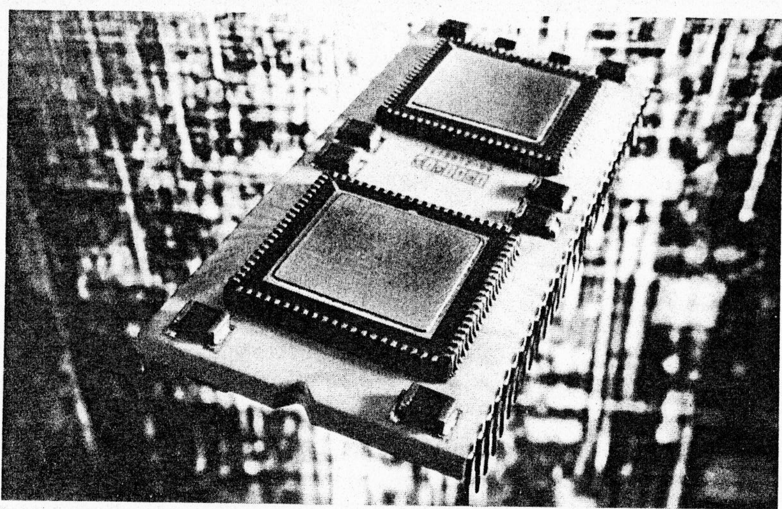
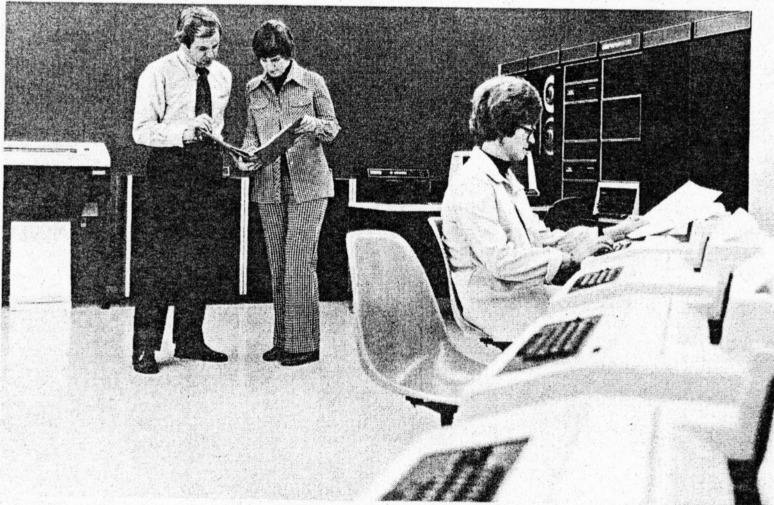

The J-11 chipset.


The PDP-11/70 system.

The tradition continues with the J-11, DIGITAL's newest high-performance microprocessor. It offers the architecture, power, and functions of the PDP-11/70 (the PDP-11 family performance leader) in a single 60-pin package. The J-11 will form the basis of a new line of DIGITAL products. These powerful systems will carry the PDP-11 architecture years into the future.

PDP-11
Architecture Handbook

digital

Copyright© 1983 Digital Equipment Corporation.
All Rights Reserved.

Digital Equipment Corporation makes no representation that the interconnection of its products in the manner described herein will not infringe on existing or future patent rights, nor do the descriptions contained herein imply the granting of license to make, use, or sell equipment constructed in accordance with this description.

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 manual.

DEC, DECnet, DECsystem-10, DECSYSTEM-20, DECtape
DECUS, DECwriter, DIBOL, Digital logo, IAS, MASSBUS, OMNIBUS
PDP, PDT, RSTS, RSX, SBI, UNIBUS, VAX, VMS, VT
are trademarks of
Digital Equipment Corporation

This handbook was designed, produced, and typeset
by DIGITAL's New Products Marketing Group
using an in-house text-processing system.

TABLE OF CONTENTS

CHAPTER 1 ARCHITECTURE AND THE PDP-11 FAMILY
INTRODUCTION 1
THE PDP-11 FAMILY CONCEPT 3
PDP-11 ARCHITECTURE AND SYSTEM PERFORMANCE 5
EVOLUTION OF THE PDP-11 6
PDP-11 MILESTONES 8
EVOLUTION OF THE LSI-11 11
PDP-11 FAMILY ALBUM 13

CHAPTER 2 KEY ELEMENTS OF PDP-11 ARCHITECTURE
INTRODUCTION 27
DATA REPRESENTATION 27
ADDRESSING AND REGISTERS 28
INSTRUCTION SETS 29
TRAPS AND INTERRUPTS 30
MAPPING TO MEMORY AND BUSSES 30
PDP-11 BUS STRUCTURES 31
OTHER TOPICS (APPENDICES) 31

CHAPTER 3 PDP-11 DATA REPRESENTATION
INTEGER DATA TYPES 35
CHARACTER DATA TYPES 36
DECIMAL STRING DATA TYPES 38
FLOATING-POINT DATA FORMATS 49

CHAPTER 4 ADDRESSING MODES
REGISTER MODE 56
REGISTER DEFERRED MODE 57
AUTOINCREMENT MODE 58
AUTOINCREMENT DEFERRED MODE 59
AUTODECREMENT MODE 59
AUTODECREMENT DEFERRED MODE 60
INDEX MODE 61
INDEX DEFERRED MODE 61
USE OF THE PC AS A GENERAL REGISTER 62
PC IMMEDIATE MODE 63
PC ABSOLUTE MODE 64
PC RELATIVE MODE 64
PC RELATIVE DEFERRED MODE 65
SUMMARY OF ADDRESSING MODES 66
GRAPHIC SUMMARY OF PDP-11 ADDRESSING MODES 69

CHAPTER 2

KEY ELEMENTS OF PDP-11 ARCHITECTURE

INTRODUCTION

This chapter is a brief introduction to the main elements of PDP-11 architecture. As we introduce each topic in this chapter, we will refer you to a specific chapter for details. Key elements of PDP-11 architecture include:

- Data representation
- Addressing and registers
- The PDP-11 instruction sets
- Traps and interrupts
- Mapping of memory and busses
- PDP-11 bus structures

DATA REPRESENTATION

The PDP-11 architecture accommodates a variety of data types, which may be separated into categories according to the groups of instructions that manipulate them. They are:

- Integer data
- Floating point data
- String data

Integer data types are manipulated by the basic PDP-11 instruction set. The string data types are manipulated by the Commercial Instruction Set, which is offered as an option on some PDP-11 processors. Floating-point data types are manipulated by the Floating-Point Instruction Set (FP-11) which runs on a Floating-Point Processor (FPP). An FPP may be either a separate processor or a microcode option.

Data representation is treated in detail in Chapter 3.

Integer data types include 8-bit bytes and 16-bit words. Integer data types are stored in memory in binary form, which is represented entirely in ones and zeroes. (Computers use binary representation because it is simple: a one can be represented by the presence of a charge or a switch set on, while a zero can be the absence of charge or a switch set off. Thus, a large number could be represented by a series of switches set on or off to represent binary digits.) In an integer data word or byte, the leftmost, or most significant bit (MSB) can be used as a sign bit. The MSB is always zero for positive values and one for negative values.

Floating point data types are the computer's way of handling very large or small numbers. They represent approximations to quantities using a scientific notation consisting of a sign, the exponent of a power of two, and a fraction between .5 (inclusive) and 1.0 (exclusive). The FP11 instruction set provides two types of floating point data, one 32-bits long and the other 64-bits long. The 32-bit data are called single-precision floating, or just floating; the 64-bit data are called double-precision floating or just double.

The instructions that manipulate floating point data are explained in Chapter 6.

String data types may be divided into two categories:

- Character string data
- Decimal string data

Character string data have their own data type in the Commercial Instruction Set (hereafter called CIS). A character string consists of a contiguous sequence of bytes in memory specified by beginning address and length. This data type is useful when representing names, data records, or text. The manipulations done on character strings include copying, searching, concatenating, and translating. A character string that contains ASCII codes for decimal digits is called a numeric string.

The CIS is treated in detail in Chapter 7.

Decimal string data have two data types: numeric strings and packed strings. Both have similar arithmetic and operational properties; they differ primarily in their representation of signs and the placement of digits in memory. Decimal strings are used to represent numbers in decimal form (which may not be used for computation), as opposed to binary integer form.

ADDRESSING AND REGISTERS

Within the processor there are locations called **general purpose registers** (GPRs) that can be used for temporary data storage, addressing, and as accumulators during computations. Eight 16-bit general purpose registers are available for use with the PDP-11 instruction set, but some of these registers have special uses. For example, one register is designated the Program Counter (PC); another is the Stack Pointer (SP).

Any operation performed by the computer can be specified by an instruction. Each instruction specifies:

- Function to be performed (operation code)
- General purpose register to be used in locating the data (operand)

- Addressing mode to specify how the registers are used

The datum being manipulated by an instruction is called the instruction operand. An instruction operand can be located in main memory, in a general register, or in the instruction itself. The method for specifying an operand's location is called the operand addressing mode. These addressing modes use the registers in a variety of ways to locate the operand or its address. Addressing and registers are explained further in Chapter 4.

INSTRUCTION SETS

There are three instruction sets available on PDP-11 processors:

- PDP-11
- Floating-point
- Commercial

The PDP-11 instruction set is standard on all PDP-11 family processors; the Commercial Instruction Set and the Floating-Point Instruction Set are optional on certain processors.

PDP-11 Instruction Set

The PDP-11 instruction set offers a wide selection of operations and addressing modes. There are seven categories of PDP-11 instructions:

- Single-operand
- Double-operand
- Branch
- Jump and Subroutine
- Trap
- Miscellaneous
- Condition code

To save memory space and simplify control and communications, PDP-11 instructions allow byte and word addressing in both single-operand and double-operand formats. Double-operand instructions let you perform several operations with a single instruction. Branch, jump, and subroutine instructions each provide a means for diverting program flow to a specified location. Trap instructions specify another form of change in program flow, but to a predetermined location. Condition code instructions set or clear the condition codes (four bits in the Processor Status Word [PSW] indicating the results of previous instructions).

See Chapter 5 for more information on the PDP-11 instruction set.

Floating-Point Instruction Set

Floating point data types are manipulated by the Floating-Point Instruction Set (FP-11), which runs on an optional floating-point processor, which may be either a separate processor or microcode. (A microcoded floating point processor is standard on the J-11 chipset.)

The Floating-Point Instruction Set is described in Chapter 6.

Commercial Instruction Set

COBOL processing makes extensive use of string data types, which are manipulated by the Commercial Instruction Set (CIS). The CIS is offered as an option on some PDP-11 processors.

The CIS is discussed in Chapter 7.

TRAPS AND INTERRUPTS

Processor Traps

PDP-11 processor traps are triggered by power failures and certain hardware and software errors. Processor traps protect the programmer and the processor. They save the current PC and Processor Status Word (PSW) and pass control to a trap-handling routine. This saves the programmer work. They also protect the processor and the operating system, if the programmer inadvertantly codes an illegal instruction, or an instruction which might violate the integrity of the operating system. A trap causes the processor to execute instructions pointed to by a certain permanently assigned address. **Trap instructions** are used to make an orderly transition to the trap routine and save the context of the CPU.

Interrupts

Interrupts are used by certain system devices to reduce their wait for CPU service. PDP-11 processors offer the programmer fast interrupt handling. Only four memory cycles are required from the time an interrupt request is issued until the first instruction of the interrupt routine begins execution. By using interrupts, the processor is relieved of doing routine control functions for peripheral devices. Instead, the processor can ignore the peripheral, which may be reading a tape or doing some time-consuming operation, until the peripheral is finished and has data ready for the CPU. Then the device will use an interrupt to get the CPU's attention before it can execute the next instruction.

Traps and interrupts are examined in Chapter 8.

MAPPING TO MEMORY AND BUSSES

Memory management matches the virtual addresses generated by the

CPU with physical addresses in memory and with physical I/O bus addresses. It also protects operating system software and shared routines from modification and allocates protected memory space for each user. The UNIBUS map is a hardware device separate from the memory management unit. The UNIBUS map converts 18-bit UNIBUS addresses to 22-bit memory addresses. There is no map on the extended LSI-11 Bus. Processors and peripherals can generate and present 22-bit addresses directly to the extended LSI-11 Bus.

Memory management and bus mapping are described in Chapter 9.

PDP-11 BUS STRUCTURES

The two PDP-11 physical I/O busses—the UNIBUS and the LSI-11 Bus—are both covered in Chapter 10. The brief, tutorial overview of the UNIBUS and LSI-11 Bus found in that chapter is augmented by appendices that contain timing diagrams and technical specifications.

UNIBUS

The UNIBUS, DIGITAL's unique data bus, was the first data bus in the history of the minicomputer industry to enable devices to send, receive, or exchange data without processor intervention or intermediate buffering in memory. The UNIBUS forms the hardware and software backbone of the PDP-11/24 and PDP-11/44 processors. Memory elements on the UNIBUS have ascending addresses starting at zero, while registers storing I/O data or the status of individual peripheral devices have addresses in the highest 8 Kbytes of addressing space. Peripheral devices may have one or more addresses.

LSI-11 Bus

The LSI-11 Bus is the low-end member of DIGITAL's bus family. Most DIGITAL microcomputers use the LSI-11 Bus or the extended LSI-11 Bus. The LSI-11 Bus operates very much like the UNIBUS, but to make it more cost-effective for microcomputer applications, it has fewer signal lines. Both the LSI-11 Bus and the UNIBUS are treated in Chapter 10.

OTHER TOPICS (APPENDICES)

Other topics related to PDP-11 architecture are included in appendices. The topic of each appendix is listed and briefly discussed below.

Assignment of Bus Addresses and Vectors

Appendix A covers both the LSI-11 Bus and the UNIBUS. Topics covered include:

- I/O Page Device Addresses

- Interrupt and Trap Vectors
- Priority Ranking for Floating Vectors
- Floating CSR Address Devices
- Device Addresses

PDP-11 Family Differences

Appendix B contains a family differences table that shows in detail the issues involved in software migration between PDP-11 family members. Any program developed using PDP-11 operating systems with higher level languages will migrate with very little difficulty. Certain assembly language applications may require slight modifications for a smooth migration.

The Floating Instruction Set

The Floating Instruction Set (FIS) is a software option for the LSI-11/2 processor. The FIS consists of four special floating instructions that accelerate floating point calculations. The FIS is covered in Appendix C.

UNIBUS Timing Diagrams

UNIBUS timing diagrams and other technical details are given in Appendix D.

LSI-11 Bus Technical Specifications

Topics covered in Appendix E include LSI-11 Bus timing diagrams, and bus pin-out descriptions.

Programming Techniques

PDP-11 processors offer the programmer a combination of flexibility and power. The instruction set, addressing modes, and programming techniques play together to help you develop new software or use existing software. Programming techniques that pertain to architecture are included in this handbook. These include:

- Stacks
- Subroutine linkage
- Reentrancy

Stacks are a basic element of the PDP-11 architecture. They are areas of memory set aside by the programmer or the operating system for temporary storage and linkage. A stack is handled on a last-in/first-out (LIFO) basis: items are retrieved in the reverse of their storage order. A PDP-11 stack starts at the highest location reserved for it and expands downward to lower addresses as items are added.

Often, one of the general purpose registers must be used in a subroutine or interrupt service routine and then returned to its original value.

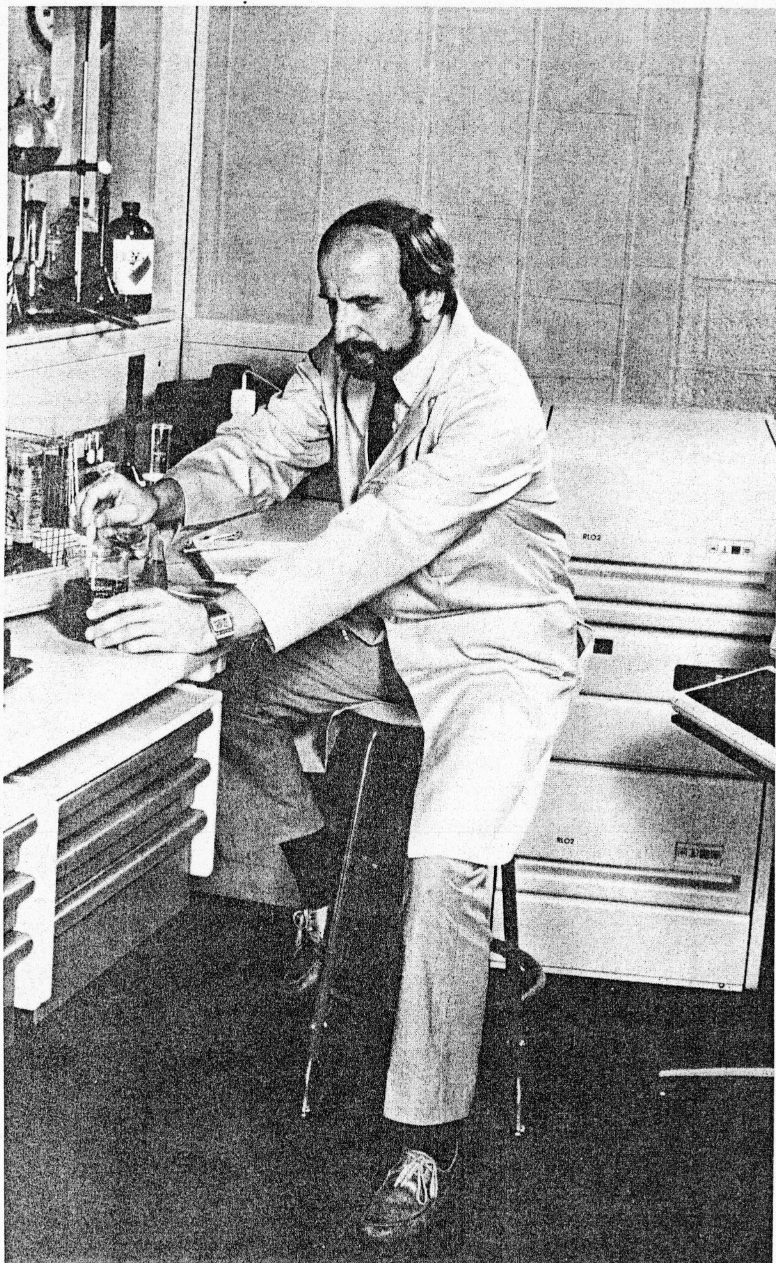
A stack can be used to store the contents of the registers involved. A stack is also useful to store the **linkage** information between a **subroutine** and its calling program. In many cases, operations performed by the subroutine can be applied directly to data located on or referenced by the stack without actually moving the data into the subroutine.

Reentrancy is the ability to share a single copy of a program among different users or different tasks. This makes more efficient use of memory. Reentrant routines differ from ordinary subroutines in that it is not necessary for reentrant routines to finish processing a given task before they can be used by another task.

PDP-11 programming techniques and examples are covered in Appendix F.

Glossary

For definitions of terminology used in this book, refer to the Glossary. The Glossary is at the end of the book, between Appendix F and the Index.



CHAPTER 3

DATA REPRESENTATION

Data representation is an important aspect of computer architecture. To deal efficiently with different kinds of information, a computer architecture must allow for a range of data types. The programmer's choice of data type should be a function of the application rather than the computer. However, some computers must use nonstandard addressing techniques with certain data types. These computers require more memory and will execute applications more slowly when using these "problem" data types. PDP-11 architecture avoids these compromises. You can use the data type that best suits your application without worrying about nonstandard addressing techniques.

Another feature of the PDP-11 family's data types is upward compatibility. The PDP-11 data types are a subset of the VAX-11 data types. This can be very convenient if you want to transfer your PDP-11 application to an environment with 32-bit addressing.

The PDP-11 data types may be separated into categories according to the groups of instructions that operate on them. They are:

- Integer data
- Character string data
- Decimal string data
- Floating point data

Integer data types are supported by the basic PDP-11 instruction set. The string data types are used by the Commercial Instruction Set, which is offered as an option on some PDP-11 processors. Floating point data types are manipulated by the Floating-Point Instruction Set (FP-11) which runs on a Floating-Point Processor (FPP) which may be either a separate processor or microcode.

The Commercial Instruction Set (CIS-11) is treated in detail in Chapter Seven. The floating point instructions are described in Chapter 6 (The Floating Point Processor—FP-11) and in Appendix C (The Floating Instruction Set—FIS).

INTEGER DATA TYPES

Integer data types include 8-bit bytes, and 16-bit words. Integer data types are stored in memory in binary form, which is represented entirely in ones and zeroes. As unsigned quantities, integers extend upward from 0. As signed quantities, the integers are represented in two's complement form. This means that a negative number is one greater than the bit-by-bit complement of its positive counterpart. Thus, posi-

tive numbers have a 0 most significant bit (MSB). The MSB or sign bit is always 1 for negative values.

Byte
A byte is eight contiguous bits starting on an addressable byte boundary or located in a register, Rn <7:0>. The bits are numbered from the right 0 through 7. The byte is specified by its address A. When interpreted as a signed quantity, a byte is a two's complement integer with bits increasing in significance from 0 through 6, and with bit 7 designating the sign. The value of the integer is in the range – 128 through 127.

For the purposes of addition, subtraction, and comparison, PDP-11 instructions also provide direct support for the interpretation of a byte as an unsigned integer with a value in the range 0 through 255.

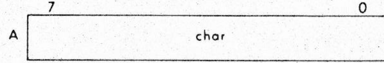
Word
A word, two contiguous bytes, starts on an arbitrary word boundary or is located in a register Rn<15:0>.

Words are specified by their address A, the address of the byte containing bit 0. When interpreted as a signed quantity, a word is a two's complement integer with bits increasing in significance from 0 through 14, and with bit 15 designating the sign. The value of the integer is in the range – 32768 through 32767. For the purposes of addition, subtraction, and comparison, PDP-11 instructions also provide direct support for the interpretation of a word as an unsigned integer with a value in the range 0 through 65535.

CHARACTER DATA TYPES

There are three different character data types. The "character" is a single byte, and is an abbreviated string of length 1. The "character string" is a contiguous group of bytes in memory. The third is a "character set."

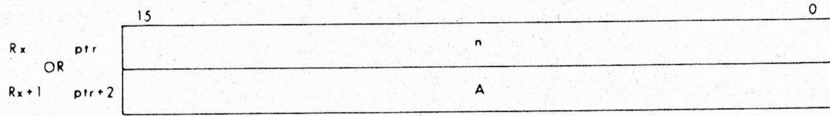
The character is an 8-bit byte:



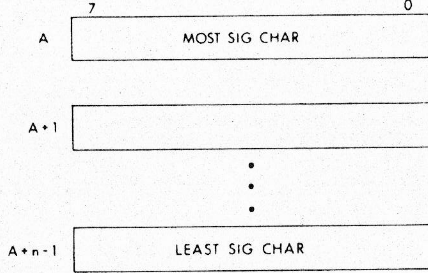
The character is used as an operand by CIS-11 instructions. When it appears in a general register, the character is in the low-order half; the high-order half of the register must be zero. When it appears in the instruction stream, the character is in the low-order half of a word; the high-order half of the word must be zero. If the high-order half of a word which contains a character is nonzero, the effect of the instruction which uses it will be UNPREDICTABLE.

A character string is a contiguous sequence of bytes in memory that begins and ends on a byte boundary. It is addressed by its most significant character (lowest address). The highest address is the least significant character. It is specified by a two-word descriptor with the attributes of length and lowest address. The length is an unsigned binary integer which represents the number of characters in the string and may range from 0 to 65,535. A character string with zero length is said to be vacant: its address is ignored. A character string with nonzero length is said to be occupied.

The character string descriptor is used as an operand by CIS-11 instructions. It appears in two consecutive general registers, or in two consecutive words in memory pointed to by a word in the instruction stream. The following figure shows the descriptor for a character string of length "n" starting at address "A" in memory:



The following figure shows the character string in memory:

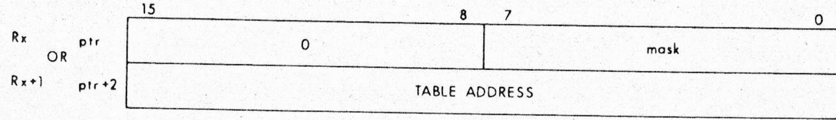


A "character set" is a subset of the 256 possible characters that can be encoded in a byte. It is specified by a descriptor which consists of the address of a 256-byte table and an 8-bit mask. The address is of the zeroth byte in the table. Each byte in the table specifies up to eight orthogonal character subsets of which the corresponding character is a member. The mask selects which combinations of these orthogonal subsets constitute the entire character set. In effect, each bit in the mask corresponds to one of eight orthogonal subsets that may be en-

coded by the table. The mask specifies the union of the selected subsets into the character set. Typical sets would be: uppercase, lowercase, nonzero digits, end of line, etc.

Operationally, a character (char) is considered to be in the character set if the evaluation of (M[table.adr + char] AND mask) is not equal to zero. The character is not in the character set if the evaluation is zero. Each byte in the table indicates which combination of up to eight orthogonal character subsets (i.e., one for each of the eight bit vectors 00000001_2 00000010_2 00000100_2 00001000_2 00010000_2 00100000_2 01000000_2 and 10000000_2) the corresponding character is a member. The mask specifies which union of the eight orthogonal character subsets constitute the total character set. For example, if the eight-bit vector 00000001_2 appearing in the table corresponds to the character subset of all uppercase alphabetic characters, 00000010_2 appearing in the table corresponds to the character subset of all lowercase alphabetic characters, and 00000100_2 appearing in the table corresponds to the decimal digits, then using the mask 00000011_2 with this table specifies the character set of all alphabetic characters, and using the mask 00000111_2 specifies the character set of all alphanumeric characters.

The character set descriptor is used as an operand by CIS-11 intructions. It appears in two consecutive general registers, or in two consecutive words in memory pointed to by a word in the instruction stream. If the high-order half of the first descriptor word is nonzero, the effect of an instruction which uses a character set will be UNPREDICTABLE.



DECIMAL STRING DATA TYPES

Two classes of decimal string data types—numeric strings and packed strings—are defined. Both have similar arithmetic and operational properties; they primarily differ in the representation of signs and the placement of digits in memory.

The numeric string data types are signed zoned, unsigned zoned, trailing overpunch, leading overpunched, trailing separate and leading separate. The packed string data types are signed packed and unsigned packed. Instructions which operate on numeric strings permit each numeric string operand to be separately specified; similarly,

packed string instructions permit each packed string operand to be separately specified. Thus, within each of the two classes of decimal strings, the operands of an instruction may be of any data type within the appropriate class.

Decimal strings exist in memory as contiguous bytes which begin and end on a byte boundary. They represent numbers consisting of 0 to 31_{10} digits, in either sign-magnitude or absolute-value form. Sign-magnitude strings (SIGNED) may be positive or negative; absolute-value strings (UNSIGNED) represent the absolute value of the magnitude. Decimal numbers are whole integer values with an implied decimal radix point immediately beyond the least significant digit; they may be conceptually extended with zero digits beyond the most significant digit.

A four-bit binary coded decimal representation is used for most digits in decimal strings. A four-bit half byte is called a "nibble" and may be used to contain a binary bit pattern which represents the value of a decimal digit. The following table shows the binary nibble contents associated with each decimal digit:

digit	nibble	digit	nibble
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

Each decimal string data type may have several representations. These representations permit a certain latitude when accepting source operands. Decimal string data types have a PREFERRED representation, which is a valid source representation and which is used to construct the destination string. Additional ALTERNATE representations are provided for some decimal data types when accepting source operands.

Decimal strings used as source operands will not be checked for validity. Instructions will produce UNPREDICTABLE results if a decimal string used as a source operand contains an invalid digit encoding, invalid sign designator, or, in the case of overpunched numbers, an invalid sign/digit encoding.

When used as a source, decimal strings with zero magnitude are unique, regardless of sign. Thus, both positive and negative zero have identical interpretations.

Conceptually, decimal string instructions first determine the correct result, and then store the decimal string representation of the correct

result in the destination string. A result of zero magnitude is considered to be positively signed. If the destination string can contain more digits than are significant in the result, the excess most significant destination string digits have zero digits stored in them. If the destination string cannot contain all significant digits of the result, the excess most significant result digits are not stored; the instruction will indicate decimal overflow. Note that negative zero is stored in the destination string as a side effect of decimal overflow where the sign of the result is negative and the destination is not large enough to contain any nonzero digits of the result.

If the destination string has zero length, no resulting digits will be stored. The sign of the result will be stored in separate and packed strings, but not in zoned and overpunched strings. Decimal overflow will indicate a nonzero result.

Decimal String Descriptors

Decimal strings are represented by a two-word descriptor. The descriptor contains the length, data type, and address of the string. It appears in two consecutive general registers (register form of instructions), or in two consecutive words in memory pointed to by a word in the instruction stream (in-line form of instructions). The unused bits are reserved by the architecture and must be 0. The effect of an instruction using a descriptor will be unpredictable if any nonzero reserved field in the descriptor contains nonzero values or a reserved data type encoding is used. The design of the numeric and packed string descriptors are identical:

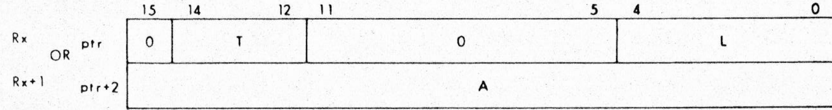
First Word

length <4:0>	Number of digits specified as an unsigned binary integer
data type <14:12>	Specifies which decimal data type representation is used

Second Word

address <15:0>	Specifies the address of the byte which contains the most significant digit of the decimal string

The following figure shows the descriptor for a decimal string of data type "T" whose length is "L" digits and whose most significant digit is at address "A":



The encodings (in binary) for the NUMERIC string data type field are:

000	signed zoned
001	unsigned zoned
010	trailing overpunch
011	leading overpunch
100	trailing separate
101	leading separate
110	—reserved to DIGITAL
111	—reserved to DIGITAL

The encodings (in binary) for the PACKED string data type field are:

000	—reserved to DIGITAL
001	—reserved to DIGITAL
010	—reserved to DIGITAL
011	—reserved to DIGITAL
100	—reserved to DIGITAL
101	—reserved to DIGITAL
110	signed packed
111	unsigned packed

Packed Strings

Packed strings can store two decimal digits in each byte. The least significant (highest addressed) byte contains the sign of the number in bits <3:0> and the least significant digit in bits <7:4>.

Signed Packed Strings — The preferred positive sign designator is 1100_2; alternate positive sign designators are 1010_2, 1110_2 and 1111_2. The preferrred negative sign designator is 1101_2; the alternate negative sign designator is 1011_2. Source strings will properly accept both the preferred and alternate designators; destination strings will be stored with the preferred designator.

Unsigned Packed Strings — The unsigned sign designator is 1111_2.

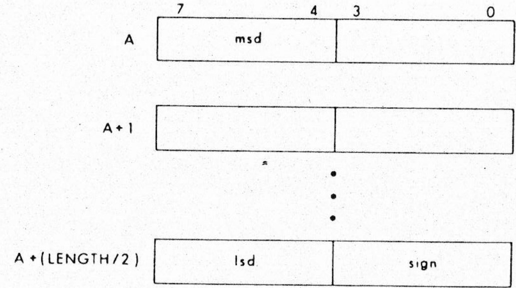
PACKED SIGN NIBBLE (in binary):

sign nibble	**preferred designator**	**alternate designators**
positive	1100	1010 , 1110 , 1111
negative	1101	1011
unsigned	1111	

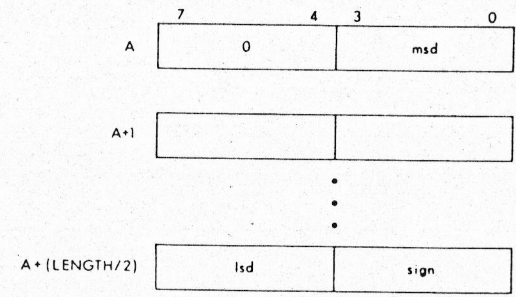
For other than the least significant byte, bytes contain two consecutive digits—the one of lower significance in bits <3:0> and the one of higher significance in bits <7:4>. For numbers whose length is odd, the most significant digit is in bits <7:4> of the lowest addressed

bytes. Numbers with an even length have their most significant digit in bits <3:0> of the lowest addressed byte; bits <7:4> of this byte must be zero for source strings, and are cleared to 0000 for destination strings. Numbers with a length of one occupy a single byte and contain their digit in bits <7:4>. The number of bytes which represent a packed string is [length/2] + 1 (integer division where the fractional portion of the quotient is discarded).

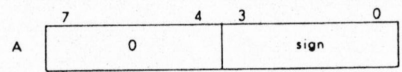
The following is a packed string with an odd number of digits:



The following is a packed string with an even number of digits:



A zero-length packed string occupies a single byte of storage; bits <7:4> of this byte must be zero for source strings, and are cleared to 0000 for destination strings. Bits <3:0> must be a valid sign for source strings, and are used to store the sign of the result for destination strings. When used as a source, zero-length strings represent operands with zero magnitude. When used as a destination, they can only reflect a result of zero magnitude without indicating overflow. The following is a zero-length packed string:



A valid packed string is characterized by:

1. A length from 0 to 31_{10} digits.
2. Every digit nibble is in the range 0000_2 to 1001_2.
3. For even length sources, bits <7:4> of the lowest addressed byte are 0000_2.
4. Signed packed strings—sign nibble is either 1010_2, 1011_2, 1100_2, 1101_2, 1110_2 or 1111_2.
5. Unsigned packed strings—sign nibble is 1111_2.

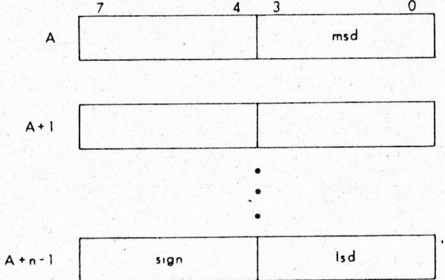
Zoned Strings

Zoned strings represent one decimal digit in each byte. Each byte is divided into two portions—the high-order nibble (bits <7:4>) and the low-order nibble (bits <3:0>). The low-order nibble contains the value of the corresponding decimal digit.

Signed Zoned Strings — When used as a source string, the high-order nibble of the least significant byte contains the sign of the number; the high-order nibbles of all other bytes are ignored. Destination strings are stored with the sign in the high-order nibble of the least significant byte, and 0011_2 in the high-order nibble of all other bytes. 0011_2 in the high-order nibble corresponds to the ASCII encoding for numeric digits. The positive sign designator is 0011_2; the negative sign designator is 0111_2.

Unsigned Zoned Strings — When used as a source string, the high-order nibbles of all bytes are ignored. Destination strings are stored with 0011_2 in the high-order nibble of all bytes.

The number of bytes needed to contain a zoned string is identical to the length of the decimal number.



A zero-length, zoned string does not occupy memory; the address portion of its descriptor is ignored. When used as a source, zero length strings provide operands with zero magnitude; when used as a destination, they can only accurately reflect a result of zero magnitude (the sign of the operation is lost). An attempt to store a nonzero result will be indicated by setting overflow.

A valid zoned string is characterized by:

1. A length from 0 to 31_{10} digits.
2. The low-order nibbles of each byte are in the range 0000_2 to 1001_2.
3. Signed zoned strings—The high order nibble of the least significant byte is either 0011_2 or 0111_2.

Overpunch Strings

Overpunch strings represent one decimal digit in each byte. Trailing overpunch strings combine the encoding of the sign and the least significant digit; leading overpunch strings combine the encoding of the sign and the most significant digit. Bytes other than the byte in which the sign is encoded are divided into two portions—the high-order nibble (bits <7:4>) and the low-order nibble (bits <3:0>). The low-order nibble contains the value of the corresponding decimal digit. When used as a source string, the high-order nibble of all bytes which do not contain the sign are ignored. Destination strings are stored with 0011_2 in the high-order nibble of all bytes which do not contain the sign. 0011_2 in the high-order nibble corresponds to the ASCII encoding for numeric digits.

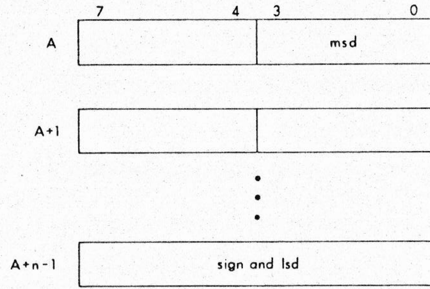
The following table shows the sign of the decimal string and the value of the digit which is encoded in the sign byte. Source strings will properly accept both the preferred and alternate designators; destination strings will store the preferred designator. The preferred designators correspond to the ASCII graphics "A" to "R," "{," and "." The alternate designators correspond to the ASCII graphics "0" to "9," "[," "?," "]," "!" and ":".

OVERPUNCH SIGN/DIGIT BYTE (in binary):

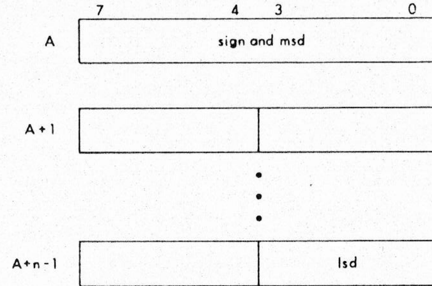
overpunch sign/digit	preferred designator	alternate designators
+0	01111011	00110000 , 01011011 , 00111111
+1	01000001	00110001
+2	01000010	00110010
+3	01000011	00110011
+4	01000100	00110100
+5	01000101	00110101
+6	01000110	00110110
+7	01000111	00110111
+8	01001000	00111000
+9	01001001	00111001
−0	01111101	01011101 , 00100001 , 00111010
−1	01001010	
−2	01001011	
−3	01001100	
−4	01001101	
−5	01001110	
−6	01001111	
−7	01010000	
−8	01010001	
−9	01010010	

The number of bytes needed to contain an overpunch string is identical to the length of the decimal number.

The following is a trailing overpunch string:



The following is a leading overpunch string:



A zero-length overpunch string does not occupy memory; the address portion of its descriptor is ignored. When used as a source, zero-length strings provide operands with zero magnitude; when used as a destination, they can only accurately reflect a result of zero magnitude (the sign of the operation is lost). An attempt to store a nonzero result will be indicated by setting overflow.

A valid overpunch string is characterized by:

1. A length from 0 to 31_{10} digits.
2. The low-order nibble of each digit byte is in the range 0000_2 to 1001_2.
3. The encoded sign/digit byte contains values from the above table of preferred and alternate overpunch sign/digit values.

Separate Strings

Separate strings represent one decimal digit in each byte. Trailing separate strings encode the sign in a byte immediately beyond the least significant digit; leading separate strings encode the sign in a byte immediately beyond the most significant digit. Bytes other than the byte in which the sign is encoded are divided into two portions—the high-order nibble (bits <7:4>) and the low-order nibble (bits <3:0>). The low order nibble contains the value of the corresponding decimal digit.

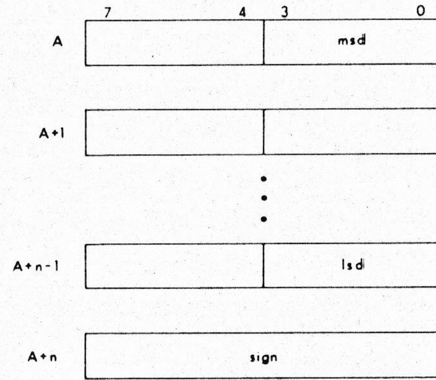
When used as a source string, the high-order nibbles of all digit bytes are ignored. Destination strings are stored with 0011_2 in the high-order nibble of all digit bytes. 0011_2 in the high-order nibble corresponds to the ASCII encoding for numeric digits. The preferred positive sign designator is 00101011_2 and the alternate positive sign designator is 00100000_2. The negative sign designator is 00101101_2. These designators correspond to the ASCII encoding for "+," "space," and "–."

SEPARATE SIGN BYTE:

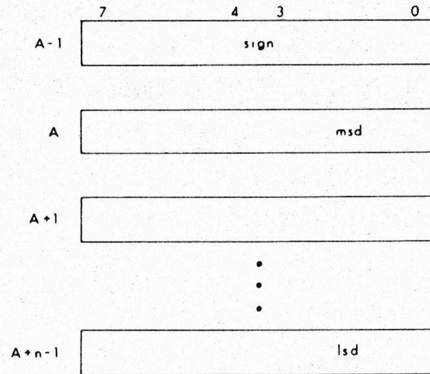
sign byte	**preferred designator**	**alternate designator**
positive	00101011_2	00100000_2
negative	00101101_2	

The number of bytes needed to contain a leading or trailing separate string is identical to (length + 1).

The following is a trailing separate string:

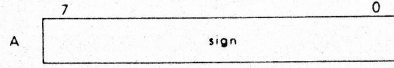


The following is a leading separate string:

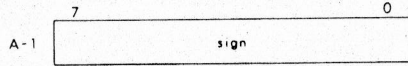


A zero-length separate string occupies a single byte of memory which contains the sign. When used as a source, zero-length strings provide operands with zero magnitude; when used as a destination, they can only reflect a result of zero magnitude without indicating overflow; the sign of the result is stored.

The following is a zero-length trailing separate string:



The following is a zero-length leading separate string:



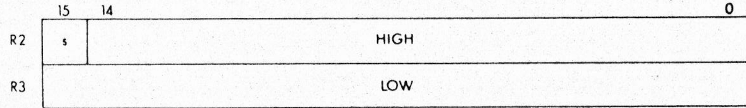
A valid separate string is characterized by:

1. A length from 0 to 31_{10} digits.
2. The low-order nibble of each digit byte is in the range 0000_2 to 1001_2.
3. The sign byte is either 00100000_2, 00101011_2 or 00101101_2.

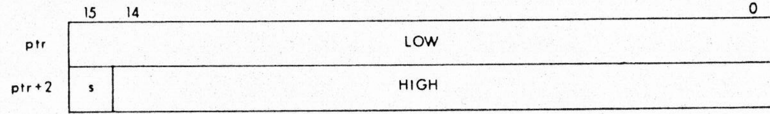
Long Integer

Long integers are 32-bit binary two's complement numbers organized as two words in consecutive registers or in memory—no descriptor is used. One word contains the high-order 15 bits. The sign is in bit <15>; bit <14> is the most significant. The other word contains the low-order 16 bits with bit <0> the least significant. The range of numbers that can be represented is −2,147,483,648 to +2,147,483,647.

The register form of decimal convert instructions uses a restricted form of long integer with the number in the general register pair R2-R3:



The in-line form of decimal convert instructions reference the long integer by a word address pointer which is part of the instruction stream:



Note that these two representations of long integers differ. There is no single representation of long integer among EAE, EIS, FPP and software. The "register form" was selected to be compatible with EIS; the "in-line form" was selected to be compatible with current standard software usage.

FLOATING POINT DATA FORMATS

Floating point data are used only by processors which include a floating point option (standard on the MICRO/J-11). The floating point instruction set (FP11) is covered in Chapter 6 of this book.

Mathematically, a floating point number may be defined as having the form (2**K)*f, where K is an integer and f is a fraction. For a nonvanishing number, K and f are uniquely determined by imposing the condition $\frac{1}{2} \leq f < 1$. The fractional part, f, of the number is then said to be normalized. For the number 0, f must be assigned the value 0, and the value of K is indeterminate.

The FP11 floating point data formats are derived from this mathematical representation for floating point numbers. The value of a floating datum is in the approximate range .29*10**−38 through 1.7*10**38. Two types of floating point data are provided. In single-precision, or floating mode, the data are 32 bits long. In double-precision, or double mode, the data are 64 bits long. Sign magnitude notation is used.

Nonvanishing Floating Point Numbers

The fractional part, f, is assumed to be normalized, so that its most significant bit must be 1. This 1 is the **hidden** bit; it is not stored explicitly in the data word, but the microcode restores it before carrying out arithmetic operations. The floating and double modes respectively reserve 23 and 55 bits for f. These bits, with the hidden bit, imply effective fractions of 24 bits and 56 bits.

Eight bits are reserved for storage of the exponent K in excess 128 (200_8) notation (i.e., K + 200_8), giving a biased exponent. Thus, exponents from −128 to +127 are represented by 0 to 377_8, or 0 to 255_{10}.

For reasons listed below, a biased exponent of 0 (true exponent of -200_8), is reserved for floating point 0. Thus, exponents are restricted to the range -127 to $+127$ inclusive (-177_8 to $+177_8$) or, in excess 200_8 notation, 1 to 377_8.

The remaining bit of the floating point word is the sign bit. The number is negative if the sign bit is a 1.

Floating Point Zero

Because of the hidden bit, the fractional part is not available to distinguish between 0 and nonvanishing numbers whose fractional part is exactly ½. Therefore, the FP11 reserves a biased exponent of 0 for this purpose, and any floating point number with a biased exponent of 0 either traps or is treated as if it were an exact 0 in arithmetic operations. An exact or clean 0 is represented by a word whose bits are all 0s. A dirty 0 is a floating point number with a biased exponent of 0 and a nonzero fractional part. An arithmetic operation for which the resulting true exponent exceeds 177_8 is regarded as producing a floating overflow; if the true exponent is less than -177_8, the operation is regarded as producing a floating underflow. A biased exponent of 0 can thus arise from arithmetic operations as a special case of overflow (true exponent $= -200_8$). Only eight bits are reserved for the biased exponent. The fractional part of results obtained from such overflow and underflow is correct.

The Undefined Variable

The undefined variable is defined as any bit pattern with a sign bit of 1 and a biased exponent of 0. The term **undefined variable** is used, for historical reasons, to indicate that these bit patterns are not assigned a corresponding floating point arithmetic value. Note that the undefined variable is frequently referred to as -0 elsewhere in this specification.

A design objective of the FP11 was to assure that the undefined variable would not be stored as the result of any floating point operation in a program run with the overflow and underflow interrupts disabled. This objective is achieved by storing an exact 0 on overflow and underflow, if the corresponding interrupt is disabled. This feature, together with an ability to detect reference to the undefined variable implemented by the FIUV bit mentioned later, is intended to provide the user with a debugging aid. If -0 occurs, it did not result from a previous floating point arithmetic instruction.

Floating Point Data

Floating point data are stored in words of memory as illustrated:

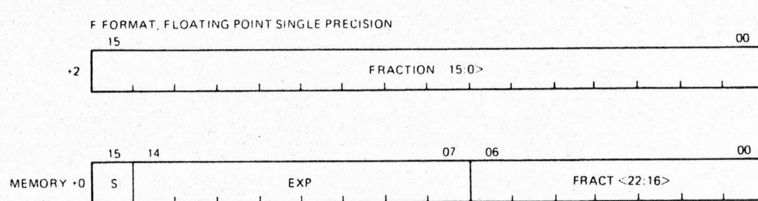


Figure 3-1 Single-Precision Format

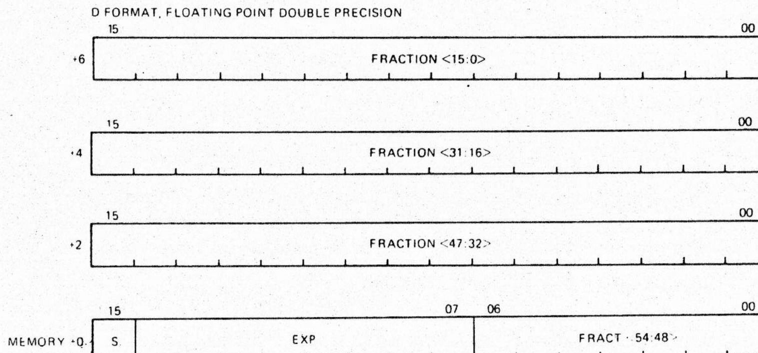
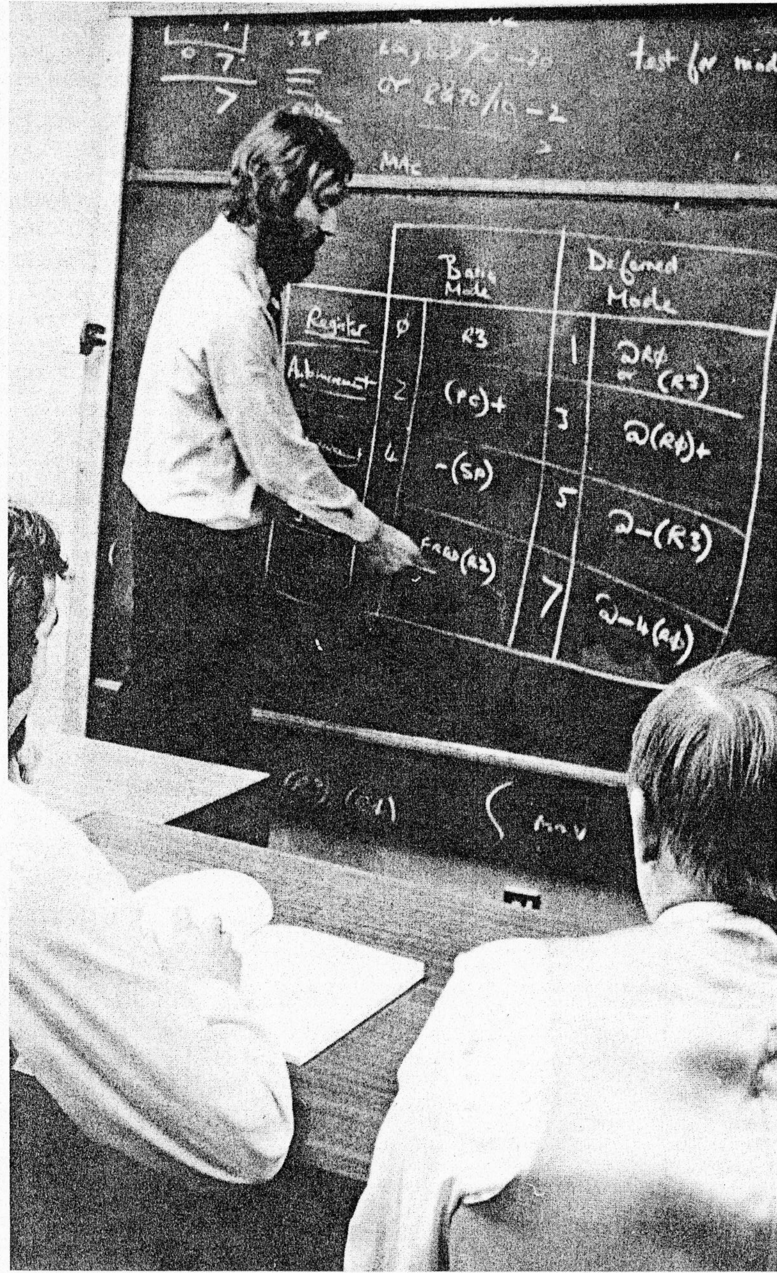


Figure 3-2 Double-Precision Format

The FP11 provides for conversion of floating point to integer format and vice versa. The processor recognizes single-precision integer (I) and double-precision integer long (L) numbers, which are stored in standard two's complement form.



CHAPTER 4

ADDRESSING MODES

In the PDP-11 family, all operand addressing is accomplished through the general purpose registers. To specify the location of data (operand address), one of eight registers is selected with an accompanying addressing mode. Each instruction specifies the:

- Function to be performed (operation code)
- General purpose register to be used when locating the source operand and/or destination operand (where required)
- Addressing mode, which specifies how the selected registers are to be used

The instruction format and addressing techniques available to the programmer are of particular importance. This combination of addressing modes and the instruction set provides the PDP-11 family with a unique number of capabilities. The PDP-11 is designed to handle structured data efficiently and with flexibility. The general purpose registers implement these functions in the following ways, by acting:

- As accumulators: holding the data to be manipulated
- As pointers: the contents of the register are the address of the operand, rather than the operand itself
- As index registers: the contents of the register are added to an additional word of the instruction to produce the address of the operand; this capability allows easy access to variable entries in a list

Using registers for both data manipulation and address calculation results in a variable length instruction format. If registers alone are used to specify the data source and destination, only one memory word is required to hold the instruction. In certain modes, two or three words may be utilized to hold the basic instruction components. Special addressing mode combinations enable temporary data storage for convenient dynamic handling of frequently accessed data. This is known as **stack-addressing**. For a discussion about using the stack, please refer to Appendix F. Register 6 is always used as the hardware stack pointer, or SP. Register 7 is used by the processor as its program counter (PC). Thus, the register arrangement to be considered in conjunction with instructions and with addressing modes is: registers 0-5 are general purpose registers, register 6 is the hardware stack pointer, and register 7 is the program counter. See Chapter 5 for a description of the full instruction set and its formats.

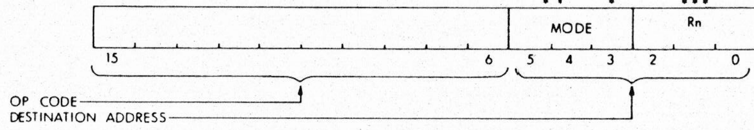
To illustrate the use of the various addressing modes clearly, the following instructions are used in this chapter:

Mnemonic	Description	Octal Code
CLR	Clear (Zero the specified destination word.)	0050DD
CLRB	Clear Byte (Zero the specified destination byte.)	1050DD
INC	Increment (Add one to contents of destination word.)	0052DD
INCB	Increment Byte (Add one to the contents of the destination byte.)	1052DD
COM	Complement (Replace the contents of the destination by its logical one's complement. Each 0 bit is set and each 1 bit is cleared.)	0051DD
COMB	Complement Byte (Replace the contents of the destination byte by its logical one's complement. Each 0 bit is set and each 1 bit is cleared.)	1051DD
ADD	Add (Add the source operand to the destination operand and store the result at the destination address.)	06SSDD

DD = destination field (6 bits)
SS = source field (6 bits)
( ) = contents of

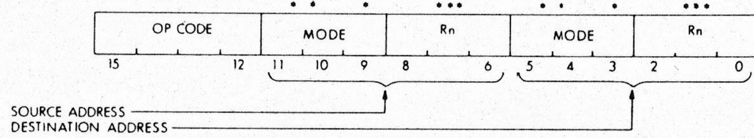
Single- and double-operand instructions use the following formats:

The instruction format for the first word of all single-operand instructions (such as clear, increment, test) is:



Single-Operand Instruction Format

The instruction format for the first word of the double-operand instruction is:



Double-Operand Instruction Format

Bits 5:3 of the source or destination fields specify the binary code of the addressing mode chosen. Bits 2:0 specify the general register to be used.

The four basic addressing modes are:

- Register
- Autoincrement
- Autodecrement
- Index

In a register mode, the content of the selected register is taken as the operand. In autodecrement mode, after the register has been modified, it contains the address of the operand. In autoincrement mode, at the start of the instruction execution, the register contains the address of the operand, and, after the instruction is executed, the ad-

dress of the next higher word or byte memory location. In index mode, the register is added to the displacement, X, to produce the address of the operand.

When bit 3 of the source/destination field is set, indirect addressing is specified and the four basic modes become deferred modes.

Prefacing the register operand(s) with an "@" sign or placing the register in parentheses indicates to the MACRO-11 assembler that deferred (or indirect) addressing mode is being used.

The indirect addressing modes are:

- Register deferred
- Autoincrement deferred
- Autodecrement deferred
- Index deferred

Program counter (register 7) addressing modes are:

- Immediate
- Absolute
- Relative
- Relative deferred

The addressing modes are explained and shown in examples in the following pages. They are summarized, in text and in graphic representation, at the end of the chapter.

REGISTER MODE — MODE 0 Rn

Register mode provides faster instruction execution. There is no need to reference memory to retrieve an operand. Any of the general registers can be used as a simple accumulator. The operand is contained in the selected register (low-order byte for byte operations). Some assemblers require that a general register be defined as follows:

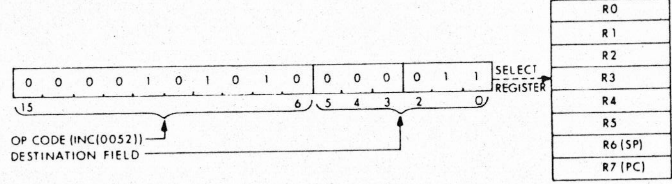
R0 = %0
R1 = %1
R2 = %2

% indicates register definition (as originally known to the assembler).

Register Mode Example

Symbolic	Instruction Octal Code	Description
INC R3	005203	Add one to the contents of R3.

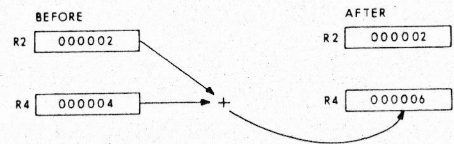
Represented as:



Register Mode Example

Symbolic	Instruction Octal Code	Description
ADD R2,R4	060204	Add the contents of R2 to the contents of R4, replacing the original contents of R4 with the sum.

Represented as:



REGISTER DEFERRED MODE — MODE 1 @Rn or (Rn)

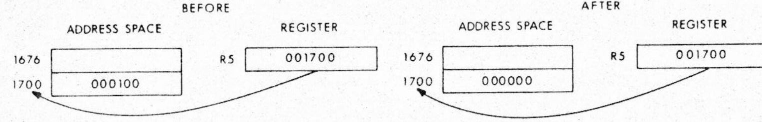
In register deferred mode, the address of the operand is stored in a general purpose register. The address contained in the general purpose register directs the CPU to the operand. The operand is located outside the CPU's general purpose registers, either in memory or in an I/O register.

This mode is used for sequential lists, indirect pointers in data structures, top-of-stack manipulations, and jump tables.

Register Deferred Mode Example

Symbolic	Instruction Octal Code	Description
CLR (R5)	005015	The contents of the location specified in R5 are cleared.

Represented as:



AUTOINCREMENT MODE — MODE 2 (Rn)+

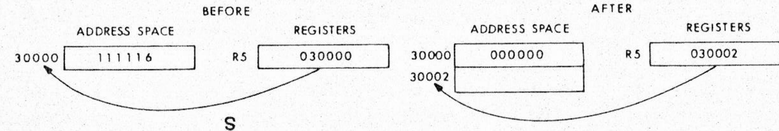
In autoincrement mode, the register contains the address of the operand; the address is automatically incremented after the operand is retrieved. The address then references the next sequential operand. This mode allows automatic stepping through a list or series of operands stored in consecutive locations. When an instruction calls for mode 2, the address stored in the register is incremented each time the instruction is executed. It is incremented by one if you are using byte instructions, by two if you are using word instructions. However, R6 and R7 are always incremented by two.

To make it easy to remember that the register is incremented after use, the + sign follows the register name.

Autoincrement Mode Example

Symbolic	Instruction Octal Code	Description
CLR (R5)+	005025	Contents of R5 are used as the address of the operand. Clear selected operand and then increment the contents of R5 by two.

Represented as:



S

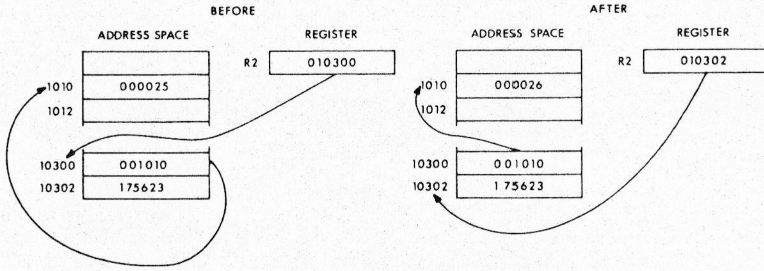
AUTOINCREMENT DEFERRED MODE — MODE 3 @(Rn)+

In autoincrement deferred mode, the register contains a pointer to the address of the operand. The + indicates that the pointer in Rn is incremented by two (for both word and byte operations) after the address is located. Mode 2, autoincrement, is used only to access operands that are stored in consecutive locations. Mode 3, autoincrement deferred, is used to access lists of operands stored anywhere in the system, i.e., the operands do not have to reside in adjoining locations. Mode 2 is used to step through a table of operands, and mode 3 is used to step through a table of addresses that point to data.

Autoincrement Deferred Example

Symbolic	Instruction Octal Code	Description
INC @(R2)+	005232	Contents of R2 are used as the address of the address of the operand. The operand is increased by one, contents of R2 are incremented by two.

Represented as:



AUTODECREMENT MODE — MODE 4 –(Rn)

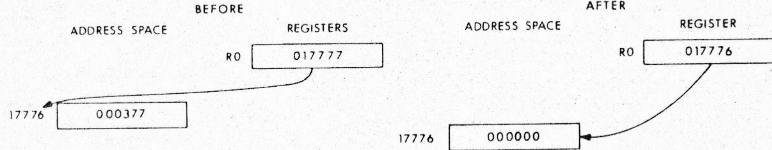
In autodecrement mode, the register contains an address that is automatically decremented; the decremented address is used to locate an operand. This mode is similar to autoincrement mode, but allows stepping through a list of words or bytes in reverse order. The address is decremented by one for bytes, by two for words. However, R6 and R7 are always decremented by two.

To remind you that the register is decremented prior to use, the – sign precedes the register name.

Autodecrement Mode Example

Symbolic	Instruction Octal Code	Description
INCB – (R0)	105240	The contents of R0 are decremented by one, then used as the address of the operand. The operand byte is increased by one.

Represented as:



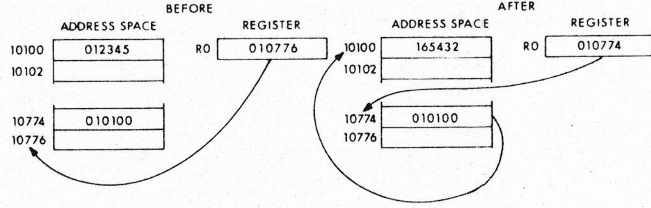
AUTODECREMENT DEFERRED MODE — MODE 5 @ – (Rn)

In autodecrement deferred mode, the register contains a pointer to the address of the operand. The pointer is first decremented by two (for both word and byte operations), then the new pointer is used to retrieve an address stored outside the CPU's general purpose registers. This mode is similar to autoincrement deferred, but allows stepping through a table of addresses in reverse order. Each address then redirects the CPU to an operand. Note that the operands do not have to reside in consecutive locations.

Autodecrement Deferred Mode Example

Symbolic	Instruction Octal Code	Description
COM @ – (R0)	005150	The contents of R0 are decremented by two and then used as the address of the address of the operand. The operand is one's complemented.

Represented as:



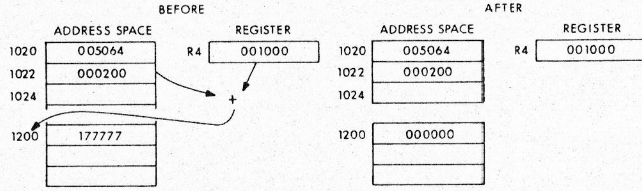
INDEX MODE — MODE 6 X(Rn)

In index mode, a base address is added to an index word to produce the effective address of an operand; the base address specifies the starting location of table or list. The index word then represents the address of an entry in the table or list relative to the starting (base) address. The base address may be stored in a register. In this case, the index word follows the current instruction. Or the locations of the base address and index word may be reversed (index word in the register, base address following the current instruction).

Index Mode Example

Symbolic	Instruction Octal Code	Description
CLR 200(R4)	005064 000200	The address of the operand is determined by adding 200 to the contents of R4. The resulting location is then cleared.

Represented as:



INDEX DEFERRED MODE — MODE 7 @X(Rn)

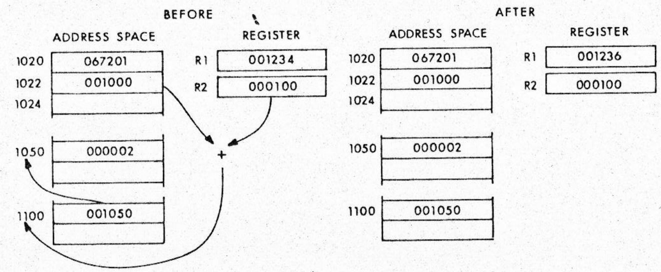
In index deferred mode, a base address is added to an index word. The result is a pointer to an address, rather than the actual address. This

mode is similar to mode 6, except that it produces a pointer to an address. The content of that address then redirects the CPU to the desired operand. Mode 7 provides for the random access of operands using a table of operand addresses.

Index Deferred Mode Example

Symbolic	Instruction Octal Code	Description
ADD @1000(R2),R1	067201 001000	1000 and the contents of R2 are summed to produce the address of the address of the source operand, the contents of which are added to the contents of R1. The result is stored in R1.

Represented as:



USE OF THE PC AS A GENERAL REGISTER

Register 7 is both a general purpose register and the program counter on the PDP-11. When the CPU uses the PC to access a word from memory, the PC is automatically incremented by two to contain the address of the next word of the instruction to be executed or the address of the next instruction to be executed. When the program uses the PC to access byte data, the PC is still incremented by two.

The PC can be used with all of the PDP-11 addressing modes. There are four modes in which the PC can provide advantages for handling position-independent code and for handling unstructured data. These modes refer to the PC and are termed immediate, absolute (or immediate deferred), relative, and relative deferred.

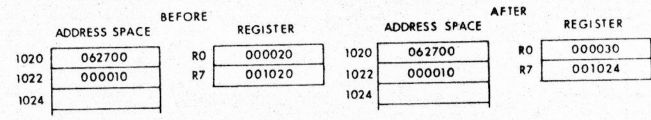
PC IMMEDIATE MODE — MODE 2 #n

Immediate mode is equivalent to using the autoincrement mode with the PC. It provides time improvements for accessing constant operands by including the constant in the memory location immediately following the instruction word.

PC Immediate Mode Example

Symbolic	Instruction Octal Code	Description
ADD #10,R0	062700 000010	The value 10 is located in the second word of the instruction and is added to the contents of R0. Just before this instruction is fetched and executed, the PC points to the first word of the instruction. The processor fetches the first word and increments the PC by two. The source operand mode is 27 (autoincrement the PC). Thus, the PC is used as a pointer to fetch the operand (the second word of the instruction) before being incremented by two to point to the next instruction.

Represented as:



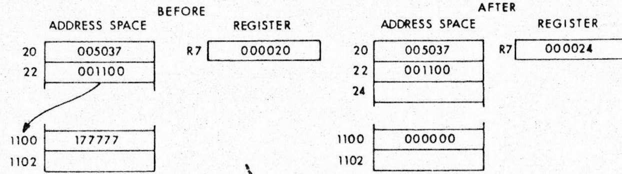
PC ABSOLUTE MODE — MODE 3 — @#A

This mode is the equivalent of immediate deferred or autoincrement deferred mode using the PC. The contents of the location following the instruction are taken as the address of the operand. Immediate data are interpreted as an absolute address (i.e., an address that remains constant no matter where in memory the assembled instruction is executed).

PC Absolute Mode Example

Symbolic	Instruction Octal Code	Description
CLR @#1100	005037 001100	Clears the contents of location 1100.

Represented as:



PC RELATIVE MODE — MODE 6 — X(PC) or A

This mode is index mode 6 using the PC. The operand's address is calculated by adding the word that follows the instruction (called an "offset") to the updated contents of the PC.

PC+2 directs the CPU to the offset that follows the instruction. PC+4 is summed with this offset to produce the effective address of the operand. PC+4 also represents the address of the next instruction in the program.

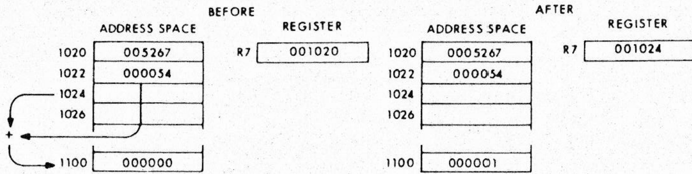
With the relative addressing mode, the address of the operand is always determined with respect to the updated PC. Therefore, if the entire program is relocated, the operand remains the same relative distance away and may be accessed with changing the instruction.

The distance between the updated PC and the operand is called an **offset**. After a program is assembled, this offset appears in the first word location that follows the instruction. This mode is useful for writing position-independent code. It is the default mode generated by the MACRO assembler.

PC Relative Mode Example

Symbolic	Instruction Octal Code	Description
INC A	005267 000054	To increment A, the contents of the memory location in the second word of the instruction are added to the updated PC to produce the address of A (1100). The contents of A are increased by one.

Represented as:



PC RELATIVE DEFERRED MODE — MODE 7 — @X(PC) or @A

This mode is index deferred (mode 7), using the PC. A pointer to an operand's address is calculated by adding an offset (which follows the instruction) to the updated PC.

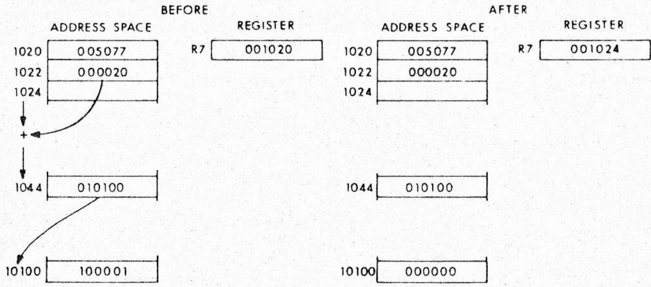
This mode is similar to the relative mode, except that it involves one additional level of addressing to obtain the operand. The sum of the offset and updated PC (PC+4) serves as a pointer to an address. When the address is retrieved, it can be used to locate the operand.

PC Relative Deferred Mode Example

Symbolic	Instruction Octal Code	Description
CLR @A	005077 000020	Adds the second word of the instruc-

Symbolic	Instruction Octal Code	Description
		tion to the updated PC to produce A—location 1044—the address of the address of the operand. Clears operand.

Represented as:



SUMMARY OF ADDRESSING MODES

Basic Addressing Modes

Binary Code	Mode	Name	Symbolic	Function
000	0	Register	Rn	Register contains operand.
010	2	Autoincrement	(Rn) +	Register is used as a pointer to sequential data, then incremented. R0-R5 are incremented by one for byte and two for word instruction. R6-R7 are always incremented by two.
100	4	Autodecrement	– (Rn)	Register is decremented and then used as a pointer to sequential data.

Basic Addressing Modes

Binary Code	Mode	Name	Symbolic	Function
				R0-R5 are decremented by one for byte and by two for word instructions. R6-R7 are always decremented by two.
110	6	Index	X(Rn)	Value X is added to Rn to produce address of operand. Neither X nor Rn is modified. X, the index value, is always found in the next memory location and increments the PC.

Indirect Addressing Modes

Binary Code	Mode	Name	Symbolic	Function
001	1	Register Deferred	@Rn or (Rn)	Register contains the address of the operand.
011	3	Autoincrement Deferred	@(Rn) +	Register is first used as a pointer to a word containing the address of the operand, then incremented (always by two, even for byte instructions).
101	5	Autodecrement Deferred	@ – (Rn)	Register is decremented (always by two, even for byte instructions) and then used as a pointer to a word containing the address of the operand.

Indirect Addressing Modes

Binary Code	Mode	Name	Symbolic	Function
111	7	Index Deferred	@X(Rn)	Value X (the index is always found in the next memory location and increments the PC by two) and Rn are added and the sum is used as a pointer to a word containing the address of the operand. Neither X nor Rn is modified.

When used with the PC, four of these modes are renamed, as you can see in the table below.

PC Register Addressing Modes

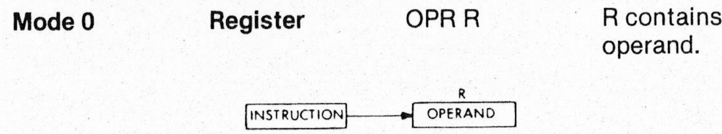
Binary Code	Mode	Name	Symbolic	Function
010	2	Immediate	#n	Operand is contained in the instruction.
011	3	Absolute	@#A	Absolute address is contained in the instruction.
110	6	Relative	A	Address of A, relative to the instruction, is contained in the instruction.
111	7	Relative Deferred	@A	Address of A, relative to the instruction, is contained in the instruction. Address of the operand is contained in A.

GRAPHIC SUMMARY OF PDP-11 ADDRESSING MODES

General Register Addressing Modes

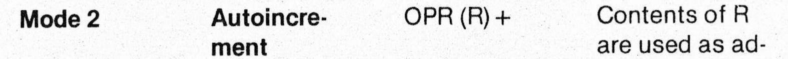
R is a general register, 0 to 7.
(R) is the contents of that register.

Mode 0 **Register** OPR R — R contains operand.

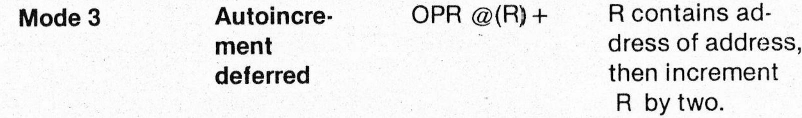


Mode 1 **Register deferred** OPR (R) — R contains address.

Mode 2 **Autoincrement** OPR (R)+ — Contents of R are used as address, then increment R. Note that R6 and R7 are always incremented by two.

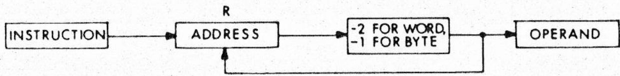


Mode 3 **Autoincrement deferred** OPR @(R)+ — R contains address of address, then increment R by two.



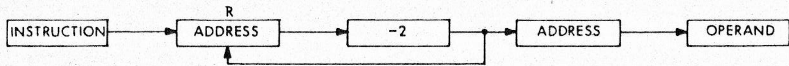
Mode 4 **Autodecrement** OPR −(R)

Decrement R, then R contains address. Note that R6 and R7 are always decremented by two.



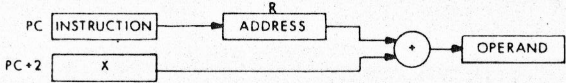
Mode 5 **Autodecrement deferred** OPR @−(R)

Decrement R by two, then R contains address of address.



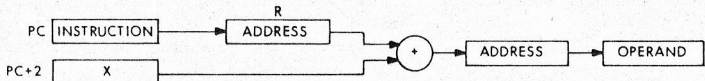
Mode 6 **Index** OPR X(R)

R + X is address. X is contained in the word following the instruction.



Mode 7 **Index deferred** OPR @X(R)

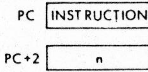
R + X is address of address. X is contained in the word following the instruction.



Program Counter Addressing Modes
Register = 7

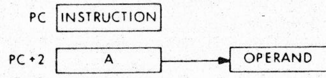
Mode 2 **Immediate** OPR #n

Literal operand n is contained in the word following the instruction.



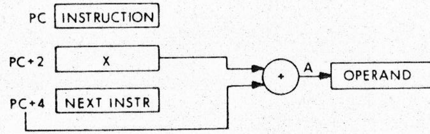
Mode 3 **Absolute** OPR @#A

Address A is contained in the word following the instruction.



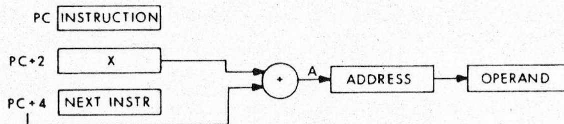
Mode 6 **Relative** OPR A

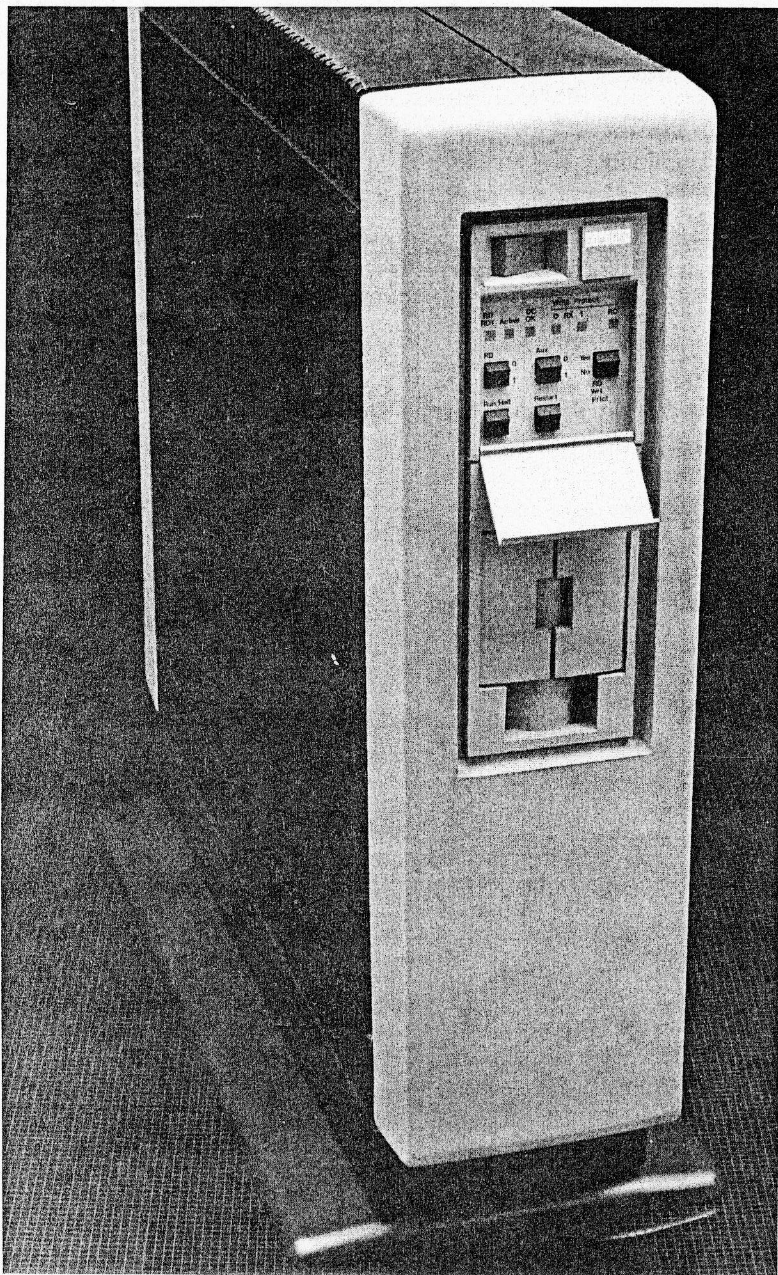
PC + 4 + X is address. PC + 4 is updated PC.



Mode 7 **Relative deferred** OPR @A

PC + 4 + X is address of address. PC + 4 is updated PC.





CHAPTER 5

INSTRUCTION SET

The PDP-11 instruction set offers a wide selection of operations and addressing modes. To save memory space and to simplify the implementation of control and communications applications, the PDP-11 instructions allow byte and word addressing in both single- and double-operand formats. By using the double-operand instructions, you can perform several operations with a single instruction. For example, ADD A,B adds the contents of location A to location B, storing the result in location B. Traditional computers would implement this instruction this way:

```
LDA A
ADD B
STR B
```

The PDP-11 instruction set also contains a full set of conditional branches that eliminate excessive use of jump instructions. PDP-11 instructions fall into one of seven categories:

- *Single-Operand*—the first part of the word, called the "opcode," specifies the operation; the second part provides information for locating the operand.
- *Double-Operand*—the first part of the word specifies the operation to be performed; the remaining two parts provide information for locating two operands.
- *Branch* — the first part of the word specifies the operation to be performed; the second part indicates where the action is to take place in the program.
- *Jump and Subroutine* — these instructions have an opcode and address part, and in the case of JSR, a register for linkage.
- *Trap* — these instructions contain an opcode only. In TRAP and EMT, the low-order byte may be used for function dispatching.
- *Miscellaneous* — HALT, WAIT, and Memory Management.
- *Condition Code* — these instructions set or clear the condition codes.

SINGLE-OPERAND INSTRUCTIONS

	Mnemonic	Instruction
General		
	CLR(B)	clear destination
	COM(B)	1's complement dst

	INC(B)	increment dst
	DEC(B)	decrement dst
	NEG(B)	2's complement negate dst
	NOP	no operation
	TST(B)	test dst
	TSTSET	test dst, set low bit (MICRO/J-11 only)
	WRTLCK	read/lock dst, write/unlock R0 into dst (MICRO/J-11 only)
Shift & Rotate		
	ASR(B)	arithmetic shift right
	ASL(B)	arithmetic shift left
	ROR(B)	rotate right
	ROL(B)	rotate left
	SWAB	swap bytes
Multiple Precision		
	ADC(B)	add carry
	SBC(B)	subtract carry
	SXT	sign extend

Instruction Format

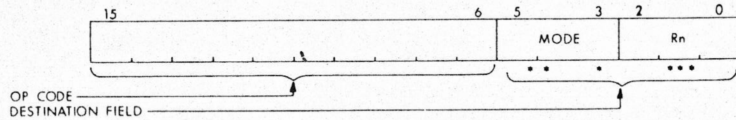


Figure 4-1 Single-Operand Instruction Format

The instruction format for single-operand instructions is:

- Bit 15 indicates word or byte operation.
- Bits 14-6 indicate the operation code, which specifies the operation to be performed.
- Bits 5-0 indicate the 3-bit addressing mode field and the 3-bit general register field. These two fields are referred to as the destination field.

DOUBLE-OPERAND INSTRUCTIONS

	Mnemonic	**Instruction**
General		
	MOV(B)	move source to destination
	ADD	add source to destination
	SUB	subtract source from destination

	CMP(B)	compare source to destination
	ASH	shift arithmetically
	ASHC	arithmetic shift combined
	MUL	multiply
	DIV	divide
Logical		
	BIT(B)	bit test
	BIC(B)	bit clear
	BIS(B)	bit set
	XOR	exclusive OR

Instruction Format

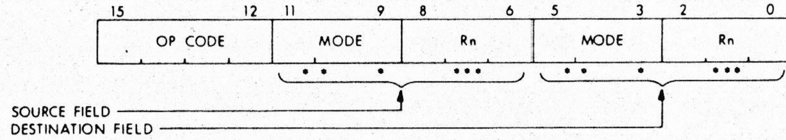


Figure 4-2 Double-Operand Instruction Format

The format of most double-operand instructions, though similar to that of single-operand instructions, has *two* fields for locating operands. One field is called the source field, the other is called the destination field. Each field is further divided into addressing mode and selected register. Each field is completely independent. The mode and register used by one field may be completely different than the mode and register used by another field.

- Bit 15 indicates word or byte operation *except* when used with opcode 6, in which case it indicates an ADD or SUBtract instruction.
- Bits 14-12 indicate the opcode, which specifies the operation to be done.
- Bits 11-6 indicate the 3-bit addressing mode field and the 3-bit general register field. These two fields are referred to as the **source** field.
- Bits 5-0 indicate the 3-bit addressing mode field and the 3-bit general register field. These two fields are referred to as the **destination** field.

- Some double-operand instructions (ASH, ASHC, MUL, DIV) must have the destination operand only in a register. Bits 15-9 specify the opcode. Bits 8-6 specify the destination register. Bits 5-0 contain the source field. XOR has a similar format, except that the source is in a register specified by bits 8-6, and the destination field is specified by bits 5-0.

Byte Instructions

Byte instructions are specified by setting bit 15. Thus, in the case of the MOV instruction, bit 15 is 0; when bit 15 is set, the mnemonic is MOVB. There are no byte operations for ADD and SUB, i.e., no ADDB or SUBB.

BRANCH INSTRUCTIONS

	Mnemonic	Instruction
Branch		
	BR	branch (unconditional)
	BNE	branch if not equal (to zero)
	BEQ	branch if equal (to zero)
	BPL	branch if plus
	BMI	branch if minus
	BVC	branch if overflow is clear
	BVS	branch if overflow is set
	BCC	branch if carry is clear
	BCS	branch if carry is set
Signed Conditional Branch		
	BGE	branch if greater than or equal (to zero)
	BLT	branch if less than (zero)
	BGT	branch if greater than (zero)
	BLE	branch if less than or equal (to zero)
	SOB	subtract one and branch (if not = 0)
Unsigned Conditional Branch		
	BHI	branch if higher
	BLOS	branch if lower or same
	BHIS	branch if higher or same
	BLO	branch if lower

Instruction Format

- The high byte (bits 15-8) of the instruction is an opcode specifying the conditions to be tested.
- The low byte (bits 7-0) of the instruction is the signed offset value in

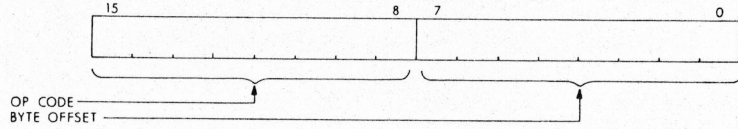


Figure 4-3 Branch Instruction Format

words that determines the new program location if the branch is taken. Thus, program control can be transferred within a range of −128 to +127 words from the updated PC.

JUMP AND SUBROUTINE INSTRUCTIONS

Mnemonic	Instruction
JMP	jump
JSR	jump to subroutine
RTS	return from subroutine
MARK	facilitates stack clean-up procedures

Instruction Format

JSR Format

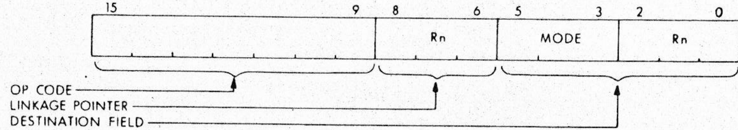


Figure 4-4 JSR Instruction Format

- Bits 15-9 are always octal 004, the opcode for JSR.
- Bits 8-6 specify the link register. Any general purpose register may be used in the link, except R6 (SP).
- Bits 5-0 designate the destination field that consists of addressing mode and general register fields. This specifies the starting address of the subroutine.
- Register R7 (the Program Counter) is frequently used for both the link and the destination. For example, you may use JSR R7, SUBR, which is coded 004767. R7 is the *only* register that can be used for both the link and destination, the other GPRs cannot. Thus, if the link is R5, any register except R5 can be used for one destination field.

RTS Format

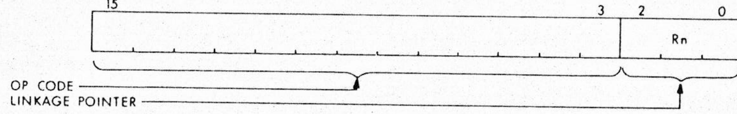


Figure 4-5 RTS Instruction Format

The RTS (return from subroutine) instruction uses the link to return control to the main program once the subroutine is finished.

- Bits 15-3 always contain octal 00020, which is the opcode for RTS.
- Bits 2-0 specify any one of the general purpose registers.
- The register specified by bits 2-0 must be the same register used as the link between the JSR causing the jump and the RTS returning control.

TRAPS AND INTERRUPTS

Mnemonic	Instruction
EMT	emulator trap
TRAP	trap
BPT	breakpoint trap
IOT	input/output trap
CSM	call to supervisor mode
RTI	return from interrupt
RTT	return from interrupt

The three ways to leave a main program are:

- *Software exit* — the program specifies a jump to some subroutine
- *Trap exit* — internal hardware on a special instruction forces a jump to an error handling routine
- *Interrupt exit* — external hardware forces a jump to an interrupt service routine

In each case, a jump to another program occurs. Once the latter program has been executed, control is returned to the proper point in the main program.

MISCELLANEOUS INSTRUCTIONS

Mnemonic	Instruction
HALT	halt
WAIT	wait for interrupt
RESET	reset UNIBUS
MTPD	move to previous data space
MTPI	move to previous instruction space
MFPD	move from previous data space
MFPI	move from previous instruction space
MTPS	move byte to processor status word
MFPS	move byte from processor status word
MFPT	move from processor type

Note that on the PDP-11/70, the four instructions for referencing the previous address space (MTPD, MTPI, MFPD, MFPI) use the General Register set indicated by PSW<11> when they are executed.

CONDITION CODE OPERATION

Mnemonic	Instruction
CLC, CLV, CLZ, CLN, CCC	clear
SEC, SEV, SEZ, SEN, SCC	set

The four condition code bits are:

- N, indicating a negative condition when set to 1
- Z, indicating a zero condition when set to 1
- V, indicating an overflow condition when set to 1
- C, indicating a carry condition when set to 1

These four bits are part of the processor status word (PS). The result of any single-operand or double-operand instruction affects one or more of the four condition code bits. A new set of condition codes is usually created after execution of each instruction. Some condition codes are not affected by the execution of certain instructions. The CPU may be asked to check the condition codes after execution of an instruction. The condition codes are used by the various instructions to check software conditions.

Z bit — Whenever the CPU sees that the result of an instruction is zero, it sets the Z bit. If the result is not zero, it clears the Z bit. There are a number of ways of obtaining a zero result:

- Adding two numbers equal in magnitude but different in sign
- Comparing two numbers of equal value
- Using the CLR or BIC instruction

N bit — The CPU looks only at the sign bit of the result. If the sign bit is set, indicating a negative value, the CPU sets the N bit. If the sign bit is clear, indicating a positive value, then the CPU clears the N bit.

C bit — The CPU sets the C bit automatically when the result of an instruction has caused a carry out of the most significant bit of the result. Otherwise, the C bit is cleared. During rotate instructions (ROL and ROR), the C bit forms a buffer between the most significant bit and the least significant bit of the word. A carry of 1 sets the C bit while a

carry of 0 clears the C bit. However, there are exceptions. For example:

- SUB and CMP set the C bit when there is no carry.
- INC and DEC do not affect the C bit.
- COM always sets the C bit, TST always clears the C bit.

V bit — The V bit is set to indicate that an overflow condition exists. An overflow means that the result of an instruction is too large to be placed in the destination. The hardware uses one of two methods to check for an overflow condition.

One way is for the CPU to test for a change of sign.

- When using single-operand instructions, such as INC, DEC, or NEG, a change of sign indicates an overflow condition.
- When using double-operand instructions, such as ADD, SUB, or CMP, in which both the source and destination have like signs, a change of sign in the result indicates an overflow condition.

Another method used by the CPU is to test the N bit and C bit when dealing with shift and rotate instructions.

- If only the N bit is set, an overflow exists.
- If only the C bit is set, an overflow exists.
- If *both* the N and C bits are set, there is no overflow condition.

More than one condition code can be set by a particular instruction. For example, both a carry and an overflow condition may exist after instruction execution.

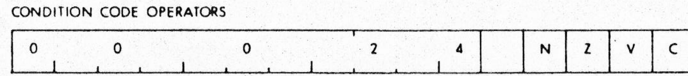


Figure 4-6 Condition Code Operators' Format

Instruction Format

The format of the condition code operators is:

- Bits 15-5 — the opcode
- Bit 4 — the "operator" which indicates set or clear with the values 1 and 0 respectively. If set, any selected bit is set; if clear, any selected bit is cleared.
- Bits 3-0 — the **mask** field. Each of these bits corresponds to one of the four condition code bits. When one of these bits is set, then the

corresponding condition code bit is set or cleared depending on the state of the "operator" (bit 4).

EXAMPLES

The following examples and explanations illustrate the use of the various types of instructions in a program.

Single-Operand Instruction Example

This routine uses a tally to control a loop, which clears out a specific block of memory. The routine has been set up to clear 30_8 byte locations beginning at memory address 600.

```
INIT:     MOV #600,R0
          MOV #30,R1

LOOP:     CLRB (R0)+
          DEC R1
          BNE LOOP
          HALT
```

Program Description

- The CLRB (R0)+ instruction clears the content of the location specified by R0 and increments R0.
- R0 is the pointer.
- Because the autoincrement addressing mode is used, the pointer automatically moves to the next memory location after execution of the CLRB instruction.
- Register R1 indicates the number of locations to be cleared and is, therefore, a counter. Counting is performed by the DEC R1 instruction. Each time a location is cleared, it is counted by decrementing R1.
- The Branch if Not Zero, BNE, instruction checks for done. If the counter is not zero, the program branches back to clear another location. If the counter is zero, indicating done, then the program halts.

Double-Operand Instruction Example

This routine moves characters to be printed from location 600 into a print buffer area in memory.

```
INIT:    MOV #600, R0          ;set up source address
         MOV #prtbuf, R1       ;set up destination address
         MOV #76, R2           ;set up loop count

START:   MOVB (R0)+, (R1)+     ;move one character
                               ;and increment
                               ;both source and
```

```
        .                   ;destination addresses
        DEC R2              ;decrement count by one
        BNE START           ;loop back if
        HALT                ;decremented counter is not
                            ;equal to zero
```

Program Description

- MOV is the instruction normally used to set up the initial conditions. Here, the first MOV places the starting address (600) into R0, which will be used as a *pointer*. The second MOV places the starting address of the print buffer into R1. The third MOV sets up R2 as a *counter* by loading the desired number of locations (76) to be printed.
- The MOVB instruction moves a byte of data to the printer buffer. The data come from the location specified by R0. The pointers R0 and R1 are then incremented to point to the next sequential location.
- The counter (R2) is then decremented to indicate one byte has been transferred.
- The program then checks the loops for done with the BNE instruction. If the counter has not reached zero, indicating more transfers must take place, then the BNE causes a branch back to START and the program continues.
- When the counter (R2) reaches zero, indicating all data have been transferred, the branch does not occur and the program halts.

Branch Instruction Example

NOTE

Branch instruction offsets are limited to the range of $+177_8$ to -200_8 words.

A payroll program has set up a series of words to identify each employee by his badge number. The high byte of the word contains the employee's badge number, the low byte contains an octal number ranging from 0 to 13 which represents his salary. These numbers represent steps within three wage classes to identify which employees are paid weekly, monthly, or quarterly. It is time to make out weekly paychecks. Unfortunately, employee information has been stored in a random order. The problem is to extract the names of only those employees who receive a weekly paycheck. Employee payroll numbers are assigned as follows: 0 to 3 — Wage Class I (weekly), 4 to 7 — Wage Class II (monthly), 10 to 13 — Wage Class III (quarterly).

600 is the starting address of memory block containing the employee payroll information. 1264 is the final address of this data area. The following program searches through the data area and finds all numbers representing Wage Class I, and, each time an appropriate number is found, stores the employee's badge number (just the high byte) on a Last-in/First-out stack which begins at location 4000.

```
INIT:     MOV #600, R0
          MOV #4000, R1

START:    CMPB(R0)+,#3


          BHI CONT

STACK:    MOVB (R0),-(R1)

CONT:     INC R0

          CMP #1264, R0

          BHIS START
```

Program Description

- R0 becomes the address pointer, R1 the stack pointer.
- Compare the contents of the first low byte with the number 3 and go to the first high byte.
- If the number is more than 3, branch to continue.
- If no branch occurs, it indicates that the number is 3 or less. Therefore, move the high byte containing the employee's number onto the stack as indicated by stack pointer R1.
- R0 is advanced to the next low byte.
- If the last address has not been examined (1264), this instruction produces a result equal to or greater than zero.
- If the result is equal to or greater than zero, examine the next memory location.

INSTRUCTION SET

The PDP-11 instruction set is presented in the following section. For ease of reference, the instructions are listed alphabetically.

SPECIAL SYMBOLS

You will find that a number of special symbols are used to describe

certain features of individual instructions. The commonly used symbols are explained below.

Symbol	Meaning
MN	Maintenance instruction
SO	Single-operand instruction
DO	Double-operand instruction
PC	Program control instruction
MS	Miscellaneous instruction
CC	Condition Code
(x)	Contents of memory location whose address is x
src	Source address
dst	Destination address
tmp	Contents of temporary internal register
←	Becomes, or moves into. For example, (dst) ← (src) means that the source becomes the destination or that the source moves into the destination location.
(SP)+	Popped or removed from the hardware stack
−(SP)	Pushed or added to the hardware stack
$\wedge$	Logical AND
v	Logical inclusive OR (either one or both)
$\forall$	Logical exclusive OR (either one, but not both)
~	Logical NOT
Reg or R	Contents of register
Rv1	Contents of register R if an odd-numbered register is specified. Contents of the register following R if R is an even-numbered register
R, Rv1	32-bit quantity obtained by concatenating R and Rv1
B	Byte
M.P.I.	Most Positive Integer—077777 (word) or 177 (byte)
M.N.I.	Most Negative Integer—100000 (word) or 200 (byte)

NOTE

Condition code bits are considered to be cleared unless they are specifically listed as set.

SUMMARY OF PDP-11 INSTRUCTION SET

Basic PDP-11 Instruction Set

ADC	BIT	COM	ROL
ADCB	BITB	COMB	ROLB
ADD	BLE	DEC	ROR
ASL	BLO	DECB	RORB
ASLB	BLOS	EMT	RTI
ASR	BLT	HALT	RTS
ASRB	BMI	INC	RTT
BCC	BNE	INCB	SBC
BCS	BPL	IOT	SBCB
BEQ	BPT	JMP	SCC, SEN, SEZ, SEV, SEC
BGE	BR	JSR	SOB
BGT	BVC	MARK	SUB
BHI	BVS	MOV	SXT
BHIS	CLR	MOVB	SWAB
BIC	CLRB	NEG	TRAP
BICB	CCC, CLN, CLZ, CLV, CLC	NEGBB	TST
BIS	CMP	NOP	TSTB
BISB	CMPB	RESET	XOR
			WAIT

The basic PDP-11 instructions are standard on:

- MICRO/T-11
- MICRO/J-11
- LSI-11/2
- FALCON SBC-11/21 (except for MARK instruction)
- MICRO/PDP-11
- PDP-11/23 PLUS
- PDP-11/24
- PDP-11/44

The PDP-11 compatibility mode on VAX-11 implements all basic PDP-11 instructions except: MARK, RESET, TRAP, WAIT, BPT, EMT, IOT, and HALT.

CSM
Available on MICRO/J-11 and PDP-11/44 only.

Extended Integer Instructions (EIS)
ASH
ASHC
DIV
MUL

EIS is standard on:
- MICRO/PDP-11
- PDP-11/23 PLUS
- PDP-11/24
- PDP-11/44
- VAX-11 compatibility mode

EIS is also available as an option on the LSI-11/2.

MFPD, MFPI, MTPD, MTPI
Available on the MICRO/J-11, LSI-11/23, MICRO/PDP-11, PDP-11/23-PLUS, PDP-11/24, PDP-11/44, and VAX-11 compatibility mode.

MFPS, MTPS
Available on the MICRO/T-11, MICRO/J-11, LSI-11/2, FALCON SBC-11/21, LSI-11/23, MICRO/PDP-11, PDP-11/23-PLUS, and PDP-11/24.

MFPT
Available on the MICRO/T-11, MICRO/J-11, FALCON SBC-11/21, LSI-11/23, MICRO/PDP-11, PDP-11/23-PLUS, PDP-11/24, and PDP-11/44.

SPL
Available on MICRO/J-11 and PDP-11/44 only.

TSTSET, WRTLCK
Available on MICRO/J-11 only.

Table 5-1 PDP-11 Instruction Set

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
ADC ADCB Add Carry	SO	0055DD 1055DD	(dst) ← (dst)+C	N: set if result < 0 Z: set if result = 0 V: set if (dst) was M.P.I. and C was 1, prior to instruction execution. C: set if (dst) was −1 and C was 1, prior to instruction execution.	Adds the contents of the C bit into the destination.
ADD Add	DO	06SSDD	(dst) ← (src) + (dst)	N: set if result < 0 Z: set if result = 0 V: set if there is arithmetic overflow as a result of the operation; that is, both operands were of the same sign and the result is of the opposite sign.	Adds the source operand to the destination operand and stores the result at the destination address. The original contents of the destination are lost. The contents of the source are not affected. 2's complement addition is performed.

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
				C: set if there is a carry from the most significant bit of the result.	
ASH Arithmetic Shift	DO	072RSS	R ← R shifted arithmetically NN places to right or left where NN = (src) <5:0>	N: set if result < 0 Z: set if result = 0 V: set if sign of register changed during shift. Cleared if NN = 0. C: loaded from last bit shifted out of register. Cleared if NN = 0.	The contents of the register are shifted right or left the number of times specified by the shift count (i.e., bits <5:0> of the source operand). The shift count is taken as the low order 6 bits of the source operand. This number ranges from −32 to +31. Negative is a right shift and positive is a left shift.
ASHC Arithmetic Shift Combined	DO	073RSS	tmp ← R, Rv1 tmp ← tmp shifted NN bits R ← tmp<31:16> Rv1 ← tmp<15:0> The double word R,Rv1 is shifted NN places to the right or left, where NN = (src) <5:0>	N: set if result < 0 Z: set if result = 0 V: set if sign bit changes during the shift. C: loaded with high-order bit when left shift; loaded with low-order bit when right shift (loaded with the last bit shifted out of the 32-bit operand).	The contents of the specified register R and the register Rv1 are treated as a single 32-bit operand, and are shifted by the number of bits specified by the count field (bits <5:0> of the source operand). The registers are replaced by the result. First, bits <31:16> of the result are stored in register R. Then, bits <15:0> of the result are stored in register Rv1. The count ranges from −32 to +31. A negative count signifies a right shift. A positive count signifies a left shift. A zero count implies no shift, but condition codes are affected. Condition codes are always set on the 32-bit result. **Note:** 1) The sign bit of the register R is replicated in shifts to the right. The least significant bit is filled with zero in shifts to the left. The C bit stores the last bit shifted out. 2) Integer overflow occurs on a left shift if any bit shifted into the sign position differs from the initial sign of the register.
ASL ASLB Arithmetic	SO SO	0063DD 1063DD	(dst) ← (dst) shifted one place to the left	N: set if high-order bit of the result is set (result < 0)	Shifts all bits of the destination left one place. The low-order bit is loaded with a 0. The C bit of the

Table 5-1 PDP-11 Instruction Set, continued

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
Shift Left				Z: set if the result = 0 V: loaded with the exclusive OR of the N bit and C bit (as set by the completion of the shift operation). C: loaded with the high-order bit of the destination.	status word is loaded from the high-order bit of the destination. ASL performs a signed multiplication of the destination by 2 with overflow indication. For example, −1 shifted left yields −2, +2 shifted left yields +4, and −3 shifted left yields −6.
ASR ASRB Arithmetic Shift Right	SO SO	0062DD 1062DD	(dst) ← (dst) shifted one place to the right	N: set if the high-order bit of the result is set (result < 0) Z: set if the result = 0 V: loaded from the exclusive OR of the N bit and C bit (as set by the completion of the shift operation). C: loaded from low-order bit of the destination	Shifts all bits of the destination right one place. The high-order bit is replicated. The C bit is loaded from the low-order bit of the destination. ASR performs signed division of the destination by 2, rounded to minus infinity. −1 shifted right remains −1, +5 shifted right yields +2, −5 shifted right yields −3.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
BCC Branch if Carry Clear	PC	103000 PLUS 8-bit offset	PC ← PC + (2 × offset) if C = 0	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the C bit and causes a branch if C is clear.
BCS Branch if Carry Set	PC	103400 PLUS 8-bit offset	PC ← PC + (2 × offset) if C = 1	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the C bit and causes a branch if C is set. Used to test for a carry in the result of a previous operation.
BEQ Branch if Equal (to zero)	PC	001400 PLUS 8-bit offset	PC ← PC + (2 × offset) if Z = 1	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the Z bit and causes a branch if Z is set. For example, it is used to test equality following a CMP operation, and to test that no bits set in the destination were also set in the source following a BIT operation, and, generally, to test that the result of the previous operation was 0.
BGE Branch if Greater than or Equal	PC	002000 PLUS 8-bit offset	PC ← PC + (2 × offset) if $N \forall V = 0$	N: unaffected Z: unaffected V: unaffected C: unaffected	Causes a branch if N and V are either both clear or both set. BGE is the complementary operation to BLT. Thus, BGE always causes a branch when it follows an operation that caused addition of two positive numbers. BGE also causes a branch in a 0 result.

Table 5-1 PDP-11 Instruction Set, continued

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
BGT Branch if Greater than	PC	003000 PLUS 8-bit offset	PC ← PC + (2 × offset) if Zv(N v V) = 0	N: unaffected Z: unaffected V: unaffected C: unaffected	Causes a branch if Z is clear and N equals V. Thus, BGT never branches following an operation that added two negative numbers, even if overflow occurred. In particular, BGT never causes a branch if it follows a CMP instruction operating on a negative source and a positive destination (even if overflow occurred). Further, BGT always causes a branch when it follows a CMP instruction operating on a positive source and negative destination. BGT does not cause a branch if the result of the previous operation was 0 (without overflow).
BHI Branch if Higher	PC	101000 PLUS 8-bit offset	PC ← PC + (2 × offset) if C = 0 and Z = 0	N: unaffected Z: unaffected V: unaffected C: unaffected	Causes a branch if the previous operation causes neither a carry nor a 0 result. This will happen in comparision (CMP) operations as long as the source has a higher unsigned value than the destination.
BHIS Branch if Higher than or Same	PC	103000 PLUS 8-bit offset	PC ← PC + (2 × offset) if C = 0	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the C bit and causes a branch if C is cleared.
BIC BICB Bit Clear	DO	04SSDD 14SSDD	(dst) ← ~ (src) Λ (dst)	N: set if high-order bit of result set Z: set if result = 0 V: cleared C: unaffected	Clears each bit in the destination that corresponds to a set bit in the source. The original contents of the destination are lost. The contents of the source are unaffected.
BIS BISB Bit Set	DO	05SSDD 15SSDD	(dst) ← (src)v(dst)	N: set if high-order bit of result set Z: set if result = 0 V: cleared C: unaffected	Performs inclusive OR operation between the source and destination operands and leaves the result at the destination address, i.e., corresponding bits set in the source are set in the destination. The contents of the destination are lost.
BIT BITB Bit Test	DO	03SSDD 13SSDD	(dst) Λ (src)	N: set if high-order bit of result set Z: set if result = 0 V: cleared	Performs logical AND comparison of the source and destination operands and modifies Condition Codes accordingly. Neither the

Table 5-1 PDP-11 Instruction Set, continued

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
				C: unaffected	source nor destination operands are affected. The BIT instruction may be used to test whether any of the corresponding bits that are set in the destination are clear in the source.
BLE Branch if Less than or Equal to	PC	003400 PLUS 8-bit offset	PC ← PC + (2 × offset) if Zv(N v V) = 1	N: unaffected Z: unaffected V: unaffected C: unaffected	Causes a branch if Z is set or if N does not equal V. Thus, BLE always branches following an operation that added two negative numbers, even if overflow occurred. In particular, BLE always causes a branch if it follows a CMP instruction operating on a negative source and a positive destination (even if overflow occurred). Further, BLE never causes a branch when it follows a CMP instruction operating on a positive source and negative destination. BLE always causes a

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
					branch if the result of the previous operation was 0.
BLO Branch if Lower	PC	103400 PLUS 8-bit offset	PC ← PC + (2 × offset) if C = 1	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the C bit and causes a branch if C is set. Used to test for a carry in the result of a previous operation.
BLOS Branch if Lower or Same	PC	101400 PLUS 8-bit offset	PC ← PC + (2 × offset) if CvZ = 1	N: unaffected Z: unaffected V: unaffected C: unaffected	Causes a branch if the previous operation caused either a carry or a zero result. BLOS is the complementary operation to BHI. The branch occurs in comparison operations as long as the source is equal to or has a lower unsigned value than the destination.
BLT Branch if Less Than	PC	002400 PLUS 8-bit offset	PC ← PC + (2 × offset) if $N \veebar V = 1$	N: unaffected Z: unaffected V: unaffected C: unaffected	Causes a branch if the exclusive OR of the N and V bits is 1. Thus, BLT always branches following an operation that added two negative numbers, even if overflow occurred. In particular, BLT always causes a branch if it follows a CMP instruction operating on a negative source and a positive destination (even if overflow occurred). Fur-

Table 5-1 PDP-11 Instruction Set, continued

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
					ther, BLT never causes a branch when if follows a CMP instruction operating on a positive source and negative destination. BLT does not cause a branch if the result of the previous operation was 0 (without overflow).
BMI Branch if Minus	PC	100400 PLUS 8-bit offset	PC ← PC + (2 × offset) if N = 1	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the N bit and causes a branch if N is set. Used to test the sign (most significant bit) of the result of the previous operation.
BNE Branch if Not Equal	PC	001000 PLUS 8-bit offset	PC ← PC + (2 × offset) if Z = 0	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the Z bit and causes a branch if the Z bit is clear. BNE is the complementary operation to BEQ. It is used to test inequality following a CMP, to test that some bits set in the destination were also set in the source, following a BIT, and generally, to test that the result of the previous operation was not 0.
BPL Branch if Plus	PC	100000 PLUS 8-bit offset	PC ← PC + (2 × offset) if N = 0	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the N bit and causes a branch if N is clear. BPL is the complementary operation of BMI.
BPT Breakpoint Trap	PC	000003	−(SP) ← PS −(SP) ← PC PC ← (14) PS ← (16)	N: loaded from trap vector Z: loaded from trap vector V: loaded from trap vector C: loaded from trap vector	Performs a trap sequence with a trap vector address of 14. Used to call debugging aids. The user is cautioned against employing code 000003 in programs run under these debugging aids. No information is transmitted in the low byte.
BR Branch (Unconditional)	PC	000400 PLUS 8-bit offset	PC ← PC + (2 × offset)	N: unaffected Z: unaffected V: unaffected C: unaffected	Provides a way of transferring program control within a range of −128 to +127 words with a one-word instruction. An unconditional branch.
BVC Branch if V bit Clear	PC	102000 PLUS 8-bit offset	PC ← PC + (2 × offset) if V = 0	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the V bit and causes a branch if the V bit is clear. BVC is the complementary operation to BVS.

Table 5-1 PDP-11 Instruction Set, continued

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
BVS Branch if V bit Set	PC	102400 PLUS 8-bit offset	PC ← PC + (2 × offset) if V = 1	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of V bit and causes a branch if the V bit is set. BVS is used to detect arithmetic overflow in the previous operation.
CLR CLRB Clear	SO	0050DD 1050DD	(dst) ← 0	N: cleared Z: set V: cleared C: cleared	Contents of specified destination are replaced with zeros.
C Clear Selected Condition Code Bits	CC	000240 PLUS 4-bit mask	Clear condition code bits. Selectable combinations of these bits may be cleared together. Condition code bits corresponding to bits in the condition code operator (bits 0-3) are modified. Clears the bit specified by the mask; i.e., bit 0, 1, 2, or 3. Bit 4 is a 0. **Operation:** PSW <3:0> ← PSW <3:0> Λ[~mask <3:0>]		
CCC Clear all Condition Code Bits	CC	00257	N, Z, V, C ← 0		
CLC Clear C	CC	000241	C ← 0		
CLN Clear N	CC	000250	N ← 0		
CLV Clear V	CC	000242	V ← 0		
CLZ Clear Z	CC	000244	Z ← 0		
CMP CMPB Compare	DO	02SSDD 12SSDD	(src) − (dst) [in detail (src) + ~ (dst) + 1]	N: set if result < 0 Z: set if result = 0 V: set if there is arithmetic overflow; i.e., operands of opposite signs and the sign of the destination is the same as the sign of the result. C: set if there is a bor-	Compares the source and destination operands and sets the condition codes, which may then be used for arithmetic and logical conditional branches. Both operands are unaffected. The only action is to set the condition codes. The comparison is customarily followed by a conditional branch instruction. Note that unlike the subtract instruction, the order of

Table 5-1 PDP-11 Instruction Set, continued

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
				row into the most significant bit, i.e., if (src)+~(dst)+1 was less than 2^{16}.	operation is (src) −(dst), not (dst) − (src).
COM COMB Complement	SO	0051DD 1051DD	(dst) ← ~ (dst)	N: set if most significant bit of result = 1 Z: set if result = 0 V: cleared C: set	Replaces the contents of the destination address by their logical complements (each bit equal to 0 set and each bit equal to 1 cleared).

CSM
Call to Supervisor Mode
	PC	0070DD	If MMR 3<3> = 1 **and** current mode ≠ Kernel then:
			begin
			Supervisor SP ← current mode SP;
			temp <15:4> ← PSW <15:4>;
			temp <3:0> ← 0;
			PSW <13:12> ← PSW <15:14>;
			PSW <15:14> ← 01;
			PSW <4> ← 0;
			−(SP) ← temp;
			−(SP) ← PC;
			−(SP) ← (dst);
			PC ← (10);
			end;
			else trap to 10 in Kernel mode;
				N: unaffected
				Z: unaffected
				V: unaffected
				C: unaffected
					CSM may be executed in User or Supervisor Mode, but is an illegal instruction in Kernel mode. CSM copies the current stack pointer (SP) to the Supervisor Mode switches to Supervisor Mode, stacks three words on the Supervisor stack, (the PSW with the Condition Codes cleared, the PC, and the argument word addressed by the operand), and sets the PC to the contents of location 10 (in Supervisor space). The called program in Supervisor space may return to the calling program by popping the argument word from the stack and executing RTI. On return, the Condition Codes are determined by the PSW word on the stack. Hence, the called program in Supervisor space may control the Condition Code values following return.

DEC DECB Decrement	SO	0053DD 1053DD	(dst) ← (dst) − 1	N: set if result < 0 Z: set if result = 0 V: set if (dst) was M.N.I. C: unaffected	Subtracts 1 from the contents of the destination.
DIV Divide	DO	071RSS	R,Rv1 ← R,Rv1/(src)	N: set if quotient < 0 (unspecified if V = 1) Z: set if quotient = 0 (unspecified if V = 1)	

Table 5-1 PDP-11 Instruction Set, continued

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
				V: set if (src) = 0 or if quotient cannot be represented as a 16-bit 2's complement number. R, Rv1 are unpredictable if V is set and C is clear. C: set if divide by 0 is attempted	
EMT Emulator Trap	PC	104000 to 104377	−(SP) ← PS −(SP) ← PC PC ← (30) PS ← (32)	N: loaded from trap vector Z: loaded from trap vector V: loaded from trap vector C: loaded from trap vector	All operation codes from 104000 to 104377 are EMT instructions and may be used to transmit information to the emulating routine (e.g., function to be performed). The trap vector for EMT is at address 30. The new PC is taken from the word at address 30; the new central processor status word (PS) is taken from the word at address 32.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
					Caution: EMT is used frequently by DIGITAL system software and is therefore not recommended for general use.
HALT	MS	000000		N: unaffected Z: unaffected V: unaffected C: unaffected	Causes the processor operation to cease. The console is given control of the processor. The console data lights display the contents of the PC (which is the address of the HALT instruction plus 2). Transfers on the UNIBUS are terminated immediately. Pressing the continue key on the console causes processor operation to resume.
INC INCB Increment	SO	0052DD 1052DD	(dst) ← (dst) + 1	N: set if result < 0 Z: set if result = 0 V: set if (dst) was M.P.I. C: unaffected	Adds 1 to the contents of the destination.
IOT I/O Trap	PC	000004	−(SP) ← PS −(SP) ← PC PC ← (20) PS ← (22)	N: loaded from trap vector Z: loaded from trap vector V: loaded from trap vector	Performs a trap sequence with a trap vector address of 20. Used to call the I/O executive routine IOX in the paper tape software system and for error reporting in the disk operating system. No information

Table 5-1 PDP-11 Instruction Set, continued

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
				C: loaded from trap vector	is transmitted in the low byte.
JMP Jump	PC	0001DD	PC ← dst	N: unaffected Z: unaffected V: unaffected C: unaffected	JMP provides more flexible program branching than provided with the branch instruction. It is not limited to $+177_8$ and -200_8 as are branch instructions. JMP does generate a second word, which makes it slower than branch instructions. Control may be transferred to any location in memory (no range limitation) and can be accomplished with the full flexibility of the addressing modes with the exception of register mode 0. Execution of a jump with mode 0 will cause an illegal instruction condition and a trap through location 4. (Program control cannot be transferred to a register.) Register

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
					deferred mode is legal and will cause program control to be transferred to the address held in the specified register. **Note that instructions are word data and therefore must be fetched from an even numbered address. A boundary error trap condition will result when the processor attempts to fetch an instruction from an odd address.**
JSR Jump to Subroutine	PC	004RDD	(tmp) ← (dst) (tmp is an internal processor register) ↓(SP) ← reg (push reg contents onto processor stack) reg ← PC (PC holds the location following JSR; this address now put in reg) PC ← tmp (PC now points to	N: unaffected Z: unaffected V: unaffected C: unaffected	In execution of the JSR, the old contents of the specified register (the linkage pointer) are automatically pushed onto the R6 stack and new linkage information is placed in the register. Thus, subroutines nested within subroutines to any depth may all be called with the same linkage register. There is no need either to plan the maximum depth at which any particular subroutine will be called or to include instructions in each routine to save and restore the linkage pointer. Further, since all linkages are

Table 5-1 PDP-11 Instruction Set, continued

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
			subroutine address)		saved in a re-entrant manner on the R6 stack, execution of a subroutine may be interrupted, and the same subroutine re-entered and executed by an interrupt service routine. Execution of the initial subroutine can then be resumed when other requests are satisfied. This process (called nesting) can proceed to any level. JSR PC, dst is a special case of the PDP-11 subroutine call suitable for subroutine calls that transmit parameters through the general purpose registers. JSR, with the PC as the linkage register, saves the use of an extra register. **Note:** If the register specified in the first operand register is autoincremented or autodecremented in the second operand (dst) evaluation,

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
					the modified register contact is pushed on SP. For example, JSR R5,@(R5)+ will cause the modified value of R5 to be pushed to SP.
MARK	PC	0064NN	SP ← PC + 2 × NN PC ← R5 R5 ← (SP)+ NN = number of parameters	N: unaffected Z: unaffected V: unaffected C: unaffected	Used as part of the standard PDP-11 subroutine return convention. MARK facilitates the stack clean-up procedures involved in subroutine exit. Assembler format is: MARK N
MFPD Move From Previous Data space MFPI Move From Previous Instruction space	MS	1065SS 0065SS	tmp ← (src) −(SP) ← tmp	N: set if the source < 0 Z: set if the source = 0 V: cleared C: unaffected	Pushes a word onto the current R6 stack from an address in previous space determined by PS<13:12>. The source address is computed using the current registers and memory map. When MFPI is executed and both previous mode and current mode are User, the instruction functions as though it were MFPD.

Table 5-1 PDP-11 Instruction Set, continued

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
PDP-11/03, and PDP-11/04)					
MFPS Move Byte from PSW	MS	1067DD	(dst) ← PS<7:0> dst lower 8 bits	N: set if PS bit 7 = 1 Z: set if PS <7:0> = 0 V: cleared C: not affected	The 8-bit contents of the PS are moved to the effective destination. If destination is mode 0, PS bit 7 is sign extended through the upper byte of the register. The destination operand is treated as a byte address.
MFPT Move From Processor	MS	000007	R0<7:0> ← processor model code R0<15:8> ← processor sub-code	N: unaffected Z: unaffected V: unaffected C: unaffected	No source operands are used. The MFPT instructions returns in the low byte of R0 a processor model code (1 on the PDP-11/44, 3 on the PDP-11/24). The high byte of R0 is loaded with a processor-specific subcode, (currently 0 on the PDP-11/24 and PDP-11/44). The condition codes are not affected. The previous contents of R0 are lost. **Note:** On processors where this instruction is not implemented, a reserved instruction trap through vector 10_8 is taken.
MOV MOVB Move	DO	01SSDD 11SSDD	(dst) ← (src)	N: set if (src) < 0 Z: set if (src) = 0 V: cleared C: unaffected	Moves the source operand to the destination location. The previous contents of the destination are lost. The source operand is not affected. Byte: Same as MOV. The MOVB to a register (unique among byte instructions) extends the most significant bit of the low-order byte (sign extension) into the high byte of the selected register. Otherwise, MOVB operates on bytes exactly as MOV operates on words.
MTPD Move To	MS	1066DD 0066DD	tmp ← SP+ (dst) ← tmp	N: set if the source < 0 Z: set if the source = 0	This instruction pops a word off the current R6 stack determined by PS

Table 5-1 PDP-11 Instruction Set, continued

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
Previous Data space MTPI Move To Previous Instruction space				V: cleared C: unaffected	bits <15:14> and stores that word into an address in previous space determined by PS bits <13:12>. The destination address is computed using the current registers and memory map.
MTPS Move Byte To PSW	MS	1064SS	PS ← (src)	N: set according to effective src operand 0-3 Z: same as above V: same as above	The eight bits of the effective operand replace the current contents of the PS <7:0>. The source operand address is treated as a byte address. Note that PS bit 4 cannot be

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
				C: same as above	set with this instruction. The src operand remains unchanged.
MUL Multipy	DO	070RSS	R,Rv1 ← R × (src)	N: set if product < 0 Z: set if product = 0 V: cleared C: set if the result is less than -2^{15} or greater than or equal to 2^{15}. Condition codes set on 32-bit result even if R is odd.	The contents of the destination register and source taken as 2's complement integers are multiplied and stored in the destination register and the succeeding register (if R is even). If R is odd, only the low-order product is stored. Assembler syntax is: MUL S,R. (Note that the actual destination is R, Rv1, which reduces to just R when R is odd.)
NEG NEGB Negate	SO	0054DD 1054DD	(dst) ← −(dst)	N: set if result < 0 Z: set if result = 0 V: set if result = M.N.I. C: cleared if result = 0; set otherwise	Replaces the contents of the destination address by its 2's complement. Note that 100000 is replaced by itself.
NOP No Operation	CC	000240 000260	None	N: unaffected Z: unaffected V: unaffected	No operation is performed.

Table 5-1 PDP-11 Instruction Set, continued

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
				C: unaffected	
RESET	MS	000005		N: unaffected Z: unaffected V: unaffected C: unaffected	Sends INIT on the UNIBUS for 10 ms. All devices on the unit are reset to their state at power-up.
ROL ROLB Rotate Left	SO	0061DD 1061DD	(dst) ← (dst) rotate left one place	N: set if the high-order bit of the result word is set (result < 0). Z: set if all bits of the result = 0 V: loaded with the exclusive OR of the N bit and C bit (as set by the completion of the rotate operation). C: set if the high-order bit of the destination was set prior to instruction execution.	Rotates all bits of the destination left one place. The high-order bit is loaded into the C bit of the status word and the previous contents of the C bit are loaded into the low-order bit of the destination.
ROR RORB Rotate Right	SO	0060DD 1060DD	(dst) ← (dst) rotate right one place	N: set if high-order bit of the result is set Z: set if all bits of result are 0 V: loaded with the exclusive OR of the N bit and the C bit as set by ROR. C: set if the low-order bit of the destination was set prior to instruction execution.	Rotates all bits of the destination right one place. The low-order bit is loaded into the C bit and the previous contents of the C bit are loaded into the high-order bit of the destination.
RTI Return from Interrupt	MS	000002	PC ← (SP)+ PS ← (SP)+	N: loaded from current R6 stack Z: loaded from current R6 stack V: loaded from current R6 stack C: loaded from current R6 stack	Used to exit from an interrupt or trap service routine. The PC and PS are restored (popped) from the R6 stack. If the RTI sets the T bit in the PS, a trace trap will occur prior to executing the next instruction. When executed in Supervisor Mode, the current and previous

Table 5-1 PDP-11 Instruction Set, continued

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
					mode bits in the restored PS cannot be Kernel. When executed in User mode, the current and previous mode bits in the restored PS can only be User. RTI cannot clear PS <11> if it was already set. When executed in user or supervisor mode, PS <7:5> are unaffected.
RTS Return from Subroutine	PC	00020R	PC ← (reg) (reg) ← (SP)+	N: unaffected Z: unaffected V: unaffected C: unaffected	Loads contents of register into PC and pops the top element of the R6 stack into the specified register. Return from a non-re-entrant subroutine is made through the same register that was used in its call. Thus, a subroutine called with a JSR PC,dst exits with an RTS PC, and a subroutine called with a JSR R5,dst may pick up parameters with addressing modes (R5)+, X(R5), or @X(R5) and finally exit, with an RTS R5.
RTT Return from Interrupt	MS	000006	PC ← (SP)+ PS ← (SP)+	N: loaded from current R6 stack Z: loaded from current R6 stack V: loaded from current R6 stack C: loaded from current R6 stack	This is the same as the RTI instruction (used to exit from an interrupt or trap service routine), the PC and PS are restored (popped) from the processor stack; if the RTI sets the T bit in the PS, a trace trap will occur prior to executing the next instruction) except that it inhibits a trace trap, while RTI permits a trace trap. If a trace trap is pending, the first instruction after the RTT will be executed prior to the next "T" trap. In the case of the RTI instruction, the "T" trap will occur immediately after the RTI. When executed in Supervisor Mode, the current and previous mode bits in the restored PS cannot be Kernel. When executed in User Mode, the current and previous mode bits in the restored PS can only be User.

Table 5-1 PDP-11 Instruction Set, continued

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
					RTT cannot clear PS<11> if it was already set. When executed in user or supervisor mode, PS <7:5> are unaffected.
SBC SBCB Subtract Carry	SO	0056DD 1056DD	(dst) ← (dst) − C	N: set if result < 0 Z: set if result = 0 V: set if (dst) = M.N.I. C: set if (dst) was 0 and C was 1 prior to instruction execution.	Subtracts the contents of the C bit from the destination.
S Set Selected Condition Codes	CC	000260 PLUS 4-bit mask	Set condition code bits. Selectable combinations of these bits may be set together. Condition code bits corresponding to bits in the condition code operator (bits 0-3) are modified; sets the bit specified by bit 0, 1, 2, or 3. Bit 4 is a 1. **Operation:** PSW <3:0> ← PSW <3:0> v mask <3:0>		
SCC Set all Condition Codes	CC	000277	N, Z, V, C ← 1		
SEC Set C	CC	000261	C ← 1		
SEN Set N	CC	000270	N ← 1		
SEV Set V	CC	000262	V ← 1		
SEZ Set Z	CC	000264	Z ← 1		
SOB	PC	077R00	R ← R−1	N: unaffected	The register is decremented. If it is

Table 5-1 PDP-11 Instruction Set, continued

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
Subtract One and Branch if not Equal to 0		PLUS 6-bit offset	if this result ≠ 0 then PC ← PC − (2 × offset)	Z: unaffected V: unaffected C: unaffected	not equal to 0, twice the offset is subtracted from the PC (now pointing to the following word). The offset is interpreted as a 6-bit positive number. This instruction provides a fast, efficient method of loop control. Assembler syntax is: SOB R,A where A is the address to which transfer is to be made if the decremented R is not equal to 0. Note that the SOB instruction cannot be used to transfer control in the forward direction.
SPL Set Priority Level	PC	00023N	PS bits <7:5> ← priority (priority = N)	N: unaffected Z: unaffected V: unaffected C: unaffected	The least significant three bits of the instruction are loaded into the program status word (PS), bits 7-5, thus causing a changed priority. The old priority is lost. Assembler syntax is: SPL N
SUB Subtract	DO	16SSDD	(dst) ← (dst) − (src) {in detail (dst) ← (dst) + ~ (src) + 1	N: set if result < 0 Z: set if result = 0 V: set if there is arithmetic overflow as a result of the operation, i.e., if the operands were of opposite signs and the sign of the source is the same as the sign of the result. C: set if there is a borrow into the most significant bit of the result, i.e., if (dst) + ~ (src) + 1 was less than 2^{16}.	Subtracts the source operand from the destination operand and leaves the result at the destination address. The original contents of the destination are lost. The contents of the source are not affected. In double precision arithmetic, the C bit, when set, indicates a borrow.

Table 5-1 PDP-11 Instruction Set, continued

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
SWAB Swap Bytes	SO	0003DD	tmp ← (dst) <7:0> (dst) <7:0> ← (dst) <15:8> (dst) <15:8> ← tmp	N: set if high-order bit of low-order byte (bit 7) of result is set Z: set if low-order byte of result = 0 V: cleared C: cleared	Exchanges high-order byte and low-order byte of the destination word (destination must be a word address).
SXT Sign Extend	SO	0067DD	(dst) ← 0 if N bit is clear (dst) ← −1 if N bit is set	N: unaffected Z: set if N bit clear V: cleared C: unaffected	If the condition code bit N is set, then a −1 is placed in the destination operand; if the N bit is clear, then a 0 is placed in the destination operand. This instruction is particularly useful in multiple precision arithmetic because it permits the sign to be extended through multiple words.
TRAP	PC	104400 to 104777	−(SP) ← PS −(SP) ← PC PC ← (34) PS ← (36)	N: loaded from trap vector Z: loaded from trap vector V: loaded from trap vector C: loaded from trap vector	Operation codes from 104400 to 104777 are TRAP instructions. TRAPs and EMTs are identical in operation, except that the trap vector for TRAP is at address 34. **Note:** Since DIGITAL software makes frequent use of EMT, the TRAP instruction is recommended for general use.
TST TSTB Test	SO	0057DD 1057DD	tmp ← (dst)	N: set if result < 0 Z: set if result = 0 V: cleared C: cleared	Sets the condition codes N and Z according to the contents of the destination address.
TSTSET Test Destination and Set Low Bit.	SO	0072DD	(R0) ← (dst)	N:set if R0 < 0 Z:set if R0 = 0 V:clear C:gets contents of bit 0	Reads/Locks destination word and stores it in R0. Writes/Unlocks (R0)v1 into destination. If mode is 0, traps to 10.

Table 5-1 PDP-11 Instruction Set, continued

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
WAIT Wait for Interrupt	MS	000001		N: unaffected Z: unaffected V: unaffected C: unaffected	Provides a way for the processor to relinquish use of the bus while it waits for an external interrupt. Having been given a WAIT command, the processor will not compete for the bus by fetching instructions or operands from memory. This permits higher transfer rates between device and memory, since no processor-induced latencies will be encountered by bus requests from the device. In WAIT, as in all instructions, the PC points to the next instruction following the WAIT operation. Thus, when an interrupt causes the PC and PS to be pushed onto the stack, the address of the next instruction following the WAIT is saved. The exit from the interrupt routine (i.e., execution of an RTI instruction) will cause resumption of the interrupted process at the instruction following the WAIT.
WRTLCK Read/Lock Destination. Write/Un-Lock R0 into destination.	SO	0073DD	(dst)←(R0)	N:set if R0 < 0 Z:set if R0 = 0 V:clear C:unchanged	Writes contents of R0 into destination using bus lock. If mode is 0, traps to 10.
XOR Exclusive OR	DO	074RDD	(dst) ← R ∀ (dst)	N: set if the result < 0 Z: set if result = 0 V: cleared C: unaffected	The exclusive OR of the register and destination operand is stored in the destination address. Contents of register are unaffected. Assembler format is XOR R,D.