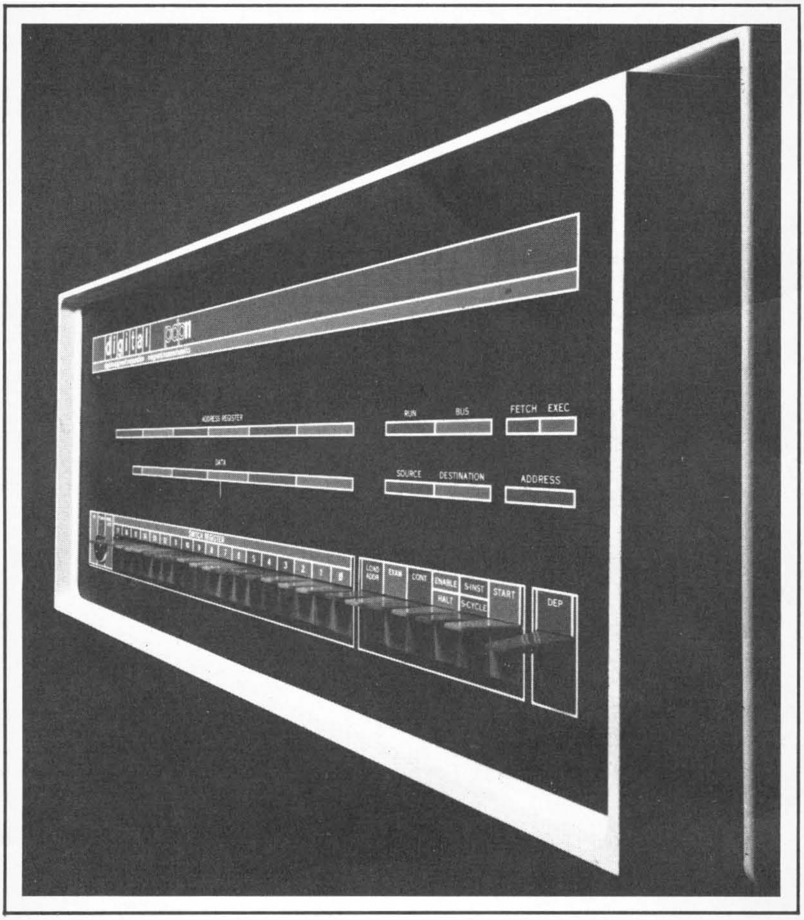
digital

# processor handbook

pdp11/20/15/r20


digital
pdp11
ADDRESS REGISTER
DATA
RUN
BUS
FETCH
EXEC
SOURCE
DESTINATION
ADDRESS
SWITCH REGISTER
LOAD ADDR
EXAM
CONT
ENABLE
HALT
S-INST
S-CYCLE
START
DEP

digital

# pdp11/20 /15 /r20 processor handbook

digital equipment corporation

**DEC Typesetting**
This Handbook was typed and edited with the aid of the DECsystem 10 time-sharing system and type was set via a DEC computer typesetting system.

digital

The PDP-11 is a family of upward-compatible computer systems. We believe that these systems represent a significant departure from traditional methods of computer design.

The initial design step was the development of a totally new language, notation, and theory of computers called the Instruction Set Processor (ISP). This language provides a concise and powerful generalized method for defining an arbitrary computer system and its operation. Along with the development of ISP, a PDP-10 program was written for simulating the operation of any computer system on the basis of its ISP description. With the aid of ISP and the machine simulation program, benchmark comparison tests were run on a large number of potential computer designs. In this manner it was possible to evaluate a variety of design choices and compare their features and advantages, without the time and expense of actually constructing physical prototypes.

Since the main design objective of the PDP-11 was to optimize total system performance, the interaction of software and hardware was carefully considered at every step in the design process. System programmers continually evaluated the efficiency of the code which would be produced by the system software, the ease of coding a program, the speed of real-time response, the power and speed that could be built into a system executive, the ease of system resource management, and numerous other potential software considerations.

The current PDP-11 Family is the result of this design effort. We believe that its general purpose register and UNIBUS organization provides unparalleled power and flexibility. This design is the basis for our continuing commitment to further PDP-11 product development.

Thus the PDP-11 Family is at once a new concept in computer systems, and a tested and tried system. The ultimate proof of this new design approach has come from the large and rapidly increasing number of PDP-11 users all around the world.

[illegible]

Kenneth H. Olsen
President,
Digital Equipment Corporation

## Introduction

This Handbook provides basic information about the PDP-11/20 general purpose 16-bit computer, the PDP-11/15 OEM computer, and the PDP-11R20 rugged computer. Since these computers are functionally identical, all statements about the PDP-11/20 apply also to the PDP-11/15 and the PDP-11R20. Part I describes the processor, its major components and how the PDP-11/20 is programmed. Part II is a summary of PDP-11 software; and Part III describes PDP-11 time-sharing, communications, and data acquisition and control systems.

The PDP-11/20 Processor Handbook is supplemented by the PDP-11 Peripherals and Interfacing Handbook, which includes detailed descriptions of PDP-11 peripherals, options, and the UNIBUS (the single data bus common to all PDP-11 family computers).

Manuals covering the various PDP-11 software packages (Paper Tape, Disk Operating System, FORTRAN, etc.) and detailed hardware maintenance manuals are also available.

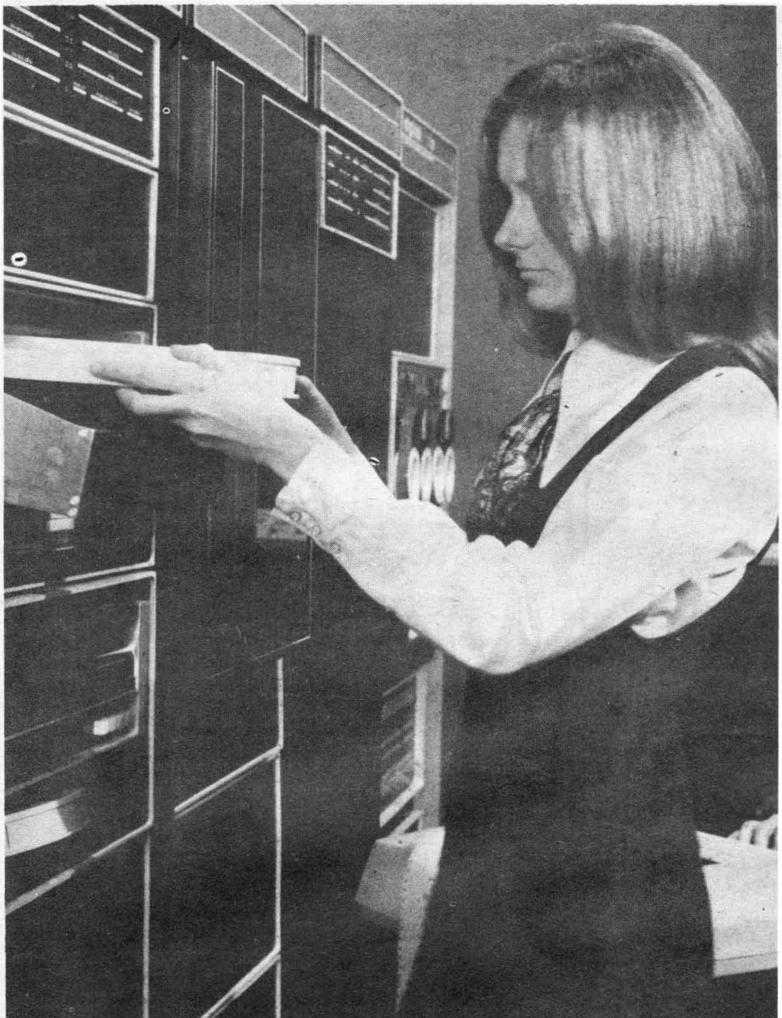
# TABLE OF CONTENTS

## PART II
## SOFTWARE

## PART III
## SYSTEMS

# part 1

PDP-11/20
PDP-11/15
PDP-11R20



PART I
CHAPTER 1

# INTRODUCTION

The PDP-11/20 is a powerful 16-bit computer in the medium-sized branch of the PDP-11 Family of computers. As the first member of the PDP-11 family it is the computer on which the whole family is based. It is a balanced, modular system with a wide range of features, peripherals, software and growth potential not normally found in 16-bit computers.

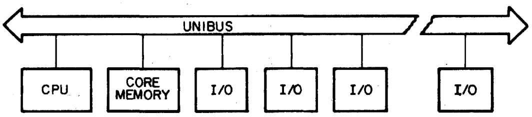
## 1.1 THE PDP-11 FAMILY

The PDP-11 Family includes several processors, a large number of peripheral devices and options, and extensive software. PDP-11 machines are architecturally similar and hardware and software upwards compatible, although each machine has some of its own characteristics. New PDP-11 systems will be compatible with existing family members. The user can chose the system which is most suitable to his application, but as needs change or grow, he can easily add or change hardware. The major characteristics of PDP-11 family computers are listed in Table 1-1.

## 1.2 GENERAL CHARACTERISTICS

### 1.2.1 The UNIBUS

All computer system components and peripherals connect to and communicate with each other on a single high-speed bus known as the UNIBUS -- the key to the PDP-11's many strengths. Since all system elements, including the central processor, communicate with each other in identical fashion via the UNIBUS, the processor has the same easy access to peripherals as it has to memory.



PDP-11 System Simplified Block Diagram

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.

Device communications on the UNIBUS are interlocked. For each command issued by a "master" device, a response signal is received from a "slave" completing the data transfer. Device-to-device communication is completely independent of physical bus length and the response times of master and slave devices.

TABLE 1-1 PDP-11 Family Computers

| | PDP-11/05 | PDP-11/15 | PDP-11/20<br>PDP-11/R20 | PDP-11/45 |
|---|---|---|---|---|
| CENTRAL PROCESSOR | KD11-B | KC11 | KA11 | KB11 |
| General Purpose Registers | 8 | 8 | 8 | 16 |
| Instructions | Basic Set | Basic Set | Basic Set | Basic Set and MUL,DIV XOR,ASH,ASHC, MARK,SXT,SOB, SPL,RTT,MFPI, MTPD,MFPD,MTPI |
| Segmentation Option | No | No | No | Yes |
| Hardware Stacks | Yes | Yes | Yes | Yes |
| Stack Overflow Detection | Yes, fixed | Yes, fixed | Yes, fixed | Yes programmable |
| Automatic Priority interrupt | single-line multi-level | Single line multi-level (four line optional) | four-line multi-level | four-line multi-level PLUS 8 software levels |
| Overlapped instruction | No | No | No | Yes |
| Floating Point Hardware | No | No | No | Internal to CPU(optional) |
| Extended Arithmetic | option | option | option | standard |
| Power Fail and Auto-Restart | standard | option | standard | standard |
| Maximum Addressable Memory Locations | 32K | 32K | 32K (128K optional) | 128K |

Interfaces to the UNIBUS are not time-dependent; there are no pulse-width or rise-time restrictions to worry about. The maximum transfer rate on the UNIBUS is one 16-bit word every 400 nanoseconds, or 2,500,000 words per second.

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. The UNIBUS is further explained in Paragraph 2.2, Chapter 2; and is covered in considerable detail in Part II of the PDP-11 Peripherals and Interfacing Handbook.

## 1.2.2 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, pointers, index registers, or as autoindexing pointers in autoincrement or autodecrement modes. The processor can perform data transfers directly between I/O devices and memory without disturbing the registers; does both single-and double-operand addressing; handles both 16-bit word and 8-bit byte data; and, by using its dynamic stacking technique, allows nested interrupts and automatic reentrant subroutine calling.

### 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 contents with a mask and branch. 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. PDP-11/20 instructions are described in Chapter 4.

The following example contrasts the rotate operation in the PDP-11 with a similar operation in a conventional minicomputer:

**PDP-11 Approach**

```
ROR A            ; rotate contents of memory location A
                 right one place
```

**Conventional Approach**

```
LDA A            ;load contents of memory location A into
                 AC
```

```
        ROR                         ;rotate contents of AC right one place
        STA A                       ;store contents of AC in location A
```

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
```

**Conventional Approach**

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

**Priority Interrupts**
A multi-line 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 line . The PDP-11/15 has only a single line of interrupt (any number of devices). A multi-line system, like that of the PDP-11/20, is optional on the PDP-11/15 (KF11-A).

Each peripheral device in the PDP-11 system has a hardware pointer to its own pair of memory words (one points to the devices's service routine, and the other contains the new status processor information). This unique identification eliminates the need for polling of devices to identify an interrupt, since the interrupt servicing 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 processors priority level. Servicing an interrupt for a device can be interrupted for servicing a higher priority device. 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.

The interrupt scheme is explained in paragraph 2.7, Chapter 2.

**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.

**Addressing**
Much of the power of the PDP-11 is derived from its wide range of addressing capabilities. PDP-11 addressing modes include list sequential addressing, full address indexing, full 16-bit word addressing, 8-bit byte addressing, and stack addressing. Variable length instruction formatting allows a minimum number of bits to be used for each addressing mode. This results in efficient use of program storage space. Addressing modes are described in Chapter 3.

**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. 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, while the processor services the interrupting device. A new status word is then automatically acquired from an area in core memory which is reserved for interrupt instructions (vector area). A return from the interrupt instruction restores the original processor status and returns to the interrupted program without software intervention. Stacks are explained in Chapter 5.

**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. Latency is minimized by the organization and logic of the UNIBUS, which samples requests and priorities in parallel with data transfers.

**Power Fail and Restart**
The PDP-11's power fail and restart system not only protects memory when power fails, but also allows the user to save the existing program location and status (including all dynamic registers), thus preventing harm to devices, and eliminating the need for reloading programs. Automatic restart is accomplished when power returns to safe operating levels, enabling remote or unattended operations of PDP-11 systems. All standard peripherals in the PDP-11 family are included in the systemized power-fail protect/restart feature. This feature is optional on the PDP-11/15 (KP11-A). Power Fail is discussed in Chapter 2, paragraph 2.

## 1.2.3 Memories
Memories with different ranges of speeds and various characteristics can be freely mixed and interchanged in a single PDP-11 system. Thus as memory needs expand and as memory technology grows, a PDP-11 can evolve with none of the growing pains and obsolescence associated with conventional computers. See Chapter 2, paragraph 2.5

## 1.2.4 Packaging
The PDP-11 has adopted a modular approach to allow custom configuring of systems, easy expansion, and easy servicing. Systems are composed of basic building blocks, called System Units, which are completely independent subsystems connected only by pluggable UNIBUS and power connections. There is no fixed wiring between them. An example of this type of subsystem is a 4,096-word memory module.

System Units can be mounted in many combinations within the PDP-11 hardware, since there are no fixed positions for memory or I/O device controllers. Additional units can be mounted easily and connected to the system in the field. In case maintenance is required, defective System Units can be replaced with spares and operation resumed within a few minutes.

## 1.3 PERIPHERALS/OPTIONS

Digital Equipment Corporation (DEC) designs and manufactures many of the peripheral devices offered with PDP-11's. As a designer and manufacturer of peripherals, DEC can offer extremely reliable equipment specifically designed for the small computer environment, lower prices, more choices and quantity discounts.

Many processor, input/output, memory, bus, storage, and communications options are available. These devices are explained in detail in the Peripherals and interfacing Handbook. Options used only by the PDP-11/15, PDP-11/20, and PDP-11R20 are discussed in Chapter 8.

### 1.3.1 I/O Devices

All PDP-11 systems are available with Teletypes as standard equipment. However, their I/O capabilities can be increased with high speed paper tape reader-punches, line printers, card readers or alphanumeric display terminals. The LA30 DECwriter, a totally DEC-designed and built teleprinter, can serve as an alternative to the Teletype. It has several advantages over standard electromechanical typewriter terminals, including higher speed, fewer mechanical parts and very quiet operation.

PDP-11 I/O devices include:

DECterminal alphanumeric display

DECwriter teleprinter

High Speed Line Printers

High Speed Paper Tape Reader and Punch

Teletypes

Card Readers

Synchronous and Asynchronous Communications Interfaces

### 1.3.2 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 147K 16-bit words. For applications which require handling of large volumes of data, DEC offers the industry compatible TU10 Magtape.

Disk storage devices include fixed-head disk units and moving-head removable cartridge and disk pack units. These devices range from the 65K RS64 DECdisk memory, to the RP02 Disk Pack system which can store up to 93.6 million words.

PDP-11 storage devices include:

- DECtape
- Magtape
- RS64 65K-256K word fixed-head disk
- RS11 256K-2M word fixed-head disk
- RK03 1-2M word moving-head disk
- RP02 10M word moving head disk

### 1.3.3 Bus Options
Several options (bus switches, bus extenders) are available for extending the UNIBUS or for configuring multi-processor or shared-peripheral systems.

## 1.4 SOFTWARE
Extensive software, consisting of disk and paper tape systems, is available for PDP-11 Family systems. The larger the PDP-11 configuration, the larger and more comprehensive the software package that comes with it.

### 1.4.1 Paper Tape Software
The Paper Tape Software system includes:

- Editor (ED11)
- Assembler (PAL11)
- Loaders
- On-Line Debugging Technique (ODT11)
- Input-Output Executive (IOX)
- Math Package (FPP11)

### 1.4.2 Disk Operating System Software
The Disk Operating System software includes:

- Text Editor (ED11)
- Relocatable Assembler (PAL11R)
- Linker (LINK11)
- File Utilities Packages (PIP)
- On Line Debugging Technique (ODT11)
- Librarian (LIBR11)

### 1.4.3 Higher Level Languages
PDP-11 users needing an interactive conversational language can use BASIC which can be run on the paper tape software system with only 4.096 words of core memory. A multi-user extension of BASIC is available so up to eight users can access a PDP-11 with only 8K of core.

**RSTS-11**
The PDP-11 Resource Timesharing System (RSTS-11) with BASIC-PLUS, an enriched version of BASIC, is available for up to 16 terminal users.

**FORTRAN**
PDP-11 FORTRAN is an ANSI-standard FORTRAN IV compiler with elements that provide easy compatability with IBM 1130 FORTRAN.

## 1.5 DATA COMMUNICATIONS
The advanced architecture of PDP-11 Family machines makes them ideal for use in data communications applications. For example, the UNIBUS performs like a multiplexer, and multiple single-line interfaces can be added without special multiplexing hardware; byte handling, the key to communications applications, is accomplished easily and efficiently by the PDP-11. To provide total systems capability in the communications area DEC has developed a full line of communications hardware and communications-oriented software.

COMTEX-11 software, is described in Part II, Chapter 4; communications hardware is explained in the Peripherals and Interfacing Handbook; and communications applications are discussed in Part III, Chapter 2.

## 1.6 DATA ACQUISITION CONTROL
The PDP-11, modular process interfaces and special state-of-the art software (RSX-11C Real-Time Executive) combine to provide efficient, low-cost and reliable systems for industrial data acquisition and control (IDACS) applications. IDACS-11 hardware is described in the Peripherals and Interfacing Handbook. RSX-11C is described in Part II, Chapter 6; and the PDP-11 in data acquisition and control applications is discussed in Part III, Chapter 3.

PART I
CHAPTER 2

# SYSTEM ARCHITECTURE

## SYSTEM DEFINITION

Digital Equipment Corporation's PDP-11 is a 16-bit, general-purpose, parallel logic computer using two's complement arithmetic. The PDP-11 is a variable word length processor which directly addresses 32,768 16-bit words or 65,536 8-bit bytes. All communication between system components is done on a single high-speed bus called a UNIBUS. Standard features of the system include eight general-purpose registers which can be used as accumulators, index registers, or address pointers, and an automatic priority interrupt system.

## 2.1 UNIBUS

The UNIBUS is a single, common path that connects the central processor, memory, and all peripherals. Addresses, data, and control information are sent along the 56 lines of the bus.

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. For example, location 10008 is a core memory location, while location 177562 is the Teletype keyboard data buffer. 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

Most UNIBUS lines are bidirectional, so that the same signals that are received as input can be driven as output. This means that a peripheral device register can be either read or loaded by the central processor or other peripheral devices; thus, the same register can be used for both input and output functions.

### 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 is 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. The priority structure is further explained in paragraph 2 .5 of this Chapter.

### 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 response time of the master and slave devices. This asynchronous operation precludes the need for synchronizing with, and waiting for, clock pulses. Thus, each device is allowed to operate at its maximum possible speed.

## 2.2 CENTRAL PROCESSOR

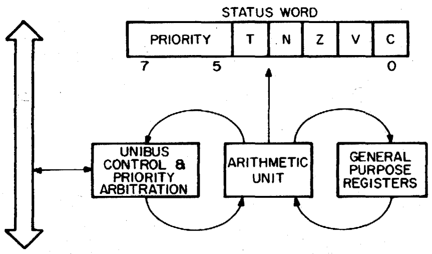
The central processor is organized around three functional blocks: the general purpose registers, arithmetic unit, and UNIBUS and priority control. Data paths conncecting these units are in a figure eight. The processor may perform the following data transfers:

register to register

memory to memory

register to memory

memory to register



STATUS WORD
PRIORITY
T
N
Z
V
C
7
5
0
UNIBUS CONTROL & PRIORITY ARBITRATION
ARITHMETIC UNIT
GENERAL PURPOSE REGISTERS

### 2.2.1 General Registers

The PDP-11/15, PDP-11/20, and PDP-11R20 processors each contain one set of eight general purpose registers. These registers (referred to as R0, R1, R2,...R7) may be used as accumulators, as auto index registers, or as pointers. General Registers R6 and R7 have unique capabilities. R6 serves as the hardware stack pointer, and R7 is the program counter. Using general registers to perform these functions greatly enhances the power and flexibility of the PDP-11. Their use is discussed in Chapter 3 and Chapter 5.

### 2.2.2 Central Processor Status Register

The Central Processor Status Register(PS) contains information on the current priority of the processor, the result of the previous operations, and an indicator for detecting the execution of an instruction to be trapped during program debugging. The priority of the central processor can be set under program control to any one of five levels. This information is held in bits 5, 6, and 7 of the PS.

Four bits of the PS are assigned to monitoring different results of previous instructions. These bits are set as follows:

Z -- if the result was zero

N -- if the result was negative

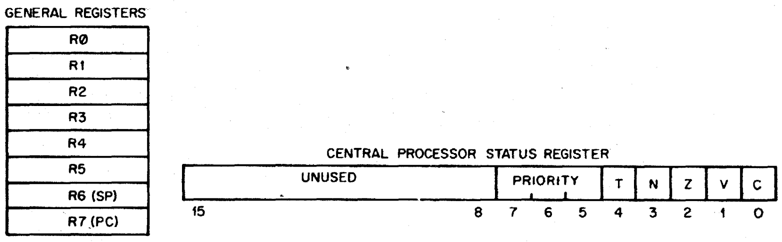
C -- if the operation resulted in a carry from the most significant bit

V -- if the operation resulted in an arithmetic overflow

The T bit is used in program debugging and can be set or cleared under program control. If this bit is set, when an instruction is fetched from memory, a processor trap will occur on completion of the instruction's execution.

The processor status word is location 177776 on the UNIBUS and can be operated on by any instruction.

Register organization for PDP-11/20, PDP-11/15 and PDP-11R20:



## 2.2.3 Processor States

This description of the KA11 (and KC11) processor is intended only to give the reader a basic description of the processor's operation. More detailed discussion, including theory of operation and logic design, is provided in the KA11 Processor Manual, DEC-11-HR2A-D.

The PDP-11 processor has five major states: fetch, source, destination, execute and service. The first four states are used during normal processor operation; service is used during special operations, such as traps and interrupts.

Fetch: locates and decodes an instruction. When fetch is completed, the processor enters another major state, depending on the type of instruction decoded. It is possible to go from fetch to any other state, including back to fetch. Every instruction starts by first entering the fetch state.

Source: decodes the source field of a double-operand instruction and transfers the source operand to the appropriate location. The source major state is entered only if the instruction is a double-operand type.

Destination: decodes the destination field of the appropriate instruction. Destination fields are present in both single and double-operand instructions. Destination operand is accessed and transferred to appropriate location.

Execute: uses the data obtained during previous major states to perform the specified operation. During this state arithmetic operations, logic functions, and tests are performed, and the Destination location is updated if required.

Service: used to execute special operations, such as interrupts, traps, etc.

Although major states follow the sequence of fetch, source, destination, execute, and service, not all major states are required for every instruction. The processor enters only the states necessary to execute the current instruction. The minimum sequence is from fetch of one instruction directly to fetch of the next instruction. Maximum sequence is fetch, source, destination, execute, service, and back to fetch.

### 2.2.4 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, Use of Reserved Instructions, Use of the T bit in the Processor Status Word, and use of the IOT, EMT, and TRAP instructions.

The T bit Trap has already been discussed in this chapter. The IOT, EMT, and TRAP instructions are described in Chapter 4.

**Power Failure**
Whenever AC power drops below 95 volts for 117v nominal power (190 volts for 235 v nominal) or outside a limit of 47 to 63Hz, as measured by DC power, 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), condition peripherals for power fail, and change the contents of location 24 to a pointer to the power-up routine.

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. Power fail and auto-restart is an option on the PDP-11/15.

**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.

**Time-Out Errors**
These errors occur when a Master Synchronization pulse is placed on the UNIBUS and there is no slave pulse within 10 μsec. This error usually occurs in attempts to address non-existant memory or peripherals.

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

**Reserved Instructions**
There is a set of illegal and reserved instructions which cause the processor to trap through location 4.

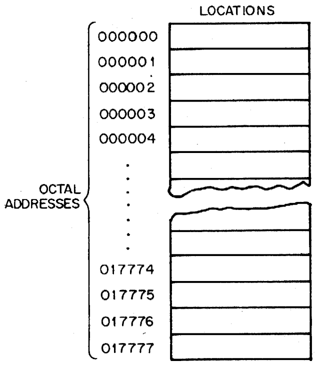
### 2.2.5 Trap Handling

Appendix B includes a list of the reserved Trap Vector Locations. When a trap occurs, the processor follows the same procedure for traps as it does for interrupts (saving the Program Counter (PC) and Processor Status Word (PS) on the new Processor Stack etc...)

## 2.3 CORE MEMORY

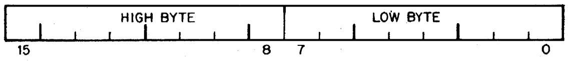
### 2.3.1 Memory Organization

A memory can be viewed as a series of locations, with a number (address) assigned to each location. Thus a 4096-word PDP-11 memory could be shown as follows:

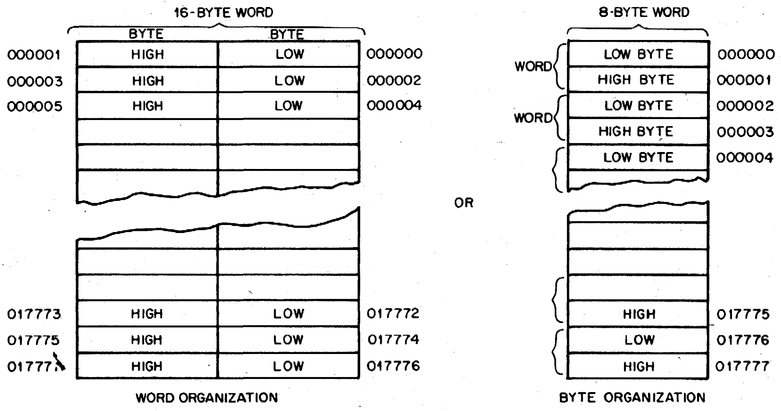


Because PDP-11 memories are designed to accommodate both 16-bit words and 8-bit bytes, the total number of addresses does not correspond to the number of words. A 4096-word memory can contain 8,192 bytes and consists of 017777 octal locations. Words always start at even-numbered locations.

A PDP-11 word is divided into a high byte and a low byte as follows:



Low bytes are stored at even-numbered memory locations and high bytes at odd-numbered memory locations. Thus it is convenient for the programmer to view the PDP-11 memory as follows:



PDP-11 memories are normally provided in 4096-word read and write modules. However, there are also 8192-word interleaved memory modules. The various PDP-11 memories, their characteristics and speeds are listed below.

**Specifications and Memory Types**

| | | | **Time** | | **Time Interleaved*** | |
|---|---|---|---|---|---|---|
| Memory | Size | Type | Access | Cycle | Access | Cycle |
| MM11-E | 4K X 16 bit | Core memory 3W-3D organization; 20 mil, medium temp core | 500ns | 1200ns | 500ns | 900ns |
| MM11-F | 4K X 16 bit | Core memory 3W-3D organization; 20 mil, medium temp core | 400ns | 950ns | 400ns | 490ns** |
| MM11-FP with parity (1 bit per byte)*** | 4K X 18 bit | Core memory 3W-3D organization; 20 mil, medium temp core | 400ns | 950ns | 400ns | 490ns** |
| M792 | 32 16 bit words | Read only; also available as bootstrap loader | 100ns | 100ns | NO | NO |

All memories are PDP-11 Unibus-compatible

Temperature: 0°to 50°C

*MM11-F and MM11-FP automatically interleaved if 8K or more is ordered. Add suffix "X" to part number when ordering MM11-E interleaved (i.e., MM11-EX).

**For a 16-bit DMA transfer into memory. A 16-bit transfer out of memory takes 800 ns.

***Available from Computer Special Systems

The areas of addresses of particular interest to the programmer are the interrupt and trap vectors, processor stack and general storage, and peripheral device registers. Most of the addresses between 000000 and 00370 are reserved for interrupt vectors, and the top 4,096 addresses are generally reserved for peripheral device registers. A detailed address map is contained in Appendix B.

The concept of word "pages" has been completely eliminated in the PDP-11. The programmer can directly address 32K word locations. A memory extension unit is available for the PDP-11/20 and PDP-11R20 to extend the number of addressable locations to 128K.

### 2.3.2 Interleaving

When an address register is incremented on successive memory cycles, the cycles are performed with a 4K memory bank and cannot be overlapped. However, a technique called "interleaving", causes successive memory cycles to be performed within alternate 4K memory banks. This allows cycles to be overlapped; that is the second memory bank can start its cycle before the first memory bank has completed its cycle, provided the bus is free. This effect is called memory interleaving and results in faster memory operation.

Memory interleave is completely transparent to the user, who addresses core as if it were one continuous 8K block. Interleaved memory allows 16-bit transfers into memory every 490 nanoseconds, and out of memory every 800 nanoseconds (using the 950 nanosecond MM11-F).

Interleaving affects 8K blocks. For example, if a system has a 12K memory, the first 8K is interleaved. If the system has 16K of memory, the first 8K would be interleaved and the second 8K would also be interleaved. Any 8K block of memory delivered from DEC is automatically interleaved.

## 2.4 SYSTEM INTERACTION

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.

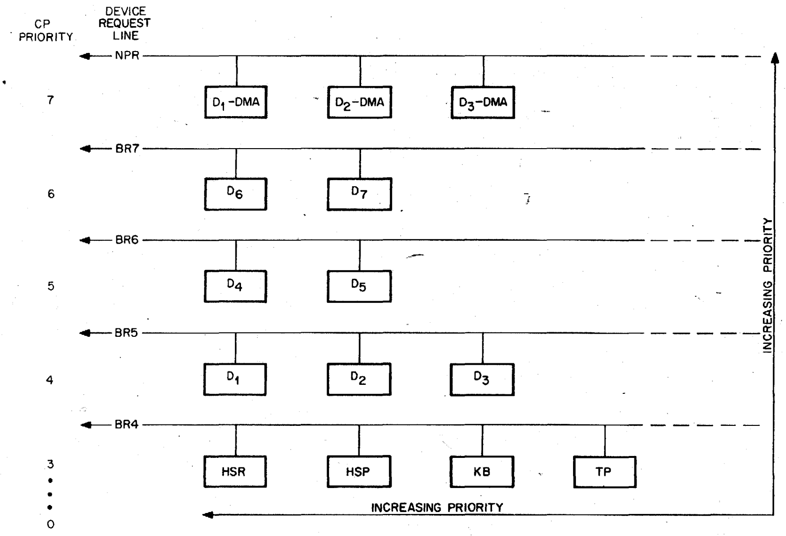
## 2.5 AUTOMATIC PRIORITY INTERRUPTS

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

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) or direct device transfers (disk refreshing a CRT display).

The PDP-11 has a multi-line, multi-level priority interrupt structure.



See Table 1-1, page 2 , for a summary of the API structures of the various PDP-11's. 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 procesor between bus cycles of an instruction execution.

Bus request 7 (BR7) is the next highest priority, and BR4 is the lowest. Levels below BR4 are not implemented in the PDP-11/20, 11/15, or 11R20. They are used in larger machines (PDP-11/45). Thus, a processor priority of 3, 2, 1, or 0 will have the same effect, i.e. all interrupt requests will be granted.

BR7 through BR4 priority requests are honored by the processor between instructions. The priority is hardwired into each device except for the processor, which is programmable. For example, Teletypes are normally assigned to Bus Request line 4. Bus request lines assigned to each peripheral device and option are shown in Appendix B.

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 PS6, 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. Although its priority level is fixed, its actual priority changes as the processor priority varies. Also, 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 can gain control of the bus in 3.5 microseconds or less. An NPR device in control of the bus may transfer 16-bit words from memory at memory speed.

Interrupt Operations - Devices that request interrupts after getting bus control on the bus request lines (BR7, BR6, BR5, BR4) can take advantage of the power and flexibility of the processor. The entire instruction set is available for manipulating data and status registers. When a device servicing program must be run, the task currently under way in the central processor is interrupted and the device service routine is initiated. Once the device request has been satisfied, the processor returns to the interrupted task. This is all accomplished through hardware, and is done automatically by the processor.

Example - A peripheral devices requires service and requests use of the bus at one of the BR levels.

1. The processor determines which device is requesting use of the bus, and compares the priority of the device with the existing processor priority.

2. If device priority is higher, the processor grants priority to the device by sending a signal along a bus grant line, and the device takes control of the bus.

3. When the device has control of the bus, it sends the processor an interrupt command with the address of the words in memory containing the address and status of the appropriate device service routine.

4. The processor then saves the current central processor status (PS) and the current program counter (PC).

5. The new PC and PS are take from the location (interrupt vector) specified by the device and the next location, and the device service routine is begun. Note that these operations all occur automatically and that no device-polling is required to determine which service routine to execute. (Appendix B contains a list of interrupt vectors.)

6. 7.2 microseconds is the time interval between the central processor's receiving the interrupt command and the fetching of the first instruction. This assumes there were no NPR transfers during this time.

7. The device service routine can resume the interrupted process by executing the RTI (Return from Interrupt) instruction. This requires 4.5 microseconds if there are no intervening NPR's. It is done by restoring the old PC and PS.

8. A device service routine can be interrupted in turn by a sufficiently high priority bus request any time after completion of its first instruction.

9. If such an interrupt occurs, the PC and the PS of the device service routine are also automatically saved (without loss of the other PC and PS that had been saved) and the new device routine is initiated. This nesting of priority interrupts can go on to any level, limited only by the core available for temporarily storing the PS and the PC.

PART I

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." (See Chapter 5)

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".

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

R0

R1

R2

R3

R4

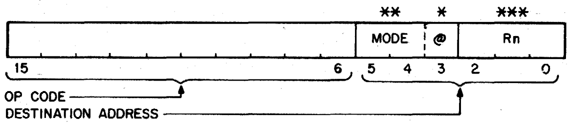
R5

R6 (Hardware Stack Pointer)

R7 (Program Counter)

## 3.1 SINGLE OPERAND ADDRESSING

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



*=SPECIFIES DIRECT OR INDIRECT ADDRESS
**=SPECIFIES HOW REGISTER WILL BE USED
***=SPECIFIES ONE OF 8 GENERAL PURPOSE REGISTERS

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

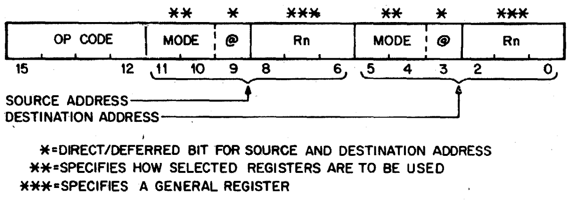
Bits 5 through 0 form a six-bit field called the destination address field. This consists of two subfields:

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

b) Bits 4 and 5 specify how the selected register will be used (address mode). Bit 3 indicates direct or 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:


**
*
***
**
*
***
OP CODE
MODE
@
Rn
MODE
@
Rn
15
12
11
10
9
8
6
5
4
3
2
0
SOURCE ADDRESS
DESTINATION ADDRESS
*=DIRECT/DEFERRED BIT FOR SOURCE AND DESTINATION ADDRESS
**=SPECIFIES HOW SELECTED REGISTERS ARE TO BE USED
***=SPECIFIES A GENERAL REGISTER

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.

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 assembler.

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

## 3.3 DIRECT ADDRESSING

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

DIRECT MODES

| Binary | Name | Assembler Syntax | Function |
|---|---|---|---|
| 0 0 0 | Register | Rn | Register contains operand |
| 0 1 0 | Autoincrement | (Rn)+ | Register is used as a pointer to sequential data then incremented |
| 1 0 0 | Autodecrement | -(Rn) | Register is decremented and then used as a pointer. |
| 1 1 0 | 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

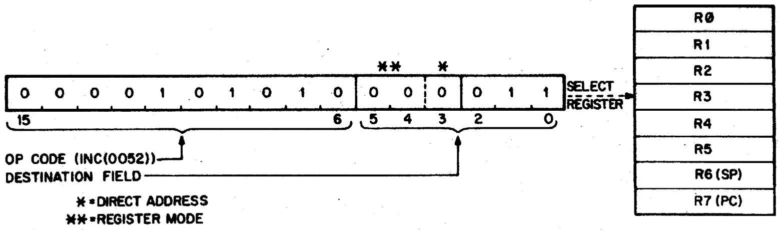
R2 = %2, etc.

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)

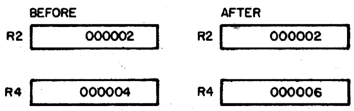
| | Symbolic | Octal Code | Instruction Name |
|---|---|---|---|
| 1. | INC R3 | 005203 | Increment |

Operation: Add one to the contents of general register 3



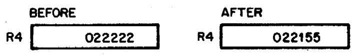
2. ADD R2,R4 060204 Add

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

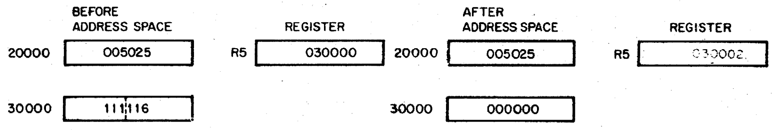
OPR (Rn)+

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**

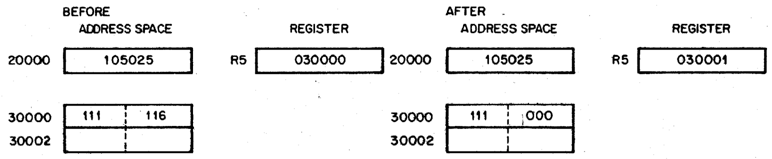
| | Symbolic | Octal Code | Instruction Name |
|---|---|---|---|
| 1. | CLR (R5) + | 005025 | Clear |

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



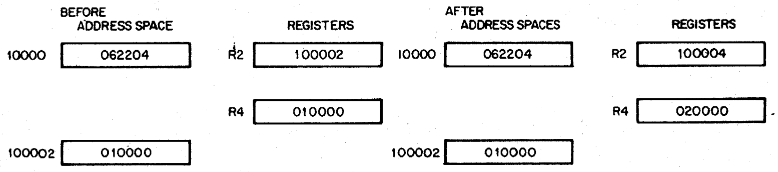
| | | | |
|---|---|---|---|
| 2. | CLRB (R5) + | 105025 | Clear Byte |

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

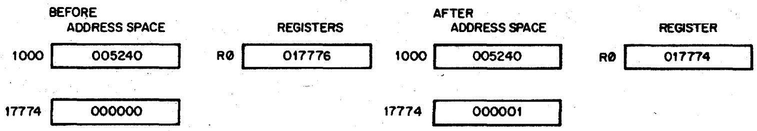
OPR-(Rn)

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 (See Chapter 5 for complete discussions of stacks).

**Autodecrement Mode Examples**

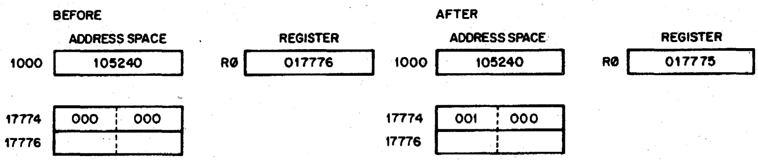
| | Symbolic | Octal Code | Instruction Name |
|---|---|---|---|
| 1. | INC-(R0) | 005240 | Increment |

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



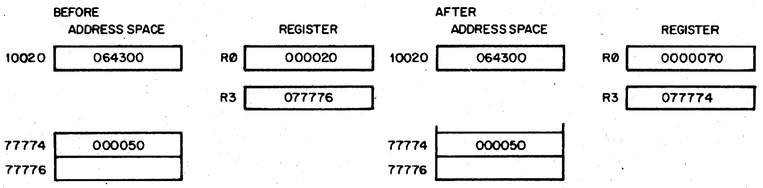
| | Symbolic | Octal Code | Instruction Name |
|---|---|---|---|
| 2. | INCB-(R0) | 105240 | Increment Byte |

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



| | Symbolic | Octal Code | Instruction Name |
|---|---|---|---|
| 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

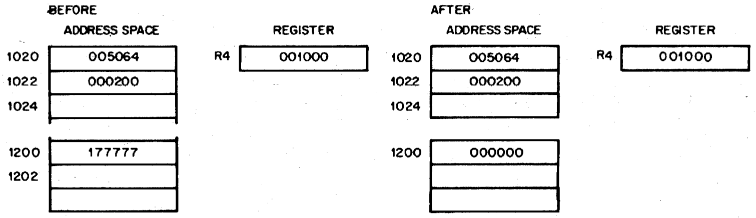
OPR X(Rn)

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**

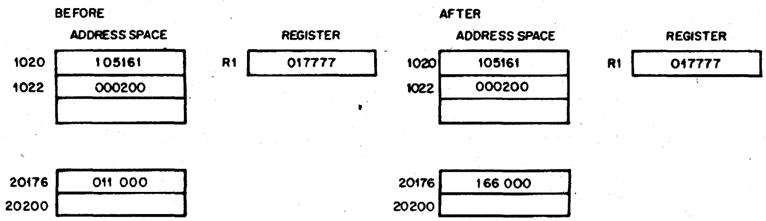
| | Symbolic | Octal Code | Instruction Name |
|---|---|---|---|
| 1. | CLR 200(R4) | 005064<br>000200 | Clear |

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



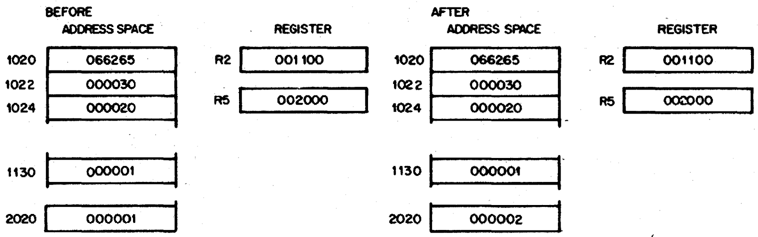
| | | | |
|---|---|---|---|
| 2. | COMB 200(R1) | 105161<br>000200 | Complement Byte |

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, ie. 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 not ambiguous). The following table summarizes the deferred versions of the basic modes:

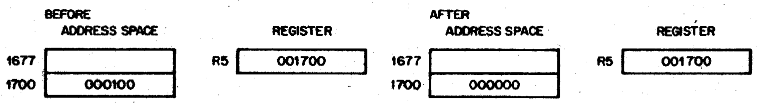
| Binary Code | Name | Assembler Syntax | Function |
|---|---|---|---|
| 0 0 1 | Register Deferred | @Rn or (Rn) | Register contains the address of the operand |
| 0 1 1 | Autoincrement Deferred | @(Rn)+ | Register is first used as a pointer to a word containing the address of the operand, then incremented (always by 2; even for byte instructions). |
| 1 0 1 | 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 |
| 1 1 1 | 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 |

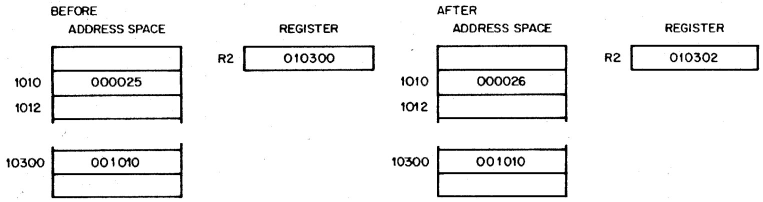
Operation: The contents of location specified in R5 are cleared.



**Autoincrement Deferred Mode Example**

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

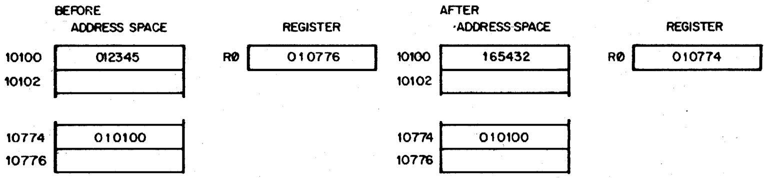
Operation: The contents of the location specified in R2 are used as the address of the address of the operand. Operand is increased by one. Contents of R2 is incremented by 2.



**Autodecrement Deferred Mode Example**

COM @-(R0)     005150     Complement

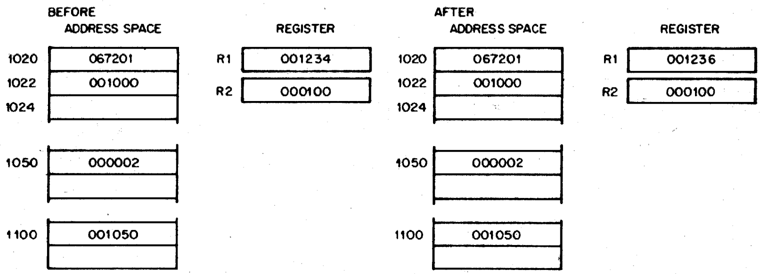
Operation: The contents of R0 are decremented by two and then used as the address of the address of the operand. Operand is one's complemented. (i.e. logically complemented)



**Index Deferred Mode Example**

ADD @1000(R2),R1  067201     Add
                  001000

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:

| Binary Code | Name | Assembler Syntax | Function |
|---|---|---|---|
| 0 1 0 | Immediate | #n | Operand follows instruction |
| 0 1 1 | Absolute | @#A | Absolute Address folows instruction |
| 1 1 0 | Relative | A | Address of A, relative to the instruction, follows the instruction. |
| 1 1 1 | Relative Deferred | @A | Address of location containing address of A, relative to the instruction follows the instruction. |

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 independent 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 which are discussed more fully in Paragraphs 3.5.1 and 3.5.2.

### 3.5.1 Immediate Mode

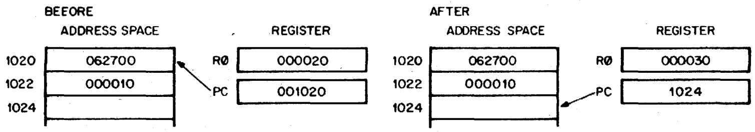
OPR #n,DD

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,RO | 062700<br>000010 | Add |

Operation: The value 10 is located in the second word of the instruction and is added to the contents of RO. 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

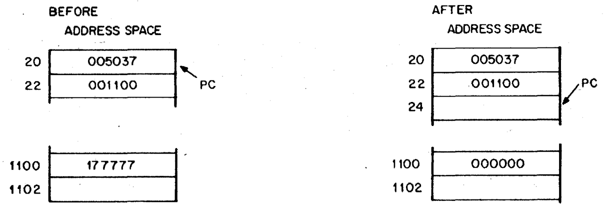
OPR @#A

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**

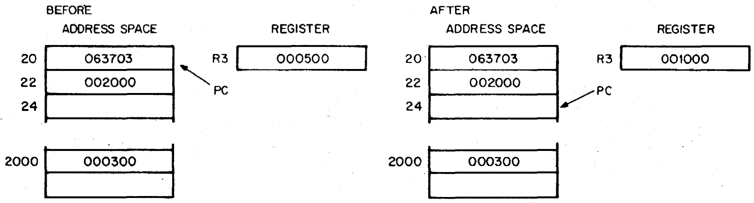
| | Symbolic | Octal Code | Instruction Name |
|---|---|---|---|
| 1. | CLR @#1100 | 005037<br>001100 | Clear |

Operation: Clear the contents of location 1100.



| | Symbolic | Octal Code | Instruction Name |
|---|---|---|---|
| 2. | ADD @#2000,R3 | 063703<br>002000 | Add |

Operation: Add contents of location 2000 to R3.



### 3.5.3 Relative Addressing

OPR A or

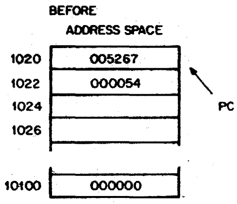
OPR X(PC), where X is the location of A relative to the instruction.

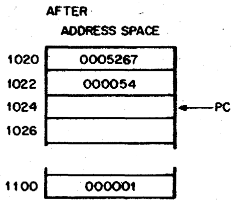
This mode is assembled as index mode using R7. The base of the address calculation, which is stored in the second or third word of the instruction, is not the address of the operand, but the number which, when added to the (PC), becomes the address of the operand. This mode is useful for writing position 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<br>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

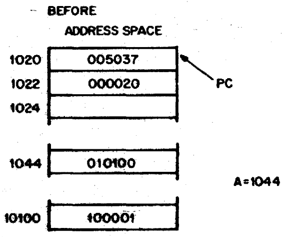
OPR@X(PC), where x is location containing address of A, relative to the instruction.

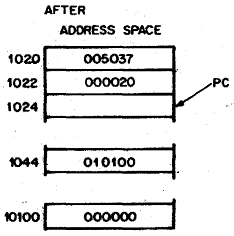
This mode is similar to the relative mode, except that the second word of the instruction, when added to the PC, contains the address of the address of the operand, rather that the address of the operand.

**Relative Deferred Mode Example**

| Symbolic | Octal Code | Instruction Name |
|---|---|---|
| CLR @A | 005077<br>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 the 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 simply leave odd addresses unmodified. Use of stacks is explained in detail in Chapter 5.

Addressing Modes Summary

The following table is a concise summary of the various PDP-11 addressing modes

DIRECT MODES

| Binary Code | Name | Assembler Syntax | Function |
|---|---|---|---|
| 000 | Register | Rn | Register contains operand |
| 010 | Autoincrement | (Rn)+ | Register contains address of operand. Register contents incremented after reference. |
| 100 | Autodecrement | -(Rn) | Register contents decremented before reference register contains address of operand |
| 110 | Index | X(Rn) | Value X (stored in a word following the instruction) is added to (Rn) to produce address of operand. Neither X nor (Rn) are modified. |

## DEFERRED MODES

| Binary Code | Name | Assembler Syntax | Function |
|---|---|---|---|
| 001 | Register Deferred | @Rn or (Rn) | Register contains the address of the operand |
| 011 | 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) |
| 101 | Autodecrement | @-(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 |
| 111 | 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 |

## PC ADDRESSING

| | | | |
|---|---|---|---|
| 010 | Immediate | #n | Operand follows instruction |
| 011 | Absolute | @#A | Absolute address follows instruction |
| 110 | Relative | A | Address of A, relative to the instruction, follows the instruction. |
| 111 | Relative Deferred | @A | Address of location containing address of A, relative to the instruction follows the instruction. |

**PART I**

**CHAPTER 4**

# INSTRUCTION SET

## 4.1 INTRODUCTION
This chapter describes the PDP-11 instructions in the following order:

**Single Operand (4.4)**
General

Shifts

Multiple Precision Instructions
Rotates

**Double Operand (4.5)**
Arithmetic Instructions

Logical Instructions

**Program Control Instructions (4.6)**
Branches

Subroutines

Traps

**Miscellaneous (4.7)**
**Condition Code Operators (4.8)**

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, timing information, a description, special comments, and examples.

MNEMONIC: This is shown at the top left hand side of the 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.)

OPERATION: The operation of each instruction is described with a single notation. The following symbols are used:

| | |
|---|---|
| ( ) = | contents of |
| src = | source address |
| dst = | destination address |
| loc = | location |
| $\leftarrow$ = | becomes |
| $\uparrow$ = | "is popped from stack" |
| $\downarrow$ = | "is pushed onto stack" |
| $\Lambda$ = | boolean AND |
| v = | boolean OR |
| $\forall$ = | exclusive OR |
| $\sim$ = | boolean not |
| Reg or R = | register |
| B = | Byte |

**Instruction Timing**
The PDP-11 is an asynchronous processor in which, in many cases, memory and processor operations are overlapped. The execution time for an instruction is the sum of a basic instruction time and the time to determine and fetch the source and/or destination operands. The following table shows the addressing times required for the various modes of addressing source and destination operands. All times stated are subject to ±20% variation.

| **Addressing Format** | **Timing** | |
|---|---|---|
| (src or dst) | src(μs)** | dst(μs) ** |
| R | 0 | 0 |
| (R) or @R | 1.5 | 1.4* |
| (R)+ | 1.5 | 1.4* |
| -(R) | 1.5 | 1.4* |
| @(R)+ | 2.7 | 2.6* |
| @-(R) | 2.7 | 2.6* |
| BASE(R) | 2.7 | 2.6* |
| @BASE(R) or @(R) | 3.9 | 3.8* |

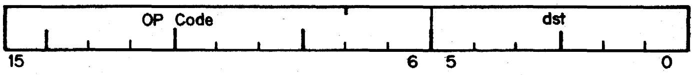
* dst time is 0.5 μs. less than listed time if instruction was a
  CoMPare, CoMPare Byte
  Bit Test, Bit Test Byte
  TeST, or TeST Byte
none of which ever modify the destination word.

** referencing bytes at odd addresses adds 0.6μs to src and dst times.

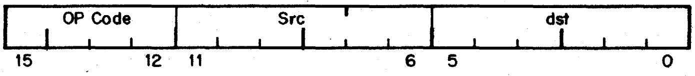
## 4.2 INSTRUCTION FORMATS

The major instruction formats are:

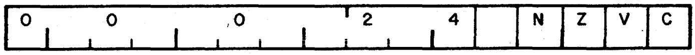
Single Operand Group



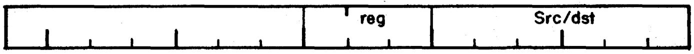
Double Operand Group



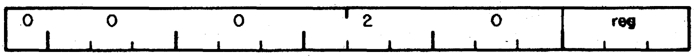
Condition Code Operators



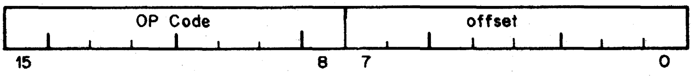
Register-Source or Destination



Subroutine Return

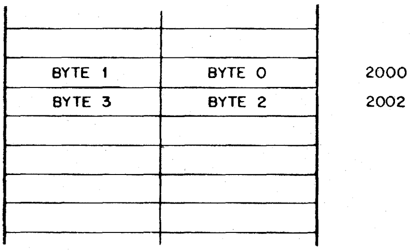


Branch



## 4.3 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 |
| CLRB | 1050DD |

**NOTE - ISP**

ISP - The Instruction Set Processor (ISP) notation has been used with each instruction. It is a precise notation for defining the action of any instruction set and is described in detail in Appendix C. It was included for the benefit of PDP-11 users who wish to gain an in depth understanding of each instruction. However, understanding ISP is not essential to understanding PDP-11 instructions.

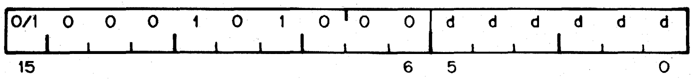
## 4.4 SINGLE OPERAND INSTRUCTIONS

| | | | | | | |
|---|---|---|---|---|---|---|
| General: | CLR | DEC | INC | NEG | TST | COM |
| | CLRB | DECB | INCB | NEGB | TSTB | COMB |
| Shifts: | ASR | ASL | | | | |
| | ASRB | ASLB | | | | |
| Multiple Precision: | ADC | SBC | | | | |
| | ADCB | SBCB | | | | |
| Rotates: | ROL | ROR | SWAB | | | |
| | ROLB | RORB | | | | |

### 4.4.1 Single Operand General Instructions

## CLR
## CLRB

Clear dst n050DD



**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 |

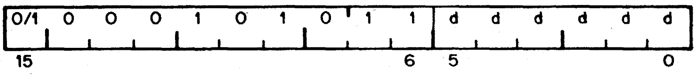
**ISP:**

```
CLR:
    D' ← 0;                     clear D, N, V, C, set Z
    N ← 0;
    Z ← 1;
    V ← 0;
    C ← 0
CLRB:
    Db' ← 0;                    clear D, N, V, C; set Z
    N ← 0;
    Z ← 1;
    V ← 0;
    C ← 0
```

2.3 μs

## DEC
## DECB

Decrement dst                                                                 n053DD



**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 |

**ISP:**

```
DEC:
   r ← D'-1; next                           result is difference of D-1
      N ← r<15>;                            negative?
     (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);    zero?
     (r<15:0> = $77777_8$) ⇒ (V ← 1 else V ← 0); overflow if largest positive number
      D ← r                                 transmit result to D
DECB:
   r ← Db'-1; next                          result is difference of D-1
      N ← r<7>;                             negative?
     (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);     zero?
     (r<7:0> = $177_8$) ⇒ (V ← 1 else V ← 0);  overflow if largest positive number
      Db ← r                                transmit result to D
```

2.3 μs

## INC
## INCB

Increment dst n052DD

| 15 | | | | | | | | | 6 | 5 | | | | | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0/1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | d | d | d | d | d | d |

**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

        NZVC                      NZVC
        0000                      0000
```

**ISP:**

```
INC:
  r ← D'+1; next                               result is sum of D+1
     N ← r<15>;                                negative?
    (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);        zero?
    (r<15:0> = 100000₈) ⇒ (V ← 1 else V ← 0);  overflow if largest negative number
     D ← r                                     transmit result to D
INCB:
  r ← Db+1; next                               result is sum of D+1
     N ← r<7>;                                 negative?
    (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);         zero?
    (r<7:0> = 200₈) ⇒ (V ← 1 else V ← 0);      overflow if largest negative number
     Db ← r                                    transmit result to D
```

2.3 μs

## NEG
## NEGB

Negate dst n0054DD

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0/1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | d | d | d | d | d | d |
| 15 | | | | | | | | | 6 | 5 | | | | | 0 |

**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 |

**ISP:**

```
NEG:
   r ← -D'; next                                   result is negative of D
      N ← r<15>;                                   negative?
      (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);          zero?
      (r<15:0> = $100000_8$) ⇒ (V ← 1 else V ← 0); overflow?
      (r<15:0> = 0) ⇒ (C ← 0 else C ← 1);          carry?
      D ← r                                        transmit result to D

NEGB:
   r ← - Db'; next                                 result is negative of D
      N ← r<7>;                                    negative?
      (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);           zero?
      (r<7:0> = $200_8$) ⇒ (V ← 1 else V ← 0);     overflow?
      (r<7:0> = 0) ⇒ (C ← 0 else C ← 1);           carry?
      Db ← r                                       transmit result to D
```

1.8 μs
2.3 μs if Mode 0

## TST
## TSTB

Test dst n057DD

| 15 | | | | | | | | | 6 | 5 | | | | | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0/1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | d | d | d | d | d | d |

**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 |

**ISP:**

```
TST:
   r ← D' - 0; next                              result is difference of D and 0
      N ← r<15>;                                 negative?
     (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);         zero?
      V ← 0;                                     clear V and C
      C ← 0

TSTB:
   r ← Db' - 0; next                             result is difference of D and 0
      N ← r<7>;                                  negative?
     (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);          zero?
      V ← 0;                                     clear V and C
      C ← 0
```

2.3 μs

## COM
## COMB

Complement dst n051DD

| 0/1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | d | d | d | d | d | d |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | | | | | | | | | 6 | 5 | | | | | 0 |

**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 |

**ISP:** 450 ns

```
COM:
   r ← ¬ D'; next                                  result is complement of D
      N ← r<15>;                                   negative?
     (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);           zero?
      V ← 0;                                       clear V
      C ← 1;                                       set C
      D ← r                                        transmit result to D
COMB:
   r ← ¬ Db'; next                                 result is complement of D
      N ← r<7>;                                    negative?
     (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);            zero?
      V ← 0;                                       clear V
      C ← 1;                                       set C
      Db ← r                                       transmit result to D
```

## 4.4.2 Shifts

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

ASR - Arithmetic shift right

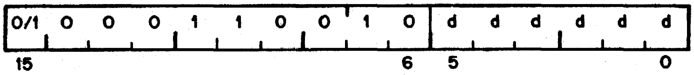
ASL - Arithmetic shift left

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.

2.3 μs
3.5 μs if odd byte

## ASR
## ASRB

Arithmetic Shift Right dst n062DD

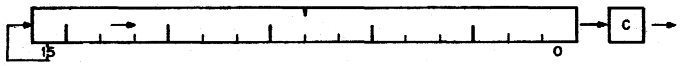


**Operation:** (dst)←(dst) shifted one place to the right

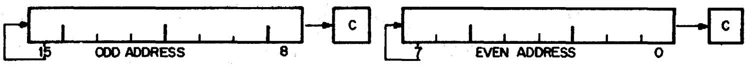
**Condition Codes:** N: set if the high-order bit of the result is set (result < 0); cleared otherwise
Z: set if the result = 0; cleared otherwise
V: loaded from the Exclusive OR of the N-bit and C-bit (as set by the completion of the shift operation)
C: loaded from low-order bit of the destination

**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:



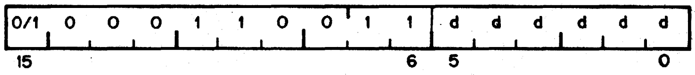
**ISP:**

```
ASR:
   r ← D'/2; next                          result is D/2
     C ← D<0>-;                            carry receives least significant bit
     N ← r<15>;                            negative?
     (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0); next   zero?
        (N ⊕ C) ⇒ (V ← 1 else V ← 0);      overflow is "Exclusive OR" of N and C
        D ← r

ASRB:
   r ← Db'/2; next                         result is D/2
     C ← Db<0>;                            carry receives least significant bit
     N ← r<7>;                             negative?
     (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0); next    zero?
        (N ⊕ C) ⇒ (V ← 1 else V ← 0);      overflow is "Exclusive OR" of N and C
        Db ← r
```

2.3 μs
3.5 μs if odd byte

## ASL
## ASLB

Arithmetic Shift Left dst n063DD

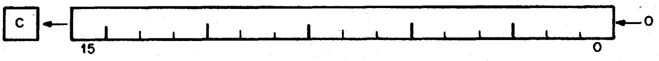


**Operation:** (dst)←(dst) shifted one place to the left

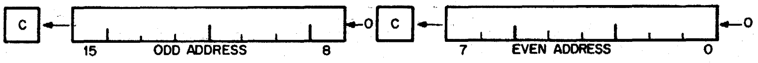
**Condition Codes:** N: set if high-order bit of the result is set (result < 0); cleared otherwise
Z: set if the result = 0; cleared otherwise
V: loaded with the exclusive OR of the N-bit and C-bit (as set by the completion of the shift operation)
C: loaded with the high-order bit of the destination

**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.
Byte: Same

Word:



Byte:



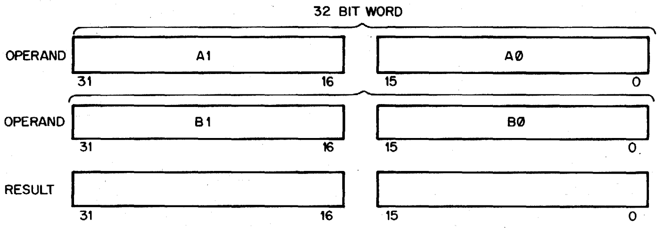
**ISP:**

```
ASL:
    r ← D'<15>□D'<13:0>□0; next                       result is DX2
        C ← D <14>; next                               bit squeezed out to C
           N ← r<15>;                                  negative?
          (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0); next     zero?
              (N ⊕ C) ⇒ (V ← 1 else V ← 0);            overflow is "Exclusive OR" of N and C
               D ← r                                   transmit result to D

ASLB:
    r ← Db'<7>□Db'<5:0>□0;next                         result is DX2
        C ← Db<6>; next                                bit squeezed out to C
           N ← r<7>;                                   negative?
          (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0); next       zero?
              (N ⊕ C) ⇒ (V ← 1 else V ← 0);            overflow is "Exclusive OR" of N and C
               Db ← r                                  transmit result to D
```

## 4.4.3 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:

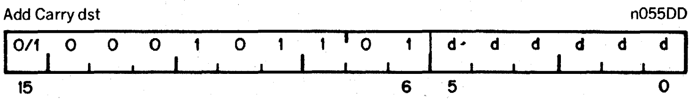
-1 = 37777777777

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

```
ADD   R1,R2 ;Add low order parts
ADC   R3    ;Add carry to high order part
ADD   R4,R3 ;Add high order parts
```

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



**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
```

**ISP:**

```
ADC:
   r ← D' + C; next
      N ← r<15>;                                       negative?
      (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);              zero?
      (r<15:0> = 100000₈) ∧ (C=1) ⇒ (V ← 1 else V ← 0); overflow if largest negative number
      (r<15:0> = 0) ∧ (C=1) ⇒ (C ← 1 else C ← 0);
      D ← r                                            transmit result to D

ADCB:
   r ← Db' + C; next
      N ← r<7>;                                        negative?
      (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);               zero?
      (r<7:0> = 200₈) ∧ (C=1) ⇒ (V ← 1 else V ← 0);    overflow if largest negative number
      (r<7:0> = 0) ∧ (C=1) ⇒ (C ← 1 else C ← 0);
      Db ← r                                           transmit result to D
```

2.3 μs

## SBC
## SBCB

Subtract Carry dst n056DD

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0/1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | d | d | d | d | d | d |
| 15 | | | | | | | | | 6 | 5 | | | | | 0 |

**Operation:** (dst)←(dst)−(C)

**Condition Codes:**
N: set if result <0; cleared otherwise
Z: set if result 0; cleared otherwise
V: set if result is 100000; cleared otherwise
C: cleared if result is 0 and C = 1; set 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
```

**ISP:**

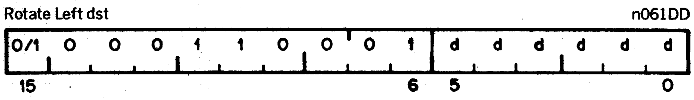
```
SBCB:
r ← Db' - C; next                                result is difference of D and C
   N ← r<7>;                                     negative?
   (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);            zero?
   (r<7:0> = 200₈) ⇒ (V ← 1 else V ← 0);         overflow?
   (r<7:0> = 0) ∧ (C=1) ⇒ (C ← 0 else C ← 1);
   Db ← r                                        transmit result to D
```

### 4.4.4 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.

2.3 μs
3.5 μs if odd byte

## ROL
## ROLB



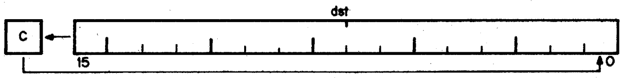
**Operation:** (dst)←(dst) rotated left one place

**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

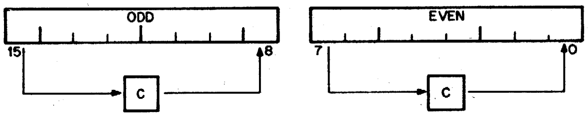
**Description:** Word: Rotate all bits of the destination left one place. Bit 15 is loaded into the C-bit of the status word and the previous contents of the C-bit are loaded into Bit 0 of the destination.
Byte: Same

**Example:**

Word:



Bytes:



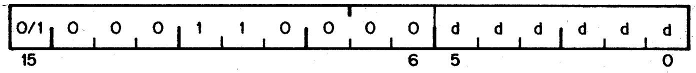
**ISP:**

```
•
ROL:
   r<16:0> ← D'<15:0>□C; next                result is D and C rotated
     N ← r<15>;                              negative?
     (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);     zero?
     C□D ← r; next                           transmit result to C and D
        (N ⊕ C) ⇒ (V ← 1 else V ← 0)         V is based on new result of N and C


ROLB:
   r<8:0> ← Db'<7:0>□C; next                 result is D and C rotated
     N ← r<7>;                               negative?
     (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);      zero?
     C□Db ← r; next                          transmit result to C and D
        (N ⊕ C) ⇒ (V ← 1 else V ← 0)         V is based on new result of N and C
```

2.3 μs
3.5 μs if odd byte

## ROR
## RORB

Rotate Right dst n060DD



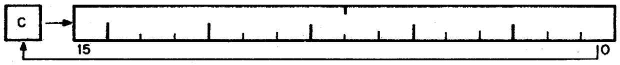
**Operation:** (dst)←(dst) rotated right one place

**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

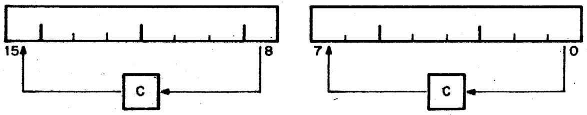
**Description:** Rotates all bits of the destination right one place. Bit 0 is loaded into the C-bit and the previous contents of the C-bit are loaded into bit 15 of the destination.
Byte: Same

**Example:**

Word:



Byte:



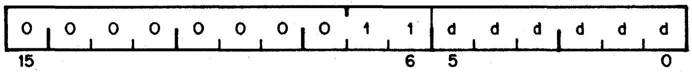
**ISP:**

```
ROR:
   r<16:0> ← D'<0>□C□D'<15:1>; next          result is D and C rotated
      N ← r<15>;                             negative?
     (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);      zero?
      C□D<15:0> ← r; next                     transmit result to C and D
         (N ⊕ C) ⇒ (V ← 1 else V ← 0)         V is based on new result of N and C
RORB:
   r<8:0> ← Db'<0>□C□Db'<7:1>; next          result is D and C rotated
      N ← r<7>;                              negative?
     (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);       zero?
      C□Db ← r; next                          transmit result to C and D
         (N ⊕ C) ⇒ (V ← 1 else V ← 0)         V is based on new result of N and C
```

2.3 μs

## SWAB

Swap Bytes dst 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 |

**ISP:**

```
SWAB:
  r ← D'<7:0>□D'<15:8>; next                result is byte swapped of D
     N ← r<7>;                               negative?
     (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);      zero?
     V ← 0;                                  clear V, C
     C ← 0;
        D ← r                                transmit result to D
```

## 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.

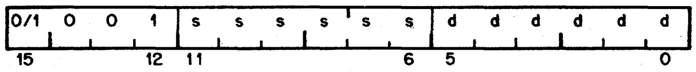
| | | | | |
|---|---|---|---|---|
| General: | MOV | ADD | SUB | CMP |
| | MOVB | | | CMPB |
| Logical: | BIS | BIT | BIC | |
| | BISB | BITB | BICB | |

### 4.5.1 Double Operand General Instructions

2.3 μs

## MOV
## MOVB

Mov src, dst n1SSDD



**Operation:** (dst)←(src)

**Condition Codes:**
N: set if (src) <0; cleared otherwise
Z: set if (src) =0; cleared otherwise
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 interregister transfer

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

## ISP:

```
MOVE:
   r[1] ← S'[2]; next                          move source  to intermediate result register, r
      N ← r<15>;                               negative?
     (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0)[4];    zero - if 16 bits of r are all zero then Z is set
                                                to 1 else Z is set to 0
      V ← 0;                                   overflow is cleared
      D' ← r                                   transmit result to destination

MOVB:
   r ← Sb'; next                               move source to intermediate result
      N ← r<7>;                                negative?
     (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);        zero?
      V ← 0;                                   clear V
      Db' ← r                                  transmit result to Db
```

2.3 μs

## ADD

Add src. dst  06SSDD

| 0 | 1 | 1 | 0 | s | s | s | s | s | s | d | d | d | d | d | d |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | | | 12 | 11 | | | | | 6 | 5 | | | | | 0 |

**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.

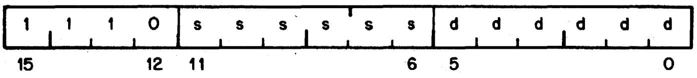
**ISP:**

```
ADD:
   r<16:0> ← S' + D'; next                  determine intermediate result sum of 17 bits
     N ← r<15>;                             negative?
     (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);    zero?
     (S<15> ≡ D<15>) ∧ (S<15> ⊕ r<15>) ⇒ (   overflow? if signs of operands agree and sign of
         V ← 1 else V ← 0);                   an operand and the sign of the result disagree
                                              then set V to 1 else set V to 0
     C ← r<16>;                             carry the 17th bit
     D ← r                                  transmit result to D
```

2.3 $\mu s$

## SUB

Subtract src, dst 16SSDD



**Operation:** (dst)←(dst)–(src) [in detail, (dst) + ~(src) + 1 (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 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 1 |

**ISP:**

```
SUB:
   r ← D' - S'; next                                 17 bit result is D minus S; actually r ← ¬ S+D+1;
     N ← r<15>;                                      negative?
    (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);              zero?
    (D<15> ≡ ¬ S<15>) ∧ (D<15> ⊕ r<15>) ⇒ (          overflow? (see add)
        V ← 1 else V ← 0);
     C ← r<16>;                                      borrow from 17th bit
     D ← r                                           move result to D
```

1.8 μs
2.3 μs if Mode 0

## CMP
## CMPB

Compare src. dst                                                                                                                   n2SSDD

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0/1 | 0 | 1 | 0 | s | s | s | s | s | s | d | d | d | d | d | d |
| 15 | | | 12 | 11 | | | | | 6 | 5 | | | | | 0 |

**Operation:** (src)–(dst) [in detail, (src) + ~ (dst) + 1]

**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).

**ISP:**

```
CMPB:
  r<8:0> ← Sb' - Db'; next                          compare affects CC only
    N ← r<7>;                                       negative?
    (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);              zero?
    (Sb<7> ≡ ¬ Db<7>) ∧ (Sb<7> ⊕ r<7>) ⇒ (           overflow? (see add)
       V ← 1 else V ← 0);
    C ← r<8>                                        8th bit is carry

CMP:
  r ← S' - D'; next                                 compare affects CC only
    N ← r<15>;                                      negative?
    (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);             zero?
    (S<15> ≡ ¬ D<15>) ∧ (S<15> ⊕ r<15>) ⇒ (          overflow? (see add)
       V ← 1 else V ← 0);
       C ← r<16>                                    17th bit is carry
```

### 4.5.2 Logical Instructions

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

2.3 μs

## BIS
## BISB

Bit Set src. dst n5SSDD

| 15 | | | 12 | 11 | | | | | 6 | 5 | | | | | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0/1 | 1 | 0 | 1 | s | s | s | s | s | s | d | d | d | d | d | d |

**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 |
| NZVC | NZVC |
| 0000 | 0000 |

**ISP:**

```
BIS:
   r ← D' V S'; next                          result is S "OR" D
      N ←r<15>;                               negative?
      (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);     zero?
      V ← 0;                                  clear V
      D ← r                                   transmit result to D
BISB:
   r ← Db' V Sb'; next                        result is S "OR" D
      N ← r<7>;                               negative?
      (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);      zero?
      V ← 0;                                  clear V
      Db ← r                                  transmit result to D
```

2.4 μs
2.9 μs if Mode 0

## BIT
## BITB

Bit Test src. dst n3SSDD

| 15 | | | 12 | 11 | | | | | 6 | 5 | | | | | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0/1 | 0 | 1 | 1 | s | s | s | s | s | s | d | d | d | d | d | d |

**Operation:** (dst)←(src)$\wedge$(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

BEQ     HELP            ; BEQ to HELP will occur if
                        ; both are off
```

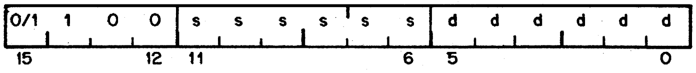
**ISP:**

```
BIT:
   r ← D' ∧ S'; next                       test result is "AND" of D and S
      N ← r<15>;                           negative?
      (r<15: 0> = 0) ⇒ (Z ← 1 else Z ← 0);  zero?
      V ← 0                                clear V

BITB:
   r ← Db' ∧ Sb'; next                     test result is "AND" of D and S
      N ← r<7>;                            negative?
      (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);    zero?
      V ← 0                                clear V
```

2.9 μs

## BIC
## BICB

Bit Clear src dst n4SSDD



**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 |

**ISP:**

```
BIC:
   r ← D' ∧ ¬ S'; next                            result is D "AND" "NOT" S
      N ← r<15>;                                  negative?
      (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);         zero?
      V ← 0;                                      clear V
      D ← r                                       transmit result to D
BICB:
   r ← Db' ∧ ¬ Sb'; next                          result is D "AND" "NOT" S
      N ← r<7>;                                   negative?
      (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);          zero?
      V ← 0;                                      clear V
      Db ← r                                      transmit result to D
```

## 4.6 PROGRAM CONTROL INSTRUCTIONS

### 4.6.1 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.

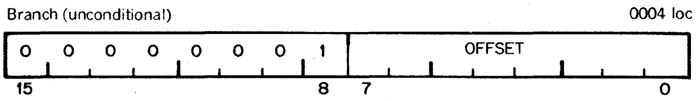
The PDP-11 assembler handles address arithmetic for the user and computes and assembles the proper offset field for branch instructions in the form:

```
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.

2.6 μs

## BR



**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.

**Example:**

```
     001000    BR xxx
     001002
     001004
xxx: 001006
     001010
```

**ISP:**

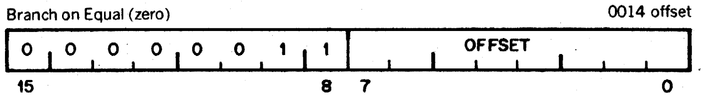
```
BR:
   PC ← PC + sign-extend(instruction<7:0> x 2)⁴
```

**Simple Conditional Branches**

BEQ
BNE
BMI
BPL
BCS
BCC
BVS
BVC

1.5 μs -- no branch
2.6 μs -- branch

## BEQ



**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.

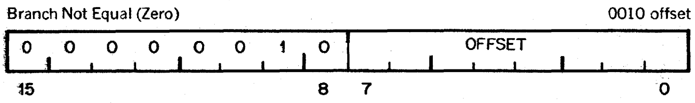
**ISP:**

BEQ:

(Z=1) $\Rightarrow^{b}$ (PC ← PC + sign-extend(instruction<7:0> x 2))

1.5 μs -- no branch
2.6 μs -- branch

## BNE



**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 ≠ B and the sequence

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

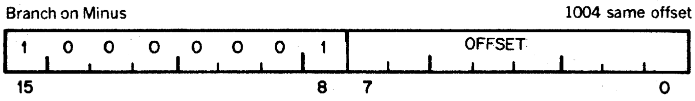
will branch to C if A + B = 0

**ISP:**

```
BNE:
    (Z=0) ⇒ (PC ← PC + sign-extend(instruction<7:0> x 2))
```

1.5 μs -- no branch
2.6 μs -- branch

## BMI



**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).

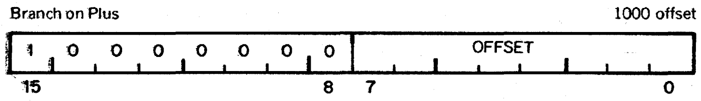
**Example:**

**ISP:**

```
BMI:
   (N=1) ⇒ (PC ← PC + sign-extend(instruction<7:0> × 2))
```

1.5 μs -- no branch
2.6 μs -- branch

## BPL



**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. BPL is the complementary operation of BMI.

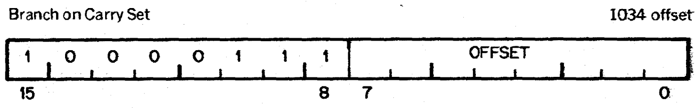
**Example:**

**ISP:**

```
BPL:
   (N=0) ⇒ (PC ← PC + sign-extend(instruction<7:0 x 2))
```

1.5 μs -- no branch
2.6 μs -- branch

## BCS



**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.

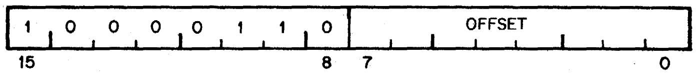
**ISP:**

```
BCS:
    (C=1) ⇒ (PC ← PC + sign-extend(instruction<7:0> x 2))        if C=1 then branch
```

1.5 µs -- no branch
2.6 µs -- branch

## BCC

Branch on Carry Clear 1030 offset



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

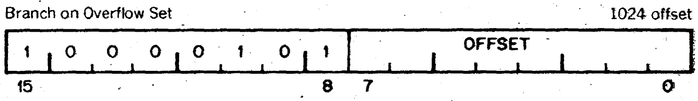
**Description:** Tests the state of the C-bit and causes a branch if C is clear. BCC is the complementary operation to BCS

**ISP:**

```
BCC:
    (C=0) ⇒ (PC ← PC + sign-extend(instruction<7:0> X 2))
```

1.5 $\mu$s -- no branch
2.6 $\mu$s -- branch

## BVS



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

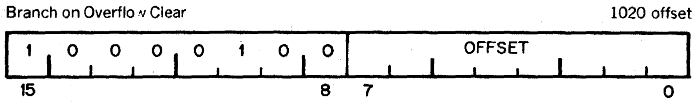
**Description:** Tests the state of V bit (overflow) and causes a branch if the V bit is set. BVS is used to detect arithmetic overflow in the previous operation.

**ISP:**

```
BVS:
   (V=1) ⇒ (PC ← PC + sign-extend(instruction<7:0> x 2))
```

1.5 μs -- no branch
2.6 μs -- branch

## BVC



**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.

**ISP:**

```
BVC:
    (V=0) ⇒ (PC ← PC + sign-extend(instruction<7:0> x 2))
```

**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 a 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 |

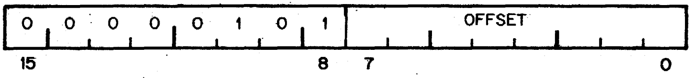
The signed conditional branch instructions are:

| | |
|---|---|
| BLT | BGE |
| BLE | BGT |

1.5 μs -- no branch
2.6 μs -- branch

## BLT

Branch on Less Than (Zero) 0024 offset



**Operation:** $PC \leftarrow PC + (2 \times \text{offset})$ if $N \veebar V = 1$

**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).

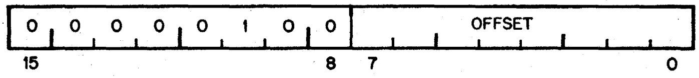
**ISP:**

```
BLT:
  (N ⊕ V) ⇒ PC ← PC + sign-extend(instruction<7:0> x 2))
```

1.5 μs -- no branch
2.6 μs -- branch

## BGE

Branch on Greater than or Equal (zero) 0020 offset



**Operation:** $PC \leftarrow PC + (2 \times offset)$ if $N \veebar V = 0$

**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 to two positive numbers. BGE will also cause a branch on a zero result.

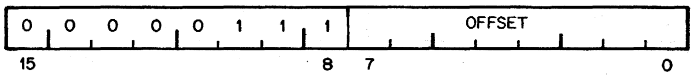
**ISP:**

```
BGE:
   (N ≡ V) ⇒ (PC ← PC + sign-extend(instruction<7:0> x 2))
```

1.5 μs -- no branch
2.6 μs -- branch

## BLE

Branch on Less than or Equal (zero) 0034 offset



**Operation:** $PC \leftarrow PC + (2 \times \text{offset})$ if $Z \vee (N \veebar V) = 1$

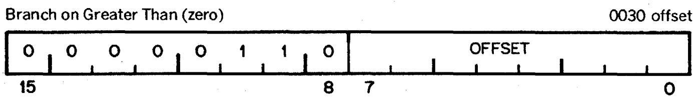
**Description:** Operation is similar to BLT but in addition will cause a branch if the result of the previous operation was zero.

**ISP:**

```
BLE:
   (Z V (N ⊕ V) ⇒ (PC ← PC + sign-extend(instruction<7:0> x 2))
```

1.5 μs -- no branch
2.6 μs -- branch

## BGT



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

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

**ISP:**

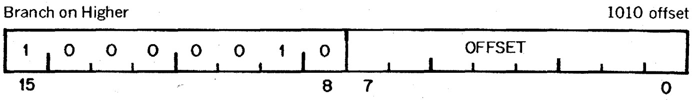
```
BGT:
   ¬(Z ∨ (N ⊕ V)) ⇒ (PC ← PC + sign-extend(instruction<7:0> x 2))
```

**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
BLOS
BHIS
BLO

1.5 μs -- no branch
2.6 μs -- branch

## BHI



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

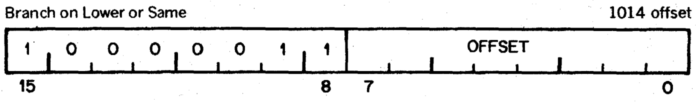
**Description:** Causes a branch if the previous operation caused neither a carry nor a zero result. This will happen in comparison (CMP) operations as long as the source has a higher unsigned value than the destination.

**ISP:**

```
BHI:
  ¬(C V Z) ⇒ (PC ← PC + sign-extend(instruction<7:0> x 2))
```

1.5 µs -- no branch
2.6 µs -- branch

## BLOS



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

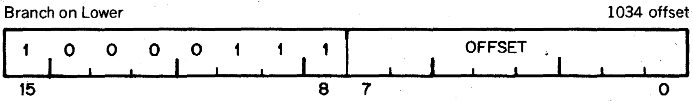
**Description:** Causes a branch if the previous operation caused either a carry or a zero result. BLOS is the complementary operation to BHI. The branch will occur in comparison operations as long as the source is equal to, or has a lower unsigned value than the destination.
Comparison of unsigned values with the CMP instruction can be tested for "higher or same" and "higher" by a simple test of the C-bit.

**ISP:**

```
BLOS:
   (C ∨ Z) ⇒ (PC ← PC + sign-extend(instruction<7:0> x 2))
```

1.5 μs -- no branch
2.6 μs -- branch

## BLO



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

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

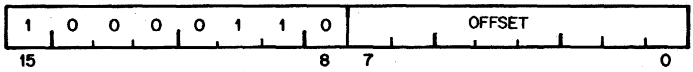
**ISP:**

```
BCS/BLO:
   (C=1) ⇒ (PC ← PC + sign-extend(instruction<7:0> x 2))
```

1.5 μs -- no branch
2.6 μs -- branch

## BHIS

Branch on Higher or Same 1030 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.

**ISP:**

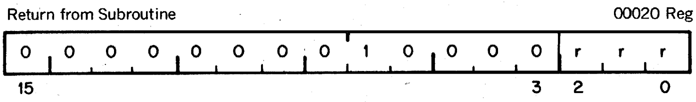
```
BCC/BHIS:
   (C=0) ⇒ (PC ← PC + sign-extend(instruction<7:0> x 2))
```

### 4.6.2 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.

3.5 μs

## RTS



**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 RS.

**ISP:**

```
RTS:
   PC ← R[dr];                                   return jump
   R[dr] ← Ms[SP];                               unstack (pop) R[dr]
   SP ← SP + 2
```

## JSR

Jump to Sub Routine 004 reg. dst

| 15 | | | | | | 9 | 8 | | 6 | 5 | | | | | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | r | r | r | d | d | d | d | d | d |

**Operation:** (tmp)←(dst) (tmp is an internal processor register)

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

reg←PC (PC holds location following JSR; this address

PC←(tmp) now put in reg)

**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.

In both JSR and JMP instructions the destination address is used to load the program counter, R7. Thus for example a JSR in destination mode 1 for general register R1 (where (R1) = 100), will access a subroutine at location 100. This is effectively one level less of deferral than operate instructions such as ADD.

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."

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.

JSR used in address mode 2 (autoincrement), increments the register before using it as an address. This is a special case, and is only true of one other instruction (JMP)

**ISP:**

```
JSR:
    SP ← SP - 2; next
       Mw6[SP] ← R[sr];                                stack (push) R[sr];
       R[sr] ← PC;                                     load R[sr] with Pc
       PC ← Daddress'                                  jump
```

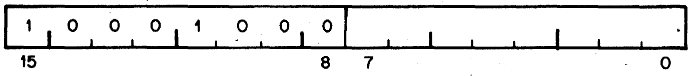
### 4.6.3 Traps

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 (PSW) are pushed onto the processor stack and replaced by the contents of a two-word trap vector containing a new PC and new PSW. The return sequence from a trap involves executing an RTI instruction which restores the old PC and old PSW by popping them from the stack. Trap vectors are located permanently assigned fixed address.

TRAP
EMT
IOT

## EMT

Emulator Traps 104000-104377



**Operation:**
$\downarrow$ (SP)←PS
$\downarrow$ (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.

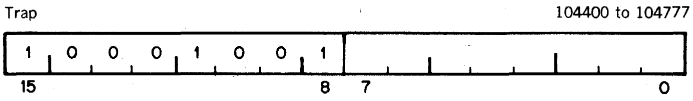
Caution: EMT is used frequently by DEC system software and is therefore not recommended for general use.

**ISP:**

```
EMT:
   SP ← SP-2; next                       place
      Mw[SP] ← PS;                        PS and
      SP ← SP-2; next
         Mw[SP] ← PC;                     PC on stack
         PC ← Mw[30];                    take new PC and PS from M[30], M[32]
         PS ← Mw[32]
```

2.25 μs

## TRAP



**Operation:** $\downarrow(SP) \leftarrow PS$
$\downarrow(SP) \leftarrow PC$
$PC \leftarrow (34)$
$PS \leftarrow (36)$

**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.

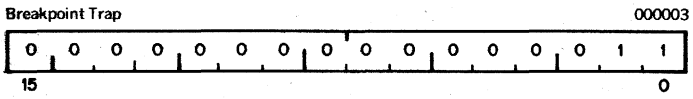
Note: Since DEC software makes frequent use of EMT, the TRAP instruction is recommended for general use.

**ISP:**

```
TRAP:
   SP ← SP-2; next                       place (push)
      Ms[SP] ← PS;                        PS and
      SP ← SP-2; next
         Mw[SP] ← PC;                     PC on stack
         PC ← Mw[34]                     take new PC and PS from M[34], M[36]
         PS ← Mw[36]
```

9.3 μs

(No mnemonic)



**Operation:** $\downarrow$(SP)$\leftarrow$PS
$\downarrow$(SP)$\leftarrow$PC
PC $\leftarrow$(14)
PS $\leftarrow$(16)

**Condition Codes:** N: loaded from trap vector
Z: loaded from trap vector
V: loaded from trap vector
C: loaded from trap vector

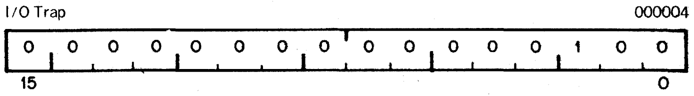
**Description:** Performs a trap sequence with a trap vector address of 14. Used to call debugging aids. The user is cautioned against employing code 000003 in programs run under these debugging aids.

**ISP:**

```
BPT:
   SP ← SP - 2; next                         place
      Mw[SP] ← PS;                            PS and
      SP ← SP - 2; next
         Mw[SP] ← PC;                         PC on stack
         PC ← Mw[14₈];                        take new PC and PS from M[14], M[16]
         PS ← Mw[16₈]
```

9.3 μs

## IOT



**Operation:** $\downarrow$(SP)$\leftarrow$PS
$\downarrow$(SP)$\leftarrow$PC
PC$\leftarrow$(20)
PS$\leftarrow$(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.

**ISP:**

```
IOT:
   SP ← SP-2; next                               place
      Mw[SP] ← PS;                                PS and
      SP ← SP-2; next
         Mw[SP] ← PC;                             PC on stack
         PC ← Mw[20];                            take new PC and PS from M[20], M[22]
         PS ← Mw[22]
```

**Reserved Instruction Traps** - These are caused by attempts to execute instruction codes reserved for future processor expansion (reserved instructions) or instructions with illegal addressing modes (illegal instructions). Order codes not corresponding to any of the instructions described are considered to be reserved instructions. JMP and JSR with register mode destinations are illegal instructions. Reserved and illegal instruction traps occur as described under EMT, but trap through vectors at addresses 10 and 4 respectively.

**Stack Overflow Trap** Stack Overflow Trap is a processor trap through the vector at address 4. It is caused by referencing addresses below $400_8$ through the processor stack pointer R6 (SP) in autodecrement or autodecrement deferred addressing. The instruction causing the overflow is completed before the trap is made.

**Bus Error Traps** - Bus Error Traps are:

1. Boundary Errors - attempts to reference word operands at odd addresses.

2. Time-Out Errors - attempts to reference addresses on the bus that made no response within 10μs in the PDP-11. In general, these are caused by attempts to reference non-existent memory, and attempts to reference non-existent peripheral devices.

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

**Trace Trap** - Trace Trap enables bit 4 of the PSW and causes processor traps at the end of instruction executions. The instruction that is executed after the instruction that set the T-bit will proceed to completion and then cause a processor trap through the trap vector at address 14. Note that the trace trap is a system debugging aid and is transparent to the general programmer.

The following are special cases and are detailed in subsequent paragraphs.

1. The traced instruction cleared the T-bit.

2. The traced instruction set the T-bit.

3. The traced instruction caused an instruction trap.

4. The traced instruction caused a bus error trap.

5. The traced instruction caused a stack overflow trap.

6. The process was interrupted between the time the T-bit was set and the fetching of the instruction that was to be traced.

7. The traced instruction was a WAIT.

8. The traced instruction was a HALT.

Note: The traced instruction is the instruction after the one that sets the T-bit.

**An instruction that cleared the T-bit** - Upon fetching the traced instruction an internal flag, the trace flag, was set. The trap will still occur at the end of execution of this instruction. The stacked status word, however, will have a clear T-bit.

**An instruction that set the T-bit** - Since the T-bit was already set, setting it again has no effect. The trap will occur.

**An instruction that caused an Instruction Trap** - The instruction trap is sprung and the entire routine for the service trap is executed. If the service routine exists with an RTI or in any other way restores the stacked status word, the T-bit is set again, the instruction following the traced instruction is executed and, unless it is one of the special cases noted above, a trace trap occurs.

**An instruction that caused a Bus Error Trap** - This is treated as an Instruction Trap. The only difference is that the error service is not as likely to exit with an RTI, so that the trace trap may not occur.

**An instruction that caused a stack overflow** - The instruction completes execution as usual - the Stack Overflow does not cause a trap. The Trace Trap Vector is loaded into the PC and PS, and the old PC and PS are pushed onto the stack. Stack Overflow occurs again, and this time the trap is made.

**An interrupt between setting of the T-bit and fetch of the traced instruction** - The entire interrupt service routine is executed and then the T-bit is set again by the exiting RTI. The traced instruction is executed (if there have been no other interrupts) and, unless it is a special case noted above, causes a trace trap.

Note that no interrupts are acknowledged between the time of fetching any trapped instruction (including one that is trapped by reason of the T-bit being set) and completing execution of the first instruction of the trap service.

**A WAIT** The trap occurs immediately. The address of the next instruction is saved on the stack

**A HALT** - The processor halts. When the continue key on the console is pressed, the instruction following the HALT is fetched and executed. Unless it is one of the exceptions noted above, the trap occurs immediately following execution.

**Power Failure Trap** - is a standard PDP-11 feature. Trap occurs whenever the AC power drops below 105 volts or outside 47 to 63 Hertz. 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 processor trap conditions occur simultaneously the following order of priorities is observed (from high to low):

1. Bus Errors
2. Instruction Traps
3. Trace Trap
4. Stack Overflow Trap
5. Power Failure Trap

The details on the trace trap process have been described in the trace trap operational description which includes cases in which an instruction being traced causes a bus error, instruction trap, or a stack overflow trap.

If a bus error is caused by the trap process handling instruction traps, trace traps, stack overflow traps, or a previous bus error, the processor is halted.

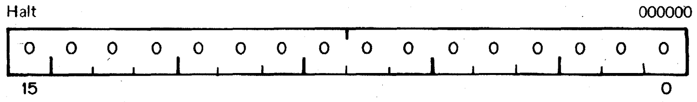
If a stack overflow is caused by the trap process in handling bus errors, instruction traps, or trace traps, the process is completed and then the stack overflow trap is sprung.

## 4.7 Miscellaneous

HALT
WAIT
RESET
JMP
RTI

1.8 μs

## HALT



**Condition Codes:** not affected

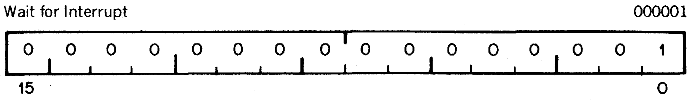
**Description:** Causes the processor operation to cease. The console is given control of the bus. The console data lights display the contents of R0; the console address lights display the address after the halt instruction. Transfers on the UNIBUS are terminated immediately. The PC points to the next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

**ISP:**

```
Off ← true                                          set activity to Off state
                                                    no more instructions can
                                                    be executed until a con-
                                                    sole action takes place
                                                    to restart processor
```

1.8 $\mu s$

## WAIT



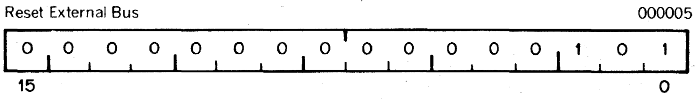
**Condition Codes:** not affected

**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 PSW to be pushed onto the processor ation. from the interrupt routine (i.e. execution of an RTI instruction) will cause resumption of the interrupted process at the instruction following the WAIT.

**ISP:**

```
WAIT:
   Wait ← true                                   set activity to Wait state; interrupts
                                                  can occur
```

## RESET



**Condition Codes:** not affected

**Description:** Sends INIT on the UNIBUS for 20ms. All devices on the UNIBUS are reset to their state at power up.
At the end of a reset sequence an effective halt is executed.

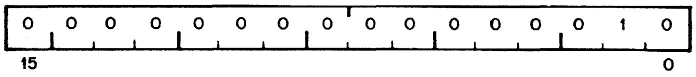
**ISP:**

```
Reset:
    Init ← 1;                                  cause a signal, Init, to be one for
    Delay (so milliseconds); next               20 milliseconds
        Init ← 0
```

4.8 μs

## RTI

Return from Interrupt 000002



**Operation:** $PC \leftarrow (SP)\uparrow$
$PSW \leftarrow (SP)\uparrow$

**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 PSW are restored (popped) from the processor stack. If a trace trap is pending, the first instruction after the RTI will be executed prior to the next "T" Trap.

**ISP:**

```
RTI:
PC ← Mw[SP];                              unstack (pop) PC for jump
SP ← SP + 2; next
   PS ← Mw[SP];                           unstack (pop) PS
   SP ← SP + 2;
   T-trap-inhibit ← true                  inhibit T-trap for 1 instruction
```

## JMP

Jump 0001DD

| 15 | | | | | | | | | 6 | 5 | | | | | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | d | d | d | d | d | d |

**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.

**ISP:**

```
JMP:
   PC ← Daddress'                                       Daddress is computed in a fashion
                                                         similar to D
```

## 4.8 Condition Code Operators

1.5 μs

**CLC SEC**
**CLZ SEZ**
**CLN SEN**
**CLV SEV**

Condition Code Operators 0002XX

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0/1 | N | Z | V | C |
| 15 | | | | | | | | | | 5 | 4 | 3 | 2 | 1 | 0 |

**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 |
| | Set all CC's | 000277 |
| | Clear all CC's | 000257 |
| | Clear V and C | 000243 |
| | No operation | 000240 |
| | No operation | 000260 |

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

## ISP:

```
CLC:
   ¬ i<4> ∧ i<0> ⇒ C ← 0                    clear C
CLN:
   ¬ i<4> ∧ i<3> ⇒ N ← 0                    clear N
CLV:
   ¬ i<4> ∧ i<1> ⇒ V ← 0                    clear V
CLZ:
   ¬ i<4> ∧ i<2> ⇒ Z ← 0                    clear Z

SEC:
     i<4> ∧ i<0> ⇒ C ← 1                    set C
SEN:
     i<4> ∧ i<3> ⇒ N ← 1                    set N
SEV:
     i<4> ∧ i<1> ⇒ V ← 1                    set V
SEZ:
     i<4> ∧ i<2> ⇒ Z ← 1                    set Z
```

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.

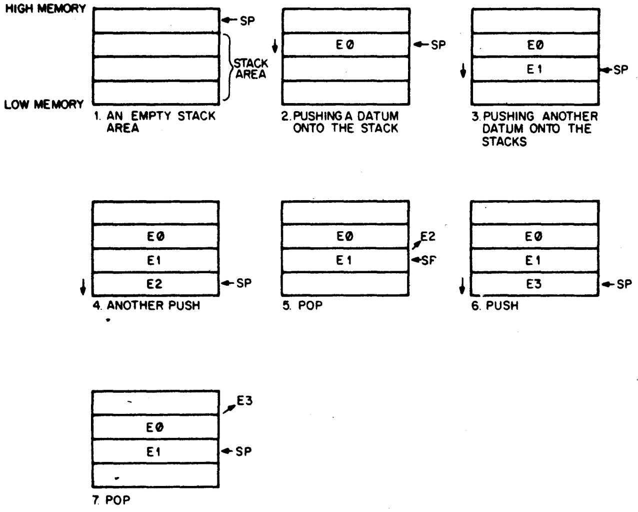


Figure 5-3: Illustration of Push and Pop Operations

PART 1
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.

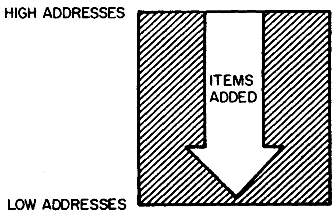


Figure 5-1: Stack Addresses

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."

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.

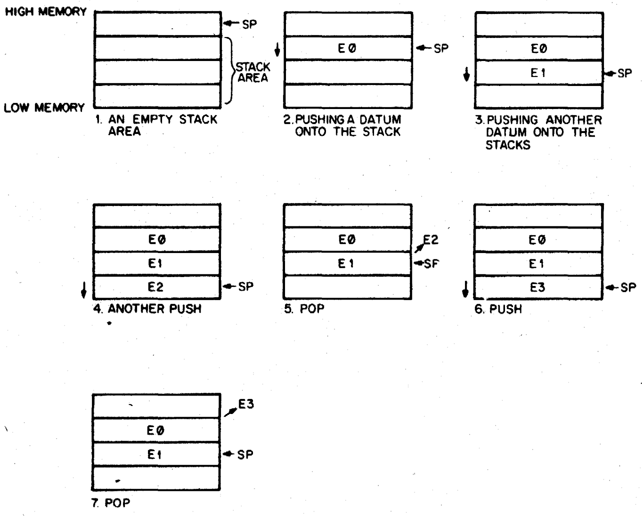


Figure 5-3: Illustration of Push and Pop Operations

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         000072                *
076326         010267                MOV R2,TEMP2 ;save R2
076330         000070                *
.              .                     .
.              .                     .
.              .                     .
.              .                     .
076410         016701                MOV TEMP1, R1 ;Restore R1
076412         000006                *
076414         016702                MOV TEMP2, R2 ;Restore R2
076416         000004                *
076410         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         012302                MOV(R3)+, R2 ;pop R2
010132         012301                MOV(R3)+,R1 ;pop R1

010134         000207                RTS PC
```

Note: In this case R3 was used as the 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.

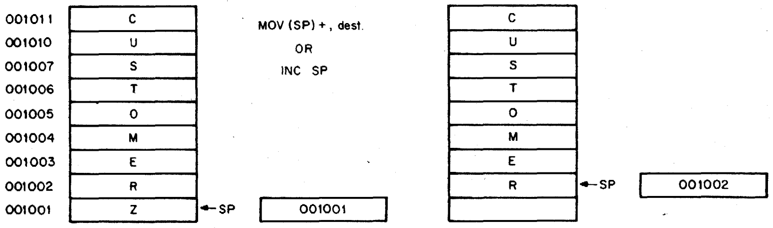


Figure 5-6: Byte Stack used as a Character Buffer

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 SUBROUTINES 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 | 004767 |
| 001002 | Index constant for SUBR | 000064 |
| 001064 | SUBR:MOV A,B | 01mmnn |

Figure 5-7: JSR using R0-R5

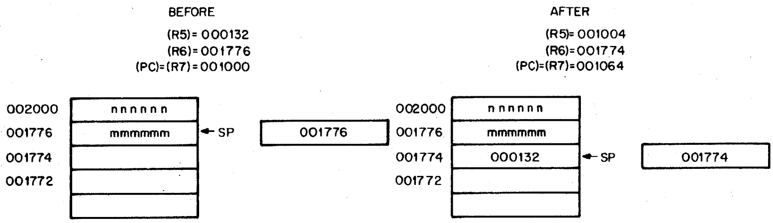


Figure 5-8: JSR

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

  010404        arg #1
  010406        arg #2               ARGUMENTS
    .             .
    .             .

  020306        SUBR:   MOV (R5)+,R1 ;get arg #1
  020301        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

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 Retrieve Arguments
                                        ;from SUB
```

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

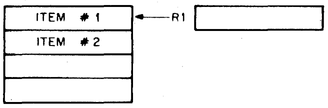
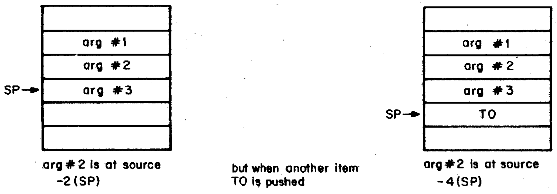


Figure 5-11: Transmitting Stacks as Arguments

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.



arg #1
arg #2
SP→
arg #3
arg #2 is at source -2(SP)
but when another item TO is pushed
arg #1
arg #2
arg #3
SP→
TO
arg #2 is at source -4(SP)

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.

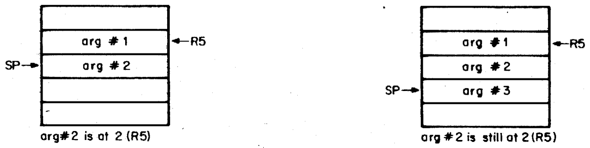


Figure 5-13: Constant Index Base Using "R6 Copy"

### 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 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.

### 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 (e.g. SQRT in FORTRAN SQRT(SQRT(X)). 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 occuring 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 negative⅓, etc.) This information is stored in Processor Status Word (PSW). Upon interrupt, the contents of the Program Counter (PC) (address of next instruction and the Processor Status Word (PSW) 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
```

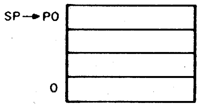
had been executed.

The new contents of the Program Counter (PC) and Processor Status Word (PSW) 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 (see interrupt vector list, Appendix D). The first word contains the interrupt service routine address (the address of the new program sequence) and the second word contains the new Processor Status Word (PSW) which will determine the machine status at the start of the interrupt service routine. The contents of the interrupt service vector is set under program control.

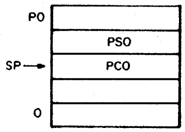
After the interrupt service routine has been completed, an RTI (return from interrupt) is performed. The two top words of the stack are automatically "popped" and placed in the PC and PS respectively , thus resuming the interrupted program.

**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.

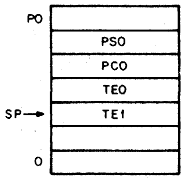
1. Process O is running; Stack Pointer (SP) points to location PO.



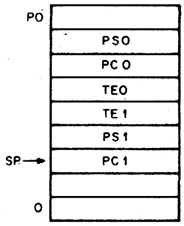
2. Interrupt stops process O with PC = PC(O) and status = PS(O); starts process 1.



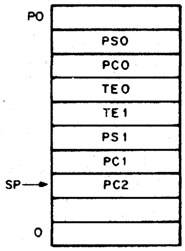
3. Process 1 uses stack for temporary storage (TEO,TE1).



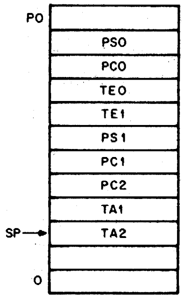
4. Process 1 interrupted with PC = PC(1) and status = PS1; process 2 is started.



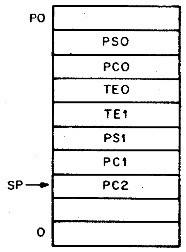
5. Process 2 is running and does a JSR R7, A to subroutine A with PC = PC(2).



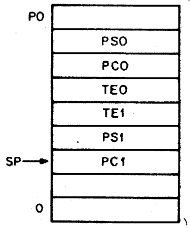
6. Subroutine A is running and uses the stack for temporary storage.



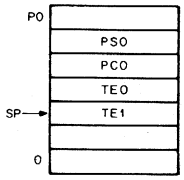
7. Subroutine A releases the temporary storage holding TA1 and TA2.



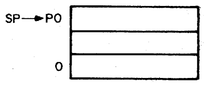
8. Subroutine A returns control to process 2 with an RTS R7, PC is reset to PC2.



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



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



11. Process 1 completes its operation with an RTI, PC is reset to PCO and status is reset to PS(O).

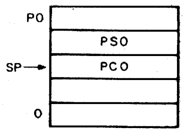


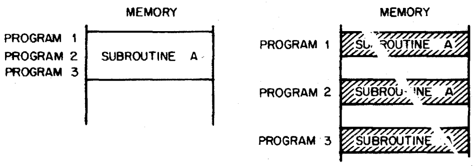
Figure 5-14: Nested Interrupt Service Routines and Subroutines

Note that the area of interrupt service programming is intimately involved with the concept of CPU and device priority levels. For a full discussion of the uses of the PDP-11 priority structure, refer to Chapter 2, System Architecture.

## 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 of 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.

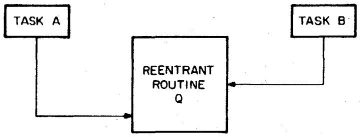


Figure 5-16: Reentrant Routine Sharing

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 an 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 to 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.

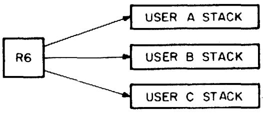


Figure 5-17: Multiple R6 Stack

## 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 in 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 RECURSION

It is often meaningful for a program routine to call itself as in the case of calculating a fourth root in FORTRAN with the expression SQRT(SQRT(X)). The ability to nest subroutine calls to the same subroutine is called recursion. The use of stack organization permits easy unambiguous recursion. The technique of recursion is of great use to the mathematical analyst as it also permits the evaluation of some otherwise non-computable mathematical functions. Although it is beyond the scope of this chapter to discuss the concept of recursive routines in detail, the reader should realize that this technique often permits very significant memory and speed economies in the linguistic operations of compilers and other higher-level software programs.

## 5.7 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 can be illustrated as in Figure 5-18.

Routine #1 is operating, it then executes:

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 PC(2) (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.

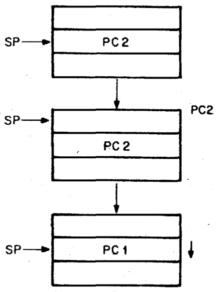


Figure 5-18: Co-Routine Interaction

PART I
CHAPTER 6

# SPECIFICATIONS

Physically, the PDP-11 is composed of a number of System Units. Each System Unit is composed of three eight-slot connector blocks mounted end-to-end as shown in Figure 6-1. The UNIBUS connects to the System Unit at the lower left and at the upper left. Power also connects to the unit in the leftmost black. A System Unit is connected to other System Units only via the UNIBUS.

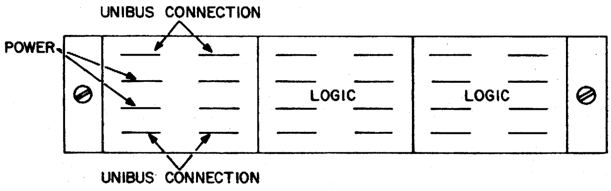


Figure 6.1 System Unit

The remainder of the System Unit contains logic for the processor, memory or an I/O device interface. This logic is composed of single height, double height, or quad height modules which are 8.5" deep.

The use of System Units allows the PDP-11 to be optimally packaged for each individual application. Up to six System Units can be mounted into a single mounting box. For a basic PDP-11/20 system, the processor/console would fill 2 1/2 System Unit spaces and 4096 words of core memory would fill one System Unit space. This leaves 2 1/2 spaces for the user-designated options. This would allow the user to add 8192 words of additional core memory, a Teletype control, and a High-Speed Paper Tape Control, or 4096 words of core memory, and six Teletype interfaces. Larger systems will require a BA11-EC or BA11-ES Extension Mounting Box which contains space for six additional System Units.

The use of System Units also facilitates expansion of systems in the field and service. To add an additional option to a PDP-11 system, the proper System Unit is mounted in the Basic or Extension Mounting Box and the UNIBUS is extended. Servicing of the PDP-11 can be done by swapping modules or by swapping System Units.

When ordering PDP-11 systems it is important that sufficient mounting hardware is ordered to accommodate each system. Particular attention should be given to the of DD11's required and whether a BA11-EC or BA11-ES Extension Mounting Box is needed.

To determine the number of DD11's to order, total the number of spaces required for each item ordered times the quantity ordered. Subtract two from this number and divide by four. Round up to the next whole number if there is a remainder. Order this number of DD11's.

$$\frac{\#\text{ of ``Spaces'' used } -2}{4} = \#\text{ of DD11's needed}$$

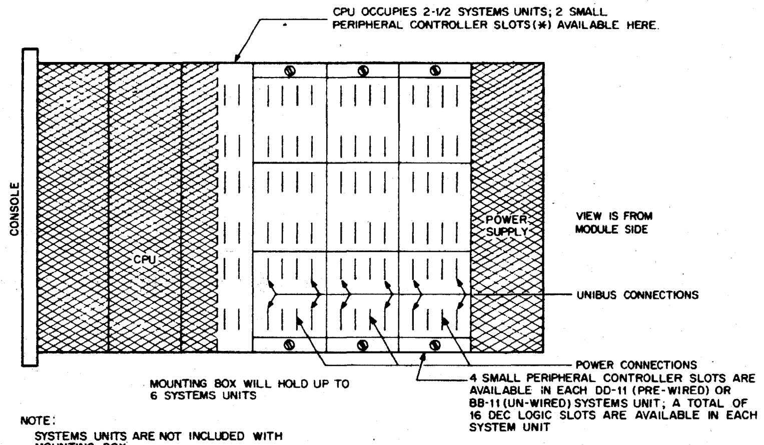
Note: Round up to a whole number.

Six System Units will mount in either the Basic or the Extension Mounting Box. To determine whether to order an Extension Mounting Box, total the products of the number of System Units required for each item ordered times the quantity ordered. Include DD11's and BB11's. Add one and divide the new total by six and round up to the next whole number if there is a remainder. If the result is one, an Extension Mounting Box is not needed. If the result is two, order an Extension Mounting Box (BA11-ES or BA11-EC) and Power Supply (H720A or H720B).

$$\frac{\#\text{ of System Units used}}{6} = \#\text{ of Mounting Boxes Required}$$

Note: Round up to a whole number. If the result is greater than one an Extension Mounting Box is needed.

DD11's are system Units prewired to mount small peripheral controllers such as a Teletype control or a High Speed Paper Tape Reader/Punch control. Each DD11 can hold four controllers and mounts in 1/6 of a Basic or Extension Mounting Box. This is in addition to the two small peripheral controller slots available in the KA-11.


CPU OCCUPIES 2-1/2 SYSTEMS UNITS; 2 SMALL PERIPHERAL CONTROLLER SLOTS (*) AVAILABLE HERE.
CONSOLE
CPU
POWER SUPPLY
VIEW IS FROM MODULE SIDE
UNIBUS CONNECTIONS
POWER CONNECTIONS
MOUNTING BOX WILL HOLD UP TO 6 SYSTEMS UNITS
4 SMALL PERIPHERAL CONTROLLER SLOTS ARE AVAILABLE IN EACH DD-11 (PRE-WIRED) OR BB-11 (UN-WIRED) SYSTEMS UNIT; A TOTAL OF 16 DEC LOGIC SLOTS ARE AVAILABLE IN EACH SYSTEM UNIT

NOTE:

SYSTEMS UNITS ARE NOT INCLUDED WITH MOUNTING BOX.

CPU PLUGS INTO 3 SYSTEMS UNITS (SUPPLIED WITH CPU.

ONE SYSTEM UNIT IS INCLUDED WITH EACH MEMORY ORDERED (EXCEPT M792)

* THESE SMALL PERIPHERAL CONTROLLERS MAY BE:

1. TTY CONTROLLER (KL-11)
2. HIGH-SPEED READER/PUNCH CONTROL
3. LINE-PRINTER CONTROL
4. CARD READER CONTROL
5. 32-WORD DIODE ROM BOOTSTRAP
6. DR-11A GENERAL PURPOSE INTERFACE

Figure 6-2 PDP-11 Box Configuration

## 6.1 PDP-11/20, PDP-11/15 COMPUTERS

The PDP-11 is available as either a tabletop or rack-mounted configuration. The rack-mounted configuration may be installed in a DEC cabinet or mounted in a customer cabinet. The PDP-11 mounts in an EIA standard 19 inch cabinet. The rack-mounted PDP-11 has tilt-slides as standard mounting hardware.

The following mounting units and cabinets are available for PDP-11 systems.

### 6.1.1 PDP-11 Tabletop Box and Power Supply For 11/20, 11/15 Systems (BA11-CC and H720)

This cover and box may be specified with a basic system and includes:

1. H720 Power Supply
2. 15' of power cord with ground wire

   For 115 V standard, parallel blade, U-ground, 15 ampere connectors (NEMA 5-15P)

   For 230 V 3 prong U-ground (NEMA 6-15P)
3. Cooling Fans
4. Filter
5. Programmers Console with 11/20 or Turn-Key Console with 11/15

Approximate Size: 11" high, 20" wide, 25 5/8" deep. Figure 6 shows the layout of this unit.

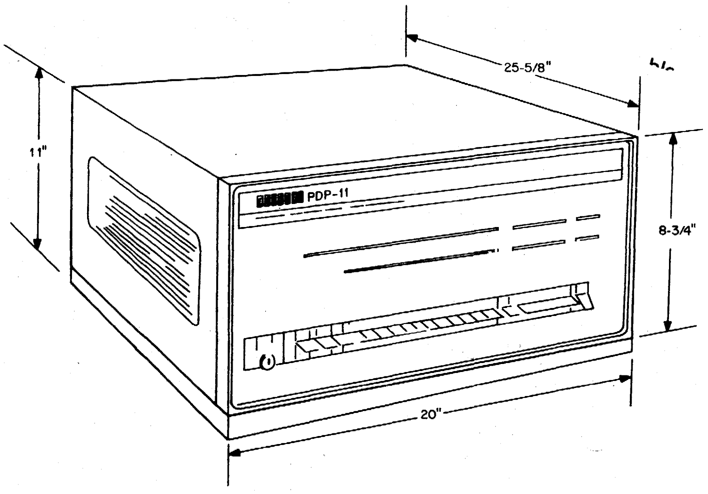


Figure 6.3 Table Top PDP-11 Dimensions

Approximate Weight: 100 lbs. (including CP, console and 4K core)

| | | |
|---|---|---|
| Power: | 120 V + 10%, 47-53 Hz (BA11-CC and H720-E) | 5 amps. single phase |
| | 230 V + 10%, 47-63 Hz (BA11-CC and H720-F) | 2.5 amps. single phase |

**6.1.2 PDP-11 Basic Mounting Box and Power Supply (BA11-CS and H720)**
This basic mounting box may be specified with a basic 11/20 or a 11/15 system and includes:

1. Tilt and Lock Chasis Slides
2. H720 Power Supply
3. 15' of power cord with ground wire

   For 115V standard, parallel-blade, U-ground, 15-ampere connector, (NEMA 5-15P)

   For 230 V 3-prong, U-ground, NEMA No. 6-15P
4. Cooling Fans
5. Filter
6. Programmer's Console with 11/20 or Turn-Key Console with 11/15

Approximate Size: 10 1/2" high, 19" wide, 23" deep. Figures 10-3, 10-4 and 10-5 show the layout of this unit and give slide dimensions.

Approximate Weight: 90 lbs. (including CP, console and 4K core)

| | | |
|---|---|---|
| Power: | 120 V + 10%, 47-63 Hz (BA11-CS and H720-E) | 5 amps. single phase |
| | 230 V + 10%, 47-63 Hz (BA11-CS and H720-F) | 2.5 amps. single phase |

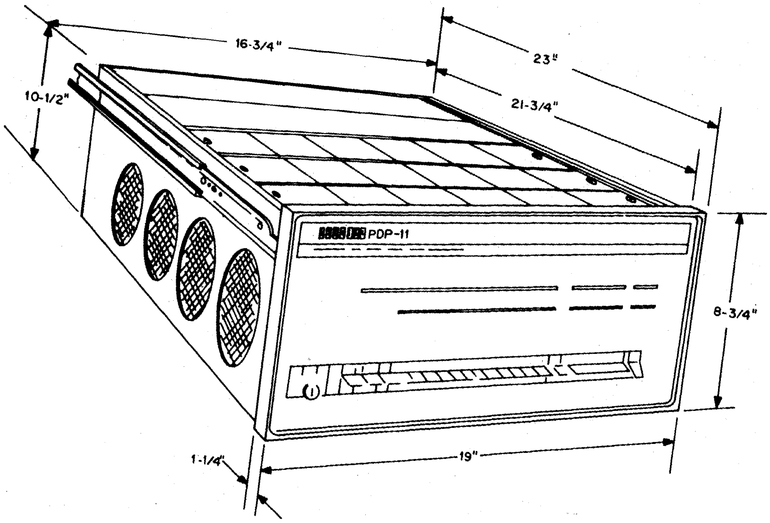


Figure 6.4 Rack-Mountable PDP-11 Dimensions

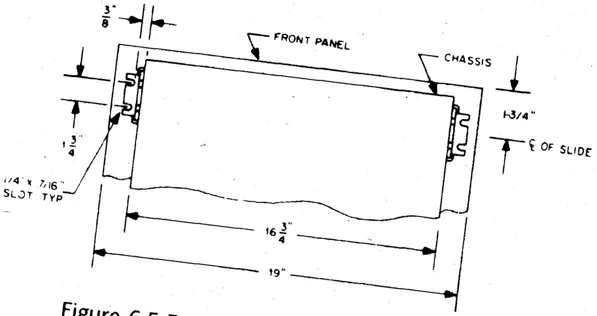


Figure 6-5 Rear View of Mounting Hardware

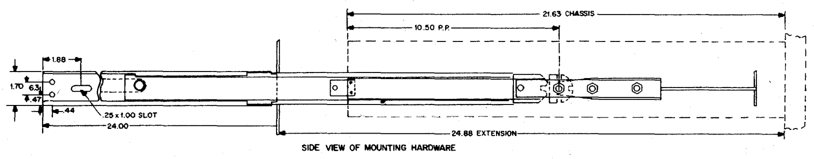


Figure 6-6 Side View of Mounting Hardware

**6.1.3 PDP-11/20 and PDP-11/15 Tabletop Extension Mounting Box (BA11-EC)**
The tabletop Extension Box is supplied, when ordered, for mounting of up to 6 additional System Units which can not be contained in the Basic Mounting Box. This unit is supplied with:

1. 15' of power cord with ground wire

   For 115 V standard, parallel blade, U-ground, 15-ampere connector (NEMA 5-15P)

   For 230 V 3-prong, U-ground, NEMA 6-15P
2. Cooling Fans
3. Filter
4. Front Panel
5. UNIBUS Cable from Basic Mounting Box, 8'6" long

Approximate Size: 11" high, 20" wide, 24" deep

| | | |
|---|---|---|
| Power: | 120 V + 10%, 47-63 Hz (when H720-E is added) | 5 amps. single phase |
| | 230 V + 10%, 47-63 Hz (when H720-F is added) | 2.5 amps. single phase |

**6.1.4 PDP-11/20 Extension Mounting Box (BA11-ES)**
The Extension Box is supplied, when ordered, for mounting of up to 6 additional System Units which can not be contained in the Basic Mounting Box. This unit contains:

1. Tilt and Lock chassis slides
2. 15' of power cord with ground wire

   For 115 V standard, parallel-blade, U-ground, 15-ampere connector (NEMA 5-15P)

   For 230 V 3-prong, U-ground (NEMA 6-15P)
3. Cooling Fans
4. Filter
5. Front Panel
6. Bus Cable from Basic Box, 8'6" long

Approximate Size: 10 1/2" high, 19" wide, 23" deep

| | | |
|---|---|---|
| Power: | 120 V + 10%, 47-63 Hz (when H720-E is added) | 5 amps. single phase |
| | 230 V + 10%, 47-63 Hz (when H720-F is added) | 2.5 amps. single phase |

**6.1.5 PDP-11 Freestanding Base Cabinet (H960-CA)**
This optional cabinet can be used to mount the BA11-CS Basic Mounting Box and a BA11-ES Extension Mounting Box supplied with Tilt and Lock chassis slides in addition to other PDP-11 equipment.

Panel capacity is six 10 1/2" high mounting spaces, each of which is covered with black plastic panels if equipment is not mounted - (5 panels, maximum, supplied).

Items supplied with the cabinet include:

1. H950-A Frame
2. H952-E Coasters
3. H-952-F Levelers
4. H-952-C Fan Assembly (in top of cabinet)
5. H-950-S Filter
6. PDP-11 Logo
7. H-950-B Rear Door
8. 10 1/2" Plastic Bezels, maximum of 5 supplied
9. Two H952-A End Panels

**6.1.6 Cable Requirements**
When an Extension Mounting Box is used, an external cable, the BC11A, is the only signal connection between mounting boxes. This external bus cable may also be used to connect other peripherals to the PDP-11. The maximum combined, internal and external, bus cable length is 50'.

**6.1.7 Environmental Requirements - PDP-11/20, PDP-11/15**

The PDP-11 is designed to operate from +10° to +50° C with a relative humidity of from 20% to 95% (without condensation).

## 6.2 PDP 11R20 RUGGEDIZED COMPUTER

The PDP-11R20 Rugged computer is available in a rack-mountable configuration which may be installed in a DEC cabinet or mounted in a customer cabinet. The PDP11R20 mounts in an EIA standard 19 inch cabinet and has tilt and lock chassis slides as standard mounting hardware.

### 6.2.1 PDP 11R20 Basic Mounting Box and Power Supply

This basic mounting box comes standard with the PDP-11R20 system and includes:

1. Tilt and Lock chassis slides
2. H720 Power Supply
3. 15' of power cord with ground wire

   For 115 V standard, three prong twist lock connector

   For 230 V three prong twist lock connector.
4. Cooling fans
5. Filters
6. Programmers Console

Approximate Size: 10 1/2" high, 19" wide, 25" deep

Approximate Weight: 110 lbs

| | |
|---|---|
| Power Line Frequency: | 47-63 Hz, 380-420 Hz |
| Power Line Voltage: | 100, 115 VAC + 10% 200, 215, 230 VAC + 10% |
| Power Line Current: | 5 amps max @ 115 VAC |
| Power Dissipation: | 500 Watts max |

### 6.2.2 BAR11EC Rugged Extension Mounting Box

The rugged extension mounting box is designed for mounting up to 6 additional system units which cannot be contained in the basic Rugged mounting box. This unit contains:

1. Tilt and Lock chassis slides
2. Cooling fans
3. Filters
4. Blank front panel
5. Rugged internal and 10' external unibus cable to connect to the basic box.

### 6.2.3 Cables

All options ordered with the rugged PDP-11 must have special rugged cables ordered with them. All cables that go into this box do so by means of 1/4 turn mil-type connectors. The convenience outlet is a 3-prong twist lock female plug

### 6.2.4 Environmental Requirements

| | |
|---|---|
| TEMPERATURE | |
| Operating: | 0°C to +55°C |
| Non-operating: | -55°to +85°C |
| HUMIDITY: | 95% RH |
| VIBRATION: | Vibration applied on 3 mutually perpendicular axis. 5-9 Hz, 1.0" double amplitude;9-500 Hz, 2.5G |
| SHOCK: | 3 shocks in each direction on 3 mutually perpendicular axis (18 shocks) |
| Operating: | 5G, 11 msec |
| Non-operating: | 15G, 11 msec |
| ALTITUDE | |
| Operating: | 10,000 feet max. |
| Non-operating: | 50,000 feet max. |
| INCLINATION: | Operates in any attitude |
| RELIABILITY: (at 25 C) | Processor: 22,000 hours MTBF<br>Power Supply: 33,000 hours MTBF<br>Memory: 11,000 hours MTBF<br>Computed from MIL-HDBK-217A, 1 Dec. '65 |

## 6.3 INSTALLATION PROCEDURE
The PDP-11 is crated for shipment to the customer site to prevent damage. Installation is provided by DEC personnel at the customers site.

Computer customers may send personnel to instruction courses on computer operation, programming, and maintenance conducted regularly in Maynard, Massachusetts, Palo Alto, California, and Reading, England.

## 6.4 SYSTEM UNITS AND CABLES
The following items are available for mounting standard and special peripheral device logic into a PDP-11 system.

### 6.4.1 Peripheral Mounting Unit (DD11-A)
The DD11 is a prewired system Unit which allows standard small peripheral interfaces to be mounted in a PDP-11 system. It accepts standard small peripheral interfaces (up to 4) such as the KL11 Teletype Control or the controller portion (PC11-M) of the High Speed Reader/Punch. For mounting, it requires one-sixth (1/6) of a BA11 Mounting Box.

### 6.4.2 Blank System Unit (BB11)
The BB11 consists of three 288-pin connector blocks connected end-to-end. This unit is unwired except for UNIBUS and power connections and allows customer-built interfaces to be integrated easily into a PDP-11 system. For mounting it requires one-sixth (1/6) of a BA11 Mounting Box.

### 6.4.3 UNIBUS Module (M920)

The M920 is a double module which connects the UNIBUS from one System Unit to the next within a Mounting Box. The printed circuit cards are separated by 1"for this purpose. A single M920 will carry all 56 UNIBUS signals and 14 grounds.

### 6.4.4 UNIBUS Cable (BC11A)

The BC11A is a 120-conductor flexprint cable used to connect System Units in different mounting boxes of a peripheral device which is removed from the mounting boxes.

The 120 signals consist of the 56 UNIBUS lines plus 64 grounds. Signals and grounds alternate to minimize cross talk.

| Type | Length |
|---|---|
| BC11A-2 | 2' |
| BC11A-5 | 5' |
| BC11A-8A | 8'6" |
| BC11A-10 | 10' |
| BC11A-15 | 15' |
| BC11A-25 | 25' |

## 6.5 PDP-11 POWER SUPPLY SUBSYSTEM H720

This Power supply is used in the Basic and Extension Mounting boxes and supplies power to all devices mounted in one of these boxes. It is included in basic PDP-11 systems, but must be ordered separately with a BA11ES or BA11EC Extension Mounting Box.

Approximate Size: 16 1/2" wide, 8" high, 6" deep

Approximate Weight: 30 lbs.

| | | | | | | |
|---|---|---|---|---|---|---|
| Power: | IN | 117V | 10% | 47-63 Hz | 6A | H720E |
| | | 230V | 10% | 47-63 Hz | 3A | H720F |
| | | 215V | 10% | 47-63 Hz | 3A | H720F |
| | | 200V | 10% | 47-63 Hz | 3A | H720F |
| | OUT | +5V | 5% | 22A | | (H720E,F) |
| | | -15V | 5% | 22A | | (H720E,F) |

+8 RMS (UNREGULATED) 1.5A (H720E,F)

-22 V (UNREGULATED) 1.0A (H720E,F)

AC LO
DC LO

## 6.6 PDP-11/20 Power Requirements

Power Dissipation: 400 watts

## 6.7 Teletype Requirements

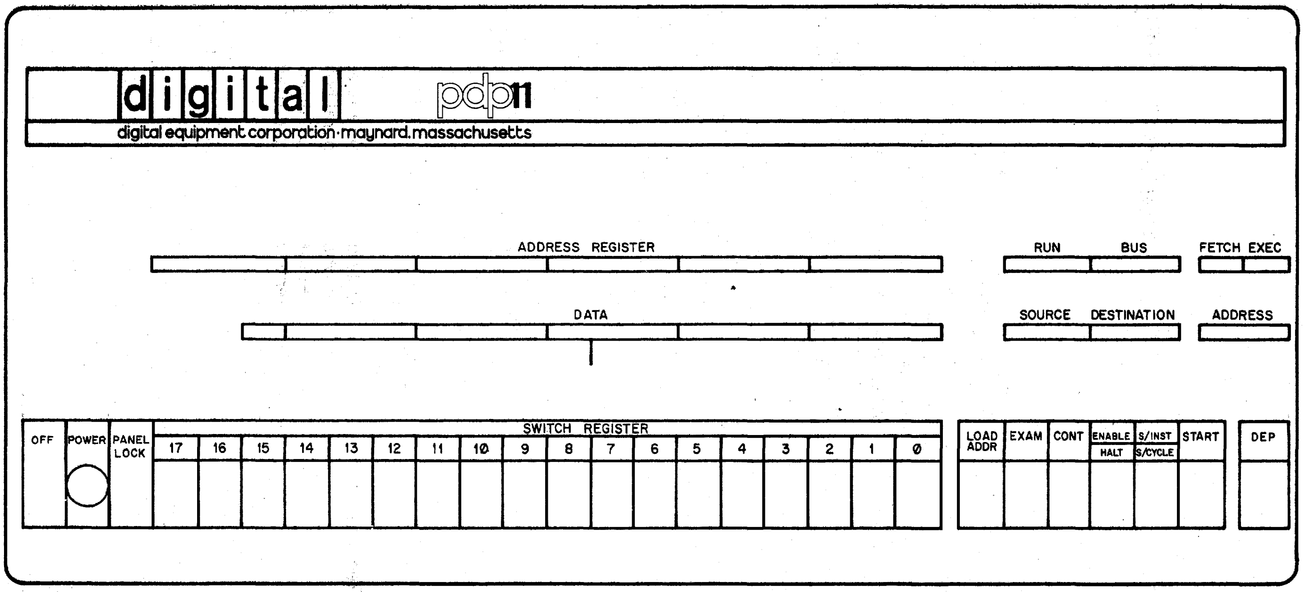
The standard Teletype requires a floor space approximately 22 1/2 inches wide by 18 1/2 inches deep. The Teletype cable length restricts its location to within 8 feet of the side of the computer.

Input Voltage: 115 Vac 10%, 60 Hz 0.45 Hz, 230 Vac 10%, 50 Hz 0.75 Hz

Line Current Drain: 2.0 amperes

Power Dissipation: 150 watts

The Teletype plugs into the rear of the PDP-11 Basic Mounting Box and is turned ON and OFF by the power switch on the front panel of the PDP-11.


digital
pdp11
digital equipment corporation·maynard.massachusetts
ADDRESS REGISTER
RUN
BUS
FETCH
EXEC
DATA
SOURCE
DESTINATION
ADDRESS
SWITCH REGISTER
OFF
POWER
PANEL LOCK
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
Ø
LOAD ADDR
EXAM
CONT
ENABLE
HALT
S/INST
S/CYCLE
START
DEP

PART I
CHAPTER 7

# CONSOLE OPERATION

The PDP-11/20, PDP-11/15, and PDP-11R20 Operators' Consoles provide users with comprehensive information regarding the status of the system, and with function switches to control the system. Each section of the Operator's Console is discussed in this chapter. The PDP-11R20 Console differs slightly in layout due to ruggedized construction constraints, but it is functionally identical to the PDP-11/20 Console. The PDP-11/15 console differs only in that there are 16 lights and switches in the Address Register, instead of 18 as in the PDP-11/20.

**INDICATOR LIGHTS**

| | | |
|---|---|---|
| RUN | On: | Indicates that the processor clock is running, processor has control of bus, and is executing an instruction. |
| | Off: | Indicates that the processor is waiting for an asynchronous peripheral data response, or that the processor has surrendered its control to the console or a peripheral. |
| | Remarks: | Flickers on and off during normal machine operation, except during the following programmed instructions: WAIT (completely on); HALT (completely off). |
| BUS | On: | Indicates that a peripheral device is controlling the bus. |
| | Remarks: | Only on when there is a bus malfunction or where a peripheral holds the bus for excessive periods of time, or in large systems when multiple devices are using the bus for DMA operations.<br><br>When Bus and Run are off, bus control has been transferred to the console. |
| FETCH | Function: | Indicates that the processor is in the FETCH state and is obtaining an instruction. |
| | Remarks: | Only Fetch and Run lights are on during the Fetch state if no non-processor requests are honored. |

| | | |
|---|---|---|
| EXEC | Function: | Indicates that the processor is the Execute state, performing an action specified by the instruction. |
| | Remarks: | Only Exec and Run indicators are on during the Execute state if no non-processor requests are honored. |
| DEST. | Function: | Indicates that the processor is in Destination state and is obtaining destination operand data. |
| | Remarks: | Destination and Run are both on during the Destination state. Address lights may be on in various combinations. Bus is off if no non-processor requests are honored. |
| SOURCE | Function: | Indicates that the processor is in the source state and is obtaining source operand data. |
| | Remarks: | Source and Run lights are both on during the Source State. Address Lights may be on in various combinations. Bus if OFF if no non-processor requests are honored. |
| ADDR. (2 lights) | Function: | Indicates bus cycles used to obtain address data during Source and Destination states. Binary code of lights indicates address cycle (1,2, or 3) machine is in source or destination state. |
| | Remarks: | When either light is on, either Source or Destination is on. Bus if off if no non-processor requests are honored. |

**SWITCH REGISTER**
18 Key-Type Switches*

| | |
|---|---|
| Function: | Used to manually load 16-bit data word or address into processor.<br>UP = ON = 1<br>DOWN = OFF = 0 |
| Remarks: | If the word in the Switch Register represents an address, it can be loaded into an sents an address, it can be loaded into an Address Register by depressing LOAD ADDR key.<br><br>If the word contains data, it can be loaded into to address specified by the ADDRESS REGISTER by lifting the DEP key. The data will appear in the DATA display. |

*16 Switches on KY11C Console (PDP-11/15)

| | |
|---|---|
| Remarks: | The console permits the user to immediately examine data just deposited with out readdressing, to re-deposit if necessary, and to continue without automatic incrementation. These sequences are associated with the functioning of DEP and EXAM Switches. The state of the switches can be read as 1's and 0's under program control by reading address 777570. |
| **CONTROL SWITCHES** | |
| LOAD ADDR. | |
| Function: (Depress to activate) | Transfers contents of switch register to bus address register. |
| Remarks: | The resulting bus address, displayed in the ADDRESS REGISTER, provides an address for EXAM, DEP, and START. |
| EXAM Function: (depress to activate) | Transfers contents of bus address for DATA display. Data address will appear in two ADDRESS REGISTER. |
| Remarks: | If the EXAM switch is depressed on succession, the contents of the next sequential bus address are displayed in DATA. This action is repeated each time EXAM is depressed provided no other Switch is used between these steps. |
| CONT Function: (depress to activate) | Causes processor to continue operation from the point at which it had stopped. If ENABLE/HALT is on ENABLE, returns bus control from console to processor and continues program operation. If ENABLE/HALT is on HALT, causes the processor to perform a single instruction or a single bus cycle and stop. |
| Remarks: | If program stops, this switch provides a restart without program clear. |
| ENABLE/HALT | |
| Function: (2-position switch) | Allows either the program or the console to control processor operation. ENABLE permits system to run normally. HALT stops the processor and passes control to the console. |
| Remarks: | Continuous program control requires the ENABLE mode. |
| | HALT mode is used to interrupt program control, perform single-step operation, or clear the system. HALT is used with the CONT switch to step the machine through |

| | |
|---|---|
| | programs and facilitate intermediate observations. |
| S-INST/S-CYCLE Function: (2 position switch) | Allows processor to step through program operation either one instruction or one bus cycle at a time. S-INST: processor halts after an instruction. S-CYCLE: processor halts after a bus cycle. |
| Remarks: | Enabled by ENABLE/HALT in HALT mode. |
| START Function: (depress to activate) | If ENABLE/HALT is on ENABLE, provides a system clear operation, then begins processor operation. A LOAD ADDR operation establishes the starting address. If ENABLE/HALT is on HALT, provides a system clear (initialize) only. Processor does not start. |
| DEP Function: | Transfers contents of console SWITCH REGISTER to bus address. |
| Remarks: | After use data will appear on DATA display, address in ADDRESS REGISTER. |

**ADDRESS REGISTER**
18-Bits, divided in 3-bit sequence.

| | |
|---|---|
| Function: | Displays the address of data examined or deposited. (16-bit in the PDP-11/15) |
| Remarks: | During a programmed HALT or WAIT instruction, display contains the address of the instruction.<br><br>During direct memory operations, the processor is not involved in data transfer functions, and the address displayed is not of the last bus operation.<br><br>When console switches are used, this display contains the following:<br>LOAD ADDR - Transferred<br>SWITCH REGISTER - data<br>DEP or EXAM - the bus address just deposited into or examined<br>S-INST or S-CYCLE - the last processor address |

**DATA**
16-Bit Display

| | |
|---|---|
| Function: | Displays data from processor data paths. This is not a single register but the sum of two later registers on the data paths (16- |

| | |
|---|---|
| | bit on the PDP-11/15) on both machines, no distinction necessary. |
| | Data is mainly loaded into this register by setting the data value into SWITCH REGISTER and lifting the DEP switch. |
| Remarks: | When console switches are used, this display contains:<br>LOAD ADDR - no indication<br>DEP - the switch register just deposited.<br>EXAM - the data from the address examined.<br>S-INST - no indication when stepping through a program by single instruction.<br>S-CYCLE - last data in the data paths.<br>WAIT - no indication<br>HALT - displays processor register R0 when bus control is transferred to console during a HALT instruction.<br>RESET - displays register - R0 for during of RESET (70 msec). |

**POWER LOCK**
OFF/POWER/PANEL LOCK

3-position switch

| | |
|---|---|
| OFF: | Removes all power from processor 3 position switch |
| POWER: | Applies primary power to processor |
| PANEL LOCK: | Disables all console controls except switch register key switches. |
| Remarks: | OFF: System is not being used<br>POWER: Normal operation; all console controls fully operational |

PART I
CHAPTER 8

# EXTENDED ARITHMETIC ELEMENT

## 8.1 EXTENDED ARITHMETIC ELEMENT KE11-A

The Extended Arithmetic Element (EAE) (KE-11A) is an option which performs multiplication, division, multiple position shifts and normalization significantly faster than software routines. It connects directly to the UNIBUS and is programmed as a peripheral, allowing overlap between CP and EAE operations.

The KE11-A performs the following operations:

**Multiply** Two 16-bit numbers are multipiled to give a 32-bit product.

Examples:

```
000002 * 000005 = 000000-000012 (2 * 5 = 10)₁₀
177775 * 000007 = 177777-177753 (-3 * 7 = -21)₁₀

176000 * 177400 = 000004-000000 (-2¹⁰ * -2⁸ = 2¹⁸)
010000 * 100000 = 174000-000000 (+12¹² *-2¹⁵ = -2²⁷)
```

**Divide** A $32_{10}$-bit dividend is divided by a $16_{10}$-bit divisor to give a $16_{10}$-bit quotient and a $16_{10}$-bit remainder. The sign of the remainder is always the same as the sign of the dividend, unless the remainder is zero(i.e.–8/3 = –2REM–2 not –3 REM 1). The KE11-A indicates overflow if more than $16_{10}$-bits would be needed to express the quotient (i.e. overflow if the quotient is out of the range $(2^{15})-1$ to $(-2^{15})$. Zero divided by zero gives overflow.

Examples:

```
000000-000013 / 000003 = 000003 REM 000002 (11₁₀/3 = 3 REM 2)
177777-177765 / 000003 = 177775 REM 177776 (-11₁₀/3 = -3 REM -2)

000010-000000 /000020 = Overflow 2¹⁹/2⁴ = 2 ¹⁵
000007-177777 / 000020 = 077777 REM 000017
2¹⁹-1/2⁴ = 2¹⁵-1 REM (2⁴-1)
177770-000000 / 000020 = 100000 REM 000000 (-2¹⁹)/2⁴ = -2¹⁵)
000007-177777 / 177760 = 100001 REM 000017
(2¹⁹)-1/-(2⁴) = -((2¹⁵)-1) REM (2⁴-1)
```

**NOTE**
All numbers are octal unless followed by a subscript "10" for decimal. Also, $32_{10}$-bit numbers are shown in octal as two sixteen bit numbers, thus, 000001-000000 is $2^{16}$.

**Normalize** A $32_{10}$-bit number is shifted left until the two most significant bits are different. Zeros fill the empty positions on the right. A count is kept of the number of places the $32_{10}$-bit number is shifted. There are three special cases:

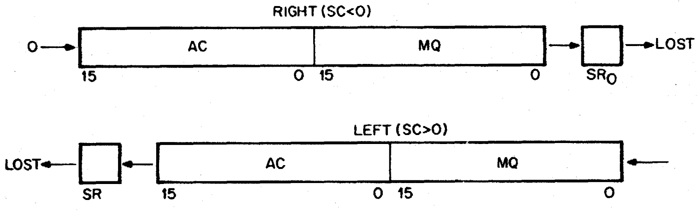
> The number is of the form 111...1100...0000 (BINARY) In this case, the number is shifted until it is 140000-000000.
>
> The number is 177777-177777. In this case the result is 140000-000000, and the count is $30_{10}$.
>
> The number is 000000-000000. In this case the result is 000000-000000, and the count is $31_{10}$.

Examples:

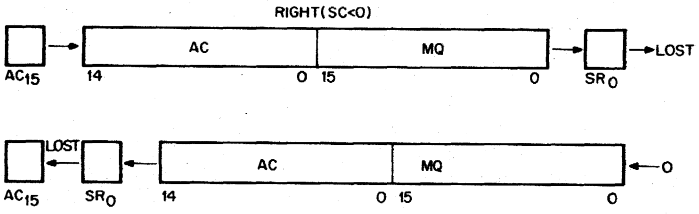
| | | | |
|---|---|---|---|
| 000041-170324 | becomes | 041741-124000 | Count: $9_{10}$ |
| 177777-174321 | becomes | 106420-000000 | Count: $20_{10}$ |
| 177740-000000 | becomes | 140000-000000 | Count: $9_{10}$ |

**Multiple Shifts** A $32_{10}$-bit number is shifted either left or right the number of places specified by a count. The count is a 6-bit 2's complement number. If the count is positive, the number is shifted left; if it is negative, the number is shifted right. This allows for shifts from 31 positions left to 32 positions right. A count of zero causes no change in the number. There are two different shift operations:

Logical Shift: Zeros always fill the vacated positions.



Arithmetic Shift: When shifting left, zeros fill the vacated positions and the most significant bit of the number is not shifted (the sign never changes). When shifting right, the most significant bit is replicated (the sign is extended).



The KE11-A indicates overflow on left shifts if the result is not the correct multiple of the original number. This occurs if the most significant bit changes on a logical shift, or if it would have changed on an arithmetic shift. No overflow is possible on right shifts.

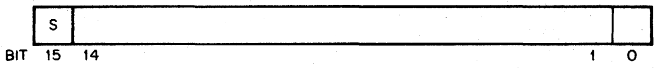
Examples:

| Original Number | Count | Logical Shift | Arithmetic Shift |
|---|---|---|---|
| 000777-177700 | 15 | 177770-000000 | 077770-000000 overflow |
| 177525-052525 | 05 | 165252-125240 | 165252-125240 |
| 000777-177700 | 73 | 000017-177776 | 000017-177776 |
| 177525-052525 | 63 | 000007-175252 | 177777-175252 |

## 8.2 PROGRAMMING

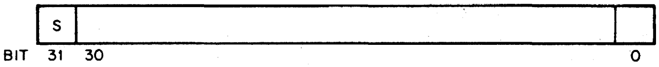
**Number Formats** All numbers in the KE11-A are in signed, 2's complement notation. This means that if the most significant bit of a number is zero, the number is positive and the rest of the number is the magnitude. If the most significant bit is one, it means that the number is negative and the rest of the number is the 2's complement of the magnitude. Zero is represented with all bits zero.

There are two different number formats in the KE11-A. One format uses $16_{10}$ bits:



This gives a range of numbers from +(215)–1 to –(215). The largest positive number is 077777 and the largest negative number is 100000. A plus one would be 000001; minus one would be 177777; and –((215)–1 would be 100001.

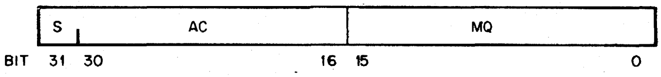
The other format uses $32_{10}$ bits:



This gives a range of numbers from (2311)–1 to –(231). The largest positive number is 077777-177777 and the largest negative number is 100000-000000. 4 The 2's complement of a number is formed by changing all 1's to 0's, all 0's to 1's, and then adding 1.

| REGISTERS | ADDRESSES |
|---|---|
| Accumulator (AC) | 777302 |
| Multiplier Quotient (MQ) | 777304 |
| Step Counter (SC) | 777310 |
| Status Register (SR) | 777311 |

**Accumulator (AC) and Multiplier Quotient (MQ)**
These are the two data registers in the KE-11A. Each is $16_{10}$-bits. They are sometimes used together to hold one $32_{10}$-bit number, in which case the MQ is the low order part of the word (bits 00-15) and the AC is the high order part (bits 16-31).



Whenever a part of this double-word register is loaded, the sign is always extended into the higher bits that were not loaded. For example:

```
MOVB    A,MQ      ;MQ BITS 8-15 AND AC BITS 0-15 EXTENDED
MOV     A,MQ      ;AC BITS 0-15 EXTENDED
MOVB    A,MQ+1    ;AC BITS 0-15 EXTENDED
MOVB    A,AC      ;AC BITS 8-15 EXTENDED
MOV     A,AC      ;NO EXTENSION
MOVB    A,AC+1    ;NO EXTENSION
```

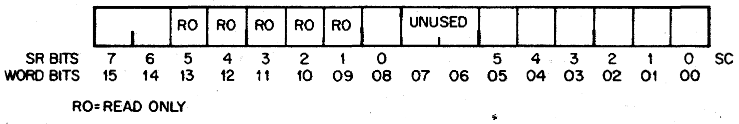
Thus, when loading the AC and the MQ with word operations, first the MQ and then the AC must be loaded. When using byte operations, first the low byte of the MQ, the high byte of the MQ, the low byte of the AC, and then the high byte of the AC must be loaded.

NOTE: This applies to all instructions that effect the destination not only MOVe.

On multiplication, the MQ initially contains the multiplier and the AC is ignored. After the multiply, the AC-MQ contains the $32_{10}$-bit product. On division, the AC-MQ initially contains the $32_{10}$-bit dividend, and after the divide, the MQ contains the quotient and the AC contains the remainder. On normalize and shifts, the AC-MQ contains the $32_{10}$-bit number which is shifted.

**Step Counter (SC)**
The SC controls the number of steps done in all operations which the KE11-A performs. It gets loaded automatically on multiply, divide, normalize and shifting. The register is six bits long, and is at address 777310.

**Status Register (SR)**
The SR contains bits which give information about the last operation performed and the status of the AC and MQ. It is 8 bits long and it is at address 777311 (the high byte of the AC address).



| BIT | NAME | FUNCTION |
|---|---|---|
| 0 | Carry | On shifts this bit contains the last bit shifted out of the AC-MQ. |
| 1 | AC = MQ | On multiply, divide, and normalize this 15 bit is cleared. When set, this bit means that every bit in the AC is the same as MQ bit 15, and therefore the number in the AC-MQ has only single word precision. |
| 2 | AC = MQ = 0 | When set, indicates that both the MQ and AC are all zero. |
| 3 | MQ = 0 | When set, indicates that the MQ is zero. |
| 4 | AC = 0 | When set, indicates that the AC is zero. |
| 5 | AC = 177777 | When set, indicates that the AC contains all ones. |
| 6 | NEG | On shifts, normalize, and multiply this bit is set if the AC sign bit is set. On divide, if there is no overflow, this bit is set if MQ sign bit is set. If there was overflow, this bit is set if the original dividend was negative. |
| 7 | --- | This bit, in conjunction with Bit 6, is used to indicate overflow conditions. It is coded with Bit 6 as follows: |

| Bit 7 | Bit 6 | |
|---|---|---|
| 0 | 0 | = Positive and no overflow |
| 0 | 1 | = Negative and overflow |
| 1 | 0 | = Positive and overflow |
| 1 | 1 | = Negative and no overflow |

The reason for coding bits 6 and 7 in this manner is so the processor condition code bits "N" and "V" can be set by a "ROLB SR" (rotate left byte) instruction. When the processor does a ROLB instruction, the old bit 6 becomes the new bit 7 and goes into condition code bit "N", and the old bit 6 exclusive-or'ed with the old bit 7 goes into condition code bit "V". Therefore, by doing a "ROLB SR" after a KE11-A operation, the "N" and "V" bits in the processor will get set, and some of the conditional branches can be used. It should be noted that the other two bits in the processor condition codes, "Z" and "C', will not be set correctly (although they will be changed) and therefore not all of the conditional branches will work.

Since it is not desirable to actually rotate the status register with the "ROLB SR", when the processor writes back the rotated SR into the KE11-A, nothing will actually change. This is done by inhibiting the SR from being written when addressed as a byte. Therefore, no instruction that attempts to write the SR as a byte will have any effect on the SR, although the KE11-A will respond normally. For example, "CLRB', "MOVB", etc. will not change the SR.

However, to allow for reentrant programming of the KE11-A, it is necessary to be able to save the SR and restore it. Therefore, when the word which contains the SR and SC is written (777310), both the SR and SC are loaded. The SC, just like the SR, however, cannot be loaded by addressing it as a byte. When reloading the registers as a word, bits 0 through 5 of the SC and bits 0, 6, and 7 of the SR are the only ones that actually change. Bits 1 to 5 of the SR always indicate the present state of the AC and MQ. Examples of reading and writing the SR and SC:

```
                              ;ASSUME THE SC = 70 AND THE SR = 140
                              ;THE COMBINED WORD IS THEN 060070
MOVB   SC,R0                  ;R0 WOULD BE 000070
MOVB   SR,R0                  ;R0 WOULD BE 000140
ROLB   SR                     ;SR WOULD REMAIN 140, "N" AND "V"
                              BITS WOULD SET
MOVB   #-1,SC                 ;SC WOULD REMAIN 70
MOVB   #-1,SR                 ;SR WOULD REMAIN 140
MOV    #-1,SC                 ;SC WOULD BE 77, SR WOULD BE 301.
                              ;WORD WOULD BE 140477
```

## 8.3 INSTRUCTIONS

Operations in the KE11-A are started by storing a number at an address. There is one address for each of the five operations that the KE11-A performs. The number must be stored as a word or as the low byte, in which case the sign is automatically extended to the high byte. Storing the number as the high byte has no effect on the KE11-A. Once an operation is initiated in the KE11-A, it will not respond to any instructions until it is finished with that operation. Thus, whenever the KE11-A is examined for a result, it will always be the correct, final answer, and never be some intermediate number. The maximum amount of time the KE11-A takes after an operation is started is 4.25 microseconds, and therefore, the most a processor can wait for a result is about 2 microseconds, due to the overlap in operation and beginning the fetch for the result.

**Multiply** The multiply operation is initiated by writing the $16_{10}$-bit multiplicand at the multiply address. This number is then multiplied by the MQ, and a $32_{10}$-bit product is left in the AC-MQ. Reading the multiply address always returns 000000.

| | |
|---|---|
| Address: | 777306 |
| Execution Time: | 4 μs |
| SR Bits: | 0 cleared |
| | 1, 2, 3, 4, 5 set conditionally |
| | 6 sign of the produce (AC) |
| | 7 no overflow possible |

**Divide** The divide operation is initiated by writing the $16_{10}$-bit divisor at the divide address. This number is then divided into the AC-MQ, and a $16_{10}$-bit quotient is left in the MQ and a $16_{10}$-bit remainder is left in the AC. Reading the divide address always returns 000000.

| | |
|---|---|
| Address: | 777300 |
| Execution Time: | 4.25 μs |
| SR Bits: | 0 cleared |
| | 1, 2, 3, 4, 5 set conditionally |
| | 6 if no overflow, sign of the quotient (MQ) if overflow, sign of the dividend (original AC sign) |
| | 7 Overflow possible |

**Normalize** The normalize operation is initiated by writing something at the normalize address. The number written there is ignored. The operation normalizes the number in the AC-MQ. The count of the number of left shifts can be read at the normalize address, where it will be in the lower six bits. (The SR will not be in the high byte). Since the count is always a positive number, reading the normalized address as a word will get a "sign extended" value, and that number can be directly added or subtracted from an exponent.

| | |
|---|---|
| Address: | 777312 |
| Execution Time: | 0-4 μs |
| SR Bits: | 0 cleared |
| | 1 set conditionally |
| | 2 unchanged |
| | 3, 4 set conditionally |
| | 5 cleared |
| | 6 sign of the AC |
| | 7 no overflow possible |

**Logical Shift** The logical shift operation is initiated by writing a six bit shift count at the logical shift address. The number in the AC-MQ is then shifted right or left the number of places determined by the count. Reading the logical shift address always returns 000000.

| | |
|---|---|
| Address: | 777314 |
| Execution Time: | 0-4 μs |
| SR Bits: | 0 Right shift: last bit shifted out of MQ(00) Left shift: last bit shifted out of AC(15) |
| | 1, 2, 3, 4, 5 set conditionally |
| | 6 sign of the AC |
| | 7 Right shift: no overflow possible Left shift: overflow is AC(15) changed at any point |

**Arithmetic Shift** The arithmetic shift operation is initiated by writing a six bit shift count and the arithmetic shift address. The number in the AC-MQ is then shifted right or left the number of places determined by the count. Reading the arithmetic shift address always returns 000000.

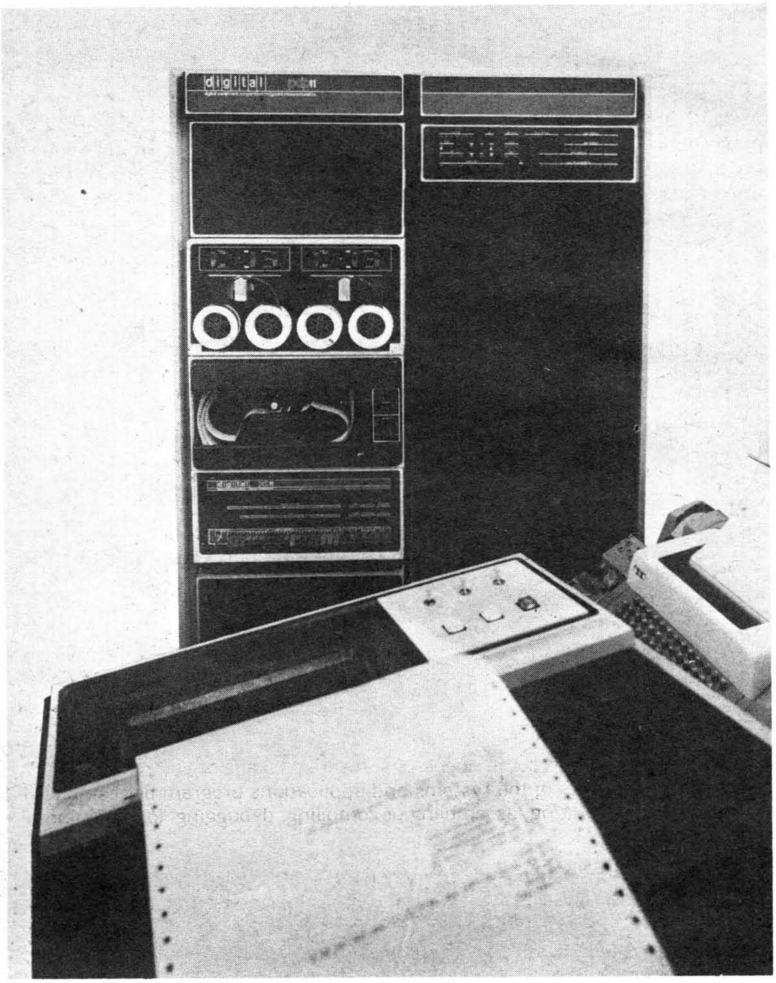
| | |
|---|---|
| Address: | 777316 |
| Execution Time: | 0-4 μs |
| SR Bits: | 0 Right shift: Last bit shifted out of MQ(0) Left shift: Last bit shifted out of AC(14) |
| | 1, 2, 3, 4, 5 set conditionally |
| | 6 sign of the AC |
| | 7 Right shift: no overflow possible Left shift: overflow if AC(15) would have changed at any point |

## 8.4 PROGRAMMING EXAMPLES

```
                                     ;THE AUTO-INCREMENT AND AUTO-
                                     DECREMENT MODES OF ADDRESSING
                                     CAN BE USED TO TAKE ADVANTAGE OF
                                     THE ORDERING OF THE KE11-A AD-
                                     DRESSES

DIV=777300
AC=777302
MQ=777304
MUL=777306
SC=777310
SR=777311
NOR=777312
LSH=777314
ASH=777316

      ;
      MOV #MQ,R0
;SET UP R0 TO ADDRESS   OF MQ. R0 ASSUMED TO HAVE THIS ADDRESS FOR
ALL OF THESE EXAMPLES
MULTIPLY EXAMPLE
MULT:  MOV A,(0)+                    ;PUT "A" INTO MQ

       MOV B,(0)                     ;MULTIPLY BY "B"

       MOV -(0),C                    ;PUT LOW ORDER PRODUCT IN C

       MOV -(0),D                    ;PUT HIGH ORDER PRODUCT IN D

       TST (0)+                      ;BUMP R0 BACK TO THE MQ

                                     ;NOTE THAT IF THE PRODUCT IS KNOWN
                                     TO BE LESS THAN 16 BITS, THE LAST
                                     TWO LINES ABOVE CAN BE ELIMINATED.

DIVIDE EXAMPLE

DIVD:  MOV A,(0)                     ;LOAD LOW ORDER DIVIDEND IN MQ

       MOV B,-(0)                    ;LOAD HIGH ORDER DIVIDEND IN AC

       MOV C,-(0)                    ;DIVIDE BY "C"

       TST (0)+                      ;BUMP R0 BACK

       MOV (0)+,D                    ;PUT REMAINDER IN "D"

       MOV (0),E                     ;PUT QUOTIENT IN "E"

NORMALIZE EXAMPLE, (ASSUME AC-MQ ALREADY LOADED)

       INC @#NOR

       SUB @#NOR,R1                  ;SUBTRACT COUNT FROM R1

SHIFT EXAMPLES
       MOV #3,@#LSH                  ;LOGICAL SHIFT LEFT BY 3

       MOV #-5,@#ASH                 ;ARITHMETIC SHIFT RIGHT BY 5
```

# part 2

## SOFTWARE


digital
pdp11

# PART II INTRODUCTION

# SOFTWARE

A comprehensive collection of proven software is available for the PDP-11. The programmer can choose from two major software systems (a number of special-purpose systems are available), depending on his particular application and hardware configuration (amount of core, external memory, and peripherals). The major software systems are:

1. **Paper Tape System**
   - BASIC Interpreter
   - PAL-11 Assembler
   - ED-11 Text Editor
   - ODT-11 and ODT-11X Debugging Programs
   - Bootstrap and Absolute Loaders
   - Binary and Octal Core Dump Programs
   - IOX, Input/Output Executive
   - Floating-Point Package
2. **Disk Operating System**
   - DOS Monitor
   - FORTRAN IV Compiler
   - PAL-11R Assembler
   - Edit-11 Text Editor
   - ODT-11R Debugging Program
   - PIP, File Utility Package
   - Link-11 Linker
   - Libr-11 Librarian

Each system contains a comprehensive software package of commonly used system programs, providing the systems and applications programmer complete facilities for writing, editing, assembling or compiling, debugging, loading, and running his own programs.

The software system to be used depends greatly on the hardware configuration of the PDP-11. The Paper Tape System software is capable of running on all PDP-11 configurations, with I/O to the user's terminal, paper tape reader and punch, and line printer. It requires only 4,096 words of core memory and a teletype (an 8K and larger version of PAL-11 assembler is also available). The Disk Operating System software requires at least 8K of core and a disk and/or DECtape, and can use virtually any peripheral.

In the Paper Tape System, input and output of programs and data are performed manually via a paper tape reader and punch; printed output can be directed to the user's terminal or line printer; the user communicates with the system programs from the terminal keyboard.

In the Disk Operating System, input and output of programs and data can be on virtually an I/O device; the user communicates with the DOS Monitor and system programs from the terminal keyboard, thus eliminating the need to manipulate paper tapes.

The descriptions in the following chapters highlight some of the benefits and features of PDP-11 software. The PDP-11 user needing complete information should refer to the various PDP-11 software manuals.

PART II
CHAPTER 1

# PAPER TAPE SOFTWARE

## 1.1 PAL-11 ASSEMBLER

PAL-11A provides the programmer a means of writing programs with meaningful symbols rather than with numerical code of usually no mnemonic value. These symbols are then assembled into absolute binary code capable of being executed by the PDP-11. The binary program is normally produced after two passes through the Assembler, although a third pass is available if desired, for either producing a listing or punching a binary tape.

A source program in the PAL-11A language is composed of a sequence of statements where each statement is on a single line as follows:

```
ABCD:      MOV X,Y          ; MOVE THE CONTENTS OF X TO LOCATION Y
```

PAL-11S (Program Assembly Language for the PDP-11, Relocatable Version) like PAL-11A, provides the PDP-11 programmer a means of writing programs with meaningful symbols rather than with numerical code of usually no mnemonic value. However, with this relocatable version, symbols are assembled into object modules which are then processed by the LINK-11S Linker. LINK-11S produces a load module that is loaded for execution. Object Modules may contain absolute and/or relocatable code; and separately assembled object modules may be linked with the aid of global symbols. The object module is produced after two passes through the Assembler. A complete octal/symbolic listing of the assembled program may also be obtained.

Some notable features of PAL-11S are:

Selective assembly pass functions

Error listing on command output device

Alphabetized, formatted symbol table listing

Relocatable object modules

Global symbols for linking between object modules

### 1.1.1 Representing Code

Binary code can be represented in a variety of ways. At one level higher than binary, the octal number system is the primary way of specifying numerical data. Decimal numbers can be specified by following a number with a decimal point. Proceeding to a level higher, symbols can be used to represent octal or decimal values by directly assigning a value to a symbol. Similarly ASCII symbols, the location counter symbol (specifying the current address), or arithmetic/logical expressions can be used to represent numerical code.

### 1.1.2 Operating Procedures

The Assembler enables the user to assemble ASCII tapes containing PAL-11A statements into an absolute binary tape. To do this two or three passes are necessary. On the first pass the Assembler creates a table of user-defined symbols and their associated values, and lists undefined symbols on the teleprinter. On the second pass the Assembler assembles the program and punches out an absolute binary tape and/or outputs an assembly listing. During the third pass (optional) the Assembler punches an absolute binary tape or outputs an assembly listing. The symbol table (and/or a list of errors) may be output on any of these passes. The input and output devices as well as various options are specified during the initial dialog.

## 1.2 EDITING THE SOURCE PROGRAM, ED-11

The PDP-11 Text Editor program (ED-11) enables the user to display his source program (or any text) on the teleprinter, make corrections or additions to it, and punch all or any portion of the program on paper tape.

This is accomplished by the typing of simple one-character commands on the keyboard.

Editor Commands can be grouped according to function:

input/output

searching for strings of characters

positioning the current character location printer

inserting, deleting, and exchanging text portions

All input/output functions are handled by IOX, the PDP-11 Input/Output Executive (See 1.6).

## 1.3 LOADING AND DUMPING CORE MEMORY

### 1.3.1 The Bootstrap Loader

The Bootstrap Loader is a program that instructs the computer to accept and store in core, data that is punched on paper tape in bootstrap format. The Bootstrap Loader is used to load very short paper tape programs of 162 16-bit words or less -- primarily the Absolute Loader and Memory Dump Programs. Either the low-speed reader or high-speed reader can be specified. Programs longer than 162 16-bit words must be assembled into absolute binary format with the PAL-11A ASSEMBLER and loaded into core with the Absolute Loader. The Bootstrap Loader is usually loaded into the highest core memory bank using the console switches and is not destroyed by DEC programs. A 32-word diode ROM hardware bootstrap is available.

### 1.3.2 The Absolute Loader

The Absolute Loader is a system program that loads into any core memory bank, data punched on paper tape in absolute binary format. It is used primarily to load the paper tape system software (excluding certain sub-programs) and the user's object programs assembled with PAL-11A.

The loader programs are loaded into the uppermost area of available core so they will be available for use with system and user programs. User programs should not use the locations used by the loaders without restoring their contents.

Major features of the Absolute Loader include:

Testing of the checksum on the input tape to assure complete, accurate loads.

Starting the loaded program upon completion of loading without additional user action, as specified by the .END statement in the program just loaded.

Specifying the load address of position-independent programs at load time rather than at assembly time, by using the desired Loader switch register option.

**1.3.3 Loading Absolute Tapes**
Any paper tape punched in absolute binary format is referred to as an absolute tape, and is loaded into core using the Absolute Loader.

**1.3.4 Core Memory Dumps**
A core memory dump program is a system program which enables the user to dump (print or punch) the contents of all or any specified portion of core memory onto a device, as indicated below.

There are two dump programs available in the Paper Tape Software System:

a. DUMPTT, which dumps the octal representation of the contents of specified portions of core onto the teleprinter, low-speed punch, high-speed punch, or line printer.

b. DUMPAB, which dumps the absolute binary code of the contents of specified portions of core onto the low-speed punch or high-speed punch.

Both dump programs are supplied on punched paper tape in bootstrap and absolute binary formats. The bootstrap tapes are loaded over the Absolute Loader. The absolute binary tapes are position-independent and may be loaded and run anywhere in core. Operation of these programs is controlled by the user at the PDP-11 console.

## 1.4 FLOATING-POINT AND MATH PACKAGE, FPP-11
The Floating-Point and Math Package for the PDP-11 (FPP-11) is a comprehensive set of subroutines that enables the user to perform a variety of arithmetic operations. FPP-11 provides for:

floating-point operations -- add, subtract, multiply, divide;

calculation of transcendental functions -- sine, cosine, arc tangent, logarithm, square root, exponential;

operations to negate, normalize, move, and compare floating-point numbers;

fixed-point operations of single- and double-precision multiply and divide;

conversion to and from ASCII strings.

Floating-point operations automatically align the binary points of operands, retaining maximum precision by discarding leading zeros. In addition to increasing precision, floating-point operations relieve the user of having to scale numbers (a problem common in fixed-point operations).

The code of the Floating-Point Package is position independent; that is, it may be stored and executed in any contiguous block of core memory without reassembly. The code is also reentrant; that is, any subroutine may be interrupted and reentered from the interrupt handler. This eliminates the necessity for multiple copies -- one for the main program and one for interrupts.

FPP-11 has considerable flexibility. It can handle numbers that are octal or decimal, fractional or integer, signed or unsigned. A number may be represented as one, two, or three binary words, or as a string of ASCII characters. Numbers may be converted from one representation to another e.g., numerical to ASCII.

FPP-11's flexibility extends to the ways of calling and of specifying operands. The subroutines may be called with the addresses of the operands specified directly or indirectly.

The indirect method using the EMT instruction employs a trap handler to perform housekeeping functions. Three calling modes for specifying source and destination addresses are available when using EMT:

1. full addressing mode using the full power of the PDP-11 address modes.
2. fast addressing mode using two general registers as pointers
3. Polish mode that pops the operands off a last-in-first-out stack, leaving the result on the top.

The direct method uses the JSR instruction, thereby requiring that housekeeping be performed by the calling program.

The complete package consists of eleven partially-interdependent modules. The symbolic tapes of the modules may be rearranged and some may be deleted before assembly to tailor FPP-11 to the main program's needs. It is also possible to delete modules without reassembly.

Four formats are available for numerical representation of data:

1. Single-Word Integer
2. Double-Word Integer
3. Floating-Point Normalized (3-word)
4. Floating-Point Unnormalized (3-word)

Following is a list of the FPP-11 subroutines:

| Subroutine name | Meaning |
|---|---|
| ADDF | ADD Floating |
| SUBF | SUBtract Floating |
| NEGF | NEGate Floating |
| MULF | MULtiply Floating |
| DIVF | DIVide Floating |
| NORM | NORMalize |
| MOVF | MOVe Floating |
| CMPF | CoMPare Floating |
| FIX | convert float to FIXed point |
| FIXD | convert float to FIXed point Double-word |

| | |
|---|---|
| FLT | convert fixed point to FLoaTing |
| FLTD | convert Double-word to FLoaTing |
| ITOA | convert Integer TO ASCII |
| JTOA | convert double word (J) TO ASCII |
| FTOA | convert Floating point TO ASCII |
| ETOA | convert Exponential form of floating point TO ASCII |
| OTOA | convert Octal TO ASCII |
| ATOI | convert ASCII TO Integer |
| ATOF | convert ASCII TO Floating point |
| ATOO | convert ASCII TO Octal |
| COS | COSine (argument in radians) |
| SIN | SINe (argument in radians) |
| ATAN | Arc TANgent |
| LOG | LOGarithm to the base e |
| EXP | EXPonential function |
| SQRT | SQuare RooT |
| MUL | MULtiply single-word integer by single word integer |
| DIV | DIVide double-word integer by single-word integer |

## 1.5 DEBUGGING OBJECT PROGRAMS ON-LINE, ODT-11

ODT-11 (On-line Debugging Technique for the PDP-11) is a system program that aids in debugging assembled object programs. From the keyboard the user is able to interact with ODT and the object program to accomplish the following:

print the contents of any location for examination or alteration,

run all or any portion of his object program using the break-point feature,

search the object program for specific bit patterns,

search the object program for words which reference a specific word,

calculate offsets for relative addresses.

A breakpoint feature facilitates monitoring the progress of program execution. A breakpoint may be set at any instruction that is not referenced by the program for data. When a breakpoint is set, ODT replaces the contents of the breakpoint location with a trap instruction so that when the program is executed and the breakpoint is encountered, program execution is suspended, the original contents of the breakpoint location are restored, and ODT regains control. ODT types a message to the user of the form Bn (Bm;n for ODT-11x) where n is the breakpoint address (and m is the breakpoint number). The breakpoints are automatically restored when execution is resumed.

## 1.6 INPUT/OUTPUT EXECUTIVE, IOX

IOX, the PDP-11 Input/Output executive, frees the user from the details of dealing directly with the I/O devices. IOX provides asynchronous I/O service for the following non-file-oriented external devices:

keyboard, teleprinter, and low-speed paper tape reader and punch

high-speed paper tape reader and punch

For line printer handling, an addition to all IOX facilities, IOXLPT is available.

Simple I/O requests can be made, specifying devices and data forms for interrupt-controlled data transfers, which can be occurring concurrently with the execution of a running user program. Multiple I/O devices may be running single or double bufferred I/O processing simultaneously.

Real-time capability is provided by allowing user programs to be executed at device priority levels upon completion of a device action or data transfer.

Communication with IOX is accomplished by IOT (Input/Output Trap) instructions in the user's program. Each IOT is followed by two or three words consisting of one of the IOX commands and its operands. The IOX commands can be divided into two categories:

> those concerned with establishing necessary conditions for performing input and output (mainly initializations), and
>
> those concerned directly with the transfers of data.

When transfer of data is occurring, IOX is operating at the priority level of the device. The calling program runs at its priority level, either concurrent with the data transfer, or sequentially.

### 1.6.1 The Device Assignment Table

Use of the Device Assignment Table (DAT) serves to make the user's programs device-independent by allowing him to reference a slot to which a device has been assigned, rather than a specific device itself. Thus, changing the input or output device becomes a simple matter of reassigning a different device to the slot indicated in the program.

## 1.7 PDP-11 BASIC PROGRAMMING LANGUAGE

PDP-11 BASIC (Beginners All-purpose Symbolic Instruction Code) is an easy-to-learn, conversational, programming language for scientific, business and educational applications. PDP-11 BASIC is directly derived from Dartmouth BASIC with a few limitations and many added features which provide more power and flexibility than is available with standard Dartmouth BASIC. Notable features include:

> Use of BASIC statements in immediate mode (no line number).
>
> Ability to use any BASIC command (RUN, LIST, etc.) in deferred mode (with a line number).
>
> Recursive subroutine calls.
>
> Multiple statements on a single line.
>
> Array names of a letter followed by a number.
>
> User programs can be halted (with CTRL/P) without clearing variables. PRINT can then be used to examine values.
>
> Ability to call assembly language functions.

Basic can run in the minimal 4K PDP-11 configuration. Any additional 4K memory increments are available for user storage unless restricted at load time (see Absolute Loader). A 12K configuration would normally provide 8K plus about 450 words of user storage, and an additional 1000 words are available if BASIC's arithmetic functions are deleted at load time.

PART II
CHAPTER 2

# DISK OPERATING SYSTEM

## 2.1 DISK OPERATING SYSTEM

The PDP-11 Disk Operating System (DOS) represents a significant advance in software development for small computers, providing capabilities which were formally available only on larger machines such as the PDP-10.

The DOS is a program development system for a PDP-11 with a minimum of 8K of core, one or more disks and DECtapes or high-speed paper tape. The DOS Monitor supports the PDP-11 user throughout the development and execution of his program by:

providing convenient, complete access to system programs such as the assembler, compiler, debugger, editor, file utility package, etc.

performing input/output transfers

handling secondary storage management

The PDP-11 DOS is a keyboard-oriented system containing a powerful Monitor and a comprehensive package of system programs. The DOS is modular and open-ended, permitting users to incorporate the programs required for a particular application and to have full access to disk and DECtape for storage and retrieval of system and user programs.

By typing appropriate commands to the DOS Monitor and system programs, the user can generate, edit, assemble or compile, debug, load, save, call, and run programs with ease.

System programs can be called into core from disk or DECtape with Monitor commands issued from the keyboard. This feature eliminates the need to manipulate numerous paper tapes, and provides the user with an efficient and convenient programming tool.

Keyboard commands enable the operator to load and run programs, dump data from core, start or restart programs at specific addresses, modify the contents of memory registers, redirect I/O with logical assignments, and retrieve system information such as time of day, date, and system status.

The user communicates with the Monitor in two ways: through keyboard instructions called commands, and through programmed instructions called requests.

Programmed requests are assembled into the user's program. Some programmed requests are used to access input/output transfer facilities, to specify where the data is, where it is going, and what format it is in. In these cases, the Monitor will take care of bringing device drivers (I/O routines) in from the disk, performing the data transfer, and notifying the user of the status of the transfer. Other requests access Monitor facilities to obtain such information as time of day, date, and system status, and to specify special functions for devices.

### 2.1.2 Monitor Core Organization
Core memory is divided into:

a user area where user programs and buffers are located;

the stack where parameters are stored temporarily during the transfer of control between routines;

The free core or buffer area which is divided into 16-word blocks assigned by the Monitor for temporary tables, for device drivers called in from disk, and for data buffering between devices and user programs;

the resident Monitor itself which includes all permanently resident routines and tables;

the interrupt vectors.

### 2.1.3 Hardware Configurations
The following DOS configurations are supported by DEC:

**Configuration I**
The reliability and speed of a large fixed-head disk are combined with DECtape an inexpensive means of storing large amounts of file-structured data, both on-line and off-line.

PDP-11/20; extra 4K core (8K total); with cabinet and Teletype

RF11/RS11 256K-word, DEC Disk and Control

TC11/TU56 Dual DECtape Transport and Control

BM792-YB ROM Bootstrap Loader

**Configuration II**
This configuration is a lower cost alternate to configuration I. It is intended for applications not requiring a lot of removable storage.

PDP-11/20; extra 4K core (8K total); with cabinet and Teletype

RF11/RS11 256K-word DEC Disk and Control

PC11 High-Speed Paper Tape Reader and Punch

BM792-YB ROM Bootstrap Loader

DD11-A Peripheral Mounting Panel for BM792-YB

**Configuration III**
This configuration is based on a small, fast 64K fixed-head disk used for systems residency. The DECtape provides the media for on-line file, data or program storage. Off-line storage is also provided by the removable DECtapes.

PDP-11/20; extra 4K core (8K total); with cabinet and Teletype

RC11/RS64 64K-word Disk and Control

TC11/TU56 Dual DECtape Transport and Control

BM792-YB ROM Bootstrap Loader

**Configuration IV**
This system combines the flexibility of a disk system with the convenience of a removable disk cartridge pack. It is particularly well suited for applications where several groups use and share the same system. Each group can easily maintain their files independently of the others.

PDP-11/20; extra 8K core (12K total) with cabinet and Teletype

RK11/RK03 1.2 million word DECpack Disk and Control and cabinet

TC11/TU56 Dual DECtape Transport and Control

BM792-YB ROM Bootstrap Loader

**Configuration V (For very high speed operation and large file storage)**
This system has all the advantages of configuration IV plus: the additional fixed-head disk increases system throughput; the DECtape provides an inexpensive means of providing large amounts of off-line file-structured data storage.

PDP-11/20; extra 8K core (12K total) with cabinet and Teletype

RK11/RK03 1.2 million word DECpack Disk and Control and cabinet

RC11/RS64 64K fixed head DEC Disk and Control

TC11/TU56 Dual DECtape Transport and Control

BM792-YB ROM Bootstrap Loader

## 2.2 PAL-11R PROGRAM ASSEMBLY LANGUAGE
PAL-11R (Program Assembly Language for the PDP-11, Relocatable Version) operates under the Disk Operating System. Like PAL-11A, its counterpart in the Paper Tape System, PAL-11R provides the PDP-11 programmer a means of writing programs with meaningful symbols rather that with numerical code of usually no mnemonic value. However, with this relocatable version, symbols are assembled into object modules which are then processed by the LINK-11 Linker. LINK-11 produces a load module that is loaded for execution by the Monitor RUN command. Object modules may contain absolute and/or relocatable code; and separately assembled object modules may be linked with the aid of global symbols. The object module is produced after two passes through the Assembler. A complete octal/symbolic listing of the assembled program may also be obtained. This listing is especially useful for documentation and debugging purposes.

Some notable features of PAL-11R are:

Selective assembly pass functions

Device and file name specifications for pass functions

Error listing on command output device

Double buffered and concurrent I/O

Alphabetized, formatted symbol table listing

Relocatable object modules

Global symbols for linking between object modules

Conditional assembly directives

Program sectioning directives

Instruction mnemonics and statement format are identical to those of PAL-11A, described in the previous chapter. However, labels in PAL-11R may have either absolute or relocatable values. In the latter case, the final (absolute) value is assigned by the Linker by adding a relocation constant to it.

PAL-11R assembler directives include those of PAL-11A, described in the previous chapter, except that .EOT is effectively ignored under the Disk Operating System.

## 2.3 EDIT-11 TEXT EDITOR

The DOS Text Editor, Edit-11, is an on-line text editing program providing character, line, and file manipulations. Edit-11 will read and write ASCII files to and from any device.

In addition to normal editing functions, Edit-11 provides for command macros and multiple input/output files.

An 8K system can accommodate about 4000 characters of text. All additional core memory is available for text storage, i.e., about 8000 characters of text for each additional 4K memory bank.

## 2.4 ODT-11R DEBUGGING PROGRAM

ODT-11R is the on-line debugging program for the PDP-11 Disk Operating System. It is a system program which aids in debugging assembled and linked object programs. From the teleprinter keyboard the user interacts with ODT-11R and the object program to:

print the contents of any location for examination or alteration,

run all or any portion of your object program using the break- point feature,

search the object program for specific bit patterns

search the object program for words which reference a specific word,

calculate offsets for relative addresses,

fill a block of words or bytes with a designated value.

## 2.5 PIP-11 FILE UTILITY PACKAGE

The File Utility Package performs file handling operations for the PDP-11 Disk Operating System (DOS). Some examples are file transfers, directory listings, and file renaming. The Package is named PIP (Perih- eral Interchange Program) to be compatible with similar programs on other DEC systems.

### 2.5.1 File Handling

The transferring of files between devices is one of PIP's primary functions. There are two basic methods of file transfer:

1. Transferring and combining -- used to combine several files from one or more source devices into one file on the destination device.

2. Transferring without combining -- used to move several files from the source devices to the destination device as in- dividual files.

A file is specified by a file extension and filename. Several files can be specified by using the asterisk * in place of the filename, extension, or both. The * symbol denotes "all".

For example:

```
DT0: < *.PAL
```

will transfer all files with the extension PAL from the systems device to DECtape unit 0.

```
MAIN.*/BR
```

will output a brief directory listing all files with the file name MAIN.

```
*.TMP/DE
```

will delete all files with the extension TMP from the systems device. Unless specified the systems device is assumed to be the disk.

A comprehensive description of PIP's features and operation is contained in the PDP-11 PIP File Utility Package, Programmer's Manual, DEC-11-PIDA-D.

## 2.6 LINK-11 LINKER

The LINK-11 Linker is a system program for linking and relocating user programs assembled by the DOS Assembler. It enables the user to separately assemble his main program and various subprograms without assigning an absolute address for each segment at assembly time.

The binary output (object module) of each assembly can be processed by LINK-11 to:

Relocate each object module and assign absolute addresses.

Link the modules by correlating global symbols defined in one module and referenced in another module.

Produce a load map which displays the assigned absolute addresses.

Create a load module which can subsequently be loaded (by the Monitor or the Absolute Loader) and executed.

The advantages of using LINK-11 include:

The source program can be divided into segments (usually sub-routines) and assembled separately. If an error is discovered in one segment, only that segment needs to be reassembled. LINK-11 can then link the newly assembled object module with other object modules.

Absolute addresses need not be assigned at assembly time; the Linker automatically assigns absolute addresses. This keeps programs from overlaying each other. This also allows subroutines to change size without influencing the placement of other routines.

Separate assemblies allow the total number of symbols to exceed the number allowed in a single assembly.

Internal symbols (which are not qlobal) need not be unique among object modules. Thus, naming rules are required for global symbols only when different programmers prepare separate subroutines for a single program.

Large numbers of commonly used routines can be kept in a library and be retrieved with the Library search facility of the Linker.

Selective DOS monitor modules which are normally disk resident and swapped on request can be selected to be core resident for the duration of a program run using the Linker's DOS monitor Library search feature.

A core library facility is provided, with the user optionally requesting that the defined symbols be written onto a file for retrieval by later linking process.

## 2.7 LIBR-11 LIBRARIAN

The PDP-11 Librarian (LIBR-11) is a system program for the Disk Operating System providing facilities for creating, modifying, deleting, and listing the contents of libraries. A library can be created from one or more files. A file consists of one or more object modules, i.e., the binary output of the DOS Assembler.

LIBR-11 is a valuable program for the DOS user because;

It eliminates having separate directory entries in a User File Directory (UFD) for each object module.)

It expedites the linking process in conjunction with the Linker's library search capabilities.

It allows for standardization and controlled updating of frequently used routines, e.g., FORTRAN cosine routine.

The user controls the operation of LIBR-11 through command strings typed on the keyboard. Specified in the command strings are such things as devices, library, file, object modules name, and switches which indicate the LIBR-11 operation desired. The user can direct LIBR-11 to:

Create a library

Update a library

Insert one or more object modules in a library

Replace one or more object module in a library

List the directory of a library

Delete one or more object modules from a library

Delete an entire library

A directory listing of the object modules of a library can be obtained merely by specifying the device on which the directory is to appear and the name of the library.

The flexibility of LIBR-11 enables the user to specify certain combinations of operations in a single command string. For example, a library can be modified, renamed, and listed in one command string.

PART II
CHAPTER 3

# FORTRAN IV

FORTRAN IV (FORmula TRANslation) language is a problem-oriented language designed to help scientists and engineers express a computation in a notation with which they are familiar. A FORTRAN source program is composed of statements in easy-to-read form. Commands are descriptive of the functions they perform, and computa- tional elements are expressed in a notation similar to that of standard mathematics.

PDP-11 FORTRAN IV is an ANSI-standard FORTRAN IV compiler with elements that provide easy language compatibility with IBM 1130 FORTRAN. Since PDP-11 FORTRAN runs in the DOS environment, it requires only the hardware necessary to run DOS. There are no other hardware requirements, but the system will take advantage of added resources; more than 8K of core provides faster compilations and/or compilation of larger programs. PDP-11 FORTRAN uses DOS monitor I/O calls, and will support all peripherals supported by the disk operating system.

Some of the advantages of PDP-11 FORTRAN are:

random access I/O

mixed mode arithmetic is supported

generalized expressions are allowed as array subscripts

implicit statements allow the user to conveniently control the data type of variables

improved error diagnostics. A useful error traceback feature specifies: a) precisely where an error occured, b) all the linkages back to the main program

arithmetic can be performed with or without the PDP-11 Extended Arithmetic Element; PDP-11 FORTRAN will provide up to 24-bit accuracy for two-word formats (real), or up to 56-bit accuracy for four words (double-precision)

character-handling capability with the LOGICAL *1 capability

the ability to conserve core memory by selecting ONE WORD integers

the ability to generate relocatable binary code directly from the compiler, or to generate intermediate assembly code for custom modifications

extensive compiler diagnostics with text accompanying the diagnostic. The text may optionally be omitted

a completed, comprehensive and reentrant math library and object time system.

PART II
CHAPTER 4

# COMMUNICATIONS SOFTWARE COMTEX-11

COMTEX-11 (Communications Oriented Multi-Task Executive) is a communications software package for the PDP-11 family of computers. COMTEX-11 provides the following benefits:

Maximizes message throughput by fast processing of bursts

Software support for PDP-11 Communication Line Adaptors

Software support for standard DEC terminals

Compact reentrant code for core savings

Efficient set of user program commands initiate COMTEX-11 functions

Modular and expandable program modules for easy adaptation to user requirements

Defines programming conventions for communication tasks

## 4.1 COMTEX-11 APPLICATIONS

COMTEX is intended for use in any system connected to communication lines or servicing multiple data terminals. Applications are:

Remote Batch

Store and Forward

Front Ends

Satellite Processors

Concentrators

Message Switching

Telemetry

## 4.2 COMTEX-11 DESCRIPTION

COMTEX is a modular, reentrant software package for servicing of communication line interfaces and communication terminals. To control the line interfaces and control or transmit to the terminals, the co-resident user program need only make executive calls to the monitor (SCIP). COMTEX, via the SCIP, returns status information to the user program by placing this data into a circular queue accessible via a COMTEX executive command.

The modular nature of COMTEX allows the user to easily replace, add to or modify the terminal-dependent code in COMTEX. The terminal-oriented routines known as TAP's (Terminal Application Programs) are completely transparent to the type of line controller. TAP's perform functions such as special character detection, terminal control and code conversion. TAPs are reentrant and table-oriented; thus, one TAP can service multiple terminals of the same type.

The routines performing line control functions, called ISRs (Interrupt Service Routines), are transparent to all functions not related to line control. The ISRs perform functions such as modem control, and the mechanics of data input and transmission. One copy of an ISR can service multiple line controllers of the same type.

All COMTEX internal operations are scheduled on a priority basis so that time-critical functions are performed at high priority levels. Functions requiring fast service are character-buffer-unloading or end-of-block detection. These tasks must be serviced quickly to prevent data overrun. Jobs such as code conversion can be performed at lower priority levels.

COMTEX-11 system-building uses the PDP-11 assembler (PAL11-S). System build parameters consist of the type of terminals, type of line control units, and number of lines. These factors determine which TAPs, ISRs and line tables are required by the system. User programs to be co-resident with COMTEX may be written for assembly using any of the PDP-11 assemblers.

Assemblers are available for host machines such as PDP-10, CDC 6000 and IBM 360 systems from the DEC User's Society (DECUS).

## 4.3 COMTEX-11 DISTRIBUTION

Technical information on all DIGITAL Communication products may be obtained from the engineering and programming teams resident in DIGITAL sales offices.

The COMTEX-11 software package including manuals, detailed flow charts, timing information, source and binary tapes, listings and training may be ordered through any DIGITAL office.

**Table 4-1 COMTEX-11 Commands**

| | | |
|---|---|---|
| LINIT | (Line INITialization) | Associates logical line number with physical characteristics of the line. |
| PUTMC | (PUT Modem Control) | Control functions to modem |
| PUTTC | (PUT Terminal Control) | Control functions to terminal |
| ASRBUF | (ASsign Receive Buffer) | Assign a buffer for input and allow input to commence |
| PUTD | (PUT Data) | Initiate data transmission |
| GETS | (Get Status) | Return status information to the user program. |
| PUTTM | (PUT TiMer) | Provides user program with time and time-out information |

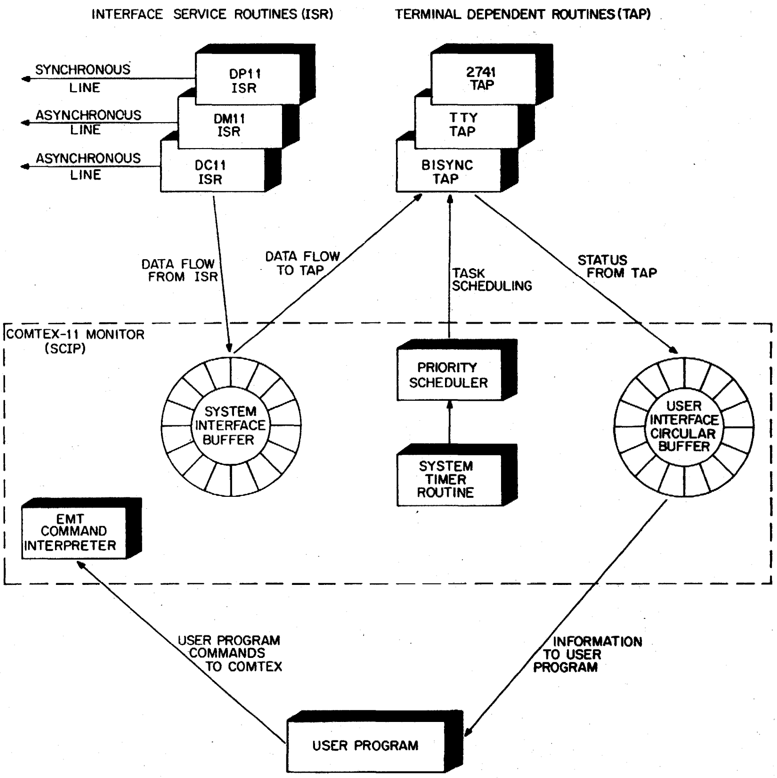


Figure 4-1 COMTEX Block Diagram

## 4.4 CORE REQUIREMENTS

**Core requirements for COMTEX-11 are:**

| | |
|---|---|
| System Control Interface Package (SCIP) | 1300 |
| KL11 Interrupt Service Routine (ISR) | 250 |
| DC11 (ISR) | 440 |
| Interactive Teletype (TAP) | 1000 |
| SCIP Table Space | 16/line |
| TAP Table Space | 22/line |
| ISR Table Space | 9/line |

PART II
CHAPTER 5

# REAL TIME EXECUTIVE RSX-11C

RSX-11C (Real Time Executive) is a software package that provides for task scheduling, input-output, operator communication and other functions required for real time multiprogrammed operation.

User tasks can be written to operate under the control of RSX-11C using either assembly language or FORTRAN IV.

The handling of program scheduling and input-output by the real-time monitor makes the use of a high-level language such as FORTRAN possible. FORTRAN IV programs including real-time calls are supported by RSX-11C. The use of FORTRAN with a general purpose real-time executive provides a software environment which makes the real-time computer a practical operation tool for the process engineer, test engineer or researcher. This means that with only a knowledge of FORTRAN he can get his PDP-11 system producing results in a matter of days, and can take advantage of FORTRAN code written for other systems.

FORTRAN programs must be compiled on a PDP-11 system under the Disk Operating System (DOS) control. Machine language programs can be assembled online if sufficient core is available. RSX minimum requirements are 12K, a KW11L real-time clock, ASR Teletype and high speed reader/punch.

## 5.1 LANGUAGES SUPPORTED

The user can write all of his tasks in FORTRAN; not only the arithmetic, logic and control functions of standard FORTRAN but also functions of task starting, sequencing and input-output.

RSX-11C supports FORTRAN calls for real time functions.

A relocatable assembler and linkage editor can also be used to build user tasks.

## 5.2 SCHEDULING STRATEGY

When a user loads a task in the system he must specify one of three levels of priority. These three software or user levels are all below the four system levels of priority which are entered due to an I/O interrupt or due to instruction trap interrrupts.

The three software (user) interrupt levels are true priority levels. For instance, if an interrupt occurs indicating it is time for a new task to begin, and the new task is of higher priority than the task interrupted, the low priority task is suspended and the higher level task activated. If the higher level task gets suspended, the lower level task is continued until the higher level task can resume operation.

### 5.2.1 System Response Time - User Levels

System response time for user tasks depends mainly on whether another user level task is running at this or a higher level. A task that runs too long at a high priority level can therefore destroy the response time of other tasks. To avoid this an important design feature of RSX-11C is a software Task Watch Dog Timer. This timer is set at the start of each task with the maximum duration a task may run, at a particular level, before suspending or exiting. This time limit is a system parameter for each priority level. Typical values may be 100 milliseconds for the highest level, one second for the intermediate level and unlimited time for the lowest level. If this time limit is exceeded the task is reduced in priority and must compete for machine time with other tasks at the next lower level. If it moves to the lowest level, it is then allocated time slices on a round robin basis with other tasks running at this level. At the end of each time slice, a check is made to see if it has exceeded a maximum run time defined for this task. If this time has been exceeded, an error report is generated.

A fourth level of priority is available and used by the system tasks. This level is higher than the three user levels and is used for functions of very short duration. No watch dog time is set for this level. User tasks of very short duration may also be loaded into this fourth level if they require exceptionally fast response times.

### 5.2.2 System Response Time - System (Interrupt) Levels

Normally, executive functions (scheduling, I/O, etc.) are active on the four hardware priority levels. However, special user code can be placed also at these levels. Programs at these levels are entered due to a hardware interrupt and may be stopped by higher priority programs.

## 5.3 MEMORY EFFICIENCY

Commonly used subroutines, such as the FORTRAN arithmetic library, formatter, etc. can be loaded as part of the RSX-11C package and shared by all user programs. This can be done because these subroutines are reentrant, i.e., they can be interrupted while being used by one task and then re-entered for use by other tasks.

## 5.4 MULTIPROGRAMMING CAPABILITY

RSX-11C can handle many concurrent real-time tasks and a single background task. The number is limited by the memory capacity of the computer, and is typically less than 128.

## 5.5 INPUT/OUTPUT

RSX-11C controls and executes all input and output operations. This is one of the areas of most concern to real-time users, because most real-time applications are characterized by a large amount of input and output.

All output transfers from the program to I/O devices are buffered. Programs are not suspended if room exists in an output buffer for characters being output.

With this feature the engineer does not have to worry about machine language I/O programming, since all I/O requests are performed by the executive in response to simple I/O commands. Executive calls of this type are identical to those used in the (DOS) Disk Operating System used for data processing in the PDP-11. Programs may be easily transferred between this operating system and RSK-11C.

## 5.6 OPERATOR COMMUNICATION

Simple operator commands are provided to load, start, stop and delete a particular program. Commands are also provided to set the time-of-day, and to interrogate system status.

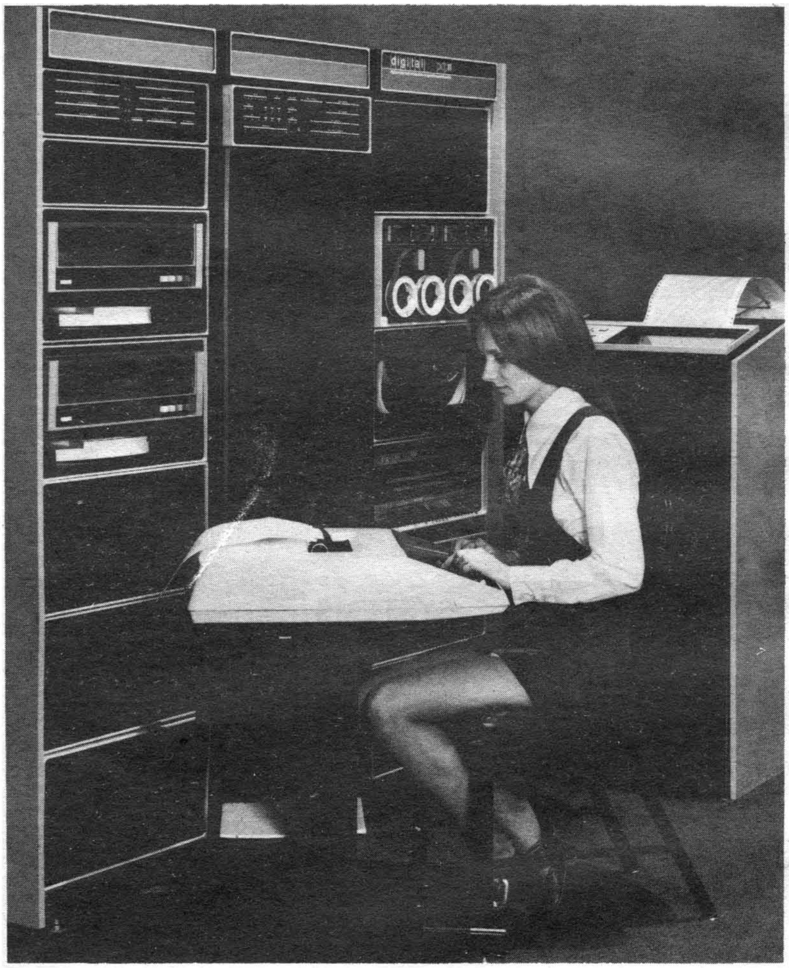
## 5.7 PROGRAM DEVELOPMENT

Program development can be done on-line or off line using the PAL-11R assembler and LINK-11. Object modules produced by the assembler must be processed by the linker to produce a binary load module which can then be loaded via the On-Line Loader Task.

If required, the assembler, linker and symbolic editor can be operated as background tasks. The On-Line Loader Task loads modules generated by the Linker. The loader checks modules being loaded against a memory map for proper fit. The On-Line Loader operation does not interfere with the operation of the real time system.

# part 3

## SYSTEMS


digital

PART III
CHAPTER 1

# TIMESHARING SYSTEM RSTS-11

RSTS-11 is a timesharing system developed for the PDP-11. "RSTS" stands for Resource Time Sharing System to reflect the capability of allowing terminal users to access high-speed input/output peripheral devices within their application programs.

Other distinguishing characteristics of RSTS-11 include:

applications program development in a greatly extended version of the Dartmouth BASIC programming language.

sequential and random access to on-line disk files with a total capacity as large as 32 million characters.

support for both local and remote interactive terminals operating at up to 1200 Baud transmission speed.

up to 16 simultaneous terminal users.

## 1.1 PROGRAMMING LANGUAGE

RSTS-11 applications programs are written in a greatly extended version of Dartmouth BASIC, named BASIC-Plus. Because of the popularity BASIC now enjoys as an educational tool, a large body of teaching materials, both textbooks and programs, have been developed which further enhance the value the language. One of the benefits of the language extension is that students are less likely to "outgrow" the language as they become more experienced in programming techniques.

BASIC is widely used in industry for computational problem-solving via time-sharing service bureau terminals. It is important that the language features have sufficient scope so that the difficulty of conversion of programs written in any of the large number of versions of BASIC be minimized.

The more significant features of BASIC-Plus include:

extensive set of character string manipulation operators and functions

an integer data type for more efficient computation (e.g., counting) operations

programmed format control for print files

programmed sensing and recovery from computational and input/output errors at the user level

access to sequential and random-access disk files

extensions to the syntax of Dartmouth BASIC to permit more concise programs and more efficient execution.

Example :

```
        If X=Y THEN A(1)=X ELSE GOTO 550 LET B1=R5 IF R5=4)
```

**1.1.1 Character String Processing**
The design of the BASIC-Plus language gives particular emphasis to flexible and efficient manipulation of alphanumeric character string data. Computer Aided Instruction applications consist largely of the input and output of large quantities of text data. The ability to handle alphanumeric records and fields is essential in business information processing.

The character string manipulation features permit the programmer to define an internal character string variable of indefinite length, concatenate strings (append strings end-to-end to form a new string), extract a substring of arbitrary length from any part of a string variable, and search for a string within a string. Character string records up to 512 records long may be stored in disk files. String functions permit the conversion of numeric values to strings and vice versa.

**1.1.2 Integer Data Type**
BASIC-Plus includes the definition of integers in addition to strings and floating point numbers. Integers are whole numbers in the range of –32,767 to +32,767. The use of integers often increases the execution efficiency of programs. The most common uses of integers are in counting and indexing operations.

**1.1.3 Print Formatting**
Many applications, such as business data processing, require more flexible control of the printing format than Dartmouth BASIC allows. BASIC-PLUS includes a PRINT USING statement which may be used to acheive precise definition of printed data format. PRINT USING allows character, decimal, and exponential data field lengths and positions to be defined, and mixed, for a print line. In addition, leading dollar or asterisk symbols may be "floated" to automatically precede the most significant digit of decimal fields. Trailing minus signs for data fields may be specified for compatibility with accounting report standards.

**1.1.4 Programmed Error Recovery**
One of the more frustrating situations for a timesharing terminal user occurs when a program is cancelled because an input/output error condition occurs (perhaps temporarily) and causes all results created (in a file, for example) to that point to be lost. This problem can be particularly serious in an administrative application which is processing files. This situation can be controlled by the applications programmer by use of the ON ERROR GOTO statement. This subroutine call statement is triggered by a variety of input-output and computa- tional errors. The called subroutine is passed, a value which identifies the error type, and attempts to recover from the error condition. If the subroutine is successful, normal execution of the application program resumes. Thus, in effect, the programmer can design an executive system within his own application which supplements the services provided by the RSTS-11 system monitor.

**1.1.5 Disk File Access**
RSTS-11 users may create and have high-speed access to program and data files stored on disk units with total file space of up to 32,000,000 bytes. Files may be created for either sequentials or random access processing, depending upon the requirements of a user's application. Up to 12 files may be open and accessible from a single program at any one time. The number of files a user may have stored in the disk library is bounded only by the total system disk capacity and the library demands of other users.

An on-line file library system means that RSTS-11 terminal users have the convenience of almost instant access to any desired file or file item. Terminal users are spared the problems and frustrations of handling paper tape each time a program is to be executed. Many applications such as on-line customer inquiry-response are possible with the large-scale file library system of RSTS-11.

Each terminal user has full control on the degree of privacy he desires for each file he creates. The disk library file directory system, which provides efficient access to files, includes a privacy-protection level which may be set only by the terminal user responsible for creation of the file. Personnel records, for example, can be given absolute protection from all other users. Other levels of protection include access limited to a particular group of users, read only, write only, and public. Files may be stored on-line on DECpack removable disk cartridge drives, DECdisk fast-access fixed-head disk units, and removable disk packs with a capability of 32 million bytes, total, for on-line storage of frequently used files.

**1.1.6 Extended BASIC Language Features**
The effectiveness of RSTS-11 in solving problems in a broad variety of application areas is significantly increased with the addition of numerous extensions to the structure (syntax) of the BASIC program statements. These highly flexible program statements permit more concise expression of complex program steps.

Some examples are:

```
LET A1=P1*R1 IF R1=5.0 OR R1=0.0

GOTO 5530 UNLESS X1$=Y1$ AND Z$

LET X(Y1,Z1)=Z1*3 FOR Z1=1 TO L

FOR I=X(J) STEP 3 WHILE L$(I)=L$(I+1) AND J+I=12

ON X(2,5) GOTO 100, 150, 200, 250, 300
```

## 1.2 PROGRAM DEVELOPMENT FACILITIES
A relatively high percentage of timesharing systems used in both schools and industrial organizations is either developing or modifying applications programs. This is because problems in these environments are often of a "one-shot" nature. Students have project assignments and engineers have computational problems requiring special programs.

RSTS-11 provides a number of features which assist terminal users in developing, modifying, and debugging BASIC-Plus programs. The following features are available:

1. Each program statement is checked for errors in syntax and format. If an error is found, a diagnostic message is reported immediately.

2. Program statements may be entered in any line-number order, so that if a user discovers that he omitted a line, he may enter it immediately without having to type any special commands.

3. Once all program statements are entered, the program may be executed immediately without having to type any special commands.

4. Program statements may be changed by simply retyping the line number and statement. (To delete a statement the line number is followed by a carriage return key).
5. For debugging purposes, STOP statements may be temporarily inserted in a program. When a STOP statement is encountered during execution, a message is typed indicating the line number of the STOP statement which interrupted execution. Like-wise a program may be interrupted "at random"by typing the CTRL/C key combination. The terminal user may then use immediate mode statements to print the values of an variables in his program, modify values of variables, and resume the execution of the program.
6. Statements in a program may be added, modified, or deleted, and the program rerun without a waiting time for recompilation of the entire program.
7. All debugging is performed at the source program level rather than requiring knowledge of PDP-11 machine level instructions.

These features permit a programming session to be carried out in a highly conversational manner, thus minimizing the user's time in developing or modifying a program.

To support the previously-listed programming facilities, RSTS-11 utilizes an incremental compiler. The compiler is core-resident, reentrant, and can be shared by all terminal users. The incremental compiler generates a highly efficient intermediate language code which allows application programs to be executed with a high degree of efficiency.

### 1.2.1 Desk Calculator Mode

The facilities of the incremental compiler also provide a "desk calculator" service to terminal users. BASIC-Plus statements which are entered without a preceding line number are compiled and executed immediately. In a sequence of one or more statements entered in immediate mode, a terminal user may assign values to variable, perform operations upon them, and print out results of computational operations Thus, the statement: PRINT A(I). SQR<A(I)< FOR I=1 TO 100 will print out a square root table.

## 1.3 INPUT/OUTPUT PERIPHERAL ACCESS

An important feature of RSTS-11, distinguishing it from most small-computer timesharing systems, is that a terminal user may "configure" a collection of input/output devices needed to execute his application with high efficiency. The objective of this resource sharing concept is to overcome the input-output bottleneck associated with the use of interactive terminals alone - whether they be used with an in-use computer or on a timesharing bureau. For example, an RSTS-11 terminal application program might use a punched-card reader for input of transaction records, a magnetic tape file for updating a sequential file which is a log of all transactions, and a high-speed line printer for printing a transaction report.

Another benefit of the resource sharing concept for organizations which cannot afford an RSTS-11 configuration with extensive on-line disk storage capacity is that infrequently used programs and data files may be stored on reels of DECtape. Two inexpensive DECtape transports are included in the RSTS-11 configuration. Because files may be transferred between reels of DECtape and on-line disk storage quickly and conveniently, the demand for on-line disk space may be effectively controlled.

Access to high speed peripherals is assigned by the RSTS-11 system monitor upon user request on a first-come, first-served basis. When a user no longer needs access to a particular peripheral device, he may type a command to the system to free the device for use by other terminal users.

## 1.4 RSTS-11 INTERNAL SYSTEM

RSTS-11 timesharing service is supported by a software system composed of: a monitor, a compiler/editor, and a runtime system. The software runs on a standard PDP-11 with a minimum of 24K words of 16-bit core memory, a 256K word fixed-head disk, a dual-transport DECtape unit, real-time clock, bootstrap loader, user terminal interfaces and power supplies and mounting hardware. The configuration may be optionally extended with aditional disk units, magnetic tape transports, line printer, high-speed paper tape reader/punch, card reader, and additional core memory.

## 1.5 MONITOR FUNCTIONS

The purpose of the monitor is to control and allocate computer resources to RSTS-11 terminal users. A major portion of the monitor is core resident to minimize terminal response time.

The monitor uses a core-disk swapping strategy to allow terminal users a large amount of core memory space (up to 8K words) while a round-robin scheduling algorithm is used to determine which user should next be allocated a slice of processor time. If the next user-program in the round robin queue is waiting for processor time, the program is swapped from a high-speed systems disk to an available core memory area. The user's program is executed for a time-slice of either approximately 100 milliseconds or until the program requests input/output service, whichever is shorter.

## 1.6 SYSTEM ACCESS

Users are authorized terminal access to RSTS-11 via a user identification code. The code is composed of three parts: a project number, a programmer number and password. Up to 120 discrete users may have accounts.

RSTS-11 terminals may operate either local to the system (hard-wired) or remotely via communications lines. A wide variety of terminals operating at speeds from 10 to 120 characters per second may be used. Teletypes, cathode ray tube displays and the new DECwriter (a 30- character-per-second hardcopy terminal) are currently supported.

PART III
CHAPTER 2

# COMMUNICATIONS

Because of its UNIBUS architecture and other advanced features, the PDP-11 is a natural communications processor. The PDP-11's adaptability to communications environments is further enhanced by DEC's advanced general purpose communications oriented software executive (COMTEX-11) and by extensive communications hardware. By combining the PDP-11 with COMTEX-11 modules and DEC's communications hardware, many systems can be configured for remote terminal, data concentration, message switching and front end preprocessing applications.

## 2.1 PDP-11 ARCHITECTURE

The PDP-11 provides the following advantages for communications applications:

The UNIBUS asynchronous data bus behaves like a multiplexer. Multiple single-line communications interfaces can be added to the PDP-11 without special multiplexing hardware.

The physical modularity of the PDP-11 makes it easy to reconfigure. PDP-11 system units connect directly to the UNIBUS and allow easy expansion of memory or communications line interfaces. Processors, memories and communications interfaces can be easily replaced in the event of failure or as more powerful units become available.

The PDP-11 handles bytes easily and efficiently. Byte handling is the crux of communications applications; and each 8-bit byte is directly addressable with a full set of byte instructions.

The PDP-11 handles large core systems easily. The UNIBUS uses 18 address bits and allows 262K bytes or 131K words to be addresses.

Eight general registers combine with addressing modes to offer very efficient string or list processing operations. General registers are used as full 16-bit index registers; this allows code conversions to be performed easily.

For example:

```
  MOV    TPB,R5      ;get the EBCD code from Rcve Buffer
  MOVB   BASE(R5)      ;convert to equivalent ASCII Code
```

Note that I/O device registers are accessed with standard instructions. This brings the full power of the PDP-11 instruction set to bear on I/O programming.

The dynamic stack capabiltiy associated with subroutine call and interrupt processing permits reentrant coding and fully nested interrupts. Reentrant code lets multiple devices share the same service routines. Nested interrupts allow higher-priority service routines to interrupt lower-priority routines.

Vectored interrupts reduce the overhead associated with an interrupt. The PDP-11 branches directly to each interrupt service routine thus saving the time usually required to identify the interrupt. This increases the number of lines a communications system can handle.

Flexible interrupt priority structure provides the system designer with full control over the hardware and software priority assignments.

UNIBUS design allows easy and inexpensive use of direct memory access devices. The single-bus system reduces the cost of cabling and electronics associated with DMA devices.

## 2.2 COMMUNICATIONS HARDWARE

DEC communications equipment is summarized below and explained in greater detail in the PDP-11 Peripherals and Interfacing Handbook.

**Asynchronous Line Interface (DC11)**

Full- or Half-Duplex Operation
Programmable Line Speed (4 speeds)
Input and Output Speed Independent
Programmable Character Size (5,6,7, or 8 bits)
Parity Check on Incomming Characters
Interfaces to Bell 103, 202, or Equivalent Modems
Auto Answering Capability
Reverse Channel for Bell 202 Operation

**Asynchronous 16-Line Single Speed Multiplexer (DM11)**

Full- or Half-Duplex Operation
DMA Character Assembly in Core Memory
DMA Message Transmission from Core Memory
Rates up to 1200 Baud
Character Size Jumper Selectable (5,6,7,8 bits)
Parity Check on Incoming Characters
Break Detection
Reverse Break Generation
64 Character Tumble Table for Buffering Incoming Characters
Transmitter and Receiver Priority Independent
Up to 16 DM11's per PDP-11 System

**Synchronous Line Interface (DP11)**

Double-Buffered Program Interrupt Character Service
Full- of Half-Duplex Operation
Programmable Sync Character
Programmable Character Size (6,7, or 8 bits)
Receiving Sync Character Stripping Program Selectable
Speeds up to 50,000 Baud
Interfaces to Bell 201 and 303 or Equivalent Modems
Auto Answering Capability
Internal Clocking Source (optional)

**Automatic Calling Unit Interface (DN11)**

Digit-Buffered Interface
Interfaces with Bell 801A or 801C or Equivalent Units.
Program Access to all Bits of the 801.

## 2.3 COMMUNICATIONS SOFTWARE

COMTEX-11, a communications oriented multi-task executive, provides extensive interrupt and data handling capability for a wide range of communications applications. Major features are:

Modularity and Expandibility

Low overhead priority task scheduling for maximum system performance

Interrupt service routines for all standard communications hardware

Terminal applications package for many common terminals

Transparent data communications front end to user's application program

COMTEX-11 is explained in more detail in PART II, Chapter 4.

## 2.4 COMMUNICATIONS APPLICATIONS

### 2.4.1 Front End Preprocessors

The PDP-11 offers a powerful, low-cost alternative to hardwired communications controllers on the front end of large computer systems. As a front end, the PDP-11 handles not only low- and medium-speed terminals such as Teletypes and CRT's but also remote-terminal controllers and remote-data concentrators. Functions performed by this type of system are similar to those of a terminal controller or a data concentrator.

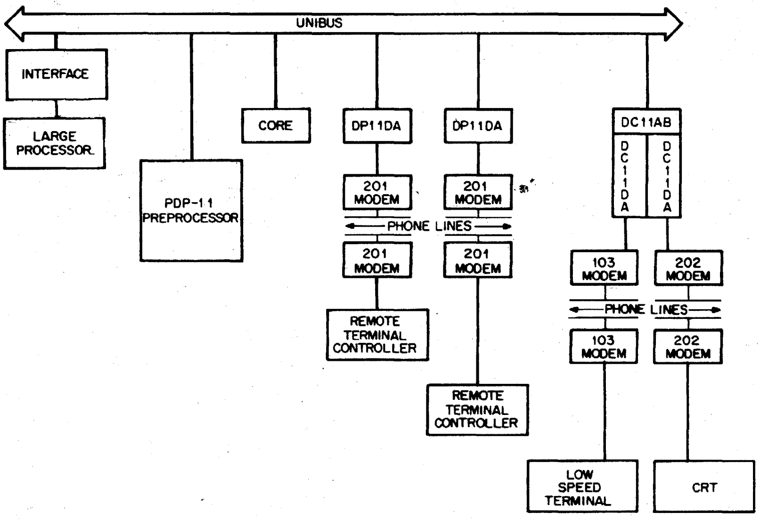


Figure 2-1 Front End Processor

### 2.4.2 Store and Forward Message Switchers

This type of system has a number of data terminals connected locally or via communications lines to a central computer. Any terminal can originate a message and transmit it to the central computer. Here the message is stored until it can be forwarded to the destination terminal. Typical functions performed by a store and forward message switcher are:

Assembly/disassembly of messages

Polling and addressing of terminals

Line control

Error control

Code and speed conversion

Message header analysis

Sequence number of messages

Time and date stamping of messages

Message routing

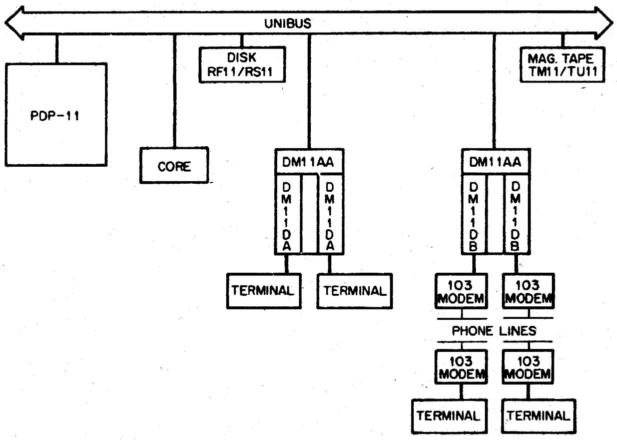


Figure 2-2 Store and Forward Message Switcher

### 2.4.3 Remote Terminal Controllers

This allows remote access to a batch processing facility. Information to be processed is stored on punched paper tape, punched cards or magnetic tape. Output can be displayed on a CRT, stored on magnetic tape, paper tape or printed on a line printer. Generally, the controller is transparent to the data being transmitted; but, it can be used to perform functions such as:

Code and speed conversion

Data compression

Line control

Error control

Message formatting

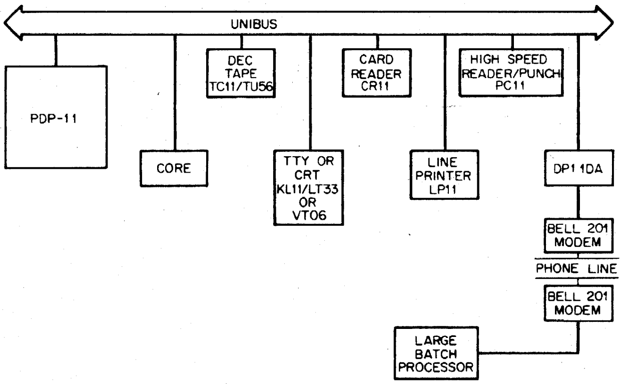


Figure 2-3 Remote Terminal Controller

**2.4.4 Data Concentrators**
A cluster of remote low-speed data terminals can often be interfaced more economically to a remote interactive computer via a data concentrator than by using a separate line per terminal. Communication line costs can be reduced by concentrating several low-speed terminals into a single medium-speed communication line using a data concentrator. Typically, a data concentrator performs the following functions:

Character-to-message assembly/disassembly

Communication Line control

Message buffering

Error control

Code conversion

Automatic answering

Automatic identification of the terminal type

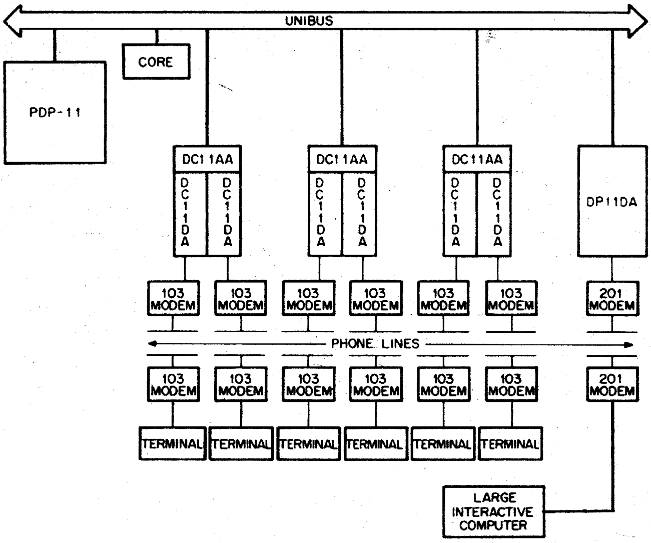


Figure 2-4 Remote Data Concentrator

PART III
CHAPTER 3

# INDUSTRIAL DATA ACQUISITION AND CONTROL SYSTEMS

Modular process interfaces, special state-of-the-art software (RSX-11C real-time executive) and the PDP-11 combine to provide efficient, low-cost and reliable systems for industrial data acquisition and control applications. IDACS-11 systems can serve either as on-the-floor satellite computers, or as stand-alone development/process control systems. These systems can provide flexible hierarchichal computer configurations with computer-to-process or computer-to-computer communication capabilities.

IDACS-11, a total system for real time data acquisition and control, consists of:

PDP-11 computer and peripheral devices

Truly industrial process interfaces

Real time operating software

## 3.1 PROCESS INTERFACES

The modular and reliable process interfaces are available for a wide variety of process signals. These industrial interfaces make possible the communications between a real live process and the PDP-11 computer. The following process I/O devices are offered for IDACS-11 systems:

flying capacitor scanner (AFC11) for low-level differential analog inputs. It is expandable to 1024 channels and is truly an industrial subsystem with high noise rejection.

universal digital controller (UDC-11) for discrete process input/output such as:

contacts, relays, switches, pushbuttons drivers for lamps or solenoids counters and analog outputs

analog-to-digital conversion subsystem (AD01-D) for single-ended high-level analog inputs. It has optional bipolar feature with automatic sign option, and it provides 10-bit precision, 14-bit resolution.

digital-to-analog converter (AA11-D) for analog outputs with 11-bit precision plus sign and bipolar output

## 3.2 REAL-TIME OPERATING SYSTEM

A real-time executive system (RSX-11C) is offered on IDACS-11 systems. It is a software package for coordinating the execution of user tasks in a multiprogramming mode. With it a test or process engineer can code tasks in FORTRAN language, compile them using PDP-11 disk operating software and then execute them. Communications to a higher level supervisory computer can be achieved with RSX-11C. RSX-11C is discussed in more detail in Chapter 5, Part II.

## 3.3 IDACS-11 APPLICATIONS

The modular structure and reliability of an IDACS-11 system makes it possible to implement the system on the the plant floor where the process is located. A small IDACS-11 satellite system can be used for:

Data acquisition from a live process

Monitoring and controlling a process or a production unit

Automated testing and quality control of components

Sequence control of a batch or an operation

Controlling a complex machine

An IDACS-11 system can be expanded to be a development and process control system. Working in this type of supervisory mode, an IDACS-11 system can be used for:

A process control system performing direct digital control, set point control, data gathering and record-keeping functions

A supervisory system communicating with in-plant satellite IDACS-11 systems or with a large central computer

A program development system for various IDACS-11 systems in a distributed network. This ensures the maximum system availabilty for new program development and debugging.

# APPENDIX A—PDP-11 INSTRUCTION REPERTOIRE

| Mnemonic | Instruction Operation | OP Code | Condition Codes ZNCV | Timing |
|---|---|---|---|---|
| **DOUBLE OPERAND GROUP: OPR scr, dst** | | | | |
| MOV(B) | MOVe (Byte)<br>(src) → (dst) | ·1SSDD | √√—0 | 2.3 |
| CMP(B) | CoMPare (Byte)<br>(src) — (dst) | ·2SSDD | √√√√ | 2.3* |
| BIT(B) | BIt Test (Byte)<br>(src) ∧ (dst) | ·3SSDD | √√—0 | 2.9* |
| BIC(B) | BIt Clear (Byte)<br>~ (src) ∧ (dst) → (dst) | ·4SSDD | √√—0 | 2.9 |
| BIS(B) | BIt Set (Byte)<br>(src) v | ·5SSDD | √√—0 | 2.3 |
| ADD | ADD<br>(src) + (dst) → (dst) | 06SSDD | √√√√ | 2.3 |
| SUB | SUBtract<br>(dst) — (src) → (dst) | 16SSDD | √√√√ | 2.3 |
| **CONDITIONAL BRANCHES: Bxx loc** | | | | |
| BR | BRanch (unconditionally)<br>loc → (PC) | 0004XX | —— | 2.6 |
| BNE | Branch if Not Equal (Zero)<br>loc → (PC) if Z = 0 | 0010XX | —— | 2.6— |
| BEQ | Branch if Equal (Zero)<br>loc → (PC) if Z = 1 | 0014XX | —— | 2.6— |
| BGE | Branch if Greater or Equal (Zero)<br>loc → (PC) if N ⊻ V = 0) | 0020XX | —— | 2.6— |
| BLT | Branch if Less Than (Zero)<br>loc → (PC) if N ⊻ V = 1 | 0024XX | —— | 2.6— |
| BGT | Branch if Greater Than (Zero)<br>loc → (PC) if Z v (N ⊻ V = 0) | 0030XX | —— | 2.6— |
| BLE | Branch if Less Than or Equal (Zero)<br>loc → (PC) if Z v (N ⊻ V) = 1 | 0034XX | —— | 2.6— |
| BPL | Branch if PLus<br>loc → (PC) if N = 0 | 1000XX | —— | 2.6— |
| BMI | Branch if MInus<br>loc → (PC) if N = 1 | 1004XX | —— | 2.6— |
| BHI | Branch if HIgher<br>loc → (PC) if C v Z = 0 | 1010XX | —— | 2.6— |
| BLOS | Branch if LOwer or Same<br>loc → (PC) if C v Z = 1 | 1014XX | —— | 2.6— |
| BVC | Branch if oVerflow Clear<br>loc → (PC) if V = 0 | 1020XX | —— | 2.6— |
| BVS | Branch if oVerflow Set<br>loc → (PC) if V = 1 | 1024XX | —— | 2.6— |
| BCC<br>(or BHIS) | Branch if Carry Clear<br>loc → (PC) if C = 0 | 1030XX | —— | 2.6— |
| BCS<br>(or BLO) | Branch if Carry Set<br>loc → (PC) if C = 1 | 1034XX | —— | 2.6— |

**SUBROUTINE CALL: JSR reg, dst**

| | | | | |
|---|---|---|---|---|
| JSR | Jump to SubRoutine<br>(dst)→ (tmp), (reg) ↓<br>(PC) → (reg), (tmp) → (PC) | 004RDD | —— | 4.4 |

**SUBROUTINE RETURN: RTS reg**

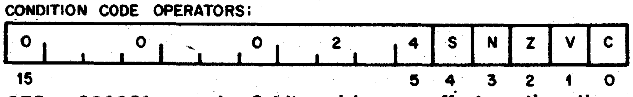
| | | | | |
|---|---|---|---|---|
| RTS | ReTurn from Subroutine<br>(reg) → PC, ↑ (reg) | 00020R | —— | 3.5 |

**SINGLE OPERAND GROUP: OPR dst**

| | | | | |
|---|---|---|---|---|
| CLR(B) | CLeaR (Byte)<br>0 → (dst) | ·050DD | 1000 | 2.3 |
| COM(B) | COMplement (Byte)<br>~ (dst) → (dst) | ·051DD | ✓✓00 | 2.3 |
| INC(B) | INCrement (Byte)<br>(dst) + 1 → (dst) | ·052DD | ✓✓—✓ | 2.3 |
| DEC(B) | DECrement (Byte)<br>(dst) — 1 → (dst) | ·053DD | ✓✓—✓ | 2.3 |
| NEG(B) | NEGate (Byte)<br>~ (dst) + 1 → (dst) | ·054DD | ✓✓✓✓ | 2.3 |
| ADC(B) | ADd Carry (Byte)<br>(dst) + (C) → (dst) | ·055DD | ✓✓✓✓ | 2.3 |
| SBC(B) | SuBtract Carry (Byte)<br>(dst) — (C) → (dst) | ·056DD | ✓✓✓✓ | 2.3 |
| TST(B) | TeST (Byte)<br>0 — (dst) | ·057DD | ✓✓00 | 2.3* |
| ROR(B) | ROtate Right (Byte)<br>rotate right 1 place with C | ·060DD | ✓✓✓✓ | 2.3° |
| ROL(B) | ROtate Left (Byte)<br>rotate left 1 place with C | ·061DD | ✓✓✓✓ | 2.3° |
| ASR(B) | Arithmetic Shift Right (Byte)<br>shift right with sign extension | ·062DD | ✓✓✓✓ | 2.3° |
| ASL(B) | Arithmetic Shift Left (Byte)<br>shift left with lo-order zero | ·063DD | ✓✓✓✓ | 2.3° |
| JMP | JuMP<br>(dst) → (PC) | 0001DD | —— | 1.2 |
| SWAB | SWAp Bytes<br>bytes of a word are exchanged | 0003DD | ✓✓00 | 2.3 |

**CONDITION CODE OPERATORS: OPR** 1.5

Condition Code Operators set or clear combinations of condition code bits. Selected bits are set if S = 1 and cleared otherwise. Condition code bits corresponding to bits set as marked in the word below are set or cleared.



Thus SEC = 000261 sets the C bit and has no effect on the other condition code bits (CLC = 000241 clears the C Bit)

**OPERATE GROUP: OPR**

| | | | | |
|---|---|---|---|---|
| HALT | HALT<br>processor stops; (RO) and the HALT address in lights | 000000 | —— | 1.8 |
| WAIT | WAIT<br>processor releases bus, waits for interrupt | 000001 | —— | 1.8 |

| | | | | |
|---|---|---|---|---|
| RTI | ReTurn from Interrupt<br>↑ (PC), ↑ (PS) | 000002 | √ √ √ √ | 4.8 |
| IOT | Input/Output Trap<br>(PS) ↓, (PC) ↓, (20) → (PC), (22) → (PS) | 000004 | √ √ √ √ | 9.3 |
| RESET | RESET<br>an INIT pulse is issued by the CP | 000005 | —— | 20 ms. |
| EMT | EMulator Trap<br>(PS) ↓, (PC) ↓, (30) → (PC), (32) → (PS) | 104000—104377 | √ √ √ √ | 9.3 |
| TRAP | TRAP<br>(PS) ↓, (PC) ↓, (34) → (PC), (36) → (PS) | 104400—104777 | √ √ √ √ | 9.3 |

**NOTATION:**

1. for order codes
   - · — word/byte bit, set for byte (+100000)
   - SS—source field,
   - DD—destination field
   - XX—offset (8 bit)
2. for operations

| | |
|---|---|
| ∧ | and, |
| v | or, |
| ~ | not, |
| ( ) | contents of, |
| ⊻ | XOR |
| ↓ | "is pushed onto the processor stack" |
| ↑ | "the contents of the top of the processor stack is popped and becomes" |
| → | "becomes" |

3. for timing

| | |
|---|---|
| * | 0.4 μs less if not register mode |
| — | 0.9 μs less if conditions for branch not met |
| ° | 1.2 μs more if addressing odd byte<br>(0.6 μs additional in addressing odd bytes otherwise) |

4. for condition codes

| | |
|---|---|
| √ | set conditionally |
| — | not affected |
| 0 | cleared |
| 1 | set |

# APPENDIX B MEMORY MAP

**PDP 11 DEVICE REGISTERS AND INTERRUPT VECTORS.**
VECTORS

| | | |
|---|---|---|
| 000 | RESERVED | |
| 004 | TIME OUT, BUS ERROR | |
| 010 | RESERVED INSTRUCTION | |
| 014 | DEBUGGING TRAP VECTOR | |
| 020 | IOT TRAP VECTOR | |
| 024 | POWER FAIL TRAP VECTOR | |
| 030 | EMT TRAP VECTOR | |
| 034 | "TRAP" TRAP VECTOR | |
| 040 | SYSTEM SOFTWARE | |
| 044 | SYSTEM SOFTWARE | COMMUNICATION WORDS |
| 050 | SYSTEM SOFTWARE | COMMUNICATION WORDS |
| 054 | SYSTEM SOFTWARE | COMMUNICATION WORDS |
| 057 | | |
| 060 | TTY IN-BR4 | |
| 064 | TTY OUT-BR4 | |
| 070 | PC11 HIGH SPEED READER-BR4 | |
| 074 | PC11 HIGH SPEED PUNCH | |
| 100 | KW11L - LINE CLOCK BR6 | |
| 104 | KW11P - PROGRAMMER REAL TIME CLOCK BR6 | |
| 110 | | |
| 114 | | |
| 120 | XY PLOTTER | |
| 124 | DR11B-(BR5 HARDWIRED) | |
| 130 | ADO1 BR5-(BR7 HARDWIRED) | |
| 134 | AFC11 FLYING CAP MULTIPLEXER BR4 | |
| 140 | AA11-A,B,C SCOPE BR4 | |
| 144 | AA11 LIGHT PIN BR5 | |
| 150 | · | |
| 154 | · | |
| 160 | · | |
| 164 | · | |
| 170 | USER RESERVED | |
| 174 | USER RESERVED | |
| 200 | LP11 LINE PRINTER CTRL-BR4 | |
| 204 | RF11 DISK CTRL-BR5 | |
| 210 | RC11 DISK CTRL-BR5 | |
| 214 | TC11 DEC TAPE CTRL-BR6 | |
| 220 | RK11 DISK CTRL-BR5 | |
| 224 | TM11 COMPATIBLE MAG TAPE CTRL-BR5 | |
| 230 | CR11/CM11 CARD READER CTRL-BR6 | |
| 234 | UDC11 (BR4, BR6 HARDWIRED) | |
| 240 | 11/45 PIRQ | |
| 244 | FPU ERROR | |
| 250 | | |
| 254 | RP11 DISK PACK CTRL-BR5 | |

|  |  |
|---|---|
| 260 |  |
| 264 |  |
| 270 | USER RESERVED |
| 274 | USER RESERVED |
| 300 | START OF FLOATING VECTORS--BR5 |
| 304 | STARTING AT 300 ALL DC11'S (BR5), THEN ALL KL11'S (BR4), THEN DP11'S (BR5) THEN DM11 (BR5), DN11 (BR5), AND DM11BB, DR11A, TYPE SET READERS, TYPE SET PUNCHES, DT11 (BR7) (DS11 VECTOR IS AT 1000) |
| 500 | FACTORY BUS TESTERS |
| . |  |
| . |  |
| . |  |
| . |  |
| . |  |
| . |  |
| . |  |
| 546 |  |

DEVICE ADDRESS

NOTE: XX MEANS A RESERVED ADDRESS FOR THAT OPTION. OPTION MAY NOT USE IT BUT IT WILL RESPOND TO BUS ADDRESS.

|  |  |
|---|---|
| 777776 | CPU STATUS |
| 777774 | 11/45 STACK LIMIT REGISTER |
| 777772 | 11/45 PIRQ REGISTER |
| 777716 | TO 777700 CPU REGISTERS |
| 777676 | TO 777600 11/45 SEGMENTATION REGISTER |
| 777656 | TO 777650 MX11 #6 |
| 777646 | TO 777640 MX11 #5 |
| 777636 | TO 777630 MX11 #4 |
| 777626 | TO 777620 MX11 #3 |
| 777616 | TO 777610 MX11 #2 |
| 777606 | TO 777600 MX11 #1 |
| 777576 | 11/45SSR2 |
| 777574 | 11/45 SSR1 |

| | | |
|---|---|---|
| 777572 | 11/45 SSR0 | |
| 777570 | CONSOLE SWITCH REGISTER | |
| 777566 | KL11 TTY OUT DBR | |
| 777564 | KL11 TTY IN CSR | |
| 777562 | KL11 TTY IN DBR | |
| 777560 | KL11 TTY OUT CSR | |
| 777556 | PC11 HSP DBR | |
| 777554 | PC11 HSP CSR | |
| 777552 | PC11 HSR DBR | |
| 777550 | PC11 HSR CSR | |
| 777546 | LKS LINE CLOCK KW11-L | |
| | | |
| 777526 | DR11A-XX-- | |
| 777524 | SEE 767776 | |
| 777522 | DR11A DBR | |
| 777520 | DR11A CSR | |
| 777516 | LP11 DBR | |
| 777514 | LP11 CSR | |
| 777512 | LP11 XX | |
| 777510 | LP11 XX | |
| 777506 | | |
| 777504 | | |
| 777502 | | |
| 777500 | | |
| | | |
| 777476 | RF11 DISK RFLA | LOOK AHEAD |
| 777474 | RF11 DISK RFMR | MAINTENANCE |
| 777472 | RF11 DISK RFDBR | |
| 777470 | RF11 DISK RFDAE | |
| 777466 | RF11 DISK RFDAR | |
| 777464 | RF11 DISK RFCAR | |
| 777462 | RF11 DISK RFWC | |
| 777460 | RF11 DISK RFDSC | |
| | | |
| 777456 | RC11 DISK RCDBR | |
| 777454 | RC11 REMAINTENANCE | |
| 777452 | RC11 RCCAR | |
| 777450 | RC11 RCWC | |
| 777446 | RC11 RCCSR1 | |
| 777444 | RC11 RCCSR1 | |
| 777442 | RC11 RCDAR | |
| 777440 | RC11 RCLA | |
| | | |
| 777434 | DT11 BUS SWITCH #7 | |
| 777432 | BUS SWITCH #6 | |
| 777430 | BUS SWITCH #5 | |
| 777426 | BUS SWITCH #4 | |
| 777424 | BUS SWITCH #3 | |
| 777422 | BUS SWITCH #2 | |
| 777420 | BUS SWITCH #1 | |
| | | |
| 777416 | RKDB | RK11 DISK |
| 777414 | RKMR | |
| 777412 | RKDA | |

| | | |
|---|---|---|
| 777410 | RKBA | |
| 777406 | RKWC | |
| 777404 | RKCS | |
| 777402 | RKER | |
| 777400 | RKDS | |

| | | |
|---|---|---|
| 777356 | TCXX | |
| 777354 | TCXX | |
| 777352 | TCXX | |

| | | |
|---|---|---|
| 777350 | TCDT | DEC TAPE (TC11) |
| 777346 | TCBA | |
| 777344 | TCWC | |
| 777342 | TCCW | |
| 777340 | TCST | |

| | | |
|---|---|---|
| 777336 | ASH | EAE (KE11-A) #2 |
| 777334 | LSH | |
| 777332 | NOR | |
| 777330 | SC | |
| 777326 | MUL | |
| 777324 | MQ | |
| 777322 | AC | |
| 777300 | DIV | |

| | | |
|---|---|---|
| 777316 | ASH | EAE (KE11-A) #1 |
| 777314 | LSH | |
| 777312 | NOR | |
| 777310 | SC | |
| 777306 | MUL | |
| 777304 | MQ | |
| 777302 | AC | |
| 777300 | DIV | |

| | | |
|---|---|---|
| 777166 | CR11 XX | |
| 777164 | CRDBR2 | CR11/CM11 CARD READER |
| 777162 | CRDBR1 | |
| 777160 | CRCSR | |

| | | |
|---|---|---|
| 776776 | AD01-D XX | |
| 776774 | AD01-D XX | |
| 776772 | ADDBR | A/D CONVERTER AD01-D |
| 776770 | ADCSR | |

| | | |
|---|---|---|
| 776766 | DAC3 | DAC AA11 |
| 776764 | DAC2 | |
| 776762 | DAC1 | |
| 776760 | DAC0 | |
| 776756 | SCOPE CONTROL - CSR | |
| 776754 | AA11 XX | |
| 776752 | AA11 XX | |
| 776750 | AA11 XX | |

| | | |
|---|---|---|
| 776740 | RPBR3 | RP11 DISK |
| 776736 | RPBR2 | |
| 776734 | RPBR1 | |
| 776732 | MAINTENANCE #3 | |
| 776730 | MAINTENANCE #2 | |
| 776726 | MAINTENANCE #1 | |
| 776724 | RPDA | |
| 776222 | RPCA | |
| 776720 | RPBA | |
| 776716 | RPWC | |
| 776714 | RPCS | |
| 776712 | RPER | |
| 776710 | RPDS | |

776676 TO 776500 MULTI TTY FIRST STARTS AT 776500

776476 TO 776406 MULTIPLE AA11'S SECOND STARTS @ 776760
776476 TO 776460 5TH AA11
776456 TO 776440 4TH AA11
776436 TO 776420 3RD AA11
776416 TO 776400 2ND AA11
NOTE 1ST AA11 IS AT 776750

776377 TO 776200 DX11
775600 DS11 AUXILIARY LOCATION
775577 TO 775540 DS11 MUX3
775537 TO 775500 DS11 MUX2
775477 TO 775440 DS11 MUX1
775436 TO 775400 DS11 MUX0
775377 TO 775200 DN11
775177 TO 775000 DM11
774777 TO 774400 DP11/DC11
774377 TO 774000 DC11/DP11

773777 TO 773000 DIODE MEMORY MATRIX

773000 BM792-YA PAPER TAPE BOOTSTRAP
773100 BM792-YB RC,RK,RP,RF AND TC11 - BOOTSTRAP
773200
773300
773400
773500
773600
773700 RESERVED FOR MAINTENANCE LOADER

772776 TO 772700 TYPESET PUNCH
772676 TO 772600 TYPESET READER

| | |
|---|---|
| 772576 | AFC-MAINTENANCE |
| 772574 | AFC-MUX ADDRESS |
| 772572 | AFC-DBR |
| 772570 | AFC-CSR |

| | |
|---|---|
| 772546 | KW11P XX |
| 772544 | KW11P COUNTER |
| 772542 | KW11P COUNT SET BUFFER |
| 772540 | KW11P CSR |
| 772536 | TM11 XX |
| 772534 | TM11 XX |
| 772532 | TM11 LRC |
| 772530 | TM11 DBR |
| 772526 | TM11 BUS ADDRESS |
| 772524 | TM11 BYTE COUNT |
| 772522 | TM11 CONTROL |
| 772520 | TM11 STATUS |
| 772512 | OST CSR |
| 772510 | OST EADRS1,2 |
| 772506 | OST ADRS2 |
| 772504 | OST ADRS1 |
| 772502 | OST MASK2 |
| 772500 | OST MASK1 |
| 772476 | DR11B DBR4 |
| 772474 | DR11B CSR4 |
| 772472 | DR11B BA4 |
| 772470 | DR11B WC4 |
| 772466 | |
| 772462 | |
| 772460 | |
| 772456 | DR11B DBR3 |
| 772454 | DR11B CSR3 |
| 772450 | DR11B BA3 |
| 772450 | DR11B WC3 |
| 772446 | |
| 772444 | |
| 772442 | |
| 772440 | |
| 772436 | DR11B DBR2 |
| 772434 | DR11B CSR2 |
| 772432 | DR11B BA2 |
| 772430 | DR11B WC2 |
| 772426 | |
| 772424 | |
| 772422 | |
| 772420 | |
| 772416 | DR11B/DATA |
| 772414 | DR11B/STATUS |
| 772412 | DR11B/BA |
| 772410 | DR11B/WC |
| 772146 TO 772110 MEMORY PARITY CSR | |
| 772146 | 15 |
| 772120 | 4 |
| 772116 | 3 |
| 772114 | 2 |
| 772112 | 1 |
| 772110 | 0 |
| 771776 | UDCS - CONTROL AND STATUS REGISTER |

| | |
|---|---|
| 771774 | UDSR - SCAN REGISTER |
| 771772 | UDCM - MAINTENANCE REGISTER |
| 771766 | UDC FUNCTIONAL I/O MODULES |
| 771000 | UDC FUNCTIONAL I/O MODULES |
| 770776 TO 770700 KG11 CRC OPTION | |
| 770776 | KG11A KGNU7 |
| 770774 | KGBCC7 |
| 770772 | KGDBR7 |
| 770770 | KGCSR7 |
| 770716 | KGNU4 |
| 770714 | KGBCC3 |
| 770712 | KGDBR2 |
| 770710 | KGCSR1 |
| 770706 | KGNU0 |
| 770704 | KGBCC0 |
| 770702 | KGDBR0 |
| 770700 | KG11A KGCSR0 |
| 770676 TO 770500 16 LINE FOR DM11BB | |
| 770676 | DM11BB #16 |
| 770674 | |
| 770672 | |
| 770670 | |
| 770666 | DM11BB #15 |
| 770664 | |
| 770662 | |
| 770660 | |
| 770656 | DM11BB #14 |
| 770654 | |
| 770652 | |
| 770650 | |
| 770646 | DM11BB #13 |
| 770644 | |
| 770642 | |
| 770640 | |
| 770636 | DM11BB #12 |
| 770634 | |
| 770632 | |
| 770630 | |
| 770626 | DM11BB #11 |
| 770624 | |
| 770622 | |
| 770620 | |
| 770616 | DM11BB #10 |
| 770614 | |
| 770612 | |
| 770610 | |
| 770606 | DM11BB #9 |
| 770604 | |
| 770602 | |
| 770600 | DM11BB #8 |
| 770076 | LATENCY TESTER |
| 770074 | LATENCY TESTER |
| 770072 | LATENCY TESTER |

770070     LATENCY TESTER
770056 TO 770000 SPECIAL FACTORY BUS TESTERS
767776 TO 764000 FOR USER and SPECIAL SYSTEMS---DR11A ASSIGNED IN USER
    AREA-STARTING AT HIGHEST ADDRESS WORKING DOWN

| | |
|---|---|
| 767776 | DR11A #0 |
| 767774 | |
| 767772 | |
| 767770 | |
| 767766 | DR11A #1 |
| 767764 | |
| 767762 | |
| 767760 | |
| 767756 | DR11A #2 |
| 767754 | |
| 767752 | |
| 767750 | |

764000     START NORMAL USER ADDRESSES HERE AND ASSIGN UPWARD.
760004 TO 760000 RESERVED FOR DIAGNOSTIC - SHOULD NOT BE ASSIGNED

# APPENDIX C - INSTRUCTION SET PROCESSOR

A DESCRIPTION OF THE PDP-11
USING THE INSTRUCTION SET PROCESSOR (ISP) NOTATION[1]

ISP is a language (or notation) which can be used to define the action of a computer's instruction set. It defines a computer, including cônsole and peripherals, as seen by a programmer. It has two goals: to be precise enough to constitute the complete specification for a computer and to still be highly readable by a human user for purposes of reference, such as this manual. The main part of the manual contained an English language description of the PDP-11, using ISP expressions as support in defining each instruction. This appendix contains an ISP description of the PDP-11, using a few English language comments as support.

The following brief introduction to the notation is given using examples from the PDP-11 Model 20 ISP description. The complete PDP-11 description follows the introduction.

A processor is completely defined at the programming level by giving its instruction set and its interpreter in terms of basic operations, data types and the system's memory. For clarity the ISP description is usually given in a fixed order:

Declare the system's memory:

> <u>Processor state</u> (the information necessary to restart the processor if stopped between instructions, e.g., general registers, PC, index registers)
>
> <u>Primary memory state</u> (the memory directly addressable from the processor)
>
> <u>Console state</u> (any external keys, switches, lights, etc., that affect the interpretation process)
>
> <u>Secondary memory</u> (the disks, drums, dectapes, magnetic tapes, etc.)
>
> <u>Transducer state</u> (memory available in any peripheral devices that is assumed in the instructions of the processor)

Declare the instruction format
Define the operand address calculation process
Declare the data types
Declare the operations on the data types
Define the instruction interpretation process including interrupts, traps, etc.
Define the instruction set and the instruction execution process (provides an ISP expression for each instruction)

Thus, the computer system is described by first declaring memory, data-types and primitive data operations. The instruction interpreter and the instruction-set is then defined in terms of these entities.

The ISP notation is similar to that used in higher level programming languages. Its statements define entities by means of expressions involving other entities in the system. For example, an instruction to increment (add-one) to memory would be

```
Increment := (M[x] ← M[x] + 1);                add one to memory, x
```

This defines an operation, called "increment", that takes the contents of memory M at an address, x, and replaces it with a value one higher. The := symbol simply assigns a name (on the left) to stand for the expression (on the right). English language comments are given in italics. Table 1 gives a reference list of notations, which are illustrated below.

ISP expressions are inherently interpreted in parallel, reflecting the underlying parallel nature of hardware operations. This is an important difference between ISP and standard programming languages, which are inherently serial. For example, in

[1] The notation derived and used in the book, <u>Computer Structures: Readings and Examples</u>, McGraw-Hill, 1971 by C. Gordon Bell and Allen Newell. The book contains ISP's of 14 computers.

Z := (M[x] ← S'+D'; M[y] ← M[x]);

both righthand sides of the data transmission operator (←) are evaluated in the current memory state in parallel and then transmission occurs. Thus the old value of M[x] would go into M[y]. Serial ordering of processing is indicated by using the term "next". For example,

Z := (M[x] ← S'+D'; next M[y] ← M[x]);

performs the righthand data transmission after the lefthand one. Thus, the new value of M[x] would be used for M[y] in this latter case.

## Memory Declarations

Memory is defined by giving a memory declaration as shown in Table 1. For example,

$$\text{Mp}[0:2^k - 1]\langle 15:0\rangle$$

declares a memory named, Mp, of $2^k$ words (where k has been given a value). The addresses of the words in memory are $0,1,\ldots,2^k$-1. Each word has 16 bits and the bits are labeled 15,14,...,0. Some other examples of memory declarations are:

| | |
|---|---|
| $\left.\begin{array}{l}\text{Boundary-error}_2\\ \text{Boundary-error}^2\end{array}\right\}$ | *boolean memories; scalar bit alternatives* |
| $\text{Activity}_3$ | *ternary digit, holding value 0,1, or 2* |
| N/Negative | *alias, N and Negative are synonomous* |
| CC<3> | *bit 3 of a register* |
| $M[0:2^{18}-1]\langle 7:0\rangle$ | *vector of $2^{18}$ 8-bit words* |
| M[0:15][0:4095]<7:0> | *array of 16 x 4096 8-bit words* |
| $\left.\begin{array}{l}\text{brop}\langle 1:0\rangle_{16}\\ \text{brop}\langle 7:0\rangle_2\end{array}\right\}$ | *alternative ways of defining a register using base 16 and base 2* |

## Renaming and Restructuring of Previously Defined Registers

Registers can be defined in terms of existing registers. In effect, each time the name to the left of the := symbol is encountered, the value is computed according to the expression to the right of :=. A process can be evoked to form the value and side-effects are possible when the value is computed.

### Examples of simple renaming in part or whole of existing memory

| | |
|---|---|
| N/Negative := CC<3> | *N is name of bit 3 of register CC* |
| SP<15:0> := R[6]<15:0> | *SP is the same as register R[6]* |

### Examples of register formed by concatenation

```
LAC<L,0:11> := L□AC<0:11>
AB<0:47> := A<0:23>□B<0:23>
Mword[0]<15:0> := Mbyte[0]<7:0>□Mbyte[1]<7:0>
```

### Examples of values and registers formed by evaluation of a process

```
ai/address-increment<1:0> := (     value of ai is 2 if ¬ byte op,
     ¬ byte-op ⇒ 2;                 else value is 1
       byte-op ⇒ 1)
Run := (Activity = 0)               Run=1 or 0 depending on value of Activity
                                     being 0 or not 0
```

## Instruction Format

Instruction formats are declared in the same fashion as memory and are not distinguishable as special non-memory entities. The instructions are carried in a register; thus it is natural to declare them by giving names to the various parts of the instruction register. Usually only a single declaration is made, the instruction/i, followed by the declarations of the parts of the instruction; the operation code, the address fields, indirect bit, etc.

### Example

This declaration would correspond to the usual box diagram:

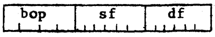
Table 1. ISP Character-Set and Expression Forms

| | |
|---|---|
| A,...,Z,a,...,z,.,-,␣, ,',",0,...,9 | name alphabet. This character set is used for names. |
| | comments. Italics are used for comments. |
| $\underbrace{M[a:b]\ \dots\ [v:w]}_{n}\langle x:y\rangle_z$ | memory declaration. An n-dimensional memory array of words where a:b ... v:w are the range of values for the first and last dimensions. The values of the first dimension are, for example, a, a+1, ..., b for a ≤ b (or a,a-1,...,b for a > b). The word length base, z, is normally 2 if not specified. The digits of the word are x,x+1,...y. |
| a := f(expression) | definition. The operator, :=, defines memory, names, process, or operations in terms of existing memory and operations. Each occurrence of "a" causes the in place substitution by f(expression). |
| b(c,...,e) := g(expression) | The definition b, may have dummy parameters, c,...,e, which are used in g(expression). |
| name' := h(expression) | side effects naming convention. In this description we have used ' to indicate that a reference to this name will cause other registers to change. |
| a ← f(expression)<br>f(expression) → a | transmission operator. The contents in register a are replaced by the value of the function. |
| ( ) | parentheses. Defines precedence and range of various operations and definitions (roughly equivalent to begin, and end). |
| {data-type} | operator and data-type modifier |
| boolean ⇒ expression; | conditional expression; equivalent to ALGOL <u>if</u> boolean <u>then</u> expression |
| boolean ⇒ (expression-1 else expression-2); | equivalent to Algol <u>if</u> boolean <u>then</u> expression-1 <u>else</u> expression-2 |
| ; next | sequential delimiter interpretation is to occur |
| □ | concatenation. Consider the registers to the left and right of □ to be one. |
| ; | statement delimiter. Separates statements. |
| , | item delimiter. Separates lists of variables. |
| a/b | division and synonym. Used in two contexts: for division and for defining the name, a, to be an alias (synonym) of the name, b. |
| ? | unknown or unspecified value |
| Φ | set value. Takes on all values for a digit of the given base, e.g., $1\Phi_2$ specifies either $10_2$ or $11_2$ |
| X(:= boolean) ⇒ expression; | instruction value definition. The name X is defined to have the value of the boolean. When the boolean is true, the expression will be evaluated. |

Table I. cont'd.

Common Arithmetic, Logical and Relational Operators

| Arithmetic | Logical | Relational |
|---|---|---|
| + add | ¬ not | ≡ identical |
| - subtract, also negative | ∧ and | ≢ not identical |
| × multiply | ∨ or | = equal |
| / divide | ⊕ exclusive-or | ≠ not equal |
| mod modulo (remainder) | ≡ equivalence | > greater than |
| $( )^2$ squared | | ≥ greater than or equal |
| $( )^a$ exponentiation | | < less than |
| ( )↑a exponentiation | | ≤ less than or equal |
| $( )_b$ base | | |
| ( )↓b base | | |
| sqrt( ) square root | | |
| abs( ) absolute value | | |
| sign-extend( ) | | |



```
i/instruction<15:0>             the instruction
   bop<3:0> := i<15:12>         specifies binary (dyadic) operations
   sf<5:0>  := i<11:6>          specifies source (first) operand
   df<5:0>  := i<5:0>           specifies second operand and destination
```

Operand Address Calculation Process

In all processors, instructions make use of operands. In most conventional processors, the operand is usually in memory or in the processor, defined as M[z], where z is the effective address. In PDP-11, a destination address, Daddress, is used in this fashion for only two instructions. It is defined in ISP by giving the process that calculates it. This process may involve only accesses to primary memory (possibly indexed), but it may also involve side effects, i.e., the modification of either of primary memory or processor memory (e.g., by incrementing a register). Note that the effective address is calculated whenever its name is encountered in evaluating an ISP expression (either in an instruction or in the interpretation expression). That is, it is evaluated on demand. Consequently, any side effects may be executed more than once.

Operation Determination Processes

Instead of effective-address, the operands are usually determined directly. For example, the 16-bit destination register is just the register selected by the dr field of an instruction, i.e.,

```
         Rd := R[dr]                the destination register
```

In one other case, the operand is just the next word following an instruction. This next word can be defined,

```
   nw'<15:0>/next-word := (Mw[PC]; PC ← PC + 2)  the next word is selected and PC is moved
```

Here, the ' shows that a reference to nw will cause side effects, in this case, PC ← PC + 2. For calculating the source operand, S, the process is:

```
         S'<15:0> := (              value for source operand
           (sm=0) ⇒ R[sr];          if mode=0 then S' is the Register addressed
                                     by instruction field sr
           (sm=1) ⇒ Mw[R[sr]]       if mode=1 the S' is indirect via R sr
           (sm=2) ∧ (sr=7) ⇒ nw;    if mode=2 and source register=PC then the
                                     next word is the operand; this can be
                                     seen by substituting the expression for nw'
```

An expression is also needed for the operand, S, which does not cause the side effects, and assuming the effects have taken place, counteracts them. Thus, S would be:

```
        S<15:0> := (
            (sm=0) ⇒ R[sr];                  no side effects
            (sm=1) ⇒ Mw[R[sr]];              no side effects
            (sm=2) ∧ (sr=7) ⇒ Mw[PC-2]       counteract previous side effects
                 :
```

In the ISP description a general process is given which determines operands for Source-Destination, word-byte, and with-without side-effects. In order to clarify what really happens, the source operand calculation, for words, with side effects, is given below.

```
Sf<5:0> := i<11:6>                          source field (6-bits) of instruction
    sm$_8$ := sf<5:3>                       source mode control field
    sd  := sf<3>                            deferred address control
    sr$_8$ := sf<2:0>                       register specification for source

nw'<15:0> := (Mw[PC]; PC ← PC+2)            next word; used as operand
Rs<15:0>  := R[sr]                          source register specification

S'<15:0>/Source := ((                       value for the source--direct addressing
    (sm=0) ⇒ Rs;                            use the register Rs as operand
    (sm=2) ∧ (sr≠7) ⇒ (Mw[Rs]               direct auto-increment (increment
       Rs ← Rs + 2);                         Rs); usually used as POP
    (sm=2) ∧ (sr=7) ⇒ nw;                   direct; actually immediate operand
    (sm=4) ⇒ (Rs ← Rs - 2; next             direct; auto-decrement (decrement
       Mw[Rs]);                              Rs); usually used as PUSH
    (sm=6) ∧ (sr≠7) ⇒ Mw[nw' + Rs];         direct; indexed via Rs--uses next-word
    (sm=6) ∧ (sr=7) ⇒ Mw[nw' + PC];         direct; relative to PC; uses next-word
                                             value for the source-defined addressing
    (sm=1) ⇒ Mw[Rs];                        defer through Rs
    (sm=3) ∧ (sr≠7) ⇒ (Mw[Mw[Rs]];          defer through stack; auto
       Rs ← Rs + 2);                         increment
    (sm=3) ∧ (sr=7) ⇒ M[nw'];               defer via next word; absolute addressing
    (sm=5) ⇒ (Rs ← Rs - 2; next             defer through stack after auto
       Mw[Mw[Rs]]);                          decrement
    (sm=7) ∧ (sr≠7) ⇒ Mw[Mw[nw' + Rs]];     defer, indexed via Rs
    (sm=7) ∧ (sr=7) ⇒ Mw[Mw[nw' + PC]]      defer relative to PC
                   );                       end calculation process;
    (sr=6) ∧ ((sm=4) ∨ (sm=5)) ∧            checks if stack overflowed for several
      (SP<400$_8$) ⇒ (Stack overflow ← 1)     modes
                   )                        end source calculation
```

**Data-Types**

A data-type specifies the encoding of a meaning into an information medium. The meaning of the data-type (what it designates or refers to) is called its referent (or value). The referent may be anything ranging from highly abstract (the uninterpreted bit) to highly concrete (the payroll account for a specific type of employee).

Every data-type has a carrier, into which all its component data-types can be mapped. The carrier is used in storing the data-type in memories and is usually a word or multiple thereof. It must be extensive enough to hold all the component data-types, but may be a larger (having error checking and correcting bits, or even unused bits). The mapping of the component data-types into the carrier is called the format. It is given as a list which associates to each component an expression involving the carrier (e.g., as in the instruction format).

ISP provides a way of naming data-types, which also serves as a basis for abbreviations. Some data-types simply have conventional names (e.g., character/ch, floating point numbers/f); others are named by their value (e.g., integer/i). Data-types which are iterates of a basic component can be named by the component suffixed by a length-type. The length-type can be array/a, implying a multi-dimensional array of fixed, but unspecified dimensions; a string/st, implying a single sequence, of variable length (on each occurrence); or a vector/v, implying a one dimensional array of a fixed but unspecified number of components. The length-type need not exist, and then this form of the name is not applicable. Thus, iv is the abbreviation for an integer vector. It is also possible to name a data-type by simply listing its components.

Data-types are often of a given precision and it has become customary to measure this in terms of the number of components that are used, e.g., triple precision integers. In ISP this is indicated by prefixing the precision symbol to the basic data-type name, e.g., di for double precision integer. Note that a double precision integer, while taking two words, is not the same thing as a two integer vector, so that the precision and the length-type, though both implying something about the size of the carrier, do not express the same thing.

A list of common data-types and their abbreviations is given in Table 2.

## Operations on Data-types

Operations produce results of specific data-types from operands of specific data-types. The data-types themselves determine by and large the possible operations that apply to them. No attempt will be made to define the various operations here, as they are all familiar. A reasonably comprehensive list is given in Table 1. An operation-modifier, enclosed in braces, { }, can be used to distinguish variant operations. The operation-modifier is usually the name of a data-type, e.g., A+B{f} is a floating point addition. Modifiers can also be a description name applying to the operation, e.g., a × 2 {rotate}.

New operations can be defined by means of forms. For example, the various add operations on differing data-types are specified by writing {data-type} after the operation.

## Instruction Interpretation Process

The instruction interpretation expression and the instruction set constitute a single ISP expression that defines the processor's action. In effect, this single expression is evaluated and all the other parts of the ISP description of a processor are evoked as indirect consequences of this evaluation. Simple interpreter without interrupt facilities show the familiar cycle of fetch-the-instruction and execute-the instruction.

Example:

```
        Run ⇒ (instruction ← M[PC]; PC ← PC + 1; next   This is a simple
               Instruction-execution; next)               interpreter, not the
                                                          one for the PDP-11
```

In more complex processors the conditions for trapping and interrupting must also be dexcribed. The effective address calculation may also be carried out in the interpreter, prior to executing the instruction, especially if it is to be calculated only once and will have a fixed value independent of anything that happens while executing instructions. Console activity can also be described in the interpreter, e.g., the effect of a switch that permits stepping through the program under manual control, or interrogating and changing memory.

The normal statement for PDP-11 interpretation is just:

```
¬ Interrupt-rq ∧ Run ⇒ (instruction ← Mw[PC]; PC ← PC + 2; next   fetch
     Instruction-execution; next                                   execute
     T-flag ⇒ (State-change(14₈); T-flag ← 0))                     trace mode
```

Table 2. Common Data-Types Abbreviations

| Primitive | | String and Vector | |
|---|---|---|---|
| b | bit or boolean | bv | bit.vector |
| by | byte | by.st | byte.string |
| ch | character | ch.st | character.string |
| cx | complex | | |
| df | double precision floating | | |
| dw | double word | | |
| d | digit | jd | j-digit number |
| f | floating | | |
| fr | fraction | | |
| hw | half word | | |
| i | integer | | |
| mx | mixed number | | |
| qw | quadruple length word | | |
| tw | triple length word | | |
| w | word | | |

## Instruction-Set and Instruction Execution Process

The instruction set and the process by which each instruction is executed are usually given together in a single definition; this process is called Instruction-execution in most ISP descriptions. This usually includes the definition of the conditions for execution, i.e., the operation code, value, the name of the instruction, a mnemonic alias, and the process for its execution. Thus, an individual instruction typically has the form:

```
MOV (:= bop = 0001₂) ⇒ (                          move word
   r ← S'; next                                   move source to intermediate register
      N ← r<15>;                                  negative?
      (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);         zero?
      V ← 0;                                      overflow cleared
      D ← r);                                     transmit result to destination
```

With this format for the instruction, the entire instruction set is simply a list of all the instructions. On any particular execution, as evoked by the interpretation expression, typically one and only one operation code correlation will be satisfied, hence one and only one instruction will be executed.

In the case of PDP-11, the text carries the definition of the individual instructions, hence they are not redefined in the appendix. Instead, the appendix defines the condition for executing the instructions. For example,

MOV := (bop = $0001_2$)

is given in the appendix, and the action of MOV is defined (in ISP) in the text.

## THE PDP-11 ISP

*PDP-11's Primary (Program) Memory and Processor State*

*The declaration of this memory includes all the state (bits, words, etc.) that a program (programmer) has access to in this part of the computer. The console is not included. The various secondary memories (e.g., disks, tapes) and input-output device state declarations are included in a following section.*

*Primary (program) Memory*

| | |
|---|---|
| Mp[0:$2^k$-1]<15:0> | *actual physical, 16-bit memory of a particular system; k = 12, ..., 17* |
| Mw/Mword[x<15:0>]<15:0> := ( | *word-accessed memory* |
| ¬ x<0> ⇒ Mp[x<15:1>]; | *word on even byte boundary, all right* |
| x<0> ⇒ (? *value* ; Boundary-error ← 1)) | *word on odd byte boundary, trap* |
| Mb/Mbyte[x<15:0>]<7:0> := ( | *byte-accessed memory* |
| ¬ x<0> ⇒ Mp[x<15:1>]<7:0>; | *take low-order bits if even* |
| x<0> ⇒ Mp[x<15:1>]<15:8>) | *take hi-order bits if odd* |

*Processor State*

| | |
|---|---|
| R[0:7]<15:0> | *eight, 16-bit General-Registers, used for accumulators, indexing and stacks* |
| SP<15:0>/Stack-Pointer := R[6] | *special stack, controlled by R[6]* |
| PC<15:0>/Program-Counter := R[7] | *location next instruction, also R[7]* |
| PS<15:0>/Processor-State-Word | *16-bit register giving rest of state* |
| Unused<7:0>/Undefined := PS<15:8> | *mapping of bits into PS* |
| P<2:0>/Priority := PS<7:5> | *interrupt level control of processor* |
| T/Trace := PS<4> | *denotes whether trap is to occur after each instruction* |
| CC<3:0>/Condition-Codes := PS<3:0> | *set as a function of instruction and results* |
| N/Negative := CC<3> | *if result = -* |
| Z/Zero := CC<2> | *if result = 0* |
| V/Overflow := CC<1> | *if result overflows* |
| C/Carry := CC<0> | *if result carried into/borrowed from most significant bit* |

*Processor-Controlled Error Flags (resulting from instruction-execution)*

| | |
|---|---|
| **Boundary-Error** | *set if word is accessed on odd byte boundary* |
| **Stack-Overflow** | *set if word accessed, via SP < $400_8$* |
| **Time-Out-Error** | *set if non-existent memory or device is referenced* |
| **Illegal-Instruction** | *set if a particular class of instructions is executed* |

*Processor-activity*

| | |
|---|---|
| **Activity$_3$** | *ternary, specifying state of processor* |
| **Run := (Activity = 0)** | *normal instruction interpretation* |
| **Wait := (Activity = 1)** | *waiting for interrupt* |
| **Off := (Activity = 2)** | *off, dormant* |

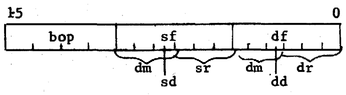
*Error-Flags (resulting from without the processor)*

| | |
|---|---|
| **Power-Fail-Flag** | *set if power is low* |
| **Power-Up-Flag** | *set when power comes on* |

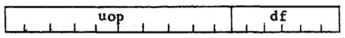
*Instruction format field declarations*

i<15:0>/instruction

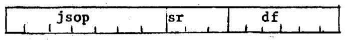
| | | |
|---|---|---|
| bop<3:0> | := i<15:12> | *binary opcode format* |
| sf<5:0> | := i<11:6> | *source field* |
| $sm_8$ | := sf<5:3> | *source mode - 3 bits* |
| sd | := sf<3> | *source defer bit* |
| $sr_8$ | := sf<2:0> | *source register - 3 bits* |
| df<5:0> | := i<5:0> | *destination field* |
| $dm_8$ | := df<5:3> | *destination mode - 3 bits* |
| dd | := df<3> | *destination defer bit* |
| $dr_8$ | := df<2:0> | *destination register - 3 bits* |
| uop<3:0>$_8$ | := i<15:6> | *unary op code (arith., logical, shifts)* |
| *df* | | *see binary op format* |
| jsop<7:0> | := i<15:9> | *jsr format* |
| *sr; df* | | *see binary op format* |
| brop<1:0>$_{16}$ | := i<15:8> | *branch format* |
| offset<7:0> | := sign-extend(i<7:0>) | *offset value* |
| trop<1:0>$_{16}$ | := i<15:8> | *trap format* |
| unused-trop<1:0>$_{16}$ | := i<7:0> | |
| eop<6:0> | := i<15:9> | *extended opcode format* |
| er<3:0> | := i<8:6> | *extended register* |
| esf<5:0> | := i<5:0> | *extended source field* |
| $esm_8$ | := esf<5:3> | *mode* |
| esd | := esf<3> | *defer* |
| $esr_8$ | := esf<2:0> | *register* |
| fop<7:0> | := i<15:8> | *floating op format* |
| fr<7:0> | := i<7:6> | *register destination* |
| fsf<5:0> | := i<5:0> | *source* |



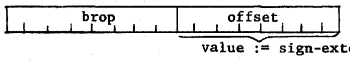
binary operand (2 operands) format



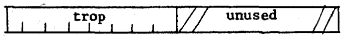
unary operand (1 operand), JMP format



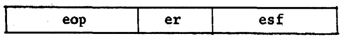
JSR format



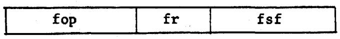
branch format



trap format



extended operation format



floating op format

```
ai/address-increment<1:0> := (
   ¬ Byte-op ⇒ 2;
     Byte-op ⇒ 1)
Byte-op := (MOVB ∨ BICB ∨ BISB ∨ BITB ∨ CLRB ∨
            COMB ∨ INCB ∨ DECB ∨ NEGB ∨ ADCB ∨
            SBCB ∨ TSTB ∨ RORB ∨ ROLB ∨ ASRR ∨
            ASLB ∨ SWAB)
Reserved-instruction := ((i =  ) ∨ (i =  ) ∨...∨(i =  ))  unused instructions
```

*Registers and Data Addressed via Instruction Format Specifications*

```
nw/next-word<15:0>   :=  Mw[PC]                    used in operand determination
nw'/next-word'<15:0>:=  (Mw[PC]; PC ← PC + 2)      with side effects
lw/last-word<15:0>   :=  Mw[PC - 2]                undoes side effects
Rs<15:0>  :=  R[sr]<15:0>                          the source register
Rd<15:0>  :=  R[dr]<15:0>                          the destination register
```

*Operand Determination for Source and Destination*
*Two types of operands are used: S', D', Sb' and Db' - for operands that cause side-effec (i.e., other registers are changed; and S, D, Sb and Db for operands that do not cause side effects. Two general procedures Wo' and Wo are used to determine these operands for side effects and no side effects, respectively*

```
S'<15:0>   :=  Oprd'<15:0>(Mw, 2,sm,sr)            source word operand side-effects
S<15:0>    :=  Oprd<15:0>(Mw, 2,sm,sv)             source word operands no side-effect
Sb'<7:0>   :=  Oprd'<7:0>(Mb, 2, sm,sr)            source byte
Sb<7:0>    :=  Oprd<7:0>(Mb, 1,sm,sr)
D'<15:0>   :=  Oprd'<15:0>(Mw, 2,dm,dr)            Destination operands
D<15:0>    :=  Oprd<15:0>(Mw, 2,dm,dr)
Db'<7:0>   :=  Oprd'<7:0>(Mb, 1, dm,dr)
Db<7:0>    :=  Oprd<7:0>(Mb, 1, dm,dr)
```

*General Operand Calculation Process (with Side Effects)*

```
Oprd'<wl:0>(M,ai,m,rg) := ((                        value for word or byte operand; dir
                                                     addressing: wl indicates length; 
                                                     mode, and rg register
   Rr<15:0> := R[rg]                                secondary definition for register
   (m=0) ⇒ Rr<wl:0>;                                0, use the register, Rr, as operand
   (m=2) ∧ (rg≠7) ⇒ (M[Rr]; next                    2, direct auto-increment (increment
      Rr ← Rr + ai);                                 Rr); usually used in pop stack
   (m=2) ∧ (rg=7) ⇒ nw'<wl:0>;                      2, direct; next-word is immediate
                                                     operand
   (m=4) ⇒ (Rr ← Rr - ai; next                      4, direct; after auto decrement
      M[Rr]);                                       usually used as PUSH stack
   (m=6) ∧ (rg≠7) ⇒ M[nw' + Rr];                    6, direct; indexed via Rr uses next
                                                     word
   (m=6) ∧ (rg=7) ⇒ M[nw' + PC];                    6, direct; relative to PC; uses nex
                                                     word value for word operand defer
                                                     addressing
   (m=1) ⇒ M[Rr];                                   1, defer through Rr
   (m=3) ∧ (rg≠7) ⇒ (M[Mw[Rr]]; next                3, defer through Mw[Rr] (usually st
      Rr ← Rr + 2);                                  auto-increment
   (m=3) ∧ (rg=7) ⇒ M[nw'];                         3, defer via next-word; absolute
                                                     addressing
   (m=5) ⇒ (Rr ← Rr - ai; next                      5, defer through stack after auto
      M[Mw[Rr]]);                                    decrement
```

```
    (m=7) ∧ (rg≠7) ⇒ M[Mw[nw' + Rr]];              7, defer indexed via Rr
    (m=7) ∧ (rg=7) ⇒ M[Mw[nw' + PC]];              7, defer relative to PC
                   );                              end calculation process
    (rg=6) ∧ ((m=4) ∨ (m=5)) ∧                     check if stack overflows
       (SP < $400_8$)) ⇒ (Stack-overflow ← 1)
                   )                               end operand calculation process
```

*General Operand Calculation Process (without Side Effects)*

```
Oprd<wl:0>(M,ai,m,rg) := (
    Rr<15:0>  :=  R[rg]
    (m=0) ⇒ Rr<wl:0>;
    (m=2) ∧ (rg≠7) ⇒ Mw[Rr - ai];                  undo previous side-effects
    (m=2) ∧ (rg=7) ⇒ lw<wl:0>;                     undo previous side-effects
    (m=4) ⇒ M[Rr];
    (m=6) ∧ (rg≠7) ⇒ M[lw + Rr];                   undo previous side-effects
    (m=6) ∧ (rg=7) ⇒ M[lw + PC];                   undo previous side-effects

    (m=1) ⇒ M[Rr];
    (m=3) ∧ (rg≠7) ⇒ M[Mw[Rr - 2]];                undo previous side-effects
    (m=3) ∧ (rg=7) ⇒ M[lw];                        undo previous side-effects
    (m=5) ⇒ M[Mw[Rr]];
    (m=7) ∧ (rg≠7) ⇒ M[Mw[lw + Rr]];               undo previous side-effects
    (m=7) ∧ (rg≠7) ⇒ M[Mw[lw + PC]])               undo previous side-effects
```

**Destination addresses for JMP and JSR**

```
Da<15:0> := ((                                     directs:
    (dm=0) ⇒ (?; Illegal-instruction ← 1);            illegal register address
    (dm=2) ∧ (dr≠7) ⇒ (Rd; Rd ← Rd + 2);             auto-increment
    (dm=2) ∧ (dr=7) ⇒ (PC; PC ← PC + 2);             null
    (dm=4) ⇒ (Rd ← Rd - 2; next Rd);                 auto-decrement
    (dm=6) ∧ (dr≠7) ⇒ (nw' + Rd);                    indexed
    (dm=6) ∧ (dr=7) ⇒ (nw' + PC);                    relative
                                                   defers:
    (dm=1) ⇒ Mw[Rd];                                  via register
    (dm=3) ∧ (dr≠7) ⇒ (Mw[Rd]; Rd ← Rd + 2);         via auto-increment
    (dm=3) ∧ (dr=7) ⇒ nw';                           absolute address
    (dm=5) ⇒ (Rd ← Rd - 2; next Mw[Rd]);             auto-decrement
    (dm=7) ∧ (dr≠7) ⇒ Mw[nw + Rd];                   via index
    (dm=7) ∧ (dr=7) ⇒ Mw[nw' + PC]); next            relative to PC

    (dr=6) ∧ ¬ ((dm=0) ∨ (dm=3) ∨ (dm=7)) ∧ (SP < $400_8$) ⇒ ( check for stack overflow
       stack-overflow ← 1))
```

*Data Type Formats*

```
by/byte<7:0>
w/word<15:0>
wi/word.integer<15:0>
bybv/byte.boolean-vector<7:0>
wbv/word.boolean-vector<15:0>
d/d.w/double.word<31:0>
```

```
f/d.f/double.word.floating<31:0>
   fs/floating.sign := f<31>
   fe/floating.exponent<7:0> := f<30:23>
   fm/floating.mantissa<22:0> := f<22:0>
t/triple.word<47:0>
q/quadruple.word<63:0>
qf/quadruple.word.floating-point<63:0>
   qfs := qf<63>
   qfe := qf<62:55>
   qfm := qf<54:0>
```

*I/O Devices and Interrupts, State Information*

```
Device[0:N-1]                                              N I/O devices - assume device J
   Device-name[J]<15:0> := J                               number to which device responses and
                                                            is addressed
   Device-interrupt-location[J]<15:0> := K                 each device has a value, K, which it
                                                            uses as an address to interrupt proces
   dob/device-output-buffer[J]<15:0>                       program controlled device data
   dib/device-input-buffer[J]<15:0>
   ds/device-status[J]<15:0>                               a register with device control state
      derr/device-error-flags[J]<3:0> := ds[J]<15:12>      common
      dbusy/device-busy[J] := ds[J]<11>                    status
      dunit/device-unit-selection[J]<2:0> := ds[J]<10:8>   assignments
      ddone[J] := ds[J]<7>
      denb/device-done-interrupt-enable := ds[J]<6>
      derrenb/device-error-interrupt-enable := ds[J]<5>
      dme/device-memory-extension[J]<4:3> := ds[J]<4:3>
      dfnc/device-function[J]<2:0> := ds[J]<2:0>
dintrq/device-interrupt-request[J] := (
   (ddone[J] ∧ denb[J] ∨ ((derr[J] ≠ 0) ∧ derrenb[J]))
   dil/device-interrupt-level[J]<7:4>                      each device is assigned to 1 of 4 level
```

*Mapping of Devices into M. Each device's registers are mapped into primary word memory, e.g., Teletype*

| | |
|---|---|
| M'[177560$_8$] := tks/ds[TTY-keyboard] | *keyboard status* |
| M'[177562$_8$] := tkb/dib[TTY-keyboard] | *keyboard input data* |
| M'[177564$_8$] := tps/ds[TTY-printer] | *teleprinter status* |
| M'[177566$_8$] := tpb/dob[TTY-printer] | *teleprinter data to print* |

*Interrupt Requests*

```
br/bus-request-for-interrupt<7:4> := (                     OR of all device requests
   (dintrq[0] ⇒ dil[0]) ∨
   (dintrq[1] ⇒ dil[1]) ∨...
   (dintrq[J] ⇒ dil[J]) ∨...
   (dintrq[N] ⇒ dil[N]))

Interrupt-rq := (intrql ≥ p)                               interrupt if a request is ≥ priority/P

intrql/interrupt-request-level<2:0> := (
   br<7> ⇒ 7;
   ¬ br<7> ∧ br<6> ⇒ 6;
   ¬ br<7> ∧ ¬ br<6> ∧ ¬ br<5> ∧ br<4> ⇒ 4)
```

*Instruction Interpretation Process*

```
Interrupt-rq ∧ Run ⇒ (Normal-interpretation);
   Normal-interpretation := (I ← Mw[PC]; PC ← PC + 2 next     fetch
      Instruction-execution; next                             execute
      T-flag ⇒ (State-change(14_8); T-flag ← 0))              trace
Interrupt-rq ∧ ¬ Off ⇒ (
   State-change(Device-interrupt-location[J]);                assume device J interrupts
   P ← intrql);
off ⇒ ( );
¬ Interrupt-rq ∧ Wait ⇒ ( );
   State-change(x) := (                                       for stacking state and restore
      SP ← SP - 2; next
      Mw[SP] ← PS;
      SP ← SP - 2; next
      Mw[SP] ← PC;
      PC ← Mw[x];
      PS ← Mw[x+2]
Boundary-Error ⇒ (State-change(4_8); Boundary-error ← 0)

Time-Out-Error → (State-change(4_8); Time-Out-Error ← 0)
Power-Fail-Flag ⇒ (state-change(24_8); Power-Fail-Flag ← 0;) program must turn off computer
Power-Up-Flag ⇒ (PC ← 24_8; Power-Up-Flag ← 0; Activity ← 0) Start Up on power-up
```

*Instruction-Set Definition*

*Each instruction is defined in ISP in the text, therefore, it will not be repeated here.*

[1]a 17 bit result, r, used only for descriptive purposes

[2]A prime is used in S (e.g., S') and D (e.g., D') to indicate that when a word is accessed in this fashion, side effects may occur. That is, registers of R may be changed.

[3]If all 16 bits of result, r = 0, then Z is set to 1 else Z is set to 0.

[4]The 8 least significant bits are used to form a 16-bit positive or negative number by extending bit 7 into 15:8.

[5]a ⇒ b means: if boolean a is true then b is executed.

[6]Mw means the memory taken as a work-organized memory.

# INDEX