	Move Translated Until Character MOVTUC srclen.rw, srcaddr.ab, esc.rb, tbladdr.ab, dstlen.rw, dstaddr.ab, {R0-5.wl}	1
9.	Scan Characters SCANC len.rw, addr.ab, tbladdr.ab, mask.rb, {R0-3.wl}	1
10.	Skip Character SKPC char.rb, len.rw, addr.ab, {R0-1.wl}	1
11.	Span Characters SPANC len.rw, addr.ab, tbladdr.ab, mask.rb, {R0-3.wl}	1

CMPC

Compare Characters

FORMAT

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

condition codes

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

exceptions

None.

opcodes

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

DESCRIPTION

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

Notes

1 After execution of CMPC3:

R0 =	number of bytes remaining in string1 (including byte that terminated comparison); R0 is 0 only if strings are equal
R1 =	address of the byte in string1 that terminated comparison; if strings are equal, address of one byte beyond string1
R2 =	R0
R3 =	address of the byte in string2 that terminated comparison; if strings are equal, address of one byte beyond string2

2 After execution of CMPC5:

R0 =	number of bytes remaining in string1 (including byte that terminated comparison); R0 is 0 only if string1 and string2 are of equal length and equal or string1 was exhausted before comparison terminated
R1 =	address of the byte in string1 that terminated comparison; if comparison did not terminate before string1 exhausted, address of one byte beyond string1
R2 =	number of bytes remaining in string2 (including byte that terminated comparison); R2 is 0 only if string2 and string1 are of equal length or string2 was exhausted before comparison terminated
R3 =	address of the byte in string2 that terminated comparison; if comparison did not terminate before string2 was exhausted, address of one byte beyond string2

3 If both strings have 0 length, condition code Z is set and N, V, and C are cleared just as in the case of two equal strings.

LOCC

Locate Character

FORMAT

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

condition codes

N	⟵ 0;
Z	⟵ R0 EQL 0;
V	⟵ 0;
C	⟵ 0;

exceptions

None.

opcodes

3A	LOCC	Locate Character

DESCRIPTION

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

Notes

1 After execution:

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

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

2 If the string has 0 length, condition code Z is set just as though each byte of the entire string were unequal to character.

MATCHC

Match Characters

FORMAT

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

condition codes

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

exceptions

None.

opcodes

39	MATCHC	Match Characters

DESCRIPTION

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

Notes

1 After execution:

R0 = if a match occurred, 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 0 length, or if the object string has 0 length, condition code Z is set, and registers R0 through R3 are left just as though the substring were found.

3 If the source string has 0 length and the object string has nonzero length, condition code Z is cleared, and registers R0 through R3 are left just as though the substring were not found.

MOVC

Move Character

FORMAT

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

condition codes

N ← 0; !MOVC3
Z ← 1;
V ← 0;
C ← 0;

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

exceptions

None.

opcodes

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

DESCRIPTION

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

Notes

1 After execution of MOVC3:

R0 =	0
R1 =	address of 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 nonzero only if source string is longer than destination string
R1 =	address of one byte beyond the last byte in source string that was moved
R2 =	0
R3 =	address of one byte beyond the destination string
R4 =	0
R5 =	0

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

4 MOVC5 with a 0 source length operand is the preferred way to fill a block of memory with the fill character.

MOVTC

Move Translated Characters

FORMAT

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

condition codes

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

exceptions

None.

opcodes

2E	MOVTC	Move Translated Characters

DESCRIPTION

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

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

Notes

1 After execution:

R0 =	number of untranslated bytes remaining in source string; R0 is nonzero only if source string is longer than destination string
R1 =	address of one byte beyond the last byte in source string that was translated
R2 =	0
R3 =	address of the translation table
R4 =	0
R5 =	address of one byte beyond the destination string

MOVTUC

Move Translated Until Character

FORMAT

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

condition codes

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

exceptions

None.

opcodes

2F	MOVTUC	Move Translated Until Character

DESCRIPTION

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

If the destination string overlaps the table, the destination string and registers R0 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 0 only if the entire source string was translated and moved without escape

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

R2 = 0

R3 = address of the table

R4 = number of bytes remaining in the destination string

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

SCANC

Scan Characters

FORMAT

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

condition codes

N ← 0;
Z ← R0 EQL 0;
V ← 0;
C ← 0;

exceptions

None.

opcodes

2A	SCANC	Scan Characters

DESCRIPTION

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

Notes

1 After execution:

R0 =	number of bytes remaining in the string (including the byte that produced the nonzero AND result); R0 is 0 only if there was no nonzero AND result
R1 =	address of the byte that produced the nonzero AND result; if no nonzero result, address of one byte beyond the string
R2 =	0
R3 =	address of the table

2 If the string has 0 length, condition code Z is set just as though the entire string were scanned.

SKPC

Skip Character

FORMAT

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

condition codes

N ⟵ 0;
Z ⟵ R0 EQL 0;
V ⟵ 0;
C ⟵ 0;

exceptions

None.

opcodes

3B	SKPC	Skip Character

DESCRIPTION

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

Notes

1. After execution:

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

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

2. If the string has 0 length, condition code Z is set just as though each byte of the entire string were equal to the character.

SPANC

Span Characters

FORMAT

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

condition codes

N ⟵ 0;
Z ⟵ R0 EQL 0;
V ⟵ 0;
C ⟵ 0;

exceptions

None.

opcodes

2B	SPANC	Span Characters

DESCRIPTION

The assembler successively uses the bytes of the string specified by the length and address operands to index into a 256-byte table whose first entry (entry number 0) address is specified by the table address operand. The logical AND is performed on the byte selected from the table and the mask operand. The operation continues until the result of the AND is 0, or until all the bytes of the string have been exhausted. If a 0 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 0 AND result); R0 is 0 only if there was no 0 AND result
R1 =	address of the byte that produced a 0 AND result; if no nonzero result, address of one byte beyond the string
R2 =	0
R3 =	address of the table

2 If the string has 0 length, the condition code Z-bit is set just as though the entire string were spanned.

9.12 Cyclic Redundancy Check Instruction

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

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

The CRC instruction scans the string and includes each byte of the data stream in the CRC being calculated. The instruction includes the byte of the data stream by performing a logical exclusive OR (XOR) with it and the rightmost eight bits of the CRC. Then the instruction shifts the CRC right one bit and inserts a 0 on the left. The instruction uses the rightmost bit of the CRC (lost by the shift) to control the logical XOR operation of the CRC polynomial with the resultant CRC. If the bit is a 1, the instruction performs a logical XOR with the polynomial and the CRC. The instruction again shifts the CRC to the right and performs a conditional logical XOR on the polynomial with the result, for a total of eight times. The actual algorithm used can shift by one, two, or four bits at a time using the appropriate entries in a specially constructed table. The instruction produces a 32-bit CRC. For shorter polynomials, the result must be extracted from the 32-bit field. The data stream must be either a multiple of eight bits in length or right-adjusted in the string with leading 0 bits.

CRC

Calculate Cyclic Redundancy Check

FORMAT

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

condition codes

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

exceptions

None.

opcodes

0B	CRC	Calculate Cyclic Redundancy Check

DESCRIPTION

The CRC of the data stream described by the string descriptor is calculated. The initial CRC is given by **inicrc**; it is normally 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 low-order bits of R0. The CRC polynomial is expressed by the contents of the 16-longword table. See the notes for the calculation of the table.

Notes

1 After execution:

R0 = result of CRC
R1 = 0
R2 = 0
R3 = address one byte beyond the end of the source string

2 If the data stream is not a multiple of eight bits, it must be right-adjusted with leading 0 fill.

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

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

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

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

```
      SUBROUTINE LIB$CRC_TABLE (POLY, TABLE)

      INTEGER*4 POLY, TABLE(0:15), TMP, X

      DO 190 INDEX = 0, 15

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

190   CONTINUE
      RETURN
      END
```

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

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

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

CCITT  (used in ADCCP, HDLC, SDLC)

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

AUTODIN-II

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

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

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

7 If the stream has 0 length, R0 receives the initial CRC.

9.13 Decimal String Instructions

Decimal string instructions operate on packed decimal strings.

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

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

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

A decimal string is specified by two operands:

1. For all decimal strings, the length is the number of digits in the string. The number of bytes in the string is a function of the length and the type of decimal string referenced (see Chapter 8).
2. The address of the lowest-addressed byte of the string. This byte contains the most significant digit for trailing numeric and packed decimal strings, as well as a sign for leading separate numeric strings. The address is specified by a byte operand of address access type.

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

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

```
3
1                                                               0
+---------------------------------------------------------------+
|                               0                               | : R0
+---------------------------------------------------------------+
|                           ADDRESS 1                           | : R1
+---------------------------------------------------------------+
|                               0                               | : R2
+---------------------------------------------------------------+
|                           ADDRESS 2                           | : R3
+---------------------------------------------------------------+
|                               0                               | : R4
+---------------------------------------------------------------+
|                           ADDRESS 3                           | : R5
+---------------------------------------------------------------+
```

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

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

9.13.1 Decimal Overflow

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

9.13.2 Zero Numbers

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

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

9.13.3 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 or CVTTP. The PC points to the opcode of the instruction causing the exception.

9.13.4 UNPREDICTABLE Results

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

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

9.13.5 Packed Decimal Operations

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

A packed decimal string contains an invalid nibble if:

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

9.13.6 Zero-Length Decimal Strings

The length of a packed decimal string can be 0. In this case, the value is 0 (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 do not occur when a zero-length trailing numeric operand is specified because no memory reference occurs. The value of a zero-length trailing numeric string is identically 0.

The length of a leading separate numeric string can be 0. In this case, one byte of storage is occupied by the sign. Memory is accessed when a zero-length operand is specified, and a reserved operand abort will occur if an invalid sign is detected. The value of a zero-length leading separate numeric string is 0.

9.13.7 Instruction Descriptions

The following instructions are described in this section:

	Description and Opcode	Number of Instructions
1.	Add Packed 4 Operand ADDP4 addlen.rw, addaddr.ab, sumlen.rw, sumaddr.ab, {R0-3.wl}	1
2.	Add Packed 6 Operand ADDP6 add1len.rw, add1addr.ab, add2len.rw, add2addr.ab, sumlen.rw, sumaddr.ab, {R0-5.wl}	1
3.	Arithmetic Shift and Round Packed ASHP cnt.rb, srclen.rw, srcaddr.ab, round.rb, dstlen.rw, dstaddr.ab, {R0-3.wl}	1

	Description and Opcode	Number of Instructions
4.	Compare Packed 3 Operand CMPP3 len.rw, src1addr.ab, src2addr.ab, {R0-3.wl}	1
5.	Compare Packed 4 Operand CMPP4 src1len.rw, src1addr.ab, src2len.rw, src2addr.ab, {R0-3.wl}	1
6.	Convert Long to Packed CVTLP src.rl, dstlen.rw, dstaddr.ab, {R0-3.wl}	1
7.	Convert Packed to Long CVTPL srclen.rw, srcaddr.ab, {R0-3.wl}, dst.wl	1
8.	Convert Packed to Leading Separate CVTPS srclen.rw, srcaddr.ab, dstlen.rw, dstaddr.ab, {R0-3.wl}	1
9.	Convert Packed to Trailing CVTPT srclen.rw, srcaddr.ab, tbladdr.ab, dstlen.rw, dstaddr.ab, {R0-3.wl}	1
10.	Convert Leading Separate to Packed CVTSP srclen.rw, srcaddr.ab, dstlen.rw, dstaddr.ab, {R0-3.wl}	1
11.	Convert Trailing to Packed CVTTP srclen.rw, srcaddr.ab, tbladdr.ab, dstlen.rw, dstaddr.ab, {R0-3.wl}	1
12.	Divide Packed DIVP divrlen.rw, divraddr.ab, divdlen.rw, divdaddr.ab, quolen.rw, quoaddr.ab, {R0-5.wl, −16(SP):−1(SP).wb}	1
13.	Move Packed MOVP len.rw, srcaddr.ab, dstaddr.ab, {R0-3.wl}	1
14.	Multiply Packed MULP mulrlen.rw, mulraddr.ab, muldlen.rw, muldaddr.ab, prodlen.rw, prodaddr.ab, {R0-5.wl}	1
15.	Subtract Packed 4 Operand SUBP4 sublen.rw, subaddr.ab, diflen.rw, difaddr.ab, {R0-3.wl}	1
16.	Subtract Packed 6 Operand SUBP6 sublen.rw, subaddr.ab, minlen.rw, minaddr.ab, diflen.rw, difaddr.ab, {R0-5.wl}	1

ADDP

Add Packed

FORMAT

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

condition codes

N $\longleftarrow$ {sum string} LSS 0;
Z $\longleftarrow$ {sum string} EQL 0;
V $\longleftarrow$ {decimal overflow};
C $\longleftarrow$ 0;

exceptions

reserved operand
decimal overflow

opcodes

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

DESCRIPTION

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

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

Notes

1 After execution of ADDP4:

R0 = 0

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

R2 = 0

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

2 After execution of ADDP6:

R0 =	0
R1 =	address of the byte containing the most significant digit of the addend1 string
R2 =	0
R3 =	address of the byte containing the most significant digit of the addend2 string
R4 =	0
R5 =	address of the byte containing the most significant digit of the sum string

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

ASHP

Arithmetic Shift and Round Packed

FORMAT

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

condition codes

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

exceptions

reserved operand
decimal overflow

opcodes

F8	ASHP	Arithmetic Shift and Round Packed

DESCRIPTION

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

A positive count operand effectively multiplies, a negative count effectively divides, and a 0 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 nonzero, or if bits 3:0 of the round operand contain an invalid packed decimal digit, the result is UNPREDICTABLE.

5 When the count operand is 0 or positive, the round operand has no effect on the result except as specified in note 4.

6 The round operand is normally 5. Truncation can be accomplished by using a 0 round operand.

CMPP

Compare Packed

FORMAT

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

condition codes

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

exceptions

reserved operand

opcodes

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

DESCRIPTION

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

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

Notes

1 After execution of CMPP3 or CMPP4:

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

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

CVTLP

Convert Long to Packed

FORMAT

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

condition codes

N	$\longleftarrow$ {dst string} LSS 0;
Z	$\longleftarrow$ {dst string} EQL 0;
V	$\longleftarrow$ {decimal overflow};
C	$\longleftarrow$ 0;

exceptions

reserved operand
decimal overflow

opcodes

F9	CVTLP	Convert Long to Packed

DESCRIPTION

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

Notes

1 After execution:

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

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

3 Overlapping operands produce correct results.

CVTPL

Convert Packed to Long

FORMAT

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

condition codes

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

exceptions

reserved operand
integer overflow

opcodes

36	CVTPL	Convert Packed to Long

DESCRIPTION

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

Notes

1 After execution:

R0 = 0

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

R2 = 0

R3 = 0

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

3 The destination operand is stored after the registers are updated as specified in note 1. You may use R0 through R3 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. On overflow, the sign of the destination may be different from the sign of the source.

5 Overlapping operands produce correct results.

CVTPS

Convert Packed to Leading Separate Numeric

FORMAT

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

condition codes

N $\leftarrow$ {src string} LSS 0;
Z $\leftarrow$ {src string} EQL 0;
V $\leftarrow$ {decimal overflow};
C $\leftarrow$ 0;

exceptions

reserved operand
decimal overflow

opcodes

08	CVTPS	Convert Packed to Leading Separate Numeric

DESCRIPTION

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

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

Notes

1 After execution:

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

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

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

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

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

CVTPT

Convert Packed to Trailing Numeric

FORMAT

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

condition codes

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

exceptions

reserved operand
decimal overflow

opcodes

24	CVTPT	Convert Packed to Trailing Numeric

DESCRIPTION

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

Conversion is effected by using the highest-addressed byte of the source string (the byte containing the sign and the least significant digit), even if the source string value is –0. The assembler uses this byte as an unsigned index into a 256-byte table whose first entry (entry number 0) address is specified by the table address operand. The byte read 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; if the source string or the table contains an invalid nibble; or if a reserved operand abort occurs.

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

4 By appropriate specification of the table, you can convert any form of trailing numeric string. See Chapter 8 for the preferred form of trailing overpunch, zoned and unsigned data. In addition, the table can be set up for absolute value, negative absolute value, or negated conversions. The translation table may be referenced even if the length of the destination string is 0.

5 Decimal overflow occurs if the destination string is too short to contain the converted result of a nonzero packed decimal source string (not including leading zeros). Conversion of a source string with 0 value never results in overflow; conversion of a nonzero source string to a zero-length destination string results in overflow.

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

CVTSP

Convert Leading Separate Numeric to Packed

FORMAT

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

condition codes

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

exceptions

reserved operand
decimal overflow

opcodes

09	CVTSP	Convert Leading Separate Numeric to Packed

DESCRIPTION

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

Notes

1 A reserved operand abort occurs if:

- The length of the source leading separate numeric string is outside the range 0 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 if a reserved operand abort occurs.

CVTTP

Convert Trailing Numeric to Packed

FORMAT

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

condition codes

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

exceptions

reserved operand
decimal overflow

opcodes

26	CVTTP	Convert Trailing Numeric to Packed

DESCRIPTION

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

Conversion is effected by using the highest-addressed (trailing) byte of the source string as an unsigned index into a 256-byte table whose first entry (entry number 0) is specified by the table address operand. The byte read-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:

- The length of the source trailing 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 value other than ASCII "0" through "9" in any high-order byte (that is, any byte except the least significant byte)
- The translation of the least significant digit produces an invalid packed decimal digit or sign nibble

2 After execution:

R0 = 0

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

R2 = 0

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

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

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

5 If the length of the source string is 0, the destination packed decimal string is set equal to 0, and the translation table is not referenced.

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

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

DIVP

Divide Packed

FORMAT

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

condition codes

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

exceptions

reserved operand
decimal overflow
divide by 0

opcodes

27	DIVP	Divide Packed

DESCRIPTION

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

Notes

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

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

- The absolute value of the remainder (which is lost) is less than the absolute value of the divisor
- The product of the absolute value of the quotient times the absolute value of the divisor is less than or equal to the absolute value of the dividend
- The sign of the quotient is determined by the rules of algebra from the signs of the dividend and the divisor; if the value of the quotient is 0, the sign is always positive

3 After execution:

R0 = 0

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

R2 = 0

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

R4 = 0

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

4 The quotient string, R0 through R5, and the condition codes are UNPREDICTABLE if: the quotient string overlaps the divisor or dividend strings; the divisor or dividend string contains an invalid nibble; the divisor is 0; or a reserved operand abort occurs.

MOVP

Move Packed

FORMAT

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

condition codes

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

exceptions

reserved operand

opcodes

34	MOVP	Move Packed

DESCRIPTION

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

Notes

1. After execution:

   R0 = 0

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

   R2 = 0

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

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

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

MULP

Multiply Packed

FORMAT

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

condition codes

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

exceptions

reserved operand
decimal overflow

opcodes

25	MULP	Multiply Packed

DESCRIPTION

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

Notes

1 After execution:

R0 = 0

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

R2 = 0

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

R4 = 0

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

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

SUBP

Subtract Packed

FORMAT

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

condition codes

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

exceptions

reserved operand
decimal overflow

opcodes

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

DESCRIPTION

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

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

Notes

1 After execution of SUBP4:

R0 = 0

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

R2 = 0

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

2 After execution of SUBP6:

R0 =	0
R1 =	address of the byte containing the most significant digit of the subtrahend string
R2 =	0
R3 =	address of the byte containing the most significant digit of the minuend string
R4 =	0
R5 =	address of the byte containing the most significant digit of the difference string

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

9.14 The EDITPC Instruction and Its Pattern Operators

The EDITPC instruction implements the common editing functions that occur when handling fixed-format output. The operation consists of converting an input packed decimal number to an output character string and generating characters for the output. When converting digits, options include filling in leading zeros, protecting leading zeros, insertion of floating sign, insertion of floating currency symbol, insertion of special sign representations, and blanking an entire field when it is 0. An example of this operation is a MOVE to a numeric edited (PICTURE) item in COBOL or PL/I. Many other applications are possible.

The operands to the EDITPC instruction are:

1 A *packed decimal string descriptor* (as input). This is a standard VAX operand pair consisting of the length of the decimal string in digits (up to 31) and the starting address of the string.
2 A *pattern* specification, consisting of the starting address of a pattern operation editing sequence. VAX MACRO interprets a pattern specification in the same way as it interprets normal instructions.
3 The *starting address of the output string*. The output string is described by its starting address only, because the pattern defines the length unambiguously.

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

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

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

After execution, the condition codes describe the following:

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

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

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

EDITPC

Edit Packed to Character String

FORMAT

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

condition codes

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

exceptions

reserved operand
decimal overflow

opcodes

38 EDITPC Edit Packed to Character String

DESCRIPTION

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

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

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

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

Notes

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

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

- The source string contains an invalid nibble.
- The EO$ADJUST_INPUT operand is outside the range 1 through 31.

- The source and destination strings overlap.
- The pattern and destination strings overlap.

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

R0 =	length of source string
R1 =	address of the byte containing the most significant digit of the source string
R2 =	0
R3 =	address of the byte containing the EO$END pattern operator
R4 =	0
R5 =	address of one byte beyond the last byte of the destination string

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

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

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

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

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

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

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

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

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

N =	{**src** has minus sign}
Z =	all source digits 0 so far
V =	nonzero digits lost
C =	significance
R0 <31:16> =	–(count of source zeros to supply)
R0 <15:0> =	remaining **srclen**
R1 =	current source location
R2 <31:16> =	implementation dependent
R2 <15:8> =	current contents of sign register
R2 <7:0> =	current contents of fill register
R3 =	location of edit pattern operator causing exception
R4 =	implementation dependent
R5 =	location of next destination byte

Table 9–1 Summary of EDITPC Pattern Operators

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

Key:

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

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

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

Key:

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

Table 9–2 EDITPC Pattern Operator Encoding

Hex	Symbol	Notes
00	EO$END	—
01	EO$END_FLOAT	—
02	EO$CLEAR_SIGNIF	—
03	EO$SET_SIGNIF	—
04	EO$STORE_SIGN	—
05 ... 1F	—	Reserved to DIGITAL
20 ... 3F	—	Reserved for all time
40	EO$LOAD_FILL	Character is in next byte
41	EO$LOAD_SIGN	Character is in next byte
42	EO$LOAD_PLUS	Character is in next byte
43	EO$LOAD_MINUS	Character is in next byte
44	EO$INSERT	Character is in next byte
45	EO$BLANK_ZERO	Unsigned length is in next byte
46	EO$REPLACE_SIGN	Unsigned length is in next byte
47	EO$ADJUST_INPUT	Unsigned length is in next byte
48 ... 5F	—	Reserved to DIGITAL
60 ... 7F	—	Reserved to CSS and customers
80,90,A0	—	Reserved to DIGITAL
81 ... 8F	EO$FILL	—
91 ... 9F	EO$MOVE	Repeat count is <3:0>

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

Hex	Symbol	Notes
A1 ... AF	EO$FLOAT	—
B0 ... FE	—	Reserved to DIGITAL
FF	—	Reserved for all time

EO$ADJUST_INPUT

Adjust Input Length

FORMAT

opcode *pattern len*

pattern operators

47	EO$ADJUST_INPUT	Adjust Input Length

DESCRIPTION

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

Note

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

EO$BLANK_ZERO

Blank Backwards when Zero

FORMAT

opcode *pattern len*

pattern operators

45	EO$BLANK_ZERO	Blank Backwards when Zero

DESCRIPTION

The EO$BLANK_ZERO pattern operator is followed by an unsigned byte integer length. If the value of the source string is 0, then the contents of the fill register are stored into the last length bytes of the destination string.

Notes

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

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

EO$END

End Edit

FORMAT

opcode *pattern*

pattern operators

00	EO$END	End Edit

DESCRIPTION

The EO$END pattern operator terminates the edit operation.

Notes

1. If there are still input digits, a reserved operand abort is taken.
2. If the source value is –0, the N condition code is cleared.

EO$END_FLOAT

End Floating Sign

FORMAT

opcode pattern

pattern operators

01	EO$END_FLOAT	End Floating Sign

DESCRIPTION

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

Note

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

EO$FILL

Store Fill

FORMAT

opcode *pattern r*

pattern operators

8x	EO$FILL	Store Fill

DESCRIPTION

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

Note

Use this pattern operator for fill (blank) insertion.

EO$FLOAT

Float Sign

FORMAT

opcode *pattern r*

pattern operators

Ax	EO$FLOAT	Float Sign

DESCRIPTION

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

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

Notes

1. If r is greater than the number of digits remaining in the source string, a reserved operand abort is taken.
2. Use this pattern operator to move digits with a floating arithmetic sign. The sign must already be set up as for EO$STORE_SIGN. A sequence of one or more EO$FLOATs can include intermixed EO$INSERTs and EO$FILLs. Significance must be clear before the first pattern operator of the sequence. The sequence must be terminated by one EO$END_FLOAT.
3. Use this pattern operator to move digits with a floating currency sign. The sign must already be set up with an EO$LOAD_SIGN. A sequence of one or more EO$FLOATs can include intermixed EO$INSERTs and EO$FILLs. Significance must be clear before the first pattern operator of the sequence. The sequence must be terminated by one EO$END_FLOAT.

EO$INSERT

Insert Character

FORMAT

opcode *pattern ch*

pattern operators

44	EO$INSERT	Insert Character

DESCRIPTION

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

Note

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

EO$LOAD_

Load Register

FORMAT

opcode *pattern ch*

pattern operators

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

DESCRIPTION

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

Notes

1. Use EO$LOAD_FILL to set up check protection (* instead of space).
2. Use EO$LOAD_SIGN to set up a floating currency sign.
3. Use EO$LOAD_PLUS to set up a nonblank plus sign.
4. Use EO$LOAD_MINUS to set up a nonminus minus sign (such as CR, DB, or the PL/I +).

EO$MOVE

Move Digits

FORMAT

opcode *pattern r*

pattern operators

9x	EO$MOVE	Move Digits

DESCRIPTION

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

The next digit is moved from the source to the destination. If the digit is nonzero, significance is set and zero is cleared. If the digit is not significant (that is, a leading 0), it is replaced by the contents of the fill register in the destination.

Notes

1. If *r* is greater than the number of digits remaining in the source string, a reserved operand abort is taken.
2. Use this pattern operator to move digits without a floating sign. If leading-zero suppression is desired, significance must be clear. If leading zeros should be explicit, significance must be set. A string of EO$MOVEs intermixed with EO$INSERTs and EO$FILLs will handle suppression correctly.
3. If check protection (*) is desired, EO$LOAD_FILL must precede the EO$MOVE.

EO$REPLACE_SIGN

Replace Sign when Zero

FORMAT

opcode *pattern len*

pattern operators

46	EO$REPLACE_SIGN	Replace Sign when Zero

DESCRIPTION

The EO$REPLACE_SIGN pattern operator is followed by an unsigned byte integer length. If the value of the source string is 0 (that is, if Z is set), the contents of the fill register are stored in the byte of the destination string that is **len** bytes before the current position.

Notes

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

2 You can use this pattern operator to correct a stored sign (EO$END_FLOAT or EO$STORE_SIGN) if a minus was stored and the source value turned out to be 0.

EO$_SIGNIF

Significance

FORMAT

opcode *pattern*

pattern operators

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

DESCRIPTION

The significance indicator is set or cleared. This controls the treatment of leading zeros (leading zeros are 0 digits for which the significance indicator is clear).

Notes

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

EO$STORE_SIGN

Store Sign

FORMAT

opcode *pattern*

pattern operators

04	EO$STORE_SIGN	Store Sign

DESCRIPTION

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

Note

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

9.15 Other VAX Instructions

The following table lists other VAX instructions:

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

PROBE*x*

Probe Accessibility

FORMAT

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

condition codes

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

exceptions

translation not valid

opcodes

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

DESCRIPTION

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

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

EXAMPLE

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

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

Notes

1. If the valid bit of the examined page table entry is set, it is UNPREDICTABLE whether the modify bit of the examined page table entry is set by a PROBER. If the valid bit is clear, the modify bit is not changed.
2. Except for note 1, above, the valid bit of the page table entry, PTE <31> , mapping the probed address is ignored.
3. A length violation gives a status of "not-accessible."
4. On the probe of a process virtual address, if the valid bit of the system page table entry is 0, a Translation Not Valid Fault occurs. This allows for the demand paging of the process page tables.
5. On the probe of a process virtual address, if the protection field of the system page table entry indicates No Access, a status of "not-accessible" is given. Thus, a single No Access page table entry in the system map is equivalent to 128 No Access page table entries in the process map.

CHM

Change Mode

FORMAT

opcode *code.rw*

condition codes

N ← 0;
Z ← 0;
V ← 0;
C ← 0;

exceptions

halt

opcodes

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

DESCRIPTION

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

A change in mode also results in a change of stack pointers; the old pointer is saved, and the new pointer is loaded. The 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 CHM*x* instruction. The code is sign extended. After execution, the appearance of the new stack is:

```
+-------------------------------------------------------------+
|                     sign-extended code                      | :(SP)
+-------------------------------------------------------------+
|                    PC of next instruction                   |
+-------------------------------------------------------------+
|                           old PSL                           |
+-------------------------------------------------------------+
```

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

Notes

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

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

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

EXAMPLES

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

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

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

REI

Return from Exception or Interrupt

FORMAT

opcode

condition codes

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

exceptions

reserved operand

opcodes

02	REI	Return from Exception or Interrupt

DESCRIPTION

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

Notes

1. The exception or interrupt service routine is responsible for restoring any registers saved and for removing any parameters from the stack.
2. As usual for faults, any Access Violation or Translation Not Valid conditions on the stack pops restore the stack pointer and fault.
3. The noninterrupt stack pointers may be fetched and stored either in privileged registers or in their allocated slots in the PCB. Only LDPCTX and SVPCTX always fetch and store in the PCB. MFPR and MTPR always fetch and store the pointers, whether in registers or in the PCB.

LDPCTX

Load Process Context

FORMAT

opcode

condition codes

N $\leftarrow$ N;
Z $\leftarrow$ Z;
V $\leftarrow$ V;
C $\leftarrow$ C;

exceptions

reserved operand
privileged instruction

opcodes

06	LDPCTX	Load Process Context

DESCRIPTION

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

Notes

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

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

SVPCTX

Save Process Context

FORMAT

opcode

condition codes

N ← N;
Z ← Z;
V ← V;
C ← C;

exceptions

privileged instruction

opcodes

07	SVPCTX	Save Process Context

DESCRIPTION

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

Notes

1. The map, ASTLVL, and PME contents of the PCB are not saved because they are rarely changed. Thus, not writing them saves overhead.
2. Some processors keep a copy of each of the per-process stack pointers in internal registers. In those processors, SVPCTX stores the internal registers in the PCB. Processors that do not keep a copy of all four per-process stack pointers in internal registers keep only the current access mode register in an internal register and switch this access mode register with the PCB contents whenever the current access mode field changes.
3. Between the SVPCTX instruction that saves the state for one process and the LDPCTX that loads the state of another, the 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.

MTPR

Move to Processor Register

FORMAT

opcode *src.rl, procreg.rl*

condition codes

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

C  <-- C;

N  <-- N;             ! If register is not replaced
Z  <-- Z;
V  <-- V;
C  <-- C;
```

exceptions

reserved operand fault
reserved instruction fault

opcodes

DA	MTPR	Move to Processor Register

DESCRIPTION

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

Notes

1. If the processor internal register does not exist, a reserved operand fault occurs.
2. A reserved instruction fault occurs if instruction execution is attempted in other than kernel mode.
3. A reserved operand fault occurs on a move to a read-only register.

MFPR

Move from Processor Register

FORMAT

opcode *procreg.rl, dst.wl*

condition codes

N	← dst LSS 0;	! If destination is replaced
Z	← dst EQL 0;	
V	← 0;	
C	← C;	
N	← N;	! If destination is not replaced
Z	← Z;	
V	← V;	
C	← C;	

exceptions

reserved operand fault
reserved instruction fault

opcodes

DB	MFPR	Move from Processor Register

DESCRIPTION

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

Notes

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

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

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

BUG

Bugcheck

FORMAT

opcode *message.bx*

condition codes

N ← N;
Z ← Z;
V ← V;
C ← C;

exceptions

reserved instruction

opcodes

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

DESCRIPTION

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

EXAMPLES

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

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

A ASCII Character Set

Table A–1 lists the ASCII characters and the decimal and hexadecimal codes for each.

Table A–1 Decimal, Hexadecimal, and ASCII Conversion

Dec	Hex	ASCII	Dec	Hex	ASCII	Dec	Hex	ASCII	Dec	Hex	ASCII
00	00	NUL	32	20	SP	64	40	@	96	60	'
01	01	SOH	33	21	!	65	41	A	97	61	a
02	02	STX	34	22	"	66	42	B	98	62	b
03	03	ETX	35	23	#	67	43	C	99	63	c
04	04	EOT	36	24	$	68	44	D	100	64	d
05	05	ENQ	37	25	%	69	45	E	101	65	e
06	06	ACK	38	26	&	70	46	F	102	66	f
07	07	BEL	39	27	'	71	47	G	103	67	g
08	08	BS	40	28	(	72	48	H	104	68	h
09	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{

ASCII Character Set

Table A–1 (Cont.) Decimal, Hexadecimal, and ASCII Conversion

Dec	Hex	ASCII	Dec	Hex	ASCII	Dec	Hex	ASCII	Dec	Hex	ASCII
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

B Hexadecimal/Decimal Conversion

Table B–1 lists the decimal value for each possible hexadecimal value in each byte of a longword. The following sections contain instructions to use the table to convert hexadecimal numbers to decimal and decimal numbers to hexadecimal.

B.1 Hexadecimal to Decimal

For each integer position of the hexadecimal value, locate the corresponding column integer and record its decimal equivalent in the conversion table. Add the decimal equivalent to obtain the decimal value.

For example:

```
D0500AD0 (hex)  =         ?(dec)

D0000000        = 3,489,660,928
  500000        =     5,242,880
     A00        =         2,560
      D0        =           208

D0500AD0        = 3,494,904,576
```

B.2 Decimal to Hexadecimal

To determine the hexadecimal equivalent of a given decimal value, perform the following steps:

1. In the conversion table, locate the largest decimal value that does not exceed the decimal number to be converted.
2. Record the hexadecimal equivalent, followed by the number of zeros that corresponds to the integer column minus 1.
3. Subtract the table decimal value from the decimal number to be converted.
4. Repeat steps 1 through 3 until the subtraction balance equals 0. Add the hexadecimal equivalents to obtain the hexadecimal value.

For example:

```
22,466 (dec)    = ?(hex)

20,480          = 5000         22,466
 1,792          =  700        -20,480
   192          =   C0
     2          =    2          1,986
                              - 1,792
22,466          = 57C2
                                  194
                              -   192
                                    2
                              -     2
                                    0
```

B.3 Powers of 2 and 16

This section lists the decimal values of powers of 2 and 16. These values are useful in converting decimal numbers to hexadecimal.

Powers of 2		Powers of 16	
2**n	n	16**n	n
256	8	1	0
512	9	16	1
1024	10	256	2
2048	11	4096	3
4096	12	65536	4
8192	13	1048576	5
16384	14	16777216	6
32768	15	268435456	7
65536	16	4294967296	8
131072	17	68719476736	9
262144	18	1099511627776	10
524288	19	17592186044416	11
1048576	20	281474976710656	12
2097152	21	4503599627370496	13
4194304	22	72057594037927936	14
8388608	23	1152921504606846976	15
16777216	24		

Table B–1 Hexadecimal to Decimal Conversion

HEXADECIMAL TO DECIMAL CONVERSION TABLE

8		7		6		5		4		3		2		1	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	268,435,456	1	16,777,216	1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	536,870,912	2	33,554,432	2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	805,306,368	3	50,331,648	3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	1,073,741,824	4	67,108,864	4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	1,342,177,280	5	83,886,080	5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	1,610,612,736	6	100,663,296	6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	1,879,048,192	7	117,440,512	7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	2,147,483,648	8	134,217,728	8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	2,415,919,104	9	150,994,944	9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	2,684,354,560	A	167,772,160	A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	2,952,790,016	B	184,549,376	B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	3,221,225,472	C	201,326,592	C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	3,489,660,928	D	218,103,808	D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	3,758,096,384	E	234,881,024	E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	4,026,531,840	F	251,658,240	F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15

ZK-2013-84

C VAX Macro Assembler Directives and Language Summary

C.1 Assembler Directives

The following table summarizes the VAX MACRO assembler directives:

Table C–1 Assembler Directives

Format	Operation
.ADDRESS address-list	Stores successive longwords of address data
.ALIGN keyword[,expression]	Aligns the location counter to the boundary specified by the keyword
.ALIGN integer[,expression]	Aligns location counter to the boundary specified by (2^integer)
.ASCIC string	Stores the ASCII string (enclosed in delimiters), preceded by a count byte
.ASCID string	Stores the ASCII string (enclosed in delimiters), preceded by a string descriptor
.ASCII string	Stores the ASCII string (enclosed in delimiters)
.ASCIZ string	Stores the ASCII string (enclosed in delimiters) followed by a 0 byte
.BLKA expression	Reserves longwords of address data
.BLKB expression	Reserves bytes for data
.BLKD expression	Reserves quadwords for double-precision floating-point data
.BLKF expression	Reserves longwords for single-precision floating-point data
.BLKG expression	Reserves quadwords for floating-point data
.BLKH expression	Reserves octawords for extended-precision floating-point data
.BLKL expression	Reserves longwords for data
.BLKO expression	Reserves octawords for data
.BLKQ expression	Reserves quadwords for data
.BLKW expression	Reserves words for data
.BYTE expression-list	Generates successive bytes of data; each byte contains the value of the specified expression
.CROSS	Enables cross-referencing of all symbols

C.1 Assembler Directives

Table C–1 (Cont.) Assembler Directives

Format	Operation
.CROSS symbol-list	Cross-references specified symbols
.DEBUG symbol-list	Makes symbol names known to the debugger
.DEFAULT DISPLACEMENT, keyword	Specifies the default displacement length for the relative addressing modes
.D_FLOATING literal-list	Generates 8-byte double-precision floating-point data
.DISABLE argument-list	Disables function(s) specified in argument-list
.DOUBLE literal-list	Equivalent to .D_FLOATING
.DSABL argument-list	Equivalent to .DISABLE
.ENABL argument-list	Equivalent to .ENABLE
.ENABLE argument-list	Enables function(s) specified in argument-list
.END [symbol]	Indicates logical end of source program; optional symbol specifies transfer address
.ENDC	Indicates end of conditional assembly block
.ENDM [macro-name]	Indicates end of macro definition
.ENDR	Indicates end of repeat block
.ENTRY symbol [,expression]	Procedure entry directive
.ERROR [expression] ;comment	Displays specified error message
.EVEN	Ensures that the current location counter has an even value (adds 1 if it is odd)
.EXTERNAL symbol-list	Indicates specified symbols are externally defined
.EXTRN symbol-list	Equivalent to .EXTERNAL
.F_FLOATING literal-list	Generates 4-byte single-precision floating-point data
.FLOAT literal-list	Equivalent to .F_FLOATING
.G_FLOATING literal-list	Generates 8-byte G_floating-point data
.GLOBAL symbol-list	Indicates specified symbols are global symbols
.GLOBL	Equivalent to .GLOBAL
.H_FLOATING literal-list	Generates 16-byte extended-precision H_floating-point data
.IDENT string	Provides means of labeling object module with additional data

Table C–1 (Cont.) Assembler Directives

Format	Operation
.IF condition [,] argument(s)	Begins a conditional assembly block of source code which is included in the assembly only if the stated condition is met with respect to the argument(s) specified
.IFF	Equivalent to .IF_FALSE
.IF_FALSE	Appears only within a conditional assembly block; begins block of code to be assembled if the original condition tests false
.IFT	Equivalent to .IF_TRUE
.IFTF	Equivalent to .IF_TRUE_FALSE
.IF_TRUE	Appears only within a conditional assembly block; begins block of code to be assembled if the original condition tests true
.IF_TRUE_FALSE	Appears only within a conditional assembly block; begins block of code to be assembled unconditionally
.IIF condition argument(s), statement	Acts as a 1-line conditional assembly block where the condition is tested for the argument specified; the statement is assembled only if the condition tests true
.IRP symbol, <argument list>	Replaces a formal argument with successive actual arguments specified in an argument list
.IRPC symbol, <BIT_STRING>	Replaces a formal argument with successive single characters specified in string
.LIBRARY macro-library-name	Specifies a macro library
.LINK "file-spec" [/qualifier[=(module-name[,...])],...]	Includes linker option records in object module
.LIST [argument-list]	Equivalent to .SHOW
.LONG expression-list	Generates successive longwords of data; each longword contains the value of the specified expression
.MACRO macro-name [formal-argument-list]	Begins a macro definition
.MASK symbol [,expression]	Reserves a word for and copies a register save mask
.MCALL macro-name-list	Specifies the system and/or user-defined macros in libraries that are required to assemble the source program

Table C–1 (Cont.) Assembler Directives

Format	Operation
.MDELETE macro-name-list	Deletes from memory the macro definitions of the macros in the list
.MEXIT	Exits from the expansion of a macro before the end of the macro is encountered
.NARG symbol	Determines the number of arguments in the current macro call
.NCHR symbol, <BIT_STRING>	Determines the number of characters in a specified character string
.NLIST [argument-list]	Equivalent to .NOSHOW
.NOCROSS	Disables cross-referencing of all symbols
.NOCROSS symbol-list	Disables cross-referencing of specified symbols
.NOSHOW	Decrements listing level count
.NOSHOW argument-list	Controls listing of macros and conditional assembly blocks
.NTYPE symbol,operand	Can appear only within a macro definition; equates the symbol to the addressing mode of the specified operand
.OCTA literal	Stores 16 bytes of data
.OCTA symbol	Stores 16 bytes of data
.ODD	Ensures that the current location counter has an odd value (adds 1 if it is even)
.OPDEF opcode value, operand-descriptor-list	Defines an opcode and its operand list
.PACKED decimal-string [,symbol]	Generates packed decimal data, 2 digits per byte
.PAGE	Causes the assembly listing to skip to the top of the next page and to increment the page count
.PRINT [expression] ;comment	Displays the specified message
.PSECT	Begins or resumes the blank program section
.PSECT section-name argument list	Begins or resumes a user-defined program section
.QUAD literal	Stores 8 bytes of data
.QUAD symbol	Stores 8 bytes of data
.REF1 operand	Generates byte operand
.REF2 operand	Generates word operand
.REF4 operand	Generates longword operand

Table C–1 (Cont.) Assembler Directives

Format	Operation
.REF8 operand	Generates quadword operand
.REF16 operand	Generates octaword operand
.REPEAT expression	Begins a repeat block; the section of code up to the next .ENDR directive is repeated the number of times specified by the expression
.REPT	Equivalent to .REPEAT
.RESTORE	Equivalent to .RESTORE_PSECT
.RESTORE_PSECT	Restores program section context from the program section context stack
.SAVE [LOCAL_BLOCK]	Equivalent to .SAVE_PSECT
.SAVE_PSECT [LOCAL_BLOCK]	Saves current program section context on the program section context stack
.SBTTL comment-string	Equivalent to .SUBTITLE
.SHOW	Increments listing level count
.SHOW argument-list	Controls listing of macros and conditional assembly blocks
.SIGNED_BYTE expression-list	Stores successive bytes of signed data
.SIGNED_WORD expression-list	Stores successive words of signed data
.SUBTITLE comment-string	Causes the specified string to be printed as part of the assembly listing page header; the string component of each .SUBTITLE is collected into a table of contents at the beginning of the assembly listing
.TITLE module-name comment-string	Assigns the first 15 characters in the string as an object module name and causes the string to appear on each page of the assembly listing
.TRANSFER symbol	Directs the linker to redefine the value of the global symbol for use in a shareable image
.WARN [expression] ;comment	Displays specified warning message
.WEAK symbol-list	Indicates that each of the listed symbols has the weak attribute
.WORD expression-list	Generates successive words of data; each word contains the value of the corresponding specified expression

C.2 Special Characters

The following table summarizes the VAX MACRO special characters:

Table C–2 Special Characters Used in VAX MACRO Statements

Character	Character Name	Function(s)
_	Underline	Character in symbol names
$	Dollar sign	Character in symbol names
.	Period	Character in symbol names, current location counter, and decimal point
:	Colon	Label terminator
=	Equal sign	Direct assignment operator and macro keyword argument terminator
	Tab	Field terminator
	Space	Field terminator
#	Number sign	Immediate addressing mode indicator
@	At sign	Deferred addressing mode indicator and arithmetic shift operator
,	Comma	Field, operand, and item separator
;	Semicolon	Comment field indicator
+	Plus sign	Autoincrement addressing mode indicator, unary plus operator, and arithmetic addition operator
–	Minus sign	Autodecrement addressing mode indicator, unary minus operator, arithmetic subtraction operator, and line continuation indicator
*	Asterisk	Arithmetic multiplication operator
/	Slash	Arithmetic division operator
&	Ampersand	Logical AND operator
!	Exclamation point	Logical inclusive OR operator
\	Backslash	Logical exclusive OR and numeric conversion indicator in macro arguments
^	Circumflex	Unary operator indicator and macro argument delimiter
[ ]	Square brackets	Index addressing mode and repeat count indicators
( )	Parentheses	Register deferred addressing mode indicators

Table C–2 (Cont.) Special Characters Used in VAX MACRO Statements

Character	Character Name	Function(s)
<>	Angle brackets	Argument or expression grouping delimiters
?	Question mark	Created label indicator in macro arguments
'	Apostrophe	Macro argument concatenation indicator
%	Percent sign	Macro string operators

C.3 Operators

C.3.1 Unary Operators

The following table summarizes the VAX MACRO unary operators:

Table C–3 Unary Operators

Unary Operator	Operator Name	Example	Effect
+	Plus sign	+A	Results in the positive value of A (default)
–	Minus sign	–A	Results in the negative (2's complement) value of A
^B	Binary	^B11000111	Specifies that 11000111 is a binary number
^D	Decimal	^D127	Specifies that 127 is a decimal number
^O	Octal	^O34	Specifies that 34 is an octal number
^X	Hexadecimal	^XFCF9	Specifies that FCF9 is a hexadecimal number
^A	ASCII	^A/ABC/	Produces an ASCII string; the characters between the matching delimiters are converted to ASCII representation
^M	Register mask	^M <R3,R4,R5>	Specifies the registers R3, R4, and R5 in the register mask
^F	Floating point	^F3.0	Specifies that 3.0 is a floating-point number
^C	Complement	^C24	Produces the 1's complement value of 24 (decimal)

C.3.2 Binary Operators

The following table summarizes the VAX MACRO binary operators:

Table C–4 Binary Operators

Binary Operator	Operator Name	Example	Operation
+	Plus sign	A+B	Addition
–	Minus sign	A–B	Subtraction
*	Asterisk	A*B	Multiplication
/	Slash	A/B	Division
@	At sign	A@B	Arithmetic Shift
&	Ampersand	A&B	Logical AND
!	Exclamation point	A!B	Logical inclusive OR
\	Backslash	A\B	Logical exclusive OR

C.3.3 Macro String Operators

The following table summarizes the macro string operators. These operators can be used only in macros.

Table C–5 Macro String Operators

Format	Function
%LENGTH(string)	Returns the length of the string
%LOCATE(string1,string2[,symbol])	Locates the substring string1 within string2 starting the search at the character position specified by symbol
%EXTRACT(symbol1,symbol2,string)	Extracts a substring from string that begins at character position specified by symbol1 and has a length specified by symbol2

C.4 Addressing Modes

The following table summarizes the VAX MACRO addressing modes:

Table C–6 Addressing Modes

Type	Addressing Mode	Format	Hex Value	Description	Indexable?
General Register	Register	Rn	5	Register contains the operand.	No
	Register Deferred	(Rn)	6	Register contains the address of the operand.	Yes
	Autoincrement	(Rn)+	8	Register contains the address of the operand; the processor increments the register contents by the size of the operand data type.	Yes
	Autoincrement Deferred	@(Rn)+	9	Register contains the address of the operand address; the processor increments the register contents by 4.	Yes
	Autodecrement	-(Rn)	7	The processor decrements the register contents by the size of the operand data type; the register then contains the address of the operand.	Yes
	Displacement	dis(Rn) B^dis(Rn) W^dis(Rn) L^dis(Rn)	 A C E	The sum of the contents of the register and the displacement is the address of the operand; B^, W^, and L^, respectively, indicate byte, word, and longword displacement.	Yes
	Displacement Deferred	@dis(Rn) @B^dis(Rn) @W^dis(Rn) @L^dis(Rn)	 B D F	The sum of the contents of the register and the displacement is the address of the operand address; B^, W^, and L^, respectively, indicate, byte, word, and longword displacement.	Yes

Key:

Rn — Any general register R0 through R12. Note that the AP, FP, or SP register can be used in place of Rn.
Rx — Any general register R0 through R12. Note that the AP, FP, or SP register can be used in place of Rx. Rx cannot be the same as the Rn specified in the base-mode for certain base modes (see Section 5.3).
dis — An expression specifying a displacement.
address — An expression specifying an address.
literal — An expression, an integer constant, or a floating-point constant.

Table C–6 (Cont.) Addressing Modes

Type	Addressing Mode	Format	Hex Value	Description	Indexable?
	Literal	#literal S^#literal	 0-3	The literal specified is the operand; the literal is stored as a short literal.	No
Program Counter	Relative	address B^address W^address L^address	 A C E	The address specified is the address of the operand; the address is stored as a displacement from the PC; B^, W^, and L^, respectively, indicate byte, word, and longword displacement.	Yes
	Relative Deferred	@address @B^address @W^address @L^address	 B D F	The address specified is the address of the operand address; the address specified is stored as a displacement from the PC; B^, W^, and L^ indicate byte, word, and longword displacement, respectively.	Yes
	Absolute	@#address	9	The address specified is the address of the operand; the address specified is stored as an absolute virtual address, not as a displacement.	Yes
	Immediate	#literal I^#literal	 8	The literal specified is the operand; the literal is stored as a byte, word, longword, or quadword.	No
	General	G^address	—	The address specified is the address of the operand; if the address is defined as relocatable, the linker stores the address as a displacement from the PC; if the address is defined as an absolute virtual address, the linker stores the address as an absolute value.	Yes

Key:

Rn — Any general register R0 through R12. Note that the AP, FP, or SP register can be used in place of Rn.
Rx — Any general register R0 through R12. Note that the AP, FP, or SP register can be used in place of Rx. Rx cannot be the same as the Rn specified in the base-mode for certain base modes (see Section 5.3).
dis — An expression specifying a displacement.
address — An expression specifying an address.
literal — An expression, an integer constant, or a floating-point constant.

Table C–6 (Cont.) Addressing Modes

Type	Addressing Mode	Format	Hex Value	Description	Indexable?
Index	Index	base-mode[Rx]	4	The base-mode specifies the base address, and the register specifies the index; the sum of the base address and the product of the contents of Rx and the size of the operand data type is the address of the operand; base mode can be any addressing mode except register, immediate, literal, index, or branch.	No
Branch	Branch	address	—	The address specified is the operand; this address is stored as a displacement from the PC; branch mode can only be used with the branch instructions.	No

Key:

Rn — Any general register R0 through R12. Note that the AP, FP, or SP register can be used in place of Rn.
Rx — Any general register R0 through R12. Note that the AP, FP, or SP register can be used in place of Rx. Rx cannot be the same as the Rn specified in the base-mode for certain base modes (see Section 5.3).
dis — An expression specifying a displacement.
address — An expression specifying an address.
literal — An expression, an integer constant, or a floating-point constant.

D Permanent Symbol Table

The permanent symbol table (PST) contains the symbols that VAX MACRO automatically recognizes. These symbols consist of both opcodes and assembler directives. Sections D.1 and D.2 below present the opcodes (instruction set) in alphabetical and numerical order, respectively. Section C.1 (in Appendix C) presents the assembler directives.

See Chapter 9 for detailed descriptions of the instruction set.

D.1 Opcodes (Alphabetic Order)

Table D–1 Opcodes and Functions

Hex Value	Mnemonic	Functional Name
9D	ACBB	Add compare and branch byte
6F	ACBD	Add compare and branch D_floating
4F	ACBF	Add compare and branch F_floating
4FFD	ACBG	Add compare and branch G_floating
6FFD	ACBH	Add compare and branch H_floating
F1	ACBL	Add compare and branch long
3D	ACBW	Add compare and branch word
58	ADAWI	Add aligned word interlocked
80	ADDB2	Add byte 2 operand
81	ADDB3	Add byte 3 operand
60	ADDD2	Add D_floating 2 operand
61	ADDD3	Add D_floating 3 operand
40	ADDF2	Add F_floating 2 operand
41	ADDF3	Add F_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
C0	ADDL2	Add long 2 operand
C1	ADDL3	Add long 3 operand
20	ADDP4	Add packed 4 operand
21	ADDP6	Add packed 6 operand
A0	ADDW2	Add word 2 operand
A1	ADDW3	Add word 3 operand

Table D–1 (Cont.) Opcodes and Functions

Hex Value	Mnemonic	Functional Name
D8	ADWC	Add with carry
F3	AOBLEQ	Add one and branch on less or equal
F2	AOBLSS	Add one and branch on less
78	ASHL	Arithmetic shift long
F8	ASHP	Arithmetic shift and round packed
79	ASHQ	Arithmetic shift quad
E1	BBC	Branch on bit clear
E5	BBCC	Branch on bit clear and clear
E7	BBCCI	Branch on bit clear and clear interlocked
E3	BBCS	Branch on bit clear and set
E0	BBS	Branch on bit set
E4	BBSC	Branch on bit set and clear
E2	BBSS	Branch on bit set and set
E6	BBSSI	Branch on bit set and set interlocked
1E	BCC	Branch on carry clear
1F	BCS	Branch on carry set
13	BEQL	Branch on equal
13	BEQLU	Branch on equal unsigned
18	BGEQ	Branch on greater or equal
1E	BGEQU	Branch on greater or equal unsigned
14	BGTR	Branch on greater
1A	BGTRU	Branch on greater unsigned
8A	BICB2	Bit clear byte 2 operand
8B	BICB3	Bit clear byte 3 operand
CA	BICL2	Bit clear long 2 operand
CB	BICL3	Bit clear long 3 operand
B9	BICPSW	Bit clear program status word
AA	BICW2	Bit clear word 2 operand
AB	BICW3	Bit clear word 3 operand
88	BISB2	Bit set byte 2 operand
89	BISB3	Bit set byte 3 operand
C8	BISL2	Bit set long 2 operand
C9	BISL3	Bit set long 3 operand
B8	BISPSW	Bit set program status word
A8	BISW2	Bit set word 2 operand
A9	BISW3	Bit set word 3 operand
93	BITB	Bit test byte

Table D–1 (Cont.) Opcodes and Functions

Hex Value	Mnemonic	Functional Name
D3	BITL	Bit test long
B3	BITW	Bit test word
E9	BLBC	Branch on low bit clear
E8	BLBS	Branch on low bit set
15	BLEQ	Branch on less or equal
1B	BLEQU	Branch on less or equal unsigned
19	BLSS	Branch on less
1F	BLSSU	Branch on less unsigned
12	BNEQ	Branch on not equal
12	BNEQU	Branch on not equal unsigned
03	BPT	Break point trap
11	BRB	Branch with byte displacement
31	BRW	Branch with word displacement
10	BSBB	Branch to subroutine with byte displacement
30	BSBW	Branch to subroutine with word displacement
1C	BVC	Branch on overflow clear
1D	BVS	Branch on overflow set
FA	CALLG	Call with general argument list
FB	CALLS	Call with stack
8F	CASEB	Case byte
CF	CASEL	Case long
AF	CASEW	Case word
BD	CHME	Change mode to executive
BC	CHMK	Change mode to kernel
BE	CHMS	Change mode to supervisor
BF	CHMU	Change mode to user
94	CLRB	Clear byte
7C	CLRD	Clear D_floating
DF	CLRF	Clear F_floating
7C	CLRG	Clear G_floating
7CFD	CLRH	Clear H_floating
D4	CLRL	Clear long
7CFD	CLRO	Clear octa
7C	CLRQ	Clear quad
B4	CLRW	Clear word
91	CMPB	Compare byte
29	CMPC3	Compare character 3 operand

Table D–1 (Cont.) Opcodes and Functions

Hex Value	Mnemonic	Functional Name
2D	CMPC5	Compare character 5 operand
71	CMPD	Compare D_floating
51	CMPF	Compare F_floating
51FD	CMPG	Compare G_floating
71FD	CMPH	Compare H_floating
D1	CMPL	Compare long
35	CMPP3	Compare packed 3 operand
37	CMPP4	Compare packed 4 operand
EC	CMPV	Compare field
B1	CMPW	Compare word
ED	CMPZV	Compare zero-extended field
0B	CRC	Calculate cyclic redundancy check
6C	CVTBD	Convert byte to D_floating
4C	CVTBF	Convert byte to F_floating
4CFD	CVTBG	Convert byte to G_floating
6CFD	CVTBH	Convert byte to H_floating
98	CVTBL	Convert byte to long
99	CVTBW	Convert byte to word
68	CVTDB	Convert D_floating to byte
76	CVTDF	Convert D_floating to F_floating
32FD	CVTDH	Convert D_floating to H_floating
6A	CVTDL	Convert D_floating to long
69	CVTDW	Convert D_floating to word
48	CVTFB	Convert F_floating to byte
56	CVTFD	Convert F_floating to D_floating
99FD	CVTFG	Convert F_floating to G_floating
98FD	CVTFH	Convert F_floating to H_floating
4A	CVTFL	Convert F_floating to long
49	CVTFW	Convert F_floating to word
48FD	CVTGB	Convert G_floating to byte
33FD	CVTGF	Convert G_floating to F_floating
56FD	CVTGH	Convert G_floating to H_floating
4AFD	CVTGL	Convert G_floating to long
49FD	CVTGW	Convert G_floating to word
68FD	CVTHB	Convert H_floating to byte
F7FD	CVTHD	Convert H_floating to D_floating
F6FD	CVTHF	Convert H_floating to F_floating

Table D–1 (Cont.) Opcodes and Functions

Hex Value	Mnemonic	Functional Name
76FD	CVTHG	Convert H_floating to G_floating
6AFD	CVTHL	Convert H_floating to long
69FD	CVTHW	Convert H_floating to word
F6	CVTLB	Convert long to byte
6E	CVTLD	Convert long to D_floating
4E	CVTLF	Convert long to F_floating
4EFD	CVTLG	Convert long to G_floating
6EFD	CVTLH	Convert long to H_floating
F9	CVTLP	Convert long to packed
F7	CVTLW	Convert long to word
36	CVTPL	Convert packed to long
08	CVTPS	Convert packed to leading separate
24	CVTPT	Convert packed to trailing
6B	CVTRDL	Convert rounded D_floating to long
4B	CVTRFL	Convert rounded F_floating to long
4BFD	CVTRGL	Convert rounded G_floating to long
6BFD	CVTRHL	Convert rounded H_floating to long
09	CVTSP	Convert leading separate to packed
26	CVTTP	Convert trailing to packed
33	CVTWB	Convert word to byte
6D	CVTWD	Convert word to D_floating
4D	CVTWF	Convert word to F_floating
4DFD	CVTWG	Convert word to G_floating
6DFD	CVTWH	Convert word to H_floating
32	CVTWL	Convert word to long
97	DECB	Decrement byte
D7	DECL	Decrement long
B7	DECW	Decrement word
86	DIVB2	Divide byte 2 operand
87	DIVB3	Divide byte 3 operand
66	DIVD2	Divide D_floating 2 operand
67	DIVD3	Divide D_floating 3 operand
46	DIVF2	Divide F_floating 2 operand
47	DIVF3	Divide F_floating 3 operand
46FD	DIVG2	Divide G_floating 2 operand
47FD	DIVG3	Divide G_floating 3 operand
66FD	DIVH2	Divide H_floating 2 operand

Table D–1 (Cont.) Opcodes and Functions

Hex Value	Mnemonic	Functional Name
67FD	DIVH3	Divide H_floating 3 operand
C6	DIVL2	Divide long 2 operand
C7	DIVL3	Divide long 3 operand
27	DIVP	Divide packed
A6	DIVW2	Divide word 2 operand
A7	DIVW3	Divide word 3 operand
38	EDITPC	Edit packed to character
7B	EDIV	Extended divide
74	EMODD	Extended modulus D_floating
54	EMODF	Extended modulus F_floating
54FD	EMODG	Extended modulus G_floating
74FD	EMODH	Extended modulus H_floating
7A	EMUL	Extended multiply
EE	EXTV	Extract field
EF	EXTZV	Extract zero-extended field
EB	FFC	Find first clear bit
EA	FFS	Find first set bit
00	HALT	Halt
96	INCB	Increment byte
D6	INCL	Increment long
B6	INCW	Increment word
0A	INDEX	Index calculation
5C	INSQHI	Insert into queue at head, interlocked
5D	INSQTI	Insert into queue at tail, interlocked
0E	INSQUE	Insert into queue
F0	INSV	Insert field
17	JMP	Jump
16	JSB	Jump to subroutine
06	LDPCTX	Load program context
3A	LOCC	Locate character
39	MATCHC	Match characters
92	MCOMB	Move complemented byte
D2	MCOML	Move complemented long
B2	MCOMW	Move complemented word
DB	MFPR	Move from processor register
8E	MNEGB	Move negated byte
72	MNEGD	Move negated D_floating

Table D–1 (Cont.) Opcodes and Functions

Hex Value	Mnemonic	Functional Name
52	MNEGF	Move negated F_floating
52FD	MNEGG	Move negated G_floating
72FD	MNEGH	Move negated H_floating
CE	MNEGL	Move negated long
AE	MNEGW	Move negated word
9E	MOVAB	Move address of byte
7E	MOVAD	Move address of D_floating
DE	MOVAF	Move address of F_floating
7E	MOVAG	Move address of G_floating
7EFD	MOVAH	Move address of H_floating
DE	MOVAL	Move address of long
7EFD	MOVAO	Move address of octa
7E	MOVAQ	Move address of quad
3E	MOVAW	Move address of word
90	MOVB	Move byte
28	MOVC3	Move character 3 operand
2C	MOVC5	Move character 5 operand
70	MOVD	Move D_floating
50	MOVF	Move F_floating
50FD	MOVG	Move G_floating
70FD	MOVH	Move H_floating
D0	MOVL	Move long
7DFD	MOVO	Move data
34	MOVP	Move packed
DC	MOVPSL	Move program status longword
7D	MOVQ	Move quad
2E	MOVTC	Move translated characters
2F	MOVTUC	Move translated until character
B0	MOVW	Move word
0A	MOVZBL	Move zero-extended byte to long
9B	MOVZBW	Move zero-extended byte to word
3C	MOVZWL	Move zero-extended word to long
DA	MTPR	Move to processor register
84	MULB2	Multiply byte 2 operand
85	MULB3	Multiply byte 3 operand
64	MULD2	Multiply D_floating 2 operand
65	MULD3	Multiply D_floating 3 operand

Table D–1 (Cont.) Opcodes and Functions

Hex Value	Mnemonic	Functional Name
44	MULF2	Multiply F_floating 2 operand
45	MULF3	Multiply F_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
C4	MULL2	Multiply long 2 operand
C5	MULL3	Multiply long 3 operand
25	MULP	Multiply packed
A4	MULW2	Multiply word 2 operand
A5	MULW3	Multiply word 3 operand
01	NOP	No operation
75	POLYD	Evaluate polynomial D_floating
55	POLYF	Evaluate polynomial F_floating
55FD	POLYG	Evaluate polynomial G_floating
75FD	POLYH	Evaluate polynomial H_floating
BA	POPR	Pop registers
0C	PROBER	Probe read access
0D	PROBEW	Probe write access
9F	PUSHAB	Push address of byte
7F	PUSHAD	Push address of D_floating
DF	PUSHAF	Push address of F_floating
7F	PUSHAG	Push address of G_floating
7FFD	PUSHAH	Push address of H_floating
DF	PUSHAL	Push address of long
7FFD	PUSHAO	Push address of octa
7F	PUSHAQ	Push address of quad
3F	PUSHAW	Push address of word
DD	PUSHL	Push long
BB	PUSHR	Push registers
02	REI	Return from exception or interrupt
5E	REMQHI	Remove from queue at head, interlocked
5F	REMQTI	Remove from queue at tail, interlocked
0F	REMQUE	Remove from queue
04	RET	Return from called procedure
9C	ROTL	Rotate long
05	RSB	Return from subroutine

Table D–1 (Cont.) Opcodes and Functions

Hex Value	Mnemonic	Functional Name
D9	SBWC	Subtract with carry
2A	SCANC	Scan for character
3B	SKPC	Skip character
F4	SOBGEQ	Subtract one and branch on greater or equal
F5	SOBGTR	Subtract one and branch on greater
2B	SPANC	Span characters
82	SUBB2	Subtract byte 2 operand
83	SUBB3	Subtract byte 3 operand
62	SUBD2	Subtract D_floating 2 operand
63	SUBD3	Subtract D_floating 3 operand
42	SUBF2	Subtract F_floating 2 operand
43	SUBF3	Subtract F_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
C2	SUBL2	Subtract long 2 operand
C3	SUBL3	Subtract long 3 operand
22	SUBP4	Subtract packed 4 operand
23	SUBP6	Subtract packed 6 operand
A2	SUBW2	Subtract word 2 operand
A3	SUBW3	Subtract word 3 operand
07	SVPCTX	Save process context
95	TSTB	Test byte
73	TSTD	Test D_floating
53	TSTF	Test F_floating
53FD	TSTG	Test G_floating
73FD	TSTH	Test H_floating
D5	TSTL	Test long
B5	TSTW	Test word
FC	XFC	Extended function call
8C	XORB2	Exclusive-OR byte 2 operand
8D	XORB3	Exclusive-OR byte 3 operand
CC	XORL2	Exclusive-OR long 2 operand
CD	XORL3	Exclusive-OR long 3 operand
AC	XORW2	Exclusive-OR word 2 operand
AD	XORW3	Exclusive-OR word 3 operand

D.2 Opcodes (Numeric Order)

Table D–2 One-Byte Opcodes

Hex Value	Mnemonic	Hex Value	Mnemonic
00	HALT	30	BSBW
01	NOP	31	BRW
02	REI	32	CVTWL
03	BPT	33	CVTWB
04	RET	34	MOVP
05	RSB	35	CMPP3
06	LDPCTX	36	CVTPL
07	SVPCTX	37	CMPP4
08	CVTPS	38	EDITPC
09	CVTSP	39	MATCHC
0A	INDEX	3A	LOCC
0B	CRC	3B	SKPC
0C	PROBER	3C	MOVZWL
0D	PROBEW	3D	ACBW
0E	INSQUE	3E	MOVAW
0F	REMQUE	3F	PUSHAW
10	BSBB	40	ADDF2
11	BRB	41	ADDF3
12	BNEQ, BNEQU	42	SUBF2
13	BEQL, BEQLU	43	SUBF3
14	BGTR	44	MULF2
15	BLEQ	45	MULF3
16	JSB	46	DIVF2
17	JMP	47	DIVF3
18	BGEQ	48	CVTFB
19	BLSS	49	CVTFW
1A	BGTRU	4A	CVTFL
1B	BLEQU	4B	CVTRFL
1C	BVC	4C	CVTBF
1D	BVS	4D	CVTWF
1E	BGEQU, BCC	4E	CVTLF
1F	BLSSU, BCS	4F	ACBF
20	ADDP4	50	MOVF
21	ADDP6	51	CMPF
22	SUBP4	52	MNEGF

Table D–2 (Cont.) One-Byte Opcodes

Hex Value	Mnemonic	Hex Value	Mnemonic
23	SUBP6	53	TSTF
24	CVTPT	54	EMODF
25	MULP	55	POLYF
26	CVTTP	56	CVTFD
27	DIVP	57	Reserved to DIGITAL
28	MOVC3	58	ADAWI
29	CMPC3	59	Reserved to DIGITAL
2A	SCANC	5A	Reserved to DIGITAL
2B	SPANC	5B	Reserved to DIGITAL
2C	MOVC5	5C	INSQHI
2D	CMPC5	5D	INSQTI
2E	MOVTC	5E	REMQHI
2F	MOVTUC	5F	REMQTI
60	ADDD2	90	MOVB
61	ADDD3	91	CMPB
62	SUBD2	92	MCOMB
63	SUBD3	93	BITB
64	MULD2	94	CLRB
65	MULD3	95	TSTB
66	DIVD2	96	INCB
67	DIVD3	97	DECB
68	CVTDB	98	CVTBL
69	CVTDW	99	CVTBW
6A	CVTDL	9A	MOVZBL
6B	CVTRDL	9B	MOVZBW
6C	CVTBD	9C	ROTL
6D	CVTWD	9D	ACBB
6E	CVTLD	9E	MOVAB
6F	ACBD	9F	PUSHAB
70	MOVD	A0	ADDW2
71	CMPD	A1	ADDW3
72	MNEGD	A2	SUBW2
73	TSTD	A3	SUBW3
74	EMODD	A4	MULW2

Table D–2 (Cont.) One-Byte Opcodes

Hex Value	Mnemonic	Hex Value	Mnemonic
75	POLYD	A5	MULW3
76	CVTDF	A6	DIVW2
77	Reserved to DIGITAL	A7	DIVW3
78	ASHL	A8	BISW2
79	ASHQ	A9	BISW3
7A	EMUL	AA	BICW2
7B	EDIV	AB	BICW3
7C	CLRQ, CLRD, CLRG	AC	XORW2
7D	MOVQ	AD	XORW3
7E	MOVAQ, MOVAD, MOVAG	AE	MNEGW
7F	PUSHAQ, PUSHAD, PUSHAG	AF	CASEW
80	ADDB2	B0	MOVW
81	ADDB3	B1	CMPW
82	SUBB2	B2	MCOMW
83	SUBB3	B3	BITW
84	MULB2	B4	CLRW
85	MULB3	B5	TSTW
86	DIVB2	B6	INCW
87	DIVB3	B7	DECW
88	BISB2	B8	BISPSW
89	BISB3	B9	BICPSW
8A	BICB2	BA	POPR
8B	BICB3	BB	PUSHR
8C	XORB2	BC	CHMK
8D	XORB3	BD	CHME
8E	MNEGB	BE	CHMS
8F	CASEB	BF	CHMU
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

Table D–2 (Cont.) One-Byte Opcodes

Hex Value	Mnemonic	Hex Value	Mnemonic
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	ESCD to DIGITAL
DE	MOVAL, MOVA	FE	ESCE to DIGITAL
DF	PUSHAL, PUSHAF	FF	ESCF to DIGITAL

Table D–3 Two-Byte Opcodes

Hex Value	Mnemonic	Hex Value	Mnemonic
00FD	Reserved to DIGITAL	30FD	Reserved to DIGITAL
01FD	Reserved to DIGITAL	31FD	Reserved to DIGITAL
02FD	Reserved to DIGITAL	32FD	CVTDH
03FD	Reserved to DIGITAL	33FD	CVTGF
04FD	Reserved to DIGITAL	34FD	Reserved to DIGITAL
05FD	Reserved to DIGITAL	35FD	Reserved to DIGITAL
06FD	Reserved to DIGITAL	36FD	Reserved to DIGITAL
07FD	Reserved to DIGITAL	37FD	Reserved to DIGITAL
08FD	Reserved to DIGITAL	38FD	Reserved to DIGITAL
09FD	Reserved to DIGITAL	39FD	Reserved to DIGITAL
0AFD	Reserved to DIGITAL	3AFD	Reserved to DIGITAL
0BFD	Reserved to DIGITAL	3BFD	Reserved to DIGITAL
0CFD	Reserved to DIGITAL	3CFD	Reserved to DIGITAL
0DFD	Reserved to DIGITAL	3DFD	Reserved to DIGITAL
0EFD	Reserved to DIGITAL	3EFD	Reserved to DIGITAL
0FFD	Reserved to DIGITAL	3FFD	Reserved to DIGITAL
10FD	Reserved to DIGITAL	40FD	ADDG2
11FD	Reserved to DIGITAL	41FD	ADDG3
12FD	Reserved to DIGITAL	42FD	SUBG2
13FD	Reserved to DIGITAL	43FD	SUBG3
14FD	Reserved to DIGITAL	44FD	MULG2
15FD	Reserved to DIGITAL	45FD	MULG3
16FD	Reserved to DIGITAL	46FD	DIVG2
17FD	Reserved to DIGITAL	47FD	DIVG3
18FD	Reserved to DIGITAL	48FD	CVTGB
19FD	Reserved to DIGITAL	49FD	CVTGW
1AFD	Reserved to DIGITAL	4AFD	CVTGL
1BFD	Reserved to DIGITAL	4BFD	CVTRGL
1CFD	Reserved to DIGITAL	4CFD	CVTBG
1DFD	Reserved to DIGITAL	4DFD	CVTWG
1EFD	Reserved to DIGITAL	4EFD	CVTLG
1FFD	Reserved to DIGITAL	4FFD	ACBG
20FD	Reserved to DIGITAL	50FD	MOVG
21FD	Reserved to DIGITAL	51FD	CMPG
22FD	Reserved to DIGITAL	52FD	MNEGG
23FD	Reserved to DIGITAL	53FD	TSTG
24FD	Reserved to DIGITAL	54FD	EMODG

Table D–3 (Cont.) Two-Byte Opcodes

Hex Value	Mnemonic	Hex Value	Mnemonic
25FD	Reserved to DIGITAL	55FD	POLYG
26FD	Reserved to DIGITAL	56FD	CVTGH
27FD	Reserved to DIGITAL	57FD	Reserved to DIGITAL
28FD	Reserved to DIGITAL	58FD	Reserved to DIGITAL
29FD	Reserved to DIGITAL	59FD	Reserved to DIGITAL
2AFD	Reserved to DIGITAL	5AFD	Reserved to DIGITAL
2BFD	Reserved to DIGITAL	5BFD	Reserved to DIGITAL
2CFD	Reserved to DIGITAL	5CFD	Reserved to DIGITAL
2DFD	Reserved to DIGITAL	5DFD	Reserved to DIGITAL
2EFD	Reserved to DIGITAL	5EFD	Reserved to DIGITAL
2FFD	Reserved to DIGITAL	5FFD	Reserved to DIGITAL
60FD	ADDH2	90FD	Reserved to DIGITAL
61FD	ADDH3	91FD	Reserved to DIGITAL
62FD	SUBH2	92FD	Reserved to DIGITAL
63FD	SUBH3	93FD	Reserved to DIGITAL
64FD	MULH2	94FD	Reserved to DIGITAL
65FD	MULH3	95FD	Reserved to DIGITAL
66FD	DIVH2	96FD	Reserved to DIGITAL
67FD	DIVH3	97FD	Reserved to DIGITAL
68FD	CVTHB	98FD	CVTFH
69FD	CVTHW	99FD	CVTFG
6AFD	CVTHL	9AFD	Reserved to DIGITAL
6BFD	CVTRHL	9BFD	Reserved to DIGITAL
6CFD	CVTBH	9CFD	Reserved to DIGITAL
6DFD	CVTWH	9DFD	Reserved to DIGITAL
6EFD	CVTLH	9EFD	Reserved to DIGITAL
6FFD	ACBH	9FFD	Reserved to DIGITAL
70FD	MOVH	A0FD	Reserved to DIGITAL
71FD	CMPH	A1FD	Reserved to DIGITAL
72FD	MNEGH	A2FD	Reserved to DIGITAL
73FD	TSTH	A3FD	Reserved to DIGITAL
74FD	EMODH	A4FD	Reserved to DIGITAL
75FD	POLYH	A5FD	Reserved to DIGITAL
76FD	CVTHG	A6FD	Reserved to DIGITAL
77FD	Reserved to DIGITAL	A7FD	Reserved to DIGITAL
78FD	Reserved to DIGITAL	A8FD	Reserved to DIGITAL
79FD	Reserved to DIGITAL	A9FD	Reserved to DIGITAL

Table D–3 (Cont.) Two-Byte Opcodes

Hex Value	Mnemonic	Hex Value	Mnemonic
7AFD	Reserved to DIGITAL	AAFD	Reserved to DIGITAL
7BFD	Reserved to DIGITAL	ABFD	Reserved to DIGITAL
7CFD	CLRH, CLRO	ACFD	Reserved to DIGITAL
7DFD	MOVO	ADFD	Reserved to DIGITAL
7EFD	MOVAH, MOVAO	AEFD	Reserved to DIGITAL
7FFD	PUSHAH, PUSHAO	AFFD	Reserved to DIGITAL
80FD	Reserved to DIGITAL	B0FD	Reserved to DIGITAL
81FD	Reserved to DIGITAL	B1FD	Reserved to DIGITAL
82FD	Reserved to DIGITAL	B2FD	Reserved to DIGITAL
83FD	Reserved to DIGITAL	B3FD	Reserved to DIGITAL
84FD	Reserved to DIGITAL	B4FD	Reserved to DIGITAL
85FD	Reserved to DIGITAL	B5FD	Reserved to DIGITAL
86FD	Reserved to DIGITAL	B6FD	Reserved to DIGITAL
87FD	Reserved to DIGITAL	B7FD	Reserved to DIGITAL
88FD	Reserved to DIGITAL	B8FD	Reserved to DIGITAL
89FD	Reserved to DIGITAL	B9FD	Reserved to DIGITAL
8AFD	Reserved to DIGITAL	BAFD	Reserved to DIGITAL
8BFD	Reserved to DIGITAL	BBFD	Reserved to DIGITAL
8CFD	Reserved to DIGITAL	BCFD	Reserved to DIGITAL
8DFD	Reserved to DIGITAL	BDFD	Reserved to DIGITAL
8EFD	Reserved to DIGITAL	BEFD	Reserved to DIGITAL
8FFD	Reserved to DIGITAL	BFFD	Reserved to DIGITAL
C0FD	Reserved to DIGITAL	E0FD	Reserved to DIGITAL
C1FD	Reserved to DIGITAL	E1FD	Reserved to DIGITAL
C2FD	Reserved to DIGITAL	E2FD	Reserved to DIGITAL
C3FD	Reserved to DIGITAL	E3FD	Reserved to DIGITAL
C4FD	Reserved to DIGITAL	E4FD	Reserved to DIGITAL
C5FD	Reserved to DIGITAL	E5FD	Reserved to DIGITAL
C6FD	Reserved to DIGITAL	E6FD	Reserved to DIGITAL
C7FD	Reserved to DIGITAL	E7FD	Reserved to DIGITAL
C8FD	Reserved to DIGITAL	E8FD	Reserved to DIGITAL
C9FD	Reserved to DIGITAL	E9FD	Reserved to DIGITAL
CAFD	Reserved to DIGITAL	EAFD	Reserved to DIGITAL
CBFD	Reserved to DIGITAL	EBFD	Reserved to DIGITAL
CCFD	Reserved to DIGITAL	ECFD	Reserved to DIGITAL
CDFD	Reserved to DIGITAL	EDFD	Reserved to DIGITAL
CEFD	Reserved to DIGITAL	EEFD	Reserved to DIGITAL

Table D–3 (Cont.) Two-Byte Opcodes

Hex Value	Mnemonic	Hex Value	Mnemonic
CFFD	Reserved to DIGITAL	EFFD	Reserved to DIGITAL
D0FD	Reserved to DIGITAL	F0FD	Reserved to DIGITAL
D1FD	Reserved to DIGITAL	F1FD	Reserved to DIGITAL
D2FD	Reserved to DIGITAL	F2FD	Reserved to DIGITAL
D3FD	Reserved to DIGITAL	F3FD	Reserved to DIGITAL
D4FD	Reserved to DIGITAL	F4FD	Reserved to DIGITAL
D5FD	Reserved to DIGITAL	F5FD	Reserved to DIGITAL
D6FD	Reserved to DIGITAL	F6FD	CVTHF
D7FD	Reserved to DIGITAL	F7FD	CVTHD
D8FD	Reserved to DIGITAL	F8FD	Reserved to DIGITAL
D9FD	Reserved to DIGITAL	F9FD	Reserved to DIGITAL
DAFD	Reserved to DIGITAL	FAFD	Reserved to DIGITAL
DBFD	Reserved to DIGITAL	FBFD	Reserved to DIGITAL
DCFD	Reserved to DIGITAL	FCFD	Reserved to DIGITAL
DDFD	Reserved to DIGITAL	FCFE	Reserved to DIGITAL
DEFD	Reserved to DIGITAL	FCFF	Reserved to DIGITAL
DFFD	Reserved to DIGITAL	FDFF	BUGL
		FEFF	BUGW
		FFFF	Reserved for all time

E Exceptions

Exceptions can be grouped into the following six classes:

- Arithmetic traps and faults
- Memory management exceptions
- Exceptions detected during operand reference
- Tracing
- Serious system failures

E.1 Arithmetic Traps and Faults

This section contains the descriptions of the exceptions that occur as the result of performing an arithmetic or conversion operation. They are mutually exclusive and are all assigned the same vector in the system control block (SCB) and the same signal "reason" code. Each exception indicates that an instruction has been completed (trap) or backed up (fault). An appropriate distinguishing exception type code is pushed onto the stack as a longword. Table E–1 lists the arithmetic exception type codes.

Table E–1 Arithmetic Exception Type Codes

Exception Type	Mnemonic	Decimal Value	Hexa-decimal Value
Traps			
integer overflow	SS$_INTOVF	1	1
integer divide-by-zero	SS$_INTDIV	2	2
floating overflow	SS$_FLTOVF	3	3
floating or decimal divide-by-zero	SS$_FLTDIV	4	4
floating underflow	SS$_FLTUND	5	5
decimal overflow	SS$_DECOVF	6	6
subscript range	SS$_SUBRNG	7	7
Faults			
floating underflow	SS$_FLTOVF_F	8	8
floating divide-by-zero	SS$_FLTDIV_F	9	9
floating underflow	SS$_FLTUND_F	10	A

E.1.1 Integer Overflow Trap

An integer overflow trap is an exception indicating that the last instruction executed had an integer overflow, which set the program status longword (PSL) V bit, and that the integer overflow was enabled (the IV bit in the PSL was set). The stored result is the low-order part of the correct result. The N and Z bits in the PSL are set according to the stored result. The type code pushed onto the stack is 1 (SS$_INTOVF).

E.1.2 Integer Divide-by-Zero Trap

An integer divide-by-zero trap is an exception indicating that the last instruction executed had an integer zero divisor. The stored result is equal to the dividend, and condition code V bit in the PSL is set. The type code pushed onto the stack is 2 (SS$_INTDIV).

E.1.3 Floating Overflow Trap

A floating overflow trap is an exception indicating that the last instruction executed resulted in an exponent greater than the largest representable exponent for the data type after normalization and rounding. The stored result contains a one in the sign field and zeros in the exponent and fraction fields. This is a reserved operand. It causes a reserved operand fault if used in a subsequent floating point instruction. The N and V condition code bits in the PSL are set, and the Z and C bits in the PSL are cleared. The type code pushed onto the stack is 3 (SS$_FLTOVF).

E.1.4 Divide-by-Zero Trap

A floating divide-by-zero trap is an exception indicating that the last instruction executed had a floating zero divisor. The stored result is the reserved operand described previously for the floating overflow trap. The condition codes are set as they are for the floating overflow trap.

A decimal string divide-by-zero trap is an exception indicating 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 onto the stack for both types of divide-by-zero is 4 (SS$_FLTDIV).

E.1.5 Floating Underflow Trap

A floating underflow trap is an exception indicating that the last instruction executed resulted in an exponent less than the smallest representable exponent for the data type after normalization and rounding, and that floating underflow was enabled (FU set). The stored result is zero. The N, V, and C condition codes bits in the PSL are cleared, and the Z bit in the PSL is set, except for the polynomial evaluation instruction POLYx. In POLYx, the trap occurs on completion of the instruction, which may be many operations after the underflow. The condition codes are set on the final result in POLYx. The type code pushed onto the stack is 5 (SS$_FLTUND).

E.1.6 Decimal String Overflow Trap

A decimal string overflow trap is an exception indicating that the last instruction executed had a decimal-string result too large for the destination string provided, and that decimal overflow was enabled (the DV bit in the PSL was set). The V condition code bit in the PSL is always set. The type code pushed onto the stack is 6 (SS$_DECOVF).

E.1.7 Subscript-Range Trap

A subscript range trap is an exception indicating that the last instruction was an INDEX instruction with a subscript operand that failed the range check. The value of the subscript operand is lower than the low operand or greater than the high operand. The result is stored in indexout, and the condition codes are set as if the subscript were within range. The type code pushed onto the stack is 7 (SS$_SUBRNG).

E.1.8 Floating Overflow Fault

A floating overflow fault is an exception indicating 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. The POLYx instruction is suspended with the first-part-done bit (FPD) set. The type code pushed onto the stack is 8 (SS$_FLTOVF_F).

E.1.9 Divide-by-Zero Floating Fault

A floating divide-by-zero fault is an exception indicating 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 onto the stack is 9 (SS$_FLTDIV_F).

E.1.10 Floating Underflow Fault

A floating underflow fault is an exception indicating 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 (the FU bit was set). The destination operand is unaffected. The saved condition codes are UNPREDICTABLE. The saved PC points to the instruction causing the fault. The POLYx instruction is suspended with FPD set. The type code pushed onto the stack is 10 (SS$_FLTUND_F).

E.2 Memory Management Exceptions

A memory management exception can be either an access control violation fault or a translation not valid fault.

E.2.1 Access Control Violation Fault

An access control violation fault is an exception indicating that the process attempted a reference not allowed at the current access mode.

E.2.2 Translation Not Valid Fault

A translation not valid fault is an exception indicating that the process attempted a reference to a page for which the valid bit in the page table had not been set.

Note that if a process attempts to reference a page for which the page table entry specifies both translation not valid fault and access control violation, an access control violation fault occurs.

E.3 Exceptions Detected During Operand Reference

Two exceptions are possible during operand reference: the reserved addressing mode fault and the reserved operand exception.

E.3.1 Reserved Addressing Mode Fault

A reserved addressing mode fault is an exception indicating that an operand specifier attempted to use an addressing mode that is disallowed. No parameters are pushed.

E.3.2 Reserved Operand Exception

A reserved operand exception is an exception indicating that an accessed operand has a format reserved for future use by DIGITAL. No parameters are pushed onto the stack. 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 the changes made because of operand specifier evaluation may be restored. Therefore, some instructions are not restartable. These exceptions are labeled as aborts rather than as faults. The saved PC is always restored properly unless the instruction attempted to modify it in a manner that results in UNPREDICTABLE results.

The reserved operand exceptions are caused by the following:

- Bit field too wide
- Invalid combination of bits in PSL restored by the return from interrupt (REI) instruction (fault)
- Invalid combination of bits in PSW mask longword during a return from procedure (RET) instruction (fault)
- Invalid combination of bits in the bit set PSW (BISPSW) or bit clear PSW (BICPSW) instructions (fault)
- Invalid call procedure with stack argument list (CALLS) or call procedure with general argument list (CALLG) instructions entry mask (fault)

- Invalid register number in the move from processor register (MFPR) instruction or move to processor register (MTPR) instruction (fault)
- Invalid PCB contents in the load processor context (LDPCTX) instruction for some implementations (abort)
- Unaligned operand in the add aligned word interlocked (ADAWI) instruction (fault)
- Invalid register contents in the move to processor register (MTPR) instruction for some implementations (fault)
- Invalid operand addresses in insert and remove queue interlocked (INSQHI, INSQTI, REMQHI, or REMQTI) instructions (fault)
- A floating point number that has the sign bit set and the exponent zero in the polynomial evaluation (POLY) instruction table (fault)
- POLY degree too large (fault)
- Decimal string too long (abort)
- Invalid digit in convert trailing numeric to packed (CVTTP) or convert separate numeric to packed (CVTSP) instructions (abort)
- Reserved pattern operator in the edit packed to character string (EDITPC) instruction (fault)
- Incorrect source string length at completion of EDITPC (abort)

E.4 Exceptions Occurring as the Consequence of an Instruction

The following exceptions may occur as a consequence of instruction execution:

- Reserved or privileged instruction fault
- Opcode reserved to customers fault
- Instruction emulation exceptions
- Compatibility mode exception
- Change mode trap
- Breakpoint fault

Each is described in the following subsections.

E.4.1 Reserved or Privileged Instruction Fault

A reserved or privileged instruction fault occurs when the processor encounters an opcode that is not specifically defined or requires higher privileges than the current mode. No parameters are pushed onto the stack. Opcode FFFF (hex) will always fault.

E.4.2 Operand 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 or privileged 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 must 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.

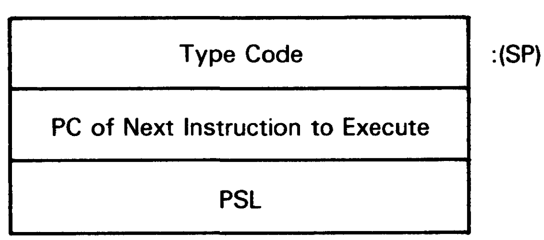
E.4.3 Instruction Emulation Exceptions

When a subset 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 system control block (SCB) vectors, depending on whether or not the first-part-done (FPD) bit in the program status longword was set at the beginning of the instruction. If the FPD bit is clear, a subset emulation trap occurs through the SCB vector at offset CB (hex), and a subset emulation trap frame is pushed onto the current stack. If the FPD bit is set, a suspended emulation fault occurs through the SCB vector at offset CC (hex), and the PC and the PSL are pushed onto the current stack.

E.4.4 Compatibility Mode Exception

A compatibility mode exception is an exception that occurs when the processor is in compatibility mode. A longword of information containing a code that indicates the exception type is pushed onto the stack. Figure E–1 shows the stack frame, which is the same as that for arithmetic exceptions.

Figure E–1 Compatibility Mode Exception Stack Frame



ZK-6351-HC

The compatibility type codes are shown in Table E–2.

Table E–2 Compatibility Mode Exception Type Codes

Exception Type	Decimal Value
Faults	
reserved opcode	0
BPT instruction	1
IOT instruction	2
EMT instruction	3
TRAP instruction	4
illegal instruction	5
Aborts	
odd address	6

All other exceptions in compatibility mode, including the access control violation fault, the translation not valid fault, and the machine check abort, occur by means of the regular native-mode vector.

E.4.5 Change Mode Trap

A change mode trap is an exception occurring when one of the change mode instructions (CHMK, CHME, CHMS, or CHMU) is executed. The instruction operand is pushed onto the exception stack.

E.4.6 Breakpoint Fault

A breakpoint fault is an exception that occurs when the breakpoint instruction (BPT) is executed. The BPT instruction pushes the current PSL onto the stack.

To proceed from a breakpoint fault, a debugger or tracing program does the following:

1. Restores the original contents of the location containing the BPT instruction.
2. Sets the T bit in the PSL saved by the BPT fault. The PSL is on the stack.
3. Resumes operation of the main instruction stream.

When the instruction that has a breakpoint completes execution, a *trace exception* occurs. At this point, the tracing program takes control and does the following:

1. Reinserts the BPT instruction.
2. Restores the T bit to its original state (usually 0).
3. Resumes operation of the main instruction stream.

Note that if both tracing and breakpointing are in progress (if the PSL T bit was set at the time of the BPT), both the BPT restoration and a normal trace exception should be processed on the trace exception by the trace handler.

E.5 Trace Fault

Program tracing is used for many purposes. Debugging programs and evaluating program performance are the most common uses of program tracing.

A trace fault is an exception that occurs between instructions when trace is enabled. One trace fault occurs before the execution of each traced instruction. The address in the PC saved when a trace fault occurs is the address of the instruction after the trace fault that would normally be executed. The trace exception for an instruction takes precedence over all other exceptions. The detection of reserved instruction faults occurs after the trace fault. If a trace fault and a memory management fault (or an odd address abort during a compatibility mode instruction fetch) occur simultaneously, exceptions are taken in UNPREDICTABLE order.

To ensure that exactly one trace occurs per instruction despite other traps and faults, the PSL contains the trace enable (T) and trace pending (TP) bits.

The PSL TP bit generates a fault before any other processing at the start of the next instruction.

The following are rules of operation for trace:

1. At the beginning of an instruction, if the trace pending (TP) bit is set, it is cleared and a trace fault is taken.
2. The value of the trace enable (T) bit is loaded into the trace pending (TP) bit.
3. The detection of interrupts and other exceptions can occur during instruction execution. In this case, TP is cleared before the exception or interrupt is initiated. The system saves the entire PSL including the T bit and TP bit on interrupt or exception initiation and restores the PSL at the end with an REI. This makes interrupts and benign exceptions totally transparent to the executing program.

   The following are conditions and results that might occur during instruction execution or before the next instruction:

   a. If the instruction faults or an interrupt is serviced, the PSL TP bit is cleared before the PSL is saved on the stack. The saved PC (the next lower word on the stack after the saved PSL) is set to the start of the faulting or interrupted instruction. Instruction execution is resumed at step 1.

   b. If the instruction aborts or takes an arithmetic trap, the PSL TP bit is not changed before the PSL is saved on the stack.

   c. If an interrupt is serviced after instruction completion and arithmetic traps but before the presence of tracing is checked at the start of the next instruction, the PSL TP bit is not changed before the PSL is saved on the stack.

E.5.1 Trace Operation When Entering a Change Mode Instruction

The routine entered by a change mode (CHMx) instruction is not traced because change mode clears T and TP in the new PSL that is used for whichever new mode is entered. However, if the T bit was set in the old PSW (the one to be saved) at the beginning of the change mode instruction, the system sets both the T and the TP bit in the saved PSL. Trace faults resume with the instruction that follows other returns from interrupt (REI) in the routine entered by the CHMx instruction. An instruction following an REI faults if T was set when the REI was executed, or if the TP bit in the saved PSL is set. In both cases, TP is set after the REI.

E.5.2 Trace Operation Upon Return From Interrupt

Note that a trace fault that occurs for an instruction following an REI instruction that had set the TP will be taken with the new PSL restored by the REI instruction. Thus, special care must be taken if exception or interrupt routines are traced.

E.5.3 Trace Operation After a BISPSW Instruction

If the T bit is set by a BISPSW instruction, trace faults begin with the second instruction after the BISPSW.

E.5.4 Trace Operation After a CALLS or CALLG Instruction

The CALLS and CALLG instructions save a clear T bit, although the T bit 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.

E.6 Serious System Failures

The following are possible serious system failures:

- Kernel stack not valid abort
- Interrupt stack not valid halt
- Machine check exception

These system failures are described in the following sections.

E.6.1 Kernel Stack Not Valid Abort

The kernel stack not valid abort is an exception indicating that the kernel stack was not valid while the processor was pushing information onto it during the initiation of an exception or interrupt. This is usually an indication of a stack overflow or other operating system error. During this process, the attempted exception is transformed into an abort that uses the interrupt stack. Only the PSL and PC of the original exception are pushed onto the interrupt stack. The interrupt priority level (IPL) is raised to 1F (hex). If the exception vector bits <1:0> are not both 1, the operation of the processor is UNDEFINED.

Software can abort the process without aborting the system. However, because of the lost information, the process cannot be continued. If the kernel stack is not valid during the normal execution of an instruction (including CHMx or REI), the normal memory management fault is initiated.

E.6.2 Interrupt Stack Not Valid Halt

An interrupt stack not valid halt results when the interrupt stack was not valid or a memory error occurred while the processor was pushing information onto the interrupt stack during the initiation of an exception or interrupt. No further interrupt requests are acknowledged on 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 routine can cause the processor to leave the halted state.

E.6.3 Machine Check Exception

A machine check exception indicates that the processor detected an internal error. As is usual for exceptions, a machine check is taken regardless of current interrupt priority level (IPL). The machine check exception vector (bits 0 to 1) 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 a 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).

Software can decide, on the basis of the information presented, whether to abort the current process if the machine check came from the process. The machine check includes any uncorrected bus and memory errors and any other processor-detected errors. Some processor errors cannot ensure the state of the machine at all. For such errors, the state is preserved as well as possible, given the circumstances.

Index

A

Abort
kernel stack not valid • E–10
Absolute expression • 3–9
Absolute mode • 5–14
assembling relative mode as • 6–22
Absolute queue • 9–82 to 9–85
manipulating • 9–85
ACBB (Add Compare and Branch Byte) instruction • 9–44 to 9–45
ACBD (Add Compare and Branch D__floating) instruction • 9–44 to 9–45
ACBF (Add Compare and Branch F__floating) instruction • 9–44 to 9–45
ACBG (Add Compare and Branch G__floating) instruction • 9–44 to 9–45
ACBH (Add Compare and Branch H__floating) instruction • 9–44 to 9–45
ACBL (Add Compare and Branch Long) instruction • 9–44 to 9–45
ACBW (Add Compare and Branch Word) instruction • 9–44 to 9–45
ADAWI (Add Aligned Word Interlocked) instruction • 9–7
ADDB2 (Add Byte 2 Operand) instruction • 9–8
ADDB3 (Add Byte 3 Operand) instruction • 9–8
ADDD2 (Add D_floating 2 Operand) instruction • 9–106
ADDD3 (Add D_floating 3 Operand) instruction • 9–106
ADDF2 (Add F_floating 2 Operand) instruction • 9–106
ADDF3 (Add F_floating 3 Operand) instruction • 9–106
ADDG2 (Add G_floating 2 Operand) instruction • 9–106
ADDG3 (Add G_floating 3 Operand) instruction • 9–106
ADDH2 (Add H_floating 2 Operand) instruction • 9–106
ADDH3 (Add H_floating 3 Operand) instruction • 9–106
ADDL2 (Add Long 2 Operand) instruction • 9–8
ADDL3 (Add Long 3 Operand) instruction • 9–8
ADDP4 (Add Packed 4 Operand) instruction • 9–145 to 9–146
ADDP6 (Add Packed 6 Operand) instruction • 9–145 to 9–146
Address
access type • 8–16
instructions • 9–33 to 9–35
storage directive (.ADDRESS) • 6–4
virtual • 8–1
.ADDRESS directive • 6–4
Addressing mode • 5–1 to 5–19
absolute • 5–14, 6–22
autodecrement • 5–7
autoincrement • 5–5 to 5–6
autoincrement deferred • 5–6 to 5–7
branch • 5–18 to 5–19
determining • 6–67 to 6–68
displacement • 5–8 to 5–9
displacement deferred • 5–9 to 5–10
general • 5–15 to 5–16
general register • 5–1 to 5–12
immediate • 5–14 to 5–15
index • 5–16 to 5–18
literal • 5–10 to 5–12, 5–15
operand specifier formats • 8–17 to 8–27
program counter • 5–12 to 5–16
register • 5–4 to 5–5
register deferred • 5–5
relative • 5–12 to 5–13, 6–19, 6–22
relative deferred • 5–13, 6–19
summary • 5–1, C–9
Address storage directive (.ADDRESS) • 6–4
ADDW2 (Add Word 2 Operand) instruction • 9–8
ADDW3 (Add Word 3 Operand) instruction • 9–8
ADWC (Add with Carry) instruction • 9–9
.ALIGN directive • 6–5 to 6–6
AND operator • 3–16
AOBLEQ (Add One and Branch Less Than or Equal) instruction • 9–46
AOBLSS (Add One and Branch Less Than) instruction • 9–47
Argument
actual • 4–1 to 4–2
in a macro • 4–1 to 4–6
length • 6–63
number of • 6–62
Arithmetic instruction
integer • 9–5 to 9–32

Arithmetic instructions • 9–141 to 9–164
floating-point • 9–101 to 9–123
Arithmetic shift operator • 3–16
.ASCIC directive • 6–8
.ASCID directive • 6–9
ASCII
character set • A–1
operator • 3–13
.ASCII directive • 6–10
ASCII string storage directive • 6–7 to 6–11
counted (.ASCIC) • 6–8
string (.ASCII) • 6–10
string-descriptor (.ASCID) • 6–9
zero-terminated (.ASCIZ) • 6–11
.ASCIZ directive • 6–11
ASHL (Arithmetic Shift Long) instruction • 9–10
ASHP (Arithmetic Shift and Round Packed) instruction • 9–147 to 9–148
ASHQ (Arithmetic Shift Quad) instruction • 9–10
Assembler directives,
summary • C–1
Assembly termination • 6–25
Assembly termination directive (.END) • 6–25
Assignment statement • 1–1, 3–17
Autodecrement mode • 5–7
operand specifier format • 8–19
Autoincrement deferred mode • 5–6 to 5–7
operand specifier format • 8–19
Autoincrement mode • 5–5 to 5–6
operand specifier format • 8–18

B

Base operand specifier • 8–24
BBC (Branch on Bit Clear) instruction • 9–50
BBCC (Branch on Bit Clear and Clear) instruction • 9–51
BBCCI (Branch on Bit Clear and Clear Interlocked) instruction • 9–52
BBCS (Branch on Bit Clear and Set) instruction • 9–51
BBS (Branch on Bit Set) instruction • 9–50
BBSC (Branch on Bit Set and Clear) instruction • 9–51
BBSS (Branch on Bit Set and Set) instruction • 9–51
BBSSI (Branch on Bit Set and Set Interlocked) instruction • 9–52
BCC (Branch on Carry Clear) instruction • 9–48 to 9–49
BCS (Branch on Carry Set) instruction • 9–48 to 9–49
BEQL (Branch on Equal) instruction • 9–48 to 9–49
BEQLU (Branch on Equal Unsigned) instruction • 9–48 to 9–49
BGEQ (Branch on Greater Than or Equal) instruction • 9–48 to 9–49
BGEQU (Branch on Greater Than or Equal Unsigned) instruction • 9–48 to 9–49
BGTR (Branch on Greater Than) instruction • 9–48 to 9–49
BGTRU (Branch on Greater Than Unsigned) instruction • 9–48 to 9–49
BICB2 (Bit Clear Byte 2 Operand) instruction • 9–11
BICB3 (Bit Clear Byte 3 Operand) instruction • 9–11
BICL2 (Bit Clear Long 2 Operand) instruction • 9–11
BICL3 (Bit Clear Long 3 Operand) instruction • 9–11
BICPSW (Bit Clear PSW) instruction • 9–71
BICW2 (Bit Clear Word 2 Operand) instruction • 9–11
BICW3 (Bit Clear Word 3 Operand) instruction • 9–11
Binary operator • 3–15 to 3–16
summary • C–8
BISB2 (Bit Set Byte 2 Operand) instruction • 9–12
BISB3 (Bit Set Byte 3 Operand) instruction • 9–12
BISL2 (Bit Set Long 2 Operand) instruction • 9–12
BISL3 (Bit Set Long 3 Operand) instruction • 9–12
BISPSW (Bit Set PSW) instruction • 9–72
BISW2 (Bit Set Word 2 Operand) instruction • 9–12
BISW3 (Bit Set Word 3 Operand) instruction • 9–12
BITB (Bit Test Byte) instruction • 9–13
BITL (Bit Test Long) instruction • 9–13
BITW (Bit Test Word) instruction • 9–13
BLBC (Branch on Low Bit Clear) instruction • 9–53
BLBS (Branch on Low Bit Set) instruction • 9–53
BLEQ (Branch on Less Than or Equal) instruction • 9–48 to 9–49
BLEQU (Branch on Less Than or Equal Unsigned) instruction • 9–48 to 9–49
Block storage allocation directives (.BLKx) • 6–12 to 6–13
BLSS (Branch on Less Than) instruction • 9–48 to 9–49
BLSSU (Branch on Less Than Unsigned) instruction • 9–48 to 9–49
BNEQ (Branch on Not Equal) instruction • 9–48 to 9–49
BNEQU (Branch on Not Equal Unsigned) instruction • 9–48 to 9–49

BPT (Breakpoint Fault) instruction • 9–73
Branch access type • 8–16
Branch mode • 5–18 to 5–19
  operand specifier format • 8–26 to 8–27
BRB (Branch Byte Displacement) instruction • 9–54
BRW (Branch Word Displacement) instruction • 9–54
BSBB (Branch to Subroutine Byte Displacement) instruction • 9–55
BSBW (Branch to Subroutine Word Displacement) instruction • 9–55
BUGL (Bugcheck Longword Message Identifier) instruction • 9–193
BUGW (Bugcheck Word Message Identifier) instruction • 9–193
BVC (Branch on Overflow Clear) instruction • 9–48 to 9–49
BVS (Branch on Overflow Set) instruction • 9–48 to 9–49
Byte data type • 8–1
.BYTE directive • 6–14 to 6–15
Byte storage directive (.BYTE) • 6–14 to 6–15

C

Call frame • 9–63
CALLG (Call Procedure With General Argument List) instruction • 9–65 to 9–66
CALLS (Call Procedure with Stack Argument List) instruction • 9–67 to 9–68
Carry condition code (C) • 8–14
CASEB (Case Byte) instruction • 9–56
CASEL (Case Long) instruction • 9–56
CASEW (Case Word) instruction • 9–56
Character set
  in source statement • 3–1 to 3–2
  special characters • C–6 to C–7
  table • A–1
Character string
  data type • 8–7
  instructions • 9–124 to 9–137
  length • 6–63
CHME (Change Mode to Executive) instruction • 9–186 to 9–187
CHMK (Change Mode to Kernel) instruction • 9–186 to 9–187
CHMS (Change Mode to Supervisor) instruction • 9–186 to 9–187
CHMU (Change Mode to User) instruction • 9–186 to 9–187
CLRB (Clear Byte) instruction • 9–14
CLRD (Clear D_floating) instruction • 9–107
CLRF (Clear F_floating) instruction • 9–107
CLRG (Clear G_floating) instruction • 9–107
CLRH (Clear H_floating) instruction • 9–107
CLRL (Clear Long) instruction • 9–14
CLRO (Clear Octa) instruction • 9–14
CLRQ (Clear Quad) instruction • 9–14
CLRW (Clear Word) instruction • 9–14
CMPB (Compare Byte) instruction • 9–15
CMPC3 (Compare Characters 3 Operand) instruction • 9–126 to 9–127
CMPC5 (Compare Characters 5 Operand) instruction • 9–126 to 9–127
CMPD (Compare D_floating) instruction • 9–108
CMPF (Compare F_floating) instruction • 9–108
CMPG (Compare G_floating) instruction • 9–108
CMPH (Compare H_floating) instruction • 9–108
CMPL (Compare Long) instruction • 9–15
CMPP3 (Compare Packed 3 Operand) instruction • 9–149
CMPP4 (Compare Packed 4 Operand) instruction • 9–149
CMPV (Compare Field) instruction • 9–38
CMPW (Compare Word) instruction • 9–15
CMPZV (Compare Zero Extended Field) instruction • 9–38
Colon (:)
  in label field • 2–2
Complement operator • 3–15
Conditional assembly block directive
  .ENDC • 6–26
  (.IF) • 6–39 to 6–41
  listing unsatisfied code • 6–87
Condition code • 8–13 to 8–15, 9–4
  carry (C) • 8–14
  negative (N) • 8–14
  overflow (V) • 8–14
  zero (Z) • 8–14
Continuation character (-)
  in listing file • 3–9
  in source statement • 2–1
Control instructions • 9–42 to 9–62
CRC (Calculate Cyclic Redundancy Check) instruction • 9–139 to 9–140
Created local label • 4–7
  range • 3–7
.CROSS directive • 6–16 to 6–17
Cross-reference directive
  .CROSS • 6–16 to 6–17
  .NOCROSS • 6–16 to 6–17
  (.NOCROSS) • 6–65

Current location counter • 3–17 to 3–18
CVTBD (Convert Byte to D_floating) instruction • 9–109 to 9–111
CVTBF (Convert Byte to F_floating) instruction • 9–109 to 9–111
CVTBG (Convert Byte to G_floating) instruction • 9–109 to 9–111
CVTBH (Convert Byte to H_floating) instruction • 9–109 to 9–111
CVTBL (Convert Byte to Long) instruction • 9–16
CVTBW (Convert Byte to Word) instruction • 9–16
CVTDB (Convert D_floating to Byte) instruction • 9–109 to 9–111
CVTDF (Convert D_floating to F_floating) instruction • 9–109 to 9–111
CVTDH (Convert D_floating to H_floating) instruction • 9–109 to 9–111
CVTDL (Convert D_floating to Long) instruction • 9–109 to 9–111
CVTDW (Convert D_floating to Word) instruction • 9–109 to 9–111
CVTFB (Convert F_floating to Byte) instruction • 9–109 to 9–111
CVTFD (Convert F_floating to D_floating) instruction • 9–109 to 9–111
CVTFG (Convert F_floating to G_floating) instruction • 9–109 to 9–111
CVTFH (Convert F_floating to H_floating) instruction • 9–109 to 9–111
CVTFL (Convert F_floating to Long) instruction • 9–109 to 9–111
CVTFW (Convert F_floating to Word) instruction • 9–109 to 9–111
CVTGB (Convert G_floating to Byte) instruction • 9–109 to 9–111
CVTGF (Convert G_floating to F_floating) instruction • 9–109 to 9–111
CVTGH (Convert G_floating to H_floating) instruction • 9–109 to 9–111
CVTGL (Convert G_floating to Long) instruction • 9–109 to 9–111
CVTGW (Convert G_floating to Word) instruction • 9–109 to 9–111
CVTHB (Convert H_floating to Byte) instruction • 9–109 to 9–111
CVTHD (Convert H_floating to D_floating) instruction • 9–109 to 9–111
CVTHF (Convert H_floating to F_floating) instruction • 9–109 to 9–111
CVTHG (Convert H_floating to G_floating) instruction • 9–109 to 9–111
CVTHL (Convert H_floating to Long) instruction • 9–109 to 9–111
CVTHW (Convert H_floating to Word) instruction • 9–109 to 9–111
CVTLB (Convert Long to Byte) instruction • 9–16
CVTLD (Convert Long to D_floating) instruction • 9–109 to 9–111
CVTLF (Convert Long to F_floating) instruction • 9–109 to 9–111
CVTLG (Convert Long to G_floating) instruction • 9–109 to 9–111
CVTLH (Convert Long to H_floating) instruction • 9–109 to 9–111
CVTLP (Convert Long to Packed) instruction • 9–150
CVTLW (Convert Long to Word) instruction • 9–16
CVTPL (Convert Packed to Long) instruction • 9–151
CVTPS (Convert Packed to Leading Separate Numeric) instruction • 9–152 to 9–153
CVTPT (Convert Packed to Trailing Numeric) instruction • 9–154 to 9–155
CVTRDL (Convert Rounded D_floating to Long) instruction • 9–109 to 9–111
CVTRFL (Convert Rounded F_floating to Long) instruction • 9–109 to 9–111
CVTRGL (Convert Rounded G_floating to Long) instruction • 9–109 to 9–111
CVTRHL (Convert Rounded H_floating to Long) instruction • 9–109 to 9–111
CVTSP (Convert Leading Separate Numeric to Packed) instruction • 9–156
CVTTP (Convert Trailing Numeric to Packed) instruction • 9–157 to 9–158
CVTWB (Convert Word to Byte) instruction • 9–16
CVTWD (Convert Word to D_floating) instruction • 9–109 to 9–111
CVTWF (Convert Word to F_floating) instruction • 9–109 to 9–111
CVTWG (Convert Word to G_floating) instruction • 9–109 to 9–111
CVTWH (Convert Word to H_floating) instruction • 9–109 to 9–111
CVTWL (Convert Word to Long) instruction • 9–16
Cyclic redundancy check instruction • 9–138 to 9–140

D

D_floating data type • 9–102
.D_FLOATING directive • 6–20
Data storage directive
  .ADDRESS • 6–4

Data storage directive (cont'd.)
.ASCIC • 6–8
.ASCID • 6–9
.ASCII • 6–10
.ASCIZ • 6–11
.BYTE • 6–14 to 6–15
.D_FLOATING • 6–20
F_FLOATING • 6–34
G_FLOATING • 6–35
H_FLOATING • 6–37
.LONG • 6–55
.OCTA • 6–69
.PACKED • 6–73
.QUAD • 6–80
.SIGNED_BYTE • 6–89
.SIGNED_WORD • 6–90 to 6–91
.WORD • 6–99
Data type • 8–1 to 8–13
byte • 8–1
character string • 8–7
floating-point • 8–3 to 8–5, 9–101 to 9–102
integer • 8–1 to 8–3
leading separate numeric string • 8–11 to 8–12
longword • 8–2
octaword • 8–3
packed decimal string • 8–12 to 8–13
quadword • 8–2
string • 8–7 to 8–13
trailing numeric string • 8–7 to 8–11
variable-length bit field • 8–5 to 8–6
word • 8–1
.DEBUG directive • 6–18
Debug directive (.DEBUG) • 6–18
Debugger
module name • 6–23
routine name • 6–23
DECB (Decrement Byte) instruction • 9–17
Decimal/hexadecimal conversion • B–1
table • B–2
Decimal overflow enable (DV) • 8–15
Decimal string instructions • 9–141 to 9–164
DECL (Decrement Long) instruction • 9–17
DECW (Decrement Word) instruction • 9–17
.DEFAULT directive • 6–19
Default displacement length directive (.DEFAULT) • 6–19
Delimiter
string argument • 4–3
Direct assignment statement • 1–1, 3–17
Directive • 1–1 to 1–2, 6–1 to 6–99
as operator • 2–3
Directive (cont'd.)
general assembler • 1–1, 6–1, 6–1 to 6–2
macro • 1–1, 6–1, 6–2 to 6–3
summary • C–1 to C–5
Disable assembler functions directive (.DISABLE) • 6–21
.DISABLE directive • 6–21
Displacement deferred mode • 5–9 to 5–10
operand specifier formats • 8–20 to 8–21
Displacement mode • 5–8 to 5–9
operand specifier formats • 8–20
DIVB2 (Divide Byte 2 Operand) instruction • 9–18
DIVB3 (Divide Byte 3 Operand) instruction • 9–18
DIVD2 (Divide D_floating 2 Operand) instruction • 9–112 to 9–113
DIVD3 (Divide D_floating 3 Operand) instruction • 9–112 to 9–113
DIVF2 (Divide F_floating 2 Operand) instruction • 9–112 to 9–113
DIVF3 (Divide F_floating 3 Operand) instruction • 9–112 to 9–113
DIVG2 (Divide G_floating 2 Operand) instruction • 9–112 to 9–113
DIVG3 (Divide G_floating 3 Operand) instruction • 9–112 to 9–113
DIVH2 (Divide H_floating 2 Operand) instruction • 9–112 to 9–113
DIVH3 (Divide H_floating 3 Operand) instruction • 9–112 to 9–113
Divide-by-zero trap • 8–15
DIVL2 (Divide Long 2 Operand) instruction • 9–18
DIVL3 (Divide Long 3 Operand) instruction • 9–18
DIVP (Divide Packed) instruction • 9–159 to 9–160
DIVW2 (Divide Word 2 Operand) instruction • 9–18
DIVW3 (Divide Word 3 Operand) instruction • 9–18
.DOUBLE directive • 6–20

E

Edit
instruction • 9–165 to 9–182
pattern operator • 9–166, 9–168 to 9–182
EDITPC (Edit Packed to Character String) instruction • 9–166 to 9–182
EDIV (Extended Divide) instruction • 9–19
EMODD (Extended Multiply and Integerize D_floating) instruction • 9–114 to 9–115
EMODF (Extended Multiply and Integerize F_floating) instruction • 9–114 to 9–115

Index

EMODG (Extended Multiply and Integerize G_ floating) instruction • 9–114 to 9–115
EMODH (Extended Multiply and Integerize H_ floating) instruction • 9–114 to 9–115
EMUL (Extended Multiply) instruction • 9–20
Enable assembler functions • 6–22 to 6–24
.ENABLE directive • 6–22 to 6–24, 6–33
.ENDC directive • 6–26
End conditional assembly directive (.END) • 6–26
.END directive • 6–25
End macro definition directive (.ENDM) • 6–27
.ENDM directive • 6–27
.ENDR directive • 6–28
.ENTRY directive • 6–29 to 6–30
Entry mask • 9–63
Entry point
  defining • 6–29 to 6–30
Entry point directive (.ENTRY) • 6–29 to 6–30
EO$ADJUST_INPUT (Adjust Input Length) pattern operator • 9–171
EO$BLANK_ZERO (Blank Backwards when Zero) pattern operator • 9–172
EO$CLEAR_SIGNIF (Clear Significance) pattern operator • 9–181
EO$END (End Edit) pattern operator • 9–173
EO$END_FLOAT (End Floating Sign) pattern operator • 9–174
EO$FILL (Store Fill) pattern operator • 9–175
EO$FLOAT (Float Sign) pattern operator • 9–176
EO$INSERT (Insert Character) pattern operator • 9–177
EO$LOAD_FILL (Load Fill Register) pattern operator • 9–178
EO$LOAD_MINUS (Load Sign Register If Minus) pattern operator • 9–178
EO$LOAD_PLUS (Load Sign Register If Plus) pattern operator • 9–178
EO$LOAD_SIGN (Load Sign Register) pattern operator • 9–178
EO$MOVE (Move Digits) pattern operator • 9–179
EO$REPLACE_SIGN (Replace Sign when Zero) pattern operator • 9–180
EO$SET_SIGNIF (Set Significance) pattern operator • 9–181
EO$STORE_SIGN (Store Sign) pattern operator • 9–182
.ERROR directive • 6–31
.EVEN directive • 6–32
Exception • E–1
  access control violation • E–4
  arithmetic • E–1
  arithmetic type code • E–1
Exception (cont'd.)
  breakpoint • E–7
  change mode • E–7
  compatibility mode • E–6
    type code • E–7
  control • 8–13 to 8–15
  customer reserved opcode • E–6
  decimal
    string overflow • E–3
  floating
    divide-by-zero • E–2, E–3
    overflow • E–2, E–3
    underflow • E–2, E–3
  instruction
    emulation • E–6
    execution • E–5
  integer
    divide-by-zero • E–2
    overflow • E–2
  kernel stack not valid • E–10
  machine check • E–10
  memory management • E–3
  operand reference • E–4
  reserved
    addressing mode • E–4
    operand • E–4
  subscript-range • E–3
  trace • E–8
  trace operation • E–8
  translation not valid • E–4
Exclusive OR operator • 3–16
Expression • 3–9 to 3–10
  absolute • 3–9
  evaluation of • 3–9
  example of • 3–10
  external • 3–9
  global • 3–9
  relocatable • 3–9, 3–18
Extent
  syntax • 7–1
.EXTERNAL directive • 6–33
External expression • 3–9
External symbol • 6–98
  attribute directive (.EXTERNAL) • 6–33
  defining • 6–22, 6–33
%EXTRACT operator • 4–10 to 4–11
EXTV (Extract Field) instruction • 9–39
EXTZV (Extract Zero Extended Field) instruction • 9–39

F

F_floating • 8–3
F_floating data type • 9–102
.F_FLOATING directive • 6–34
Fault
  access control violation • E–4
  arithmetic • E–1
  arithmetic type code • E–1
  breakpoint • E–7
  customer reserved opcode • E–6
  floating
    divide-by-zero • E–3
    overflow • E–2, E–3
    underflow • E–3
  instruction execution • E–5
  memory management • E–3
  privileged instruction • E–5
  reserved
    addressing mode • E–4
    opcode • E–5
  trace • E–8
  translation not valid • E–4
FFC (Find First Clear) instruction • 9–40
FFS (Find First Set) instruction • 9–40
Field • 2–1 to 2–4
  comment • 2–1, 2–3 to 2–4
  label • 2–1, 2–2
  Must Be Zero (MBZ) • 7–1
  operand • 2–3
  operator • 2–3
  variable-length bit • 8–5 to 8–6
.FLOAT directive • 6–34
Floating overflow fault • 8–15
Floating-point
  accuracy • 9–103 to 9–104
  rounding • 9–103 to 9–104
  zero • 9–102
Floating-point constants (.D_FLOATING) • 6–20
Floating-point data type • 8–3 to 8–5, 9–101 to 9–102
  D_floating • 8–4
  G_floating • 8–4
  H_floating • 8–5
Floating-point instructions • 9–101 to 9–123
Floating-point number • 9–101
  format • 3–3
  F_floating • 6–34
  G_floating • 6–35
  H_floating • 6–37
Floating-point number (cont'd.)
  in source statement • 3–3 to 3–4
  rounding • 6–23
  storage • 6–20
  storing • 6–34, 6–35, 6–37
  truncating • 6–23
Floating-point operator • 3–14
Floating-point storage directive
  .D_FLOATING • 6–20
  .F_FLOATING • 6–34
  .G_FLOATING • 6–35
Floating underflow enable (FU) • 8–14
Formal argument • 4–1 to 4–2
Frame
  call • 9–63
  stack • 9–63

G

G_floating data type • 9–102
.G_FLOATING directive • 6–35
General mode • 5–15 to 5–16
General register mode • 5–1 to 5–12
.GLOBAL directive • 6–36
Global expression • 3–9
Global label • 2–2
Global symbol • 3–6, 6–98
  attribute directive (.GLOBAL) • 6–36
  defining • 6–22, 6–33, 6–36
  defining for shareable image • 6–94 to 6–96

H

.H_FLOATING directive • 6–37
H_floating-point storage directive (.H_FLOATING) • 6–37
Halt
  interrupt stack not valid • E–10
HALT (Halt) instruction • 9–74
Hexadecimal/decimal conversion • B–1
  table • B–2

I

.IDENT directive • 6–38
Identification directive (.IDENT) • 6–38

.IF directive • 6–39 to 6–41
.IF_FALSE directive • 6–42 to 6–44
.IF_TRUE directive • 6–42 to 6–44
.IF_TRUE_FALSE directive • 6–42 to 6–44
.IIF directive • 6–45
Immediate conditional assembly block directive (.IIF) • 6–45
Immediate mode • 5–14 to 5–15
  contrasted with literal mode • 5–15
INCB (Increment Byte) instruction • 9–21
INCL (Increment Long) instruction • 9–21
Inclusive OR operator • 3–16
INCW (Increment Word) instruction • 9–21
Indefinite repeat argument directive (.IRP) • 6–46 to 6–47
Indefinite repeat character directive (.IRPC) • 6–48 to 6–49
INDEX (Compute Index) instruction • 9–75 to 9–76
Index mode • 5–16 to 5–18
  operand specifier format • 8–23 to 8–24
INSQHI (Insert Entry into Queue at Head, Interlocked) instruction • 9–89 to 9–90
INSQTI (Insert Entry into Queue at Tail, Interlocked) instruction • 9–91 to 9–92
INSQUE (Insert Entry in Queue) instruction • 9–93 to 9–94
Instruction • 1–1, 9–1 to 9–193
  address • 9–33 to 9–35
  arithmetic • 9–5 to 9–32, 9–101 to 9–123, 9–141 to 9–164
  as operator • 2–3
  character string • 9–124 to 9–137
  control • 9–42 to 9–62
  decimal string • 9–141 to 9–164
  floating-point • 9–101 to 9–123
  format • 8–15 to 8–27
  integer • 9–5 to 9–32
  logical • 9–5 to 9–32
  packed decimal • 9–141 to 9–164
  procedure call • 9–63 to 9–69
  queue • 9–82 to 9–100
  string • 9–124 to 9–137, 9–141 to 9–164
  variable-length bit field • 9–36 to 9–41
Instruction notation
  operand specifier • 9–2 to 9–3
  operation description • 9–3 to 9–4
INSV (Insert Field) instruction • 9–41
Integer
  data type • 8–1 to 8–3
  in source statement • 3–3
  unsigned • 8–1, 8–2
Integer instructions • 9–5 to 9–32
Integer overflow enable (IV) • 8–14
.IRPC directive • 6–48 to 6–49
.IRP directive • 6–46 to 6–47

J

JMP (Jump) instruction • 9–58
JSB (Jump To Subroutine) instruction • 9–59

K

Keyword argument • 4–3

L

Label
  created local • 4–7
  global • 2–2
  user-defined local • 3–7 to 3–8, 4–7
LDPCTX (Load Process Context) instruction • 9–189
Leading separate numeric string
  data type • 8–11 to 8–12
%LENGTH operator • 4–8 to 4–9
.LIBRARY directive • 6–50
.LINK directive • 6–51 to 6–53
  /INCLUDE qualifier • 6–51
  /LIBRARY qualifier • 6–51
  /SELECTIVE_SEARCH qualifier • 6–52
  /SHAREABLE qualifier • 6–52
.LIST directive • 6–54
  See also .SHOW directive
Listing
  table of contents • 6–92
Listing control directive
  .IDENT • 6–38
  .LIST • 6–54
  .NLIST • 6–64
  .NOSHOW • 6–66, 6–87 to 6–88
  .PAGE • 6–74
  .SHOW • 6–87 to 6–88
Listing level count • 6–88
Literal mode • 5–10 to 5–12
  contrasted with immediate mode • 5–15
  operand specifier format • 8–21 to 8–23

Local label
saving • 6–85 to 6–86
user-defined • 3–7 to 3–8
Local label block
ending • 6–22
starting • 6–22
Local symbol • 3–6
%LOCATE operator • 4–9 to 4–10
Location control directive
.ALIGN • 6–5 to 6–6
.BLKx • 6–12 to 6–13
Location counter alignment directive
(.ODD) • 6–70
Location counter control directive
(.EVEN) • 6–32
LOCC (Locate Character) instruction • 9–128
Logical AND operator
See AND operator
Logical exclusive OR operator
See Exclusive OR operator
Logical inclusive OR operator
See Inclusive OR operator
Logical instruction • 9–5 to 9–32
.LONG directive • 6–55
Longword data type • 8–2
Longword storage directive (.LONG) • 6–55

M

Macro • 4–1 to 4–11
nested • 4–4 to 4–5
passing numeric value to • 4–6
with the same name as an opcode • 6–57
Macro argument • 4–1 to 4–6
actual • 4–1 to 4–2
concatenated • 4–5 to 4–6
delimited • 4–3 to 4–4, 4–5
formal • 4–1 to 4–2
keyword • 4–3
positional • 4–3
string • 4–3 to 4–5
Macro call • 4–1
as operator • 2–3
listing • 6–87
number of arguments • 6–62
Macro call directive (.MCALL) • 6–59
Macro definition • 4–1
default value • 4–2
end • 6–27
Macro definition (cont'd.)
labeling in • 4–7
listing • 6–87
Macro definition directive
(.MACRO) • 6–56 to 6–57
Macro deletion directive (.MDELETE) • 6–60
.MACRO directive • 6–56 to 6–57
Macro exit directive (.MEXIT) • 6–61
Macro expansion
listing • 6–87
printing • 4–1
terminating • 6–61
Macroinstruction
See Macro
Macro library
adding a name to • 6–50
Macro library directive (.LIBRARY) • 6–50
Macro link directive (.LINK) • 6–51 to 6–53
Macro name • 3–6
Macro operator
%EXTRACT • 4–10 to 4–11
%LENGTH • 4–8 to 4–9
%LOCATE • 4–9 to 4–10
string • 4–8 to 4–11
Macro string operator
summary • C–8
Mask
entry • 9–63
register • 3–13 to 3–14
register save • 6–29, 6–58
.MASK directive • 6–58
MATCHC (Match Characters) instruction • 9–129
MBZ field • 7–1
.MCALL directive • 6–59
MCOMB (Move Complemented Byte) instruction • 9–22
MCOML (Move Complemented Long) instruction • 9–22
MCOMW (Move Complemented Word) instruction • 9–22
.MDELETE directive • 6–60
Memory management
exception • E–3
fault • E–3
Message display directive
(.ERROR) • 6–31
(.PRINT) • 6–75
Message warning display directive
(.WARN) • 6–97
.MEXIT directive • 6–61
MFPR (Move from Processor Register) instruction • 9–192

MNEGB (Move Negated Byte) instruction • 9–23
MNEGD (Move Negated D_floating) instruction • 9–116
MNEGF (Move Negated F_floating) instruction • 9–116
MNEGG (Move Negated G_floating) instruction • 9–116
MNEGH (Move Negated H_floating) instruction • 9–116
MNEGL (Move Negated Long) instruction • 9–23
MNEGW (Move Negated Word) instruction • 9–23
Modify access type • 8–16
Module name
  made available to debugger • 6–23
MOVAB (Move Address Byte) instruction • 9–34
MOVAD (Move Address D_floating) instruction • 9–34
MOVAF (Move Address F_floating) instruction • 9–34
MOVAG (Move Address G_floating) instruction • 9–34
MOVAH (Move Address H_floating) instruction • 9–34
MOVAL (Move Address Long) instruction • 9–34
MOVAO (Move Address Octa) instruction • 9–34
MOVAQ (Move Address Quad) instruction • 9–34
MOVAW (Move Address Word) instruction • 9–34
MOVB (Move Byte) instruction • 9–24
MOVC3 (Move Character 3 Operand) instruction • 9–130 to 9–131
MOVC5 (Move Character 5 Operand) instruction • 9–130 to 9–131
MOVD (Move D_floating) instruction • 9–117
MOVF (Move F_floating) instruction • 9–117
MOVG (Move G_floating) instruction • 9–117
MOVH (Move H_floating) instruction • 9–117
MOVL (Move Long) instruction • 9–24
MOVO (Move Octa) instruction • 9–24
MOVP (Move Packed) instruction • 9–161
MOVPSL (Move PSL) instruction • 9–77
MOVQ (Move Quad) instruction • 9–24
MOVTC (Move Translated Characters) instruction • 9–132
MOVTUC (Move Translated Until Character) instruction • 9–133 to 9–134
MOVW (Move Word) instruction • 9–24
MOVZBL (Move Zero-Extended Byte to Long) instruction • 9–25
MOVZBW (Move Zero-Extended Byte to Word) instruction • 9–25
MOVZWL (Move Zero-Extended Word to Long) instruction • 9–25
MTPR (Move to Processor Register) instruction • 9–191
MULB2 (Multiply Byte 2 Operand) instruction • 9–26
MULB3 (Multiply Byte 3 Operand) instruction • 9–26
MULD2 (Multiply D_floating 2 Operand) instruction • 9–118
MULD3 (Multiply D_floating 3 Operand) instruction • 9–118
MULF2 (Multiply F_floating 2 Operand) instruction • 9–118
MULF3 (Multiply F_floating 3 Operand) instruction • 9–118
MULG2 (Multiply G_floating 2 Operand) instruction • 9–118
MULG3 (Multiply G_floating 3 Operand) instruction • 9–118
MULH2 (Multiply H_floating 2 Operand) instruction • 9–118
MULH3 (Multiply H_floating 3 Operand) instruction • 9–118
MULL2 (Multiply Long 2 Operand) instruction • 9–26
MULL3 (Multiply Long 3 Operand) instruction • 9–26
MULP (Multiply Packed) instruction • 9–162
MULW2 (Multiply Word 2 Operand) instruction • 9–26
MULW3 (Multiply Word 3 Operand) instruction • 9–26
Must Be Zero
  See also MBZ
  See Field

N

.NARG directive • 6–62
.NCHR directive • 6–63
Negative condition code (N) • 8–14
.NLIST directive • 6–64
  See also .NOSHOW directive
.NOCROSS directive • 6–16 to 6–17, 6–65
NOP (No Operation) instruction • 9–78
.NOSHOW directive • 6–66, 6–87 to 6–88
.NTYPE directive • 6–67 to 6–68
Number
  See also Integer, Floating-point number, and Packed decimal string
  in source statement • 3–2 to 3–4
Number of arguments directive (.NARG) • 6–62

Number of characters directive (.NCHR) • 6–63
Numeric control operator • 3–14 to 3–15
Numeric string
  leading separate • 8–11 to 8–12
  trailing • 8–7 to 8–11

O

Object module
  identifying • 6–38
  naming • 6–93
  title • 6–93
.OCTA directive • 6–69
Octaword data type • 8–3
Octaword storage directive (.OCTA) • 6–69
.ODD directive • 6–70
One's complement
  of expression • 3–15
Opcode
  creating • 6–71 to 6–72
  defining • 6–81
  format • 8–15
  redefining • 6–57, 6–71 to 6–72
  summary • D–1 to D–17
    alphabetic order • D–1
    numeric order • D–10
  with the same name as a macro • 6–57
Opcode definition directive (.OPDEF) • 6–71 to 6–72
.OPDEF directive • 6–71 to 6–72
Operand • 2–3
  determining addressing mode of • 6–67 to 6–68
  primary • 8–24
  reserved • 9–102, 9–103, 9–142
Operand generation directive
  (.REF16) • 6–81
  (.REF2) • 6–81
  (.REF4) • 6–81
  (.REF8) • 6–81
Operand specifier • 8–16 to 8–27
  access type notation • 9–2
  access types • 8–16
  base • 8–24
  data type notation • 9–2 to 9–3
  data types • 8–16
  notation • 9–2 to 9–3
Operand specifier addressing mode formats • 8–17 to 8–27
  autodecrement mode • 8–19
Operand specifier addressing mode formats (cont'd.)
  autoincrement deferred mode • 8–19
  autoincrement mode • 8–18
  branch mode • 8–26 to 8–27
  displacement deferred mode • 8–20 to 8–21
  displacement mode • 8–20
  index mode • 8–23 to 8–24
  literal mode • 8–21 to 8–23
  register deferred mode • 8–18
  register mode • 8–17 to 8–18
Operand type directive (.NTYPE) • 6–67 to 6–68
Operator • 2–3
  AND • 3–16
  arithmetic shift • 3–16
  ASCII • 3–13
  binary • 3–15 to 3–16, C–8
  complement • 3–15
  exclusive OR • 3–16
  floating-point • 3–14
  inclusive OR • 3–16
  macro • 4–8 to 4–11
  macro string • C–8
  numeric control • 3–14 to 3–15
  pattern • 9–168 to 9–182
  radix control • 3–11 to 3–12
  register • 3–13 to 3–14
  summary • C–7 to C–8
  textual • 3–12 to 3–14
  unary • 3–10 to 3–11, C–7
Overflow condition code (V) • 8–14

P

Packed decimal instructions • 9–141 to 9–164
Packed decimal string • 9–141 to 9–143
  data type • 8–12 to 8–13
  format • 3–4
  in source statement • 3–4
  storing • 6–73
Packed decimal string directive (.PACKED) • 6–73
.PACKED directive • 6–73
Page ejection directive (.PAGE) • 6–74
Pattern operator • 9–166, 9–168 to 9–182
Period (.)
  current location counter • 3–17
Permanent symbol • 3–4 to 3–5, 3–6
Permanent symbol table • D–1 to D–17
POLYD (Polynomial Evaluation D_floating) instruction • 9–119 to 9–121

POLYF (Polynomial Evaluation F_floating) instruction • 9–119 to 9–121
POLYG (Polynomial Evaluation G_floating) instruction • 9–119 to 9–121
POLYH (Polynomial Evaluation H_floating) instruction • 9–119 to 9–121
POPL instruction • 9–27
POPR (Pop Registers) instruction • 9–79
Positional argument • 4–3
Primary operand • 8–24
.PRINT directive • 6–75
PROBER (Probe Read) instruction • 9–184 to 9–185
PROBEW (Probe Write) instruction • 9–184 to 9–185
Procedure call instructions • 9–63 to 9–69
Processor status longword • 8–13
Processor status word • 8–13 to 8–15
  condition codes • 8–13 to 8–14
  decimal overflow enable (DV) • 8–15
  floating underflow enable (FU) • 8–14
  integer overflow enable (IV) • 8–14
  trace trap enable (T) • 8–14
Program counter mode • 5–12 to 5–16
Program section
  absolute • 6–78, 6–79
  alignment • 6–79
  attributes • 6–76 to 6–78, 6–79
  defining • 6–76 to 6–79
  directive
    (.PSECT) • 6–76 to 6–79
    (.RESTORE_PSECT) • 6–84
    (.SAVE_PSECT) • 6–85 to 6–86
  name • 6–76, 6–79
  restoring context of • 6–84
  saving context of • 6–85 to 6–86
  saving local label • 6–85 to 6–86
  unnamed • 6–78, 6–79
.PSECT directive • 6–76 to 6–79
PSW
  See Processor status word
PUSHAB (Push Address Byte) instruction • 9–35
PUSHAD (Push Address D_floating) instruction • 9–35
PUSHAF (Push Address F_floating) instruction • 9–35
PUSHAG (Push Address G_floating) instruction • 9–35
PUSHAH (Push Address H_floating) instruction • 9–35
PUSHAL (Push Address Long) instruction • 9–35
PUSHAQ (Push Address Quad) instruction • 9–35
PUSHAW (Push Address Word) instruction • 9–35
PUSHL (Push Long) instruction • 9–27
PUSHR (Push Registers) instruction • 9–80

Q

.QUAD directive • 6–80
Quadword • 8–2
Quadword storage directive (.QUAD) • 6–80
Queue • 9–82 to 9–87
  absolute • 9–82 to 9–85
  header • 9–82, 9–85
  inserting entries • 9–82 to 9–85, 9–85 to 9–87
  removing entries • 9–84 to 9–85, 9–87
  self-relative • 9–85 to 9–87
Queue instructions • 9–82 to 9–100

R

Radix control operator • 3–11 to 3–12
Range
  syntax • 7–1
Read access type • 8–16
.REFn directive • 6–81
Register deferred mode • 5–5
  operand specifier format • 8–18
Register mask operator • 3–13 to 3–14, 6–29
Register mode • 5–4 to 5–5
  operand specifier format • 8–17 to 8–18
Register name • 3–5, 3–6
Register save mask • 6–29, 6–58
Register save mask directive (.MASK) • 6–58
REI (Return from Exception or Interrupt) instruction • 9–188
Relative deferred mode • 5–13
  setting default displacement length • 6–19
Relative mode • 5–12 to 5–13
  assembled as absolute mode • 6–22
  setting default displacement length • 6–19
Relocatable expression • 3–9
REMQHI (Remove Entry from Queue at Head, Interlocked) instruction • 9–95 to 9–96
REMQTI (Remove Entry from Queue at Tail, Interlocked) instruction • 9–97 to 9–98
REMQUE (Remove Entry from Queue) instruction • 9–99 to 9–100
Repeat block
  argument substitution • 6–46 to 6–47

Repeat block (cont'd.)
character substitution • 6–48 to 6–49
end • 6–28
listing range definitions of • 6–87
listing range expansions of • 6–87
listing specifiers • 6–87
terminating repetition • 6–61
Repeat block directive (.REPEAT) • 6–82 to 6–83
.REPEAT directive • 6–82 to 6–83
Repeat range end directive (.ENDR) • 6–28
Reserved operand • 9–102, 9–103, 9–142
.RESTORE_PSECT directive • 6–84
RET (Return from Procedure) instruction • 9–69 to 9–70
ROTL (Rotate Long) instruction • 9–28
Routine name
made available to debugger • 6–23
RSB (Return from Subroutine) instruction • 9–60

S

.SAVE_PSECT directive • 6–85 to 6–86
SBWC (Subtract with Carry) instruction • 9–29
SCANC (Scan Characters) instruction • 9–135
Section name
made available to debugger • 6–23
Self-relative queue • 9–85 to 9–87
Shift operator • 3–16
.SHOW directive • 6–87 to 6–88
.SIGNED_BYTE storage directive • 6–89
.SIGNED_WORD storage directive • 6–90 to 6–91
Significance indicator • 9–181
SKPC (Skip Character) instruction • 9–136
SOBGEQ (Subtract One and Branch Greater Than or Equal) instruction • 9–61
SOBGTR (Subtract One and Branch Greater Than) instruction • 9–62
Source statement
See Statement
SPANC (Span Characters) instruction • 9–137
Stack frame • 9–63
Statement • 1–1
character set • 3–1 to 3–2
comment • 2–3 to 2–4
continuation of • 2–1
format • 2–1 to 2–4
label • 2–2
operand • 2–3
operator • 2–3, C–7 to C–8
special characters • C–6 to C–7
String argument • 4–3 to 4–5
String data type
character • 8–7
leading separate numeric • 8–11 to 8–12
packed decimal • 8–12 to 8–13
trailing numeric • 8–7 to 8–11
String instructions • 9–124 to 9–137, 9–141 to 9–164
String operator
in macro • 4–8 to 4–11
SUBB2 (Subtract Byte 2 Operand) instruction • 9–30
SUBB3 (Subtract Byte 3 Operand) instruction • 9–30
Subconditional assembly block directive • 6–42 to 6–44
.IF_FALSE • 6–42 to 6–44
.IF_TRUE • 6–42 to 6–44
.IF_TRUE_FALSE • 6–42 to 6–44
Subconditional assembly block directive (.IF_x) • 6–42 to 6–44
SUBD2 (Subtract D_floating 2 Operand) instruction • 9–122
SUBD3 (Subtract D_floating 3 Operand) instruction • 9–122
SUBF2 (Subtract F_floating 2 Operand) instruction • 9–122
SUBF3 (Subtract F_floating 3 Operand) instruction • 9–122
SUBG2 (Subtract G_floating 2 Operand) instruction • 9–122
SUBG3 (Subtract G_floating 3 Operand) instruction • 9–122
SUBH2 (Subtract H_floating 2 Operand) instruction • 9–122
SUBH3 (Subtract H_floating 3 Operand) instruction • 9–122
SUBL2 (Subtract Long 2 Operand) instruction • 9–30
SUBL3 (Subtract Long 3 Operand) instruction • 9–30
SUBP4 (Subtract Packed 4 Operand) instruction • 9–163 to 9–164
SUBP6 (Subtract Packed 6 Operand) instruction • 9–163 to 9–164
.SUBTITLE directive • 6–92
Subtitle listing control directive (.SUBTITLE) • 6–92
SUBW2 (Subtract Word 2 Operand) instruction • 9–30
SUBW3 (Subtract Word 3 Operand) instruction • 9–30

Summary of OPCODES
alphabetic order • D–1
numeric order • D–10
SVPCTX (Save Process Context) instruction • 9–190
Symbol • 3–4 to 3–7
cross-referencing • 6–16 to 6–17, 6–65
determining value of • 3–6
external • 6–33, 6–98
global • 3–6, 6–33, 6–36, 6–94, 6–98
in operand field • 3–6
in operator field • 3–6
local • 3–6
macro name • 3–6
made available to debugger • 6–22
permanent • 3–4 to 3–5, 3–6
register name • 3–5, 3–6
suppressing • 6–23
transferral to VAX Symbolic Debugger • 6–18
undefined • 6–22
user-defined • 3–5 to 3–6, 3–6
Symbol attribute directive (.WEAK) • 6–98
Symbol definition for shareable image • 6–94 to 6–96
Symbol for shareable image directive (.TRANSFER) • 6–94 to 6–96
System failure • E–9

T

Tab stops
in source statement • 2–1
Term in MACRO statement • 3–9
Textual operator • 3–12 to 3–14
.TITLE directive • 6–93
Title listing control directive (.TITLE) • 6–93
Traceback • 6–23
Trace trap enable (T) • 8–14
Trailing numeric string
data type • 8–7 to 8–11
.TRANSFER directive • 6–94 to 6–96
Trap
arithmetic • E–1
arithmetic type code • E–1
change mode • E–7
decimal
string overflow • E–3
decimal overflow • 8–15
Trap (cont'd.)
divide-by-zero • 8–15
floating
divide-by-zero • E–2
overflow • E–2
underflow • E–2
integer
divide-by-zero • E–2
overflow • E–2
integer overflow • 8–14
subscript-range • E–3
trace • 8–14
TSTB (Test Byte) instruction • 9–31
TSTD (Test D_floating) instruction • 9–123
TSTF (Test F_floating) instruction • 9–123
TSTG (Test G_floating) instruction • 9–123
TSTH (Test H_floating) instruction • 9–123
TSTL (Test Long) instruction • 9–31
TSTW (Test Word) instruction • 9–31

U

Unary operator • 3–10 to 3–11
summary • C–7
UNDEFINED results • 7–1
UNPREDICTABLE results • 7–1
User-defined local label • 3–7 to 3–8
range • 3–7
User-defined symbol • 3–5 to 3–6, 3–6

V

Variable bit base address access type • 8–16
Variable-length bit field
bytes referenced • 8–6
data type • 8–5 to 8–6
Variable-length bit field instructions • 9–36 to 9–41
Virtual address • 8–1

W

.WARN directive • 6–97
.WEAK directive • 6–98
Word data type • 8–1
.WORD directive • 6–99

Word storage directive (.WORD) • 6–99
Write access type • 8–16

X

XFC (Extended Function Call) instruction • 9–81
XORB2 (Exclusive OR Byte 2 Operand) instruction • 9–32
XORB3 (Exclusive OR Byte 3 Operand) instruction • 9–32
XORL2 (Exclusive OR Long 2 Operand) instruction • 9–32
XORL3 (Exclusive OR Long 3 Operand) instruction • 9–32
XORW2 (Exclusive OR Word 2 Operand) instruction • 9–32
XORW3 (Exclusive OR Word 3 Operand) instruction • 9–32

Z

Zero condition code (Z) • 8–14

Reader's Comments

VAX MACRO and Instruction Set Reference Manual
AA–LA89A–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less ______________________________

What I like best about this manual is ______________________________

What I like least about this manual is ______________________________

I found the following errors in this manual:

Page Description

Additional comments or suggestions to improve this manual:

I am using **Version** ______ of the software this manual describes.

Name/Title ______________________________ Dept. ______________

Company ______________________________ Date ______________

Mailing Address ______________________________

______________________________ Phone ______________

-- Do Not Tear - Fold Here and Tape

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01-3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

-- Do Not Tear - Fold Here

Reader's Comments

VAX MACRO and Instruction Set Reference Manual
AA–LA89A–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less ______________________________

____

____

What I like best about this manual is ______________________________

____

____

What I like least about this manual is ______________________________

____

____

I found the following errors in this manual:

Page	Description
____	______________________________
____	______________________________
____	______________________________
____	______________________________
____	______________________________

Additional comments or suggestions to improve this manual:

____

____

____

____

I am using **Version** ______ of the software this manual describes.

Name/Title ______________________________ Dept. ____________

Company ______________________________ Date ____________

Mailing Address ______________________________

______________________________ Phone ____________

— Do Not Tear - Fold Here and Tape

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

— Do Not Tear - Fold Here