0603-352
Computer Organization

VAX Architecture & Assembly Reference

R · I · T
Department of Computer Science

1. Introduction

This document is intended to be a quick introduction to the architecture and assembly language of DEC's[1] VAX-11 computer systems. While it is in no way a complete substitute for the *VAX Architecture Reference Manual* or *VAX MACRO and Instruction Set Reference Manual*, it should give you enough background on the computer to help you with the work you will do in this course.[2]

The document is organized around the two fundamental aspects of this machine: the physical architecture itself, which determines the characteristics of data types and information representation; and the assembly language (VAX MACRO), which defines the interface between the user and the hardware.

2. Architecture

The VAX-11 (VAX) series of computers are 32-bit, variable-length instruction, two's complement machines. Memory addresses are 32 bits in length.

2.1. Register Structure

General Registers

There are 16 general registers, named `R0` through `R15`. Each is 32 bits in length. `R15` doubles as the program counter (`PC`); `R14` contains the runtime stack pointer (`SP`); `R13` is the stack frame pointer (`FP`); and `R12` is the argument list pointer (`AP`). All general registers can be modified by the executing program.

Processor Status Longword

The `PSL` is a 32-bit register. The low-order 16 bits form the *Processor Status Word* (`PSW`), and contain user-accessible information; the high-order 16 bits contain privilege information which is unavailable to the user. Of greatest interest to programmers are `PSW[3:0]`, which contain the Negative, Zero, oVerflow, and Carry (`NSVC`) bits, sometimes referred to as the *Condition Code* (`CC`).

[1]DEC and VAX are trademarks of Digital Equipment Corporation.

[2]This document is based on earlier documents written by Warren R. Carithers, Department of Computer Science at RIT (`wrc@cs.rit.edu`).

2.2. Hardware Data Types

The VAX supports the following hardware data types[3]:

Type	Size	Description
byte	8 bits	unsigned or signed integer value
word	16 bits	unsigned or signed integer value
longword	32 bits	unsigned or signed integer value
quadword	64 bits	unsigned or signed integer value
octaword	128 bits	unsigned or signed integer value
Character String	0-65535 bytes	one character per byte
Numeric String	0-31 bytes	digit-character representation
Queue	> 4 bytes per entry	up to $2*10^9$ entries.

Multi-byte items consist of bytes at consecutive addresses in memory. Bit positions are numbered from least significant (0) to most significant ($n-1$); within multi-byte items, bytes are stored from least-significant byte (LSB) to most-significant byte (MSB), with the LSB at the lowest address. The address of any data item is the address of the byte containing bit zero of that item.

There are also four floating point representations using varying sizes of exponent and mantissa, and a packed decimal representation which encodes digits into four-bit sequences, stored two per byte.

2.3. Instruction Format

Instructions in the VAX instruction set are variable length. All instructions have a one-byte opcode, followed by an operand specifier for each operand. Operand specifiers consist of a mode byte describing the addressing mode and general register being used, and zero or more bytes containing additional information. Each instruction has a fixed number of operands, although different instructions have from zero to six operands.

2.3.1. Addressing Modes

The VAX supports sixteen addressing modes. Each operand is represented in memory with an operand specifier, which consists of a mode byte followed by from zero to five additional bytes of information. The mode byte is broken into two fields: a four-bit mode specifier and a four-bit register designator.

In the following table, ''length'' refers to the length of the operand, as specified by the instruction, and $c(X)$ is the contents of X:

[3]We won't be using all of these, but the are included here to give you some idea of the variety of data types provided by the VAX. This is also true of the instruction, addressing mode, and assembler directive tables found later in this document.

Mode	Name	Syntax Examples	Effective Address
0-3	literal	`#`***lit***, `S^#`***lit***	none
4	indexed	***i***`[R`***n***`]`	***i*** `+ (c(R`***n***`) *` ***length***`)`
5	register	`R`***n***	none
6	register deferred	`(R`***n***`)`	`c(R`***n***`)`
7	autodecrement	`-(R`***n***`)`	decrement `R`***n*** by ***length***, then `c(R`***n***`)`
8	autoincrement	`(R`***n***`)+`	`c(R`***n***`)`, then increment `R`***n*** by ***length***
9	autoincrement deferred	`@(R`***n***`)+`	`c(c(R`***n***`))`, then increment `R`***n*** by 4
A	byte displacement	***D***`(R`***n***`)`, `B^`***D***`(R`***n***`)`	***D*** `+ c(R`***n***`)`
B	byte disp. deferred	`@`***D***`(R`***n***`)`, `@B^`***D***`(R`***n***`)`	`c(`***D*** `+ c(R`***n***`))`
C	word displacement	***D***`(R`***n***`)`, `W^`***D***`(R`***n***`)`	***D*** `+ c(R`***n***`)`
D	word disp. deferred	`@`***D***`(R`***n***`)`, `@W^`***D***`(R`***n***`)`	`c(`***D*** `+ c(R`***n***`))`
E	long displacement	***D***`(R`***n***`)`, `L^`***D***`(R`***n***`)`	***D*** `+ c(R`***n***`)`
F	long disp. deferred	`@`***D***`(R`***n***`)`, `@L^`***D***`(R`***n***`)`	`c(`***D*** `+ c(R`***n***`))`

Here is a description of each of these modes:

literal

Literals operands are ones whose values are found in the instruction rather than in a register or somewhere else in memory. Literals can be represented using modes 0, 1, 2, and 3, or with mode 8 (autoincrement, see below). Literals cannot be used as destination operands.

Literals expressed with the syntax `S^#`***nnn*** are represented using one of the modes 0, 1, 2, and 3; those expressed as `I^#`***nnn*** are represented using mode 8. For literals expressed as `#`***nnn***, the assembler examines the value (***nnn***) and selects the representation based on that value: integers in the range 0..63, inclusive, are represented using mode 0, 1, 2, or 3; integers outside that range, or addresses, are represented using mode 8.

For modes 0, 1, 2, and 3, the low-order two bits of the mode is combined with the four bits of the register field to yield an unsigned six-bit field, which contains the literal value (0..63). For mode 8, the literal (now called an *immediate* operand) is held in a series of bytes within the instruction sequence; see the discussion following the next table.

register

Register mode, `R`***n***, is used to indicate that the data to be manipulated will be found in the specified register, rather than in memory. Register mode cannot be used for any operand which must have a memory address (e.g., as a branch destination).

Register deferred mode, `(R`***n***`)`, is used to indicate that the specified register contains a memory address which is the actual (effective) address of the operand.

autoincrement/autodecrement

The autoincrement and autodecrement modes, `(R`***n***`)+` and `-(R`***n***`)`, combine the effect of register deferred mode with automatic modification of the specified register. For both types, the register is modified through the addition or subtraction of the length of the operand, in bytes; thus, for a longword instruction, the modification value is 4.

Autoincrement mode uses the initial contents of the specified register as the effective address, and then adds the length to the register. Autodecrement mode first subtracts the length from the register, then uses the resulting value as the effective address.

Autoincrement deferred mode, `@(R`***n***`)+`, takes the initial contents of the register as the address of an indirect longword in memory; the register is then incremented by 4, which is the length of the indirect longword. The contents of the indirect longword are retrieved from memory, and are used as the final effective address for the operand.

displacement

The displacement modes are used whenever the address of the operand is to be calculated at execution time by adding the contents of a specified register and a one-, two-, or four-byte offset value. The three basic displacement modes (byte, word, and longword) indicate the size of the offset field within the instruction. In all cases, the displacement is sign-extended to 32 bits (if necessary), and the sum of the extended displacement and the current contents of the specified register is computed; this sum is the effective address of the operand.

Displacement operands have the syntax ***d***`(R`***n***`)`, where ***d*** is the displacement. Normally, the assembler selects the displacement field size by examining the displacement itself: integers in the range -128..127 are represented using byte displacement; those outside this range, but within the range -32768..32767 are represented using word displacement; and other integers, and addresses, are represented using longword displacement. This can be overridden by the programmer through the use of the prefix operators `B^`, `W^`, and `L^` to force a specific displacement size to be used.

Displacement deferred operands, `@`***d***`(R`***n***`)`, compute the effective address the same way, by adding the sign-extended displacement and the contents of the specified register. This is taken as the address of an indirect longword; its contents are retrieved from memory, and are used as the final effective address of the operand.

All displacement modes occupy two to five bytes in the instruction sequence: the mode byte is immediately followed by the one-, two-, or four-byte displacement.

indexed

Indexed mode is a ''combination'' addressing mode; it must be used in conjuction with one of the modes described above, with the exception of literal mode. This other addressing mode is called the *base* mode.

Indexed mode is specified by following the base mode with an index specifier, `[R`***n***`)`, in the source code. In the machine code, the index mode byte *precedes* the base mode byte; e.g., the operand `(R4)+[R6]` is represented as a mode byte of `46` (indexed mode on `R6`) followed by the byte `84` (autoincrement mode on `R4`).

The base mode is evaluated as described above to form the initial effective address. Next, the contents of the index register are *scaled* by multiplying them by the operand length; this product is added to the initial effective address to form the final effective address of the operand.

The assembler also uses some operand modes in conjunction with the program counter (`R15`) to encode memory references. These modes are referred to as *program counter* modes in this case, with the following names:

Mode	Name	Syntax Examples	Effective Address
`8`	immediate	`#`***`lit`***, `I^#`***`lit`***	`c(R15)`, then increment `R15` by ***`length`***
`9`	absolute	`@#`***`addr`***	`c(c(R15))`, then increment `R15` by `4`
`A`	byte relative	***`addr`***, `B^`***`addr`***	`c(R15) +` calculated displacement
`B`	byte rel. deferred	`@`***`addr`***, `@B^`***`addr`***	`c(c(R15) +` calculated displacement)
`C`	word relative	***`addr`***, `W^`***`addr`***	`c(R15) +` calculated displacement
`D`	word rel. deferred	`@`***`addr`***, `@W^`***`addr`***	`c(c(R15) +` calculated displacement)
`E`	long relative	***`addr`***, `L^`***`addr`***	`c(R15) +` calculated displacement
`F`	long rel. deferred	`@`***`addr`***, `@L^`***`addr`***	`c(c(R15) +` calculated displacement)

For memory operands whose addresses are known to the assembler (i.e., they are within the module being assembled), the assembler uses displacement modes; they are known as *relative* or *PC-relative* addresses. The displacement is calculated by subtracting the address of the first byte *following* the *displacement field itself* in the instruction from the address of the operand, yielding a signed absolute distance to the operand. The size of the field is chosen according to the displacement's value: those in the range -128..127 use byte displacement; those in the range -32768..32767 use word displacement; and all others use longword displacement. Again, the `B^/W^/L^` prefixes are optional.

For memory operands whose addresses aren't known (e.g., operands defined outside the module being assembled), in situations where the calculated displacement is too large to fit within a 32-bit signed longword, or when the `@#` prefix is explicitly used on the operand, the assembler uses *absolute* addressing - the operand address is placed directly in the instruction sequence, and mode 9 (autoincrement deferred) addressing is used from *R15*. These encodings always use a longword to hold the operand address; for external operands, the longword left by the assembler is filled in by the operating system when the program is loaded into memory for execution.

Literal operands whose values are not absolute or out of the inclusive range 0 to 63, or literals which are explicitly specified with the `I^` prefix, are encoded using mode 8 (autoincrement) and *R15*. Again, a longword extension is always used to hold the literal.

2.4. The Instruction Set

The following notation will be used to define the instruction set. For clarity (and for historical reasons), instruction mnemonics are given in upper case, although the assembler ignores case on input. Operands are described using the following format:

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

where ***`<name>`*** is a suggestive name (e.g., `src`, `dest`) for the operand, ***`<access type>`*** is a letter which describes how the operand will be used (according to the table below), and ***`<data type>`*** is a letter describing the size of the data item being used (again, according to the table below.) Implied operands (locations accessed by the instruction but not specified in an operand) are enclosed in brackets.

Type	Meaning
a	Calculate the effective address of the specified operand. Address is returned in a longword which is the actual instruction operand. Context of address calculation is given by ***<data type>*** (i.e., size to be used in autoincrement, autodecrement, and indexing.)
b	No operand reference. Operand specifier is a branch displacement. Size of branch displacement is given by ***<data type>***.
m	Operand is read, potentially modified, and written. Note that if the operand is not actually modified, it may not be written back; however, it will always be checked for both read and write accessibility.
r	Operand is read only.
v	Calculate the EA of the specified operand. If the EA is in memory, the address is returned in a longword which is the actual instruction operand. Context of address calculation is given by ***<data type>***. If the EA is `Rn`, the operand is in `Rn` or `R[n+1]'Rn`.
w	Operand is written only.

The data types `b`, `w`, and `l` refer to byte, word, and data sizes; `x` refers to the first data type specified by the mnemonic; and `y` refers to the second data type specified by the mnemonic.

2.4.1. Arithmetic, Logical, and Data Movement

Mnemonic	Operand(s)	Type(s)	Opcode(s)	Name
ADDx2	add.rx,sum.mx	B/W/L	80/A0/C0	add 2 operand
ADDx3	add1.rx,add2.rx,sum.wx	B/W/L	81/A1/C1	add 3 operand
ASHL	count.rb,src.rl,dst.wl	n.a.	78	arithmetic shift
BICx2	mask.rx,dst.mx	B/W/L	8A/AA/CA	bit clear 2 operand
BICx3	mask.rx,src.rx,dst.wx	B/W/L	8B/AB/CB	bit clear 3 operand
BICPSW	mask.rw	n.a.	B9	bit clear in PSW
BISx2	mask.rx,dst.mx	B/W/L	88/A8/C8	bit set 2 operand
BISx3	mask.rx,src.rx,dst.wx	B/W/L	89/A9/C9	bit set 3 operand
BISPSW	mask.rw	n.a.	B8	bit set in PSW
BITx	mask.rx,src.rx	B/W/L	93/B3/D3	bit test
CLRx	dst.wx	B/W/L	94/B4/D4	clear
CMPx	src1.rx,src2.rx	B/W/L	91/B1/D1	compare
CVTBy	src.rb,dst.wy	W/L	99/98	convert byte to y
CVTWy	src.rw,dst.wy	B/L	33/32	convert word to y
CVTLy	src.rl,dst.wy	B/W	F6/F7	convert long to y
DECx	dst.mx	B/W/L	97/B7/D7	decrement
DIVx2	divr.rx,quo.mx	B/W/L	86/A6/C6	divide 2 operand
DIVx3	divr.rx,divd.rx,quo.wx	B/W/L	87/A7/C7	divide 3 operand
INCx	dst.mx	B/W/L	96/B6/D6	increment

Mnemonic	Operand(s)	Type(s)	Opcode(s)	Name
MOVAx	src.ax,dst.wl	B/W/L	9E/3E/DE	move address
MOVx	src.rx,dst.wx	B/W/L	90/B0/D0	move
MULx2	mulr.rx,prod.mx	B/W/L	84/A4/C4	multiply 2 operand
MULx3	mulr.rx,muld.rxprod.wx	B/W/L	85/A5/C5	multiply 3 operand
PUSHAx	src.ax	B/W/L	9F/3F/DF	push address onto stack
PUSHL	src.rl	n.a.	DD	push longword onto stack
PUSHR	mask.rw	n.a.	BB	push register(s) onto stack
POPR	mask.rw	n.a.	BA	pop register(s) from stack
ROTL	count.rb,src.rl,dst.rl	n.a.	9C	rotate longword
SUBx2	sub.rx,dif.mx	B/W/L	82/A2/C2	subtract 2 operand
SUBx3	sub.rx,min.rx,dif.wx	B/W/L	83/A3/C3	subtract 3 operand
TSTx	src.rx	B/W/L	95/B5/D5	test

2.4.2. Control

Mnemonic	Operand(s)	Type(s)	Opcode(s)	Name
AOBLEQ	limit.rl,index.ml,displ.bb	n.a.	F3	add 1, branch if <= limit
AOBLSS	limit.rl,index.ml,displ.bb	n.a.	F2	add 1, branch if < limit
SOBGEQ	index.ml,displ.bb	n.a.	F4	sub 1, branch if >= 0
SOBGTR	index.ml,displ.bb	n.a.	F5	sub 1, branch if > 0
JMP	dst.ab	n.a.	17	jump
JSB	dst.ab	n.a.	16	jump to subroutine
BSBx	displ.bx	B/W	10/30	branch to subroutine
RSB		n.a.	05	return from subroutine
CALLG	arglst.ab,dst.ab	n.a.	FA	call procedure
CALLS	arglst.ab,dst.ab	n.a.	FB	call procedure
RET		n.a.	04	return from procedure
Bx	displ.bb	NEQ/EQL	12/13	branch if <>, =
		GTR/LEQ	14/15	branch if >, <=
		GEQ/LSS	18/19	branch if >=, <
BRx	displ.bx	B/W	11/31	branch
BCx	displ.bb	C/S	1E/1F	branch if carry clear/set
BVx	displ.bb	C/S	1C/1D	branch if overflow clear/set

2.4.3. Character String

Mnemonic	Operand(s)	Type(s)	Opcode(s)	Name
CMPC3	len.rw,s1a.ab,s2a.ab	n.a.	29	compare chars. 3 operand
CMPC5	s1len.rw,s1a.ab,fill.rb, s2len.rw,s2a.ab	n.a.	2D	compare chars. 5 operand

Mnemonic	Operand(s)	Type(s)	Opcode(s)	Name
`MOVC3`	`len.rw,src.ab,dst.ab`	`n.a.`	`28`	move chars. 3 operand
`MOVC5`	`slen.rw,src.ab,fill.rb,` `dlen.rw,dst.ab`	`n.a.`	`2C`	move chars. 5 operand

2.4.4. Other

Mnemonic	Operand(s)	Type(s)	Opcode(s)	Name
`CHMx`	`code.rw`	`K/E/S/U`	`BC/BD/BE/BF`	change mode

3. Assembly Language

The assembly language for the VAX is known by several different names (VAX Assembly Language, VAX MACRO, etc.). In this document, we will refer to it as VAX MACRO.

Assembly language programs consist of sequences of lines of VAX MACRO code; each is interpreted individually by the assembler. Source lines come in two varieties: *statements*, which are translated by the assembler into the machine instructions are source lines which will be translated by the assembler into *instructions* which are to be executed by the hardware; and *assembler directives*, which are commands to the assembler which affect the way it performs the assembly (e.g., by reserving blocks of memory, etc.)

3.1. VAX MACRO Statement Format

A VAX MACRO source statement contains up to four fields: label, operator, operand, and comment. All fields are optional; each has the purpose described below. Fields are separated by one or more whitespace (blank and/or tab) characters.

3.1.1. Label Field

Labels in VAX MACRO can be of any length, but only the first 31 characters are significant. They are composed of alphanumeric characters and the special characters dot (`.`), dollar sign ($), and underscore (_). The first character must not be numeric. A label is terminated with either a single colon (`:`), which defines the label as a local symbol, or a doubled colon (`::`), which defines the label as a global symbol. Once a label is defined, it cannot be redefined in the same source program; if it is redefined, the assembler displays an error message at each definition and each reference. The value of a symbol defined as a label is the *location counter* (LC) value when the definition occurs.

3.1.2. Operator Field

This field can contain an *instruction mnemonic* (something which will be translated into a machine instruction) or an *assembler directive* (a request of or command to the assembler itself). The operator field is terminated by the first whitespace character following the operator name.

3.1.3. Operand Field

The format of the operand field is determined by the contents of the operator field. This field contains operands for instructions or arguments for assembler directives. The operand field is terminated by the end of the line or the semicolon which starts a comment field. Within the operand field, individual operands are separated by commas.

3.1.4. Comment Field

This field begins with a semicolon character and continues through the end of the line. Any printable ASCII character may occur in a comment. If a comment is continued on subsequent lines, each continuation must begin with a semicolon. A comment may appear on a line by itself.

3.2. VAX MACRO Source Statement Components

3.2.1. Character Set

The following characters are valid in VAX MACRO source statements:

- Alphabetic characters (`A` through `Z` and `a` through `z`). Except within ASCII character strings, case is ignored.
- Digit characters (''0'' through ''9'').
- Special characters, as shown in the table below.

Character(s)	Name(s)	Function
`_`	underline	character in symbol names
`$`	dollar sign	character in symbol names
`.`	period	character in symbol names
`:`	colon	label terminator
`;`	semicolon	comment field indicator
`=`	equal sign	direct assignment
	space, tab	field separator/terminator
`,`	comma	field, operand, and item separator
`#`	number sign	immediate indicator
`@`	at sign	deferred mode indicator
`+`	plus sign	autoincrement mode; unary plus & arithmetic addition
`-`	minus sign	autodecrement mode; unary minus & arithmetic subtraction
`*`	asterisk	arithmetic multiplication
`/`	slash	arithmetic integer division
`^`	circumflex	unary operators
`[ ]`	brackets	index mode, repeat count
`( )`	parentheses	register deferred mode
`< >`	angle brackets	argument/expression grouping

3.2.2. Numbers

Integers can be used in any expression, including expressions in operands and direct assignment statements. An integer consists of an optional sign followed by a series of digits which are legal in the current radix. By default, the radix is decimal; a *radix control operator* may be used to alter the default radix, as described below.

3.2.3. Symbols

3.2.3.1. Permanent Symbols

Permanent symbols are predefined within the assembler and cannot be redefined by the user. These consist of instruction mnemonics, VAX MACRO directives, and register names. Mnemonics and directives are described elsewhere. Register names have the form `R`***n***, where ***n*** is an integer between 0 and 15, inclusive, or are the symbols `AP`, `FP`, `SP`, or `PC` (which are alternate names for registers `R12`, `R13`, `R14`, and `R15`, respectively).

3.2.3.2. User-Defined Symbols

User-defined symbols are 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. Symbols may be of any length, but must be unique within the first 31 characters.

3.2.4. Terms and Expressions

A *term* can be any of the following: a number; a symbol; the LC; a text operator followed by text; or any of these preceded by a unary operator. Terms are always evaluated as longword values.

The LC term (`.`) takes on the value of the LC at the start of the current operand - thus, for assembly of separate operands within a single statement, the LC may have different values.

Expressions are combinations of terms joined by binary operators and evaluated as longword values. All operators have equal precedence; expressions are evaluated from left to right. However, angle brackets may be used to alter the order of evaluation. Any part of an expression which is enclosed in angle brackets is evaluated first. Angle brackets can be used to apply a unary operator to an entire expression. Unary operators are evaluated before binary operators. Expressions fall into these categories:

- *Relocatable*. The value of the expression is fixed relative to the start of the program - i.e., it represents an address within the program's address space. The LC is relocatable.
- *Absolute*. The value of the expression is an assembly-time constant. An expression whose terms are all absolute is absolute. An expression consisting of a relocatable term minus another relocatable term is absolute.
- *External*. The expression contains one or more symbols that are not defined in the current module, but rather are being imported from another module.

The *mode* (category) of an expression depends on its component terms:

- Any expression containing an external term or subexpression is itself external.
- Absolute and relocatable terms (or subexpressions) can be combined as follows, where A, R, and I are absolute, relocatable, and invalid (error), and x is any valid mode:

Expression	Resulting Mode	Expression	Resulting Mode
A `+-*/` *A*	absolute	*R* `*/` *A*	invalid
R `-` *R*	absolute	*R* `+*/` *R*	invalid
R `+-` *A*	relocatable	*I* `+-*/` *x*	invalid
A `+` *R*	relocatable	*x* `+-*/` *I*	invalid

Any type of expression can be used in most VAX MACRO statements; direct assignment statements and some assembler directives, however, place restrictions on the types of expressions which may be used.

3.2.5. Unary Operators

A unary operator modifies a term or an expression. Expressions modified by unary operators must be enclosed in angle brackets. The following operators are defined:

Operator	Name	Example	Description
`+`	plus sign	`+A`	positive value of `A`
`-`	minus sign	`-A`	negative (two's complement) value of `A`
`^B`	binary	`^B01010110`	a binary representation of the number 86_{10}
`^D`	decimal	`^D86`	a decimal representation of the number 86_{10}
`^O`	octal	`^O127`	an octal representation of the number 86_{10}
`^X`	hex	`^X76`	a hex representation of the number 86_{10}
`^A`	ASCII	`^A/pdq/`	ASCII string containing the characters `pdq`
`^C`	complement	`^C^X24`	`11011011` (one's complement of 24_{16})
`^M`	register mask	`^M<R8,R9>`	16-bit mask for specified register(s)

More than one unary operator can be applied to a single term or expression; thus, `-+-A` is equivalent to `-<+<-A>>`.

3.2.5.1. Radix Control Operators

VAX MACRO accepts terms in four radixes: binary, decimal (default), octal, and hex. The unary prefix operators `^B` (binary), `^D` (decimal), `^O` (octal), and `^X` (hex) are called *radix control operators*. Within a number prefaced by a radix control operator, only digit characters valid for that radix are acceptable. If a radix control operator is used, no sign may be applied to a number. Expressions involving more than one term which are modified by radix control operators must be enclosed in angle brackets; the radix control operator applies to all terms in the expression unless overridden by another operator - thus, in the expression `^X<F1C3+FFF2+^D20>`, the first and second terms are assumed to be hex, while the `^D` on the final term causes the assembler to view it as decimal.

3.2.5.2. Textual Operators

3.2.5.2.1. ASCII Operator

The `^A` (ASCII) operator is used to convert a string of printable characters to their 8-bit representations. The string of characters must be enclosed by a pair of matching delimiters. The delimiters can be any printable ASCII character except the space, the tab, or the semicolon. The string may never be longer than 16 characters; within an instruction, the string may not be longer than the size of the data type being used.

3.2.5.2.2. Register Mask Operator

A *register mask* is specified with in the form `^M<`***`x`***`>`, where ***`x`*** is a comma-separated list of register names. A register mask defines a 16-bit quantity where one bit is set for each register named in the mask. Register names `R0` through `R12` set bits 0 through 12, respectively; `AP`, `FP`, `SP`, and `PC` set bits 12, 13, 14, and 15, respectively. The symbol `IV` also sets bit 14; this is used to specify the setting of an *integer overflow* trap bit, which allows detection of overflow on integer arithmetic operations.

Some register names cannot be used with some statements: `PUSHR` and `POPR` do not allow use of the `IV` or `PC` names (bit 14 is used to specify register 14, and the `PC` register cannot be saved this way); the `.ENTRY` directive does not allow the use of `R0`, `R1`, `FP`, `SP`, or `PC` names (see below).

3.2.6. Binary Operators

A binary operator is used to perform an operation on a pair of operands. Expressions must be enclosed in angle brackets. The arithmetic operators `+`, `-`, `*`, and `/` are valid; each performs its standard arithmetic operation on longword values. All operators have equal precedence.

3.2.7. Direct Assignment Statement

A direct assignment statement equates a symbol to a specific value. Unlike a label, a symbol defined via a direct assignment statement can be redefined as many times as desired. The direct assignment statement has the form

```
    name = value
```

where ***`name`*** is the symbol and ***`value`*** is an expression which does not contain any symbols not defined at that point in the assembly. A direct assignment statement may have a comment field.

3.2.8. Location Counter

The LC symbol (`.`) always has the value of the address of the current byte. VAX MACRO sets the LC to zero at the beginning of the assembly. Every VAX MACRO statement which allocates memory in the resulting assembled object module increments the LC by the number of bytes allocated.

The LC value can be set by the direct assignment statement. The LC must be set equal to a relocatable expression; this may be an expression constructed with the LC value (e.g., `. = . + 10`).

Note that when the LC is used in an expression in the operand field of a statement, its value is the base address of that operand within the generated instruction, *not* the address of opcode byte at the beginning of the instruction.

3.3. Statements

Statements are lines of source code which are assembled into machine instructions. They are specified to the assembler through the use of an *instruction mnemonics* in the operator field of the source line. (Instruction mnemonics are listed in the earlier section entitled *The Instruction Set*.) The number and types of operands on statements must agree with those expected for that instruction, or the assembler will report an error.

3.4. Assembler Directives

Assembler directives (also called *pseudo-ops* or *pseudo-instructions*) are instructions to the assembler about how to perform the translation, and are not instructions to the machine.

As with statements, assembler directives are specified through the use of a mnemonic symbol in the operator field of the source line. Commonly, directive names are symbols which begin with a dot (`.`) character, which distinguishes them visually from instruction mnemonics.

3.4.1. Listing and Module Control

`.TITLE` ***`module-name   comment-string`***

Assigns a name to the object module. ***`Module-name`*** is the first six or fewer non-whitespace characters following the directive. If `.TITLE` is not used in a module, VAX MACRO assigns the default name `.MAIN` to the module. If multiple `.TITLE` directives are specified, the last directive establishes the module name. ***`Comment-string`*** is an optional description of this module; only the first 40 characters are retained by the assembler.

`.ENTRY` ***`name,expression`***

Defines an *entry point*, into the module; ***`name`*** is the symbolic name of the entry point, and is automatically marked as an exported symbol. Two bytes of memory are allocated, labelled by ***`name`***, and filled with ***`expression`***, which is a register mask specifier. While ***`name`*** labels the word containing ***`value`***, entry to the module will actually occur at the location ***`name`***`+2`, which is the address of the first instruction at this entry point.

`.ENTRY` is intended to be used with the `CALLS`, `CALLG`, and `RET` procedure-call instructions; it should ***not*** be used for entry points which are invoked with `JSB`, `BSBx`, or other transfer instructions, which do not use a register mask.

`.END` ***`[symbol]`***

Terminates the source program. Any text following the `.END` in the source file is ignored. The optional argument, ***`symbol`***, is taken as the entry point for the main program, and should be defined with the `.ENTRY` directive. If multiple object modules are linked to form a single load module, exactly one object module should specify a starting address.

3.4.2. Data Storage

`.ASCIx` ***`string`***

These cause ASCII characters to be placed into bytes of memory, one per byte. Each directive is followed by a string of characters enclosed in a pair of matching delimiters. Delimiters can be any printable ASCII characters except space, tab, equal sign, semicolon, left angle bracket, or a character within the delimited string. The assembler preserves the case of characters in the string. Any character except `NUL` (ASCII code 0), `CR` (carriage return, code 13), and `LF` (line feed or newline, code 10) can be in the delimited string.

Any ASCII character (including those restricted above) can also be generated by enclosing an assembly-time expression whose value is the character's ASCII code in angle brackets outside the delimiters.

Four directives are available. `.ASCII` merely causes the string to be placed into memory. `.ASCIC` generates a prefix byte containing the count of characters generated; the string may be up to 255 characters in length. `.ASCID` generates a *string descriptor* in front of the string, consisting of two longwords: the first contains a 16-bit string length (in the low-order half); the second, a pointer to the first byte of the string. `.ASCIZ` generates a string with a trailing byte containing the ASCII `NUL` character.

`.BYTE` ***expression-list***

Generates a series of bytes of memory which contains values from the ***expression-list***. Within the ***expression-list***, entries are separated by commas. One byte is generated per simple ***expression*** in the list. Each ***expression*** is evaluated as a longword, then truncated to a byte; an ***expression*** whose high-order three bytes are other than `0` or `^XFFFF` causes the assembler to produce an error message. A relocatable ***expression*** is not allowed.

An ***expression*** can be followed by a *repetition factor* which specifies the number of times the ***expression*** is to be repeated; ***exp1*** `[` ***exp2*** `]` specifies that ***exp2*** bytes are to be generated, each of which will contain ***exp1***.

`.ADDRESS` ***address-list***

Generates a series of longwords containing addresses in the object module. One longword is generated for each entry in ***address-list***. Repetition factors are not allowed.

`.WORD` ***expression-list***

Like `.BYTE`, but generates words rather than bytes, and an assembler error message is produced if the high-order two bytes of the longword result are other than `0` or `^XFFFF`.

`.LONG` ***expression-list***

Like `.BYTE`, but generates longwords rather than bytes.

3.4.3. Location Control

`.ALIGN` ***how[,expression]***

How is either an integer in the range 0 to 9, or one of the symbols `BYTE`, `WORD`, `LONG`, `QUAD`, or `PAGE`. If an integer, N, the LC is forced to an address that is a multiple of 2^N; the symbols may be used to specify particular values, as follows: `BYTE`, 0; `WORD`, 1; `LONG`, 2; `QUAD`, 4; `PAGE`, 9. The optional ***expression*** parameter specifies the fill value to be placed into each skipped byte; the default is zero. ***Expression*** must be absolute, and must not contain any symbols not defined at that point in the assembly. If the LC is already at the specified alignment, no adjustment is performed.

While alignment of data items is never required by VAX hardware, due to the design of the memory unit, aligned data items can be retrieved faster from memory; therefore, alignment of data items can improve execution speed.

`.EVEN`
: Like `.ALIGN`, but forces the LC to an even value. If the LC is already even, no adjustment is performed.

`.ODD`
: Like `.ALIGN`, but forces the LC to an odd value. If the LC is already odd, no adjustment is performed.

`.BLKx` ***`expression`***
: Allocates block storage space for ***`expression`*** things of size ***`x`***. ***`X`*** can be `A` (address), `B` (byte), `L` (longword), `O` (octaword), `Q` (quadword), or `W` (word). ***`Expression`*** must be absolute, and must not contain any symbols not defined at that point in the assembly.

3.4.4. Symbol Control

These directives control accessibility of symbols between modules. Symbols not defined within the source module and not declared external are considered undefined.

`.GLOBAL` ***`symbol-list`***
: Defines the symbols in ***`symbol-list`*** as global symbols. If defined in the current module, a symbol is available externally; otherwise, the symbol is declared to be an externally defined symbol.

`.EXTERNAL` ***`symbol-list`***
: Declares the symbols in ***`symbol-list`*** as externally-defined symbols. Similar to `.GLOBAL`, but cannot be used to export symbols defined in the source module being assembled.

3.4.5. Instruction Generation

`.OPDEF` ***`mnemonic  value,descriptor-list`***
: Defines an operation which is inserted into a user-defined opcode table. This table is searched by the assembler before the permanent symbol table. ***`Mnemonic`*** is an identifier which specifies the name of the new operation; it should not be delimited. ***`Value`*** is an absolute expression which specifies the opcode value for this operation. No symbol names which have not been defined by this point in the assembly may be used in the ***`value`***. ***`Value`*** must be in the range 0 through 65535, but values 252 through 255 cannot be used. If ***`value`*** is less than 252, it represents a one-byte opcode; otherwise, bits `7:0` of ***`value`*** form the first byte of the opcode, and bits `15:8` form the second byte of the opcode. Each operand for the new operation is represented by a descriptor in the ***`descriptor-list`***. A descriptor is of the form ***`ad`***, where ***`a`*** is `A` (address), `R` (read only), `M` (modify), `W` (write only), `V` (field), or `B` (branch), and ***`d`*** is `B` (byte), `W` (word), `L` (longword), `Q` (quadword), or `O` (octaword). The combinations `BL`, `BQ`, and `BO` are illegal. The descriptors are used to control the assembler's handling of operand expressions. Redefinition of an existing mnemonic is allowed.