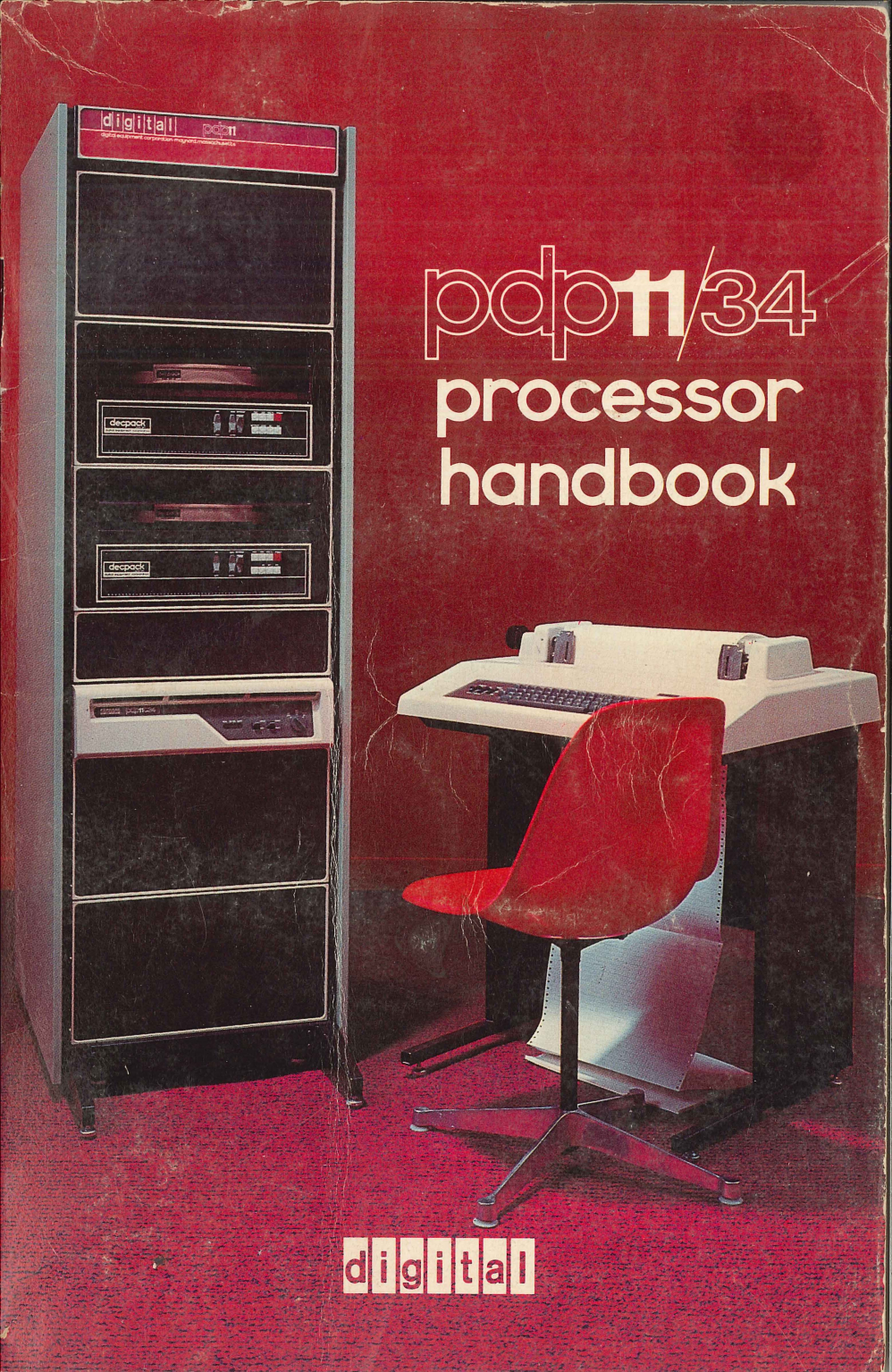


pdp11/34 processor handbook

digital

DIGITAL EQUIPMENT CORPORATION, Corporate Headquarters: Maynard, Massachusetts 01754, Telephone: (617) 897-5111
SALES AND SERVICE OFFICES
UNITED STATES—ALABAMA, Huntsville • ARIZONA, Phoenix and Tucson • CALIFORNIA, El Segundo, Los Angeles, Oakland, Ridgecrest, San Diego, San Francisco (Mountain View), Santa Ana, Santa Clara, Stanford, Sunnyvale and Woodland Hills • COLORADO, Englewood • CONNECTICUT, Fairfield and Meriden • DISTRICT OF COLUMBIA, Washington (Lanham, MD) • FLORIDA, Ft. Lauderdale and Orlando • GEORGIA, Atlanta • HAWAII, Honolulu • ILLINOIS, Chicago (Rolling Meadows) • INDIANA, Indianapolis • IOWA, Bettendorf • KENTUCKY, Louisville • LOUISIANA, New Orleans (Metairie) • MARYLAND, Odenton • MASSACHUSETTS, Marlborough, Waltham and Westfield • MICHIGAN, Detroit (Farmington Hills) • MINNESOTA, Minneapolis • MISSOURI, Kansas City (Independence) and St. Louis • NEW HAMPSHIRE, Manchester • NEW JERSEY, Cherry Hill, Fairfield, Metuchen and Princeton • NEW MEXICO, Albuquerque • NEW YORK, Albany, Buffalo (Cheektowaga), Long Island (Huntington Station), Manhattan, Rochester and Syracuse • NORTH CAROLINA, Durham/Chapel Hill • OHIO, Cleveland (Euclid), Columbus and Dayton • OKLAHOMA, Tulsa • OREGON, Eugene and Portland • PENNSYLVANIA, Allentown, Philadelphia (Bluebell) and Pittsburgh • SOUTH CAROLINA, Columbia • TENNESSEE, Knoxville and Nashville • TEXAS, Austin, Dallas and Houston • UTAH, Salt Lake City • VIRGINIA, Richmond • WASHINGTON, Bellevue • WISCONSIN, Milwaukee (Brookfield) •
INTERNATIONAL—ARGENTINA, Buenos Aires • AUSTRALIA, Adelaide, Brisbane, Canberra, Melbourne, Perth and Sydney • AUSTRIA, Vienna • BELGIUM, Brussels • BOLIVIA, La Paz • BRAZIL, Rio de Janeiro and Sao Paulo • CANADA, Calgary, Edmonton, Halifax, London, Montreal, Ottawa, Toronto, Vancouver and Winnipeg • CHILE, Santiago • DENMARK, Copenhagen • FINLAND, Helsinki • FRANCE, Grenoble and Paris • GERMANY, Berlin, Cologne, Frankfurt, Hamburg, Hannover, Munich and Stuttgart • HONG KONG • INDIA, Bombay • INDONESIA, Djakarta • IRELAND, Dublin • ITALY, Milan and Turin • JAPAN, Osaka and Tokyo • MALAYSIA, Kuala Lumpur • MEXICO, Mexico City • NETHERLANDS, Utrecht • NEW ZEALAND, Auckland • NORWAY, Oslo • PUERTO RICO, Santurce • SINGAPORE • SWEDEN, Gothenburg and Stockholm • SWITZERLAND, Geneva and Zurich • UNITED KINGDOM, Birmingham, Bristol, Edinburgh, Leeds, London, Manchester and Reading • VENEZUELA, Caracas •

digital

pdp11/34 processor handbook

digital equipment corporation

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 PDP-11 FAMILY

The PDP-11 family includes several central processor units (CPU's), a large number of peripheral devices and options, and extensive software. Future equipment will be compatible with existing family members. The user can choose the system which is most suitable for his application, but as needs change, he can easily add or change hardware.

1.2 PDP-11/34 COMPUTER

The PDP-11/34 is a systems level computer that includes increased memory expansion to 124K words, memory relocation and protection, faster processing speeds, and hardware multiply and divide instructions. The computer system is mounted in a 5¼", 10½", or 21" chassis that mounts in a standard 19" cabinet. The PDP-11/34 processor is prewired to accept additional memory (parity core or MOS) and standard peripheral device controllers including communications interfaces, mass storage controllers, etc. Additional mounting space is provided within the 10½" and 21" computer chassis for more complex controllers. The computer power supply within the chassis is capable of powering the optional internal devices.

The PDP-11/34 computer, as a member of the PDP-11 family, has the following features:

- Single & double operand instructions
  powerful and convenient set of programming instructions
- Hardware implemented multiply and divide instructions
- 16-bit word (two 8-bit bytes)
  direct addressing of 32K words or 64K bytes (K = 1024)
- Parity detection on each 8-bit byte
- Hardware address expansion and protection allowing memory addressing to 124K words
- Word or byte processing
  very efficient handling of 8-bit data without the need to rotate, swap, or mask
- Asynchronous operation
  system components run at their highest possible speed, replacement with faster subsystems means faster operation without other hardware or software changes

- Modular component design
  extreme ease and flexibility in configuring systems
- Stack processing
  hardware sequential memory manipulation makes it easy to handle structured data, subroutines, and interrupts
- Direct Memory Access (DMA)
  inherent in the architecture is direct memory access for multiple devices
- 8 internal general-purpose registers
  used interchangeably for accumulators or address generation
- Automatic Priority Interrupt
  four-line, multi-level system permits grouping of interrupt lines according to response requirements
- Vectored interrupts
  fast interrupt response without device polling
- Power Fail & Automatic Restart
  Hardware detection and software protection for fluctuations in the AC power

The minimum PDP-11/34 includes:

- Parity MOS or core memory
- Memory management
  Program protection and relocation for memory expansion to 124K 16-bit words
- Automatic bootstrap loader
  Automatic starts from a variety of peripheral devices
- Self-test feature
  ROM hardware automatically performs diagnostics on the CPU and memory
- Operator's front panel
  Allows complete control of the computer via any ASCII terminal. All front panel functions are key entries on the terminal, thereby eliminating the need and cost of a programmer's lights and switches console.

The following optional equipment is available:

- Battery backup for MOS memory
- Programmer's console
- Serial communications line interface and line frequency clock
- Large variety of standard PDP-11 peripherals

1.3 SCOPE

This Handbook describes the PDP-11/34 computer designed and manufactured by Digital Equipment Corporation.

The intent is to provide extensive information on operation of the computer in general, performance, features and basic programming. This Handbook is not intended to be the sole reference for the computer. More comprehensive and detailed information is available in the PDP-

11/34 User's Guide, Maintenance Manual, and Programming Manuals. Improvements and modifications in equipment made after January 1976 are not reflected in this Handbook.

1.4 PERIPHERALS/OPTIONS

Digital Equipment Corporation designs and manufactures many of the peripheral devices offered with PDP-11's. As a designer and manufacturer of peripherals, DIGITAL can offer extremely reliable equipment, lower prices, quantity discounts, and a wide range of computer options to choose from.

I/O DEVICES

All PDP-11 systems can use any ASCII terminal as the basic I/O device. However, I/O capabilities can be increased with high-speed paper tape reader-punches, line printers, card readers or alphanumberic display terminals. The LA36 DECwriter, a totally DIGITAL designed and built teleprinter, has several advantages over standard electromechanical typewriter terminals, including higher speed, fewer mechanical parts and very quiet operation.

PDP-11 devices include:

Cassette, TA11
Floppy disk, RX01
DECterminal alphanumeric display, VT50
DECwriter teleprinter, LA36
High Speed Line Printers, LS11, LP11, LV11
High Speed Paper Tape Reader and Punch, PC11
Teletypes, LT33
Card Readers, CR11, CD11, CM11
Graphics Terminal, GT40
Synchronous and Asynchronous Communications Interfaces

Storage Devices

Storage devices range from convenient, small-reel magnetic tape (DECtape) units to mass storage magnetic tapes and disk memories. With the UNIBUS, a large number of storage devices, in any combination, may be connected to a PDP-11 system. TU56 DECtapes, highly reliable tape units with small tape reels, designed and built by DEC, are ideal for applications with modest storage requirements. Each DECtape provides storage for 144K 16-bit words. For applications which require handling of large volumes of data, DEC offers the industry compatible TU16 Magtape.

Disk storage include fixed-head disk units and moving-head removable cartridge and disk pack units. These devices range from the 256K word RS03 fixed head disk, to the RP04 Disk Pack which can store up to 44 million words.

1.5 SOFTWARE

The PDP-11 family of central processors and peripherals is supported by a comprehensive family of licensed software products. This software family includes support for small stand-alone configurations, disk based real-time and program development systems, large multi-programming and time-sharing systems, and many diverse dedicated applications. Some examples of general purpose operating systems and standard high level language processors are:

- PAPER TAPE SYSTEM (PTS-11)—A core only high-speed paper tape system with program development in assembly language. Editor, debugger, and linker are supplied along with a relocating assembler.
- CASSETTE PROGRAMMING SYSTEM (CAPS-11)—A small program development system with a core based monitor, utilizing dual magnetic tape cassettes as file structured media. Complete program development utilities such as a relocating assembler, linker, editor, debugger, and file interchange program are included.
- SINGLE USER ON-LINE PROGRAM DEVELOPMENT SYSTEM (RT-11)—A small, powerful, easy-to-use disk (or DECtape) based system for program development or fast on-line (real-time) applications. A Foreground/Background version can accommodate simultaneous program development in the background with on-line applications in the foreground. A MACRO assembler, linker, editor, debugger, and file utility programs are included.
- MULTI-TASKING PROCESS CONTROL SYSTEM (RSX-11M)—An efficient multi-tasking system suitable for controlling many processes simultaneously, in a protected environment with concurrent development of new programs. Utilities include a MACRO assembler, task builder (linker), editor, debugger, and file utility programs.
- COMPREHENSIVE MULTI-PROGRAMMING SYSTEM (RSX-11D)—The total job operating system. As a compatible extension of RSX-11M, the system allows concurrent fully hardware protected execution of multiple on-line jobs, with BATCH program development. Complete utilities include a MACRO assembler, task builder (linker), editor, debugger, and file utility programs.
- EXTENDED RESOURCE TIME SHARING SYSTEM (RSTS/E)—A disk-based time-sharing system implementing BASIC-PLUS, an enriched version of the popular BASIC language. Up to 32 simultaneous users share system resource via interactive terminals. Additional features such as output spooling, and comprehensive file protection are included.

Languages

- BASIC-11—An extended version of Dartmouth Standard BASIC is available for PTS-11, CAPS-11 and RT-11. Many applications, such as signal processing and graphics are accessed by the user through extensions to this simple, yet powerful, language. A multiuser version is available under PTS-11 and RT-11.
- PDP-11 FORTRAN IV—An extended version of ANSI standard FORTRAN is supplied with RSX-11M and RSX-11D, and available under RT-11. As an optimizing compiler, FORTRAN IV is designed for fast compilation, yet requires very little main memory, and generates highly efficient code without sacrificing execution speed. Under RT-11, FORTRAN IV features the same signal-processing and graphics extensions as BASIC-11.
- PDP-11 COBOL—To supplement the business data processing needs often associated with large scale PDP-11 system applications, an ANSI-74 COBOL language is available under RSX-11D. Running as a BATCH job, COBOL enhances the RSX-11D total job computing system, where some business data processing is required.

In addition to the above mentioned general purpose licensed software products, DIGITAL offers a great number of optional and applications oriented products. A wide range of educational, consulting, and maintenance services are also offered, to ensure full utility of any PDP-11 system. For a complete and detailed listing of DIGITAL software products and services, consult the latest CATALOG OF SOFTWARE PRODUCTS and SERVICES.

1.6 NUMBER SYSTEMS

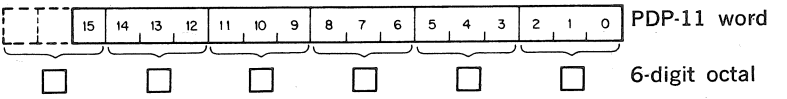
Throughout this Handbook, 3 number systems will be used; octal, binary, and decimal. So as not to clutter all numbers with subscripted bases, the following general convention will be used:

Octal—for address locations, contents of addresses, and operation codes for instructions; in most cases there will be words of 6 octal digits

Binary—for describing a single binary element; when referring to a PDP-11 word it will be 16 bits long

Decimal—for all normal referencing to quantities

Octal Representation



The 16-bit PDP-11 word can be represented conveniently as a 6-digit octal word. Bit 15, the Most Significant Bit (MSB), is used directly as the Most Significant Digit of the octal word. The other 5 octal digits are formed from the corresponding groups of 3 bits in the binary word.

When an extended address of 18 bits is used (shown later in the Handbook), the Most Significant Digit of the octal word is formed from bits 17, 16, and 15. For unsigned numbers, the correspondence between decimal and octal is:

Decimal	Octal	
0	000000	
$(2^{16}-1)=$ 65,535	177777	(16-bit limit)
$(2^{18}-1)=$262,143	777777	(18-bit limit)

2's Complement Numbers
In this system, the first bit (bit 15) is used to indicate the sign;

0=positive
1=negative

For positive numbers, the other 15 bits represent the magnitude directly; for negative numbers, the magnitude is the 2's complement of the remaining 15 bits. (The 2's complement is equal to the 1's complement plus one.) The ordering of numbers is shown below:

Decimal	2's Complement (Octal)	
	Sign Bit	Magnitude Bits
largest positive +32,767	0	77777
+32,766	0	77776
+1	0	00001
0	0	00000
−1	1	77777
−2	1	77776
−32,767	1	00001
most negative −32,768	1	00000

CHAPTER 2

SYSTEM ARCHITECTURE

2.1 UNIBUS

Most computer system components and peripherals connect to and communicate with each other on a single high-speed bus known as the UNIBUS— a key to the PDP-11's many strengths. Addresses, data, and control information are sent along the 56 lines of the bus.

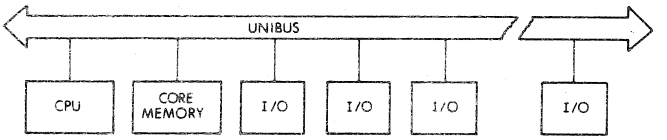


Figure 2-1 PDP-11 System Simplified Block Diagram

The form of communication is the same for every device on the UNIBUS. The processor uses the same set of signals to communicate with memory as with peripheral devices. Peripheral devices also use this set of signals when communicating with the processor, memory or other peripheral devices. Each device, including memory locations, processor registers, and peripheral device registers, is assigned an address on the UNIBUS. Thus, peripheral device registers may be manipulated as flexibly as core memory by the central processor. All the instructions that can be applied to data in core memory can be applied equally well to data in peripheral device registers. This is an especially powerful feature, considering the special capability of PDP-11 instructions to process data in any memory location as though it were an accumulator.

2.1.1 Bidirectional Lines

With bidirectional and asynchronous communications on the UNIBUS, devices can send, receive, and exchange data independently without processor intervention. For example, a cathode ray tube (CRT) display can refresh itself from a disk file while the central processor unit (CPU) attends to other tasks. Because it is asynchronous, the UNIBUS is compatible with devices operating over a wide range of speeds.

2.1.2 Master-Slave Relation

Communication between two devices on the bus is in the form of a master-slave relationship. At any point in time, there is one device that has control of the bus. This controlling device is termed the "bus master." The master device controls the bus when communicating with another device on the bus, termed the "slave." A typical example of this relationship is the processor, as master, fetching an instruction from memory (which is always a slave). Another example is the disk, as

master, transferring data to memory, as slave. Master-slave relationships are dynamic. The processor, for example, may pass bus control to a disk. The disk, as master, could then communicate with a slave memory bank.

Since the UNIBUS is used by the processor and all I/O devices, there is a priority structure to determine which device gets control of the bus. Every device on the UNIBUS which is capable of becoming bus master is assigned a priority. When two devices, which are capable of becoming a bus master, request use of the bus simultaneously, the device with the higher priority will receive control.

2.1.3 Interlocked Communication

Communication on the UNIBUS is interlocked so that for each control signal issued by the master device, there must be a response from the slave in order to complete the transfer. Therefore, communication is independent of the physical bus length (as far as timing is concerned) and the timing of each transfer is dependent only upon the response time of the master and slave devices. The asynchronous operation precludes the need for synchronizing with, and waiting for, clock impulses. Thus, each system is allowed to operate at its maximum possible speed.

Input/output devices transferring directly to or from memory are given highest priority and may request bus mastership and steal bus and memory cycles during instruction operations. The processor resumes operation immediately after the memory transfer. Multiple devices can operate simultaneously at maximum direct memory access (DMA) rates by "stealing" bus cycles.

Full 16-bit words or 8-bit bytes of information can be transferred on the bus between a master and a slave. The information can be instructions, addresses, or data. This type of operation occurs when the processor, as master, is fetching instructions, operands, and data from memory, and storing the results into memory after execution of instructions. Direct data transfers occur between a peripheral device control and memory.

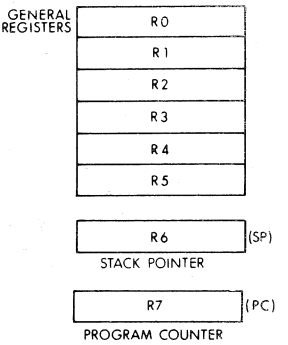
2.2 CENTRAL PROCESSOR

The central processor, connected to the UNIBUS as a subsystem, controls the time allocation of the UNIBUS for peripherals and performs arithmetic and logic operations and instruction decoding. It contains multiple high-speed general-purpose registers which can be used as accumulators, address pointers, index registers, and other specialized functions. The processor can perform data transfers directly between I/O devices and memory without disturbing the processor registers; does both single- and double-operand addressing and handles both 16-bit word and 8-bit byte data.

2.2.1 General Registers

The central processor contains 8 general registers which can be used for a variety of purposes. The registers can be used as accumulators, index registers, autoincrement registers, autodecrement registers, or as stack

pointers for temporary storage of data. Chapter 3 on Addressing describes these uses of the general registers in more detail. Arithmetic operations can be from one general register to another, from one memory or device register to another, or between memory or a device register and a general register. Refer to Figure 2-2.



Fgure 2-2 The General Registers

R7 is used as the machine's program counter (PC) and contains the address of the next instruction to be executed. It is a general register normally used only for addressing purposes and not as an accumulator for arithmetic operations.

R6 is normally used as the Stack Pointer indicating the last entry in the appropriate stack (a common temporary storage area with "Last-in First-Out" characteristics). The Memory Management feature provides 2 separate stack pointers, 1 for each of 2 operating modes (Kernel and User).

2.2.2 Instruction Set

The instruction complement uses the flexibility of the general-purpose registers to provide over 400 powerful hard-wired instructions—the most comprehensive and powerful instruction repertoire of any computer in the 16-bit class. Unlike conventional 16-bit computers, which usually have three classes of instructions (memory reference instructions, operate or AC control instructions and I/O instructions) all operations in the PDP-11 are accomplished with one set of instructions. Since peripheral device registers can be manipulated as flexibly as core memory by the central processor, instructions that are used to manipulate data in core memory may be used equally well for data in peripheral device registers. For example, data in an external device register can be tested or modified directly by the CPU, without bringing it into memory or disturbing the general registers. One can add data directly to a peripheral device register, or compare logically or arithmetically. Thus all PDP-11 instructions can be used to create a new dimension in the treatment of computer I/O and the need for a special class of I/O instructions is eliminated.

The basic order code of the PDP-11 uses both single and double operand address instructions for words or bytes. The PDP-11 therefore performs

very efficiently in one step, such operations as adding or subtracting two operands, or moving an operand from one location to another.

PDP-11 Approach

```
ADD A,B     ;add contents of location A to location B, store results at location B
```

Conventional Approach

```
LDA A       ;load contents of memory location A into AC
ADD B       ;add contents of memory location B to AC
STA B       ;store result at location B
```

Addressing

Much of the power of the PDP-11 is derived from its wide range of addressing capabilities. PDP-11 addressing modes include sequential addressing forwards or backwards, addressing indexing, indirect addressing, 16-bit word addressing, 8-bit byte addressing, and stack addressing. Variable length instruction formating allows a minimum number of bits to be used for each addressing mode. This results in efficient use of program storage space.

2.2.3 Processor Status Word

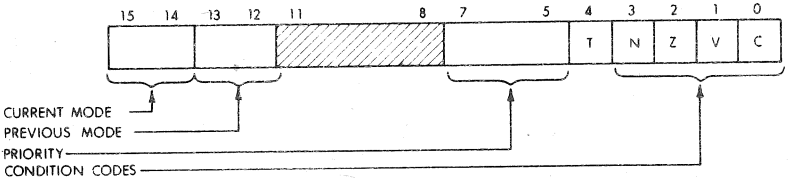


Figure 2-3 Processor Status Word

The Processor Status word (PS), at location 777776, contains information on the current status of the PDP-11. This information includes the current processor priority: current and previous modes; the condition codes describing the results of the last instruction; and an indicator for detecting the execution of an instruction to be trapped during program debugging.

Processor Priority

The Central Processor operates an any one of eight levels of priority, 0-7. When the CPU is operating at level 7 an external device cannot interrupt it with a request for service. The Central Processor must be operating at a lower priority than the external device's request in order for the interruption to take effect. The current priority is maintained in the processor status word (bits 5-7). The 8 processor levels enable high speed devices requiring rapid servicing to take priority over, and disable interruption by, low speed devices.

Condition Codes
The condition codes contain information on the result of the last CPU operation.

The bits are set as follows:

Z = 1, if the result was zero

N = 1, if the result was negative

C = 1, if the operation resulted in a carry from the MSB

V = 1, if the operation resulted in an arithmetic overflow

Current and Previous Mode
These bits indicate the relocation and protection mode of the Memory Management unit at the present time (current) and prior to the last mode change (previous). Two modes are implemented in the PDP-11/34, refered to as Kernel and User modes, see Chapter 7.

Trap
The trap bit (T) can be set or cleared under program control. When set, a processor trap will occur through location 14 on completion of instruction execution and a new Processor Status Word will be loaded. This bit is especially useful for debugging programs as it provides an efficient method of installing breakpoints.

2.2.4 Stacks

In the PDP-11, a stack is a temporary data storage area which allows a program to make efficient use of frequently accessed data. A program can add or delete words or bytes within the stack. The stack uses the "last-in, first-out" concept; that is, various items may be added to a stack in sequential order and retrieved or deleted from the stack in reverse order. On the PDP-11, a stack starts at the highest location reserved for it and expands linearly downward to the lowest address as items are added. The stack is used automatically by program interrupts, subroutine calls, and trap instructions. When the processor is interrupted, the central processor status word and the program counter are saved (pushed) onto the stack area. A new status word and program counter are then automatically acquired from an area in memory which is reserved for interrupt service routine pointers (vector area). A return from the interrupt instruction restores the original processor status and program counter and returns to the interrupted program without software intervention.

2.3 MEMORY

Memory Organization

A memory can be viewed as a series of locations, with a number (address) assigned to each location. Thus 16,384 bytes of PDP-11 memory could be shown as in Figure 2-4.

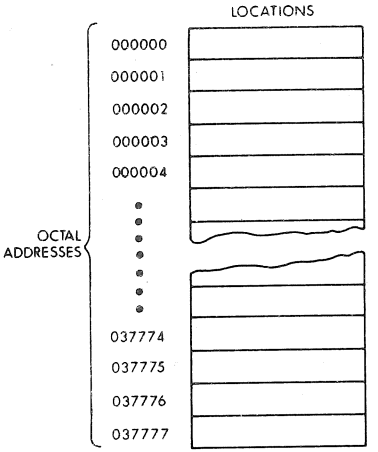


Figure 2-4 Memory Addresses

Because PDP-11 memories are designed to accommodate both 16-bit words and 8-bit bytes, the total number of addresses does not correspond to the number of words but to the number of bytes. An 8K-word memory contains 16K bytes and consist of 037777 octal locations. Words always start at even-numbered locations.

A PDP-11 word is divided into a high byte and a low byte as shown in Figure 2-5.

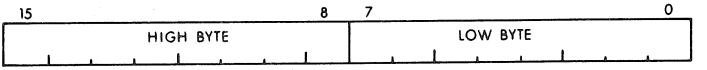


Figure 2-5 High & Low Byte

Low bytes are stored at even-numbered memory locations and high bytes at odd-numbered memory locations. Thus it is convenient to view the PDP-11 memory as shown in Figure 2-6.

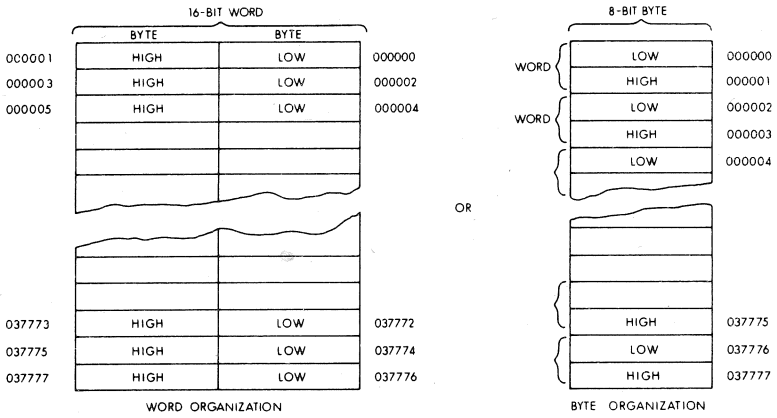


Figure 2-6 Word and Byte Addresses

Certain memory locations have been reserved by the system for interrupt and trap handling, processor stacks, general registers, and peripheral device registers. Addresses from 0 to 370_8 are always reserved and those to 777_8 are reserved on large system configurations for traps and interrupt handling.

A 16-bit word can address a maximum of 32K words of memory. However, the top 4,096 word locations are reserved for peripheral and register addresses and the user therefore has 28K to program. With the PDP-11/34 the user can expand above 28K with the Memory Management unit. This device provides an 18-bit effective memory address which permits addressing up to 124K words of actual memory.

If Memory Management is not enabled, an octal address between 160 000 and 177 777 is interpreted as 760 000 to 777 777. That is, if bit 15, 14 and 13 are 1's then bits 17 and 16 (the extended address bits) are considered to be 1's, which relocates the last 4K words (28K-32K) to the highest 4K words (124K-128K) of the UNIBUS.

2.4 AUTOMATIC PRIORITY INTERRUPTS

The multi-level automatic priority interrupt system permits the processor to respond automatically to conditions outside the system. Any number of separate devices can be attached to each level.

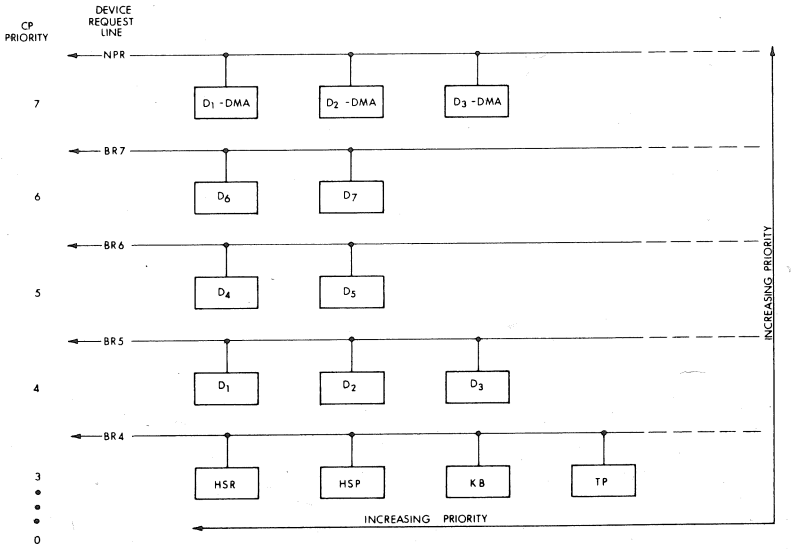


Figure 2-7 UNIBUS Priority

Each peripheral device in the PDP-11 system has a pointer to its own pair of memory words (one points to the devices's service routine, and the other contains the new processor status information). This unique identification eliminates the need for polling of devices to identify an interrupt, since the interrupt service hardware selects and begins executing the appropriate service routine after having automatically saved the status of the interrupted program segment.

The devices' interrupt priority and service routine priority are independent. This allows adjustment of system behavior in response to real-time conditions, by dynamically changing the priority level of the service routine.

The interrupt system allows the processor to continually compare its own programmable priority with the priority of any interrupting devices and to acknowledge the device with the highest level above the processor's priority level. The servicing of an interrupt for a device can be interrupted in order to service an interrupt of a higher priority. Service to the lower priority device is resumed automatically upon completion of the higher level servicing. Such a process, called nested interrupt servicing, can be carried out to any level without requiring the software to save and restore processor status at each level.

When a device (other than the central processor) is capable of becoming bus master and requests use of the bus, it is generally for one of two purposes:

1. To make a non-processor transfer of data directly to or from memory

2. To interrupt program execution and force the processor to go to a specific address where an interrupt service routine is located.

Direct Memory Access

All PDP-11's provide for direct access to memory. Any number of DMA devices may be attached to the UNIBUS. Maximum priority is given to DMA devices, thus allowing memory data storage or retrieval at memory cycle speeds. Response time is minimized by the organization and logic of the UNIBUS, which samples requests and priorities in parallel with data transfers.

Direct memory or direct data transfers can be accomplished between any two peripherals without processor supervision. These non-processor request transfers, called NPR level data transfers, are usually made for Direct Memory Access (memory to/from mass storage).

Bus Requests

Bus requests from external devices can be made on one of five request lines. Highest priority is assigned to non-processor request (NPR). These are direct memory access type transfers, and are honored by the processor during an instruction execution.

The processor's priority can be set under program control to one of eight levels using bits 7, 6, and 5 in the processor status register. These bits set a priority level that inhibits granting of bus requests on lower levels or on the same level. When the processor's priority is set to a level, for example priority level 6, all bus requests on BR6 and below are ignored.

When more than one device is connected to the same bus request (BR) line, a device nearer the central processor has a higher priority than a device farther away. Any number of devices can be connected to a given BR or NPR line.

Thus the priority system is two-dimensional and provides each device with a unique priority. Each device may be dynamically, selectively enabled or disabled under program control.

Once a device other than the processor has control of the bus, it may do one of two types of operations: data transfers or interrupt operations.

NPR Data Transfers

NPR data transfers can be made between any two peripheral devices without the supervision of the processor. Normally, NPR transfers are between a mass storage device, such as a disk, and core memory. The structure of the bus also permits device-to-device transfers, allowing customer-designed peripheral controllers to access other devices, such as disks, directly.

An NPR device has very fast access to the bus and can transfer at high data rates once it has control. The processor state is not affected by the transfer; therefore the processor can relinquish control while an instruction is in progress. This can occur at the end of any bus cycles

except in between a read-modify-write sequence. An NPR device in control of the bus may transfer 16-bit words from memory at memory speed.

BR Transfers

Devices that gain bus control with one of the Bus Request lines (BR 7-BR4) can take full advantage of the Central Processor by requesting an interrupt. In this way, the entire instruction set is available for manipulating data and status registers.

When a service routine is to be run, the current task being performed by the central processor is interrupted, and the device service routine is initiated. Once the request has been satisfied, the Processor returns to its former task.

Interrupt Procedure

Interrupt handling is automatic in the PDP-11. No device polling is required to determine which service routine to execute. The operations required to service an interrupt are as follows:

1. Processor relinquishes control of the bus, priorities permitting.

2. When a master gains control, it sends the processor an interrupt command and an unique memory address which contains the address of the device's service routine, called the interrupt vector address. Immediately following this pointer address is a word (located at vector address +2) which is to be used as a new Processor Status Word.

3. The processor stores the current Processor Status (PS) and the current Program Counter (PC) into CPU temporary registers.

4. The new PC and PS (interrupt vector) are taken from the specified address. The old PS and PC are then pushed onto the current stack. The service routine is then initiated.

5. The device service routine can cause the processor to resume the interrupted process by executing the Return from Interrupt instruction, described in Chapter 4, which pops the two top words from the current processor stack and uses them to load the PC and PS registers.

A device routine can be interrupted by a higher priority bus request any time after the new PC and PS have been loaded. If such an interrupt occurs, the PC and PS of the service routine are automatically stored in the temporary registers and then pushed onto the new current stack, and the new device routine is initiated.

Interrupt Servicing

Every hardware device capable of interrupting the processor has a unique set of locations (2 words) reserved for its interrupt vector. The first word contains the location of the device's service routine, and the second, the Processor Status Word that is to be used by the service routine. Through

proper use of the PS, the programmer can switch the priority level of the processor, and modify the Processor's Priority level to mask out lower level interrupts.

Reentrant Code

Both the interrupt handling hardware and the subroutine call hardware facilitate writing reentrant code for the PDP-11. This type of code allows a single copy of a given subroutine or program to be shared by more than one process or task. This reduces the amount of core needed for multi-task applications such as the concurrent servicing of many peripheral devices.

Power Fail and Restart

Whenever AC power drops below 95 volts for 110v power (190 volts for 220v) or outside a limit of 47 to 63 Hz, the power fail sequence is initiated. The Central Processor automaticaily traps to location 24 and the power fail program has 2 msec. to save all volatile information (data in registers), and to condition peripherals for power fail.

When power is restored the processor traps to location 24 and executes the power up routine to restore the machine to its state prior to power failure.

CHAPTER 3

ADDRESSING MODES

Data stored in memory must be accessed, and manipulated. Data handling is specified by a PDP-11 instruction (MOV, ADD etc.) which usually indicates:

the function (operation code)

a general purpose register to be used when locating the source operand and/or a general purpose register to be used when locating the destination operand.

an addressing mode (to specify how the selected register(s) is/are to be used)

Since a large portion of the data handled by a computer is usually structured (in character strings, in arrays, in lists etc.), the PDP-11 has been designed to handle structured data efficiently and flexibly. The general registers may be used with an instruction in any of the following ways:

as accumulators. The data to be manipulated resides within the register.

as pointers. The contents of the register are the address of the operand, rather than the operand itself.

as pointers which automatically step through core locations. Automatically stepping forward through consecutive core locations is known as autoincrement addressing; automatically stepping backwards is known as autodecrement addressing. These modes are particularly useful for processing tabular data.

as index registers. In this instance the contents of the register, and the word following the instruction are summed to produce the address of the operand. This allows easy access to variable entries in a list.

PDP-11's also have instruction addressing mode combinations which facilitate temporary data storage structures for convenient handling of data which must be frequently accessed. This is known as the " stack."

In the PDP-11 any register can be used as a "stack pointer"under program control, however, certain instructions associated with subroutine linkage and interrupt service automatically use Register 6 as a "hardware stack pointer". For this reason R6 is frequently referred to as the "SP".

R7 is used by the processor as its program counter (PC). It is recommended that R7 not be used as a stack pointer.

An important PDP-11 feature, which must be considered in conjunction with the addressing modes, is the register arrangement;

Six general purpose registers, (R0-R5)

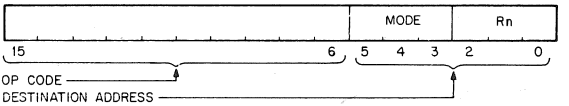
A hardware Stack Pointer (SP), register (R6)

A Program Counter (PC), register (R7).

Instruction mnemonics and address mode symbols are sufficient for writing machine language programs. The programmer need not be concerned about conversion to binary digits; this is accomplished automatically by the PDP-11 MACRO Assembler.

3.1 SINGLE OPERAND ADDRESSING

The instruction format for all single operand instructions (such as clear, increment, test) is:



Bits 15 through 6 specify the operation code that defines the type of instruction to be executed.

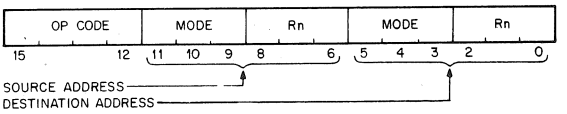
Bits 5 through 0 form a six-bit field called the destination address field. This consists of two subfields:

a) Bits 0 through 2 specify which of the eight general purpose registers is to be referenced by this instruction word.

b) Bits 3 through 5 specify how the selected register will be used (address mode). Bit 3 is set to indicate deferred (indirect) addressing.

3.2 DOUBLE OPERAND ADDRESSING

Operations which imply two operands (such as add, subtract, move and compare) are handled by instructions that specify two addresses. The first operand is called the source operand, the second the destination operand. Bit assignments in the source and destination address fields may specify different modes and different registers. The Instruction format for the double operand instruction is:



The source address field is used to select the source operand, the first operand. The destination is used similarly, and locates the second operand and the result. For example, the instruction ADD A, B adds the contents (source operand) of location A to the contents (destination operand) of location B. After execution B will contain the result of the addition and the contents of A will be unchanged.

Examples in this section and further in this chapter use the following sample PDP-11 instructions:

Mnemonic	Description	Octal Code
CLR	clear (zero the specified destination)	0050DD
CLRB	clear byte (zero the byte in the specified destination)	1050DD
INC	increment (add 1 to contents of destination)	0052DD
INCB	increment byte (add 1 to the contents of destination byte)	1052DD
COM	complement (replace the contents of the destination by their logical complement; each 0 bit is set and each 1 bit is cleared)	0051DD
COMB	complement byte (replace the contents of the destination byte by their logical complement; each 0 bit is set and each 1 bit is cleared).	1051DD
ADD	add (add source operand to destination operand and store the result at destination address)	06SSDD

DD = destination field (6 bits)

SS = source field (6 bits)

( ) = contents of

3.3 DIRECT ADDRESSING

The following table summarizes the four basic modes used with direct addressing.

DIRECT MODES

Mode	Name	Assembler Syntax	Function
0	Register	Rn	Register contains operand
2	Autoincrement	(Rn)+	Register is used as a pointer to sequential data then incremented
4	Autodecrement	-(Rn)	Register is decremented and then used as a pointer.
6	Index	X(Rn)	Value X is added to (Rn) to produce address of operand. Neither X nor (Rn) are modified.

3.3.1 Register Mode

OPR Rn

With register mode any of the general registers may be used as simple accumulators and the operand is contained in the selected register. Since they are hardware registers, within the processor, the general registers operate at high speeds and provide speed advantages when used for operating on frequently-accessed variables. The PDP-11 assembler interprets and assembles instructions of the form OPR Rn as register mode operations. Rn represents a general register name or number and OPR is used to represent a general instruction mnemonic. Assembler syntax requires that a general register be defined as follows:

R0 = %0 (% sign indicates register definition)

R1 = %1

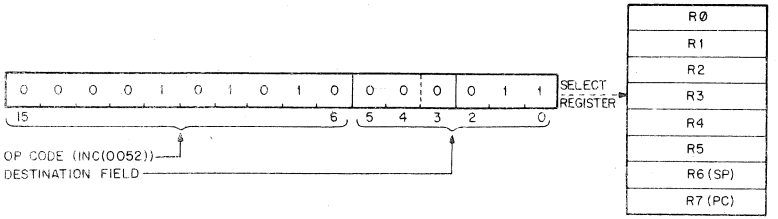
R2 = %2, etc.

Registers are typically referred to by name as R0, R1, R2, R3, R4, R5, R6 and R7. However R6 and R7 are also referred to as SP and PC, respectively.

Register Mode Examples
(all numbers in octal)

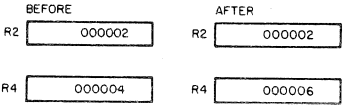
	Symbolic	Octal Code	Instruction Name
1.	INC R3	005203	Increment

Operation: Add one to the contents of general register 3



2. ADD R2,R4 060204 Add

Operation: Add the contents of R2 to the contents of R4.



3. COMB R4 105104 Complement Byte

Operation: One's complement bits 0-7 (byte) in R4. (When general registers are used, byte instructions only operate on bits 0-7; i.e. byte 0 of the register)

3.3.2 Autoincrement Mode

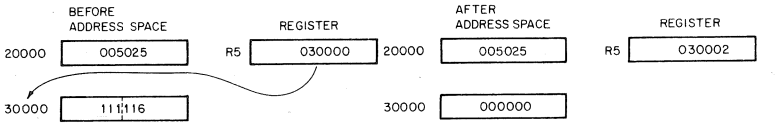
OPR (Rn)+

This mode provides for automatic stepping of a pointer through sequential elements of a table of operands. It assumes the contents of the selected general register to be the address of the operand. Contents of registers are stepped (by one for bytes, by two for words, always by two for R6 and R7) to address the next sequential location. The autoincrement mode is especially useful for array processing and stacks. It will access an element of a table and then step the pointer to address the next operand in the table. Although most useful for table handling, this mode is completely general and may be used for a variety of purposes.

Autoincrement Mode Examples

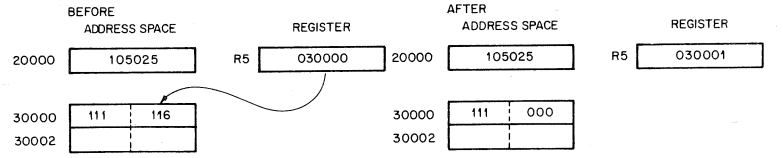
	Symbolic	Octal Code	Instruction Name
1.	CLR (R5) +	005025	Clear

Operation: Use contents of R5 as the address of the operand. Clear selected operand and then increment the contents of R5 by two.



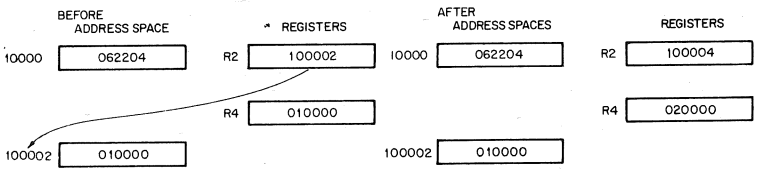
2.	CLRB (R5) +	105025	Clear Byte

Operation: Use contents of R5 as the address of the operand. Clear selected byte operand and then increment the contents of R5 by one.



3.	ADD (R2) + ,R4	062204	Add

Operation: The contents of R2 are used as the address of the operand which is added to the contents of R4. R2 is then incremented by two.



3.3.3 Autodecrement Mode

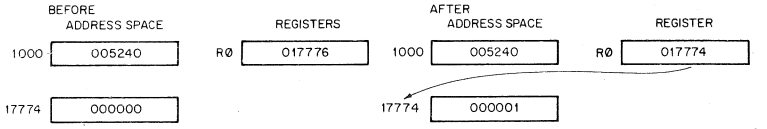
OPR-(Rn)

This mode is useful for processing data in a list in reverse direction. The contents of the selected general register are decremented (by two for word instructions, by one for byte instructions) and then used as the address of the operand. The choice of postincrement, predecrement features for the PDP-11 were not arbitrary decisions, but were intended to facilitate hardware/software stack operations.

Autodecrement Mode Examples

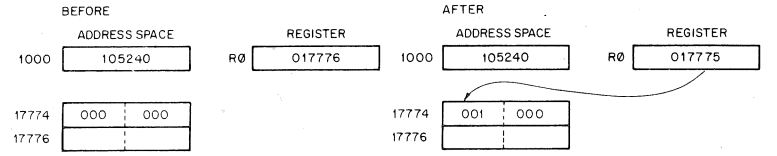
	Symbolic	Octal Code	Instruction Name
1.	INC-(R0)	005240	Increment

Operation: The contents of R0 are decremented by two and used as the address of the operand. The operand is increased by one.



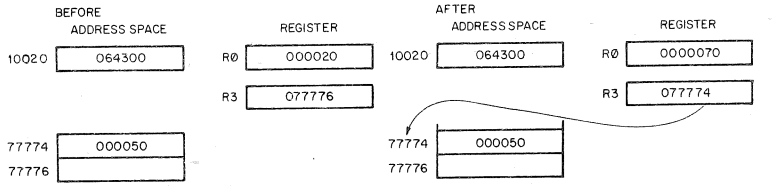
2.	INCB-(R0)	105240	Increment Byte

Operation: The contents of R0 are decremented by one then used as the address of the operand. The operand byte is increased by one.



3.	ADD-(R3),R0	064300	Add

Operation: The contents of R3 are decremented by 2 then used as a pointer to an operand (source) which is added to the contents of R0 (destination operand).



3.3.4 Index Mode

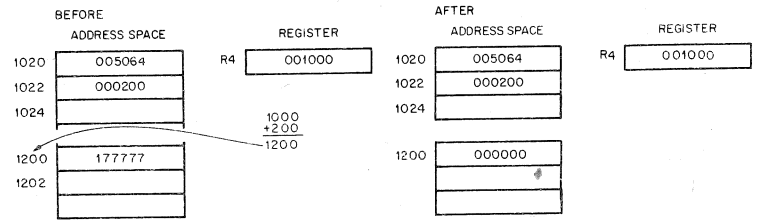
OPR X(Rn)

The contents of the selected general register, and an index word following the instruction word, are summed to form the address of the operand. The contents of the selected register may be used as a base for calculating a series of addresses, thus allowing random access to elements of data structures. The selected register can then be modified by program to access data in the table. Index addressing instructions are of the form OPR X(Rn) where X is the indexed word and is located in the memory location following the instruction word and Rn is the selected general register.

Index Mode Examples

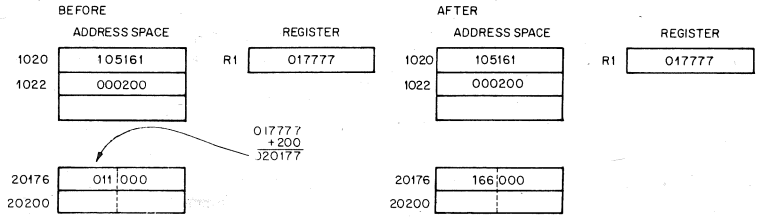
	Symbolic	Octal Code	Instruction Name
1.	CLR 200(R4)	005064 000200	Clear

Operation: The address of the operand is determined by adding 200 to the contents of R4. The location is then cleared.



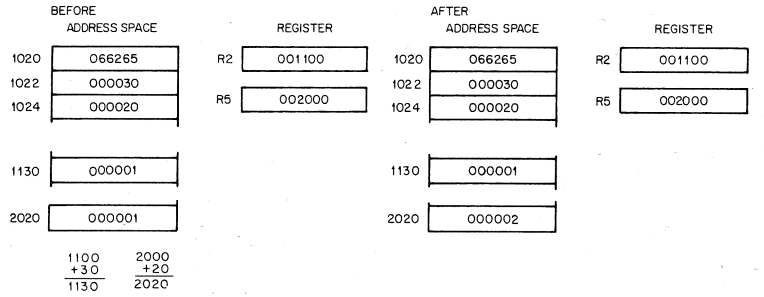
2.	COMB 200(R1)	105161 000200	Complement Byte

Operation: The contents of a location which is determined by adding 200 to the contents of R1 are one's complemented. (i.e. logically complemented)



3. ADD 30(R2),20(R5) 066265 Add
000030
000020

Operation: The contents of a location which is determined by adding 30 to the contents of R2 are added to the contents of a location which is determined by adding 20 to the contents of R5. The result is stored at the destination address, i.e. 20(R5)



3.4 DEFERRED (INDIRECT) ADDRESSING

The four basic modes may also be used with deferred addressing. Whereas in the register mode the operand is the contents of the selected register, in the register deferred mode the contents of the selected register is the address of the operand.

In the three other deferred modes, the contents of the register selects the address of the operand rather than the operand itself. These modes are therefore used when a table consists of addresses rather than operands. Assembler syntax for indicating deferred addressing is "@"(or "( )"when this is not ambiguous). The following table summarizes the deferred versions of the basic modes:

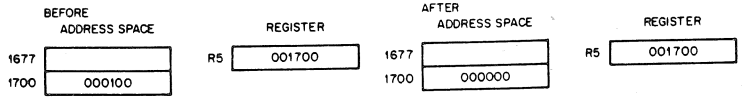
Mode	Name	Assembler Syntax	Function
1	Register Deferred	@Rn or (Rn)	Register contains the address of the operand
3	Autoincrement Deferred	@(Rn)+	Register is first used as a pointer to a word containing the address of the operand, then incremented (always by 2; even for byte instructions).
5	Autodecrement 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
7	Index Deferred	@X(Rn)	Value X (stored in a word following the instruction) 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) are modified.

Since each deferred mode is similar to its basic mode counterpart, separate descriptions of each deferred mode are not necessary. However, the following examples illustrate the deferred modes.

Register Deferred Mode Example

Symbolic	Octal Code	Instruction Name
CLR @R5	005015	Clear

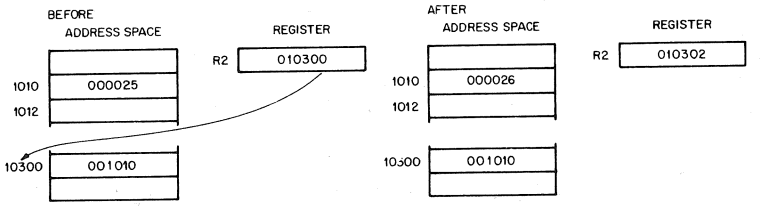
Operation: The contents of location specified in R5 are cleared.



Autoincrement Deferred Mode Example

Symbolic	Octal Code	Instruction Name
INC@(R2)+	005232	Increment

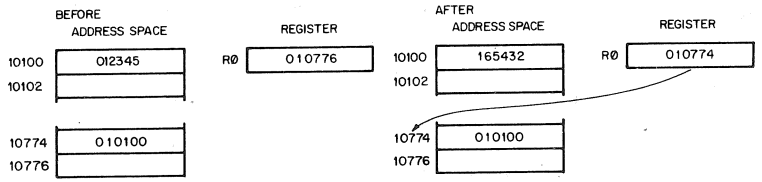
Operation: The contents of R2 are used as the address of the address of the operand. Operand is increased by one. Contents of R2 is incremented by 2.



Autodecrement Deferred Mode Example

Symbolic	Octal Code	Complement
COM @-(R0)	005150	

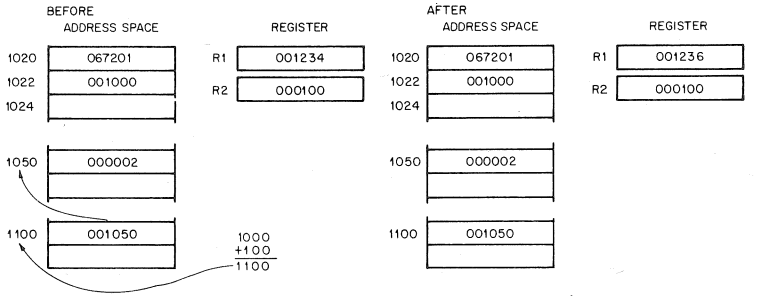
Operation: The contents of R0 are decremented by two and then used as the address of the address of the operand. Operand is one's complemented. (i.e. logically complemented)



Index Deferred Mode Example

Symbolic	Octal Code	Instruction Name
ADD @ 1000(R2),R1	067201 001000	Add

Operation: 1000 and contents of R2 are summed to produce the address of the address of the source operand the contents of which are added to contents of R1; the result is stored in R1.



3.5 USE OF THE PC AS A GENERAL REGISTER

Although Register 7 is a general purpose register, it doubles in function as the Program Counter for the PDP-11. Whenever the processor uses the program counter to acquire a word from memory, the program counter is automatically incremented by two to contain the address of the next word of the instruction being executed or the address of the next instruction to be executed. (When the program uses the PC to locate byte data, the PC is still incremented by two.)

The PC responds to all the standard PDP-11 addressing modes. However, there are four of these modes with which the PC can provide advantages for handling position independent code (PIC - see Chapter 5) and unstructured data. When regarding the PC these modes are termed immediate, absolute (or immediate deferred), relative and relative deferred, and are summarized below:

Mode	Name	Assembler Syntax	Function
2	Immediate	#n	Operand follows instruction
3	Absolute	@#A	Absolute Address follows instruction
6	Relative	A	Relative Address (index value) follows the instruction.
7	Relative Deferred	@A	Index value (stored in the word following the instruction) is the relative address for the address of the operand.

The reader should remember that the special effect modes are the same as modes described in 3.3 and 3.4, but the general register selected is R7, the program counter.

When a standard program is available for different users, it often is helpful to be able to load it into different areas of core and run it there. PDP-11's can accomplish the relocation of a program very efficiently through the use of position inde-

pendent code (PIC) which is written by using the PC addressing modes. If an instruction and its objects are moved in such a way that the relative distance between them is not altered, the same offset relative to the PC can be used in all positions in memory. Thus, PIC usually references locations relative to the current location. PIC is discussed in more detail in Chapter 5.

The PC also greatly facilitates the handling of unstructured data. This is particularly true of the immediate and relative modes.

3.5.1 Immediate Mode

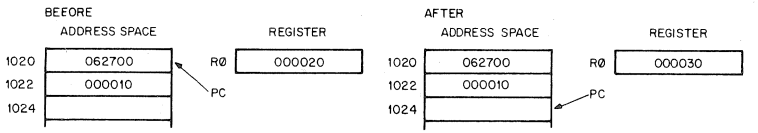
OPR #n,DD

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.

Immediate Mode Example

Symbolic	Octal Code	Instruction Name
ADD #10,R0	062700 000010	Add

Operation: 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.



3.5.2 Absolute Addressing

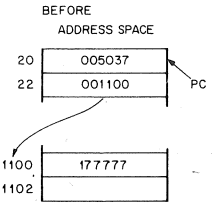
OPR @#A

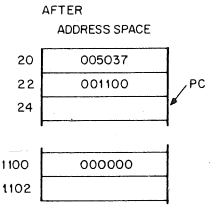
This mode is the equivalent of immediate deferred or autoincrement deferred using the PC. The contents of the location following the instruction are taken as the address of the operand. Immediate data is interpreted as an absolute address (i.e., an address that remains constant no matter where in memory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

Absolute Mode Examples

	Symbolic	Octal Code	Instruction Name
1.	CLR @#1100	005037 001100	Clear

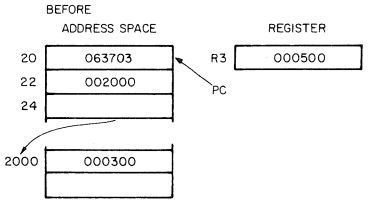
Operation: Clear the contents of location 1100.

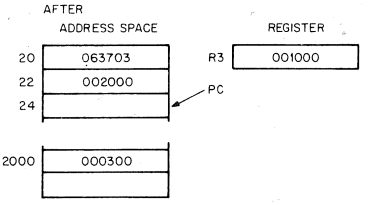




2.	ADD @#2000,R3	063703 002000	

Operation: Add contents of location 2000 to R3.





3.5.3 Relative Addressing

OPR A or OPR X(PC)

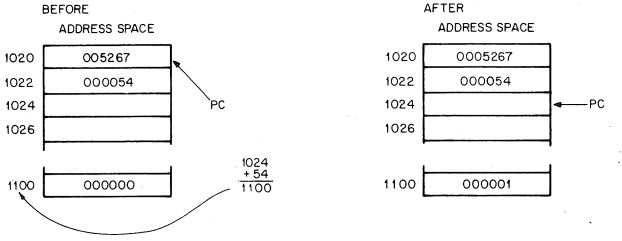
where X is the location of A relative to the instruction.

This mode is assembled as index mode using R7. The base of the address calculation, which is stored in the second or third word of the instruction, is not the address of the operand, but the number which, when added to the (PC), becomes the address of the operand. This mode is useful for writing position independent code (see Chapter 5) since the location referenced is always fixed relative to the PC. When instructions are to be relocated, the operand is moved by the same amount.

Relative Addressing Example

Symbolic	Octal Code	Instruction Name
INC A	005267 000054	Increment

Operation: To increment location A, contents of memory location immediately following instruction word are added to (PC) to produce address A. Contents of A are increased by one.



3.5.4 Relative Deferred Addressing

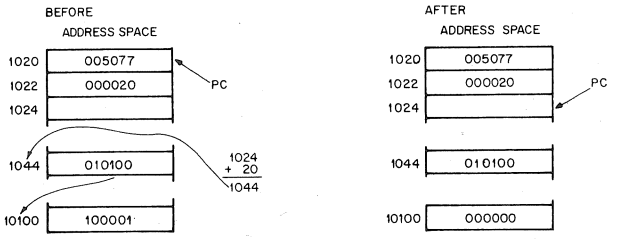
OPR@A or
OPR@X(PC), where x is location containing address of A, relative to the instruction.

This mode is similar to the relative mode, except that the second word of the instruction, when added to the PC, contains the address of the address of the operand, rather than the address of the operand.

Relative Deferred Mode Example

Symbolic	Octal Code	Instruction Name
CLR @A	005077 000020	Clear

Operation: Add second word of instruction to PC to produce address of address of operand. Clear operand.



3.6 USE OF STACK POINTER AS GENERAL REGISTER

The processor stack pointer (SP, Register 6) is in most cases the general register used for the stack operations related to program nesting. Autodecrement with Register 6 "pushes" data on to the stack and autoincrement with Register 6 "pops" data off the stack. Index mode with SP permits random access of items on the stack. Since the SP is used by the processor for interrupt handling, it has a special attribute: autoincrements and autodecrements are always done in steps of two. Byte operations using the SP in this way leave odd addresses unmodified.

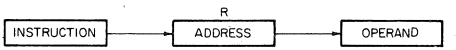
3.7 SUMMARY OF ADDRESSING MODES

3.7.1 General Register Addressing

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

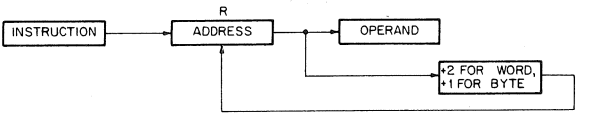
Mode 0 **Register** OPR R R contains operand

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

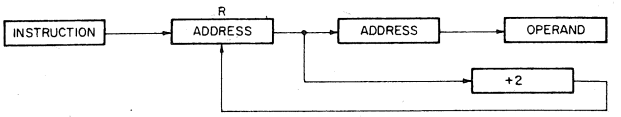


Mode 2 **Auto-increment** OPR (R)+

R contains address, then increment (R)

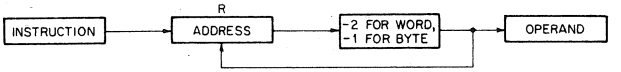


Mode 3 **Auto-increment deferred** OPR @(R)+ R contains address of address, then increment (R) by 2

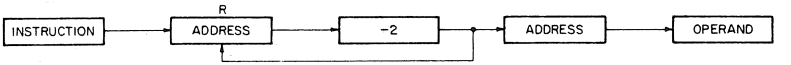


Mode 4 **Auto-decrement** OPR —(R)

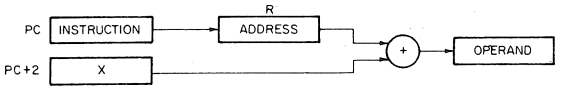
Decrement (R), then R contains address



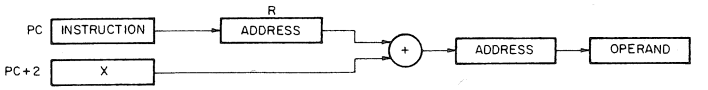
Mode 5 **Auto-decrement deferred** OPR @—(R) Decrement (R) by 2, then R contains address of address



Mode 6 **Index** OPR X(R) (R) + X is address



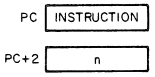
Mode 7 **Index deferred** OPR @X(R) (R) + X is address of address



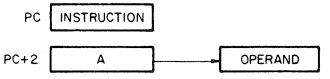
3.7.2 Program Counter Addressing

Register = 7

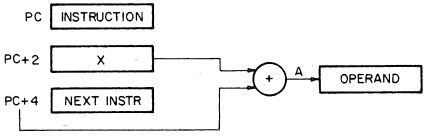
Mode 2 **Immediate** OPR #n Operand n follows instruction



Mode 3 **Absolute** OPR @#A Address A follows instruction

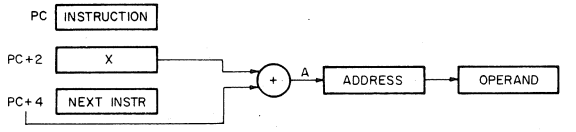


Mode 6 **Relative** OPR A $\underbrace{PC + 4}_{\text{updated PC}} + X$ is address



Mode 7 **Relative deferred** OPR @A

$\underbrace{PC + 4}_{\text{updated PC}} + X$ is address of address



CHAPTER 4

INSTRUCTION SET

4.1 INTRODUCTION

The specification for each instruction includes the mnemonic, octal code, binary code, a diagram showing the format of the instruction, a symbolic notation describing its execution and the effect on the condition codes, a description, special comments, and examples.

MNEMONIC: This is indicated at the top corner of each page. When the word instruction has a byte equivalent, the byte mnemonic is also shown.

INSTRUCTION FORMAT: A diagram accompanying each instruction shows the octal op code, the binary op code, and bit assignments. (Note that in byte instructions the most significant bit (bit 15) is always a 1.)

SYMBOLS:

( ) = contents of

SS or src = source address

DD or dst = destination address

loc = location

← = becomes

↑ = "is popped from stack"

↓ = "is pushed onto stack"

$\wedge$ = boolean AND

v = boolean OR

$\forall$ = exclusive OR

$\sim$ = boolean not

Reg or R = register

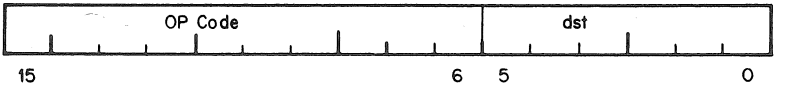
B = Byte

■ = { 0 for word
    { 1 for byte

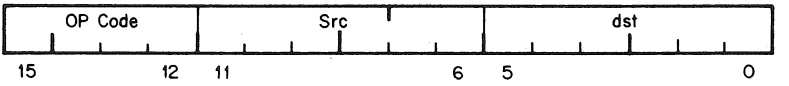
4.2 INSTRUCTION FORMATS

The major instruction formats are:

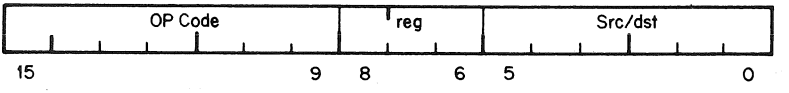
Single Operand Group



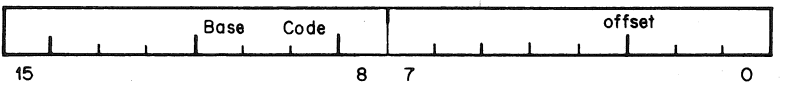
Double Operand Group



Register-Source or Destination

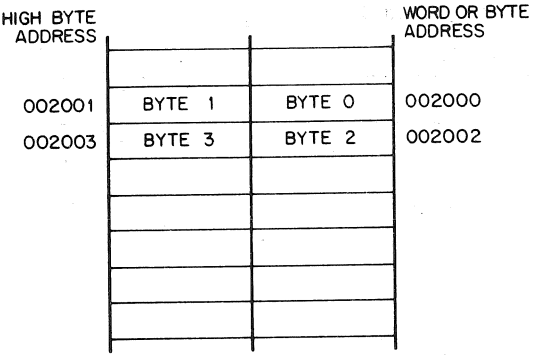


Branch



Byte Instructions

The PDP-11 processor includes a full complement of instructions that manipulate byte operands. Since all PDP-11 addressing is byte-oriented, byte manipulation addressing is straightforward. Byte instructions with autoincrement or autodecrement direct addressing cause the specified register to be modified by one to point to the next byte of data. Byte operations in register mode access the low-order byte of the specified register. These provisions enable the PDP-11 to perform as either a word or byte processor. The numbering scheme for word and byte addresses in core memory is:



The most significant bit (Bit 15) of the instruction word is set to indicate a byte instruction.

Example:

Symbolic	Octal	
CLR	0050DD	Clear Word
CLRB	1050DD	Clear Byte

NOTE

The term PC (Program Counter) in the **Operation** explanation of the instructions refers to the updated PC.

4.3 LIST OF INSTRUCTIONS

Instructions are shown in the following sequence. Other instructions are found in Chapters 6 and 7.

SINGLE OPERAND

Mnemonic	Instruction	Op Code	Page
General			
CLR(B)	clear destination	■050DD	4-6
COM(B)	complement dst	■051DD	4-7
INC(B)	increment dst	■052DD	4-8
DEC(B)	decrement dst	■053DD	4-9
NEG(B)	negate dst	■054DD	4-10
TST(B)	test dst	■057DD	4-11
Shift & Rotate			
ASR(B)	arithmetic shift right	■062DD	4-13
ASL(B)	arithmetic shift left	■063DD	4-14
ROR(B)	rotate right	■060DD	4-15
ROL(B)	rotate left	■061DD	4-16
SWAB	swap bytes	0003DD	4-17
Multiple Precision			
ADC(B)	add carry	■055DD	4-19
SBC(B)	subtract carry	■056DD	4-20
SXT	sign extend	0067DD	4-20
Processor Status			
MFPS	move from PS	1067DD	4-21
MTPS	move to PS	1064SS	4-22

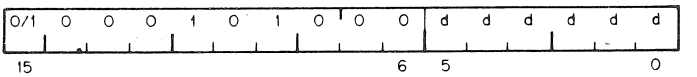
DOUBLE OPERAND

General			
MOV(B)	move source to destination	■1SSDD	4-23
CMP(B)	compare src to dst	■2SSDD	4-24
ADD	add src to dst	06SSDD	4-25
SUB	subtract src from dst	16SSDD	4-26
Logical			
BIT(B)	bit test	■3SSDD	4-28
BIC(B)	bit clear	■4SSDD	4-29
BIS(B)	bit set	■5SSDD	4-30
XOR	exclusive OR	074RDD	4-31

PROGRAM CONTROL

CLR
CLRB

clear destination ■050DD



Operation: (dst)←0

Condition Codes:
N: cleared
Z: set
V: cleared
C: cleared

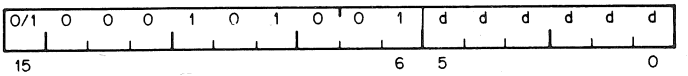
Description: Word: Contents of specified destination are replaced with zeroes.
Byte: Same

Example:

```
                                CLR R1

            Before                          After
(R1) = 177777                   (R1) = 000000

          N Z V C                         N Z V C
          1 1 1 1                         0 1 0 0
```

COM
COMB

complement dst ■051DD



Operation: (dst)←~(dst)

Condition Codes:
N: set if most significant bit of result is set; cleared otherwise
Z: set if result is 0; cleared otherwise
V: cleared
C: set

Description: Replaces the contents of the destination address by their logical complement (each bit equal to 0 is set and each bit equal to 1 is cleared)
Byte: Same

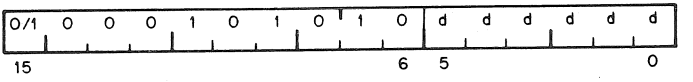
Example:

COM R0

Before	After
(R0) = 013333	(R0) = 164444
N Z V C	N Z V C
0 1 1 0	1 0 0 1

INC
INCB

increment dst ■052DD



Operation: (dst)←(dst) + 1

Condition Codes:
N: set if result is <0; cleared otherwise
Z: set if result is 0; cleared otherwise
V: set if (dst) held 077777; cleared otherwise
C: not affected

Description: Word: Add one to contents of destination
Byte: Same

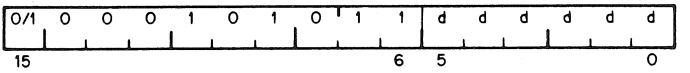
Example:

INC R2

Before	After
(R2) = 000333	(R2) = 000334
N Z V C	N Z V C
0 0 0 0	0 0 0 0

DEC
DECB

decrement dst ■053DD



Operation: (dst)←(dst)−1

Condition Codes:
N: set if result is <0; cleared otherwise
Z: set if result is 0; cleared otherwise
V: set if (dst) was 100000; cleared otherwise
C: not affected

Description: Word: Subtract 1 from the contents of the destination
Byte: Same

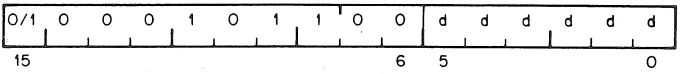
Example:

DEC R5

Before	After
(R5) = 000001	(R5) = 000000
N Z V C	N Z V C
1 0 0 0	0 1 0 0

NEG
NEGB

negate dst ■054DD



Operation: (dst)← −(dst)

Condition Codes:
N: set if the result is <0; cleared otherwise
Z: set if result is 0; cleared otherwise
V: set if the result is 100000; cleared otherwise
C: cleared if the result is 0; set otherwise

Description: Word: Replaces the contents of the destination address by its two's complement. Note that 100000 is replaced by itself -(in two's complement notation the most negative number has no positive counterpart).
Byte: Same

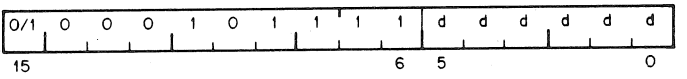
Example:

NEG R0

Before	After
(R0) = 000010	(R0) = 177770
N Z V C	N Z V C
0 0 0 0	1 0 0 1

TST
TSTB

test dst ■057DD



Operation: (dst)←(dst)

Condition Codes:
N: set if the result is <0; cleared otherwise
Z: set if result is 0; cleared otherwise
V: cleared
C: cleared

Description: Word: Sets the condition codes N and Z according to the contents of the destination address
Byte: Same

Example:

TST R1

Before	After
(R1) = 012340	(R1) = 012340
N Z V C	N Z V C
0 0 1 1	0 0 0 0

Shifts

Scaling data by factors of two is accomplished by the shift instructions:

ASR - Arithmetic shift right

ASL - Arithmetic shift left

The sign bit (bit 15) of the operand is replicated in shifts to the right. The low order bit is filled with 0 in shifts to the left. Bits shifted out of the C bit, as shown in the following examples, are lost.

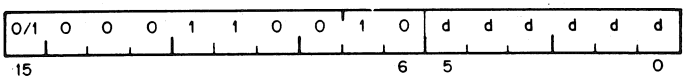
Rotates

The rotate instructions operate on the destination word and the C bit as though they formed a 17-bit "circular buffer'. These instructions facilitate sequential bit testing and detailed bit manipulation.

ASR
ASRB

arithmetic shift right

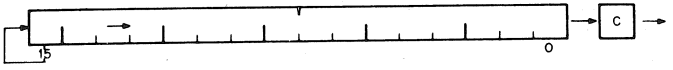
■062DD



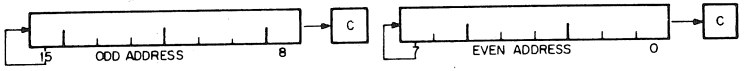
Operation: (dst)←(dst) shifted one place to the right

Condition Codes:
N: set if the high-order bit of the result is set (result < 0); cleared otherwise
Z: set if the result = 0; cleared otherwise
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

Description: Word: Shifts all bits of the destination right one place. Bit 15 is replicated. The C-bit is loaded from bit 0 of the destination. ASR performs signed division of the destination by two.
Word:

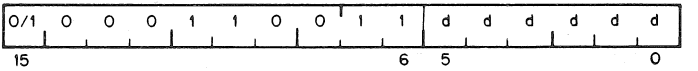


Byte:



ASL
ASLB

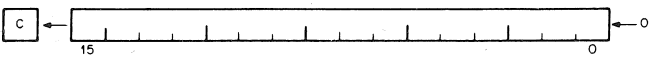
arithmetic shift left ■063DD



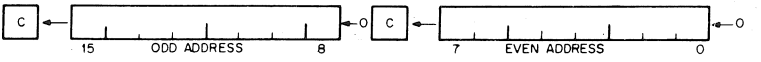
Operation: (dst)←(dst) shifted one place to the left

Condition Codes:
N: set if high-order bit of the result is set (result < 0); cleared otherwise
Z: set if the result = 0; cleared otherwise
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

Description:
Word: Shifts all bits of the destination left one place. Bit 0 is loaded with an 0. The C-bit of the status word is loaded from the most significant bit of the destination. ASL performs a signed multiplication of the destination by 2 with overflow indication.
Word:



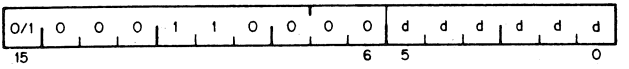
Byte:



ROR
RORB

rotate right

■060DD

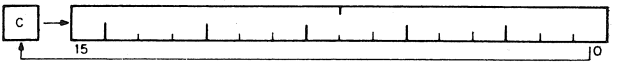


Condition Codes: N: set if the high-order bit of the result is set (result < 0); cleared otherwise
Z: set if all bits of result = 0; cleared otherwise
V: loaded with the Exclusive OR of the N-bit and C-bit (as set by the completion of the rotate operation)
C: loaded with the low-order bit of the destination

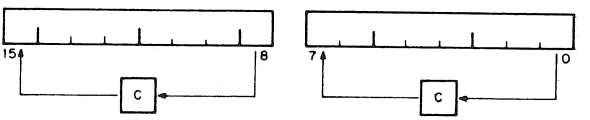
Description: Rotates all bits of the destination right one place. Bit 0 is loaded into the C-bit and the previous contents of the C-bit are loaded into bit 15 of the destination.
Byte: Same

Example:

Word:

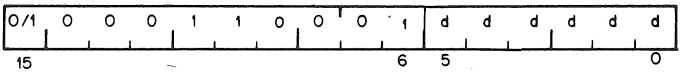


Byte:



ROL
ROLB

rotate left ■061DD

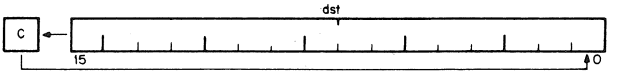


Condition Codes: N: set if the high-order bit of the result word is set (result < 0): cleared otherwise
Z: set if all bits of the result word = 0; cleared otherwise
V: loaded with the Exclusive OR of the N-bit and C-bit (as set by the completion of the rotate operation)
C: loaded with the high-order bit of the destination

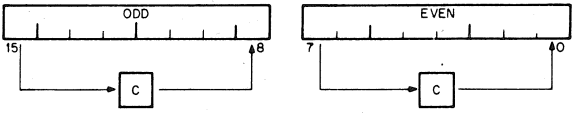
Description: Word: Rotate all bits of the destination left one place. Bit 15 is loaded into the C-bit of the status word and the previous contents of the C-bit are loaded into Bit 0 of the destination.
Byte: Same

Example:

Word:

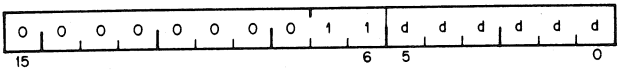


Bytes:



SWAB

swap bytes 0003DD



Operation: Byte 1/Byte 0 ← Byte 0/Byte 1

Condition Codes:
N: set if high-order bit of low-order byte (bit 7) of result is set; cleared otherwise
Z: set if low-order byte of result = 0; cleared otherwise
V: cleared
C: cleared

Description: Exchanges high-order byte and low-order byte of the destination word (destination must be a word address).

Example:

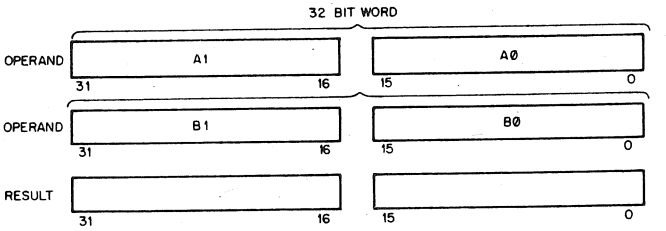
SWAB R1

Before	After
(R1) = 077777	(R1) = 177577
N Z V C	N Z V C
1 1 1 1	0 0 0 0

Multiple Precision

It is sometimes necessary to do arithmetic on operands considered as multiple words or bytes. The PDP-11 makes special provision for such operations with the instructions ADC (Add Carry) and SBC (Subtract Carry) and their byte equivalents.

For example two 16-bit words may be combined into a 32-bit double precision word and added or subtracted as shown below:



Example:

The addition of -1 and -1 could be performed as follows:

-1 = 37777777777

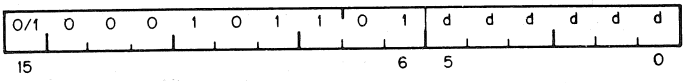
(R1) = 177777 (R2) = 177777 (R3) = 177777 (R4) = 177777

```
ADD   R1,R2
ADC   R3
ADD   R4,R3
```

1. After (R1) and (R2) are added, 1 is loaded into the C bit
2. ADC instruction adds C bit to (R3); (R3) = 0
3. (R3) and (R4) are added
4. Result is 37777777776 or -2

ADC
ADCB

add carry ■055DD



Operation: (dst)←(dst) + (C)

Condition Codes:
N: set if result <0; cleared otherwise
Z: set if result =0; cleared otherwise
V: set if (dst) was 077777 and (C) was 1; cleared otherwise
C: set if (dst) was 177777 and (C) was 1; cleared otherwise

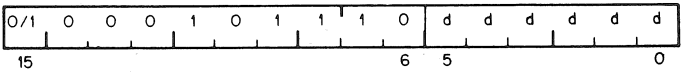
Description: Adds the contents of the C-bit into the destination. This permits the carry from the addition of the low-order words to be carried into the high-order result.
Byte: Same

Example: Double precision addition may be done with the following instruction sequence:

```
ADD   A0,B0      ; add low-order parts
ADC   B1         ; add carry into high-order
ADD   A1,B1      ; add high order parts
```

SBC
SBCB

subtract carry ■056DD



Operation: (dst)←(dst)−(C)

Condition Codes:
N: set if result 0; cleared otherwise
Z: set if result 0; cleared otherwise
V: set if (dst) was 100000; cleared otherwise
C: set if (dst) was 0 and C was 1; cleared otherwise

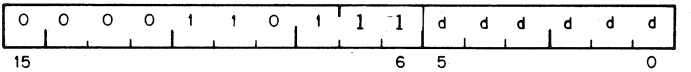
Description: Word: Subtracts the contents of the C-bit from the destination. This permits the carry from the subtraction of two low-order words to be subtracted from the high order part of the result.
Byte: Same

Example: Double precision subtraction is done by:

```
SUB   A0,B0
SBC   B1
SUB   A1,B1
```

SXT

sign extend 0067DD



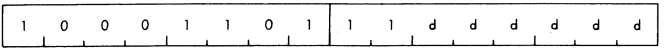
Operation: (dst) ← 0 if N bit is clear
(dst) ← -1 N bit is set

Condition Codes:
N: unaffected
Z: set if N bit clear
V: cleared
C: unaffected

Description: If the condition code bit N is set then a −1 is placed in the destination operand: if 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.

MFPS

move byte from processor status word 1067DD



Operation: (dst) ← PS <0:7>
dst lower 8 bits

Condition Code Bits:
N = set if PS bit 7 = 1; cleared otherwise
Z = set if PS <0:7> = 0; cleared otherwise
V = cleared
C = not affected

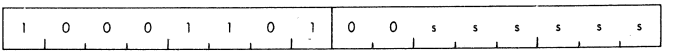
Description: The 8 bit contents of the PS are moved to the effective destination. If destination is mode 0, PS bit 7 is sign extended through the upper byte of the register. The destination operand address is treated as a byte address.

Example:

```
MFPS R0

        before              after

R0 [0]                  R0 [000014]
PS [000014]             PS [000000]
```

MTPS

move byte to processor status word 1064SS



Operation: PS <0:7> ← (SRC)

Condition Codes:
N = set if (SRC) <7> = 1; cleared otherwise
Z = set if (SRC) <0:7> = 0; cleared otherwise
V = cleared
C = not affected

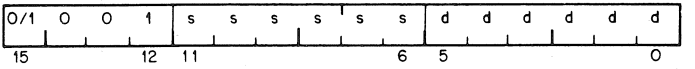
Description: The 8 bits of the effective operand replaces the current contents of the PS <0:7>. The source operand address is treated as a byte address.
Note that the T bit (PS bit 4) cannot be set with this instruction. The SRC operand remains unchanged. This instruction can be used to change the priority bits (PS <5:7>) in the PS.

4.5 DOUBLE OPERAND INSTRUCTIONS

Double operand instructions provide an instruction (and time) saving facility since they eliminate the need for "load"and "save" sequences such as those used in accumulator-oriented machines.

MOV
MOVB

move source to destination ■1SSDD



Operation: (dst)←(src)

Condition Codes:
N: set if (src) <0; cleared
Z: set if (src) = 0; cleared
V: cleared
C: not affected

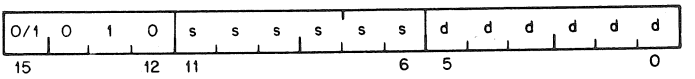
Description: Word: Moves the source operand to the destination location. The previous contents of the destination are lost. The contents of the source address are not affected.
Byte: Same as MOV. The MOVB to a register (unique among byte instructions) extends the most significant bit of the low order byte (sign extension). Otherwise MOVB operates on bytes exactly as MOV operates on words.

Example:

```
MOV    XXX,R1              ; loads Register 1 with the contents of memory location; XXX represents a programmer-defined mnemonic used to represent a memory location

MOV    #20,R0              ; loads the number 20 into Register 0; "#"indicates that the value 20 is the operand

MOV @#20,-(R6)             ; pushes the operand contained in location 20 onto the stack

MOV    (R6)+,@ #177566     ; pops the operand off a stack and moves it into memory location 177566 (terminal print buffer)

MOV    R1,R3               ; performs an inter register transfer

MOVB @#177562,@#177566     ; moves a character from terminal keyboard buffer to terminal buffer
```

CMP
CMPB

compare src to dst ■2SSDD



Operation: (src)–(dst)

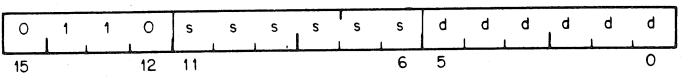
Condition Codes:
N: set if result <0; cleared otherwise
Z: set if result =0; cleared otherwise
V: set if there was arithmetic overflow; that is, operands were of opposite signs and the sign of the destination was the same as the sign of the result; cleared otherwise
C: cleared if there was a carry from the most significant bit of the result; set otherwise

Description: 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 compare is customarily followed by a conditional branch instruction. Note that unlike the subtract instruction the order of operation is (src)–(dst), not (dst)–(src).

ADD

add src to dst

06SSDD



Operation: (dst)←(src) + (dst)

Condition Codes:
N: set if result <0; cleared otherwise
Z: set if result = 0; cleared otherwise
V: set if there was arithmetic overflow as a result of the operation; that is both operands were of the same sign and the result was of the opposite sign; cleared otherwise
C: set if there was a carry from the most significant bit of the result; cleared otherwise

Description: 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. Two's complement addition is performed.

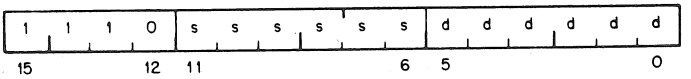
Examples:

Add to register:	ADD 20,R0
Add to memory:	ADD R1,XXX
Add register to register:	ADD R1,R2
Add memory to memory:	ADD@ # 17750,XXX

XXX is a programmer-defined mnemonic for a memory location.

SUB

subtract src from dst 16SSDD



Operation: (dst)←(dst)−(src)

Condition Codes:
N: set if result <0; cleared otherwise
Z: set if result =0; cleared otherwise
V: set if there was arithmetic overflow as a result of the operation, that is if operands were of opposite signs and the sign of the source was the same as the sign of the result; cleared otherwise
C: cleared if there was a carry from the most significant bit of the result; set otherwise

Description: Subtracts the source operand from the destination operand and leaves the result at the destination address. The orignial 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"

Example:

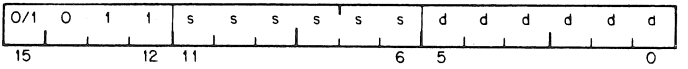
SUB R1,R2

Before	After
(R1) = 011111	(R1) = 011111
(R2) = 012345	(R2) = 001234
N Z V C	N Z V C
1 1 1 1	0 0 0 0

Logical
These instructions have the same format as the double operand arithmetic group. They permit operations on data at the bit level.

BIT
BITB

bit test ■3SSDD



Operation: (src) Λ (dst)

Condition Codes: N: set if high-order bit of result set; cleared otherwise
Z: set if result = 0; cleared otherwise
V: cleared
C: not affected

Description: Performs logical "and" comparison of the source and destination operands and modifies condition codes accordingly. Neither the 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 also set in the source or whether all corresponding bits set in the destination are clear in the source.

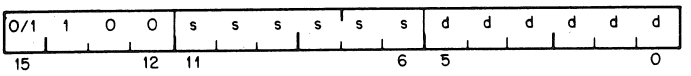
Example:

```
BIT    #30.R3          ; test bits 3 and 4 of R3 to see
                       ; if both are off
```

$(30)_8$=0 000 000 000 011 000

BIC
BICB

bit clear ■4SSDD



Operation: (dst)←~(src)Λ(dst)

Condition Codes:
N: set if high order bit of result set; cleared otherwise
Z: set if result = 0; cleared otherwise
V: cleared
C: not affected

Description: 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.

Example:

BIC R3,R4

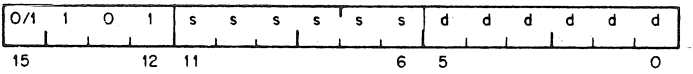
Before	After
(R3) = 001234	(R3) = 001234
(R4) = 001111	(R4) = 000101
N Z V C	N Z V C
1 1 1 1	0 0 0 1

Before: (R3)=0 000 001 010 011 100
(R4)=0 000 001 001 001 001

After: (R4)=0 000 000 001 000 001

BIS
BISB

bit set ■5SSDD



Operation: (dst)←(src) v (dst)

Condition Codes:
N: set if high-order bit of result set, cleared otherwise
Z: set if result = 0; cleared otherwise
V: cleared
C: not affected

Description: Performs "Inclusive OR" operation between the source and destination operands and leaves the result at the destination address; that is, corresponding bits set in the source are set in the destination. The contents of the destination are lost.

Example:

BIS R0,R1

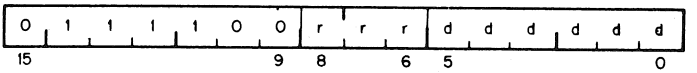
Before	After
(R0) = 001234	(R0) = 001234
(R1) = 001111	(R1) = 001335
N Z V C	N Z V C
0 0 0 0	0 0 0 0

Before: (R0)=0 000 001 010 011 100
(R1)=0 000 001 001 001 001

After: (R1)=0 000 001 011 011 101

XOR

exclusive OR 074RDD



Operation: (dst)←R∨(dst)

Condition Codes:
N: set if the result <0; cleared otherwise
Z: set if result =0; cleared otherwise
V: cleared
C: unaffected

Description: 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

Example:

XOR R0,R2

Before	After
(R0) = 001234	(R0) = 001234
(R2) = 001111	(R2) = 000325

Before: (R0)=0 000 001 010 011 100
(R2)=0 000 001 001 001 001

After: (R2)=0 000 000 011 010 101

4.6 PROGRAM CONTROL INSTRUCTIONS

Branches

The instruction causes a branch to a location defined by the sum of the offset (multiplied by 2) and the current contents of the Program Counter if:

a) the branch instruction is unconditional

b) it is conditional and the conditions are met after testing the condition codes (status word).

The offset is the number of words from the current contents of the PC. Note that the current contents of the PC point to the word following the branch instruction.

Although the PC expresses a byte address, the offset is expressed in words. The offset is automatically multiplied by two to express bytes before it is added to the PC. Bit 7 is the sign of the offset. If it is set, the offset is negative and the branch is done in the backward direction. Similarly if it is not set, the offset is positive and the branch is done in the forward direction.

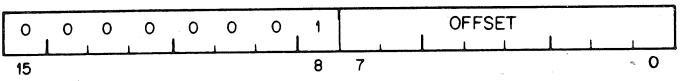
The 8-bit offset allows branching in the backward direction by 200$_8$ words (400$_8$ bytes) from the current PC, and in the forward direction by 177$_8$ words (376$_8$ bytes) from the current PC.

The PDP-11 assembler handles address arithmetic for the user and computes and assembles the proper offset field for branch instructions in the form:

Bxx loc

Where "Bxx" is the branch instruction and "loc" is the address to which the branch is to be made. The assembler gives an error indication in the instruction if the permissable branch range is exceeded. Branch instructions have no effect on condition codes.

branch (unconditional) 000400 Plus offset



Operation: PC ← PC + (2 x offset)

Description: Provides a way of transferring program control within a range of -128 to +127 words with a one word instruction.

New PC address = updated PC + (2 X offset)

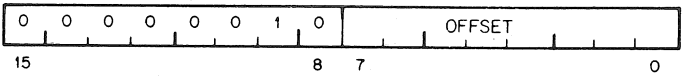
Updated PC = address of branch instruction + 2

Example: With the Branch instruction at location 500, the following offsets apply.

New PC Address	Offset Code	Offset (decimal)
474	375	−3
476	376	−2
500	377	−1
502	000	0
504	001	+1
506	002	+2

BNE

branch if not equal (to zero) 001000 Plus offset



Operation: PC ← PC + (2 x offset) if Z = 0

Condition Codes: Unaffected

Description: 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 in the source, following a BIT, and generally, to test that the result of the previous operation was not zero.

Example:

```
CMP   A,B        ; compare A and B
BNE   C          ; branch if they are not equal
```

will branch to C if $A \neq B$

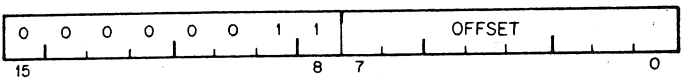
and the sequence

```
ADD   A,B        ; add A to B
BNE   C          ; Branch if the result is not
                   equal to 0
```

will branch to C if $A + B \neq 0$

BEQ

branch if equal (to zero) 001400 Plus offset



Operation: PC ← PC + (2 x offset) if Z = 1

Condition Codes: Unaffected

Description: Tests the state of the Z-bit and causes a branch if Z is set. As an example, it is used to test equality following a CMP operation, 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 zero.

Example:

```
CMP   A,B          ; compare A and B
BEQ   C            ; branch if they are equal
```

will branch to C if A = B (A - B = 0)
and the sequence

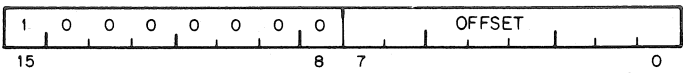
```
ADD   A,B          ; add A to B
BEQ   C            ; branch if the result = 0
```

will branch to C if A + B = 0.

BPL

branch if plus

100000 Plus offset



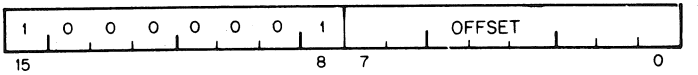
Operation: PC ← PC + (2 x offset) if N = 0

Description: Tests the state of the N-bit and causes a branch if N is clear, (positive result).

BMI

branch if minus

100400 Plus offset



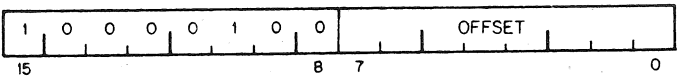
Operation: PC ← PC + (2 x offset) if N = 1

Condition Codes: Unaffected

Description: Tests the state of the N-bit and causes a branch if N is set. It is used to test the sign (most significant bit) of the result of the previous operation), branching if negative.

BVC

branch if overflow is clear 102000 Plus offset



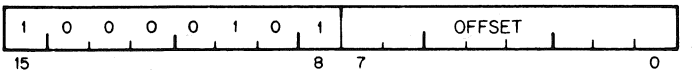
Operation: PC ← PC + (2 x offset) if V = 0

Description: Tests the state of the V bit and causes a branch if the V bit is clear. BVC is complementary operation to BVS.

BVS

branch if overflow is set

102400 Plus offset

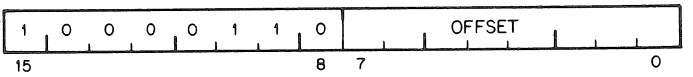


Operation: PC ← PC + (2 x offset) if V = 1

Description: Tests the state of V bit (overflow) and causes a branch if the V bit is set. BVS is used to detect arithmetic overflow in the previous operation.

BCC

branch if carry is clear — 103000 Plus offset

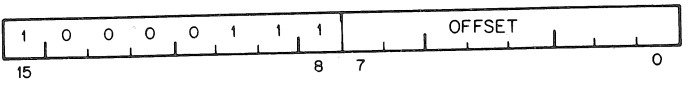


Operation: PC ← PC + (2 x offset) if C = 0

Description: Tests the state of the C-bit and causes a branch if C is clear. BCC is the complementary operation to BCS

BCS

branch if carry is set 103400 Plus offset



Operation: PC ← PC + (2 x offset) if C = 1

Description: Tests the state of the C-bit and causes a branch if C is set. It is used to test for a carry in the result of a previous operation.

Signed Conditional Branches
Particular combinations of the condition code bits are tested with the signed conditional branches. These instructions are used to test the results of instructions in which the operands were considered as signed (two's complement) values.

Note that the sense of signed comparisons differs from that of unsigned comparisons in that in signed 16-bit, two's complement arithmetic the sequence of values is as follows:

largest	077777
	077776
positive	.
	.
	.
	000001
	000000
	177777
	177776
	.
negative	.
	.
	100001
smallest	100000

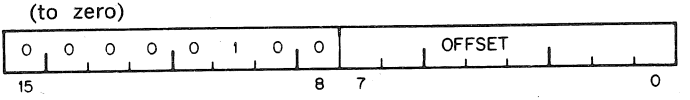
whereas in unsigned 16-bit arithmetic the sequence is considered to be

highest	177777
	.
	.
	.
	.
	.
	000002
	000001
lowest	000000

BGE

branch if greater than or equal (to zero)

002000 Plus offset

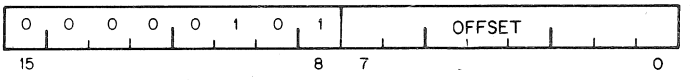


Operation: PC ← PC + (2 x offset) if N ⊻ V = 0

Description: Causes a branch if N and V are either both clear or both set. BGE is the complementary operation to BLT. Thus BGE will always cause a branch when it follows an operation that caused addition of two positive numbers. BGE will also cause a branch on a zero result.

BLT

branch if less than (zero) 002400 Plus offset

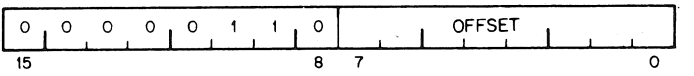


Operation: PC ← PC + (2 x offset) if N ⊻ V = 1

Description: Causes a branch if the "Exclusive Or"of the N and V bits are 1. Thus BLT will always branch following an operation that added two negative numbers, even if overflow occurred. In particular, BLT will always cause a branch if it follows a CMP instruction operating on a negative source and a positive destination (even if overflow occurred). Further, BLT will never cause a branch when it follows a CMP instruction operating on a positive source and negative destination. BLT will not cause a branch if the result of the previous operation was zero (without overflow).

BGT

branch if greater than (zero) 003000 Plus offset

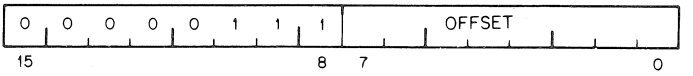


Operation: PC ← PC + (2 x offset) if Z v(N ⊻ V) = 0

Description: Operation of BGT is similar to BGE, except BGT will not cause a branch on a zero result.

BLE

branch if less than or equal (to zero) 003400 Plus offset



Operation: PC ← PC + (2 x offset) if Z v(N ⊻ V) = 1

Description: Operation is similar to BLT but in addition will cause a branch if the result of the previous operation was zero.

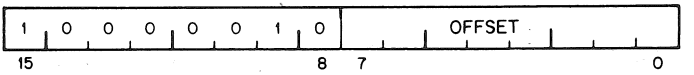
Unsigned Conditional Branches

The Unsigned Conditional Branches provide a means for testing the result of comparison operations in which the operands are considered as unsigned values.

BHI

branch if higher

101000 Plus offset



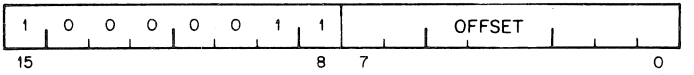
Operation: PC ← PC + (2 x offset) if C = 0 and Z = 0

Description: Causes a branch if the previous operation caused neither a carry nor a zero result. This will happen in comparison (CMP) operations as long as the source has a higher unsigned value than the destination.

BLOS

branch if lower or same

101400 Plus offset

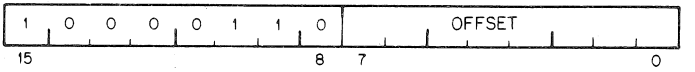


Operation: PC ← PC + (2 x offset) if C v Z = 1

Description: Causes a branch if the previous operation caused either a carry or a zero result. BLOS is the complementary operation to BHI. The branch will occur in comparison operations as long as the source is equal to, or has a lower unsigned value than the destination.

BHIS

branch if higher or same 103000 Plus offset



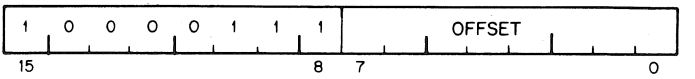
Operation: PC ← PC + (2 x offset) if C = 0

Description: BHIS is the same instruction as BCC. This mnemonic is included only for convenience.

BLO

branch if lower

103400 Plus offset

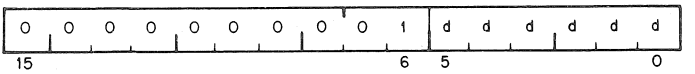


Operation: PC ← PC + (2 x offset) if C = 1

Description: BLO is same instruction as BCS. This mnemonic is included only for convenience.

JMP

jump 0001DD



Operation: PC←(dst)

Condition Codes: not affected

Description: JMP provides more flexible program branching than provided with the 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. (Program control cannot be transferred to a register.) Register 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 must therefore 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.

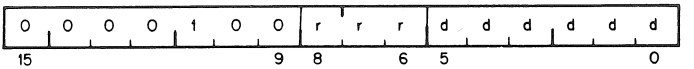
Deferred index mode JMP instructions permit transfer of control to the address contained in a selectable element of a table of dispatch vectors.

Subroutine Instructions

The subroutine call in the PDP-11 provides for automatic nesting of subroutines, reentrancy, and multiple entry points. Subroutines may call other subroutines (or indeed themselves) to any level of nesting without making special provision for storage or return addresses at each level of subroutine call. The subroutine calling mechanism does not modify any fixed location in memory, thus providing for reentrancy. This allows one copy of a subroutine to be shared among several interrupting processes. For more detailed description of subroutine programming see Chapter 5.

JSR

jump to subroutine 004RDD



Operation: ↓(SP)←reg (push reg contents onto processor stack)

reg←PC (PC holds location following JSR; this address now put in reg)

PC←(dst) (PC now points to subroutine destination)

Description: In execution of the JSR, the old contents of the specified register (the "LINKAGE POINTER") are automatically pushed onto the processor stack and new linkage information 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 saved in a reentrant manner on the processor stack execution of a subroutine may be interrupted, the same subroutine reentered 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.

A subroutine called with a JSR reg,dst instruction can access the arguments following the call with either autoincrement addressing, (reg)+, (if arguments are accessed sequentially) or by indexed addressing, X(reg), (if accessed in random order). These addressing modes may also be deferred, @(reg)+ and @X(reg) if the parameters are operand addresses rather than the operands themselves.

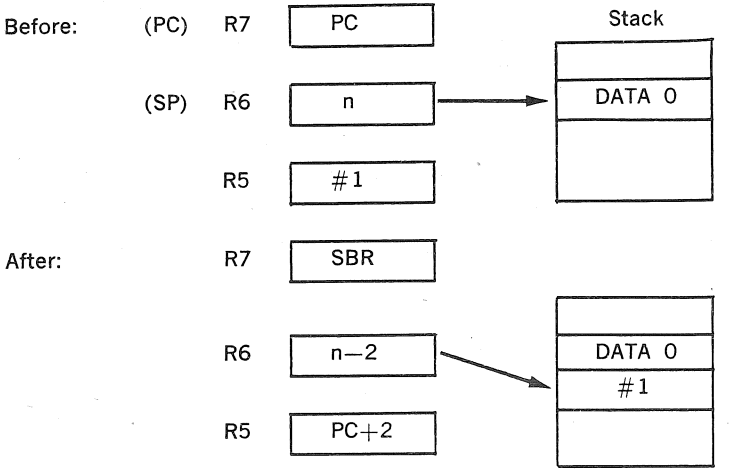
JSR PC, dst is a special case of the PDP-11 subroutine call suitable for subroutine calls that transmit parameters through the general registers. The SP and the PC are the only registers that may be modified by this call.

Another special case of the JSR instruction is JSR PC, @(SP)+ which exchanges the top element of the processor stack and the contents of the program counter. Use of this instruction allows two routines to swap program control and resume operation when recalled where they left off. Such routines are called "co-routines."

Similar to the JMP instruction, a JSR instruction with a destination mode of 0 will cause an "illegal instruction" trap since program control cannot be passed to a register. The address specified by the destination must similarly be even.

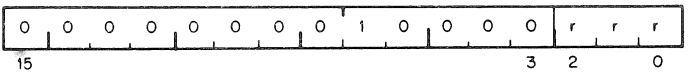
Return from a subroutine is done by the RTS instruction. RTS reg loads the contents of reg into the PC and pops the top element of the processor stack into the specified register.

Example: JSR R5, SBR



RTS

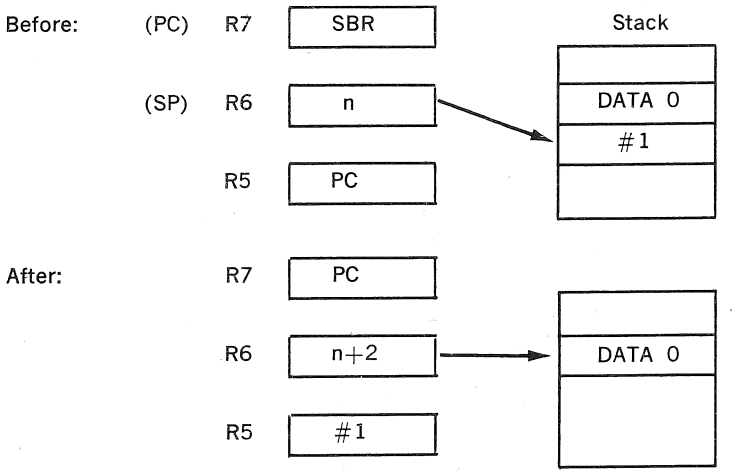
return from subroutine 00020R



Operation: PC←reg
reg←(SP)↑

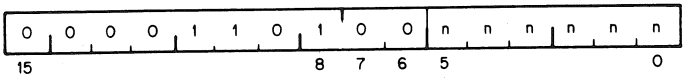
Description: Loads contents of reg into PC and pops the top element of the processor stack into the specified register.
Return from a non-reentrant subroutine is typically made through the same register that was used in its call. Thus, a subroutine called with a JSR PC, dst exits with a RTS PC and a subroutine called with a JSR R5, dst, may pick up parameters with addressing modes (R5)+, X(R5), or @X(R5) and finally exits with an RTS R5

Example: RTS R5



MARK

mark 00 64 NN



Operation: SP ← PC + 2nn nn = number of parameters
PC ← R5
R5 ← (SP)↑

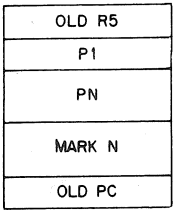
Condition Codes: unaffected

Description: 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

Example:

```
MOV   R5,-(SP)          ;place old R5 on stack
MOV   P1,-(SP)          ;place N parameters
MOV   P2,-(SP)          ;on the stack to be
                        ;used there by the
                        :subroutine
MOV   PN,-(SP)
MOV   #MARKN,-(SP)      ;places the instruction
                        ;MARK N on the stack
MOV   SP ,R5            ;set up address at Mark N in-
                         struction
JSR   PC,SUB            ;jump to subroutine
```

At this point the stack is as follows:



And the program is at the address SUB which is the beginning of the subroutine.

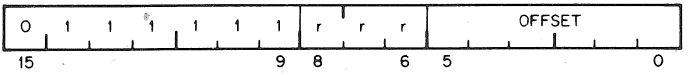
```
SUB:                    ;execution of the subroutine it-
                         self


        RTS R5          ;the return begins: this causes
```

the contents of R5 to be placed in the PC which then results in the execution of the instruction MARK N. The contents of old PC are placed in R5

MARK N causes: (1) the stack pointer to be adjusted to point to the old R5 value; (2) the value now in R5 (the old PC) to be placed in the PC; and (3) contents of the the old R5 to be popped into R5 thus completing the return from subroutine.

SOB

subtract one and branch (if ≠ 0) 077R00 Plus offset



Operation: R← R −1 if this result ≠ 0 then PC ← PC −(2 x offset)

Condition Codes: unaffected

Description: The register is decremented. If it is not equal to 0, twice the offset is subtracted from the PC (now pointing to the following word). The offset is interpreted as a sixbit positive number. This instruction provides a fast, efficient method of loop control. Assembler syntax is:

SOB R,A

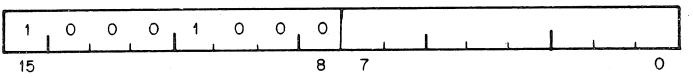
Where A is the address to which transfer is to be made if the decremented R is not equal to 0. Note that the SOB instruction can not be used to transfer control in the forward direction.

Traps

Trap instructions provide for calls to emulators, I/O monitors, debugging packages, and user-defined interpreters. A trap is effectively an interrupt generated by software. When a trap occurs the contents of the current Program Counter (PC) and Program Status Word (PS) are pushed onto the processor stack and replaced by the contents of a two-word trap vector containing a new PC and new PS. The return sequence from a trap involves executing an RTI or RTT instruction which restores the old PC and old PS by popping them from the stack. Trap vectors are located at permanently assigned fixed addresses.

EMT

emulator trap 104000—104377



Operation:

↓(SP)←PS
↓(SP)←PC
PC←(30)
PS←(32)

Condition Codes:

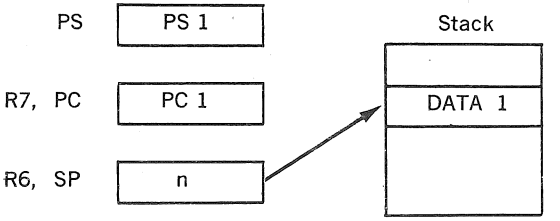
N: loaded from trap vector
Z: loaded from trap vector
V: loaded from trap vector
C: loaded from trap vector

Description:

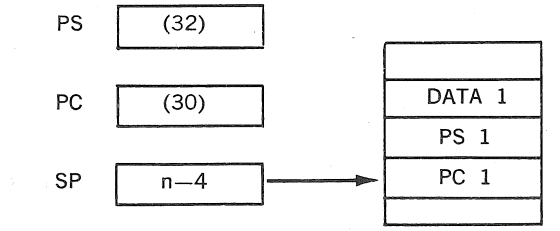
All operation codes from 104000 to 104377 are EMT instructions and may be used to transmit information to the emulating routine (e.g., function to be performed). The trap vector for EMT is at address 30. The new PC is taken from the word at address 30; the new central processor status (PS) is taken from the word at address 32.

Caution: EMT is used frequently by DEC system software and is therefore not recommended for general use.

Before:

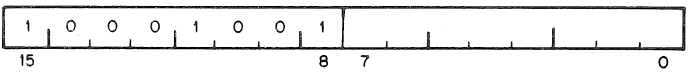


After:



TRAP

trap 104400—104777



Operation:

↓(SP)←PS
↓(SP)←PC
PC←(34)
PS←(36)

Condition Codes:

N: loaded from trap vector
Z: loaded from trap vector
V: loaded from trap vector
C: loaded from trap vector

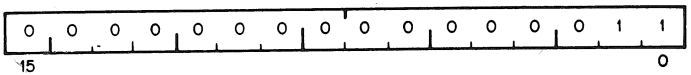
Description:

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 DEC software makes frequent use of EMT, the TRAP instruction is recommended for general use.

BPT

breakpoint trap 000003



Operation:

↓(SP)←PS
↓(SP)←PC
PC ← (14)
PS ← (16)

Condition Codes:

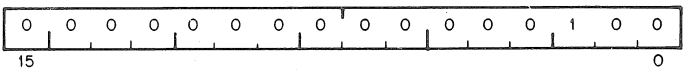
N: loaded from trap vector
Z: loaded from trap vector
V: loaded from trap vector
C: loaded from trap vector

Description:

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.)

IOT

input/output trap 000004



Operation:

↓(SP)←PS
↓(SP)←PC
PC←(20)
PS←(22)

Condition Codes:

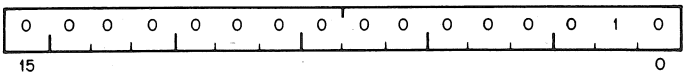
N:loaded from trap vector
Z:loaded from trap vector
V:loaded from trap vector
C:loaded from trap vector

Description:

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 is transmitted in the low byte)

RTI

return from interrupt 000002



Operation: PC←(SP)↑
PS←(SP)↑

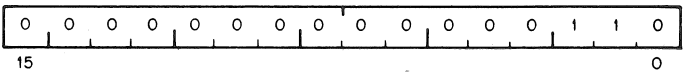
Condition Codes:
N: loaded from processor stack
Z: loaded from processor stack
V: loaded from processor stack
C: loaded from processor stack

Description: 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 is taken immediately following the RTI.

RTT

return from interrupt 000006



Operation: PC←(SP)↑
PS←(SP)↑

Condition Codes: N: loaded from processor stack
Z: loaded from processor stack
V: loaded from processor stack
C: loaded from processor stack

Description: If the RTT sets the T bit in the PS, the trace trap is not taken, and if no other trap conditions exist, the next instruction (pointed to by the PC) is executed.

Reserved Instruction Traps
These are caused by attempts to execute instruction codes reserved for future processor expansion (reserved instructions). Order codes not corresponding to any of the instructions described are considered to be reserved instructions. Reserved instruction traps occur as described under EMT, but trap through a vector at address 10.

Illegal Instruction Traps
These are caused by attempts to execute instructions with illegal addressing modes. JMP and JSR with register mode destination (mode 0) are illegal and trap as described under EMT, but through a vector at address 4.

Stack Overflow Traps
These are caused by instruction execution which depresses the SP (R6) below the vector limit of 400. They may occur during address calculation for SRC or DST modes 4 or 5 (auto-decrement modes) if the specified register is R6, during the JSR instruction (pushes PC on stack), or during any trap operation or instruction (except a trap servicing a stack overflow). Stack overflow traps occur as described under EMT, but trap through a vector at address 4. Stack overflow traps do not abort the instruction but allow it to proceed to completion.

Bus Error Traps

1. Boundary Errors - attempts to reference instructions or word operands at odd addresses.
2. Time-Out Errors - attempts to reference addresses on the bus that made no response within a certain length of time. In general, these are caused by attempts to reference non-existent memory, and attempts to reference non-existent peripheral devices.

   Bus error traps abort execution of the instruction and traps immediately following their occurrence.

Bus error traps cause processor traps through the trap vector address 4.

Trace Traps
These occur if the trace bit in the PS (bit 4) is set at the end of an instruction execution. The trace bit is not affected by references to the PS as an address or by execution of the MTPS instruction. The trace bit can only be set by data loaded into the PS during trap operations (interrupts, instruction traps, processor error traps, etc.) or execution of an RTI or RTT instruction. Execution of the RTT instruction inhibits a trace trap and thus is used to return from the trace trap routine back to the main code executed with the trace bit set. The WAIT instruction does not disable the trace bit trap.

Power Failure Trap. Trap occurs whenever the AC power drops below 95 volts or outside 47 to 63 Hertz. The instruction executes to completion and then the trap is recognized. Two milliseconds are then allowed for power down processing. Trap vector for power failure is at locations 24 and 26.

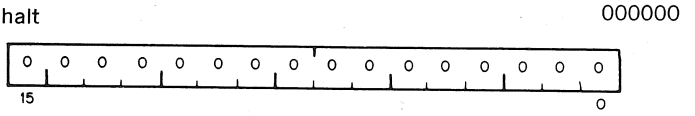
Trap Priorities

In case multiple trap conditions occur simultaneously, the following order of priorities is observed (from high to low):

1. Odd Address
2. Memory Management Violation
3. Timeout
4. Parity Error
5. Trap Instruction
6. Trace Trap
7. Stack Overflow
8. Power Fail
9. Interrupt
10. HALT From Console

If a bus error occurs during the trap process (pushing data on the stack or fetching data from a vector) the processor halts.

HALT



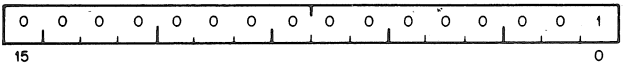
Condition Codes: not affected

Description: Causes the processor operation to cease. The console is given control of the bus. The console data lights display the contents of R0; the console address lights display the address after the halt instruction. Transfers on the UNIBUS are terminated immediately. The PC points to the next instruction to be executed. Pressing the continue key on the console causes processor operation to resume. No INIT signal is given.

Note: A halt issued in User Mode will generate a trap.

WAIT

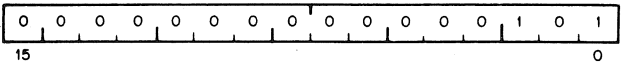
wait for interrupt 000001



Condition Codes: not affected

Description: 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 bus use by fetching instructions or operands from memory. This permits higher transfer rates between a 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 processor 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.

reset external bus 000005



Condition Codes: not affected

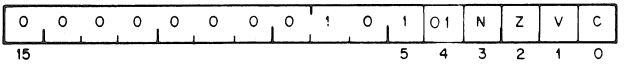
Description: Sends INIT on the UNIBUS. All devices on the UNIBUS are reset to their state at power up.

NOTE

A RESET fetched in User Mode executes as a NOP.

CLN SEN
CLZ SEZ
CLV SEV
CLC SEC
CCC SCC

condition code operators 0002XX



Description: Set and clear condition code bits. Selectable combinations of these bits may be cleared or set together. Condition code bits corresponding to bits in the condition code operator (Bits 0-3) are modified according to the sense of bit 4, the set/clear bit of the operator. i.e. set the bit specified by bit 0, 1, 2 or 3, if bit 4 is a 1. Clear corresponding bits if bit 4 = 0.

Mnemonic	Operation	OP Code
CLC	Clear C	000241
CLV	Clear V	000242
CLZ	Clear Z	000244
CLN	Clear N	000250
SEC	Set C	000261
SEV	Set V	000262
SEZ	Set Z	000264
SEN	Set N	000270
SCC	Set all CC's	000277
CCC	Clear all CC's	000257
	Clear V and C	000243
NOP	No Operation	000240

Combinations of the above set or clear operations may be ORed together to form combined instructions.

CHAPTER 5

PROGRAMMING TECHNIQUES

In order to produce programs which fully utilize the power and flexibility of the PDP-11, the reader should become familiar with the various programming techniques which are part of the basic design philosophy of the PDP-11. Although it is possible to program the PDP-11 along traditional lines such as "accumulator orientation" this approach does not fully exploit the architecture and instruction set of the PDP-11.

5.1 THE STACK

A "stack", as used on the PDP-11, is an area of memory set aside by the programmer for temporary storage or subroutine/interrupt service linkage. The instructions which facilitate "stack" handling are useful features not normally found in low-cost computers. They allow a program to dynamically establish, modify, or delete a stack and items on it. The stack uses the "last-in, first-out" concept, that is, various items may be added to a stack in sequential order and retrieved or deleted from the stack in reverse order. On the PDP-11, a stack starts at the highest location reserved for it and expands linearly downward to the lowest address as items are added to the stack.

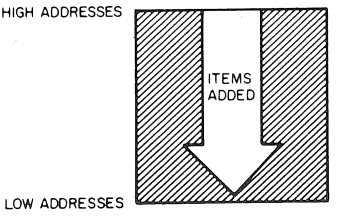


Figure 5-1: Stack Addresses

The programmer does not need to keep track of the actual locations his data is being stacked into. This is done automatically through a "stack pointer." To keep track of the last item added to the stack (or "where we are" in the stack) a General Register always contains the memory address where the last item is stored in the stack. In the PDP-11 any register except Register 7 (the Program Counter-PC) may be used as a "stack pointer" under program control; however, instructions associated with subroutine linkage and interrupt service automatically use Register 6 (R6) as a hardware "Stack Pointer." For this reason R6 is frequently referred to as the system "SP."

Stacks in the PDP-11 may be maintained in either full word or byte units. This is true for a stack pointed to by any register except R6, which must be organized in full word units only.

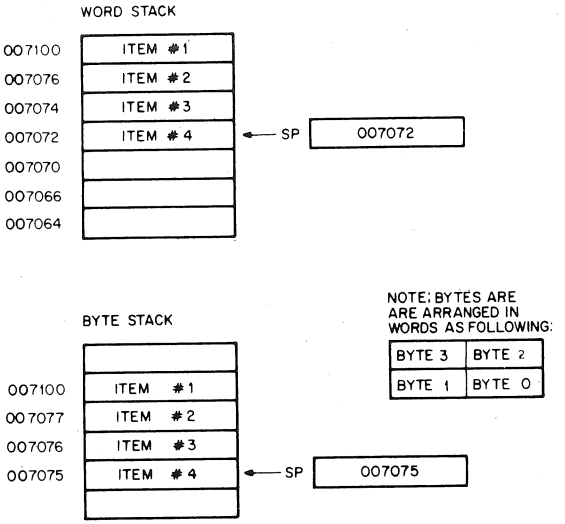


Figure 5-2: Word and Byte Stacks

Items are added to a stack using the autodecrement addressing mode with the appropriate pointer register. (See Chapter 3 for description of the autoincrement/decrement modes).

This operation is accomplished as follows;

```
MOV Source,-(SP)        ;MOV Source Word onto the stack
                        or
MOVB Source,-(SP)       ;MOVB Source Byte onto the stack
```

This is called a "push" because data is "pushed onto the stack."

To remove an item from stack the autoincrement addressing mode with the appropriate SP is employed. This is accomplished in the following manner:

```
MOV (SP)+,Destination      ;MOV Destination Word off the stack

                or

MOVB (SP)+,Destination     ;MOVB Destination Byte off the stack
```

Removing an item from a stack is called a "pop" for "popping from the stack." After an item has been "popped," its stack location is considered free and available for other use. The stack pointer points to the last-used location implying that the next (lower) location is free. Thus a stack may represent a pool of shareable temporary storage locations.

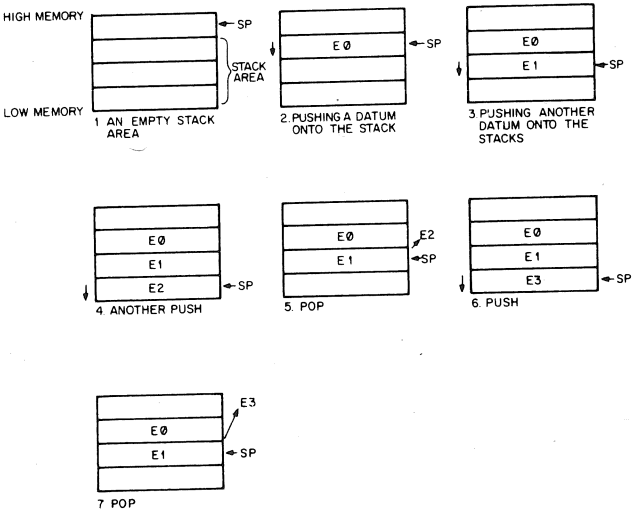


Figure 5-3: Illustration of Push and Pop Operations

As an example of stack usage consider this situation: a subroutine (SUBR) wants to use registers 1 and 2, but these registers must be returned to the calling program with their contents unchanged. The subroutine could be written as follows:

```
Address     Octal Code      Assembler Syntax

076322      010167  SUBR:   MOV R1,TEMP1 ;save R1
076324      000074          *
076326      010267          MOV R2,TEMP2 ;save R2
076330      000072          *
  .           .               .
  .           .               .
  .           .               .
  .           .               .
076410      016701          MOV TEMP1, R1 ;Restore R1
076412      000006          *
076414      016702          MOV TEMP2, R2 ;Restore R2
076416      000004          *
076420      000207          RTS PC
076422      000000          TEMP1: 0
076424      000000          TEMP2: 0
```

*Index Constants

Figure 5-4: Register Saving Without the Stack

OR: Using the Stack

```
Address     Octal Code      Assembler Syntax

010020      010143  SUBR:   MOV R1, -(R3) ;push R1
010022      010243          MOV R2, -(R3) ;push R2
  .           .               .
  .           .               .
  .           .               .
  .           .               .
010130      012301          MOV (R3)+, R2 ;pop R2
010132      012302          MOV (R3)+, R1 ;pop R1
010134      000207          RTS PC
```

Note: In this case R3 was used as a Stack Pointer

Figure 5-5: Register Saving using the Stack

The second routine uses four less words of instruction code and two words of temporary "stack" storage. Another routine could use the same stack space at some later point. Thus, the ability to share temporary storage in the form of a stack is a very economical way to save on memory usage.

As a further example of stack usage, consider the task of managing an input buffer from a terminal. As characters come in, the terminal user may wish to delete characters from his line; this is accomplished very easily by maintaining a byte stack containing the input characters. Whenever a backspace is received a character is "popped" off the stack and eliminated from consideration. In this example, a programmer has the choice of "popping" characters to be eliminated by using either the MOVB (MOVE BYTE) or INC (INCREMENT) instructions.

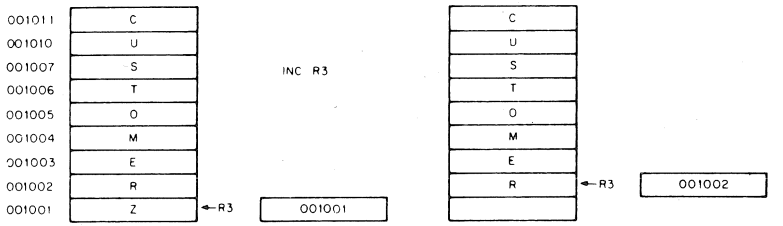


Figure 5-6: Byte Stack used as a Character Buffer

NOTE that in this case using the increment instruction (INC) is preferable to MOVB since it would accomplish the task of eliminating the unwanted character from the stack by readjusting the stack pointer without the need for a destination location. Also, the stack pointer (SP) used in this example cannot be the system stack pointer (R6) because R6 may only point to word (even) locations.

5.2 SUBROUTINE LINKAGE

5.2.1 Subroutine Calls

Subroutines provide a facility for maintaining a single copy of a given routine which can be used in a repetitive manner by other programs located anywhere else in memory. In order to provide this facility, generalized linkage methods must be established for the purpose of control transfer and information exchange between subroutines and calling programs. The PDP-11 instruction set contains several useful instructions for this purpose.

PDP-11 subroutines are called by using the JSR instruction which has the following format.

a general register (R) for linkage ——┐
JSR R,SUBR
an entry location (SUBR) for the subroutine —┘

When a JSR is executed, the contents of the linkage register are saved on the system R6 stack as if a MOV reg,-(SP) had been performed. Then the same register is loaded with the memory address following the JSR instruction (the contents of the current PC) and a jump is made to the entry location specified.

Address	Assembler Syntax	Octal Code
001000	JSR R5,SUBR	004567
001002	index constant for SUBR	000060
.	.	.
001064	SUBR: MOV A,B	01nnmm

Figure 5-7: JSR using R5

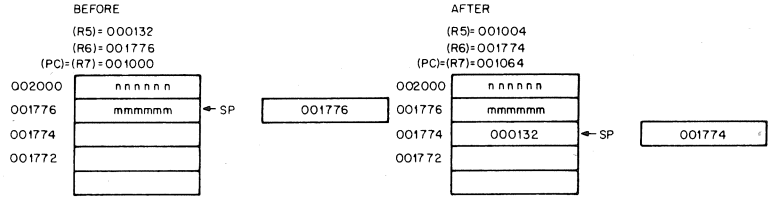


Figure 5-8: JSR

Note that the instruction JSR R6,SUBR is not normally considered to be a meaningful combination.

5.2.2 Argument Transmission

The memory location pointed to by the linkage register of the JSR instruction may contain arguments or addressses of arguments. These arguments may be accessed from the subroutine in several ways. Using Register 5 as the linkage register, the first argument could be obtained by using the addressing modes indicated by (R5), (R5)+,X(R5) for actual data, or @(R5)+, etc. for the address of data. If the autoincrement mode is used, the linkage register is automatically updated to point to the next argument.

Figures 5-9 and 5-10 illustrate two possible methods of argument transmission.

```
Address Instructions and Data

010400          JSR R5,SUBR
010402          Index constant for SUBR        SUBROUTINE CALL
010404          arg #1                         ARGUMENTS
010406          arg #2
.               .
.               .
.               .
.               .
.               .
020306  SUBR:   MOV (R5)+,R1          ;get arg #1
020310          MOV (R5)+,R2          ;get arg #2 Retrieve Arguments
                                      from SUB
```

Figure 5-9; Argument Transmission -Register Autoincrement Mode

```
Address   Instructions and Data

010400    JSR R5,SUBR
010402    index constant for SUBR     SUBROUTINE CALL
010404    077722                      Address of Arg #1
010406    077724                      Address of Arg. #2
010410    077726                      Address of Arg. #3
.         .                           .
.         .                           .
.         .                           .
077722    Arg #1
077724    arg #2                      arguments
077726    arg #3
.         .                           .
.         .                           .
.         .                           .
020306    SUBR:   MOV @(R5)+,R1  ;get arg #1
020301            MOV @(R5)+,R2  ;get arg #2
```

Figure 5-10: Argument Transmission-Register Autoincrement Deferred Mode

Another method of transmitting arguments is to transmit only the address of the first item by placing this address in a general purpose register. It is not necessary to have the actual argument list in the same general area as the subroutine call. Thus a subroutine can be called to work on data located anywhere in memory. In fact, in many cases, the operations performed by the subroutine can be applied directly to the data located on or pointed to by a stack without the need to ever actually move this data into the subroutine area.

```
Calling Program: MOV     POINTER, R1
                 JSR     PC,SUBR

SUBROUTINE       ADD     (R1)+,(R1)    ;Add item #1 to item #2, place
                                        result in item #2, R1 points
                                        to item #2 now
                         etc.
                         or

                 ADD     (R1),2(R1)    ;Same effect as above except that
                                        R1 still points to item #1
                                                        etc.
```

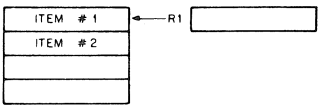


Figure 5-11: Transmitting Stacks as Arguments

Because the PDP-11 hardware already uses general purpose register R6 to point to a stack for saving and restoring PC and PS (processor status word) information, it is quite convenient to use this same stack to save and restore intermediate results and to transmit arguments to and from subroutines. Using R6 in this manner permits extreme flexibility in nesting subroutines and interrupt service routines.

Since arguments may be obtained from the stack by using some form of register indexed addressing, it is sometimes useful to save a temporary copy of R6 in some other register which has already been saved at the beginning of a subroutine. In the previous example R5 may be used to index the arguments while R6 is free to be incremented and decremented in the course of being used as a stack pointer. If R6 had been used directly as the base for indexing and not "copied", it might be difficult to keep track of the position in the argument list since the base of the stack would change with every autoincrement/decrement which occurs.

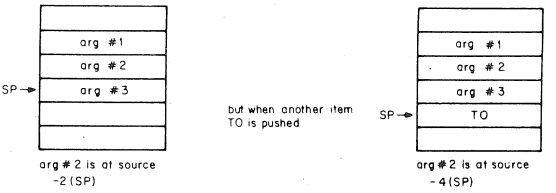


Figure 5-12: Shifting Indexed Base

However, if the contents of R6 (SP) are saved in R5 before any arguments are pushed onto the stack, the position relative to R5 would remain constant.

Figure 5-13: Constant Index Base Using "R6 Copy"

5.2.3 Subroutine Return

In order to provide for a return from a subroutine to the calling program an RTS instruction is executed by the subroutine. This instruction should specify the same register as the JSR used in the subroutine call. When executed, it causes the register specified to be moved to the PC and the top of the stack to be then placed in the register specified. Note that if an RTS PC is executed, it has the effect of returning to the address specified on the top of the stack.

Note that the JSR and the JMP Instructions differ in that a linkage register is always used with a JSR; there is no linkage register with a JMP and no way to return to the calling program.

When a subroutine finishes, it is necessary to "clean-up" the stack by eliminating or skipping over the subroutine arguments. One way this can be done is by insisting that the subroutine keep the number of arguments as its first stack item. Returns from subroutines would then involve calculating the amount by which to reset the stack pointer, resetting the stack pointer, then restoring the original contents of the register which was used as the copy of the stack pointer. The PDP-11/40, however, has a much faster and simpler method of performing these tasks. The MARK instruction which is stored on a stack in place of "number of argument" information may be used to automatically perform these "clean-up" chores.

5.2.4 PDP-11 Subroutine Advantages

There are several advantages to the PDP-11 subroutine calling procedure.

a. arguments can be quickly passed between the calling program and the subroutine.

b. if the user has no arguments or the arguments are in a general register or on the stack the JSR PC,DST mode can be used so that none of the general purpose registers are taken up for linkage.

c. many JSR's can be executed without the need to provide any saving procedure for the linkage information since all linkage information is automatically pushed onto the stack in sequential order. Returns can simply be made by automatically popping this information from the stack in the opposite order of the JSR's.

Such linkage address bookkeeping is called automatic "nesting" of subroutine calls. This feature enables the programmer to construct fast, efficient linkages in a simple, flexible manner. It even permits a routine to call itself in those cases where this is meaningful. Other ramifications will appear after we examine the PDP-11 interrrupt procedures.

5.3 INTERRUPTS

5.3.1 General Principles

Interrupts are in many respects very similar to subroutine calls. However, they are forced, rather than controlled, transfers of program execution occurring because of some external and program-independent event (such as a stroke on the teleprinter keyboard). Like subroutines, interrupts have linkage information such

that a return to the interrupted program can be made. More information is actually necessary for an interrupt transfer than a subroutine transfer because of the random nature of interrupts. The complete machine state of the program immediately prior to the occurrence of the interrupt must be preserved in order to return to the program without any noticeable effects. (i.e. was the previous operation zero or negative, etc.) This information is stored in the Processor Status Word (PS). Upon interrupt, the contents of the Program Counter (PC) (address of next instruction) and the PS are automatically pushed onto the R6 system stack. The effect is the same as if:

```
MOV PS ,-(SP)          ; Push PS
MOV R7,-(SP)           ; Push PC
```

had been executed.

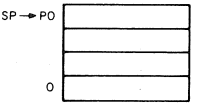
The new contents of the PC and PS are loaded from two preassigned consecutive memory locations which are called an "interrupt vector". The actual locations are chosen by the device interface designer and are located in low memory addresses of Kernel virtual space (see interrupt vector list, Appendix B). The first word contains the interrupt service routine address (the address of the new program sequence) and the second word contains the new PS which will determine the machine status including the operational mode and register set to be used by the interrupt service routine. The contents of the interrupt service vector are set under program control.

After the interrupt service routine has been completed, an RTI (return from interrupt) is performed. The two top words of the stack are automatically "popped" and placed in the PC and PS respectively, thus resuming the interrupted program.

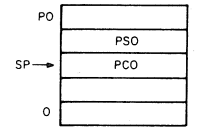
5.3.2 Nesting

Interrupts can be nested in much the same manner that subroutines are nested. In fact, it is possible to nest any arbitrary mixture of subroutines and interrupts without any confusion. By using the RTI and RTS instructions, respectively, the proper returns are automatic.

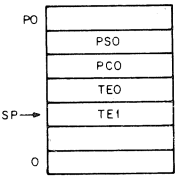
1. Process 0 is running; SP is pointing to location P0.



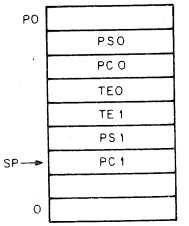
2. Interrupt stops process 0 with PC = PC0, and status = PS0 ;starts process 1.



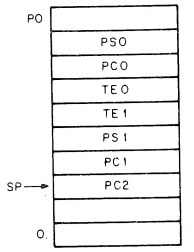
3. Process 1 uses stack for temporary storage (TE0, TE1).



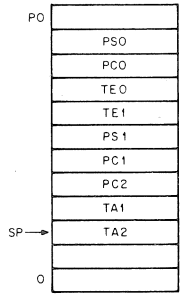
4. Process 1 interrupted with PC = PC1 and status = PS1; process 2 is started



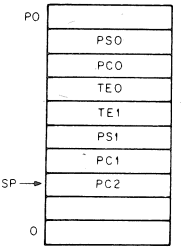
5. Process 2 is running and does a JSR R7,A to Subroutine A with PC = PC2.



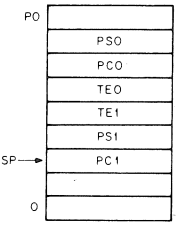
6. Subroutine A is running and uses stack for temporary storage.



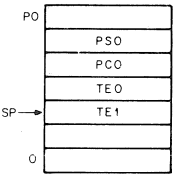
7. Subroutine A releases the temporary storage holding TA1 and TA2.



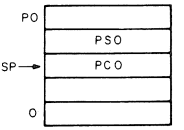
8. Subroutine A returns control to process 2 with an RTS R7,PC is reset to PC2.



9. Process 2 completes with an RTI instruction (dismisses interrupt) PC is reset to PC(1) and status is reset to PS1; process 1 resumes.



10. Process 1 releases the temporary storage holding TE0 and TE1.



11. Process 1 completes its operation with an RTI PC is reset to PC0 and status is reset to PS0.

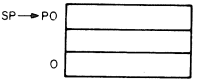


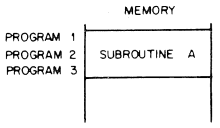
Figure 5-14: Nested Interrupt Service Routines and Subroutines

Note that the area of interrupt service programming is intimately involved with the concept of CPU and device priority levels.

5.4 REENTRANCY

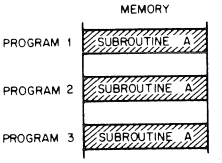
Further advantages of stack organization become apparent in complex situations which can arise in program systems that are engaged in the concurrent handling of several tasks. Such multi-task program environments may range from relatively simple single-user applications which must manage an intermix of I/O interrupt service and background computation to large complex multi-programming systems which manage a very intricate mixture of executive and multi-user programming situations. In all these applications there is a need for flexibility and time/memory economy. The use of the stack provides this economy and flexibility by providing a method for allowing many tasks to use a single copy of the same routine and a simple, unambiguous method for keeping track of complex program linkages.

The ability to share a single copy of a given program among users or tasks is called reentrancy. Reentrant program routines differ from ordinary subroutines in that it is unnecessary for reentrant routines to finish processing a given task before they can be used by another task. Multiple tasks can be in various stages of completion in the same routine at any time. Thus the following situation may occur:



PDP-11 Approach

Programs 1, 2, and 3 can share Subroutine A.



Conventional Approach

A separate copy of Subroutine A must be provided for each program.

Figure 5-15: Reentrant Routines

The chief programming distinction between a non-shareable routine and a reentrant routine is that the reentrant routine is composed solely of "pure code", i.e. it contains only instructions and constants. Thus, a section of program code is reentrant (shareable) if and only if it is "non self-modifying", that is it contains no information within it that is subject to modification.

Using reentrant routines, control of a given routine may be shared as illustrated in Figure 5-16.

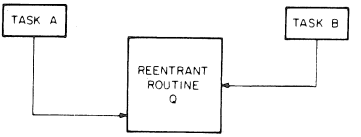


Figure 5-16: Reentrant Routine Sharing

1. Task A has requested processing by Reentrant Routine Q.
2. Task A temporarily relinquishes control (is interrupted) of Reentrant Routine Q before it finishes processing.
3. Task B starts processing in the same copy of Reentrant Routine Q.
4. Task B relinquishes control of Reentrant Routine Q at some point in its processing.
5. Task A regains control of Reentrant Routine Q and resumes processing from where it stopped.

The use of reentrant programming allows many tasks to share frequently used routines such as device interrupt service routines, ASCII-Binary conversion routines, etc. In fact, in a multi-user system it is possible for instance, to construct a reentrant FORTRAN compiler which can be used as a single copy by many user programs.

As an application of reentrant (shareable) code, consider a data processing program which is interrupted while executing a ASCII-to-Binary subroutine which has been written as a reentrant routine. The same conversion routine is used by the device service routine. When the device servicing is finished, a return from interrupt (RTI) is executed and execution for the processing program is then resumed where it left off inside the same ASCII-to-Binary subroutine.

Shareable routines generally result in great memory saving. It is the hardware implemented stack facility of the PDP-11 that makes shareable or reentrant routines reasonable.

A subroutine may be reentered by a new task before its completion by the previous task as long as the new execution does not destroy any linkage information or intermediate results which belong to the previous programs. This usually amounts to saving the contents of any general purpose registers, to be used and restoring them upon exit. The choice of whether to save and restore this information in the calling program or the subroutine is quite arbitrary and depends on the particular application. For example in controlled transfer situations (i.e. JSR's) a main program which calls a code-conversion utility might save the contents of registers which it needs and restore them after it has regained control, or the code conversion routine might save the contents of registers which it uses and restore them upon its completion. In the case of interrupt service routines this save/restore process must be carried out by the service routine itself since the interrupted program has no warning of an impending interrupt. The advantage of

using the stack to save and restore (i.e. "push" and "pop") this information is that it permits a program to isolate its instructions and data and thus maintain its reentrancy.

In the case of a reentrant program which is used in a multi-programming environment it is usually necessary to maintain a separate R6 stack for each user although each such stack would be shared by all the tasks of a given user. For example, if a reentrant FORTRAN compiler is to be shared between many users, each time the user is changed, R6 would be set to point to a new user's stack area as illustrated in Figure 5-17.

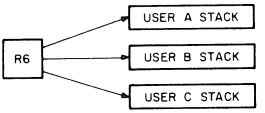


Figure 5-17: Multiple R6 Stack

5.5 POSITION INDEPENDENT CODE - PIC

Most programs are written with some direct references to specific addresses, if only as an offset from an absolute address origin. When it is desired to relocate these programs in memory, it is necessary to change the address references and/or the origin assignments. Such programs are constrained to a specifiec set of locations. However, the PDP-11 architecture permits programs to be constructed such that they are not constrained to specific locations. These Position Independent programs do not directly reference any absolute locations in memory. Instead all references are "PC-relative" i.e. locations are referenced im terms of offsets from the current location (offsets from the current value of the Program Counter (PC)). When such a program has been translated to machine code it will form a program module which can be loaded anywhere in memory as required.

Position Independent Code is exceedingly valuable for those utility routines which may be disk-resident and are subject to loading in a dynamically changing program environment. The supervisory program may load them anywhere it determines without the need for any relocation parameters since all items remain in the same positions relative to each other (and thus also to the PC).

Linkages to program routines which have been written in position independent code (PIC) must still be absolute in some manner. Since these routines can be located anywhere in memory there must be some fixed or readily locatable linkage addresses to facilitate access to these routines. This linkage address may be a simple pointer located at a fixed address or it may be a complex vector composed of numerous linkage information items.

5.6 CO-ROUTINES

In some situations it happens that two program routines are highly interactive. Using a special case of the JSR instruction i.e. JSR PC,@(R6)+ which exchanges the top element of the Register 6 processor stack and the contents of the Program Counter (PC), two routines may be permitted to swap program control and resume operation where they stopped, when recalled. Such routines are called "co-routines". This control swapping is illustrated in Figure 5-18.

Routine #1 is operating, it then executes:

MOV #PC2,-(R6)

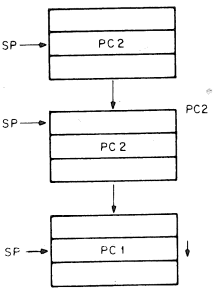
JSR PC,@(R6)+

with the following results:

1) PC2 is popped from the stack and the SP autoincremented

2) SP is autodecremented and the old PC (i.e. PC1) is pushed

3) control is transferred to the location PC2 (i.e. routine #2)



Routine #2 is operating, it then executes:

JSR PC ,@(R6)+

with the result the PC2 is exchanged for PC1 on the stack and control is transferred back to routine #1.

Figure 5-18 - Co-Routine Interaction

5.7 PROCESSOR TRAPS

There are a series of errors and programming conditions which will cause the Central Processor to trap to a set of fixed locations. These include Power Failure, Odd Addressing Errors, Stack Errors, Timeout Errors, Memory Parity Errors, Memory Management Violations, Use of Reserved Instructions, Use of the T bit in the Processor Status Word, and use of the IOT, EMT, and TRAP instructions.

5.7.1 Power Failure

Whenever AC power drops below 95 volts for 115v power (190 volts for 230v) or outside a limit of 47 to 63 Hz, the instruction executes to completion and the power fail sequence is initiated. The Central Processor automatically traps to location 24 and the power fail program has 2 msec. to save all volatile information (data in registers), and condition peripherals for power fail.

When power is restored the processor fetches its PC and PS from locations 24 and 26 and executes the power up routine to restore the machine to its state prior to power failure.

5.7.2 Odd Addressing Errors

This error occurs whenever a program attempts to execute a word instruction on an odd address (in the middle of a word boundary). The instruction is aborted and the CPU traps through location 4.

5.7.3 Time-out Errors

These errors occur when a Master Synchronization pulse is placed on the UNIBUS and there is no slave pulse within a certain length of time. This error usually occurs in attempts to address non-existent memory or peripherals.

The offending instruction is aborted and the processor traps through location 4.

5.7.4 Reserved Instructions

There is a set of reserved instructions which cause the processor to trap through location 10.

5.7.5 Illegal Instruction

JMP and JSR with DST mode of 0 are considered illegal addressing modes and trap through location 4.

5.7.6 Trap Handling

When a trap occurs, the processor saves the PC and PS on the Processor Stack and begins to execute the trap routine pointed to by the trap vector.

CHAPTER 6

THE PDP-11/34 COMPUTER

6.1 DESCRIPTION

The PDP-11/34 computer system can contain up to 124K words of parity MOS or core memory. The mounting assembly for the central processor is available in 3 sizes. Chassis heights of 5¼", 10½", or 21" allow the user to optimize space utilization for the particular application.

The basic PDP-11/34 includes the following capabilities and equipment:

- Central processor
- Parity memory (MOS or core)
- Automatic bootstrap loader program in ROM memory
- Operator's console
- Self-test diagnostics
- Memory management, relocation and protection
- Extended instruction set (EIS)

Optional equipment includes:

- Serial line interface and clock
- Console terminal
- Programmer's console
- Battery backup unit for MOS memory
- Standard PDP-11 peripherals

Extended Instruction Set

The Extended Instruction Set (EIS) provides the capability of performing hardware fixed point arithmetic and allows direct implementation of multiply, divide, and multiple shifting. A double-precision 32-bit word can be handled. The Extended Instruction Set executes compatibly with the EIS available on the PDP-11/35 and 11/40. Refer to Section 6.10.

Memory Management

Memory Management is an advanced memory extension, relocation, and protection feature which will:

- Extend memory space from 28K to 124K words
- Allow efficient segmentation of core for multi-user environments
- Provide effective protection of memory segments in multi-user environments

Memory Management in the PDP-11/34 is totally compatible with the Memory Management (KT11-D) option on the PDP-11/35 and 11/40.

The machine operates in two modes; Kernel and User. When the machine is in Kernel mode a program has complete control of the machine;

when in User mode the processor is inhibited from executing certain instructions and can be denied direct access to the peripherals or other protected memory locations in the system. This hardware feature can be used to provide complete executive protection in a multi-programming environment. A software operating system can insure that no user (who is operating in User mode) can cause a failure (crash) of the entire system.

Refer to Chapter 7 for a detailed description of the Memory Management unit.

6.2 SPECIFICATIONS

Computer	PDP-11/34		
Main Market	End User & OEM		
Memory			
Max size:	124K words		
Type:	core or MOS		
Parity:	standard		
Central Processor			
Instructions:	basic set + XOR, SOB, MARK, SXT, RTT, MFPS, MTPS EIS set: (MUL, ASH, DIV, ASHC) mem mgt set: (MFPI, MTPI, MFPD, MTPD)		
Programming modes:	user & kernel		
No. of general registers:	8		
Auto hardware interrupts:	yes		
Auto software interrupts:	no		
Power fail/auto restart:	yes		
Mechanical & Environmental			
Chassis height:	5¼"	10½"	21"
Weight:	45 lbs	110 lbs	200 lbs.
Input power:	350W	700W	1000W
	115 VAC, nom. (90 to 132v), 50/60 Hz, or 230 VAC, nom. (180 to 264v), 50/60 Hz		
Operating temperature:	5°C to 50°C		
Relative humidity:	10% to 95%, non-condensing		
Equipment			
I/O serial interface:	optional		
Line frequency clock:	optional		
Console terminal:	optional		
Operators console:	standard		
Programmer's console:	optional		
Hardware bootstrap:	standard		
Extended arithmetic:	standard		
Autodiagnostics:	standard		

Floating point:	currently not available
Stack limit address:	fixed (at 400)
Memory management:	standard
Cabinet:	optional with 5¼" and 10½" units; standard with 21" units

6.2.1 Processor Backplane Configuration

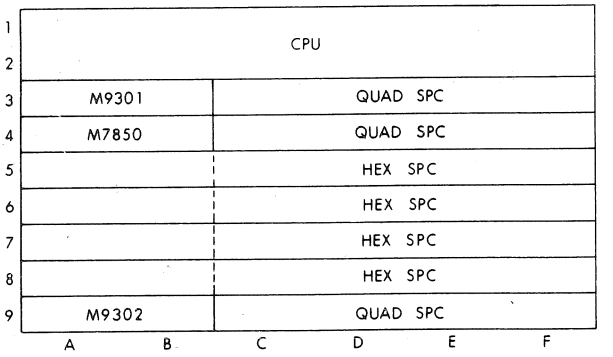
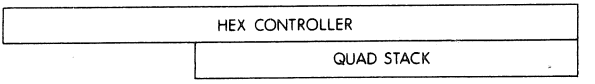


Figure 6-1 Processor Backplane

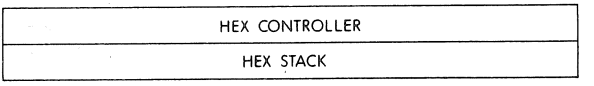
The processor backplane consists of a double system unit (SU) comprising 9 Hex slots. All PDP-11/34 systems contain the CPU, M9301 Bootstrap/Terminator, M7850 parity control, and M9302 (or a UNIBUS jumper to the next SU) as shown in Figure 6-1. Memory is added as follows depending on whether the system uses core or MOS.

Core: Core memory is available in two size increments, 8K and 16K words.

The 8K core, designated MM11-C, consists of a Hex and Quad module as follows:



The 16K core, designated MM11-D, consists of 2 Hex modules as follows:



MOS: MOS memory is available in 8 or 16K increments and all increments consist of a single Hex module.

8 and 16K increments are designated MS11-F, and MS11-J.

NOTE

The M7850 parity control may be moved to slot 5 to optimize usage of the MM11-C memory in slots 4 and 5.

The following backpanel configurations comprise the basic PDP-11/34 computer.

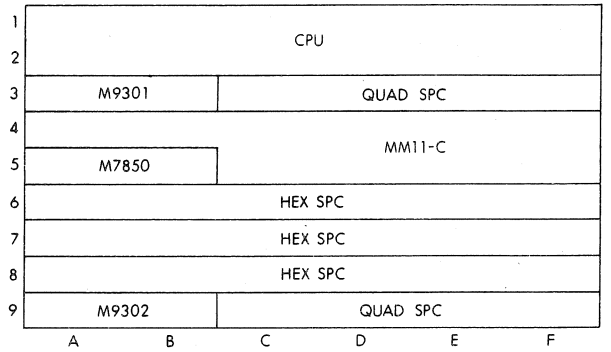


Figure 6-2 8K Core using MM11-C

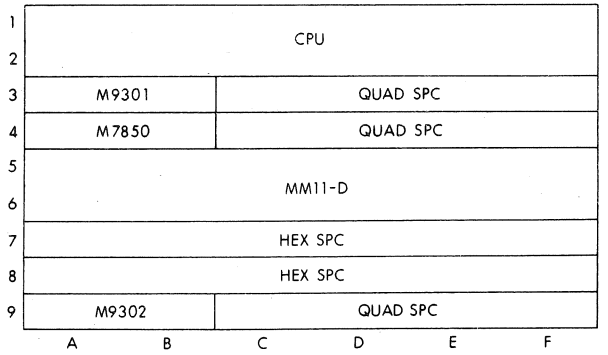


Figure 6-3 16K Core using MM11-D

Additional memory or Quad and Hex SPC options (DL11-W, TA11 controller, RX11 controller, etc.) may be added to the processor backplane as space allows.

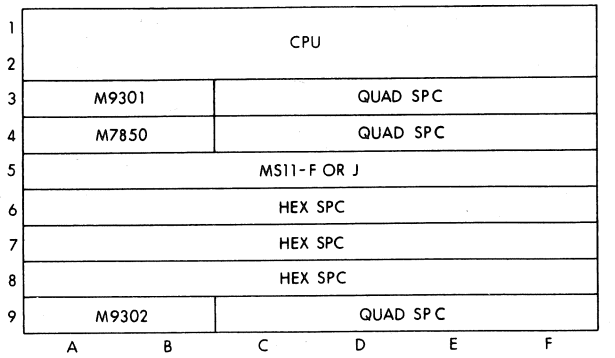


Figure 6-4 8 or 16K MOS using MS11-F or J

6.2.2 Chassis Configuration

5¼" Chassis—the previously described processor backpanel is 5¼" high and fills the 5¼" chassis. Further expansion must occur by adding an additional chassis or converting to a 10½" or 21" chassis.

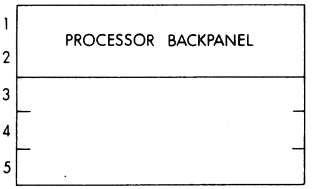


Figure 6-5 PDP-11/34 back panel in BALL-K (10½" chassis)

The BA11-K 10½" chassis contains mounting room for 5 system units. The processor backpanel occupies the first two, leaving 3 SU's for further expansion.

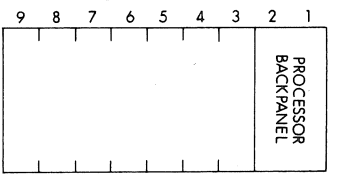


Figure 6-6 PDP-11/34 backpanel in BA11-F (21" chassis)

The BA11-F 21" chassis contains mounting room for 9 system units. The processor backpanel occupies the first two, leaving 7 SU's for further expansion. For large memory configurations, it may be desirable to utilize memories which do not mount in the processor backpanel but are more power efficient. This leaves more expansion space in the processor backpanel and uses system units in the BA11-F.

6.3 MOS & CORE MEMORY

The PDP-11/34 is available with both MOS and core memory. The two types of memory may be freely intermixed in the computer system; the difference in timing is accommodated by the architecture of the asynchronous UNIBUS.

Parity

All main memory in a PDP-11/34 system contains parity to enhance system integrity. Parity is generated and checked on all references between the CPU and memory, and any parity errors are flagged for resolution under program control. Odd parity is used, with 1 parity bit per 8-bit byte, for a total of 18 bits per word.

A double height module, M7850, contains parity control logic. Its control & status register (CSR) address is selectable between 772 100 and 772 136.

The CSR captures the high order address bits of a memory location with a parity error. A single M7850 provides parity generation and detection logic for all memory mounted in its back panel.

MOS

The basic unit of MOS memory, MS11-JP, contains 16K words of parity MOS memory. Each 16K words of MOS requires 1 hex mounting space.

Core

The basic unit of core memory, MM11-DP, contains 16K words of parity core memory. Each 16K words of core memory requires 2 hex mounting spaces.

6.4 BATTERY BACKUP

Core memory is non-volatile; the contents are preserved when power is removed. However, MOS memory is volatile. If power is interrupted, an auxiliary power supply must be provided if information in the memory is to be saved. With the 5½" and 10½" CPU assemblies there is an optional Battery Backup Unit that can preserve the contents of 32K words of MOS memory for about 2 hours. This auxiliary power unit is a battery that is charged up by the main AC power when the computer system is operating normally. In this normal mode, the battery backup has no effect on the MOS memory. But if power is interrupted, voltage sensing circuitry within the backup option will automatically cause the MOS to be powered from this auxiliary power. The MOS information will be retained by being refreshed at a low cycle rate, thereby using minimum power.

6.5 M9301 MODULE

The M9301 module, which is included with the PDP-11/34, provides 4 functions for the computer system.

1. It contains a read-only memory (ROM) that holds diagnostic routines for verifying computer operation.
2. It contains, also in ROM, the several bootstrap loader programs for starting up the system.
3. It contains the Console Emulator Routine in ROM for issuing console commands from the terminal.
4. It provides termination resistors for the UNIBUS.

There are 2 versions of the M9301 module available:

	M9301-YA	M9301-YB
Main user	OEM	End User
Able to run secondary bootstrap program directly upon power up or reboot	yes*	no
Automatic entry to Console Emulator Routine	yes*	yes
Needs an ASCII terminal	no	yes

* Selection of one of these 2 operations is made by setting of switches contained on the module.

Diagnostics
Both versions of the M9301 contain diagnostics to check both the processor and memory in a Go/No-Go mode. Execution of the diagnostics occur automatically but may be disabled by switches on the M9301.

Bootstrap Loader
The M9301-YA contains independent bootstrap programs that can bootstrap programs into memory from a selected peripheral device. Through front panel control or following Power Up, the computer can directly execute a bootstrap, without the operator having to manually key in the initial program. The bootstrap program for the peripheral device is determined by switches on the M9301. This is useful in remote applications where no operator is present.

The M9301-YB, after execution of the CPU diagnostics, turns control of the system to the user at the console terminal. The system prints out status information and is ready to accept simple user commands for checking or modifying information within the computer, starting a program already in memory, or executing a device bootstrap.

The inclusion of a bootstrap loader in non-destructible read-only memory is a tremendous convenience in system operation. Bootstrap programs do not have to be manually loaded into the computer for system initialization.

Console Emulation
The normal console functions traditionally performed through front panel switches can be obtained by typing simple commands on the console terminal. LOAD, EXAMINE, DEPOSIT, START, and BOOT functions are available.

Termination
The M9301 contains resistors for proper impedance termination at the beginning of the UNIBUS (transmission line).

6.6 M9302 MODULE

The M9302 provides resistors for proper termination of the UNIBUS. It also contains logic which detects the assertion of certain UNIBUS signals and responds to them. Devices which request transfers on the UNIBUS receive and stop a serially passed "request granted" signal from

the processor. If this signal ever reaches the end of the UNIBUS, no device along the serial chain stopped it. The M9302 receives all such unheeded grants and responds to allow the CPU to proceed.

6.7 DL11-W (M7856)

The DL11-W option provides 2 capabilities:

1. Serial line interface to an ASCII terminal, such as an LA36 DECwriter, VT50 video terminal, or an LT33 Teletype.
2. Line time clock.

Serial Communication Line Interface

The interface is program compatible with the standard DIGITAL serial interfaces, DL11-A,-B,-C, and -D. It can handle speeds from 110 to 9600 baud. It provides serial-to-parallel (and vice-versa) data conversion.

Line Clock

The clock is program compatible with the KW11-L, the standard line clock option used with other PDP-11 computers. The clock senses the 50 or 60 Hz line frequency for internal timing.

There are switches on the module for selection of parameters such as:

register addresses
baud rate
communications data formats

6.8 OPERATOR'S CONSOLE

The operator's console is the front panel link between the user and the computer. It contains a minimum number of switches and lights. All normally used console functions are available through the combination of the operator's console and an ASCII terminal; e.g. LA36 DECwriter.

Console Switches

POWER	OFF	DC power to the computer is off.
	ON	Power is applied to the computer (and the system).
	STNBY	Standby; no DC power to the computer, but DC power is applied to MOS memory (to retain data) and the fans remain on.

CAUTION

AC power is removed only by disconnecting the line cord.

CONT/HALT	CONT	The program is allowed to continue.
	HALT	The program is stopped.
BOOT/INIT	INIT	The switch is spring returned to the BOOT position. When the switch is depressed to INITialize and then returned to BOOT, the operation depends on the setting of the CONT/HALT switch.

HALT: The processor only is initialized and no "UNIBUS INIT" is generated. Upon lifting the CONT/HALT switch, the M9301 routine is executed allowing examination of system peripherals without clearing their contents with "UNIBUS INIT".

CONT: Initialize and then execute the M9301 program.

When the BOOT switch is released, the following action takes place:

(a) For both M9301-YA and M9301-YB:
(when the switches are set for this operation)

1. Run basic CPU diagnostics.
2. Print out (on the console terminal) contents of R0, R4, SP, and PC at the time of power up, followed by a dollar sign ($) on the next line.
3. Enter Console Emulator Routine, awaiting keyboard commands.
4. When a device bootstrap command is issued, first run processor memory diagnostics, then execute secondary bootstrap program from the designated peripheral device.

(b) For the M9301-YA (OEM) version only:
(when M9301-YA switches are set for this operation)

1. Run basic CPU diagnostics.
2. Run memory diagnostics.
3. Run secondary bootstrap program from the preselected peripheral device.

NOTE

When utilizing the stand alone switch setting described as alternative (b) above, the switches must be reset to enable execution of the console emulator routine.

Indicators

BATT	off	Battery voltage is below minimum level to maintain MOS contents, or battery is absent.
	slow flash (1 flash/2 sec)	Battery is charging, but voltage is above the minimum level to maintain MOS contents if power is removed.
	fast flash (10 flashes/sec)	Primary power has been lost; battery is discharging, but MOS memory contents are being maintained, and voltage is still above minimum limit.
	continuous on	Battery is fully charged and present.
DC ON	on	DC power is applied to logic circuitry.
	off	DC power is off.

RUN	on	A program is running.
	off	The program is stopped.

6.9 CONSOLE EMULATION

The M9301 module contains a console emulator routine. When this routine is used in conjunction with the user's terminal, functions quite similar to those found on the programmer's console of traditional PDP-11 family computers are generated.

Summary of the Console Emulator Functions

LOAD — This function loads the address to be manipulated into the system.

EXAMINE — Allows the operator to examine the contents of the address that was loaded and/or deposited.

DEPOSIT — Allows the operator to write into the address that was loaded and/or examined.

START — Initializes the system and starts execution of the program at the address loaded.

BOOT — Allows the booting of a specified device by typing in a two character code and optional unit number.

Console Emulator Operation

The console emulator allows the user to perform LOAD, EXAMINE, DEPOSIT, START, and BOOT functions by typing in the appropriate code on the keyboard.

Entry Into the Console Emulator

There are three ways of entering the Console Emulator:

- Move the Power Switch to the On position.
- Depress the BOOT Switch.
- Automatic entry on return from a power failure.

After the Console Emulator Routine has started and the basic CPU diagnostics have all run successfully, a series of numbers representing the contents of R0, R4, SP and PC respectively, will be printed by the terminal. This sequence will be followed by a $ on the next line.

Example—a typical printout on power up:

XXXXXX	XXXXXX	XXXXXX	XXXXXX
$			
R0	R4	R6	PC
		STACK	PROGRAM
PROMPT		POINTER	COUNTER
CHARACTER		(SP)	

Notes: X signifies an octal number (0-7).

Whenever there is a power up routine, or the BOOT switch is released from the INIT position, the PC at this time will be stored. The stored value is printed out as above (noted as the PC).

Using the Console Emulator

After the $—Once the system has been powered up or booted, and R0, R4, SP, PC and $ have been printed, the Console Emulator routine can be used.

Keyboard Input Symbols—The discussion of keyboard input format uses the following symbols:

- Space Bar: (SB)
- Carriage Return Key: (CR)
- Any number 0-7 (Octal Number) Key: (X)

Keyboard INPUT Format—Load, examine, deposit, start. All character keys shown in the following discussion represent themselves with the exception of those in parentheses.

FUNCTION								
Load address	L (SB)	(X)	(X)	(X)	(X)	(X)	(X)	(CR)
Examine	E (SB)							
Deposit	D (SB)	(X)	(X)	(X)	(X)	(X)	(X)	(CR)
Start	S (CR)							

Order of Significance of Input Keys—The first character that is typed will be the most significant character. Conversely, the last character that is typed is the least significant character.

Number of Characters—The console emulator routine can accept up to six octal numbers in the range of 0-32K. If all six numbers are inputted, the most significant number should be a one or a zero.

Leading Zeros—When an address or data word contains leading zeros, these zeros can be omitted when loading the address or depositing the data.

Example Using the Load, Examine, Deposit, and Start Function—Assume that a user wishes to:

1. Turn on power
2. Load address 700
3. Examine location 700
4. Deposit 777 into location 700
5. Examine location 700
6. Start at location 700

USER	TERMINAL DISPLAY			
1. turns on power	XXXXXX	XXXXXX	XXXXXX	XXXXXX
2. L (SB) 700 (CR)	$ L 700			
3. E (SB)	$ E 000700 XXXXXX			
4. D (SB) 777 (CR)	$ D 777			
5. E (SB)	$ E 000700 000777			
6. S (CR)	$ S			

Even Addresses Only—The console emulator routine will not work with odd addresses. Even numbered addresses must always be used.

Successive Operations

Examine—Successive examine operations are permitted. The address is loaded for the first examine only. Successive examines cause the address to increment by two and will display consecutive addresses along with their contents.

Example of Successive Examine Operations—Examine Addresses 500-506

Operator Input	Terminal Display
L (SB) 500 (CR)	$L 500
E (SB)	$E 000500 XXXXXX
E (SB)	$E 000502 XXXXXX
E (SB)	$E 000504 XXXXXX
E (SB)	$E 000506 XXXXXX

Deposit—Successive deposit operations are permitted. The procedure is identical to that used with examine.

Example of Successive Deposit Operations

Deposit: 60 into Location 500
2 into Location 502
4 into Location 504

Operation Input	Terminal Display
L (SB) 500 (CR)	$L 500
D (SB) 60 (CR)	$D 60
D (SB) 2 (CR)	$D 2
D (SB) 4 (CR)	$D 4

Alternate Deposit-Examine Operations—This mode of operation will not increment the address. The address will contain the last data which was deposited.

Example of Alternate Deposit-Examine Operations—Load address 500, deposit the following numbers with examines after every deposit: 1000, 2000, 5420.

Operation Input	Terminal Display
L (SB) 500 (CR)	$L 500
D (SB) 1000 (CR)	$D 1000
E (SB)	$E 000500 001000
D (SB) 2000 (CR)	$D 2000
E (SB)	$E 000500 002000
D (SB) 5420 (CR)	$D 5420
E (SB)	$E 000500 005420

Limits of Operation—The M9301 console emulator routine can directly manipulate the lower 28K of memory and the 4K I/O page. Refer to the PDP-11/34 User's Guide for a procedure to utilize the Memory Management unit to examine or deposit in expanded memory.

Booting from the Keyboard
Once the $ symbol has been displayed in response to system power coming up, or the boot switch being depressed, the system is ready to load a bootstrap from the device which the operator selects.

Console Emulator Boot Procedure

1. Find the two character boot code on Table 6-1 that corresponds to the peripheral to be booted.
2. Load medium, papertape, magtape, disc, etc., into the peripheral if required.
3. Verify that the peripheral indicators signify that the peripheral is ready (if applicable).
4. Type the two character code obtained from the table.
5. If there is more than one unit of a given peripheral, type the unit number to be booted (0-7). If no number is typed the default number will be 0.
6. Type (CR), this initiates the boot.

Table of Bootstrap Routine Codes—Supported by both YA and YB versions of the M9301.

Table 6-1 Bootstrap Codes

Device	Description	Boot Command
RK11	Disk cartridge	DK
RP11	RP02/03 disk pack	DP
TC11	DECTAPE	DT
TM11	800 BPI Magtape	MT
TA11	Magnetic cassette	CT
RX11	Diskette	DX
DL11	ASR-33 teletype	TT
PC11	Papertape	PR

Supported by the YB version only (in addition to all the above).

RJS03/04	Fixed Head disk	DS
RJP04	Disk pack	DB
TJU16	Magnetic tape	MM

Before Booting . . .—Always remember:

1. The medium (papertape, disc, magtape, cassette, etc.) must be placed in the peripheral to be booted prior to booting.
2. The machine will not be under the control of the console emulator routine after booting.
3. The program which is booted in must:
   1) be self starting
   2) allow the user to begin execution by using the CONT function, or
   3) be restartable after the console emulator is recalled.

4. Actuating the boot switch will always abort the program being run. The contents of the general registers (R0-R7) will be destroyed. There is no way to continue with the program which was aborted. Some programs are designed to be restartable.

6.10 EIS ARITHMETIC OPERATION

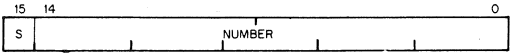
The extended Instruction Set adds the following instruction capability:

Mnemonic	Instruction	Op Code
MUL	multiply	070RSS
DIV	divide	071RSS
ASH	shift arithmetically	072RSS
ASHC	arithmetic shift combined	073RSS

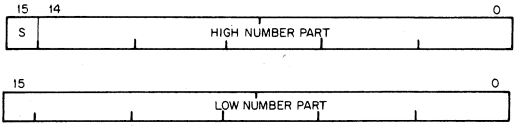
The EIS instructions are directly compatible with the larger 11 computers.

The number formats are:

16-bit single word:



32-bit double word:

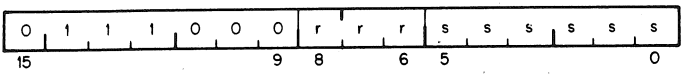


S is the sign bit. S = 0 for positive quantities
S = 1 for negative quantities; number is in 2's complement notation

Interrupts are serviced at the end of an EIS instruction.

MUL

multiply 070RSS



Operation: R, Rv1 ← R x(src)

Condition Codes:
N: set if product is <0; cleared otherwise
Z: set if product is 0; cleared otherwise
V: cleared
C: set if the result is less than -2^{15} or greater than or equal to $2^{15}-1$.

Description: The contents of the destination register and source taken as two'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.)

Example: 16-bit product (R is odd)

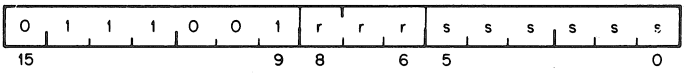
```
CLC              ;Clear carry condition code
MOV  #400,R1
MUL  #10,R1
BCS ERROR          ;Carry will be set if
                   ;product is less than
                   ;-2^15 or greater than or equal to 2^15
                   ;no significance lost

   Before          After

(R1) = 000400      (R1) = 004000
```

Assembler format for all EIS instructions is:
OPR src, R

DIV

divide

071RSS



Operation: R, Rv1 ← R, Rv1 /(src)

Condition Codes:
N: set if quotient <0; cleared otherwise
Z: set if quotient =0; cleared otherwise
V: set if source =0 or if the absolute value of the register is larger than the absolute value of the source. (In this case the instruction is aborted because the quotient would exceed 15 bits.)
C: set if divide 0 attempted; cleared otherwise

Description: The 32-bit two's complement integer in R and **Rv1** is divided by the source operand. The quotient is left in R; the remainder in Rv1. Division will be performed so that the remainder is of the same sign as the dividend. R must be even.

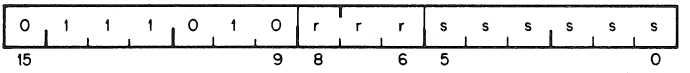
Example:

```
CLR R0
MOV #20001,R1
DIV #2,R0
```

Before	After	
(R0) = 000000	(R0) = 010000	Quotient
(R1) = 020001	(R1) = 000001	Remainder

ASH

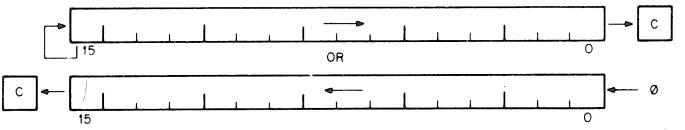
shift arithmetically 072RSS



Operation: R⇐ R Shifted arithmetically NN places to right or left
Where NN = low order 6 bits of source

Condition Codes:
N: set if result <0; cleared otherwise
Z: set if result =0; cleared otherwise
V: set if sign of register changed during shift; cleared otherwise
C: loaded from last bit shifted out of register

Description: The contents of the register are shifted right or left the number of times specified by the shift count. 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 a right shift and positive is a left shift.



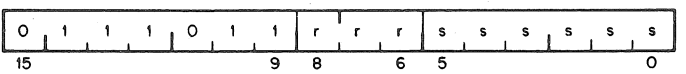
6 LSB of source	Action in general register
011111	Shift left 31 places
000001	shift left 1 place
111111	shift right 1 place
100000	shift right 32 places

Example: ASH R0, R3

Before	After
(R3)=001234	(R3)=012340
(R0)=000003	(R0)=000003

ASHC

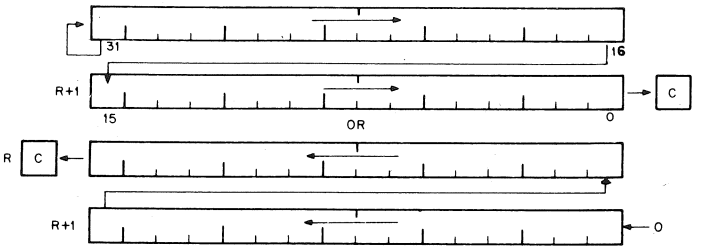
arithmetic shift combined 073RSS



Operation: R, Rv1⩽R, Rv1 The double word is shifted NN places to the right or left, where NN = low order six bits of source

Condition Codes:
N: set if result <0; cleared otherwise
Z: set if result =0; cleared otherwise
V: set if sign bit changes during the shift; cleared otherwise
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)

Description: The contents of the register and the register ORed with one are treated as one 32 bit word, R + 1 (bits 0-15) and R (bits 16-31) are shifted right or left the number of times specified by the shift count. 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. When the register chosen is an odd number the register and the register OR'ed with one are the same. In this case the right shift becomes a rotate (for up to a shift of 16). The 16 bit word is rotated right the number of bits specified by the shift count.



CHAPTER 7

MEMORY MANAGEMENT

7.1 GENERAL

7.1.1 Memory Management

This chapter describes the Memory Management unit of the 11/34 Central Processor. The PDP-11/34 provides the hardware facilities necessary for complete memory management and protection. It is designed to be a memory management facility for systems where the memory size is greater than 28K words and for multi-user, multi-programming systems where protection and relocation facilities are necessary.

7.1.2 Programming

The Memory Management hardware has been optimized towards a multi-programming environment and the processor can operate in two modes, Kernel and User. When in Kernel mode, the program has complete control and can execute all instructions. Monitors and supervisory programs would be executed in this mode.

When in User Mode, the program is prevented from executing certain instructions that could:

a) cause the modification of the Kernel program.
b) halt the computer.
c) use memory space assigned to the Kernel or other users.

In a multi-programming environment several user programs would be resident in memory at any given time. The task of the supervisory program would be: control the execution of the various user programs, manage the allocation of memory and peripheral device resources, and safeguard the integrity of the system as a whole by careful control of each user program.

In a multi-programming system, the Management Unit provides the means for assigning pages (relocatable memory segments) to a user program and preventing that user from making any unauthorized access to those pages outside his assigned area. Thus, a user can effectively be prevented from accidental or willful destruction of any other user program or the system executive program.

Hardware implemented features enable the operating system to dynamically allocate memory upon demand while a program is being run. These features are particularly useful when running higher-level language programs, where, for example, arrays are constructed at execution time. No fixed space is reserved for them by the compiler. Lacking dynamic memory allocation capability, the program would have to calculate and allow sufficient memory space to accommodate the worst case. Memory Management eliminates this time-consuming and wasteful procedure.

7.1.3 Basic Addressing

The addresses generated by all PDP-11 Family Central Processor Units (CPUs) are 18-bit direct byte addresses. Although the PDP-11 Family word length is 16 bits, the UNIBUS and CPU addressing logic actually is 18 bits. Thus, while the PDP-11 word can only contain address references up to 32K words (64K bytes) the CPU and UNIBUS can reference addresses up to 128K words (256K bytes). These extra two bits of addressing logic provide the basic framework for expanding memory references.

In addition to the word length constraint on basic memory addressing space, the uppermost 4K words of address space is always reserved for UNIBUS I/O device registers. In a basic PDP-11 memory configuration (without Management) all address references to the uppermost 4K words of 16-bit address space (160000-177777) are converted to full 18-bit references with bits 17 and 16 always set to 1. Thus, a 16-bit reference to the I/O device register at address 173224 is automatically internally converted to a full 18-bit reference to the register at address 773224. Accordingly, the basic PDP-11 configuration can directly address up to 28K words of true memory, and 4K words of UNIBUS I/O device registers.

7.1.4 Active Page Registers

The Memory Management Unit uses two sets of eight 32-bit Active Page Registers. An APR is actually a pair of 16-bit registers: a Page Address Register (PAR) and a Page Descriptor Register (PDR). These registers are always used as a pair and contain all the information needed to describe and relocate the currently active memory pages.

One set of APR's is used in Kernel mode, and the other in User mode. The choice of which set to be used is determined by the current CPU mode contained in the Processor Status word.

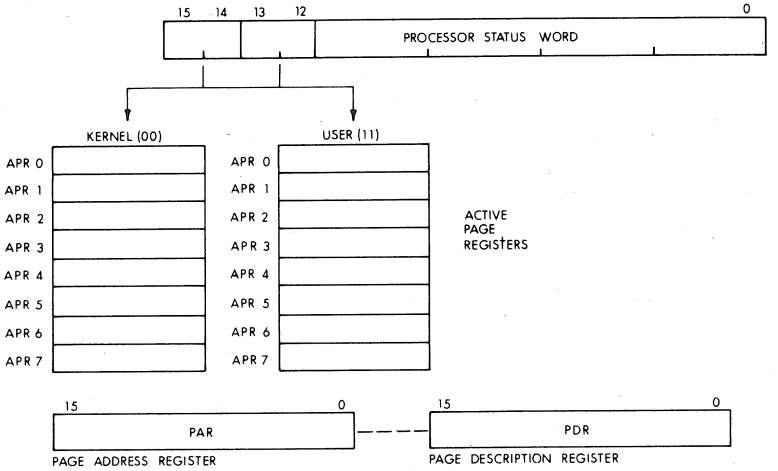


Figure 7-1 Active Page Registers

7.1.5 Capabilities Provided by Memory Management

Memory Size (words):	124K, max (plus 4K for I/O & registers)
Address Space:	Virtual (16 bits) Physical (18 bits)
Modes of Operation:	Kernel & User
Stack Pointers:	2 (one for each mode)
Memory Relocation:	
Number of Pages:	16 (8 for each mode)
Page Length:	32 to 4,096 words
Memory Protection:	no access read only read/write

7.2 RELOCATION

7.2.1 Virtual Addressing

When the Memory Management Unit is operating, the normal 16-bit direct byte address is no longer interpreted as a direct Physical Address (PA) but as a Virtual Address (VA) containing information to be used in constructing a new 18-bit physical address. The information contained in the Virtual Address (VA) is combined with relocation and description information contained in the Active Page Register (APR) to yield an 18-bit Physical Address (PA).

Because addresses are automatically relocated, the computer may be considered to be operating in virtual address space. This means that no matter where a program is loaded into physical memory, it will not have

to be "re-linked"; it always appears to be at the same virtual location in memory.

The virtual address space is divided into eight 4K-word pages. Each page is relocated separately. This is a useful feature in multi-programmed timesharing systems. It permits a new large program to be loaded into discontinuous blocks of physical memory.

A page may be as small as 32 words, so that short procedures or data areas need occupy only as much memory as required. This is a useful feature in real-time control systems that contain many separate small tasks. It is also a useful feature for stack and buffer control.

A basic function is to perform memory relocation and provide extended memory addressing capability for systems with more than 28K of physical memory. Two sets of page address registers are used to relocate virtual addresses to physical addresses in memory. These sets are used as hardware relocation registers that permit several user's programs, each starting at virtual address 0, to reside simultaneously in physical memory.

7.2.2 Program Relocation

The page address registers are used to determine the starting address of each relocated program in physical memory. Figure 7-2 shows a simplified example of the relocation concept.

Program A starting address 0 is relocated by a constant to provide physical address 6400_8.

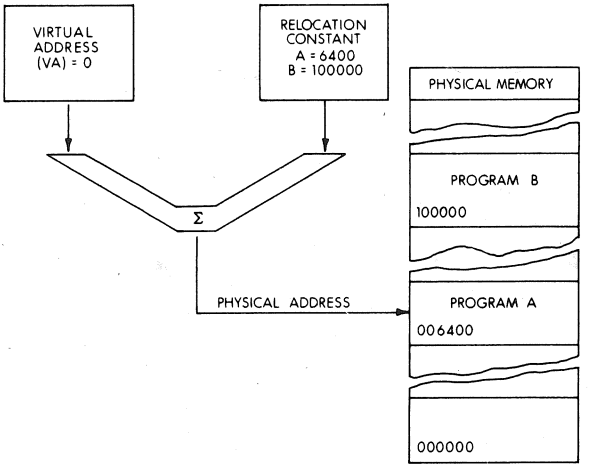


Figure 7-2 Simplified Memory Relocation Example

If the next processor virtual address is 2, the relocation constant will then cause physical address 6402_8, which is the second item of Program A, to be accessed. When Program B is running, the relocation constant is changed to 100000_8. Then, Program B virtual addresses starting at 0, are relocated to access physical addresses starting at 100000_8. Using the active page address registers to provide relocation eliminates the need to "relink" a program each time it is loaded into a different physical memory location. The program always appears to start at the same address.

A program is relocated in pages consisting of from 1 to 128 blocks. Each block is 32 words in length. Thus, the maximum length of a page is 4096 (128 x 32) words. Using all of the eight available active page registers in a set, a maximum program length of 32,768 words can be accommodated. Each of the eight pages can be relocated anywhere in the physical memory, as long as each relocated page begins on a boundary that is a multiple of 32 words. However, for pages that are smaller then 4K words, only the memory actually allocated to the page may be accessed.

The relocation example shown in Figure 7-3 illustrates several points about memory relocation.

a) Although the program appears to be in contiguous address space to the processor, the 32K-word physical address space is actually scattered through several separate areas of physical memory. As long as the total available physical memory space is adequate, a program can be loaded. The physical memory space need not be contiguous.

b) Pages may be relocated to higher or lower physical addresses, with respect to their virtual address ranges. In the example Figure 7-3, page 1 is relocated to a higher range of physical addresses, page 4 is relocated to a lower range, and page 3 is not relocated at all (even though its relocation constant is non-zero).

c) All of the pages shown in the example start on 32-word boundaries.

d) Each page is relocated independently. There is no reason why two or more pages could not be relocated to the same physical memory space. Using more than one page address register in the set to access the same space would be one way of providing different memory access rights to the same data, depending upon which part of a program was referencing that data.

Memory Units

Block:	32 words
Page:	1 to 128 blocks (32 to 4,096 words)
No. of pages:	8 per mode
Size of relocatable memory:	27,768 words, max (8 x 4,096)

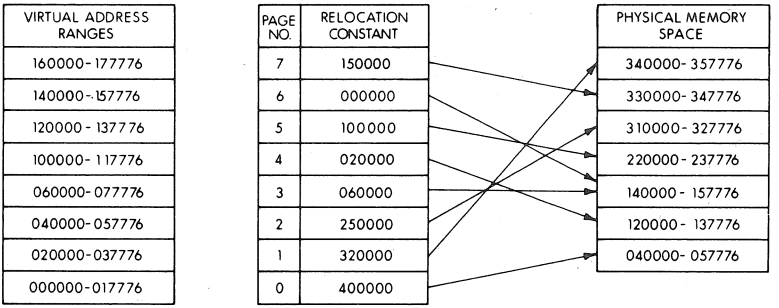


Figure 7-3 Relocation of a 32K Word Program into 124K Word Physical Memory

7.3 PROTECTION

A timesharing system performs multiprogramming; it allows several programs to reside in memory simultaneously, and to operate sequentially. Access to these programs, and the memory space they occupy, must be strictly defined and controlled. Several types of memory protection must be afforded a timesharing system. For example:

a) User programs must not be allowed to expand beyond allocated space, unless authorized by the system.

b) Users must be prevented from modifying common subroutines and algorithms that are resident for all users.

c) Users must be prevented from gaining control of or modifying the operating system software.

The Memory Management option provides the hardware facilities to implement all of the above types of memory protection.

7.3.1 Inaccessible Memory

Each page has a 2-bit access control key associated with it. The key is assigned under program control. When the key is set to 0, the page is defined as non-resident. Any attempt by a user program to access a non-resident page is prevented by an immediate abort. Using this feature to provide memory protection, only those pages asociated with the current program are set to legal access keys. The access control keys of all other program pages are set to 0, which prevents illegal memory references.

7.3.2 Read-Only Memory

The access control key for a page can be set to 2, which allows read (fetch) memory references to the page, but immediately aborts any attempt to write into that page. This read-only type of memory protection

can be afforded to pages that contain common data, subroutines, or shared algorithms. This type of memory protection allows the access rights to a given information module to be user-dependent. That is, the access right to a given information module may be varied for different users by altering the access control key.

A page address register in each of the sets (Kernel and User modes) may be set up to reference the same physical page in memory and each may be keyed for different access rights. For example, the User access control key might be 2 (read-only access), and the Kernel access control key might be 6 (allowing complete read/write access).

7.3.3 Multiple Address Space

There are two complete separate PAR/PDR sets provided: one set for Kernel mode and one set for User mode. This affords the timesharing system with another type of memory protection capability. The mode of operation is specified by the Processor Status Word current mode field, or previous mode field, as determined by the current instruction.

Assuming the current mode PS bits are valid, the active page register sets are enabled as follows:

PS(bits15, 14)	PAR/PDR Set Enabled
00	Kernel mode
01, 10	Illegal (all references aborted on access)
11	User mode

Thus, a User mode program is relocated by its own PAR/PDR set, as are Kernel programs. This makes it impossible for a program running in one mode to accidentally reference space allocated to another mode when the active page registers are set correctly. For example, a user cannot transfer to Kernel space. The Kernel mode address space may be reserved for resident system monitor functions, such as the basic Input/Output Control routines, memory management trap handlers, and timesharing scheduling modules. By dividing the types of timesharing system programs functionally between the Kernel and User modes, a minimum amount of space control housekeeping is required as the timeshared operating system sequences from one user program to the next. For example, only the User PAR/PDR set needs to be updated as each new user program is serviced. The two PAR/PDR sets implemented in the Memory Management Unit are shown in Figure 7-1.

7.4 ACTIVE PAGE REGISTERS

The Memory Management Unit provides two sets of eight Active Page Registers (APR). Each APR consists of a Page Address Register (PAR) and a Page Descriptor Register (PDR). These registers are always used as a pair and contain all the information required to locate and describe the current active pages for each mode of operation. One PAR/PDR set is used in Kernel mode and the other is used in User mode. The current mode bits (or in some cases, the previous mode bits) of the Processor Status Word determine which set will be referenced for each memory access. A program operating in one mode cannot use the PAR/PDR sets of the other mode to access memory. Thus, the two sets are

a key feature in providing a fully protected environment for a time-shared multi-programming system.

A specific processor I/O address is assigned to each PAR and PDR of each set. Table 7-1 is a complete list of address assignment.

NOTE

UNIBUS devices cannot access PARs or PDRs

In a fully-protected multi-programming environment, the implication is that only a program operating in the Kernel mode would be allowed to write into the PAR and PDR locations for the purpose of mapping user's programs. However, there are no restraints imposed by the logic that will prevent User mode programs from writing into these registers. The option of implementing such a feature in the operating system, and thus explicitly protecting these locations from user's programs, is available to the system software designer.

Table 7-1 PAR/PDR Address Assignments

Kernel Active Page Registers			User Active Page Registers		
No.	PAR	PDR	No.	PAR	PDR
0	772340	772300	0	777640	777600
1	772342	772302	1	777642	777602
2	772344	772304	2	777644	777604
3	772346	772306	3	777646	777606
4	772350	772310	4	777650	777610
5	772352	772312	5	777652	777612
6	772354	772314	6	777654	777614
7	772356	772316	7	777656	777616

7.4.1 Page Address Registers (PAR)

The Page Address Register (PAR), shown in Figure 7-4, contains the 12-bit Page Address Field (PAF) that specifies the base address of the page.

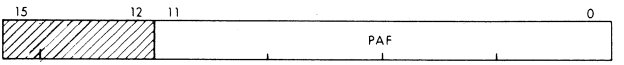


Figure 7-4 Page Address Register

Bits 15-12 are unused and reserved for possible future use.

The Page Address Register may be alternatively thought of as a relocation constant, or as a base register containing a base address. Either interpretation indicates the basic function of the Page Address Register (PAR) in the relocation scheme.

7.4.2 Page Descriptor Registers (PDR)

The Page Descriptor Register (PDR), shown in Figure 7-5, contains information relative to page expansion, page length, and access control.

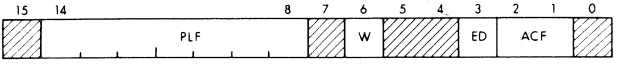


Figure 7-5 Page Descriptor Register

Access Control Field (ACF)
This 2-bit field, bits 2 and 1, of the PDR describes the access rights to this particular page. The access codes or "keys" specify the manner in which a page may be accessed and whether or not a given access should result in an abort of the current operation. A memory reference that causes an abort is not completed and is terminated immediately.

Aborts are caused by attempts to access non-resident pages, page length errors, or access violations, such as attempting to write into a read-only page. Traps are used as an aid in gathering memory management information.

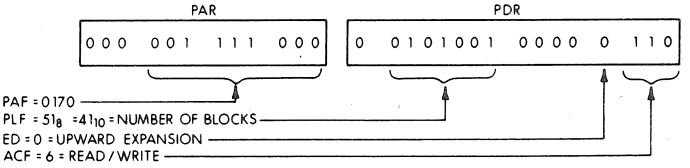
In the context of access control, the term "write" is used to indicate the action of any instruction which modifies the contents of any addressable word. A "write" is synonymous with what is usually called a "store" or "modify" in many computer systems. Table 7-2 lists the ACF keys and their functions. The ACF is written into the PDR under program control.

Table 7-2 Access Control Field Keys

AFC	Key	Description	Function
00	0	Non-resident	Abort any attempt to access this non-resident page
01	2	Resident read-only	Abort any attempt to write into this page.
10	4	(unused)	Abort all Accesses.
11	6	Resident read/ write	Read or Write allowed. No trap or abort occurs.

Expansion Direction (ED)
The ED bit located in PDR bit position 3 indicates the authorized direction in which the page can expand. A logic 0 in this bit (ED = 0) indicates the page can expand upward from relative zero. A logic 1 in this bit (ED = 1) indicates the page can expand downward toward relative zero. The ED bit is written into the PDR under program control. When the expansion direction is upward (ED = 0), the page length is increased by adding blocks with higher relative addresses. Upward expansion is usually specified for program or data pages to add more program or table space. An example of page expansion upward is shown in Figure 7-6.

When the expansion direction is downward (ED = 1), the page length is increased by adding blocks with lower relative addresses. Downward expansion is specified for stack pages so that more stack space can be added. An example of page expansion downward is shown in Figure 7-7.



NOTE:
To specify a block length of 42 for an upward expandable page, write highest authorized block no. directly into high byte of PDR. Bit 15 is not used because the highest allowable block number is 177_8.

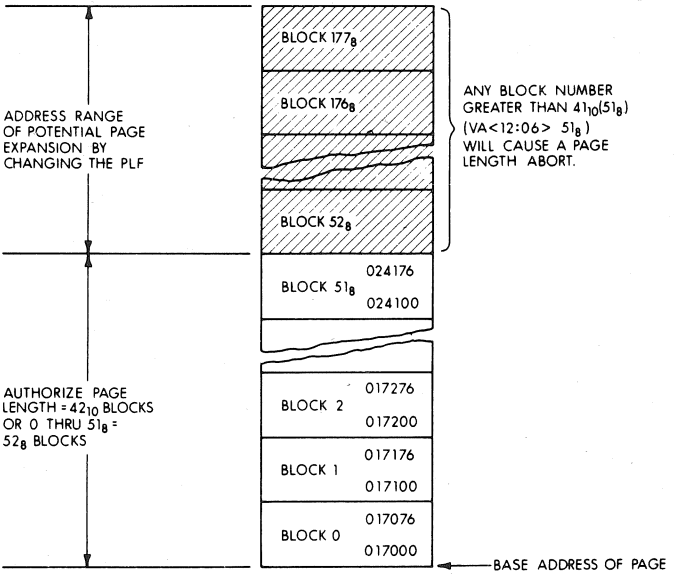


Figure 7-6 Example of an Upward Expandable Page

Written Into (W)
The W bit located in PDR bit position 6 indicates whether the page has been written into since it was loaded into memory. W = 1 is affirmative. The W bit is automatically cleared when the PAR or PDR of that page is written into. It can only be set by the control logic.

In disk swapping and memory overlay applications, the W bit (bit 6) can be used to determine which pages in memory have been modified by a user. Those that have been written into must be saved in their current form. Those that have not been written into (W = 0), need not be saved and can be overlayed with new pages, if necessary.

Page Length Field (PLF)
The 7-bit PLF located in PDR (bits 14-8) specifies the authorized length of the page, in 32-word blocks. The PLF holds block numbers from 0 to 177_8; thus allowing any page length from 1 to 128_{10} blocks. The PLF is written in the PDR under program control.

PLF for an Upward Expandable Page
When the page expands upward, the PLF must be set to one less than the intended number of blocks authorized for that page. For example, if 52_8 (42_{10}) blocks are authorized, the PLF is set to 51_8 (41_{10}) (Figure 7-6). The hardware compares the virtual address block number, VA (bits 12-6) with the PLF to determine if the virtual address is within the authorized page length.

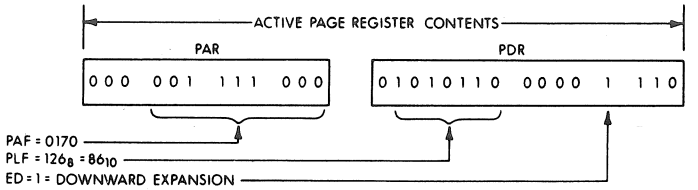
When the virtual address block number is less than or equal to the PLF, the virtual address is within the authorized page length. If the virtual address is greater than the PLF, a page length fault (address too high) is detected by the hardware and an abort occurs. In this case, the virtual address space legal to the program is non-contiguous because the three most significant bits of the virtual address are used to select the PAR/PDR set.

PLF for a Downward Expandable Page
The capability of providing downward expansion for a page is intended specifically for those pages that are to be used as stacks. In the PDP-11, a stack starts at the highest location reserved for it and expands downward toward the lowest address as items are added to the stack.

When the page is to be downward expandable, the PLF must be set to authorize a page length, in blocks, that starts at the highest address of the page. That is always Block 177_8. Refer to Figure 7-7, which shows an example of a downward expandable page. A page length of 42_{10} blocks is arbitrarily chosen so that the example can be compared with the upward expandable example shown in Figure 7-6.

NOTE

The same PAF is used in both examples. This is done to emphasize that the PAF, as the base address, always determines the lowest address of the page, whether it is upward or downward expandable.



To specify page length for a downward expandable page, write complement of blocks required into high byte of PDR.

In this example, a 42-block page is required.
PLF is derived as follows:

$42_{10} = 52_8$; two's complement $= 126_8$.

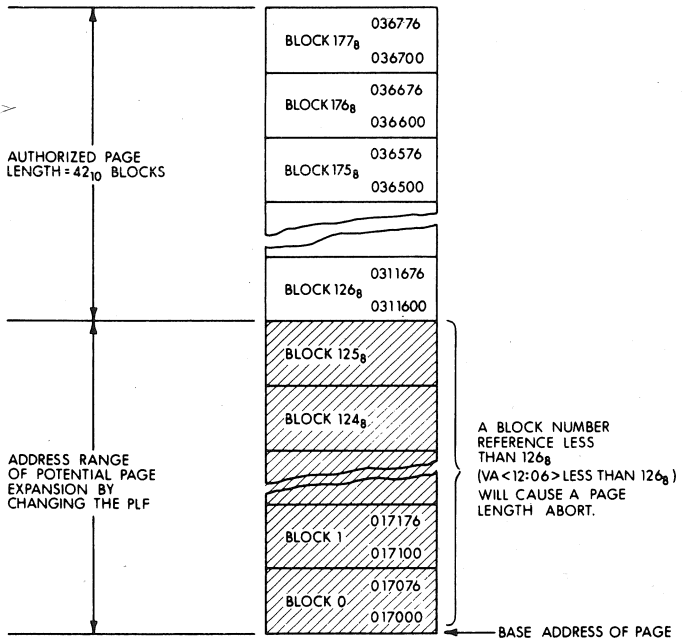


Figure 7-7 Example of a Downward Expandable Page

The calculations for complementing the number of blocks required to obtain the PLF is as follows:

MAXIMUM BLOCK NO.	MINUS	REQUIRED LENGTH	EQUALS	PLF
177_8	—	52_8	=	125_8
127_{10}	—	42_{10}	=	85_{10}

7.5 VIRTUAL & PHYSICAL ADDRESSES

The Memory Management Unit is located between the Central Processor Unit and the UNIBUS address lines. When Memory Management is enabled, the Processor ceases to supply address information to the Unibus. Instead, addresses are sent to the Memory Management Unit where they are relocated by various constants computed within the Memory Management Unit.

7.5.1 Construction of a Physical Address

The basic information needed for the construction of a Physical Address (PA) comes from the Virtual Address (VA), which is illustrated in Figure 7-8, and the appropriate APR set.

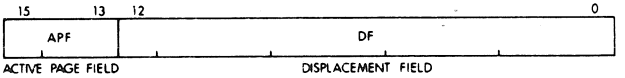


Figure 7-8 Interpretation of a Virtual Address

The Virtual Address (VA) consists of:

1. The Active Page Field (APF). This 3-bit field determines which of eight Active Page Registers (APR0-APR7) will be used to form the Physical Address (PA).

2. The Displacement Field (DF). This 13-bit field contains an address relative to the beginning of a page. This permits page lengths up to 4K words (2^{13} = 8K bytes). The DF is further subdivided into two fields as shown in Figure 7-9.

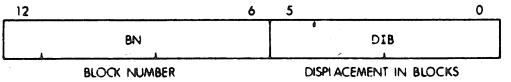


Figure 7-9. Displacement Field of Virtual Address

The Displacement Field (DF) consists of:

1. The Block Number (BN). This 7-bit field is interpreted as the block number within the current page.

2. The Displacement in Block (DIB). This 6-bit field contains the displacement within the block referred to by the Block Number.

The remainder of the information needed to construct the Physical Address comes from the 12-bit Page Address Field (PAF) (part of the Active Page Register) and specifies the starting address of the memory which that APR describes. The PAF is actually a block number in the physical memory, e.g. PAF = 3 indicates a starting address of 96, ($3 \times 32 = 96$) words in physical memory.

The formation of the Physical Address is illustrated in Figure 7-10.

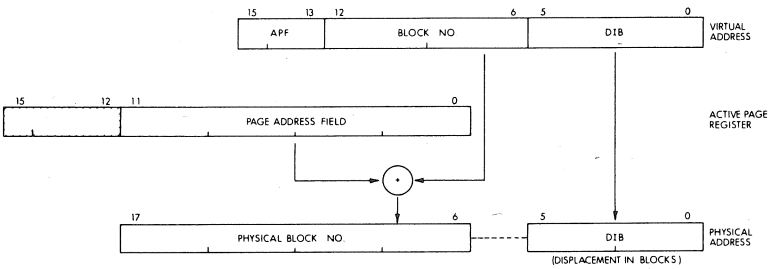


Figure 7-10 Construction of a Physical Address

The logical sequence involved in constructing a Physical Address is as follows:

1. Select a set of Active Page Registers depending on current mode.

2. The Active Page Field of the Virtual Address is used to select an Active Page Register (APR0-APR7).

3. The Page Address Field of the selected Active Page Register contains the starting address of the currently active page as a block number in physical memory.

4. The Block Number from the Virtual Address is added to the block number from the Page Address Field to yield the number of the block in physical memory which will contain the Physical Address being constructed.

5. The Displacement in Block from the Displacement Field of the Virtual Address is joined to the Physical Block Number to yield a true 18-bit Physical Address.

7.5.2 Determining the Program Physical Address

A 16-bit virtual address can specify up to 32K words, in the range from 0 to 177776_8 (word boundaries are even octal numbers). The three most significant virtual address bits designate the PAR/PDR set to be referenced during page address relocation. Table 7-3 lists the virtual address ranges that specify each of the PAR/PDR sets.

Table 7-3 Relating Virtual Address to PAR/PDR Set

Virtual Address Range	PAR/PDR Set
000000-17776	0
020000-37776	1
040000-57776	2
060000-77776	3
100000-117776	4
120000-137776	5
140000-157776	6
160000-177776	7

NOTE

Any use of page lengths less than 4K words causes holes to be left in the virtual address space.

7.6 STATUS REGISTERS

Aborts generated by the protection hardware are vectored through Kernel virtual location 250. Status Registers #0 and #2 are used to determine why the abort occurred. Note that an abort to a location which is itself an invalid address will cause another abort. Thus the Kernel program must insure that Kernel Virtual Address 250 is mapped into a valid address, otherwise a loop will occur which will require console intervention.

7.6.1 Status Register 0 (SR0)

SR0 contains abort error flags, memory management enable, plus other essential information required by an operating system to recover from an abort or service a memory management trap. The SR0 format is shown in Figure 7-11. Its address is 777 572.

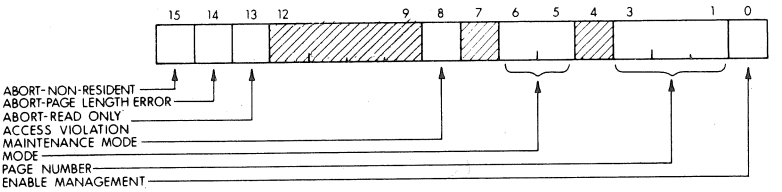


Figure 7-11 Format of Status Register #0 (SR0)

Bits 15-13 are the abort flags. They may be considered to be in a "priority queue" in that "flags to the right" are less significant and should be ignored. For example, a "non-resident" abort service routine would ignore page length and access control flags. A "page length" abort service routine would ignore an access control fault.

NOTE

Bit 15, 14, or 13, when set (abort conditions) cause the logic to freeze the contents of SR0 bits 1 to 6 and status register SR2. This is done to facilitate recovery from the abort.

Protection is enabled when an address is being relocated. This implies that either SR0, bit 0 is equal to 1 (Memory Management enabled) or that SR0, bit 8, is equal to 1 and the memory reference is the final one of a destination calculation (maintenance/destination mode).

Note that SR0 bits 0 and 8 can be set under program control to provide meaningful memory management control information. However, information written into all other bits is not meaningful. Only that information which is automatically written into these remaining bits as a result of hardware actions is useful as a monitor of the status of the memory management unit. Setting bits 15-13 under program control will not cause traps to occur. These bits, however, must be reset to 0 after an abort or trap has occurred in order to resume monitoring memory management.

Abort-Nonresident
Bit 15 is the "Abort-Nonresident" bit. It is set by attempting to access a page with an access control field (ACF) key equal to 0 or 4 or by enabling relocation with an illegal mode in the PS.

Abort—Page Length
Bit 14 is the "Abort-Page Length" bit. It is set by attempting to access a location in a page with a block number (virtual address bits 12-6) that is outside the area authorized by the Page Length Field (PFL) of the PDR for that page.

Abort-Read Only
Bit 13 is the "Abort-Read Only" bit. It is set by attempting to write in a "Read-Only" page having an access key of 2.

NOTE

There are no restrictions that any abort bits could not be set simultaneously by the same access attempt.

Maintenance/Destination Mode
Bit 8 specifies maintenance use of the Memory Management Unit. It is used for diagnostic purposes. For the instructions used in the initial diagnostic program, bit 8 is set so that only the final destination reference is relocated. It is useful to prove the capability of relocating addresses.

Mode of Operation
Bits 5 and 6 indicate the CPU mode (User or Kernel) associated with the page causing the abort. (Kernel = 00, User = 11).

Page Number
Bits 3-1 contain the page number of reference. Pages, like blocks, are numbered from 0 upwards. The page number bit is used by the error recovery routine to identify the page being accessed if an abort occurs.

Enable Relocation and Protection
Bit 0 is the "Enable" bit. When it is set to 1, all addresses are relocated

and protected by the memory management unit. When bit 0 is set to 0, the memory management unit is disabled and addresses are neither relocated nor protected.

7.6.2 Status Register 2 (SR2)

SR2 is loaded with the 16-bit Virtual Address (VA) at the beginning of each instruction fetch but is not updated if the instruction fetch fails. SR2 is read only; a write attempt will not modify its contents. SR2 is the Virtual Address Program Counter. Upon an abort, the result of SR0 bits 15, 14, or 13 being set, will freeze SR2 until the SR0 abort flags are cleared. The address of SR2 is 777 576.

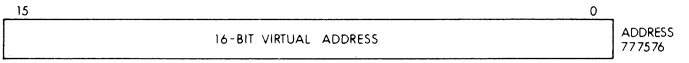


Figure 7-12 Format of Status Register 2(SR2)

7.7 INSTRUCTIONS

Memory Management provides the ability to communicate between two spaces, as determined by the current and previous modes of the Processor Status word (PS).

Mnemonic	Instruction	Op Code
MFPI	move from previous instruction space	0065SS
MTPI	move to previous instruction space	0066DD
MFPD	move from previous data space	1065SS
MTPD	move to previous data space	1066DD

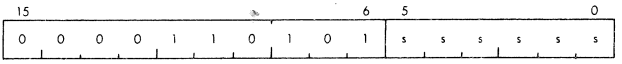
These instructions are directly compatible with the larger 11 computers.

The PDP-11/45 Memory Management unit, the KT11-C, implements a separate instruction and data address space. In the PDP-11/34, there is no differentiation between instruction or data space. The 2 instructions MFPD and MTPD (Move to and from previous data space) execute identically to MFPI and MTPI.

MFPD
MFPI

move from previous data space 1065SS

move from previous instruction space 0065SS



Operation: (temp)←(src)
↓(SP)←(temp)

Condition Codes:
N: set if the source <0; otherwise cleared
Z: set if the source =0; otherwise cleared
V: cleared
C: unaffected

Description: This instruction pushes a word onto the current stack from an address in previous space, Processor Status (bits 13, 12). The source address is computed using the current registers and memory map.

Example:

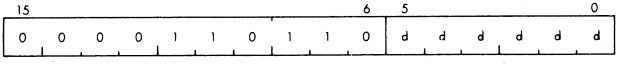
```
MFPI @ (R2)      R2 = 1000
                 1000 = 37526
```

The execution of this instruction causes the contents of (relative) 37526 of the previous address space to be pushed onto the current stack as determined by the PS (bits 15, 14).

MTPD
MTPI

move to previous data space 1066DD

move to previous instruction space 0066DD



Operation: (temp) ←(SP) ↑
(dst) ←(temp)

Condition Codes:
N: set if the sourse <0; otherwise cleared
Z: set if the source =0; otherwise cleared
V: cleared
C: unaffected

Description: This instruction pops a word off the current stack determined by PS (bits 15, 14) and stores that word into an address in previous space PS (bits 13, 12). The destination address is computed using the current registers and memory map. An example is as follows:

Example:

MTPI @ (R2)

R2 = 1000
1000 = 37526

The execution of this instruction causes the top word of the current stack to get stored into the (relative) 37526 of the previous address space.

MTPI AND MFPI, MODE 0, REGISTER 6 ARE UNIQUE IN THAT THESE INSTRUCTIONS ENABLE COMMUNICATIONS TO AND FROM THE PREVIOUS USER STACK.

; MFPI, MODE 0, NOT REGISTER 6

```
MOV   #KM+PUM, PSW      ; KMODE, PREV USER
MOV   #-1, -2(6)        ; MOVE -1 on kernel stack -2
CLR   %0
INC   @#SR0             ; ENABLE MEM MGT
MFPI  %0                ; -(KSP)←R0 CONTENTS
```

The -1 in the kernel stack is now replaced by the contents of R0 which is 0.

; MFPI, MODE 0, REGISTER 6

```
MOV   #UM+PUM, PSW
CLR   %6                ; SET R16=0
MOV   #KM+PUM, PSW      ; K MODE, PREV USER
MOV   #-1, -2 (6)
INC   @#SR0             ; ENABLE MEM MGT
MFPI  %6                ; -(KSP)←R16 CONTENTS
```

The -1 in the kernel stack is now replaced by the contents of R16 (user stack pointer which is 0).

To obtain info from the user stack if the status is set to kernel mode, prev user, two steps are needed.

```
MFPI  %6                ; get contents of R16=user pointer
MFPI  @(6)+             ; get user pointer from kernel stack
                        ; use address obtained to get data
                        ; from user mode using the prev
                        ; mode
```

The desired data from the user stack is now in the kernel stack and has replaced the user stack address.

```
; MTPI, MODE 0      , NOT REGISTER 6

        MOV    #KM+PUM, PSW      ; KERNEL MODE, PREV USES
        MOV    #TAGX, (6)        ; PUT NEW PC ON STACK
        INC    @#SR0             ; ENABLE KT
        MTPI   %7                ; %7← (6)+
        HLT                      ; ERROR
TA6X:   CLR    @#SR0             ; DISABLE MEM MGT
```

The new PC is popped off the current stack and since this is mode 0 and not register 6 the destination is register 7.

```
; MTPI, MODE 0, REGISTER 6

        MOV    #UM+PUM, PSW      ; user mode, Prev User
        CLR    %6                ; set user SP=O (R16)
        MOV    #KM+PUM, PSW      ; Kernel mode, prev user
        MOV    #-1, -(6)         ; MOVE -1 into K stack (R6)
        INC    @#SR0             ; Enable MEM MGT
        MTPI   %6                ; %16 ←(6)+
```

The 0 in R16 is now replaced with −1 from the contents of the kernel stack.

To place info on the user stack if the status is set to kernel mode, prev user mode, 3 separate steps are needed.

```
        MFPI   %6                ; Get content of R16=user pointer
        MOV    #DATA, -(6)       ; put data on current stack
        MTPI   @(6)+             ; @(6)+ [final address relocated]←
                                 (R6)+
```

The data desired is obtained from the kernel stack then the destination address is obtained from the kernel stack and relocated through the previous mode.

Mode Description

In Kernel mode the operating program has unrestricted use of the machine. The program can map users' programs anywhere in core and thus explicitly protect key areas (including the device registers and the Processor Status word) from the User operating environment.

In User mode a program is inhibited from executing a HALT instruction and the processor will trap through location 10 if an attempt is made to execute this instruction. A RESET instruction results in execution of a NOP (no-operation) instruction.

There are two stacks called the Kernel Stack and the User Stack, used by the central processor when operating in either the Kernel or User mode, respectively.

Stack Limit violations are disabled in User mode. Stack protection is provided by memory protect features.

Interrupt Conditions

The Memory Management Unit relocates all addresses. Thus, when Management is enabled, all trap, abort, and interrupt vectors are considered to be in Kernel mode Virtual Address Space. When a vectored transfer occurs, control is transferred according to a new Program Counter (PC) and Processor Status Word (PS) contained in a two-word vector relocated through the Kernel Active Page Register Set.

When a trap, abort, or interrupt occurs the "push" of the old PC, old PS is to the User/Kernel R6 stack specified by CPU mode bits 15, 14 of the new PS in the vector (00 = Kernel, 11 = User). The CPU mode bits also determine the new APR set. In this manner it is possible for a Kernel mode program to have complete control over service assignments for all interrupt conditions, since the interrupt vector is located in Kernel space. The Kernel program may assign the service of some of these conditions to a User mode program by simply setting the CPU mode bits of the new PS in the vector to return control to the appropriate mode.

User Processor Status (PS) operates as follows:

PS Bits	User RTI, RTT	User Traps, Interrupts	Explicit PS Access
Cond. Codes (3-0)	loaded from stack	loaded from vector	*
Trap (4)	loaded from stack	loaded from vector	cannot be changed
Priority (7-5)	cannot be changed	loaded from vector	*
Previous (13-12)	cannot be changed	copied from PS (15, 14)	*
Current (15-14)	cannot be changed	loaded from vector	*

* Explicit operations can be made if the Processor Status is mapped in User space.

APPENDIX A

INSTRUCTION TIMING

INSTRUCTION EXECUTION TIME

The execution time for an instruction depends on the instruction itself, the modes of addressing used, and the type of memory being referenced. In the most general case, the Instruction Execution Time is the sum of a Source Address Time, a Destination Address Time, and an Execute, Fetch Time.

$$\text{Instr Time} = \text{SRC Time} + \text{DST Time} + \text{EF Time}$$

Some of the instructions require only some of these times, and are so noted. All Timing information is in microseconds, unless otherwise noted. Times are typical; processor timing can vary ± 10%.

BASIC INSTRUCTION SET TIMING

Double Operand

$$\text{Instr Time} = \text{SRC Time} + \text{DST Time} + \text{EF Time}$$

Single Operand

$$\text{Instr Time} = \text{DST Time} + \text{EF Time}$$

Branch, Jump, Control, Trap, & Misc

$$\text{Instr Time} = \text{EF Time}$$

NOTES

1) The times specified apply to both word and byte instructions whether odd or even byte.
2) Timing is given without regard for NPR or BR servicing.
3) If the memory management is enabled execution times increase by 0.12 μsec for each memory cycle used.
4) All timing is based on memory with the following performance characteristics:

Memory	Access Time	Cycle Time
Core (MM11-DP)	.510 μsec	1.0 μsec
MOS (MS11-JP)	.635	.775

I. SOURCE ADDRESS TIME

Instruction	Source Mode	Memory Cycles	Core (MM11-DP)	MOS (MS11-JP)
Double Operand	0	0	0.00 μsec	0.00 μsec
	1	1	1.13	1.26
	2	1	1.33	1.46
	3	2	2.37	2.62
	4	1	1.28	1.41
	5	2	2.57	2.82
	6	2	2.57	2.82
	7	3	3.80	4.18

II. DESTINATION TIME

Instruction	Destination Mode	Memory Cycles	Core	MOS
Modifying Single Operand and Modifying Double Operand (Except MOV, SWAB, ROR, ROL ASR ASL)	0	0	0.00	0.00
	1	2	1.62	1.74
	2	2	1.77	1.89
	3	3	2.90	3.15
	4	2	1.77	1.89
	5	3	3.00	3.25
	6	3	3.10	3.35
	7	4	4.29	4.66
MOV	0	0	0.00	0.00
	1	1	0.93	0.93
	2	1	0.93	0.93
	3	2	2.17	2.29
	4	1	1.13	1.13
	5	2	2.22	2.34
	6	2	2.37	2.49
	7	3	3.50	3.75
MTPS	0	0	0.00	0.00
	1	1	0.95	0.95
	2	1	1.13	1.26
	3	2	2.26	2.51
	4	1	1.13	1.26
	5	2	2.26	2.51
	6	2	2.44	2.69
	7	3	3.57	4.20

	Destination Mode	Memory Cycles	Core	MOS
MFPS	0	0	0.00	0.00
	1	1	0.64	0.64
	2	1	0.64	0.64
	3	2	1.95	2.08
	4	1	0.82	0.82
	5	2	1.95	2.08
	6	2	2.13	2.26
	7	3	3.26	3.51

III. EXECUTE, FETCH TIME

DOUBLE OPERAND

Instruction	Memory Cycles	Core	MOS
ADD, SUB, CMP, BIT, BIC, BIS, XOR	1	2.03	2.16
MOV	1	1.83	1.96

SINGLE OPERAND

CLR, COM, INC, DEC, ADC, SBC, TST	1	1.83	1.96
SWAB, NEG	1	2.03	2.16
ROR, ROL, ASR, ASL	1	2.18	2.31
MTPS	2	2.99	3.12
MFPS	2	1.99	2.12

EIS INSTRUCTIONS (use with DST times)

MUL	1	*8.82	*8.95
DIV (overflow)	1	2.78	2.91
		12.48	12.61
ASH	1	**4.18	**4.31
ASHC	1	**4.18	**4.31

MEMORY MANAGEMENT INSTRUCTIONS

MFPI (D)	2	3.07	3.14
MTPI (D)	2	3.37	3.34

* Add 200ns for each bit transition in serial data from LSB to MSB
** Add 200ns per shift

Instruction	Destination Mode	Memory Cycles	Core	MOS
SWAB, ROR, ROL, ASR, ASL	0	0	0.00	0.00
	1	2	1.42	1.54
	2	2	1.57	1.69
	3	3	2.70	2.95
	4	2	1.62	1.74
	5	3	2.80	3.05
	6	3	2.90	3.15
	7	4	4.09	4.46
Non-Modifying Single Operand and Double Operand	0	0	0.00	0.00
	1	1	1.13	1.26
	2	1	1.28	1.41
	3	2	2.42	2.67
	4	1	1.33	1.46
	5	2	2.52	2.77
	6	2	2.62	2.87
	7	3	3.80	4.18
MFPI (D) MTPI (D)	0	0	0.00	0.00
	1	1	0.98	1.24
	2	1	1.32	1.44
	3	2	2.20	2.45
	4	1	1.18	1.44
	5	2	2.20	2.45
	6	2	2.40	2.65
	7	3	3.59	3.96

BRANCH INSTRUCTIONS

Instruction	Memory Cycles	Core	MOS
BR, BNE, BEQ, (Branch) BPL, BMI, BVC, BVS, BCC, BCS, BGE, BLT, BGT, BLE, BHI, BLOS, BHIS, BLO	1	2.18	2.31
(No Branch)	1	1.63	1.76
SOB (Branch)	1	2.38	2.51
(No Branch)	1	1.98	2.11

JUMP INSTRUCTIONS

	Destination Mode	Memory Cycles	Core	MOS
JMP	1	1	1.83	1.96
	2	1	2.18	2.31
	3	2	3.12	3.37
	4	1	2.03	2.16
	5	2	3.07	3.32
	6	2	3.07	3.32
	7	3	4.25	4.78
JSR	1	2	3.32	3.44
	2	2	3.47	3.59
	3	3	4.40	4.65
	4	2	3.32	3.44
	5	3	4.40	4.65
	6	3	4.60	4.85
	7	4	5.69	6.06

Instruction	Memory Cycles	Core	MOS
RTS	2	3.32	3.57
MARK	2	4.27	4.52
RTI, RTT	3	4.60	4.98
Set or Clear C,V,N,Z	1	2.03	2.16
HALT	1	1.68	1.81
WAIT	1	1.68	1.81
RESET	1	100 msec	100 msec
IOT, EMT, TRAP, BPT	5	7.32	7.7

LATENCY

Interrupts (BR requests) are acknowledged at the end of the current instruction. For a typical instruction, with an instruction execution time of 4 μsec, the average time to request acknowledgement would be 2 μsec.

Interrupt service time, which is the time from BR acknowledgement to the first subroutine instruction, is 7.32 μsec, max. for core, and 7.7 μsec for MOS.

NPR (DMA) latency, which is the time from request to bus mastership for the first NPR device, is 2.5 μsec, max.

APPENDIX B
INSTRUCTION INDEX

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

DIGITAL EQUIPMENT CORPORATION, Corporate Headquarters: Maynard, Massachusetts 01754, Telephone: (617) 897-5111

SALES AND SERVICE OFFICES

UNITED STATES—ALABAMA, Huntsville • ARIZONA, Phoenix and Tucson • CALIFORNIA, El Segundo, Los Angeles, Oakland, Ridgecrest, San Diego, San Francisco (Mountain View), Santa Ana, Santa Clara, Stanford, Sunnyvale and Woodland Hills • COLORADO, Englewood • CONNECTICUT, Fairfield and Meriden • DISTRICT OF COLUMBIA, Washington (Lanham, MD) • FLORIDA, Ft. Lauderdale and Orlando • GEORGIA, Atlanta • HAWAII, Honolulu • ILLINOIS, Chicago (Rolling Meadows) • INDIANA, Indianapolis • IOWA, Bettendorf • KENTUCKY, Louisville • LOUISIANA, New Orleans (Metairie) • MARYLAND, Odenton • MASSACHUSETTS, Marlborough, Waltham and Westfield • MICHIGAN, Detroit (Farmington Hills) • MINNESOTA, Minneapolis • MISSOURI, Kansas City (Independence) and St. Louis • NEW HAMPSHIRE, Manchester • NEW JERSEY, Cherry Hill, Fairfield, Metuchen and Princeton • NEW MEXICO, Albuquerque • NEW YORK, Albany, Buffalo (Cheektowaga), Long Island (Huntington Station), Manhattan, Rochester and Syracuse • NORTH CAROLINA, Durham/Chapel Hill • OHIO, Cleveland (Euclid), Columbus and Dayton • OKLAHOMA, Tulsa • OREGON, Eugene and Portland • PENNSYLVANIA, Allentown, Philadelphia (Bluebell) and Pittsburgh • SOUTH CAROLINA, Columbia • TENNESSEE, Knoxville and Nashville • TEXAS, Austin, Dallas and Houston • UTAH, Salt Lake City • VIRGINIA, Richmond • WASHINGTON, Bellevue • WISCONSIN, Milwaukee (Brookfield) •

INTERNATIONAL—ARGENTINA, Buenos Aires • AUSTRALIA, Adelaide, Brisbane, Canberra, Melbourne, Perth and Sydney • AUSTRIA, Vienna • BELGIUM, Brussels • BOLIVIA, La Paz • BRAZIL, Rio de Janeiro and Sao Paulo • CANADA, Calgary, Edmonton, Halifax, London, Montreal, Ottawa, Toronto, Vancouver and Winnipeg • CHILE, Santiago • DENMARK, Copenhagen • FINLAND, Helsinki • FRANCE, Grenoble and Paris • GERMANY, Berlin, Cologne, Frankfurt, Hamburg, Hannover, Munich and Stuttgart • HONG KONG • INDIA, Bombay • INDONESIA, Djakarta • IRELAND, Dublin • ITALY, Milan and Turin • JAPAN, Osaka and Tokyo • MALAYSIA, Kuala Lumpur • MEXICO, Mexico City • NETHERLANDS, Utrecht • NEW ZEALAND, Auckland • NORWAY, Oslo • PUERTO RICO, Santurce • SINGAPORE • SWEDEN, Gothenburg and Stockholm • SWITZERLAND, Geneva and Zurich • UNITED KINGDOM, Birmingham, Bristol, Edinburgh, Leeds, London, Manchester and Reading • VENEZUELA, Caracas •

digital

PRINTED IN U.S.A. EB 05710 76 010/20 — 04/09.40