8 Basic Architecture

The following sections describe the basic VAX architecture, including the following:

- Address space
- Data types
- Processor status longword (PSL)
- Permanent exception enables
- Instruction and addressing mode formats

8.1 VAX Addressing

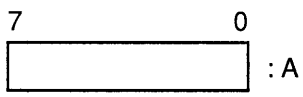
The basic addressable unit in VAX MACRO is the 8-bit byte. Virtual addresses are 32 bits long. Therefore, the virtual address space is 2^{32} (approximately 4.3 billion) bytes. Virtual addresses as seen by the program are translated into physical memory addresses by the memory management mechanism.

8.2 Data Types

The following sections describe the VAX data types.

8.2.1 Byte

A byte is 8 contiguous bits starting on an addressable byte boundary. The bits are numbered from right to left 0 to 7.

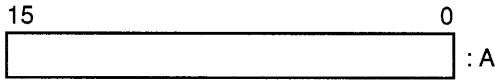


ZK–1119A–GE

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 to bit 6, with bit 7 the sign bit. The value of the integer is in the range –128 to +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 to bit 7. The value of the unsigned integer is in the range 0 to 255.

8.2.2 Word

A word is 2 contiguous bytes starting on an arbitrary byte boundary. The 16 bits are numbered from right to left 0 to 15.

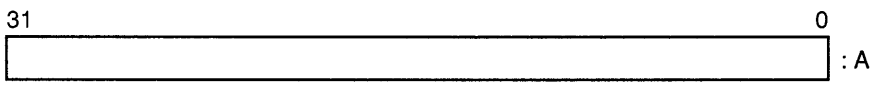


ZK-1120A-GE

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 to bit 14, with bit 15 the sign bit. The value of the integer is in the range –32,768 to +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 to bit 15. The value of the unsigned integer is in the range 0 to 65,535.

8.2.3 Longword

A longword is 4 contiguous bytes starting on an arbitrary byte boundary. The 32 bits are numbered from right to left 0 to 31.

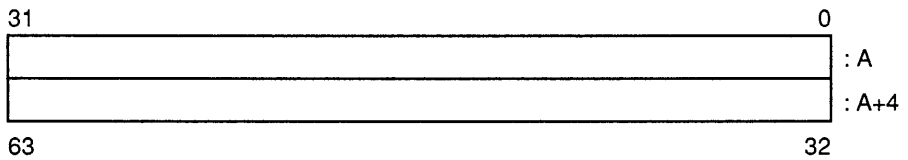


ZK-1121A-GE

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 to bit 30, with bit 31 the sign bit. The value of the integer is in the range –2,147,483,648 to +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 to bit 31. The value of the unsigned integer is in the range 0 to 4,294,967,295.

8.2.4 Quadword

A quadword is 8 contiguous bytes starting on an arbitrary byte boundary. The 64 bits are numbered from right to left 0 to 63.

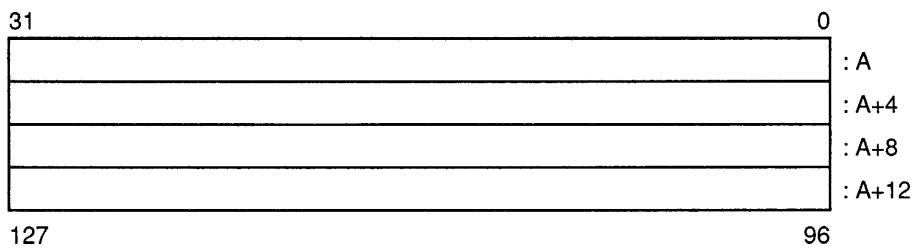


ZK-1122A-GE

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 to 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 to 127.

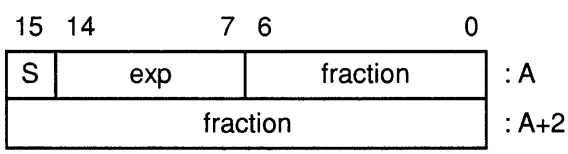


ZK-1123A-GE

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 to 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 4 contiguous bytes starting on an arbitrary byte boundary. The 32 bits are labeled from right to left 0 to 31.

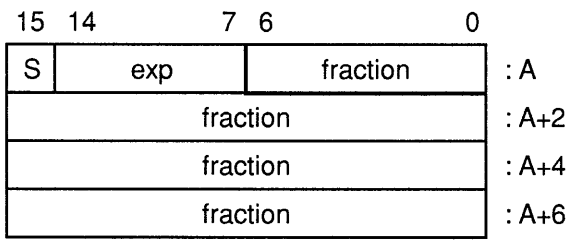


ZK-1124A-GE

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 to 31 and 0 to 6. The 8-bit exponent field encodes the values 0 to 255. An exponent value of zero, together with a sign bit of zero, is taken to indicate that the F_floating datum has a value of zero. Exponent values of 1 to 255 indicate true binary exponents of –127 to +127. An exponent value of zero, 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 to 1.7*10**38. The precision of an F_floating datum is approximately one part in 2**23; that is, typically 7 decimal digits.

8.2.7 D_floating

A D_floating datum is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from right to left 0 to 63.

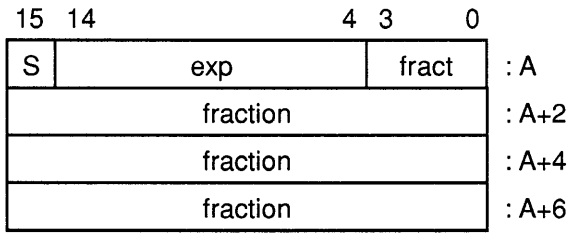


ZK–1125A–GE

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 to 63, 32 to 47, 16 to 31, and 0 to 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 to 63.

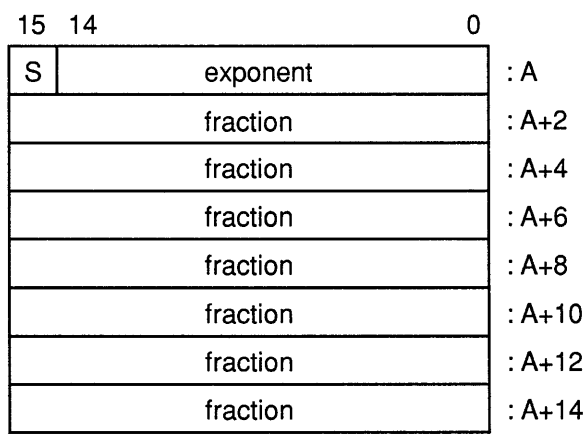


ZK-1126A-GE

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 to 63, 32 to 47, 16 to 31, and 0 to 3. The 11-bit exponent field encodes the values 0 to 2047. An exponent value of zero, together with a sign bit of zero, is taken to indicate that the G_floating datum has a value of zero. Exponent values of 1 to 2047 indicate true binary exponents of –1023 to +1023. An exponent value of zero, 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 to .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 to 127.



ZK-1127A-GE

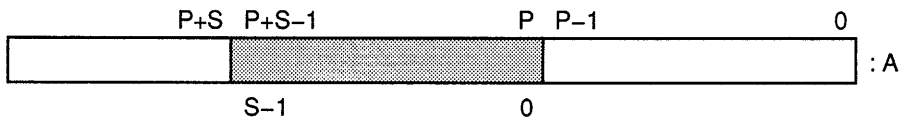
An H_floating datum is specified by its address, A, which is the address of the byte containing bit 0. The form of an H_floating datum is sign magnitude with bit 15 as the sign bit, bits 14:0 as an excess 16,384 binary exponent, and bits 127:16 as a normalized 113-bit fraction with the redundant most-significant fraction bit not represented. Within the fraction, bits of increasing significance range from bits 112 to 127, 96 to 111, 80 to 95, 64 to 79, 48 to 63, 32 to 47, and 16 to 31. The 15-bit exponent field encodes the values 0 to 32,767. An exponent value of zero, together with a sign bit of 0, is taken to indicate that the H_floating datum has a value of zero. Exponent values of 1 to 32,767 indicate true binary exponents of −16,383 to +16,383. An exponent value of zero, 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 to .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:

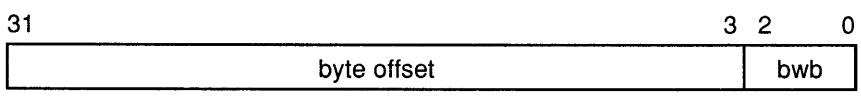
- Address A of a byte
- Bit position P, which is the starting location of the field with respect to bit 0 of the byte at A
- Size S of the field

The specification of a bit field is indicated by the following figure, where the field is the shaded area.



ZK−1128A−GE

For bit strings in memory, the position is in the range −2**31 to 2**31−1 and is conveniently viewed as a signed 29-bit byte offset and a 3-bit, bit-within-byte field.



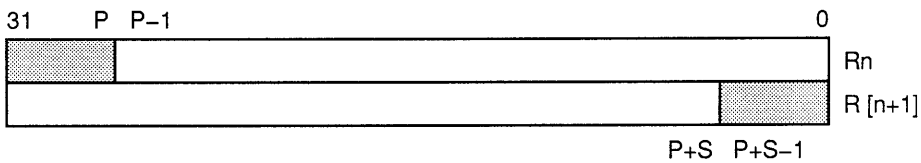
ZK−1129A−GE

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 to 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 to 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 zero has a value identically equal to zero.

A variable-length bit field may be contained in 1 to 5 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 to 31. The position operand specifies the starting position (0 to 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.

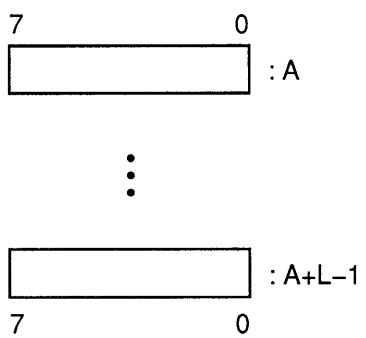


ZK–1130A–GE

For further details on the specification of variable-length bit fields, see the descriptions of the variable-length bit field instructions in Section 9.2.3.

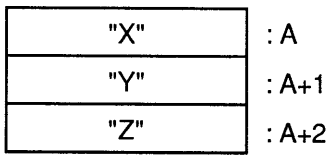
8.2.11 Character String

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:



ZK–1131A–GE

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



ZK-1132A-GE

The length L of a string is in the range 0 to 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 to 9).

The representation for the high-order digits is as follows:

Digit	Decimal	Hex	ASCII Character
0	48	30	0
1	49	31	1
2	50	32	2
3	51	33	3
4	52	34	4
5	53	35	5
6	54	36	6
7	55	37	7
8	56	38	8
9	57	39	9

The highest-addressed byte of a trailing numeric string represents an encoding of both the least-significant digit and the sign of the numeric string. The VAX numeric string instructions support any encoding; however, Digital software uses three encodings. These are as follows:

- 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

(continued on next page)

Table 8–1 (Cont.) Representation of Least-Significant Digit and Sign in Zoned Numeric Format

Digit	Decimal	Hex	ASCII Character
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	

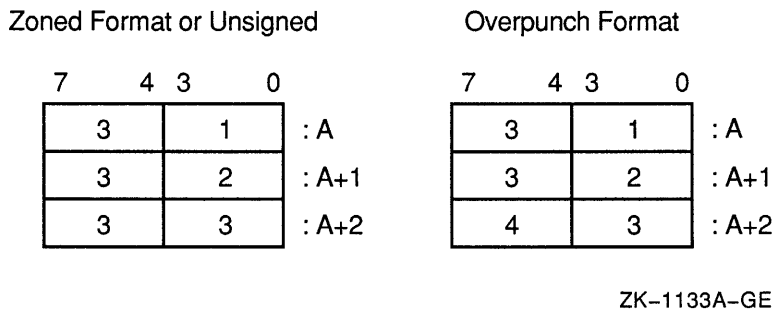
(continued on next page)

Table 8–2 (Cont.) Representation of Least-Significant Digit and Sign in Overpunch Format

			ASCII Character	
Digit	**Decimal**	**Hex**	**Norm**	**Alt.**
-5	78	4E	N	
-6	79	4F	O	
-7	80	50	P	
-8	81	51	Q	
-9	82	52	R	

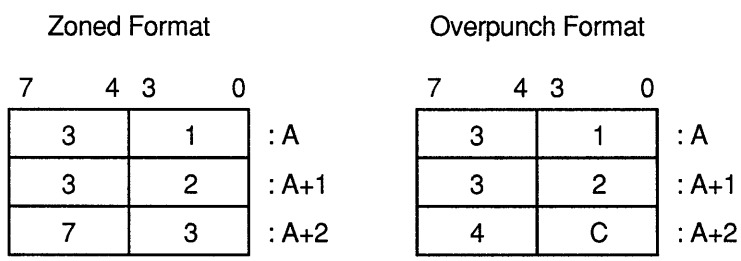
The length L of a trailing numeric string must be in the range 0 to 31 (0 to 31 digits). The value of a zero-length string is zero.

The address A of the string specifies the byte of the string containing the most-significant digit. Digits of decreasing significance are assigned to increasing addresses. Thus "123" is represented as follows:



ZK–1133A–GE

The trailing numeric string with a value of "–123" is represented as follows:



ZK–1134A–GE

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:

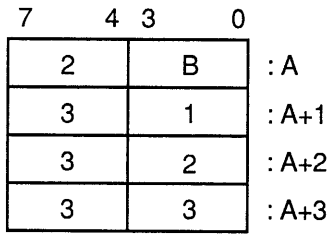
Sign	Decimal	Hex	ASCII character
+	43	2B	+
+	32	20	{blank}
–	45	2D	–

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

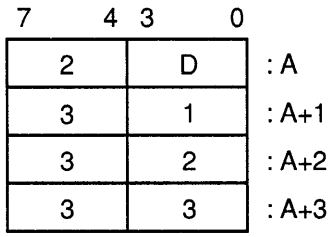
The length L of a leading separate numeric string must be in the range 0 to 31 (0 to 31 digits). The value of a zero-length string is zero.

The address A of the string specifies the byte of the string containing the sign. Digits of decreasing significance are assigned to bytes of increasing addresses. Thus "+123" is represented as follows:



ZK–1135A–GE

The leading separate numeric string with a value of "–123" is represented as follows:



ZK-1136A-GE

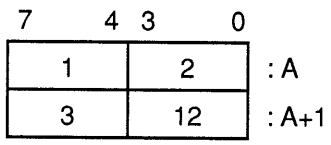
8.2.14 Packed Decimal String

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

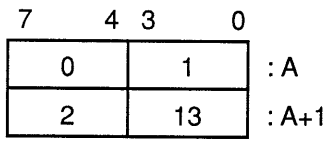
The preferred sign representation is 12 for "+" and 13 for "–". The length L is the number of digits in the packed decimal string (not counting the sign); L must be in the range 0 to 31. When the number of digits is odd, the digits and the sign fit into a string of bytes whose length is defined by the following equation: $L/2$ (*integer part only*) + 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:



ZK-1137A-GE

The packed decimal number "–12" has a length of 2 and is represented as follows:

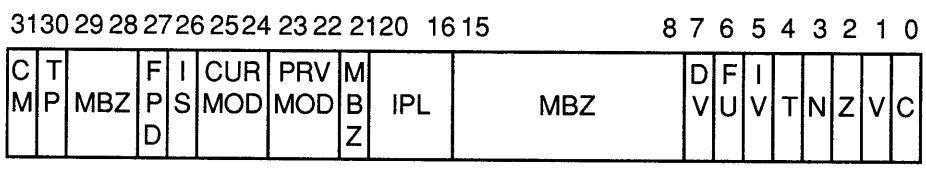


ZK-1138A-GE

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 as follows:



ZK-1139A-GE

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 zero. 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

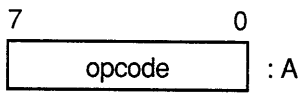
A floating overflow fault is forced after the execution of a floating-point instruction that produced a result too large to be represented in the result operand.

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.

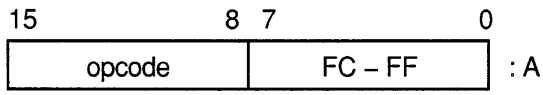
8.5.1 Opcode Formats

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



ZK-1140A-GE

The opcode may extend over 2 bytes; the length depends on the contents of the byte at address A. If, and only if, the value of the byte is FC (hex) to FF (hex), the opcode is 2 bytes long.



ZK–1141A–GE

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 to 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.

In Section 8.7, Table 8–5 is a summary of general register addressing and Table 8–6 is a summary of program counter addressing.

Notation for Describing Addressing Modes

The following notation describes the addressing modes:

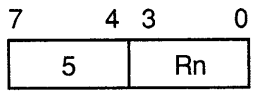
+	Addition
–	Subtraction
*	Multiplication
<-	Is replaced by
=	Is defined as
'	Concatenation
Rn or R[n]	The contents of register n
PC or SP	The contents of register 15 or 14, respectively
(x)	The contents of a location in memory whose address is x
{ }	Arithmetic parentheses that indicate precedence
SEXT(x)	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 as follows:



ZK-1142A-GE

No specifier extension follows.

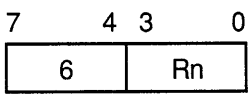
In register mode addressing, the operand is the contents of either register n or (for quadword, D_floating, and certain field operands) register n+1 concatenated with register n.

```
operand =    Rn                              ! If 1 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 as follows:



ZK-1143A-GE

No specifier extension follows.

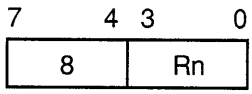
In register deferred mode addressing, the address of the operand is the contents of register n.

```
OA = Rn
operand = (OA)
```

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

8.6.3 Autoincrement Mode

The operand specifier format is as follows:



ZK-1144A-GE

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

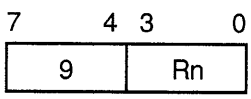
Rn <- Rn + size

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 as follows:



ZK-1145A-GE

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)

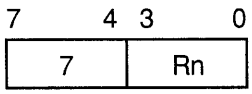
Rn <- Rn + 4

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 as follows:



ZK-1146A-GE

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

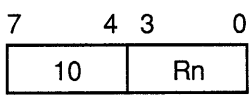
OA = Rn

operand = (OA)

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

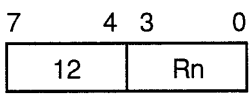
8.6.6 Displacement Mode

There are three operand specifier formats.



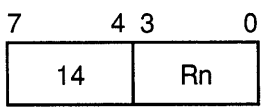
ZK-1147A-GE

The specifier extension is a signed byte displacement that follows the operand specifier. This is the byte displacement mode.



ZK-1148A-GE

The specifier extension is a signed word displacement that follows the operand specifier. This is the word displacement mode.



ZK-1149A-GE

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.

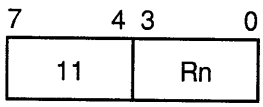
```
OA = Rn + SEXT(displ)          ! If byte or word displacement
     or
     Rn + displ                ! If longword displacement
operand = (OA)
```

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.

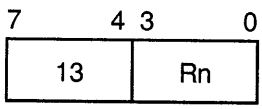
8.6.7 Displacement Deferred Mode

There are three operand specifier formats.



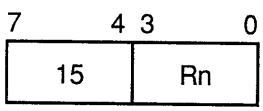
ZK-1150A-GE

The specifier extension is a signed byte displacement that follows the operand specifier. This is the byte displacement deferred mode.



ZK-1151A-GE

The specifier extension is a signed word displacement that follows the operand specifier. This is the word displacement deferred mode.



ZK-1152A-GE

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.

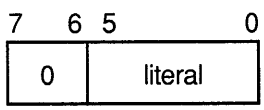
```
OA = (Rn + SEXT(displ))          ! If byte or word displacement
     or
     (Rn + displ)                ! If longword displacement

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 as follows:



ZK–1153A–GE

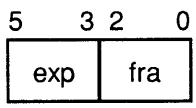
No specifier extension follows.

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)
```

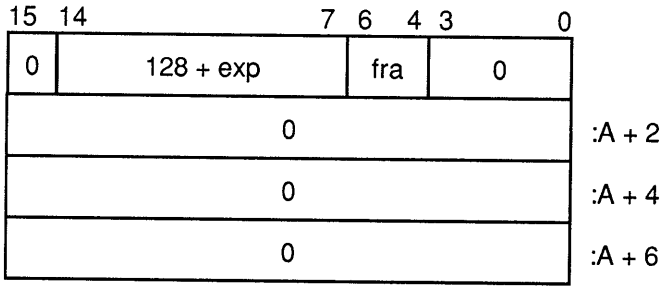
Thus, for these data types, you may use literal mode for values in the range 0 to 63.

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:



ZK–1154A–GE

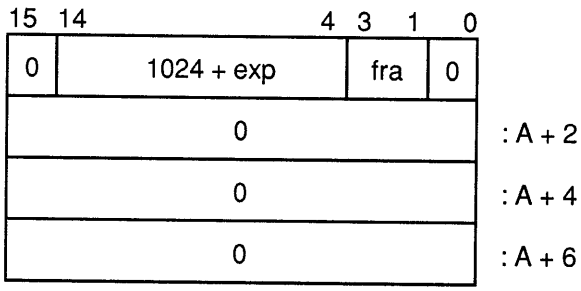
You use the exponent and fraction fields to form an F_floating or D_floating operand as follows:



ZK–1155A–GE

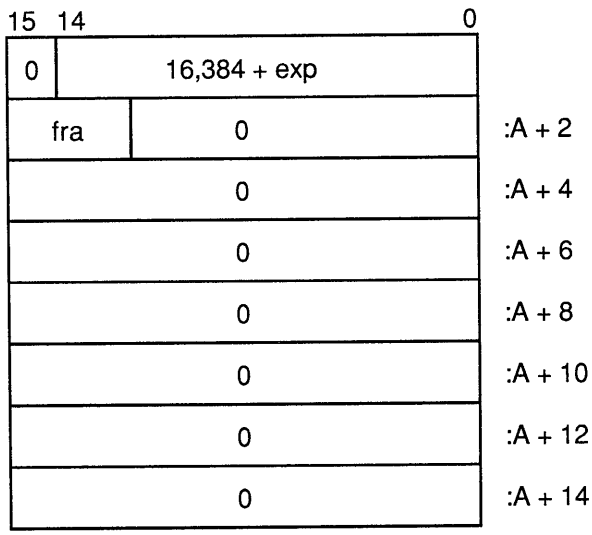
Note that bits 63:32 are not present in an F_floating operand.

You use the exponent and fraction fields to form a G_floating operand as follows:



ZK–1156A–GE

You use the exponent and fraction fields to form an H_floating operand as follows:



ZK–1157A–GE

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

(continued on next page)

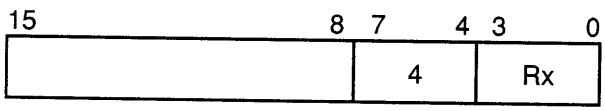
Table 8–4 (Cont.) Floating-Point Literals Expressed as Rational Numbers

Exponent	0	1	2	3	4	5	6	7
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 as follows:



ZK–1158A–GE

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]

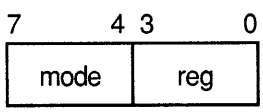
8.7 Summary of General Mode Addressing

This section provides summaries of general register and program counter (PC) addressing.

Table 8–5 is a summary of general register addressing and Table 8–6 is a summary of PC addressing.

8.7.1 General Register Addressing

The general register addressing format is as follows:



ZK–1159A–GE

Table 8–5 General Register Addressing

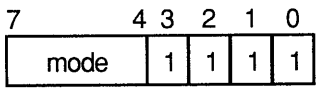
Hex	Dec	Name	Assembler	r mw a v	PC	SP	AP FP	Can Be Indexed?
0–3	0–3	Literal	S^#literal	y f f f f	—	—	—	f
4	4	Indexed	i[Rx]	y y y y y	f	y	y	f
5	5	Register	Rn	y y y f y	u	uq	uo	f
6	6	Register deferred	Rn	y y y y y	u	y	y	y
7	7	Autodecrement	-(Rn)	y y y y y	u	y	y	ux
8	8	Autoincrement	(Rn)+	y y y y y	p	y	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

8.7.2 Program Counter Addressing

The program counter addressing format is as follows:



ZK-1326A-GE

Table 8–6 Program Counter Addressing

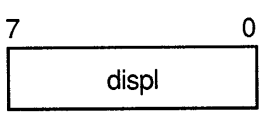
Hex	Dec	Name	Assembler	r mw a v	Can Be Indexed?
8	8	Immediate	I^#constant	y u u y y	u
9	9	Absolute	@#address	y y y y y	y
A	10	Byte relative	B^address	y y y y y	y
B	11	Byte relative deferred	@B^address	y y y y y	y
C	12	Word relative	W^address	y y y y y	y
D	13	Word relative deferred	@W^address	y y y y y	y
E	14	Longword relative	L^address	y y y y y	y
F	15	Longword 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

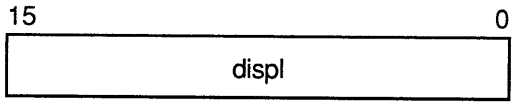
8.8 Branch Mode Addressing Formats

There are two operand specifier formats.



ZK-1160A-GE

The operand specifier is a signed byte displacement.



ZK-1161A-GE

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.