processor handbook

pdp11/40
digital

pdp11/40

processor
handbook

digital equipment corporation
Copyright © 1972, by Digital Equipment Corporation

DEC, PDP, UNIBUS are registered trademarks of Digital Equipment Corporation.
# TABLE OF CONTENTS

## CHAPTER 1 INTRODUCTION .............................................................. 1-1

1.1 GENERAL .................................................................................. 1-1

1.2 GENERAL CHARACTERISTICS ....................................................... 1-2
  1.2.1 The UNIBUS ........................................................................... 1-2
  1.2.2 Central Processor .................................................................. 1-3
  1.2.3 Memories ............................................................................... 1-5
  1.2.4 Floating Point ....................................................................... 1-5
  1.2.5 Memory Management ............................................................ 1-5

1.3 PERIPHERALS/OPTIONS ............................................................... 1-5
  1.3.1 I/O Devices ........................................................................... 1-6
  1.3.2 Storage Devices ..................................................................... 1-6
  1.3.3 Bus Options .......................................................................... 1-6

1.4 SOFTWARE .................................................................................. 1-6
  1.4.1 Paper Tape Software ............................................................. 1-7
  1.4.2 Disk Operating System Software ........................................... 1-7
  1.4.3 Higher Level Languages ........................................................ 1-7

1.5 NUMBER SYSTEMS ..................................................................... 1-7

## CHAPTER 2 SYSTEM ARCHITECTURE .................................................. 2-1

2.1 SYSTEM DEFINITION ................................................................... 2-1

2.2 UNIBUS ...................................................................................... 2-1
  2.2.1 Bidirectional Lines ............................................................... 2-1
  2.2.2 Master-Slave Relation .......................................................... 2-2
  2.2.3 Interlocked Communication .................................................. 2-2

2.3 CENTRAL PROCESSOR .................................................................. 2-2
  2.3.1 General Registers ................................................................. 2-3
  2.3.2 Processor Status Word ......................................................... 2-4
  2.3.3 Stack Limit Register ............................................................. 2-5

2.4 EXTENDED INSTRUCTION SET & FLOATING POINT ......................... 2-5

2.5 CORE MEMORY .......................................................................... 2-6

2.6 AUTOMATIC PRIORITY INTERRUPTS ........................................... 2-7
  2.6.1 Using the Interrupts ............................................................... 2-9
  2.6.2 Interrupt Procedure .............................................................. 2-9
  2.6.3 Interrupt Servicing ............................................................... 2-10

2.7 PROCESSOR TRAPS .................................................................. 2-10
  2.7.1 Power Failure ....................................................................... 2-10
  2.7.2 Odd Addressing Errors ......................................................... 2-10
  2.7.3 Time-out Errors .................................................................... 2-11
  2.7.4 Reserved Instructions ........................................................... 2-11
  2.7.5 Trap Handling ...................................................................... 2-11
CHAPTER 6  MEMORY MANAGEMENT

6.1 PDP-11 FAMILY BASIC ADDRESSING LOGIC ....................... 6-1
6.2 VIRTUAL ADDRESSING .................................................. 6-2
6.3 INTERRUPT CONDITIONS UNDER MANAGEMENT CONTROL .......... 6-2
6.4 CONSTRUCTION OF A PHYSICAL ADDRESS ......................... 6-3
6.5 MANAGEMENT REGISTERS .............................................. 6-4
   6.5.1 Page Address Register ........................................ 6-5
   6.5.2 Page Descriptor Register ..................................... 6-5
6.6 FAULT REGISTERS .......................................................... 6-7
   6.6.1 Status Register #0 ................................................ 6-7
   6.6.2 Status Register #2 .............................................. 6-8

CHAPTER 7  INTERNAL PROCESSOR OPTIONS

7.1 GENERAL ........................................................................... 7-1
7.2 EIS OPTION ...................................................................... 7-1
7.3 FLOATING POINT OPTION ............................................... 7-3
7.4 STACK LIMIT OPTION .................................................... 7-5

CHAPTER 8  CONSOLE OPERATION

8.1 CONSOLE ELEMENTS ........................................................ 8-1
8.2 STATUS INDICATORS ...................................................... 8-2
8.3 CONSOLE SWITCHES ....................................................... 8-3
8.4 DISPLAYS ......................................................................... 8-4

CHAPTER 9  SPECIFICATIONS

9.1 PACKAGING ...................................................................... 9-1
9.2 CPU OPERATING SPECIFICATIONS ................................... 9-1
9.3 OTHER EQUIPMENT ....................................................... 9-1
9.4 PDP-11 FAMILY OF COMPUTERS ..................................... 9-4

Appendix A  Instruction Set Processor ...................................... A-1
Appendix B  Memory Map ....................................................... B-1
Appendix C  PDP-11/40 Instruction Timing ............................. C-1
Appendix D  Instruction Index and Numerical Op Code List ......... D-1
Appendix E  Summary of PDP11 Instructions ............................ E-1
CHAPTER 1

INTRODUCTION

1.1 GENERAL

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

This Handbook describes the PDP-11/40, one of the latest computers in the PDP-11 family from Digital Equipment Corporation (DEC). This powerful, low-priced machine is packaged in a 21” front panel slide chassis, allowing convenient access and expansion when mounted in a standard rack. The PDP-11/40 was designed to fit a broad range of applications, from small stand alone situations where the computer consists of only 8K of memory and a processor, to large multi-user, multi-task applications requiring up to 124K of addressable memory space. Among its major features are a fast central processor with a choice of floating point and sophisticated memory management, both of which are hardware options.

Some of the PDP-11/40 features are:

- 16-bit word (two 8-bit bytes)
  direct addressing of 32K 16-bit words or 64K 8-bit bytes ($K = 1024$)

- Word or byte processing
  very efficient handling of 8-bit characters

- Asynchronous operation
  systems run at their highest possible speed, replacement with faster devices means faster operation with no other hardware or software changes

- Modular component design
  extreme ease and flexibility in configuring systems

- Stack Processing
  hardware sequential memory manipulation makes it easy to handle structured data, subroutines, and interrupts
• 8 fast general-purpose registers
  very fast integrated circuits used interactively for instruction processing
• Automatic priority processing
  four-line, multi-level system is dynamically alterable
• Vectored interrupts
  fast interrupt response without device polling
• Single & double operand instructions
  powerful and convenient set of micro-programmed instructions

DEC References
The following publications contain supplementary and useful information:

Title
PDP-11 Peripherals and Interfacing
  Handbook
PDP-11 UNIBUS Interface Manual
Introduction to Programming
Small Computer Handbook

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.

![Figure 1-1 PDP-11 System Simplified Block Diagram](image)

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.

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.

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, address pointers, index registers, and other specialized functions. The processor can perform data transfers directly between I/O devices and memory without disturbing the processor registers; does both single- and double-operand addressing and handles both 16-bit word and 8-bit byte data.

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.

The basic order code of the PDP-11 uses both single and double operand address instructions for words or bytes. The PDP-11 therefore performs very efficiently in one step, such operations as adding or subtracting two operands, or moving an operand from one location to another.

PDP-11 Approach

ADD A,B ;add contents of location A to location B, store result at location B
Priority Interrupts

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

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

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 sequential addressing forwards or backwards, address indexing, indirect addressing, 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.
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.

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.

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.

1.2.4 Floating Point (optional)
A Floating Point Unit functions as an integral part of the PDP-11/40 processor, not as a bus device.

1.2.5 Memory Management (optional)
PDP-11/40 Memory Management is an advanced memory extension, relocation, and protection feature which will:

- extend memory space from 28K to 124K words
- allow efficient segmentation of core for multi-user environments
- provide effective protection of memory segments in multi-user environments

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 manu-
facturer of peripherals, DEC can offer extremely reliable equipment, lower prices, more choice and quantity discounts.

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 144K 16-bit words. For applications which require handling of large volumes of data, DEC offers the industry compatible TU10 Magtape.

Disk storage include fixed-head disk units and moving-head removable cartridge and disk pack units. These devices range from the 64K 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 64K-256K word fixed-head disk
- RS11 256K-2M word fixed-head disk
- RK05 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 avail-
able 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 (I0X)
- Math Package (FPP11)

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

- Text Editor (ED11)
- MACRO Assembler (MACRO-11)
- 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.

BATCH
The BATCH System adds job stream processing to the DOS System.

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.

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

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

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

Decimal—for all normal referencing to quantities
Octal Representation

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

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

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Octal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000000</td>
</tr>
<tr>
<td>(2^{16}-1)=65,535</td>
<td>177777 (16-bit limit)</td>
</tr>
<tr>
<td>(2^{18}-1)=262,143</td>
<td>777777 (18-bit limit)</td>
</tr>
</tbody>
</table>

2's Complement Numbers

In this system, the first bit (bit 15) is used to indicate the sign;

0 = positive
1 = negative

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

<table>
<thead>
<tr>
<th>Decimal</th>
<th>2's Complement (Octal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>largest positive +32,767</td>
<td>0  77777</td>
</tr>
<tr>
<td>+32,766</td>
<td>0  77776</td>
</tr>
<tr>
<td>+1</td>
<td>0  00001</td>
</tr>
<tr>
<td>0</td>
<td>0  00000</td>
</tr>
<tr>
<td>-1</td>
<td>1  77777</td>
</tr>
<tr>
<td>-2</td>
<td>1  77776</td>
</tr>
<tr>
<td>-32,767</td>
<td>1  00001</td>
</tr>
<tr>
<td>most negative -32,768</td>
<td>1  00000</td>
</tr>
</tbody>
</table>
CHAPTER 2

SYSTEM ARCHITECTURE

2.1 SYSTEM DEFINITION

The PDP-11/40 is a 16-bit, general-purpose, parallel logic computer using 2's complement arithmetic. The processor can directly address 32,768 16-bit words or 65,536 8-bit bytes.

The Central Processing Unit performs all arithmetic and logical operations required in the system. A Floating Point Unit mounts integrally into the Central Processor as does a Memory Management Unit which provides a full memory management facility through relocation and protection.

The PDP-11/40 hardware has been optimized towards a multi-programming environment and the processor therefore operates in two modes (Kernel and User). By taking full advantage of this feature, a software operating system can insure that no user (who is operating in User mode) can cause a failure (crash) of the entire system. Full control of the entire system is retained at the console or by an operator who is in Kernel mode.

2.2 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. 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.

2.2.1 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.

2.2.2 Master-Slave Relation
Communication between two devices on the bus is in the form of a master-slave relationship. At any point in time, there is one device that has control of the bus. This controlling device is termed the "bus master". The master device controls the bus when communicating with another device on the bus, termed the "slave". A typical example of this relationship is the processor, as master, fetching an instruction from memory (which is always a slave). Another example is the disk, as master, transferring data to memory, as slave. Master-slave relationships are dynamic. The processor, for example, may pass bus control to a disk. The disk, as master, could then communicate with a slave memory bank.

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

2.2.3 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.

2.3 CENTRAL PROCESSOR
The PDP-11/40 performs all arithmetic and logical operations required in the system. It also acts as the arbitration unit for UNIBUS control by regulating bus requests and transferring control of the bus to the requesting device with the highest priority.

Space is provided within the central processor for the following options:

- Extended Instruction Set
- Floating Point Unit
- Memory Management Unit
- Programmable Stack Limit

The machine operates in two modes; Kernel and User. When the machine is in Kernel mode a program has complete control of the machine; when in User mode the processor is inhibited from executing certain instructions and can be denied direct access to the peripherals on the system. This hardware feature can be used to provide complete executive protection in a multi-programming environment.

The central processor contains 8 general registers which can be used as accumulators, index registers, or as stack pointers. A stack, as used
in the PDP-11, is an area of memory set aside by the programmer for temporary storage or subroutine/interrupt service linkage. A program can add or delete words or bytes within the stack. The stack uses the "last-in, first-out" concept; that is, various items may be added to a stack in sequential order and retrieved or deleted from the stack in reverse order. On the PDP-11, a stack starts at the highest location reserved for it and expands linearly downward to the lowest address as items are added. Stacks are extremely useful for nesting programs, creating re-entrant coding, and as temporary storage where a Last-In, First-Out structure is desirable. One of the general registers is used as the PDP-11/40's Program Counter. Two others are used as Processor Stack Pointers, one for each operational mode.

The CPU performs all of the computer's computation and logic operations in a parallel binary mode through step by step execution of individual instructions.

2.3.1 General Registers

The general registers can be used for a variety of purposes; the uses varying with requirements. The general registers can be used as accumulators, index registers, autoincrement registers, autodecrement registers, or as stack pointers for temporary storage of data. Chapter 3 on Addressing describes these uses of the general registers in more detail. Arithmetic operations can be from one general register to another, from one memory or device register to another, or between memory or a device register and a general register.

![Figure 2-1 The General Registers](image)

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

The R6 register is normally used as the Processor Stack Pointer indicating the last entry in the appropriate stack (a common temporary storage area with "Last-in First-Out" characteristics). The two stacks (with the Memory Management option) are called the Kernel Stack and the User Stack. When the Central Processor is operating in Kernel mode it uses the Kernel Stack and in User mode, the User Stack. When an interrupt or trap occurs, the PDP-11/40 automatically saves its current status on the Processor Stack selected by the service routine. This stack-based architecture facilitates reentrant programming.

2.3.2 Processor Status Word

![Processor Status Word Diagram]

The Processor Status Word (PS), located at location 777776, contains information on the current status of the PDP-11/40. This information includes the current processor priority: current and previous operational modes; the condition codes describing the results of the last instruction; and an indicator for detecting the execution of an instruction to be trapped during program debugging.

Modes (with Memory Management Option)
Mode information includes the present mode, either User or Kernel (bits 15, 14) and the mode the machine was in prior to the last interrupt or trap (bits 13, 12).

The two modes permit a fully protected environment for a multi-programming system by providing the user with two distinct sets of Processor Stacks and Memory Management Registers for memory mapping. In User mode a program is inhibited from executing a "HALT" instruction and the processor will trap through location 10 if an attempt is made to execute this instruction. Furthermore, the processor will ignore the "RESET" instruction. In Kernel mode, the processor will execute all instructions.

A program operating in Kernel mode can map users' programs anywhere in core and thus explicitly protect key areas (including the device registers and the Processor Status Word) from the User operating environment.
Processor Priority
The Central Processor operates at any one of eight levels of priority, 0-7. When the CPU is operating at level 7 an external device cannot interrupt it with a request for service. The Central Processor must be operating at a lower priority than the external device's request in order for the interruption to take effect. The current priority is maintained in the processor status word (bits 5-7). The 8 processor levels provide an effective interrupt mask.

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

The bits are set as follows:

\[ \begin{align*}
Z &= 1, \text{if the result was zero} \\
N &= 1, \text{if the result was negative} \\
C &= 1, \text{if the operation resulted in a carry from the MSB} \\
V &= 1, \text{if the operation resulted in an arithmetic overflow}
\end{align*} \]

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

Interrupts and trap instructions both automatically cause the previous Processor Status Word and Program Counter to be saved and replaced by the new values corresponding to those required by the routine servicing the interrupt or trap. The user can, thus, cause the central processor to automatically switch modes, or disable the Trap Bit whenever a trap or interrupt occurs.

2.3.3 Stack Register (with Memory Management option)
All PDP-11's have a Stack Overflow Boundary at location 4000. The Kernel Stack Boundary, in the PDP-11/40 is a variable boundary set through the Stack Limit Register found at location 777774.

Once the Kernel stack exceeds its boundary, the Processor will complete the current instruction and then trap to location 4 (Yellow or Warning Stack Violation). If, for some reason, the program persists beyond the 16-word limit, the processor will abort the offending instruction, set the stack point (R6) to 4 and trap to location 4 (Red or Fatal Stack Violation).

2.4 EXTENDED INSTRUCTION SET & FLOATING POINT
The Extended Instruction Set (EIS) option fits within the Central Processor mounting assembly. It provides the capability of performing hardware fixed point arithmetic and allows direct implementation of multiply, divide, and multiple shifting. A double-precision 32-bit word can be handled.

The Floating Point Unit, which uses the EIS as a prerequisite, fits within the CPU mounting assembly. This option enables the execution of 4
special instructions for floating point addition, subtraction, multiplication, and division. The EIS and Floating Point hardware provide significant time and coding improvement over comparable software routines.

2.5 CORE MEMORY
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 in Figure 2-3.

![Figure 2-3 Memory Addresses](image)

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 shown in Figure 2-4.

![Figure 2-4 High & Low Byte](image)

Low bytes are stored at even-numbered memory locations and high bytes at odd-numbered memory locations. Thus it is convenient to view the PDP-11 memory as shown in Figure 2-5.
Certain memory locations have been reserved by the system for interrupt and trap handling, processor stacks, general registers, and peripheral device registers. Kernel virtual addresses from 0 to 370 are always reserved and those to 777 are reserved on large system configurations for traps and interrupt handling. The top 4,096 word addresses (from 770000 up) have been reserved for general registers and peripheral devices.

A 16-bit word used for byte addressing can address a maximum of 32K words. However, the top 4,096 word locations are traditionally reserved for peripheral and register addresses and the user therefore has 28K of core to program. To expand above 28K the user must use the Memory Management Unit. This device provides an 18-bit effective memory address which permits addressing up to 124K words of actual memory. The unit also provides a facility which permits individual user programs up to 32K in length and provides a relocation and protection facility through two sets of 8 registers.

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.

2.6 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.

Bus requests from external devices can be made on one of five request lines. Highest priority is assigned to non-processor request (NPR). These are direct memory access type transfers, and are honored by the processor 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-11/40. 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.

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 2.6 microseconds or less. An NPR device in control of the bus may transfer 16-bit words from memory at memory speed.

2.6.1 Using the Interrupts

Devices that gain bus control with one of the Bus Request lines (BR 7 - BR 4), can take full advantage of the Central Processor by requesting an interrupt. In this way, the entire instruction set is available for manipulating data and status registers.

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

2.6.2 Interrupt Procedure

Interrupt handling is automatic in the PDP-11/40. No device polling is required to determine which service routine to execute. The operations required to service an interrupt are as follows:

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

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

3. The processor stores the current Processor Status Word (PS) and the current Program Counter (PC) into CPU temporary registers.
4. The new PC and PS (the interrupt vector) are taken from the specified address. The old PS and PC are then pushed onto the current stack as indicated by bits 15,14 of the new PS and the previous mode in effect is stored in bits 13,12 of the new PS. The service routine is then initiated.

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

This instruction requires 2.9 μsec providing there is no NPR request.

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

2.6.3 Interrupt Servicing

Every hardware device capable of interrupting the processor has a unique set of locations (2 words) reserved for its interrupt vector. The first word contains the location of the device's service routine, and the second, the Processor Status Word that is to be used by the service routine. Through proper use of the PS, the programmer can switch the operational mode of the processor, and modify the Processor’s Priority level to mask out lower level interrupts.

2.7 PROCESSOR TRAPS

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

2.7.1 Power Failure

Whenever AC power drops below 95 volts for 115v power (190 volts for 230v) or outside a limit of 47 to 63 Hz, 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), and condition peripherals for power fail.

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

2.7.2 Odd Addressing Errors

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

2.7.3 Time-out Errors
These errors occur when a Master Synchronization pulse is placed on the UNIBUS and there is no slave pulse within 15μsec. This error usually occurs in attempts to address non-existent memory or peripherals.

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

2.7.4 Reserved Instructions
There is a set of illegal and reserved instructions which cause the processor to trap through location 10.

2.7.5 Trap Handling
Appendix B includes a list of the reserved Trap Vector locations, and System Error Definitions which cause processor traps. When a trap occurs, the processor follows the same procedure for traps as it does for interrupts (saving the PC and PS on the new Processor Stack etc. . . .)

In cases where traps and interrupts occur concurrently, the processor will service the conditions according to the following priority sequence.

Odd Addressing Error
Fatal Stack Violations (Red)
Memory Management Violations
Timeout Errors
Trap Instructions
Trace Trap
Warning Stack Violation (Yellow)
Power Failure
Processor Priority level 7
Floating Point Exception Trap
BR 7

Processor 0
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 autodecrement addressing; automatically stepping backwards is known as autodecrement addressing. These modes are particularly useful for processing tabular data.
- as index registers. In this instance the contents of the register, and the word following the instruction are summed to produce the address of the operand. This allows easy access to variable entries in a list.

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

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

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

An important PDP-11/40 feature, which must be considered in conjunction with the addressing modes, is the register arrangement;
Six general purpose registers (R0-R5)

A hardware stack pointer (R6), (2 with Memory Management)

A Program Counter (PC) register (R7).

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

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

![Instruction Format Diagram]

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

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

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

b) Bits 3 through 5 specify how the selected register will be used (address mode). Bit 3 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:
The source address field is used to select the source operand, the first operand. The destination is used similarly, and locates the second operand and the result. For example, the instruction ADD A, B adds the contents (source operand) of location A to the contents (destination operand) of location B. After execution B will contain the result of the addition and the contents of A will be unchanged.

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

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

DD = destination field (6 bits)
SS = source field (6 bits)
( ) = contents of
### 3.3 DIRECT ADDRESSING

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

#### DIRECT MODES

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

#### 3.3.1 Register Mode

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:

\[
R0 = \%0 \quad (\% \text{ sign indicates register definition})
\]

\[
R1 = \%1
\]

\[
R2 = \%2, \text{ etc.}
\]

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)

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>Octal Code</th>
<th>Instruction Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. INC R3</td>
<td>005203</td>
<td>Increment</td>
</tr>
</tbody>
</table>

Operation: Add one to the contents of general register 3

---

3-4
2. ADD R2, R4 060204 Add
Operation: Add the contents of R2 to the contents of R4.

BEFORE
\[
\begin{array}{c}
R2 \quad 000002 \\
R4 \quad 000004 \\
\end{array}
\]

AFTER
\[
\begin{array}{c}
R2 \quad 000002 \\
R4 \quad 000006 \\
\end{array}
\]

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)

BEFORE
\[
\begin{array}{c}
R4 \quad 022222 \\
\end{array}
\]

AFTER
\[
\begin{array}{c}
R4 \quad 022155 \\
\end{array}
\]

3.3.2 Autoincrement Mode

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

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>Octal Code</th>
<th>Instruction Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR (R5) +</td>
<td>005025</td>
<td>Clear</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>BEFORE ADDRESS SPACE</th>
<th>REGISTER</th>
<th>AFTER ADDRESS SPACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>20000</td>
<td>005025</td>
<td>20000</td>
</tr>
<tr>
<td>30000</td>
<td>111116</td>
<td>30000</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>BEFORE ADDRESS SPACE</th>
<th>REGISTER</th>
<th>AFTER ADDRESS SPACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>20000</td>
<td>105025</td>
<td>20000</td>
</tr>
<tr>
<td>30000</td>
<td>11116</td>
<td>30002</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>BEFORE ADDRESS SPACE</th>
<th>REGISTERS</th>
<th>AFTER ADDRESS SPACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>062204</td>
<td>10000</td>
</tr>
<tr>
<td>10002</td>
<td>010000</td>
<td>10002</td>
</tr>
<tr>
<td>R2</td>
<td>100002</td>
<td>R2</td>
</tr>
<tr>
<td>R4</td>
<td>010000</td>
<td>R4</td>
</tr>
</tbody>
</table>

3-6
3.3.3 Autodecrement Mode

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

Autodecrement Mode Examples

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>Octal Code</th>
<th>Instruction Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. INC-(R0)</td>
<td>005240</td>
<td>Increment</td>
</tr>
</tbody>
</table>

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

2. INCB-(R0) | 105240 | Increment Byte |

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

3. ADD-(R3).R0 | 064300 | Add |

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

OPR X(Rn)

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

Index Mode Examples

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>Octal Code</th>
<th>Instruction Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR 200(R4)</td>
<td>005064</td>
<td>Clear</td>
</tr>
<tr>
<td></td>
<td>000200</td>
<td></td>
</tr>
</tbody>
</table>

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 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
000030
000020

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

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

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

<table>
<thead>
<tr>
<th>Mode</th>
<th>Name</th>
<th>Assembler Syntax</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Register Deferred</td>
<td>@Rn or (Rn)</td>
<td>Register contains the address of the operand</td>
</tr>
<tr>
<td>3</td>
<td>Autoincrement Deferred</td>
<td>@(Rn) +</td>
<td>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).</td>
</tr>
<tr>
<td>5</td>
<td>Autodecrement Deferred</td>
<td>@-(Rn)</td>
<td>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</td>
</tr>
<tr>
<td>7</td>
<td>Index Deferred</td>
<td>@X(Rn)</td>
<td>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.</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>Octal Code</th>
<th>Instruction Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR @R5</td>
<td>005015</td>
<td>Clear</td>
</tr>
</tbody>
</table>

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

---

3-10
Autoincrement Deferred Mode Example
Symbolic: INC@R2 +
Octal Code: 005232
Instruction Name: Increment

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

Autodecrement Deferred Mode Example
Symbolic: COM@-(R0)
Octal Code: 005150
Instruction Name: 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
Symbolic: ADD @ 1000(R2),R1
Octal Code: 067201
Instruction Name: Add

Operation: 1000 and contents of R2 are summed to produce the address of the address of the source operand the contents of which are added to contents of R1; the result is stored in R1.
3.5 USE OF THE PC AS A GENERAL REGISTER

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

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

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

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

When a standard program is available for different users, it often is helpful to be able to load it into different areas of core and run it there. PDP-11's can accomplish the relocation of a program very efficiently through the use of position inde-
pendent code (PIC) which is written by using the PC addressing modes. If an instruc-
tion 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 partic-
ularly true of the immediate and relative modes.

3.5.1 Immediate Mode

**OPR #n,DD**

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

**Immediate Mode Example**

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>Octal Code</th>
<th>Instruction Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD #10,R0</td>
<td>062700</td>
<td>Add 000010</td>
</tr>
</tbody>
</table>

**Operation:**
The value 10 is located in the second word of the instruction and is added to the contents of R0.
Just before this instruction is fetched and executed, the PC points to the first word of the
instruction. The processor fetches the first word and increments the PC by two. The source operand
mode is 27 (autoincrement the PC). Thus, the PC is used as a pointer to fetch the operand (the sec-
ond 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 us-
ing 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 as-
sembled instruction is executed).
Absolute Mode Examples

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

Operation: Clear the contents of location 1100.

2. ADD @ #2000,R3 | 063703 | 002000


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

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>Octal Code</th>
<th>Instruction Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>INC A</td>
<td>005267</td>
<td>Increment</td>
</tr>
<tr>
<td></td>
<td>000054</td>
<td></td>
</tr>
</tbody>
</table>

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.

BEFORE

<table>
<thead>
<tr>
<th>ADDRESS SPACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1020</td>
</tr>
<tr>
<td>1022</td>
</tr>
<tr>
<td>1024</td>
</tr>
<tr>
<td>1026</td>
</tr>
<tr>
<td>10100</td>
</tr>
</tbody>
</table>

AFTER

<table>
<thead>
<tr>
<th>ADDRESS SPACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1020</td>
</tr>
<tr>
<td>1022</td>
</tr>
<tr>
<td>1024</td>
</tr>
<tr>
<td>1026</td>
</tr>
<tr>
<td>10100</td>
</tr>
</tbody>
</table>

3.5.4 Relative Deferred Addressing

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

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

Relative Deferred Mode Example

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>Octal Code</th>
<th>Instruction Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR @A</td>
<td>005077</td>
<td>Clear</td>
</tr>
<tr>
<td></td>
<td>000020</td>
<td></td>
</tr>
</tbody>
</table>

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

BEFORE

<table>
<thead>
<tr>
<th>ADDRESS SPACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1020</td>
</tr>
<tr>
<td>1022</td>
</tr>
<tr>
<td>1024</td>
</tr>
<tr>
<td>1044</td>
</tr>
<tr>
<td>10100</td>
</tr>
</tbody>
</table>

AFTER

<table>
<thead>
<tr>
<th>ADDRESS SPACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1020</td>
</tr>
<tr>
<td>1022</td>
</tr>
<tr>
<td>1024</td>
</tr>
<tr>
<td>1044</td>
</tr>
<tr>
<td>10100</td>
</tr>
</tbody>
</table>
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. Auto-decrement with Register 6 "pushes" data on to the stack and autoincrement with Register 6 "pops" data off the stack. Index mode with SP permits random access of items on the stack. Since the SP is used by the processor for interrupt handling, it has a special attribute: autoincrements and autodecrements are always done in steps of two. Byte operations using the SP in this way leave odd addresses unmodified.

With the Memory Management option there are two R6 registers selected by the PS; but at any given time there is only one in operation.

3.7 SUMMARY OF ADDRESSING MODES

3.7.1 General Register Addressing

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

Mode 0  Register  OPR R  R contains operand

Mode 1  Register deferred  OPR (R)  R contains address

Mode 2  Auto-increment  OPR (R)+
R contains address, then increment (R)

R2 FOR WORD, R1 FOR BYTE
Mode 3  Auto-increment  OPR  @(R)+  R contains address of address, then increment (R) by 2

Mode 4  Auto-decrement  OPR  -(R)
Decrement (R), then R contains address

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

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

Mode 7  Index deferred  OPR  @X(R)  (R) + X is address of address
3.7.2 Program Counter Addressing

Register = 7

Mode 2  Immediate  OPR #n  Operand n follows instruction

Mode 3  Absolute  OPR @#A  Address A follows instruction

Mode 6  Relative  OPR A  PC + 4 + X is address updated PC

Mode 7  Relative deferred  OPR @A  PC + 4 + X is address of address updated PC
CHAPTER 4

INSTRUCTION SET

4.1 INTRODUCTION

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

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

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

SYMBOLS:

( ) = contents of
SS or src = source address
DD or dst = destination address
loc = location
← = becomes
↑ = "is popped from stack"
↓ = "is pushed onto stack"
∧ = boolean AND
∨ = boolean OR
⊕ = exclusive OR
¬ = boolean not
Reg or R = register
B = Byte

\[ n = \begin{cases} 0 & \text{for word} \\ 1 & \text{for byte} \end{cases} \]
4.2 INSTRUCTION FORMATS
The major instruction formats are:

Single Operand Group

Double Operand Group

Register-Source or Destination

Branch

4-2
**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:

![Address Chart]

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

**Example:**

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>Octal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR</td>
<td>0050DD</td>
<td>Clear Word</td>
</tr>
<tr>
<td>CLRB</td>
<td>1050DD</td>
<td>Clear Byte</td>
</tr>
</tbody>
</table>
4.3 LIST OF INSTRUCTIONS
The PDP-11/40 instruction set is shown in the following sequence.

SINGLE OPERAND

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Op Code</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR(B)</td>
<td>clear destination</td>
<td>050DD</td>
<td>4.6</td>
</tr>
<tr>
<td>COM(B)</td>
<td>complement dst</td>
<td>051DD</td>
<td>4.7</td>
</tr>
<tr>
<td>INC(B)</td>
<td>increment dst</td>
<td>052DD</td>
<td>4.8</td>
</tr>
<tr>
<td>DEC(B)</td>
<td>decrement dst</td>
<td>053DD</td>
<td>4.9</td>
</tr>
<tr>
<td>NEG(B)</td>
<td>negate dst</td>
<td>054DD</td>
<td>4.10</td>
</tr>
<tr>
<td>TST(B)</td>
<td>test dst</td>
<td>057DD</td>
<td>4.11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shift &amp; Rotate</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASR(B)</td>
</tr>
<tr>
<td>ASL(B)</td>
</tr>
<tr>
<td>ROR(B)</td>
</tr>
<tr>
<td>ROL(B)</td>
</tr>
<tr>
<td>SWAB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Multiple Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC(B)</td>
</tr>
<tr>
<td>SBC(B)</td>
</tr>
<tr>
<td>SXT</td>
</tr>
</tbody>
</table>

DOUBLE OPERAND

<table>
<thead>
<tr>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV(B)</td>
</tr>
<tr>
<td>CMP(B)</td>
</tr>
<tr>
<td>ADD</td>
</tr>
<tr>
<td>SUB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logical</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT(B)</td>
</tr>
<tr>
<td>BIC(B)</td>
</tr>
<tr>
<td>BIS(B)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL</td>
</tr>
<tr>
<td>DIV</td>
</tr>
<tr>
<td>ASH</td>
</tr>
<tr>
<td>ASHC</td>
</tr>
<tr>
<td>XOR</td>
</tr>
</tbody>
</table>
## PROGRAM CONTROL

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Op Code or Base Code</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Branch</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BR</td>
<td>branch (unconditional)</td>
<td>000400</td>
<td>4-37</td>
</tr>
<tr>
<td>BNE</td>
<td>branch if not equal (to zero)</td>
<td>001000</td>
<td>4-38</td>
</tr>
<tr>
<td>BEQ</td>
<td>branch if equal (to zero)</td>
<td>001400</td>
<td>4-39</td>
</tr>
<tr>
<td>BPL</td>
<td>branch if plus</td>
<td>100000</td>
<td>4-40</td>
</tr>
<tr>
<td>BMI</td>
<td>branch if minus</td>
<td>100400</td>
<td>4-41</td>
</tr>
<tr>
<td>BVC</td>
<td>branch if overflow is clear</td>
<td>102000</td>
<td>4-42</td>
</tr>
<tr>
<td>BVS</td>
<td>branch if overflow is set</td>
<td>102400</td>
<td>4-43</td>
</tr>
<tr>
<td>BCC</td>
<td>branch if carry is clear</td>
<td>103000</td>
<td>4-44</td>
</tr>
<tr>
<td>BCS</td>
<td>branch if carry is set</td>
<td>103400</td>
<td>4-45</td>
</tr>
<tr>
<td><strong>Signed Conditional Branch</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BGE</td>
<td>branch if greater than or equal (to zero)</td>
<td>002000</td>
<td>4-47</td>
</tr>
<tr>
<td>BLT</td>
<td>branch if less than (zero)</td>
<td>002400</td>
<td>4-48</td>
</tr>
<tr>
<td>BGT</td>
<td>branch if greater than (zero)</td>
<td>003000</td>
<td>4-49</td>
</tr>
<tr>
<td>BLE</td>
<td>branch if less than or equal (to zero)</td>
<td>003400</td>
<td>4-50</td>
</tr>
<tr>
<td><strong>Unsigned Conditional Branch</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BHI</td>
<td>branch if higher</td>
<td>101000</td>
<td>4-52</td>
</tr>
<tr>
<td>BLO</td>
<td>branch if lower or same</td>
<td>101400</td>
<td>4-53</td>
</tr>
<tr>
<td>BHS</td>
<td>branch if higher or same</td>
<td>103000</td>
<td>4-54</td>
</tr>
<tr>
<td>BLO</td>
<td>branch if lower</td>
<td>103400</td>
<td>4-55</td>
</tr>
<tr>
<td><strong>Jump &amp; Subroutine</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP</td>
<td>jump</td>
<td>0001DD</td>
<td>4-56</td>
</tr>
<tr>
<td>JSR</td>
<td>jump to subroutine</td>
<td>004RDD</td>
<td>4-58</td>
</tr>
<tr>
<td>RTS</td>
<td>return from subroutine</td>
<td>00020R</td>
<td>4-60</td>
</tr>
<tr>
<td>MARK</td>
<td>mark</td>
<td>006400</td>
<td>4-61</td>
</tr>
<tr>
<td>SOB</td>
<td>subtract one and branch (if ( i \neq 0 ))</td>
<td>077R00</td>
<td>4-63</td>
</tr>
<tr>
<td><strong>Trap &amp; Interrupt</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EMT</td>
<td>emulator trap</td>
<td>104000—104377</td>
<td>4-65</td>
</tr>
<tr>
<td>TRAP</td>
<td>trap</td>
<td>104400—104777</td>
<td>4-66</td>
</tr>
<tr>
<td>BPT</td>
<td>breakpoint trap</td>
<td>000003</td>
<td>4-67</td>
</tr>
<tr>
<td>IOT</td>
<td>input/output trap</td>
<td>000004</td>
<td>4-68</td>
</tr>
<tr>
<td>RTI</td>
<td>return from interrupt</td>
<td>000002</td>
<td>4-69</td>
</tr>
<tr>
<td>RTT</td>
<td>return from interrupt</td>
<td>000006</td>
<td>4-70</td>
</tr>
<tr>
<td><strong>MISCELLANEOUS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HALT</td>
<td>halt</td>
<td>000000</td>
<td>4-74</td>
</tr>
<tr>
<td>WAIT</td>
<td>wait for interrupt</td>
<td>000001</td>
<td>4-75</td>
</tr>
<tr>
<td>RESET</td>
<td>reset external bus</td>
<td>000005</td>
<td>4-76</td>
</tr>
<tr>
<td>MFPI</td>
<td>move from previous instruction space</td>
<td>0065SS</td>
<td>4-77</td>
</tr>
<tr>
<td>MTPI</td>
<td>move to previous instruction space</td>
<td>0066DD</td>
<td>4-78</td>
</tr>
<tr>
<td><strong>Condition Code Operation</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLC, CLV, CLZ, CLN, CCC</td>
<td>clear</td>
<td>000240</td>
<td>4-79</td>
</tr>
<tr>
<td>SEC, SEV, SEZ, SEN, SCC</td>
<td>set</td>
<td>000260</td>
<td>4-79</td>
</tr>
</tbody>
</table>
4.4 SINGLE OPERAND INSTRUCTIONS

CLR
CLRB

clear destination

\[0/1 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad d \quad d \quad d \quad d \quad d \quad d \quad d \quad d\]

Operation: \((dst)\times 0\)

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

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

Example:

<table>
<thead>
<tr>
<th>Before</th>
<th>CLR R1</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R1) = 177777</td>
<td>(R1) = 000000</td>
<td></td>
</tr>
<tr>
<td>NZVC 1111</td>
<td>NZVC 0100</td>
<td></td>
</tr>
</tbody>
</table>

\[050DD\]
## COM

**COMB**

complement dst  

\[
\begin{array}{cccccccc}
0/1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & d & \ldots & d & d & d & d & d & 0 \\
\hline
15 & 6 & 5 & 4 & 3 & 2 & 1 & 0
\end{array}
\]

### Operation:
\[(dst) \rightarrow \sim (dst)\]

### 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:
**Before**  
\[(R0) = 013333\]

<table>
<thead>
<tr>
<th>NZVC</th>
<th>0110</th>
</tr>
</thead>
</table>

**After**  
\[(R0) = 164444\]

<table>
<thead>
<tr>
<th>NZVC</th>
<th>1001</th>
</tr>
</thead>
</table>
INC
INCB

increment dst ■052DD

Operation: (dst) = (dst) + 1

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

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

Example: INC R2

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

After (R2) = 000334
N Z V C
0 0 0 0
Decrement dst

**Operation:** 
\[(dst) \rightarrow (dst) - 1\]

**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:**

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R5) = 000001</td>
<td>(R5) = 000000</td>
</tr>
<tr>
<td>(NZVC)</td>
<td>(NZVC)</td>
</tr>
<tr>
<td>1000</td>
<td>0100</td>
</tr>
</tbody>
</table>

\[0/1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 1 \ d \ d \ d \ d \ d \ d \]

053DD
NEG
NEGB

negate dst

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:

Before After
(R0) = 000010 (R0) = 177770

N Z V C N Z V C
0 0 0 0 1 0 0 1
TST
TSTB

Operation:

\[(\text{dst}) \leftarrow (\text{dst})\]

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:

<table>
<thead>
<tr>
<th>Before</th>
<th>TST R1</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R1) = 012340</td>
<td></td>
<td>(R1) = 012340</td>
</tr>
<tr>
<td>N Z V C</td>
<td>N Z V C</td>
<td></td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>
**Shifts**

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

- **ASR** - Arithmetic shift right
- **ASL** - Arithmetic shift left

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

**Rotates**

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

\[ \text{Operation: } (\text{dst}) \ll (\text{dst}) \text{ shifted one place to the right} \]

\[ \text{Condition Codes: } \]
- 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

\[ \text{Description: } \]
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:

\[ \text{Byte: } \]

\[ \text{Odd Address: } \]

\[ \text{Even Address: } \]
ASL  
ASLB

arithmetic shift left  

\[ \text{Operation: } (\text{dst}) \leftarrow \text{shifted one place to the left} \]

\[ \text{Condition Codes:} \]

- \( N \): 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

\[ \text{Description:} \]

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

Word:
rotate right

\[\text{\textbf{ROR}}\]
\[\text{\textbf{RORB}}\]

\[\text{\#060DD}\]

\[\begin{array}{cccccccc}
0/1| & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
15 | & d & d & d & d & d & d & d & d \\
\end{array}\]

**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.

**Example:**
- **Word:**
- **Byte:**

\[\begin{array}{c}
\text{C} \\
15 \\
\end{array}\]

\[\begin{array}{c}
\text{C} \\
15 \\
\end{array}\]

\[\begin{array}{c}
\text{C} \\
7 \\
\end{array}\]

\[\begin{array}{c}
\text{C} \\
0 \\
\end{array}\]
**ROL**
**ROLB**

rotate left

<table>
<thead>
<tr>
<th>0/1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>d</th>
<th>d</th>
<th>d</th>
<th>d</th>
<th>d</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

**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:

*Image of diagrams showing the rotation process and condition codes.*
SWAB

swap bytes 0003DD

Operation: Byte 1/Byte 0 ↔ Byte 0/Byte 1

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

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

Example:

Before
(R1) = 077777
NZVC
1111

After
(R1) = 177577
NZVC
0000

4-17
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:

![Diagram of 32-bit word structure](image)

Example:

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

\[-1 = 37777777777\]

\[(R1) = 177777\]

\[(R2) = 177777\]

\[(R3) = 177777\]

\[(R4) = 177777\]

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

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

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.

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
subtract carry  

Operation: 
\[(dst) \leftarrow (dst) - (C)\]

Condition Codes: 
- \(N\): set if result < 0; cleared otherwise
- \(Z\): set if result 0; cleared otherwise
- \(V\): set if \((dst)\) was 100000; cleared otherwise
- \(C\): cleared if \((dst)\) was 0 and \(C\) was 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
```
SXT

sign extend

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|--|---|---|---|---|---|---|---|---|---|
| 15 | 10 | 0  | 1  | 1  | 0  | 1 | 1 | -1| -1| -1| -1| -1| -1| -1| -1| -1|
| 00  | 01  | 11  | 0  | 1  | 0  | 1 | 1 | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

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

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

Description: If the condition code bit N is set then a -1 is placed in the destination operand; if N bit is clear, then a 0 is placed in the destination operand. This instruction is particularly useful in multiple precision arithmetic because it permits the sign to be extended through multiple words.

Example:

Before
(A) = 012345
NZVC 1000

SXT A

After
(A) = 177777
NZVC 1000
4.5 DOUBLE OPERAND INSTRUCTIONS

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

move source to destination

\[ \text{Operation: } (\text{dst}) \leftrightarrow (\text{src}) \]

\[ \text{Condition Codes: } \]
N: set if \((\text{src}) < 0\); cleared otherwise
Z: set if \((\text{src}) = 0\); cleared otherwise
V: cleared
C: not affected

\[ \text{Description: } \]
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.

\[ \text{Example: } \]
MOV XXX,R1 ; loads Register 1 with the contents of memory location; XXX represents a programmer-defined mnemonic used to represent a memory location
MOV #20,R0 ; loads the number 20 into Register 0; '#' indicates that the value 20 is the operand
MOV @#20,-(R6) ; pushes the operand contained in location 20 onto the stack
MOV (R6)+,@ #177566 ; pops the operand off a stack and moves it into memory location 177566 (terminal print buffer)
MOV R1,R3 ; performs an inter register transfer
MOVB @#177562,@ #177566 ; moves a character from terminal keyboard buffer to terminal buffer
CMP
CMPB

compare src to dst

<table>
<thead>
<tr>
<th>O/1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>ss</th>
<th>ss</th>
<th>ss</th>
<th>ss</th>
<th>d</th>
<th>d</th>
<th>d</th>
<th>d</th>
<th>d</th>
<th>d</th>
<th>d</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>12</td>
<td>11</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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).
Operation: \( (\text{dst}) \leftarrow (\text{src}) + (\text{dst}) \)

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: \( \text{ADD}\ 20,\text{R0} \)
- Add to memory: \( \text{ADD}\ \text{R1,XXX} \)
- Add register to register: \( \text{ADD}\ \text{R1,R2} \)
- Add memory to memory: \( \text{ADD@}\ #\ 17750,\text{XXX} \)

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

subtract src from dst 16SSDD

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>s</th>
<th>s</th>
<th>s</th>
<th>s</th>
<th>s</th>
<th>s</th>
<th>d</th>
<th>d</th>
<th>d</th>
<th>d</th>
<th>d</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>12</td>
<td>11</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Operation:** 
(dst)\(\text{OF}\)\((\text{dst})-(\text{src})\) \[\text{in detail (dst)OF (dst) + ~(src) + 1}\]

**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 original contents of the destination are lost. The contents of the source are not affected. In double-precision arithmetic the C-bit, when set, indicates a "borrow".

**Example:**

```
Before                      After
(R1) = 011111                (R1) = 011111
(R2) = 012345                (R2) = 001234

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

4-26
**Logical**

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

**BITB**

bit test

<table>
<thead>
<tr>
<th>0/1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>s</th>
<th>s</th>
<th>s</th>
<th>s</th>
<th>s</th>
<th>d</th>
<th>d</th>
<th>d</th>
<th>d</th>
<th>d</th>
<th>d</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>12</td>
<td>11</td>
<td>6</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

Operation: (src) ∧ (dst)

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

Description: Performs logical "and" comparison of the source and destination operands and modifies condition codes accordingly. Neither the source nor destination operands are affected. The BIT instruction may be used to test whether any of the corresponding bits that are set in the destination are also set in the source or whether all corresponding bits set in the destination are clear in the source.

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

(30)₂ = 0 000 000 000 011 000

4-28
bit clear

Operation: $(dst) \leftarrow \sim (src) \land (dst)$

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:

Before: $\text{BIC R3, R4}$

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R3) = 001234</td>
<td>(R3) = 001234</td>
</tr>
<tr>
<td>(R4) = 001111</td>
<td>(R4) = 000101</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NZVC</th>
<th>NZVC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>0001</td>
</tr>
</tbody>
</table>

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

After: $(R4) = 000 000 001 000 001$
BIS
BISB

bit set

\[
\begin{array}{cccccccccccc}
0/1 & 1 & 0 & 1 & s & s & s & s & t & s & d & d & d & d & d & d \\
15 & 12 & 11 & 6 & 5 & 0 & & & & & & & & & &
\end{array}
\]

Operation: \((\text{dst}) \oplus (\text{src}) \lor (\text{dst})\)

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:
- \((R0) = 001234\)
- \((R1) = 001111\)
- \(N\ Z\ V\ C\ =\ 0\ 0\ 0\ 0\)

After:
- \((R0) = 001234\)
- \((R1) = 001335\)
- \(N\ Z\ V\ C\ =\ 0\ 0\ 0\ 0\)

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

After:
- \((R1) = 000\ 001\ 011\ 011\ 101\)
multiply

(EIS option) **MUL**

070RSS

```
0 1 1 1 0 0 0 r r r s s s s s
```

**Operation:**

\[ R, \ Rv1 \leftrightarrow \ R \times (\text{src}) \]

**Condition Codes:**

- **N:** set if product is \(<0\); cleared otherwise
- **Z:** set if product is \(0\); cleared otherwise
- **V:** cleared
- **C:** set if the result is less than \(-2^\text{15}\) or greater than or equal to \(2^\text{15}-1\).

**Description:**

The contents of the destination register and source taken as two's complement integers are multiplied and stored in the destination register and the succeeding register (if \(R\) is even). If \(R\) is odd only the low order product is stored. Assembler syntax is: **MUL S,R**.

(Note that the actual destination is \(R, Rv1\) which reduces to just \(R\) when \(R\) is odd.)

**Example:**

16-bit product (\(R\) is odd)

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

Before       After
(R1) = 000400  (R1) = 004000
```

4-31
DIV (EIS option)

Operation: \( R, Rv1 \leftarrow R, Rv1 \div (\text{src}) \)

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

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

Example:
- CLR R0
- MOV #20001,R1
- DIV #2,R0

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
<th>Quotient</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R0) = 000000</td>
<td>(R0) = 010000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(R1) = 020001</td>
<td>(R1) = 000001</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4-32
shift arithmetically

01111010

Operation: \( R \leftarrow R \) Shifted arithmetically NN places to right or left
Where NN = low order 6 bits of source.

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

Description:
The contents of the register are shifted right or left the number of times specified by the shift count. The shift count is taken as the low order 6 bits of the source operand. This number ranges from -32 to +31. Negative is a right shift and positive is a left shift.

6 LSB of source

<table>
<thead>
<tr>
<th>011111</th>
<th>Action in general register</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>Shift left 31 places</td>
</tr>
<tr>
<td>111111</td>
<td>shift left 1 place</td>
</tr>
<tr>
<td>100000</td>
<td>shift right 1 place</td>
</tr>
<tr>
<td>011111</td>
<td>shift right 32 places</td>
</tr>
</tbody>
</table>

Example:
Before
\[
\begin{align*}
(R3) &= 001234 \\
(R0) &= 000003
\end{align*}
\]
After
\[
\begin{align*}
(R3) &= 012340 \\
(R0) &= 000003
\end{align*}
\]
**ASHC (EIS option)**

**arithmetic shift combined**

| 0 | 1 | 1 | 1 | 0 | 1 | 1 | r | r | r | s | s | s | s | s | s | s | s | s | 0 |
| 15 | 9 | 8 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

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

**Condition Codes:**
- N: set if result <0; cleared otherwise
- Z: set if result = 0; cleared otherwise
- V: set if sign bit changes during the shift; cleared otherwise
- C: loaded with high order bit when left Shift; loaded with low order bit when right shift (loaded with the last bit shifted out of the 32-bit operand)

**Description:**
The contents of the register and the register ORed with one are treated as one 32 bit word. R + 1 (bits 0-15) and R (bits 16-31) are shifted right or left the number of times specified by the shift count. The shift count is taken as the low order 6 bits of the source operand. This number ranges from -32 to +31. Negative is a right shift and positive is a left shift. When the register chosen is an odd number the register and the register OR'ed with one are the same. In this case the right shift becomes a rotate (for up to a shift of 16). The 16 bit word is rotated right the number of bits specified by the shift count.

---

**Diagram:**

[Diagram showing the shift and rotate operations]

---

4-34
**XOR**

**exclusive OR**

| Operation: | (dst) = Rv(dst) |
| Condition Codes: | N: set if the result <0; cleared otherwise  
Z: set if result =0; cleared otherwise  
V: cleared  
C: unaffected |
| Description: | The exclusive OR of the register and destination operand is stored in the destination address. Contents of register are unaffected. Assembler format is: XOR R,D |
| Example: | XOR R0,R2 |

**Example:**

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R0) = 001234</td>
<td>(R0) = 001234</td>
</tr>
<tr>
<td>(R2) = 001111</td>
<td>(R2) = 000325</td>
</tr>
</tbody>
</table>

**Before:**

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

**After:**

(R2) = 0 000 000 011 010 101

4-35
4.6 PROGRAM CONTROL INSTRUCTIONS

Branches

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

a) the branch instruction is unconditional

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

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

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

The 8-bit offset allows branching in the backward direction by 200, words (400, bytes) from the current PC, and in the forward direction by 177, words (376, 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 permissible branch range is exceeded. Branch instructions have no effect on condition codes.
BR

branch (unconditional) 000400 Plus offset

| 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Operation: \( \text{PC} \leftarrow \text{PC} + (2 \times \text{offset}) \)

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

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

Updated PC = address of branch instruction + 2

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

<table>
<thead>
<tr>
<th>New PC Address</th>
<th>Offset Code</th>
<th>Offset (decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>474</td>
<td>375</td>
<td>-3</td>
</tr>
<tr>
<td>476</td>
<td>376</td>
<td>-2</td>
</tr>
<tr>
<td>500</td>
<td>377</td>
<td>-1</td>
</tr>
<tr>
<td>502</td>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>504</td>
<td>001</td>
<td>+1</td>
</tr>
<tr>
<td>506</td>
<td>002</td>
<td>+2</td>
</tr>
</tbody>
</table>
**BNE**

Branch if not equal (to zero)  

| 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 is not equal to 0 |
| will branch to C if A + B ≠ 0 |
branch if equal (to zero) 001400 Plus offset

<table>
<thead>
<tr>
<th>15</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation: \( PC \leftarrow PC + (2 \times \text{offset}) \) 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:  

\[
\begin{align*}
\text{CMP} & \quad A,B \quad \text{; compare A and B} \\
\text{BEQ} & \quad C \quad \text{; branch if they are equal}
\end{align*}
\]

will branch to C if \( A = B \) \( (A - B = 0) \)

and the sequence

\[
\begin{align*}
\text{ADD} & \quad A,B \quad \text{; add A to B} \\
\text{BEQ} & \quad C \quad \text{; branch if the result = 0}
\end{align*}
\]

will branch to C if \( A + B = 0 \).
BPL

branch if plus 100000 Plus offset

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

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

branch if minus  

100400 Plus offset

---

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

**Condition Codes:**  
Unaffected

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

branch if overflow is clear

102000 Plus offset

Operation:

\[ PC \leftarrow PC + (2 \times \text{offset}) \text{ if } V = 0 \]

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

branch if overflow is set

102400 Plus offset

15 1 0 0 0 0 1 0 1

OFFSET

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

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

branch if carry is clear

103000 Plus offset

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>
| 15 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | OFFSET

**Operation:**

PC ← PC + (2 x offset) if C = 0

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

branch if carry is set

103400 Plus offset

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

OFFSET

Operation: \[ PC \leftarrow PC + (2 \times \text{offset}) \text{ if } C = 1 \]

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

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

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

- **largest**
  - 077777
  - 077776

- **positive**
  - ...
  - 000001
  - 000000
  - 177777
  - 177776

- **negative**
  - ...
  - 100001

- **smallest**
  - 100000

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

- **highest**
  - 177777
  - ...
  - ...
  - ...
  - ...
  - 000002
  - 000001

- **lowest**
  - 000000
branch if greater than or equal  
(to zero)  

002000 Plus offset

<table>
<thead>
<tr>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Operation:**

PC ← PC + (2 x offset) if N v V = 0

**Description:**

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

branch if less than (zero) 002400 Plus offset

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

OFFSET

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

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

branch if greater than (zero) 003000 Plus offset

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

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

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

![Binary Representation]

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

Description: Operation is similar to BLT but in addition will cause a branch if the result of the previous operation was zero.
Unsigned Conditional Branches
The Unsigned Conditional Branches provide a means for testing the result of comparison operations in which the operands are considered as unsigned values.
BHI

branch if higher

101000 Plus offset

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>8</td>
<td>7</td>
<td>OFFSET</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Operation:**

$$PC \leftarrow PC + (2 \times \text{offset}) \text{ if } C = 0 \text{ and } Z = 0$$

**Description:**

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

branch if lower or same

101400 Plus offset

<table>
<thead>
<tr>
<th>15</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>OFFSET</th>
</tr>
</thead>
</table>

**Operation:**

PC ← PC + (2 x offset) if C v Z = 1

**Description:**

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

branch if higher or same

103000 Plus offset

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>8</td>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

OFFSET

Operation: \[ PC \leftarrow PC + (2 \times \text{offset}) \text{ if } C = 0 \]

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

branch if lower 103400 Plus offset

\[
\begin{array}{cccccccc}
1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
\hline
15 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0
\end{array}
\]

**Operation:**
\[
PC \leftrightarrow PC + (2 \times \text{offset}) \text{ if } C = 1
\]

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

jump 0001DD

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>d</th>
<th>d</th>
<th>d</th>
<th>d</th>
<th>d</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation:  PC< (dst)

Condition Codes:  not affected

Description:  JMP provides more flexible program branching than provided with the branch instructions. Control may be transferred to any location in memory (no range limitation) and can be accomplished with the full flexibility of the addressing modes, with the exception of register mode 0. Execution of a jump with mode 0 will cause an "illegal instruction" condition. (Program control cannot be transferred to a register.) Register deferred mode is legal and will cause program control to be transferred to the address held in the specified register. Note that instructions are word data and must therefore be fetched from an even-numbered address. A "boundary error" trap condition will result when the processor attempts to fetch an instruction from an odd address.

Deferred index mode JMP instructions permit transfer of control to the address contained in a selectable element of a table of dispatch vectors.
Subroutine Instructions
The subroutine call in the PDP-11 provides for automatic nesting of subroutines, reentrancy, and multiple entry points. Subroutines may call other subroutines (or indeed themselves) to any level of nesting without making special provision for storage or return addresses at each level of subroutine call. The subroutine calling mechanism does not modify any fixed location in memory, thus providing for reentrancy. This allows one copy of a subroutine to be shared among several interrupting processes. For more detailed description of subroutine programming: see Chapter 5.
**JSR**

jump to subroutine

004RDD

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>d</th>
<th>d</th>
<th>d</th>
<th>d</th>
<th>d</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Operation:</th>
<th>Description:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(tmp) &lt;- (dst)</td>
<td>(tmp is an internal processor register)</td>
</tr>
<tr>
<td>r(SP) &lt;- reg</td>
<td>(push reg contents onto processor stack)</td>
</tr>
<tr>
<td>reg &lt;- PC</td>
<td>(PC holds location following JSR; this address now put in reg)</td>
</tr>
<tr>
<td>PC &lt;- (tmp)</td>
<td>(PC now points to subroutine destination)</td>
</tr>
</tbody>
</table>

**Description:**

In execution of the JSR, the old contents of the specified register (the "LINKAGE POINTER") are automatically pushed onto the processor stack and new linkage information placed in the register. Thus subroutines nested within subroutines to any depth may all be called with the same linkage register. There is no need either to plan the maximum depth at which any particular subroutine will be called or to include instructions in each routine to save and restore the linkage pointer. Further, since all linkages are saved in a reentrant manner on the processor stack execution of a subroutine may be interrupted, the same subroutine reentered and executed by an interrupt service routine. Execution of the initial subroutine can then be resumed when other requests are satisfied. This process (called nesting) can proceed to any level.

A subroutine called with a JSR reg,dst instruction can access the arguments following the call with either autoincrement addressing, (reg) + , (if arguments are accessed sequentially) or by indexed addressing, X(reg), (if accessed in random order). These addressing modes may also be deferred, @ (reg) + and @X(reg) if the parameters are operand addresses rather than the operands themselves.
JSR PC, dst is a special case of the PDP-11 subroutine call suitable for subroutine calls that transmit parameters through the general registers. The SP and the PC are the only registers that may be modified by this call.

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

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

Example:

<table>
<thead>
<tr>
<th>Before:</th>
<th>JSR R5, SBR</th>
</tr>
</thead>
<tbody>
<tr>
<td>(PC) R7</td>
<td>PC</td>
</tr>
<tr>
<td>(SP) R6</td>
<td>n</td>
</tr>
<tr>
<td>R5</td>
<td>#1</td>
</tr>
<tr>
<td>After:</td>
<td>R7</td>
</tr>
<tr>
<td></td>
<td>SBR</td>
</tr>
<tr>
<td>R6</td>
<td>n–2</td>
</tr>
<tr>
<td>R5</td>
<td>PC+2</td>
</tr>
</tbody>
</table>

Stack

DATA 0

#1
**RTS**

return from subroutine

```
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 r r r
```

**Operation:**
- PC ← \text{reg}
- \text{reg} ← (\text{SP}) \downarrow

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

**Example:**

**Before:**

- (PC) R7: SBR
- (SP) R6: \text{n}
- R5: PC

**After:**

- R7: PC
- R6: n+2
- R5: \#1

**Stack:**

- DATA 0
- \#1
**MARK**

**Operation:**
- \( SP \leftarrow SP + 2 \times nn \)
- \( nn = \text{number of parameters} \)
- \( PC \leftarrow R5 \)
- \( R5 \leftarrow (SP) \uparrow \)

**Condition Codes:** unaffected

**Description:** Used as part of the standard PDP-11 subroutine return convention. MARK facilitates the stack clean up procedures involved in subroutine exit. Assembler format is: MARK N

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

At this point the stack is as follows:
And the program is at the address SUB which is the beginning of the subroutine.

SUB:

;execution of the subroutine itself

RTS R5

;the return begins: this causes the contents of R5 to be placed in the PC which then results in the execution of the instruction MARK N. The contents of old PC are placed in R5

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

subtract one and branch (if $\neq 0$)  

077R00 Plus offset

<table>
<thead>
<tr>
<th>15</th>
<th>9</th>
<th>8</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$r$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>r</td>
<td>r</td>
<td>Offset</td>
</tr>
</tbody>
</table>

Operation: $R \leftarrow R - 1 \text{ if this result } \neq 0 \text{ then } PC \leftarrow PC - (2 \times \text{offset})$

Condition Codes: unaffected

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

SOB $R, A$

Where $A$ is the address to which transfer is to be made if the decremented $R$ is not equal to 0. Note that the SOB instruction cannot be used to transfer control in the forward direction.
Traps
Trap instructions provide for calls to emulators, I/O monitors, debugging packages, and user-defined interpreters. A trap is effectively an interrupt generated by software. When a trap occurs the contents of the current Program Counter (PC) and Program Status Word (PS) are pushed onto the processor stack and replaced by the contents of a two-word trap vector containing a new PC and new PS. The return sequence from a trap involves executing an RTI or RTT instruction which restores the old PC and old PS by popping them from the stack. Trap vectors are located at permanently assigned fixed addresses.
emulator trap

104000—104377

Operation:

\[ \begin{array}{c}
\downarrow (SP) \rightarrow PS \\
\downarrow (SP) \rightarrow PC \\
PC \rightarrow (30) \\
PS \rightarrow (32)
\end{array} \]

Condition Codes:

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

Description:

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

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

Before:

<table>
<thead>
<tr>
<th>PS</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DATA 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R7, PC</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R6, SP</th>
<th>n—4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After:

<table>
<thead>
<tr>
<th>PS</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(32)</td>
<td>DATA 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PC</th>
<th>PS 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(30)</td>
<td>PC 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>n—4</td>
<td></td>
</tr>
</tbody>
</table>

4-65
TRAP

trap

104400—104777

Operation:

\[ \uparrow (SP) \leftarrow PS \]
\[ \uparrow (SP) \leftarrow PC \]
\[ PC \leftarrow (34) \]
\[ PS \leftarrow (36) \]

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.
breakpoint trap

000003

Operation:
\[ \downarrow (SP) \rightarrow PS \]
\[ \downarrow (SP) \rightarrow PC \]
\[ PC \leftarrow (14) \]
\[ PS \leftarrow (16) \]

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.
(no information is transmitted in the low byte.)
IOT

input/output trap

Operation:

\[ \n(SP) \rightarrow PS \\
(SP) \rightarrow PC \\
PC \rightarrow (20) \\
PS \rightarrow (22) \]

Condition Codes:

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

Description:

Performs a trap sequence with a trap vector address of 20. Used to call the I/O Executive routine IOX in the paper tape software system, and for error reporting in the Disk Operating System.

(no information is transmitted in the low byte)
RTI

return from interrupt 000002

| 15 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 |

Operation: \[\text{PC} \leftarrow (SP)\]
\[\text{PS} \leftarrow (SP)\]

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

Description: Used to exit from an interrupt or TRAP service routine. The PC and PS are restored (popped) from the processor stack.
RTT

return from interrupt

Operation:  
PC ← (SP) ↑
PS ← (SP) ↑

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

Description: This is the same as the RTI instruction except that it inhibits a trace trap, while RTI permits a trace trap. If a trace trap is pending, the first instruction after the RTT will be executed prior to the next "T" trap. In the case of the RTI instruction the "T" trap will occur immediately after the RTI.
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

Bus Error Traps: Bus Error Traps are:

1. Boundary Errors: attempts to reference instructions or word operands at odd addresses.

2. Time-Out Errors: attempts to reference addresses on the bus that made no response within 15μs in the PDP-11/40. 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 PS 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.
9. The traced instruction was a Return from Trap

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 interrupts may be acknowledged immediately after the loading of the new PC and PS at the trap vector location. To lock out all interrupts, the PS at the trap vector should raise the processor priority to level 7.

A WAIT: The trap occurs immediately.

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.

A Return from Trap: The return from trap instruction either clears or sets the T bit. It inhibits the trace trap. If the T-bit was set and RTT is the traced instruction the trap is delayed until completion of the next instruction.

Power Failure Trap: is a standard PDP-11 feature. Trap occurs whenever the AC power drops below 95 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):

Odd Address
Fatal Stack Violation
Memory Management Violation
Timeout
Trap Instructions
Trace Trap
Warning Stack Violation
Power Failure

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

halt

\[ \begin{array}{ccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
\end{array} \]

Condition Codes: not affected

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

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

wait for interrupt 000001

<table>
<thead>
<tr>
<th>Condition Codes:</th>
<th>not affected</th>
</tr>
</thead>
</table>

**Description:** Provides a way for the processor to relinquish use of the bus while it waits for an external interrupt. Having been given a WAIT command, the processor will not compete for bus use by fetching instructions or operands from memory. This permits higher transfer rates between a device and memory, since no processor-induced latencies will be encountered by bus requests from the device. In WAIT, as in all instructions, the PC points to the next instruction following the WAIT operation. Thus when an interrupt causes the PC and PS to be pushed onto the processor stack, the address of the next instruction following the WAIT is saved. The exit from the interrupt routine (i.e. execution of an RTI instruction) will cause resumption of the interrupted process at the instruction following the WAIT.
RESET

reset external bus

Condition Codes: not affected

Description: Sends INIT on the UNIBUS for 10 ms. All devices on the UNIBUS are reset to their state at power up.
move from previous instruction space

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\
6 & 5 & s & s & s & s & s & s
\end{array}
\]

Operation: \((\text{temp}) \leftarrow (\text{src})\)
\[\land (\text{SP}) \leftarrow (\text{temp})\]

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

Description: This instruction is provided in order to allow inter-address space communication when the PDP-11/40 is using the Memory Management unit. The address of the source operand is determined in the current address space. That is, the address is determined using the SP and memory segments determined by PS (bits 15, 14). The address itself is then used in the previous mode (as determined by PS (bits 13, 12) to get the source operand). This operand is then pushed on to the current R6 stack.
MTPI (Memory Management option)

move to previous instruction space

0066DD

Operation: \((\text{temp})\uparrow(\text{SP})\uparrow\)
\((\text{dst})\uparrow(\text{temp})\)

Condition Codes:

- N: set if the source \(<0\); otherwise cleared
- Z: set if the source \(=0\); otherwise cleared
- V: cleared
- C: unaffected

Description: The address of the destination operand is determined in the current address space. MTPI then pops a word off the current stack and stores that word in the destination address in the previous mode's (bits 13, 12 of PS).
**Condition Code Operators**

<table>
<thead>
<tr>
<th>CLN</th>
<th>SEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLZ</td>
<td>SEZ</td>
</tr>
<tr>
<td>CLV</td>
<td>SEV</td>
</tr>
<tr>
<td>CLC</td>
<td>SEC</td>
</tr>
<tr>
<td>CCC</td>
<td>SCC</td>
</tr>
</tbody>
</table>

**condition code operators**

<table>
<thead>
<tr>
<th>0002XX</th>
</tr>
</thead>
</table>

**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**

<table>
<thead>
<tr>
<th>Operation</th>
<th>OP Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLC</td>
<td>Clear C</td>
</tr>
<tr>
<td>CLV</td>
<td>Clear V</td>
</tr>
<tr>
<td>CLZ</td>
<td>Clear Z</td>
</tr>
<tr>
<td>CLN</td>
<td>Clear N</td>
</tr>
<tr>
<td>SEC</td>
<td>Set C</td>
</tr>
<tr>
<td>SEV</td>
<td>Set V</td>
</tr>
<tr>
<td>SEZ</td>
<td>Set Z</td>
</tr>
<tr>
<td>SEN</td>
<td>Set N</td>
</tr>
<tr>
<td>SCC</td>
<td>Set all CC's</td>
</tr>
<tr>
<td>CCC</td>
<td>Clear all CC's</td>
</tr>
<tr>
<td></td>
<td>Clear V and C</td>
</tr>
<tr>
<td>NOP</td>
<td>No Operation</td>
</tr>
</tbody>
</table>

Combinations of the above set or clear operations may be ORed together to form combined instructions.
CHAPTER 5

PROGRAMMING TECHNIQUES

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

5.1 THE STACK

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

![Figure 5-1: Stack Addresses](image)

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

![Figure 5-2: Word and Byte Stacks](image)

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

This operation is accomplished as follows:

\[ \text{MOV Source,}-(\text{SP}) \] ;MOV Source Word onto the stack
or

\[ \text{MOVB Source,}-(\text{SP}) \] ;MOVB Source Byte onto the stack

This is called a "push" because data is "pushed onto the stack."
To remove an item from stack the autoincrement addressing mode with the appropriate SP is employed. This is accomplished in the following manner:

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

or

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

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

![Workflow Diagram](image)

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:

<table>
<thead>
<tr>
<th>Address</th>
<th>Octal Code</th>
<th>Assembler Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>076322</td>
<td>010167</td>
<td>SUBR: MOV R1, TEMP1 ; save R1</td>
</tr>
<tr>
<td>076324</td>
<td>000074</td>
<td></td>
</tr>
<tr>
<td>076326</td>
<td>010267</td>
<td>MOV R2, TEMP2 ; save R2</td>
</tr>
<tr>
<td>076330</td>
<td>000072</td>
<td></td>
</tr>
<tr>
<td>076410</td>
<td>016701</td>
<td>MOV TEMP1, R1 ; Restore R1</td>
</tr>
<tr>
<td>076412</td>
<td>000006</td>
<td></td>
</tr>
<tr>
<td>076414</td>
<td>016702</td>
<td>MOV TEMP2, R2 ; Restore R2</td>
</tr>
<tr>
<td>076416</td>
<td>000002</td>
<td></td>
</tr>
<tr>
<td>076420</td>
<td>000207</td>
<td>RTS PC</td>
</tr>
<tr>
<td>076422</td>
<td>000000</td>
<td>TEMP1: 0</td>
</tr>
<tr>
<td>076424</td>
<td>000000</td>
<td>TEMP2: 0</td>
</tr>
</tbody>
</table>

**Index Constants**

**Figure 5-4: Register Saving Without the Stack**

**OR: Using the Stack**

<table>
<thead>
<tr>
<th>Address</th>
<th>Octal Code</th>
<th>Assembler Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>010020</td>
<td>010143</td>
<td>SUBR: MOV R1, -(R3) ; push R1</td>
</tr>
<tr>
<td>010022</td>
<td>010243</td>
<td>MOV R2, -(R3) ; push R2</td>
</tr>
<tr>
<td>010130</td>
<td>012301</td>
<td>MOV (R3) + . R2 ; pop R2</td>
</tr>
<tr>
<td>010132</td>
<td>012302</td>
<td>MOV (R3) + . R1 ; pop R1</td>
</tr>
<tr>
<td>010134</td>
<td>000207</td>
<td>RTS PC</td>
</tr>
</tbody>
</table>

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

**Figure 5-5: Register Saving using the Stack**

The second routine uses four less words of instruction code and two words of temporary "stack" storage. Another routine could use the same stack space at some later point. Thus, the ability to share temporary storage in the form of a stack is a very economical way to save on memory usage.
As a further example of stack usage, consider the task of managing an input buffer from a terminal. As characters come in, the terminal user may wish to delete characters from his line; this is accomplished very easily by maintaining a byte stack containing the input characters. Whenever a backspace is received a character is “popped” off the stack and eliminated from consideration. In this example, a programmer has the choice of “popping” characters to be eliminated by using either the MOVX (MOVE BYTE) or INC (INCREMENT) instructions.

![Figure 5-6: Byte Stack used as a Character Buffer](image)

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

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

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

\[
\text{JSR } R, \text{SUBR}
\]

\[
\text{a general register (R) for linkage}
\]

\[
\text{an entry location (SUBR) for the subroutine}
\]
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.

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembler Syntax</th>
<th>Octal Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000</td>
<td>JSR R5, SUBR</td>
<td>004567</td>
</tr>
<tr>
<td>001002</td>
<td>index constant for SUBR 000064</td>
<td></td>
</tr>
<tr>
<td>001064</td>
<td>SUBR MOV A, B</td>
<td>0nn1mm</td>
</tr>
</tbody>
</table>

**Figure 5-7: JSR using R5**

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 addresses 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
... 
020306  SUBR MOV (R5) + , R1 ; get arg #1
020310  MOV (R5) + , R2 ; get arg #2 Retrieve Arguments from SUB
```

**Figure 5-9: Argument Transmission - Register Autoincrement Mode**
Figure 5-10: Argument Transmission-Register Autoincrement Deferred Mode

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

Calling Program:

```
MOV POINTER, R1
JSR PC, SUBR
```

SUBROUTINE

```
ADD (R1) + , (R1) ; Add item #1 to item #2, place result in item #2, R1 points to item #2 now

etc.
```

or

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

etc.
```

Figure 5-11: Transmitting Stacks as Arguments
Because the PDP-11 hardware already uses general purpose register R6 to point to a stack for saving and restoring PC and PS (processor status word) information, it is quite convenient to use this same stack to save and restore intermediate results and to transmit arguments to and from subroutines. Using R6 in this manner permits extreme flexibility in nesting subroutines and interrupt service routines.

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

![Figure 5-12: Shifting Indexed Base](image)

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"](image)

5-8
5.2.3 Subroutine Return
In order to provide for a return from a subroutine to the calling program an RTS instruction is executed by the subroutine. This instruction should specify the same register as the JSR used in the subroutine call. When executed, it causes the register specified to be moved to the PC and the top of the stack to be then placed in the register specified. Note that if an RTS PC is executed, it has the effect of returning to the address specified on the top of the stack.

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

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

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

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

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

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

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

5.3 INTERRUPTS
5.3.1 General Principles
Interrupts are in many respects very similar to subroutine calls. However, they are forced, rather than controlled, transfers of program execution occurring because of some external and program-independent event (such as a stroke on the teleprinter keyboard). Like subroutines, interrupts have linkage information such
that a return to the interrupted program can be made. More information is actually necessary for an interrupt transfer than a subroutine transfer because of the random nature of interrupts. The complete machine state of the program immediately prior to the occurrence of the interrupt must be preserved in order to return to the program without any noticeable effects. (i.e. was the previous operation zero or negative, etc.) This information is stored in the Processor Status Word (PS). Upon interrupt, the contents of the Program Counter (PC) (address of next instruction) and the PS are automatically pushed onto the R6 system stack. The effect is the same as if:

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

had been executed.

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

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

### 5.3.2 Nesting

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

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

2. Interrupt stops process 0 with PC = PC0, and status = PS0; starts process 1.
3. Process 1 uses stack for temporary storage (TE0, TE1).

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

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

6. Subroutine A is running and uses stack for temporary storage.
7. Subroutine A releases the temporary storage holding TA1 and TA2.

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

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

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

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

Figure 5.14: Nested Interrupt Service Routines and Subroutines

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

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

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

![Diagram of PDP-11 Approach and Conventional Approach]

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.
1. Task A has requested processing by Reentrant Routine Q.

2. Task A temporarily relinquishes control (is interrupted) of Reentrant Routine Q before it finishes processing.

3. Task B starts processing in the same copy of Reentrant Routine Q.

4. Task B relinquishes control of Reentrant Routine Q at some point in its processing.

5. Task A regains control of Reentrant Routine Q and resumes processing from where it stopped.

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

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

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

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

In the case of a reentrant program which is used in a multi-programming environment it is usually necessary to maintain a separate R6 stack for each user although each such stack would be shared by all the tasks of a given user. For example, if a reentrant FORTRAN compiler is to be shared between many users, each time the user is changed, R6 would be set to point to a new user's stack area as illustrated in Figure 5.17.

![Figure 5.17: Multiple R6 Stack](image)

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 specific 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 CO-ROUTINES

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

Routine #1 is operating, it then executes:

\[
\text{MOV } \#\text{PC2, -(R6)}
\]

\[
\text{JSR PC, @(R6) +}
\]

with the following results:

1) PC2 is popped from the stack and the SP autoincremented
2) SP is autodecremented and the old PC (i.e. PC1) is pushed
3) control is transferred to the location PC2 (i.e. routine #2)

Routine #2 is operating, it then executes:

\[
\text{JSR PC, @(R6) +}
\]

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

Figure 5-18 - Co-Routine Interaction
5.7 MULTI-PROGRAMMING
The PDP 11/40's architecture with its two modes of operation and its Memory Management provides an ideal environment for multi-programming systems.

In any multi-programming system there must be some method of transferring information and control between programs operating in the same or different modes. The PDP 11/40 provides the user with these communication paths.

5.7.1 Control Information
Control is passed inwards (User to Kernel) by all traps and interrupts. All trap and interrupt vectors are located in Kernel virtual space. Thus all traps and interrupts pass through Kernel space to pick up their new PC and PS and determine the new mode of processing.

Control is passed outwards (Kernel to User) by the RTI and RTT instructions.

5.7.2 Data
Data is transferred between modes by two instructions: Move From Previous Instruction space (MFPI) and Move To Previous Instruction space (MTPI). The instructions are fully described in Chapter 4. However, it should be noted that these instructions have been designed to allow data transfers to be under the control of the inner mode (Kernel) program and not the outer, thus providing protection of an inner program from an outer.

5.7.3 Processor Status Word
The PDP 11/40 protects the PS from implicit references by User programs which could result in damage to an inner level program.

A program operating in Kernel mode can perform any manipulation of the PS. Programs operating at the outer level are inhibited from changing bits 5-7 (the Processor's Priority). They are also restricted in their treatment of bits 15, 14 (Current Mode), and bits 13, 12 (Previous Mode) these bits may only be set, they are only cleared by an interrupt or trap.

Thus, a programmer can pass control outwards through the RTI and RTT instructions to set bits in the mode fields of his PS. To move inwards, however, bits must be cleared and he must therefore, issue a trap or interrupt.

The Kernel can further protect the PS from explicit references (Move data to location 777776-the PS) through Memory Management.
CHAPTER 6

MEMORY MANAGEMENT

The PDP-11/40 Memory Management Unit provides the hardware facilities necessary for complete memory management and protection. It is designed to be a memory management facility for systems where the system memory size is greater than 28K words and for multi-user, multi-programming systems where memory protection and relocation facilities are necessary.

In order to most effectively utilize the power efficiency of the PDP-11/40 in medium and large scale systems it is necessary to run several programs simultaneously. In such multi-programming environments several user programs would be resident in memory at any given time. The task of the supervisory program would be: control the execution of the various user programs, manage the allocation of memory and peripheral device resources, and safeguard the integrity of the system as a whole by careful control of each user program.

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

The basic characteristics of the PDP-11/40 Memory Management Unit are:

- 8 User mode memory pages
- 8 Kernel mode memory pages
- 8 pages in each mode for instructions and data
- page length from 32 to 4096 words
- each page provided with full protection and relocation
- transparent operation
- 3 modes of memory access control
- memory extension to 124K words (248K bytes)

6.1 PDP-11 FAMILY BASIC ADDRESSING LOGIC

The addresses generated by all PDP-11 Family Central Processor Units (CPUs) are 18-bit direct byte addresses. Although the PDP-11 Family word length and operational logic is all 16-bit length, the UNIBUS and CPU addressing logic actually is 18-bit length. Thus, while the PDP-11 word can only contain address references up to 32K words (64K bytes) the CPU and UNIBUS can reference addresses up to 128K words (256K bytes). These extra two bits of addressing logic provide the basic framework for expanded memory paging.
In addition to the word length constraint on basic memory addressing space, the uppermost 4K words of address space is always reserved for UNIBUS I/O device registers. In a basic PDP-11/40 memory configuration (without Management) all address references to the uppermost 4K words of 16-bit address space (170000-177777) are converted to full 18-bit references with bits 17 and 16 always set to 1. Thus, a 16-bit reference to the I/O device register at address 173224 is automatically converted to a full 18-bit reference to the register at address 773224. Accordingly, the basic PDP-11/40 configuration can directly address up to 28K words of true memory, and 4K words of UNIBUS I/O device registers. Memory configurations beyond this require the PDP-11/40 Memory Management Unit.

6.2 VIRTUAL ADDRESSING
When the PDP-11/40 Memory Management Unit is operating, the normal 16-bit direct byte address is no longer interpreted as a direct Physical Address (PA) but as a Virtual Address (VA) containing information to be used in constructing a new 18-bit physical address. The information contained in the Virtual Address (VA) is combined with relocation and description information contained in the Active Page Register (APR) to yield an 18-bit Physical Address (PA). Memory can be dynamically allocated in pages each composed of from 1 to 128 blocks of 32 words.

![Virtual Address Mapping into Physical Address](image)

The starting address for each page is an integral multiple of 32 words, and has a maximum size of 4096 words. Pages may be located anywhere within the 128K Physical Address space. The determination of which set of 8 pages registers is used to form a Physical Address is made by the current mode of operation of the CPU, i.e. Kernel or User mode.

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

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

6.4 CONSTRUCTION OF A PHYSICAL ADDRESS
The basic information needed for the construction of a Physical Address (PA) comes from the Virtual Address (VA), which is illustrated in Figure 6-2, and the appropriate APR set.

![Figure 6-2 Interpretation of a Virtual Address](image)

The Virtual Address (VA) consists of:

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

2. The Displacement Field (DF). This 13-bit field contains an address relative to the beginning of a page. This permits page lengths up to 4K words ($2^{13} = 8K$ bytes). The DF is further subdivided into two fields as shown in Figure 6-3.

![Figure 6-3 Displacement Field of Virtual Address](image)

The Displacement Field (DF) consists of:

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

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

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

6-3
The formation of a physical address takes 150 ns.

The formation of the Physical Address is illustrated in Figure 6-4.

![Figure 6-4 Construction of a Physical Address]

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

1. Select a set of Active Page Registers depending on current mode.
2. The Active Page Field of the Virtual Address is used to select an Active Page Register (APRO-APR7).
3. The Page Address Field of the selected Active Page Register contains the starting address of the currently active page as a block number in physical memory.
4. The Block Number from the Virtual Address is added to the block number from the Page Address Field to yield the number of the block in physical memory which will contain the Physical Address being constructed.
5. The Displacement in Block from the Displacement Field of the Virtual Address is joined to the Physical Block Number to yield a true 18-bit PDP-11/40 Physical Address.

### 6.5 MANAGEMENT REGISTERS

The PDP-11/40 Memory Management Unit uses two sets of eight 32-bit Active Page Registers. An APR is actually a pair of 16-bit registers: a Page Address Register (PAR) and a Page Descriptor Register (PDR). These registers are always used as a pair and contain all the information needed to describe and locate the currently active memory pages.

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

The various Memory Management Registers are located in the uppermost 4K of PDP-11 physical address space along with the UNIBUS I/O device registers.
6.5.1 Page Address Registers
The Page Address Register is the first word of the 32-bit Active Page Register; it contains the Page Address Field, a 12-bit field, which specifies the starting address of the page as a block number in physical memory.

6.5.2 Page Descriptor Register
The Page Descriptor Register contains information relative to page expansion, length, and access control.
Access Control Field (ACF)
This 2-bit field, occupying bits 2-1 of the Page Descriptor Register contains the access rights to this particular segment. The access codes or “keys” specify the manner in which a page may be accessed and whether or not a given access should result in an abort of the current operation. A memory reference which causes an abort is not completed. Aborts are used to catch “missing page faults,” prevent illegal accesses, etc.

In the context of access control the term “write” is used to indicate the action of any instruction which modifies the contents of any addressable word. Except in those cases where references are made to the 4K word UNIBUS I/O register area, a “write” is synonymous with what is usually called a “store” or “modify” in many computer systems.

The modes of access control are as follows:

<table>
<thead>
<tr>
<th>ACF</th>
<th>Key</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>non-resident abort all accesses</td>
</tr>
<tr>
<td>01</td>
<td>2</td>
<td>read only abort on write attempt</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>(unused) abort all accesses</td>
</tr>
<tr>
<td>11</td>
<td>6</td>
<td>read/write no system abort action</td>
</tr>
</tbody>
</table>

Access Information Bits
W Bit (bit 6)—This bit indicates whether or not this page has been modified (i.e. written into) since the PSR was loaded. (W = 1 is Affirmative) The W Bit is useful in applications which involve disk swapping and memory overlays. It is used to determine which pages have been modified and hence must be saved in their new form and which pages have not been modified and can be simply overlayed.

Note that the W bit is reset to 0 whenever the Active Page Register (either PAR or PDR) is modified (written into).

Expansion Direction (ED)
This one-bit field, located at bit 3 of the Page Descriptor Register, specifies whether the segment expands upward from relative zero (ED = 0) or downwards toward relative zero (ED = 1). Relative zero, in this case, is the PAF. Expansion is done by changing the Page Length Field. In expanding upwards, blocks with higher relative addresses are added; in expanding downwards, blocks with lower relative addresses are added to the page. Upward expansion is usually used to add more program space, while downward expansion is used to add more stack space.

Page Length Field (PLF)
The seven-bit field, occupying bits 14-8 of the Page Descriptor Register, specifies the number of blocks in the page. A page consists of at least one and at most 128 blocks, and occupies contiguous core locations. If the page expands upwards, this field contains the length of the page minus one (in blocks). If the page expands downwards, this field contains 128 minus the length of the page (in blocks).
A Page Length Error occurs when the Block Number of the virtual address is greater than the Page Length Field, if the segment expands upwards, or if the page expands downwards, when the BN is less than the PLF.

**Reserved Bits**

Bits 15, 4 and 5 are reserved for future use, and are always 0. Bits 7 and 0 are used by the PDP-11/45, and in the PDP-11/40 they are set to 0.

**6.6 FAULT REGISTERS**

Aborts generated by the hardware are vectored through Kernel virtual location 250. Status Registers #0 and #2 (#1 is used by the PDP-11/45) are used to determine why the abort occurred. Note that an abort to a location which is itself an invalid address will cause another abort. Thus the Kernel program must insure that Kernel Virtual Address 10 is mapped into a valid address, otherwise a loop will occur which will require console intervention.

**6.6.1 Status Register #0 (SR0) (status and error indicators)**

SR0 contains error flags, the page number whose reference caused the abort, and various other status flags. The register is organized as shown in Figure 6-8.

![Figure 6-8 Format of Status Register #0 (SR0)](image)

Bits 15-13 when set (error conditions) cause Memory Management to freeze the contents of bits 1-7 and Status Register #2.

Note that Status Register #0 (SR0) bits 0, and 8 can be set under program control to provide meaningful page control information. However, information written into all other bits is not meaningful. Only that information which is automatically written into these remaining bits as a result of hardware actions is useful as a monitor of the status of the Memory Management Unit. Setting bits 15-13 under program control will not cause traps to occur; these bits however must be reset to 0 after an abort has occurred in order to resume page status monitoring.

**Abort—Non-Resident**

Bit 15 is the "Abort—Non-Resident" bit. It is set by attempting to access
a page with an Access Code Field key equal to 0 or 4. It is also set by attempting to use Memory Management with a mode of 1 or 2.

Abort—Page Length
Bit 14 is the "Abort—Page Length" bit. It is set by attempting to access a location in a page with a block number (Virtual Address bits 12-6) that is outside the area authorized by the Page Length Field of the Active Page Register for that page. Bits 14 and 15 may be set simultaneously by the same access attempt.

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

Maintenance/Designation Mode
Bit 8 specifies Maintenance use of the Memory Management Unit. It is provided for diagnostic purposes only.

Mode
Bits 5, 6 indicate the CPU mode (User/Kernel) associated with the page causing the abort. (Kernel = 00, User = 11). If an illegal mode is specified, management will abort and set bit 15.

Page Number
Bits 3-1 contain the page number of a reference causing a fault. Note that pages, like blocks, are numbered from 0 upwards.

Enable Management
Bit 0 is the "Enable Management" bit. When it is set to 1, all addresses are relocated by the Management unit. When bit 0 is set to 0 the Unit is inoperative and addresses are not relocated or protected.

6.6.2 Status Register #2
SR2 is loaded with the 16-bit Virtual Address at the beginning of each instruction fetch. SR2 is Read-Only; it can not be written, SR2 is the Virtual Address Program Counter.
CHAPTER 7
INTERNAL PROCESSOR OPTIONS

7.1 GENERAL
This chapter describes 3 options which mount in the Central Processor, assembly unit. The Extended Instruction Set (EIS) option allows extended manipulation of fixed point numbers. The Floating Point option (which requires the EIS option) enables direct operations on single precision 32-bit words. The Stack Limit option allows dynamic adjustment of the lower boundary of permissible stack addresses.

The options are contained on individual modules that plug into dedicated, prewired slots.

KE11-E   EIS option
KE11-F   Floating Point option
KJ11-A   Stack Limit option

The basic processor timing is not degraded, and NPR latency is not affected by the use of these options.

7.2 EIS OPTION
The Extended Instruction Set option adds the following instruction capability:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL</td>
<td>multiply</td>
<td>070RSS</td>
</tr>
<tr>
<td>DIV</td>
<td>divide</td>
<td>071RSS</td>
</tr>
<tr>
<td>ASH</td>
<td>shift arithmetically</td>
<td>072RSS</td>
</tr>
<tr>
<td>ASHC</td>
<td>arithmetic shift combined</td>
<td>073RSS</td>
</tr>
</tbody>
</table>

The EIS instructions are directly compatible with the larger 11 computer, the PDP-11/45. The detailed operation of these instructions is covered in Chapter 4.

The number formats are:

16-bit single word:

\[
\begin{array}{c}
\text{S} \\
\text{NUMBER}
\end{array}
\]

32-bit double word:

\[
\begin{array}{c}
\text{S} \\
\text{HIGH NUMBER PART} \\
\text{LOW NUMBER PART}
\end{array}
\]

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

Interrupts are serviced at the end of an EIS instruction.
7-3 FLOATING POINT OPTION
The Floating Point instructions used with this option are unique to the PDP-11/40. However, the Op Codes used do not conflict with any other instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>FADD</td>
<td>floating add</td>
<td>07500R</td>
</tr>
<tr>
<td>FSUB</td>
<td>floating subtract</td>
<td>07501R</td>
</tr>
<tr>
<td>FMUL</td>
<td>floating multiply</td>
<td>07502R</td>
</tr>
<tr>
<td>FDIV</td>
<td>floating divide</td>
<td>07503R</td>
</tr>
</tbody>
</table>

The number format is:

![Diagram of Floating Point Number Format]

S = sign of fraction; 0 for positive, 1 for negative
Exponent = 8 bits for the exponent, in excess (200), notation
Fraction = 23 bits plus 1 hidden bit (all numbers are assumed to be normalized)

The number format is essentially a sign and magnitude representation. The format is identical with the 11/45 for single precision numbers.

**Fraction**
The binary radix point is to the left (in front of bit 6 of the High Argument), so that the value of the fraction is always less than 1 in magnitude. Normalization would always cause the first bit after the radix point to be a 1, such that the fractional value would be between 1/2 and 1. Therefore, this bit can be understood and not be represented directly, to achieve an extra 1 bit of resolution.

The first bit to the right of the radix point (hidden bit) is always a 1. The next bit for the fraction is taken from bit 6 of the High Argument. The result of a Floating Point operation is always rounded away from zero, increasing the absolute value of the number.

**Exponent**
The 8-bit Exponent field (bits 14 to 7) allow exponent values between -128 and +127. Since an excess (200), or (128),0 number system is used, the correspondence between actual values and coded representation is as follows:

<table>
<thead>
<tr>
<th>Actual Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>Octal</td>
</tr>
<tr>
<td>+127</td>
<td>377</td>
</tr>
<tr>
<td>+1</td>
<td>201</td>
</tr>
<tr>
<td>0</td>
<td>200</td>
</tr>
<tr>
<td>−1</td>
<td>177</td>
</tr>
<tr>
<td>−128</td>
<td>000</td>
</tr>
</tbody>
</table>
If the actual value of the exponent is equal to \(-128\), meaning a total value (including the fraction) of less than \(2^{-128}\), the floating point number will be assumed to be 0, regardless of the sign or fraction bits. The hardware will generate a clean 0 (a 32-bit word of all zeros).

**Example of a Number**

\[ +(12)_o = +(1100)_2, \]

\[ = +(2^4)_o \times (.11)_2 \]

\[ [16 \times (\frac{1}{2} + \frac{1}{4}) = 12] \]

<table>
<thead>
<tr>
<th>S</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10 000 100</td>
<td>1000000</td>
</tr>
</tbody>
</table>

hidden bit is a 1
radix point is understood

**Registers**

There are no pre-assigned registers for the Floating Point option. A general purpose register is used as a pointer to specify a stack address. The contents of the register are used to locate the operands and answer for the Floating Point operations as follows:

\[(R) = \text{High B argument address}\]

\[(R)+2 = \text{Low B argument address}\]

\[(R)+4 = \text{High A argument address}\]

\[(R)+6 = \text{Low A argument address}\]

After the Floating Point operation, the answer is stored on the stack as follows:

\[(R)+4 = \text{address for High part of answer}\]

\[(R)+6 = \text{address for Low part of answer}\]

where \((R)\) is the original contents of the general register used.

After execution of the instruction, the general register will point to the High answer, at \((R)+4\).

**Condition Codes**

Condition codes are set or cleared as shown in the Instruction Descriptions, in the next part of this section. If a trap occurs as a function of a Floating Instruction, the condition codes are re-interpreted as follows:

\[ V = 1, \text{ if an error occurs} \]

\[ N = 1, \text{ if underflow or divide-by-zero} \]

\[ C = 1, \text{ if divide by zero} \]

\[ Z = 0 \]

<table>
<thead>
<tr>
<th>V</th>
<th>N</th>
<th>C</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overflow</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Underflow</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Divide by 0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

7-3
Traps occur through the vector at location 244. A Floating Point instruction will be aborted if a BR request is issued before the instruction is within approximately 8 μsec of completion. The Program Counter will point to the aborted Floating instruction so that the Interrupt will look transparent.

INSTRUCTIONS

**FADD**

floating add 07500R

```
0 1 1 1 1 0 1 0 0 0 0 0 0 r r r
15  3  2  0
```

**Operation:** 

\[ [(R)+4, (R)+6] \leftarrow [(R)+4, (R)+6] + [(R),(R)+2], \text{ if result } \geq 2^{-128}; \text{ else } [(R)+4, (R)+6] \leftarrow 0 \]

**Condition Codes:**

- **N:** set if result < 0; cleared otherwise
- **Z:** set if result = 0; cleared otherwise
- **V:** cleared
- **C:** cleared

**Description:**

Adds the A argument to the B argument and stores the result in the A Argument position on the stack. General register R is used as the stack pointer for the operation.

\[ A \leftarrow A + B \]

**FSUB**

floating subtract 07501R

```
0 1 1 1 1 0 1 0 0 0 0 0 1 r r r
15  3  2  0
```

**Operation:**

\[ [(R)+4, (R)+6] \leftarrow [(R)+4, (R)+6] - [(R),(R)+2], \text{ if result } \geq 2^{-128}; \text{ else } [(R)+4, (R)+6] \leftarrow 0 \]

**Condition Codes:**

- **N:** set if result < 0; cleared otherwise
- **Z:** set if result = 0; cleared otherwise
- **V:** cleared
- **C:** cleared

**Description:**

Subtracts the B Argument from the A Argument and stores the result in the A Argument position on the stack.

\[ A \leftarrow A - B \]
**FMUL**

floating multiply

07502R

![Opcode Diagram](image)

**Operation:**

\[(R)+4, (R)+6\] → \[(R)+4, (R)+6\] × \[(R), (R)+2\] if result \(\geq 2^{-128}\); else \[(R)+4, (R)+6\] → 0

**Condition Codes:**

- N: set if result \(< 0\); cleared otherwise
- Z: set if result = 0; cleared otherwise
- V: cleared
- C: cleared

**Description:**

Multiplies the A Argument by the B Argument and stores the result in the A Argument position on the stack.

\(A \leftarrow A \times B\)

**FDIV**

floating divide

070503R

![Opcode Diagram](image)

**Operation:**

\[(R)+4, (R)+6\] → \[(R)+4, (R)+6\] / \[(R), (R)+2\] if result \(\geq 2^{-128}\); else \[(R)+4, (R)+6\] → 0

**Condition Codes:**

- N: set if result \(< 0\); cleared otherwise
- Z: set if result = 0; cleared otherwise
- V: cleared
- C: cleared

**Description:**

Divides the A Argument by the B Argument and stores the result in the A Argument position on the stack. If the divisor (B Argument) is equal to zero, the stack is left untouched.

\(A \leftarrow A / B\)

### 7.4 STACK LIMIT OPTION

This option allows program control of the lower limit for permissible stack addresses. The limit may be varied in increments of (400), bytes or (200), words.

There is a Stack Limit Register, with the following format:

![Stack Limit Register Diagram](image)
The Stack Limit Register can be addressed as a word at location 777774, or as a byte at location 777775. The register is accessible to the processor and console, but not to any bus device.

The 8 bits, 15 through 8, contain the stack limit information. These bits are cleared by System Reset, Console Start, or the RESET instruction. The lower 8 bits are not used. Bit 8 corresponds to a value of \((400)_{10}\), or \((256)_{10}\).

**Stack Limit Violations**

When instructions cause a stack address to exceed (go lower than) a limit set by the programmable Stack Limit Register, a Stack Violation occurs. There is a Yellow Zone (grace area) of 16 words below the Stack Limit which provides a warning to the program so that corrective steps can be taken. Operations that cause a Yellow Zone Violation are completed, then a bus error trap is effected. The error trap, which itself uses the stack, executes without causing an additional violation, unless the stack has entered the Red Zone.

A Red Zone Violation is a Fatal Stack Error. (Odd stack or non-existent stack are the other Fatal Stack Errors.) When detected, the operation causing the error is aborted, the stack is repositioned to address 4, and a bus error occurs. The old PC and PS are pushed into locations 0 and 2, and the new PC and PS are taken from locations 4 and 6.

**Stack Limit Addresses**

The contents of the Stack Limit Register (SL) are compared to the stack address to determine if a violation has occurred. The least significant bit of the register (bit 8) has a value of \((400)_{10}\). The determination of the violation zones is as follows:

- **Yellow Zone** \(\equiv (SL) + (340 \text{ through } 377)\), execute, then trap
- **Red Zone** \(\leq (SL) + (337)\), abort, then trap to location 4

If the Stack Limit Register contents were zero:
- **Yellow Zone** \(\equiv 340 \text{ through } 377\)
- **Red Zone** \(\equiv 000 \text{ through } 337\)
CHAPTER 8

CONSOLE OPERATION

8.1 CONSOLE ELEMENTS
The PDP-11/40 Operator's Console provides the following facilities:

Power Switch (with a key lock)
ADDRESS Register display (18 bits)
DATA Register display (16 bits)
Switch Register (18 switches)
Status Lights
  RUN
  PROCESSOR
  BUS
  CONSOLE
  USER
  VIRTUAL
Control Switches
  LOAD ADRS (Load Address)
  EXAM (Examine)
  CONT (Continue)
  ENABLE/HALT
  START
  DEP (Deposit)

8.2 STATUS INDICATORS

RUN  Lights when the processor clock is running. It is off when the processor is waiting for an asynchronous peripheral data response, or during a RESET instruction. It is on during a WAIT or HALT instruction.

PROCESSOR  Lights when the processor has control of the bus.

BUS  Lights when the UNIBUS is being used.

CONSOLE  Lights when in console mode (manual operation). Machine is stopped and is not executing the stored program.

USER  Lights when the CPU is executing program instructions in User mode.

VIRTUAL  Lights when the ADDRESS Register display shows the 16-bit Virtual Address.
8.3 CONSOLE SWITCHES

POWER

OFF

Power to the processor is off.

ON

Power to the processor is on and all console switches function normally.

LOCK

Power to the processor is on, but the Control Switches are disabled. The Switch Register is still functional.

Switch Register

(Up = 1)
(Down = 0)

Used to manually load data or an address into the processor.

Control Switches

LOAD ADRS
(depress to activate)

Transfers contents of the Switch Register to the Bus Address register.

EXAM
(depress to activate)

The resulting Bus Address is displayed in the ADDRESS Register, and provides an address for EXAM, DEP, and START. The LOAD Address is not modified during program execution. To restart a program at the previous Start Location, the START switch is activated.

CONT
(depress to activate)

Causes the processor to continue operation from the point at which it had stopped. The switch has no effect when the CPU is in the RUN state. If the program had stopped, this switch provides a restart without a System Reset.

ENABLE/HALT

ENABLE

Allows the CPU to perform normal operations under program control.

HALT

Causes the CPU to stop. Depressing the CONT switch will now cause execution of a single instruction.
START
(depress to activate)

If the CPU is in the RUN state, the START switch has no effect.

If the program had stopped, depressing the START switch causes a System Reset signal to occur; the program will then continue only if the ENABLE/HALT switch is in ENABLE.

DEP
(raise to activate)

Deposits contents of the Switch Register into the location specified by the Bus Address. If the DEP switch is raised again, the Switch Register contents (which were probably modified) are loaded into the next word location. (Bus Address is incremented automatically). If an odd address is specified, the next lower even address word will be used.

8.4 DISPLAYS

ADDRESS Register

Displays the address of data just examined or deposited. During a programmed HALT or WAIT instruction, the display shows the next instruction address.

DATA Register

Displays data just examined or deposited. During HALT, general register RO contents are displayed. During Single Instruction operation, the Processor Status word (PS) is displayed.
CHAPTER 9

SPECIFICATIONS

9.1 PACKAGING
The PDP-11/40 Central Processor is housed in a 21” slide chassis unit that mounts in a standard 19” rack (see Figure 9-1). The included power supply has sufficient excess capacity to drive core memory modules and peripheral logic mounted within the unit. The first 9 slots of the assembly are prewired for basic and optional CPU modules. In addition, space is provided within the chassis for mounting 7 System Units, each of which can hold 4 large (hex) modules. The power supply does not slide out, but stays mounted stably in the cabinet. The slide chassis provides convenient access to all logic modules. With a cabinet the PDP-11/40 weighs about 400 lbs.

9.2 CPU OPERATING SPECIFICATIONS
Temperature: +10° to +50°C
Relative Humidity: 20% to 95% (without condensation)
Input Power: 115 VAC ± 10%, 47 to 63 Hz
or 230 VAC ± 10%, 47 to 63 Hz

A system using a PDP-11/40 CPU loaded with 3 System Units of memory and peripheral logic draws about 12 amps at 115 VAC, or 6 amps at 230 VAC.

9.3 OTHER EQUIPMENT
Digital Equipment Corporation manufactures and sells a wide range of peripheral equipment, cabinets, and mounting assemblies. The PDP-11/40 CPU can be the heart of the system suited to your needs. There are several other PDP-11 computers available, offering price/performance choices.

All PDP-11 computers and systems are shipped with extensive support documentation, such as:

- instruction manuals
- system and diagnostic software
- installation and mounting information
- systems checkout report
Figure 9-1 PDP-11/40 Assembly Unit
### 9.4 PDP-11 FAMILY OF COMPUTERS

<table>
<thead>
<tr>
<th>CENTRAL PROCESSOR</th>
<th>11/05</th>
<th>11/10</th>
<th>11/15</th>
<th>11/20</th>
<th>11/40</th>
<th>11/45</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main Market</strong></td>
<td>OEM</td>
<td>End User</td>
<td>OEM</td>
<td>End User</td>
<td>OEM &amp; End User</td>
<td>OEM &amp; End User</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>core</td>
<td>core</td>
<td>core</td>
<td>bipolar, MOS, core</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Reg to Reg Transfer</strong></td>
<td>2.7 μs</td>
<td>2.3 μs</td>
<td>0.9 μs</td>
<td>0.3 0.45 0.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Max Mem Size (words)</strong></td>
<td>32K</td>
<td>32K</td>
<td>128K</td>
<td>128K</td>
<td>128K</td>
<td></td>
</tr>
<tr>
<td><strong>General Purpose Reg</strong></td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Stack Processing</strong></td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Micro-programmed</strong></td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Instructions</strong></td>
<td>basic set</td>
<td>basic set</td>
<td>basic set + XOR, SOB, MARK, SXT, RTT</td>
<td>same as 11/40 + MUL, DIV, ASH, ASHC, SPL</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Extended Arithmetic (hardware)</strong></td>
<td>option (external)</td>
<td>option (external)</td>
<td>option (internal) MUL, DIV, ASH, ASHC</td>
<td>standard (int)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Floating Point</strong></td>
<td>software only</td>
<td>software only</td>
<td>hardware option 32-bit word</td>
<td>hardware option 32 or 64-bit word</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Stack Limit Address</strong></td>
<td>400 (fixed)</td>
<td>400 (fixed)</td>
<td>400 or programmable (option)</td>
<td>programmable</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Memory Management</strong></td>
<td>not available</td>
<td>not available</td>
<td>option (subset)</td>
<td>option (full)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Modes</strong></td>
<td>1</td>
<td>1</td>
<td>1 std, 2 opt</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Automatic Priority Interrupt</strong></td>
<td>4-line multi-level</td>
<td>1-line multi-level (4-line, opt)</td>
<td>4-line multi-level</td>
<td>4-line multi-level + 8 software levels</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Power Fail and Auto-Restart</strong></td>
<td>standard</td>
<td>option</td>
<td>standard</td>
<td>standard</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A DESCRIPTION OF THE PDP-11 USING THE INSTRUCTION SET PROCESSOR (ISP) NOTATION

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 console 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. 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:

- Processor state (the information necessary to restart the processor if stopped between instructions, e.g., general registers, PC, index registers)
- Primary memory state (the memory directly addressable from the processor)
- Console state (any external keys, switches, lights, etc., that affect the interpretation process)
- Secondary memory (the disks, drums, dectapes, magnetic tapes, etc.)
- Transducer state (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

\[ \text{Increment} := (M[x] = M[x] + 1); \]

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

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,

\[ Z := (M[x] \rightarrow S' + O'; \ next \ M[y] \rightarrow M[x]); \]

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,

\[ \text{Mp}[0:2^k - 1]:<15:0>; \]

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 \(15, 14, \ldots, 0\). Some other examples of memory declarations are:

\[
\begin{align*}
\text{Boundary-error} &= \{\text{boolean memories; scalar bit alternatives}\}, \\
\text{Boundary-error'} &= \{\text{ternary digit, holding value 0, 1, or 2}\}, \\
\text{N/Negative} &= \{\text{alias, N and Negative are synonymous}\}, \\
M[0:2^{18}-1]:<7:0> &= \{\text{vector of } 2^{18}\text{ 8-bit words}\}, \\
M[0:15][0:4095]:<7:0> &= \{\text{array of } 16 \times 4096\text{ 8-bit words}\}, \\
br opc1:0> &= \{\text{alternative ways of defining a register}\}, \\
br opc7:0> &= \{\text{using base 16 and base 2}\}.
\end{align*}
\]

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

\[
\begin{align*}
\text{N/Negative} &= := \text{CC}<3, >, \quad \text{N is name of bit 3 of register CC} \\
\text{SP}:<15:0> &= := \text{R}[6]:<15:0>, \quad \text{SP is the same as register R[6]}
\end{align*}
\]

Examples of register formed by concatenation

\[
\begin{align*}
\text{LAC}<L, 0:11> &= := \text{LCAO}<0:11>, \\
\text{A}[6]<47> &= := \text{A}[0:23]<23>, \\
\text{Mword}[0:<15:0> &= := \text{Mbyte}[0]<7:0>:\text{Mbyte}[1]<7:0>.
\end{align*}
\]

Examples of values and registers formed by evaluation of a process

\[
\begin{align*}
\text{ai/Address-increment}<1:0> &= := (\text{value of ai is 2 if byte-op, else value is 1}), \\
\text{byte-op} &= := 2, \\
\text{byte-op} &= := 1), \\
\text{Run} &= := (\text{Activity} = 0), \\
\text{Run} &= := 1 \text{ or 0 depending on value of Activity being 0 or not 0}
\end{align*}
\]

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.

mathematical operators. For example, +, - , *, / , ^ , =, !=.

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.

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).

The definition b, may have dummy parameters, c, ..., e, which are used in g(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.

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).

operator and data-type modifier

conditional expression; equivalent to ALGOL if boolean then expression

equivalent to Algol if boolean then expression-1 else expression-2

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.

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

set value. Takes on all values for a digit of the given base, e.g., 12, specifies either 10 or 112.

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.
Common Arithmetic, Logical and Relational Operators

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Logical</th>
<th>Relational</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ add</td>
<td>¬ not</td>
<td>= identical</td>
</tr>
<tr>
<td>- subtract, also negative</td>
<td>∧ and</td>
<td>≠ not identical</td>
</tr>
<tr>
<td>× multiply</td>
<td>∨ or</td>
<td>= equal</td>
</tr>
<tr>
<td>/ divide</td>
<td>⊕ exclusive-or</td>
<td>≥ not equal</td>
</tr>
<tr>
<td>( )² squared</td>
<td>≡ equivalence</td>
<td>≥ greater than or equal</td>
</tr>
<tr>
<td>( )ⁿ exponentiation</td>
<td>base</td>
<td>&lt; less than</td>
</tr>
<tr>
<td>( )ⁿ exponentiation</td>
<td>base</td>
<td>≤ less than or equal</td>
</tr>
<tr>
<td>( )ᵇ base</td>
<td>( )ᵇ base</td>
<td></td>
</tr>
<tr>
<td>sqrt( )</td>
<td>sqrt( )</td>
<td></td>
</tr>
<tr>
<td>abs( )</td>
<td>abs( )</td>
<td></td>
</tr>
<tr>
<td>sign-extend( )</td>
<td>sign-extend( )</td>
<td></td>
</tr>
</tbody>
</table>

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, $D_{address}$, 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,

$$nw' := (nw[PC] : PC := PC + 2)$$

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:

$$S' := (\text{value for source operand})$$

if mode=0 then $S'$ is the Register addressed by instruction field $sr$

if mode=1 the $S'$ is indirect via $R[sr]$

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:

\[
S<15:0> := ( \\
(\texttt{mm}0) = R[\texttt{sr}] ; \\
(\texttt{mm}1) = \text{HW}[R[\texttt{sr}]] ; \\
(\texttt{mm}2) \land (\texttt{sr}=7) = \text{HW}[\texttt{PC}-2] \\
) \\
\]

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.

\[
S<15:0> := <11:6> \\
\texttt{mm}0 := s<5:3> \\
\texttt{mm}1 := s<6> \\
\texttt{mm}2 := s<2:0> \\
\texttt{mm}3 <15:0> := (\text{HW}[\texttt{PC}] ; \texttt{PC} \rightarrow \texttt{PC}+2) \\
R<15:0> := R[\texttt{sr}] \\
\]

value for the source-direct addressing
source field (8-bits) of instruction
use the register \( R_\text{e} \) as operand
direct auto-increment (increment)
use \( P_\text{P} \) as operand
direct; actually immediate operand
direct; auto-decrement (decrement)
use \( \text{S}_{\text{PUSH}} \) as operand
direct; indexed via \( R_\text{e} \) uses next-word
direct; relative to \( \text{PC} \); uses next-word
value for the source-defined addressing
defer through \( R_\text{e} \)
defer through stack; auto increment
defer via next word; absolute addressing
defer through stack after auto decrement
defer, indexed via \( R_\text{e} \)
defer relative to \( \text{PC} \)
end calculation process;

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

A-5
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/character, floating point numbers/floating point numbers); others are named by their value (e.g., integer/integer). 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, implying a multi-dimensional array of fixed, but unspecified dimensions; a string, implying a single sequence, of variable length (on each occurrence); or a vector, 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., d 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+B{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
```

In more complex processors the conditions for trapping and interrupting must also be described. 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 A Run = (instruction = Mw[PC]; PC = PC + 2; next fetch
Instruction-execution; next
execute
T-flag = (State-change(148); T-flag = 0))
```

A-6
Table 2. Common Data-Types Abbreviations

<table>
<thead>
<tr>
<th>Primitive</th>
<th>String and Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>bv</td>
</tr>
<tr>
<td>by</td>
<td>by.st</td>
</tr>
<tr>
<td>ch</td>
<td>ch.st</td>
</tr>
<tr>
<td>cx</td>
<td></td>
</tr>
<tr>
<td>df</td>
<td></td>
</tr>
<tr>
<td>dw</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
</tr>
<tr>
<td>fr</td>
<td></td>
</tr>
<tr>
<td>hw</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td></td>
</tr>
<tr>
<td>mx</td>
<td></td>
</tr>
<tr>
<td>qw</td>
<td></td>
</tr>
<tr>
<td>tw</td>
<td></td>
</tr>
<tr>
<td>w</td>
<td></td>
</tr>
</tbody>
</table>

**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:

\[
\text{MOV} (:= \text{bop} = 0001_2) = (r = \text{Sr'; next}) = (\text{move word}) \; \text{to intermediate register}
\]

\[
\text{R} = \text{r<:l5-}; \quad (\text{negative?})
\]

\[
(\text{r<:l5.0} = 0) \implies (\text{Z = 1 else Z = 0}); \quad (\text{zero?})
\]

\[
V = 0; \quad (\text{overflow cleared})
\]

\[
D = r); \quad (\text{transmit result to destination})
\]

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,

\[
\text{MOV} := (\text{bop} = 0001_2)
\]

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

\[ \text{Mp}[0:2^{k}-1]<15:0> \]

- \( \text{Mw/Mword}[x<15:0>]<15:0> := ( \)
  - \( x<0> = \text{Mp}[x<15:1>]; \)
  - \( x<0> = (\text{value} \cdot \text{Boundary-error} = 1) \)

- \( \text{Mb/Mbyte}[x<15:0>]<7:0> := ( \)
  - \( x<0> = \text{Mp}[x<15:1>]<7:0>; \)
  - \( x<0> = \text{Mp}[x<15:1>]<15:8> \)

Processor State

\[ \text{R}[0:7]<15:0> \]

- \( \text{SP}<15:0>://\text{Stack-Pointer} := \text{R}[6] \)
- \( \text{PC}<15:0>://\text{Program-Counter} := \text{R}[7] \)
- \( \text{PS}<15:0>://\text{Processor-State-Word} \)

Unused<7:0>/Undefined := PS<15:8>

- \( P<3:0>://\text{Priority} := \text{PS}<7:5> \)
- \( T/\text{Trace} := \text{PS}<4> \)

CC<3:0>/\text{Condition-Codes} := \text{PS}<3:0>

- \( N/\text{Negative} := \text{CC}<3> \)
- \( Z/\text{Zero} := \text{CC}<2> \)
- \( V/\text{Overflow} := \text{CC}<1> \)
- \( C/\text{Carry} := \text{CC}<0> \)

Processor-Controlled Error Flags (resulting from instruction-execution)

- Boundary-Error
- Stack-Overflow
- Time-Out-Error
- Illegal-Instruction

Processor-activity

Activity3

- \( \text{Run} := (\text{Activity} = 0) \)
- \( \text{Wait} := (\text{Activity} = 1) \)
- \( \text{Off} := (\text{Activity} = 2) \)

Error-Flags (resulting from without the processor)

- Power-Fail-Flag
- Power-Up-Flag

A-8
Instruction format field declarations

ice:15:0/instruction

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bop</td>
<td>binary opcode format</td>
</tr>
<tr>
<td>sf</td>
<td>source field</td>
</tr>
<tr>
<td>smg</td>
<td>source mode - 3 bits</td>
</tr>
<tr>
<td>sd</td>
<td>source defer bit</td>
</tr>
<tr>
<td>srg</td>
<td>source register - 3 bits</td>
</tr>
<tr>
<td>dt&lt;5:0&gt;</td>
<td>destination field</td>
</tr>
<tr>
<td>dm&lt;5:3&gt;</td>
<td>destination mode - 3 bits</td>
</tr>
<tr>
<td>dd</td>
<td>destination defer bit</td>
</tr>
<tr>
<td>dr&lt;8&gt;</td>
<td>destination register - 3 bits</td>
</tr>
<tr>
<td>uop&lt;3:0&gt;</td>
<td>unary op code (arith., logical, shifts)</td>
</tr>
<tr>
<td>df</td>
<td>see binary op format</td>
</tr>
<tr>
<td>jsop&lt;7:0&gt;</td>
<td>JSR format</td>
</tr>
<tr>
<td>sr</td>
<td>see binary op format</td>
</tr>
<tr>
<td>brop&lt;1:0&gt;</td>
<td>branch format</td>
</tr>
<tr>
<td>offset&lt;7:0&gt;</td>
<td>offset value</td>
</tr>
<tr>
<td>trop&lt;1:0&gt;</td>
<td>trap format</td>
</tr>
<tr>
<td>unused-trop&lt;1:0&gt;</td>
<td>see binary op format</td>
</tr>
<tr>
<td>eop&lt;6:0&gt;</td>
<td>extended opcode format</td>
</tr>
<tr>
<td>er&lt;3:0&gt;</td>
<td>extended register</td>
</tr>
<tr>
<td>esf&lt;5:0&gt;</td>
<td>extended source field</td>
</tr>
<tr>
<td>esm&lt;3&gt;</td>
<td>mode</td>
</tr>
<tr>
<td>esd&lt;3&gt;</td>
<td>defer</td>
</tr>
<tr>
<td>esrg&lt;2:0&gt;</td>
<td>register</td>
</tr>
<tr>
<td>fop&lt;7:0&gt;</td>
<td>floating op format</td>
</tr>
<tr>
<td>fr&lt;7:6&gt;</td>
<td>register destination</td>
</tr>
<tr>
<td>fsf&lt;5:0&gt;</td>
<td>source</td>
</tr>
</tbody>
</table>

Binary operand (2 operands) format

<table>
<thead>
<tr>
<th>bop</th>
<th>sf</th>
<th>df</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unary operand (1 operand), JMP format

<table>
<thead>
<tr>
<th>uop</th>
<th>df</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

JSR format

<table>
<thead>
<tr>
<th>jsop</th>
<th>sr</th>
<th>df</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Branch format

<table>
<thead>
<tr>
<th>brop</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>value := sign-extend (offset)</td>
<td></td>
</tr>
</tbody>
</table>

Trap format

<table>
<thead>
<tr>
<th>trop</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended operation format

<table>
<thead>
<tr>
<th>eop</th>
<th>er</th>
<th>esf</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Floating op format

<table>
<thead>
<tr>
<th>fop</th>
<th>fr</th>
<th>fsf</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A-9
\[ \text{si}/\text{address-increment}<1:0> : = ( \\
\text{- Byte-op} = 2; \\
\text{Byte-op} = 1) \\
\text{Byte-op} : = (\text{MOVB} \lor \text{BICB} \lor \text{BISB} \lor \text{BITB} \lor \text{CLR} \lor \\
\text{COMB} \lor \text{INCB} \lor \text{DECB} \lor \text{NEG} \lor \text{ADC} \lor \\
\text{SBC} \lor \text{TST} \lor \text{FORD} \lor \text{ROL} \lor \text{ASR} \lor \\
\text{ASL} \lor \text{SWAP}) \\
\text{Reserved-instruction} : = ((i = 1) \lor (i = 0) \lor \ldots \lor (i = )) \text{ unused instructions} \\
\]
(m=7) ∧ (rg=7) → M[Mw[|nw + Rr|]];
(m=7) ∧ (rg=7) → M[Mw[|m' + PC|]];

(SP < 400₈) → (Stack-overflow ≠ 1)
end operand calculation process
check if stack overflows
defer indexed via Rr
defer relative to PC
end calculation process
undo previous side-effects
undo previous side-effects
undo previous side-effects
undo previous side-effects
undo previous side-effects
undo previous side-effects
undo previous side-effects

General Operand Calculation Process (without Side Effects)
Oprd<15:0> = (Mw[m,i,m,rg]) :=

Rr<15:0> := R[rg]
(m=0) = Rr<15:0>;

(m=2) ∧ (rg=7) = Mw[Rr - a1];
(m=2) ∧ (rg=7) = lw<15:0>;

(m=4) = Mw[Rr];

(m=6) ∧ (rg=7) = Mlw[Rr];
(m=6) ∧ (rg=7) = Mlw[PC];

(m=1) = Mw[Rr];

(m=3) ∧ (rg=7) = Mw[Rr - 2];
(m=3) ∧ (rg=7) = Mlw;

(m=5) = Mw[Rr];

(m=7) ∧ (rg=7) = Mlw[Rr + 1];
(m=7) ∧ (rg=7) = Mlw[PC]]

Destination addresses for JMP and JSR
Dr<15:0> :=

(dm=0) = (?; illegal-instruction - 1);
(dm=2) ∧ (dr=7) = (Rd; Rd - Rd + 2);
(dm=2) ∧ (dr=7) = (PC; PC - PC + 2);
(dm=4) = (Rd - Rd - 2; next Rd);
(dm=6) ∧ (dr=7) = (nw' + Rd);

(dm=1) = Mw[Rd];

(dm=3) ∧ (dr=7) = (Mw[Rd]; Rd - Rd + 2);
(dm=3) ∧ (dr=7) = nw';

(dm=5) = (Rd - Rd - 2; next Mw[Rd]);
(dm=7) ∧ (dr=7) = Mlw[nw + Rd];

(dm=7) ∧ (dr=7) = Mlw[nw' + PC]); next

(dm=6) ∧ = ((dm=0) ∧ (dm=3) ∧ (dm=7)) ∧ (SP < 400₈) = (check for stack overflow
stack-overflow = 1))

Data Type Formats
by/byte<7:0>
w/word<15:0>
wI/word.integer<15:0>
by8/byte.boolean-vector<7:0>
w8/word.boolean-vector<15:0>
d/d.w/double.word<31:0>
I/O Devices and Interrupts, State Information

Device[0..N-1]

Device-name[J]<15:0> := J

Device-interrupt-location[J]<15:0> := K

dob/device-output-buffer[J]<15:0>:
dib/device-input-buffer[J]<15:0>:

ds/device-status[J]<15:0>:
dunit/device-unit-selection[J]<2:0> := ds[J]<10:8>
ddone[J] := ds[J]<7>
demb/device-done-interrupt-enable := ds[J]<6>
derrenb/device-error-interrupt-enable := ds[J]<5>
dfunc/device-function[J]<2:0> := ds[J]<2:0>

intrq/device-interrupt-request[J] := (ddone[J] \& denb[J] \& \neg (derr[J] \& 0) \& \neg derrenb[J])
dil/device-interrupt-level[J]<7:4> := each device is assigned to 1 of 4 levels

Mapping of Devices into M. Each device's registers are mapped into primary word memory, e.g.,

Teleprinter

M'[177560_8] := tks/ds[TTY-keyboard]
M'[177562_8] := tkb/dib[TTY-keyboard]
M'[177564_8] := tps/ds[TTY-printer]
M'[177566_8] := tpb/dob[TTY-printer]

Interrupt Requests
br/bus-request-for-interrupt[7:4] := (intrq[0] \& dil[0]) \lor
(intrq[1] \& dil[1]) \lor...
(intrq[J] \& dil[J]) \lor...
(intrq[N] \& dil[N])

OR of all device requests

Interrupt-req := (intrq \geq p)

interrupt if a request is \geq priority/P

intrql/interrupt-request-level[2:0] := (br<7> = 7;
\neg br<7> \& br<6> = 6;
\neg br<7> \& \neg br<6> \& \neg br<5> \& br<4> = 4)

N I/O devices - assume device J
number to which device responses and is addressed
each device has a value, K, which it uses as an address to interrupt processor program controlled device data

a register with device control state common

status assignments

keyboard status

keyboard input data

teleprinter status

teleprinter data to print

A-12
Instruction Interpretation Process

Interrupt rq ∧ Run = (Normal-interpretation);

Normal-interpretation := (1 - MW[PC]; PC = PC + 2 next
executed
Instruction-execution; next

T-flag = (State-change(46)); T-flag = 0)

Interrupt rq ∧ Off = (State-change(Device-interrupt-location[J]));

P = intrql);

off = ( );

- Interrupt rq ∧ Wait = ( );

State-change(x) := ( fetch
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(48); Boundary-error = 0)

Time-Out-Error = (State-change(48); Time-Out-Error = 0)

Power-Fail-Flag = (state-change(248); Power-Fail-Flag = 0;) program must turn off computer

Power-Up-Flag = (PC = 248; 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.

ISP for Floating Point Processor/FPP

Device-interrupt-location [FPP] := M'[2448]

FEC<15:0>:

Floating point processor error code register

FOE := (FEC=2) floating op code error

FDE := (FEC=4) floating divide by zero

FICE := (FEC=6) floating integer conversion error

FVE := (FEC=8) floating overflow

FUE := (FEC=10) floating underflow

FUVE := (FEC=12) floating undefined variable

FAC<0:5>:63:0> 6 floating point accumulators

FS<63:0>:

Floating point status register

FPSC<15:0>:

Floating point processor status register

FER := FPSR<15> floating error

FIE := FPSR<14> interrupt enable

FIU := FPSR<11> interrupt on undefined variable

FIU := FPSR<10> interrupt on underflow

FIV := FPSR<9> interrupt on overflow

FIC := FPSR<8> interrupt on integer conversion error

FD := FPSR<7> floating double precision mode

FL := FPSR<6> floating long integer mode

FT := FPSR<5> floating truncate mode

FM := FPSR<4> floating maintenance mode
FN := FPSR<3>
FZ := FPSR<2>
FV := FPSR<1>
FC := FPSR <0>

Floating negative
Floating zero
Floating overflow
Floating carry

Instruction format
O<3>: := i<15:12>
F<3>: := i<11:8>
A<1>: := i<7:6>
op code
floating op code
accumulator

General Definitions
XL := (((FD=0) = 1-2^-24;
(FD=1) = 1-2^-56)
XLL := 2^-128
XUL := 2^127 * XL
JL := (((FL=0) = 2^-15;
FL=1) = 2^31-1)
largest fraction
smallest non-zero number
largest number
largest integer

Address Calculation
FPS<63>: := (
(dwn=0) = FAC(dr);
(dwn=0) = (
(FD=0) = D<15:0>Mw(PC+2);
(FD=1) = D<15:0>Mw(PC+2)C
Mw[PC+4|Mw(PC+6)])
FPS'<63>: := (
(dwn=0) = FAC(dr);
(dwn=0) = (
(FD=0) = D'<15:0>Mw'
(FD=1) = D'<15:0>Mw'Mw')
floating point processor source
floating point processor source with side effects
floating point processor destination
floating point processor destination with side effects
floating source, CPU mode
floating source with side effects, CPU mode
floating destination, CPU mode
floating destination with side effects, CPU mode
destination floating register

dA 17 bit result, r, used only for descriptive purposes
2A 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.
3If all 16 bits of result, r = 0, then Z is set to 1 else Z is set to 0.
4The 8 least significant bits are used to form a 16-bit positive or negative number by extending bit 7 into 15:8.
5a → b means: if boolean a is true then b is executed.
6Mw means the memory taken as a work-organized memory.
# APPENDIX B  MEMORY MAP

## INTERRUPT VECTORS.

<table>
<thead>
<tr>
<th>Vector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>004</td>
<td>TIME OUT, BUS ERROR</td>
</tr>
<tr>
<td>010</td>
<td>RESERVED INSTRUCTION</td>
</tr>
<tr>
<td>014</td>
<td>DEBUGGING TRAP VECTOR</td>
</tr>
<tr>
<td>020</td>
<td>IOT TRAP VECTOR</td>
</tr>
<tr>
<td>024</td>
<td>POWER FAIL TRAP VECTOR</td>
</tr>
<tr>
<td>030</td>
<td>EMT TRAP VECTOR</td>
</tr>
<tr>
<td>034</td>
<td>&quot;TRAP&quot; TRAP VECTOR</td>
</tr>
<tr>
<td>040</td>
<td>SYSTEM SOFTWARE</td>
</tr>
<tr>
<td>044</td>
<td>SYSTEM SOFTWARE</td>
</tr>
<tr>
<td>050</td>
<td>SYSTEM SOFTWARE</td>
</tr>
<tr>
<td>054</td>
<td>SYSTEM SOFTWARE</td>
</tr>
<tr>
<td>060</td>
<td>TTY IN-BR4</td>
</tr>
<tr>
<td>064</td>
<td>TTY OUT-BR4</td>
</tr>
<tr>
<td>070</td>
<td>PC11 HIGH SPEED READER-BR4</td>
</tr>
<tr>
<td>074</td>
<td>PC11 HIGH SPEED PUNCH</td>
</tr>
<tr>
<td>100</td>
<td>KW11L · LINE CLOCK BR6</td>
</tr>
<tr>
<td>104</td>
<td>KW11P · PROGRAMMER REAL TIME CLOCK BR6</td>
</tr>
<tr>
<td>120</td>
<td>XY PLOTTER</td>
</tr>
<tr>
<td>124</td>
<td>DR11B-(BR5 HARDWIRED)</td>
</tr>
<tr>
<td>130</td>
<td>ADO1 BR5-(BR7 HARDWIRED)</td>
</tr>
<tr>
<td>134</td>
<td>AFC11 FLYING CAP MULTIPLEXER BR4</td>
</tr>
<tr>
<td>140</td>
<td>AA11-A,B,C SCOPE BR4</td>
</tr>
<tr>
<td>144</td>
<td>AA11 LIGHT PIN BR5</td>
</tr>
<tr>
<td>170</td>
<td>USER RESERVED</td>
</tr>
<tr>
<td>174</td>
<td>USER RESERVED</td>
</tr>
<tr>
<td>200</td>
<td>LP11 LINE PRINTER CTRL-BR4</td>
</tr>
<tr>
<td>204</td>
<td>RF11 DISK CTRL-BR5</td>
</tr>
<tr>
<td>210</td>
<td>RC11 DISK CTRL-BR5</td>
</tr>
<tr>
<td>214</td>
<td>TC11 DEC TAPE CTRL-BR6</td>
</tr>
<tr>
<td>220</td>
<td>RK11 DISK CTRL-BR5</td>
</tr>
<tr>
<td>224</td>
<td>TM11 COMPATIBLE MAG TAPE CTRL-BR5</td>
</tr>
<tr>
<td>230</td>
<td>CR11/CM11 CARD READER CTRL-BR6</td>
</tr>
<tr>
<td>234</td>
<td>UDC11 (BR4, BR6 HARDWIRED)</td>
</tr>
<tr>
<td>240</td>
<td>11/45 PIRQ</td>
</tr>
<tr>
<td>244</td>
<td>FPU ERROR</td>
</tr>
<tr>
<td>254</td>
<td>RP11 DISK PACK CTRL-BR5</td>
</tr>
<tr>
<td>260</td>
<td></td>
</tr>
<tr>
<td>264</td>
<td></td>
</tr>
<tr>
<td>270</td>
<td>USER RESERVED</td>
</tr>
<tr>
<td>274</td>
<td>USER RESERVED</td>
</tr>
<tr>
<td>300</td>
<td>START OF FLOATING VECTORS</td>
</tr>
</tbody>
</table>
DEVICE ADDRESSES

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  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 OUT CSR
777562  KL11 TTY IN DBR
777560  KL11 TTY IN CSR
777556  PC11 HSP DBR
777554  PC11 HSP CSR
777552  PC11 HSR DBR
777550  PC11 HSR CSR
777546  LKS LINE CLOCK KW11-L

777516  LP11 DBR
777514  LP11 CSR
777512  LP11 XX
777510  LP11 XX

777476  RF11 DISK RFLA LOOK AHEAD
777474  RF11 DISK RFMR MAINTENANCE
777472  RF11 DISK RFDBR
777470  RF11 DISK RFDAE
777466  RF11 DISK RDFAR
777464  RF11 DISK RFCAR
777462  RF11 DISK RFWC
777460  RF11 DISK RFDS

777456  RC11 DISK RCDBR
777454  RC11 MAINTENANCE
777452  RC11 RCCAR
777450  RC11 RCWC
777446  RC11 RCCSR-
777444  RC11 RCCSR1
777442  RC11 RCER
777440  RC11 RCLA

B-2
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  TCCM
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 CARD READER
777162  CRDBR1
777160  CRCSR

776776  ADO1·D XX
776774  ADO1·D XX
776772  ADDBR  A/D CONVERTER ADO1·D
776770  ADCSR

B-3
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
774377 TO 774000 DC11

773777 TO 773000 DIODE MEMORY MATRIX

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

B-4
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
772416  DR11B/DATA
772414  DR11B/STATUS
772412  DR11B/BA
772410  DR11B/WC
772136 TO 772110 MEMORY PARITY CSR

772136  15
772120  4
772116  3
772114  2
772112  1
772110  0

771776  UDCS · CONTROL AND STATUS REGISTER
771774  UDSR · SCAN REGISTER
771772  MCLK · MAINTENANCE REGISTER
771766  UDC FUNCTIONAL I/O MODULES
771000  UDC FUNCTIONAL I/O MODULES

770776 TO 770700 KG11 CRC OPTION
770776  KG11A KGGNU7
770774  KGDBR7
770772  KGBBC7
770770  KGCSR7
770716  KGGNU1
770714  KGBCC1
770712  KGDBR1
770710  KGCSR1
770706  KGGNU0
770704  KGDBR0
770702  KGBCC0
770700  KG11A KGCSRO
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

B-6
START NORMAL USER ADDRESSES HERE AND ASSIGN UPWARD.
760004 TO 760000 RESERVED FOR DIAGNOSTIC - SHOULD NOT BE ASSIGNED
APPENDIX C

PDP-11/40 INSTRUCTION TIMING

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

\[ \text{Instr Time} = \text{SRC Time} + \text{DST Time} + \text{EF Time} \]

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

I. BASIC INSTRUCTION SET TIMING

Double Operand
all instructions,
   except MOV: \[ \text{Instr Time} = \text{SRC Time} + \text{DST Time} + \text{EF Time} \]
MOV Instruction: \[ \text{Instr Time} = \text{SRC Time} + \text{EF Time} \]

Single Operand
all instr, except MFPI, MTPI: \[ \text{Instr Time} = \text{DST Time} + \text{EF Time} \]
MFPI, MTPI instructions: \[ \text{Instr Time} = \text{EF Time} \]

Branch, Jump, Control, Trap, & Misc
all instructions: \[ \text{Instr Time} = \text{EF Time} \]

NOTES:
1. The times specified generally apply to Word instructions. In most cases Even Byte instructions have the same times, with some Odd Byte instructions taking longer. All exceptions are noted.

2. Timing is given without regard for NRP or BR servicing. Memory types MM11-S, MF11-L, and ML11 are assumed with direct use of the special processor MSYNA signal and with memory within the CPU mounting assembly. Use of the regular Unibus BUS MSYN signal means 0.08 \( \mu \)sec must be added for each memory cycle.

3. If the Memory Management (KT11-D) option is installed, instruction execution times increase by 0.15 \( \mu \)sec for each memory cycle used.
### SOURCE ADDRESS TIME

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source Mode</th>
<th>SRC Time (A)</th>
<th>Memory Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0.00 μsec</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>.78</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>.84</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>3</td>
<td>1.74</td>
<td>2</td>
</tr>
<tr>
<td>Operand</td>
<td>4</td>
<td>.84</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1.74</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>1.46</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>2.36</td>
<td>3</td>
</tr>
</tbody>
</table>

**NOTE (A):** For Source Modes 1 thru 7, add 0.34 μsec for Odd Byte instructions.

### DESTINATION ADDRESS TIME

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Destination Mode</th>
<th>DST Time (B)</th>
<th>Memory Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>0</td>
<td>0.00 μsec</td>
<td>0</td>
</tr>
<tr>
<td>Operand, and</td>
<td>1</td>
<td>.78 ( .90)</td>
<td>1</td>
</tr>
<tr>
<td>Double (except MOV, JMP, JSR)</td>
<td>5</td>
<td>1.74 (1.80)</td>
<td>2</td>
</tr>
<tr>
<td>Operand</td>
<td>4</td>
<td>.84 ( .90)</td>
<td>1</td>
</tr>
<tr>
<td>(except MOV, JMP, JSR)</td>
<td>6</td>
<td>1.46 (1.74)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>2.36 (2.64)</td>
<td>1</td>
</tr>
</tbody>
</table>

**NOTE (B):** For Destination Modes 1 thru 7, add 0.34 μsec for Odd Byte instructions. Use higher values in parentheses ( ) for ADD, SUB, CMP, BIT, BIC, or BIS and a Source Mode of 0.

### EXECUTE, FETCH TIME

**Double Operand**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>SRC Mode 0 DST Mode 0</th>
<th>SRC Mode 1 to 7 DST Mode 0</th>
<th>SRC Mode 0 to 7 DST Mode 1 to 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>(use with SRC Time &amp; DST Time)</td>
<td>MF Mem</td>
<td>MF Mem</td>
<td>MF Mem</td>
</tr>
<tr>
<td>ADD, CMP, BIT, BIC, BIS</td>
<td>0.99 μs 1</td>
<td>1.60 μs 1</td>
<td>1.76 μs 2</td>
</tr>
<tr>
<td>SUB</td>
<td>.99 1</td>
<td>1.60 1</td>
<td>1.90 2</td>
</tr>
<tr>
<td>XOR</td>
<td>.99 1</td>
<td>—</td>
<td>1.76 2</td>
</tr>
</tbody>
</table>

**NOTE (C):** For Destination Modes 1 thru 7, add 0.48 μsec for Odd Byte instructions.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>DST Mode</th>
<th>SRC Mode</th>
<th>EF Time (Word instr)</th>
<th>EF Time (Odd or Even Byte)</th>
<th>Memory Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0.90 μsec</td>
<td>1.80 μsec</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1 to 7</td>
<td>1.46</td>
<td>1.80 μsec</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0 to 7</td>
<td>2.42</td>
<td>2.56</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0 to 7</td>
<td>2.42</td>
<td>2.56</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>3</td>
<td>0 to 7</td>
<td>3.18</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>(use with SRC Time)</td>
<td>4</td>
<td>0 to 7</td>
<td>2.42</td>
<td>2.56</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0 to 7</td>
<td>3.18</td>
<td>3.32</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>2.84</td>
<td>2.98</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1 to 7</td>
<td>3.18</td>
<td>3.32</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>3.68</td>
<td>3.82</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1 to 7</td>
<td>4.02</td>
<td>4.16</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

**Single Operand**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Destination Mode 0</th>
<th>Destination Mode 1 to 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>(use with DST Time)</td>
<td>EF Time</td>
<td>Mem Cycles</td>
</tr>
<tr>
<td>CLR, COM, NEG, INC, DEC, ADC, SBC, TST, ROL, ASL, SWAB</td>
<td>0.99 μs</td>
<td>1</td>
</tr>
<tr>
<td>ROR, ASR</td>
<td>1.25 (E)</td>
<td>1</td>
</tr>
<tr>
<td>SXT</td>
<td>.90</td>
<td>1</td>
</tr>
</tbody>
</table>

**NOTE (D):** For Destination Modes 1 thru 7, add 0.48 μsec for Odd Byte instructions.

**NOTE (E):** For RORB and ASRB, add 0.14 μsec for Even or Odd Byte instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instr Time</th>
<th>Mem Cycles</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>MFPI</td>
<td>3.74 μs</td>
<td>2</td>
<td>These two instructions are implemented only if Memory Management is installed.</td>
</tr>
<tr>
<td>MTPI</td>
<td>3.68</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

**Branch Instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instr Time (Branch)</th>
<th>Instr Time (No Branch)</th>
<th>Memory Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>BR, BNE, BEQ, BPL, BMI, BVC, BVS, BCC, BCS, BGE, BLT, BGT, BLE, BHI, BLO, BHIS, BLO</td>
<td>1.76 μsec</td>
<td>1.40 μsec</td>
<td>1</td>
</tr>
<tr>
<td>SOB</td>
<td>2.36</td>
<td>2.04</td>
<td>1</td>
</tr>
</tbody>
</table>
### Jump Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Destination Mode</th>
<th>Instr Time</th>
<th>Memory Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.80 μsec</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2.10</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2.30</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>JMP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1.90</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2.30</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2.36</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2.92</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>JSR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3.04</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3.44</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3.50</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>4.06</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

### Control, Trap, & Misc Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instr Time</th>
<th>Mem Cyc</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTS</td>
<td>2.42 μsec</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>MARK</td>
<td>2.56</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RTI, RTT</td>
<td>2.92</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SET N,Z,V,C</td>
<td>1.72</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CLR N,Z,V,C</td>
<td>2.02</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>HALT</td>
<td>2.42</td>
<td>1</td>
<td>Console loop for a switch setting is 0.44 μsec.</td>
</tr>
<tr>
<td>WAIT</td>
<td>2.24</td>
<td>1</td>
<td>WAIT loop for a BR is 1.12 μsec.</td>
</tr>
<tr>
<td>RESET</td>
<td>80 msec</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>IOT, EMT</td>
<td>5.80 μsec</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

### LATENCY

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

Interrupt service time, which is the time from BR acknowledgement to the first subroutine instruction, is 5.42 μsec, max.

NPR (DMA) latency, which is the time from request to bus mastership for the first NPR device, is 3.50 μsec, max.
II. EIS, KE11-E, INSTRUCTION TIMING

Instr Time = SRC Time + EF Time

<table>
<thead>
<tr>
<th>Source</th>
<th>Mode</th>
<th>SRC Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0.28 µsec</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>.78</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>.98</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>1.74</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>.98</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>1.74</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>1.74</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>2.64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>EF Time</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL</td>
<td>8.88 µsec</td>
<td></td>
</tr>
<tr>
<td>DIV</td>
<td>11.30</td>
<td></td>
</tr>
<tr>
<td>ASH (right)</td>
<td>2.58</td>
<td>Add 0.30 µsec per shift.</td>
</tr>
<tr>
<td>ASH (left)</td>
<td>2.78</td>
<td>Add 0.30 µsec per shift.</td>
</tr>
<tr>
<td>ASHC (no shift)</td>
<td>2.78</td>
<td></td>
</tr>
<tr>
<td>ASHC (shift)</td>
<td>3.26</td>
<td>Add 0.30 µsec per shift.</td>
</tr>
</tbody>
</table>

LATENCY

Interrupts are acknowledged at the end of the current instruction. Interrupt service time is 5.42 µsec, max. NPR latency is 3.50 µsec, max.

III. FLOATING POINT, KE11-F, INSTRUCTION TIMING

Instr Time=Basic Time+Shift Time for binary pts+Shift Time for norm

<table>
<thead>
<tr>
<th>Instr</th>
<th>Basic Time</th>
<th>Time per shift to line up binary points (0 to 23 shifts)</th>
<th>Time per shift for normalization (0 to 25 shifts)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FADD</td>
<td>18.78 µsec</td>
<td>0.30 µsec</td>
<td>0.34 µsec</td>
</tr>
<tr>
<td>FSUB</td>
<td>19.08</td>
<td>.30</td>
<td>.34</td>
</tr>
<tr>
<td>FMUL</td>
<td>29.00</td>
<td>—</td>
<td>.34</td>
</tr>
<tr>
<td>FDIV</td>
<td>46.72</td>
<td>—</td>
<td>.34</td>
</tr>
</tbody>
</table>

Basic instruction times shown for FADD and FSUB assume exponents are equal or differ by one.
LATENCY

If an interrupt request of higher priority than the operating program occurs during a Floating Point instruction, the current instruction will be aborted unless it is near completion. The maximum time from interrupt request to acknowledgement during Floating Point instruction execution is 20.08 μsec. Interrupt service time is 5.42 μsec, max. NPR latency is 3.50 μsec, max.
## APPENDIX D
### INSTRUCTION INDEX

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC(B)</td>
<td>4-19</td>
</tr>
<tr>
<td>ADD</td>
<td>4-25</td>
</tr>
<tr>
<td>ASL(B)</td>
<td>4-14</td>
</tr>
<tr>
<td>ASH</td>
<td>4-33</td>
</tr>
<tr>
<td>ASHC</td>
<td>4-34</td>
</tr>
<tr>
<td>ASR(B)</td>
<td>4-13</td>
</tr>
<tr>
<td>BCC</td>
<td>4-44</td>
</tr>
<tr>
<td>BCS</td>
<td>4-45</td>
</tr>
<tr>
<td>BEQ</td>
<td>4-39</td>
</tr>
<tr>
<td>BGE</td>
<td>4-47</td>
</tr>
<tr>
<td>BGT</td>
<td>4-49</td>
</tr>
<tr>
<td>BHI</td>
<td>4-52</td>
</tr>
<tr>
<td>BHI S</td>
<td>4-54</td>
</tr>
<tr>
<td>BIC(B)</td>
<td>4-29</td>
</tr>
<tr>
<td>BIS(B)</td>
<td>4-30</td>
</tr>
<tr>
<td>BIT(B)</td>
<td>4-28</td>
</tr>
<tr>
<td>BLT</td>
<td>4-48</td>
</tr>
<tr>
<td>BLE</td>
<td>4-50</td>
</tr>
<tr>
<td>BLO</td>
<td>4-55</td>
</tr>
<tr>
<td>BLO S</td>
<td>4-53</td>
</tr>
<tr>
<td>BMI</td>
<td>4-41</td>
</tr>
<tr>
<td>BNE</td>
<td>4-38</td>
</tr>
<tr>
<td>BPL</td>
<td>4-40</td>
</tr>
<tr>
<td>BPT</td>
<td>4-67</td>
</tr>
<tr>
<td>BR</td>
<td>4-37</td>
</tr>
<tr>
<td>BVC</td>
<td>4-42</td>
</tr>
<tr>
<td>BVS</td>
<td>4-43</td>
</tr>
<tr>
<td>CLR(B)</td>
<td>4-6</td>
</tr>
<tr>
<td>CMP(B)</td>
<td>4-24</td>
</tr>
<tr>
<td>COM(B)</td>
<td>4-7</td>
</tr>
<tr>
<td>COND. CODES</td>
<td>4-79</td>
</tr>
<tr>
<td>DEC(B)</td>
<td>4-9</td>
</tr>
<tr>
<td>DIV</td>
<td>4-32</td>
</tr>
<tr>
<td>EMT</td>
<td>4-65</td>
</tr>
<tr>
<td>FADD</td>
<td>7-4</td>
</tr>
<tr>
<td>FDIV</td>
<td>7-5</td>
</tr>
<tr>
<td>FMUL</td>
<td>7-5</td>
</tr>
<tr>
<td>FSUB</td>
<td>7-4</td>
</tr>
<tr>
<td>HALT</td>
<td>4-74</td>
</tr>
<tr>
<td>INC(B)</td>
<td>4-8</td>
</tr>
<tr>
<td>IOT</td>
<td>4-68</td>
</tr>
<tr>
<td>JMP</td>
<td>4-56</td>
</tr>
<tr>
<td>JSR</td>
<td>4-58</td>
</tr>
<tr>
<td>JSR</td>
<td>4-58</td>
</tr>
<tr>
<td>BGT</td>
<td>4-49</td>
</tr>
<tr>
<td>BHI S</td>
<td>4-54</td>
</tr>
<tr>
<td>MOV(B)</td>
<td>4-23</td>
</tr>
<tr>
<td>MTPI</td>
<td>4-78</td>
</tr>
<tr>
<td>MUL</td>
<td>4-31</td>
</tr>
<tr>
<td>NEG(B)</td>
<td>4-10</td>
</tr>
<tr>
<td>NOP</td>
<td>4-79</td>
</tr>
<tr>
<td>RESET</td>
<td>4-76</td>
</tr>
<tr>
<td>ROL(B)</td>
<td>4-16</td>
</tr>
<tr>
<td>ROR(B)</td>
<td>4-15</td>
</tr>
<tr>
<td>RTI</td>
<td>4-69</td>
</tr>
<tr>
<td>RTS</td>
<td>4-60</td>
</tr>
<tr>
<td>RTT</td>
<td>4-70</td>
</tr>
<tr>
<td>SBC(B)</td>
<td>4-20</td>
</tr>
<tr>
<td>SOB</td>
<td>4-63</td>
</tr>
<tr>
<td>SUB</td>
<td>4-26</td>
</tr>
<tr>
<td>SWAB</td>
<td>4-17</td>
</tr>
<tr>
<td>SXT</td>
<td>4-21</td>
</tr>
<tr>
<td>TRAP</td>
<td>4-66</td>
</tr>
<tr>
<td>TST(B)</td>
<td>4-11</td>
</tr>
<tr>
<td>WAIT</td>
<td>4-75</td>
</tr>
<tr>
<td>XOR</td>
<td>4-35</td>
</tr>
<tr>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>00 00 00</td>
<td>HALT</td>
</tr>
<tr>
<td>00 00 01</td>
<td>WAIT</td>
</tr>
<tr>
<td>00 00 02</td>
<td>RTI</td>
</tr>
<tr>
<td>00 00 03</td>
<td>BPT</td>
</tr>
<tr>
<td>00 00 04</td>
<td>IOT</td>
</tr>
<tr>
<td>00 00 05</td>
<td>RESET</td>
</tr>
<tr>
<td>00 00 06</td>
<td>RTT</td>
</tr>
<tr>
<td>00 00 07</td>
<td>(unused)</td>
</tr>
<tr>
<td>00 01 DD</td>
<td>JMP</td>
</tr>
<tr>
<td>00 02 0R</td>
<td>RTS</td>
</tr>
<tr>
<td>00 02 27</td>
<td>(unused)</td>
</tr>
<tr>
<td>00 02 3N</td>
<td>SPL</td>
</tr>
<tr>
<td>00 02 40</td>
<td>NOP</td>
</tr>
<tr>
<td>00 02 41</td>
<td></td>
</tr>
<tr>
<td>00 02 77</td>
<td>cond codes</td>
</tr>
<tr>
<td>00 03 DD</td>
<td>SWAB</td>
</tr>
<tr>
<td>00 03 XX</td>
<td>BR</td>
</tr>
<tr>
<td>00 10 XX</td>
<td>BNE</td>
</tr>
<tr>
<td>00 14 XX</td>
<td>BEQ</td>
</tr>
<tr>
<td>00 20 XX</td>
<td>BGE</td>
</tr>
<tr>
<td>00 24 XX</td>
<td>BLT</td>
</tr>
<tr>
<td>00 30 XX</td>
<td>BGT</td>
</tr>
<tr>
<td>00 34 XX</td>
<td>BLE</td>
</tr>
<tr>
<td>00 34 XX</td>
<td>BLE</td>
</tr>
<tr>
<td>00 34 XX</td>
<td>BLE</td>
</tr>
<tr>
<td>00 4R DD</td>
<td>JSR</td>
</tr>
<tr>
<td>00 4R DD</td>
<td>JSR</td>
</tr>
<tr>
<td>00 50 DD</td>
<td>CLR</td>
</tr>
<tr>
<td>00 51 DD</td>
<td>COM</td>
</tr>
<tr>
<td>00 52 DD</td>
<td>INC</td>
</tr>
<tr>
<td>00 53 DD</td>
<td>DEC</td>
</tr>
<tr>
<td>00 54 DD</td>
<td>NEG</td>
</tr>
<tr>
<td>00 55 DD</td>
<td>ADC</td>
</tr>
<tr>
<td>00 56 DD</td>
<td>SBC</td>
</tr>
<tr>
<td>00 57 DD</td>
<td>TST</td>
</tr>
<tr>
<td>00 57 DD</td>
<td>TST</td>
</tr>
<tr>
<td>00 57 DD</td>
<td>TST</td>
</tr>
</tbody>
</table>

D-2
## APPENDIX E SUMMARY OF PDP11 INSTRUCTIONS

### GENERAL REGISTER ADDRESSING

<table>
<thead>
<tr>
<th>Mode</th>
<th>Name</th>
<th>Symbolic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>register</td>
<td>R</td>
<td>(R) is operand [ex. R2 = %2]</td>
</tr>
<tr>
<td>1</td>
<td>register deferred</td>
<td>(R)</td>
<td>(R) is address</td>
</tr>
<tr>
<td>2</td>
<td>auto-increment</td>
<td>(R)+</td>
<td>(R) is adrs; (R)+(1 or 2)</td>
</tr>
<tr>
<td>3</td>
<td>auto-incr deferred</td>
<td>@(R)+</td>
<td>(R) is adrs of adrs; (R)+2</td>
</tr>
<tr>
<td>4</td>
<td>auto-decrement</td>
<td>-(R)</td>
<td>(R) → (1 or 2); (R) is adrs</td>
</tr>
<tr>
<td>5</td>
<td>auto-decr deferred</td>
<td>@-(R)</td>
<td>(R) → 2; (R) is adrs of adrs</td>
</tr>
<tr>
<td>6</td>
<td>index</td>
<td>X(R)</td>
<td>(R)+X is adrs</td>
</tr>
<tr>
<td>7</td>
<td>index deferred</td>
<td>@X(R)</td>
<td>(R)+X is adrs of adrs</td>
</tr>
</tbody>
</table>

### PROGRAM COUNTER ADDRESSING

<table>
<thead>
<tr>
<th>Mode</th>
<th>Name</th>
<th>Symbolic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>immediate</td>
<td>#n</td>
<td>operand n follows instr</td>
</tr>
<tr>
<td>3</td>
<td>absolute</td>
<td>@ #A</td>
<td>address A follows instr</td>
</tr>
<tr>
<td>6</td>
<td>relative</td>
<td>A</td>
<td>instr adrs +4+X is adrs</td>
</tr>
<tr>
<td>7</td>
<td>relative deferred</td>
<td>@A</td>
<td>instr adrs +4+X is adrs of adrs</td>
</tr>
</tbody>
</table>

### LEGEND

**Op Codes**

- **#** = 0 for word/1 for byte
- **S** = source field (6 bits)
- **D** = destination field (6 bits)
- **R** = gen register (3 bits), 0 to 7
- **XXX** = offset (8 bits), +127 to -128
- **N** = number (3 bits)
- **NN** = number (6 bits)

**Boolean**

- **∧** = AND
- **v** = inclusive OR
- **✈** = exclusive OR
- **∼** = NOT

**Operations**

- ( ) = contents of
- **s** = contents of source
- **d** = contents of destination
- **r** = contents of register
- **←** = becomes
- **X** = relative address
- **%** = register definition

**Condition Codes**

- ***= conditionally set or cleared
- **-= not affected
- **0 = cleared
- **1 = set

**NOTE:**

- **△** = Applies to the 11/40, & 11/45 computers
- **●** = Applies to the 11/45 computer
## SINGLE OPERAND: OPR dst

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Instruction</th>
<th>dst</th>
<th>Result</th>
<th>NZVC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR(B)</td>
<td>050DD</td>
<td>clear</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>COM(B)</td>
<td>051DD</td>
<td>complement (1's)</td>
<td>~d</td>
<td>* *</td>
<td>0 1</td>
</tr>
<tr>
<td>INC(B)</td>
<td>052DD</td>
<td>increment</td>
<td>d + 1</td>
<td>* * *</td>
<td>*</td>
</tr>
<tr>
<td>DEC(B)</td>
<td>053DD</td>
<td>decrement</td>
<td>d - 1</td>
<td>* * *</td>
<td>*</td>
</tr>
<tr>
<td>NEG(B)</td>
<td>054DD</td>
<td>negate (2's compl)</td>
<td>-d</td>
<td>* * * *</td>
<td>*</td>
</tr>
<tr>
<td>TST(B)</td>
<td>057DD</td>
<td>test</td>
<td>d</td>
<td>* *</td>
<td>0 0</td>
</tr>
</tbody>
</table>

### Rotate & Shift

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROR(B)</td>
<td>060DD</td>
<td>rotate right</td>
<td>* * * *</td>
</tr>
<tr>
<td>ROL(B)</td>
<td>061DD</td>
<td>rotate left</td>
<td>* * * *</td>
</tr>
<tr>
<td>ASR(B)</td>
<td>062DD</td>
<td>arith shift right</td>
<td>d/2</td>
</tr>
<tr>
<td>ASL(B)</td>
<td>063DD</td>
<td>arith shift left</td>
<td>2d</td>
</tr>
<tr>
<td>SWAB</td>
<td>0003DD</td>
<td>swap bytes</td>
<td>* * * 0</td>
</tr>
</tbody>
</table>

### Multiple Precision

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC(B)</td>
<td>055DD</td>
<td>add carry</td>
<td>d + C</td>
</tr>
<tr>
<td>SBC(B)</td>
<td>056DD</td>
<td>subtract carry</td>
<td>d - C</td>
</tr>
<tr>
<td>SXT</td>
<td>0067DD</td>
<td>sign extend</td>
<td>0 or -1</td>
</tr>
</tbody>
</table>

## DOUBLE OPERAND: OPR src,dst

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Instruction</th>
<th>Operation</th>
<th>NZVC</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV(B)</td>
<td>1SSDD</td>
<td>move</td>
<td>d ← s</td>
<td>* * 0 _</td>
</tr>
<tr>
<td>CMP(B)</td>
<td>2SSDD</td>
<td>compare</td>
<td>s - d</td>
<td>* * * *</td>
</tr>
<tr>
<td>ADD</td>
<td>06SSDD</td>
<td>add</td>
<td>d ← s + d</td>
<td>* * * *</td>
</tr>
<tr>
<td>SUB</td>
<td>16SSDD</td>
<td>subtract</td>
<td>d ← d - s</td>
<td>* * * *</td>
</tr>
</tbody>
</table>

### Logical

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Instruction</th>
<th>Operation</th>
<th>NZVC</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT(B)</td>
<td>3SSDD</td>
<td>bit test (AND)</td>
<td>s ∧ d</td>
<td>* * 0 _</td>
</tr>
<tr>
<td>BIC(B)</td>
<td>4SSDD</td>
<td>bit clear</td>
<td>d ← (~s) ∧ d</td>
<td>* * 0 _</td>
</tr>
<tr>
<td>BIS(B)</td>
<td>5SSDD</td>
<td>bit set (OR)</td>
<td>d ← s ∨ d</td>
<td>* * 0 _</td>
</tr>
</tbody>
</table>

### Register

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Instruction</th>
<th>Operation</th>
<th>NZVC</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL</td>
<td>070RSS</td>
<td>multiply</td>
<td>r ← r x s</td>
<td>* * 0 *</td>
</tr>
<tr>
<td>DIV</td>
<td>071RSS</td>
<td>divide</td>
<td>r ← r/s</td>
<td>* * * *</td>
</tr>
<tr>
<td>ASH</td>
<td>072RSS</td>
<td>shift arithmetically</td>
<td>* * * *</td>
<td></td>
</tr>
<tr>
<td>ASHC</td>
<td>073RSS</td>
<td>arith shift combined</td>
<td>* * * *</td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td>074RDD</td>
<td>exclusive OR</td>
<td>d ← r ⊕ d</td>
<td>* * 0 _</td>
</tr>
</tbody>
</table>

E-2
### BRANCH B _ _ location

<table>
<thead>
<tr>
<th>Base Code</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 8 7 XXX</td>
<td></td>
</tr>
</tbody>
</table>

If condition is satisfied:
Branch to location,
New PC ← Updated PC + (2 x offset)

Op Code = Base Code + XXX

#### Base Mnemonic Code Instruction Branch Condition

### Branches

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Code</th>
<th>Instruction</th>
<th>Branch Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>BR</td>
<td>000400</td>
<td>branch (unconditional)</td>
<td>(always)</td>
</tr>
<tr>
<td>BNE</td>
<td>001000</td>
<td>br if not equal (to 0)</td>
<td>≠ 0 Z = 0</td>
</tr>
<tr>
<td>BEQ</td>
<td>001400</td>
<td>br if equal (to 0)</td>
<td>= 0 Z = 1</td>
</tr>
<tr>
<td>BPL</td>
<td>100000</td>
<td>branch if plus</td>
<td>+ N = 0</td>
</tr>
<tr>
<td>BMI</td>
<td>100400</td>
<td>branch if minus</td>
<td>N = 1</td>
</tr>
<tr>
<td>BVC</td>
<td>102000</td>
<td>br if overflow is clear</td>
<td>V = 0</td>
</tr>
<tr>
<td>BVS</td>
<td>102400</td>
<td>br if overflow is set</td>
<td>V = 1</td>
</tr>
<tr>
<td>BCC</td>
<td>103000</td>
<td>br if carry is clear</td>
<td>C = 0</td>
</tr>
<tr>
<td>BCS</td>
<td>103400</td>
<td>br if carry is set</td>
<td>C = 1</td>
</tr>
</tbody>
</table>

### Signed Conditional Branches

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Code</th>
<th>Instruction</th>
<th>Branch Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGE</td>
<td>002000</td>
<td>br if greater or eq (to 0)</td>
<td>≥ 0 N = V = 0</td>
</tr>
<tr>
<td>BLT</td>
<td>002400</td>
<td>br if less than (0)</td>
<td>&lt; 0 N = V = 1</td>
</tr>
<tr>
<td>BGT</td>
<td>003000</td>
<td>br if greater than (0)</td>
<td>&gt; 0 Z = ω (N + V) = 0</td>
</tr>
<tr>
<td>BLE</td>
<td>003400</td>
<td>br if less or equal (to 0)</td>
<td>≤ 0 Z = ω (N + V) = 1</td>
</tr>
</tbody>
</table>

### Unsigned Conditional Branches

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Code</th>
<th>Instruction</th>
<th>Branch Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>BHI</td>
<td>101000</td>
<td>branch if higher</td>
<td>C = Z = 0</td>
</tr>
<tr>
<td>BLOS</td>
<td>101400</td>
<td>branch if lower or same</td>
<td>C = Z = 1</td>
</tr>
<tr>
<td>BHIS</td>
<td>103000</td>
<td>branch if higher or same</td>
<td>C = 0</td>
</tr>
<tr>
<td>BLO</td>
<td>103400</td>
<td>branch if lower</td>
<td>C = 1</td>
</tr>
</tbody>
</table>

### JUMP & SUBROUTINE:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP</td>
<td>0001DD</td>
<td>jump</td>
<td>PC ← dst</td>
</tr>
<tr>
<td>JSR</td>
<td>004RDD</td>
<td>jump to subroutine</td>
<td></td>
</tr>
<tr>
<td>RTS</td>
<td>0020R</td>
<td>return from subroutine</td>
<td>use same R</td>
</tr>
<tr>
<td>MARK</td>
<td>0064NN</td>
<td>mark</td>
<td>aid in subr return</td>
</tr>
<tr>
<td>SOB</td>
<td>077RNN</td>
<td>subtract 1 &amp; br (if ≠ 0)</td>
<td>(R) — 1, then if (R) ≠ 0: PC ← Updated PC — (2 x NN)</td>
</tr>
</tbody>
</table>
### TRAP & INTERRUPT:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Code</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMT</td>
<td>104000 to 104377</td>
<td>emulator trap</td>
<td>PC at 30, PS at 32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(not for general use)</td>
<td></td>
</tr>
<tr>
<td>TRAP</td>
<td>104400 to 104777</td>
<td>trap</td>
<td>PC at 34, PS at 36</td>
</tr>
<tr>
<td>BPT</td>
<td>000003</td>
<td>breakpoint trap</td>
<td>PC at 14, PS at 16</td>
</tr>
<tr>
<td>IOT</td>
<td>000004</td>
<td>input/output trap</td>
<td>PC at 20, PS at 22</td>
</tr>
<tr>
<td>RTI</td>
<td>000002</td>
<td>return from interrupt</td>
<td></td>
</tr>
<tr>
<td>▲RTT</td>
<td>000006</td>
<td>return from interrupt</td>
<td>inhibit T bit trap</td>
</tr>
</tbody>
</table>

### MISCELLANEOUS:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Code</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>HALT</td>
<td>000000</td>
<td>halt</td>
</tr>
<tr>
<td>WAIT</td>
<td>000001</td>
<td>wait for interrupt</td>
</tr>
<tr>
<td>RESET</td>
<td>000005</td>
<td>reset external bus</td>
</tr>
<tr>
<td>NOP</td>
<td>000240</td>
<td>(no operation)</td>
</tr>
<tr>
<td>SPL</td>
<td>00023N</td>
<td>set priority level (to N)</td>
</tr>
<tr>
<td>▲MFPI</td>
<td>0065SS</td>
<td>move from previous instr space</td>
</tr>
<tr>
<td>▲MTPI</td>
<td>0066DD</td>
<td>move to previous instr space</td>
</tr>
<tr>
<td>▯MFPD</td>
<td>1065SS</td>
<td>move from previous data space</td>
</tr>
<tr>
<td>▯MTPD</td>
<td>1066DD</td>
<td>move to previous data space</td>
</tr>
</tbody>
</table>

### CONDITION CODE OPERATORS:

```
<table>
<thead>
<tr>
<th>Op Code</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLC</td>
<td>clear C</td>
</tr>
<tr>
<td>CLV</td>
<td>clear V</td>
</tr>
<tr>
<td>CLZ</td>
<td>clear Z</td>
</tr>
<tr>
<td>CLN</td>
<td>clear N</td>
</tr>
<tr>
<td>CCC</td>
<td>clear all cc bits</td>
</tr>
<tr>
<td>SEC</td>
<td>set C</td>
</tr>
<tr>
<td>SEV</td>
<td>set V</td>
</tr>
<tr>
<td>SEZ</td>
<td>set Z</td>
</tr>
<tr>
<td>SEN</td>
<td>set N</td>
</tr>
<tr>
<td>SCC</td>
<td>set all cc bits</td>
</tr>
</tbody>
</table>
```

```plaintext
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Code</th>
<th>Instruction</th>
<th>N Z V C</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLC</td>
<td>000241</td>
<td>clear C</td>
<td>_ _ _ 0</td>
</tr>
<tr>
<td>CLV</td>
<td>000242</td>
<td>clear V</td>
<td>_ _ 0 _</td>
</tr>
<tr>
<td>CLZ</td>
<td>000244</td>
<td>clear Z</td>
<td>_ 0 _ _</td>
</tr>
<tr>
<td>CLN</td>
<td>000250</td>
<td>clear N</td>
<td>0 _ _ _</td>
</tr>
<tr>
<td>CCC</td>
<td>000257</td>
<td>clear all cc bits</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>SEC</td>
<td>000261</td>
<td>set C</td>
<td>_ _ 1 _</td>
</tr>
<tr>
<td>SEV</td>
<td>000262</td>
<td>set V</td>
<td>_ 1 _ _</td>
</tr>
<tr>
<td>SEZ</td>
<td>000264</td>
<td>set Z</td>
<td>_ 1 _ _</td>
</tr>
<tr>
<td>SEN</td>
<td>000270</td>
<td>set N</td>
<td>1 _ _ _</td>
</tr>
<tr>
<td>SCC</td>
<td>000277</td>
<td>set all cc bits</td>
<td>1 1 1 1</td>
</tr>
</tbody>
</table>
```

E-4
## PDP11/40 Floating Point Unit:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Register</th>
<th>Description</th>
<th>N  Z  V  C</th>
</tr>
</thead>
<tbody>
<tr>
<td>FADD</td>
<td>07500R</td>
<td>Floating Add</td>
<td>*  *  0  0</td>
</tr>
<tr>
<td>FSUB</td>
<td>07501R</td>
<td>Floating Subtract</td>
<td>*  *  0  0</td>
</tr>
<tr>
<td>FMUL</td>
<td>07502R</td>
<td>Floating Multiply</td>
<td>*  *  0  0</td>
</tr>
<tr>
<td>FDIV</td>
<td>07503R</td>
<td>Floating Divide</td>
<td>*  *  0  0</td>
</tr>
</tbody>
</table>

## Device Register Addresses

<table>
<thead>
<tr>
<th>Device</th>
<th>Control &amp; Status</th>
<th>Data Buffer</th>
<th>Interrupt Vector</th>
<th>Priority Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>KW11-L</td>
<td>Line Clock</td>
<td>777 546</td>
<td>—</td>
<td>100 BR6</td>
</tr>
<tr>
<td>KW11-P</td>
<td>Real Time Clock</td>
<td>772 540</td>
<td>772 542</td>
<td>104 BR6</td>
</tr>
<tr>
<td></td>
<td>control &amp; status</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>counter</td>
<td>772 544</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LA30</td>
<td>DECwriter</td>
<td>777 560</td>
<td>777 562</td>
<td>60 BR4</td>
</tr>
<tr>
<td></td>
<td>keyboard</td>
<td>777 564</td>
<td>777 566</td>
<td>64 BR4</td>
</tr>
<tr>
<td></td>
<td>printer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LP11</td>
<td>Line Printer</td>
<td>777 514</td>
<td>777 516</td>
<td>200 BR4</td>
</tr>
<tr>
<td>LT33</td>
<td>Teletype</td>
<td>777 560</td>
<td>777 562</td>
<td>60 BR4</td>
</tr>
<tr>
<td></td>
<td>keyboard</td>
<td>777 564</td>
<td>777 566</td>
<td>64 BR4</td>
</tr>
<tr>
<td></td>
<td>printer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC11</td>
<td>Paper Tape</td>
<td>777 550</td>
<td>777 552</td>
<td>70 BR4</td>
</tr>
<tr>
<td></td>
<td>reader</td>
<td>777 554</td>
<td>777 556</td>
<td>74 BR4</td>
</tr>
<tr>
<td></td>
<td>punch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RC11/RS64</td>
<td>Disk (64K words)</td>
<td>777 456</td>
<td>777 456</td>
<td>210 BR5</td>
</tr>
<tr>
<td></td>
<td>look ahead</td>
<td>777 440</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>disk address</td>
<td>777 442</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>error status</td>
<td>777 444</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>command &amp; status</td>
<td>777 446</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>word count</td>
<td>777 450</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>current address</td>
<td>777 452</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>maintenance</td>
<td>777 454</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RF11/RS11</td>
<td>Disk (256K words)</td>
<td>777 472</td>
<td>777 472</td>
<td>204 BR5</td>
</tr>
<tr>
<td></td>
<td>control status</td>
<td>777 460</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>word count</td>
<td>777 462</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>current mem adrs</td>
<td>777 464</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>disk address</td>
<td>777 466</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>adrs ext error</td>
<td>777 470</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>maintenance</td>
<td>777 474</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>segment address</td>
<td>777 476</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RK11/RK05 Disk Cartridge
- drive status: 777 400
- error: 777 402
- control status: 777 404
- word count: 777 406
- current address: 777 410
- disk address: 777 412
- maintenance: 777 414

TC11/TU56 DECtape
- control: 777 340
- command: 777 342
- word count: 777 344
- current address: 777 346

TM11/TU10 Magtape
- status: 772 520
- command: 772 522
- byte counter: 772 524
- current address: 772 526
- read lines: 772 532

**PROCESSOR REGISTER ADDRESSES**

**Processor Status Word**

| PS | 777 776 |

- **Stack Limit Register** — 777 774
- **Program Interrupt Request** — 777 772

**General Registers**

- R0 — 777 700
- R1 — 777 701
- R2 — 777 702
- R3 — 777 703
- R4 — 777 704
- R5 — 777 705
- R6 — 777 706
- R7 — 777 707

**Console Switches & Display Register** — 777 570

**INTERRUPT VECTORS**

- 000 (reserved)
- 004 Time Out & other errors
- 010 illegal & reserved instr
- 014 BPT
- 020 IOT
- 024 Power Fail
- 030 EMT
- 034 TRAP
ABSOLUTE LOADER

Starting Address: ____ 500

Memory Size: 4K 017
8K 037
12K 057
16K 077
20K 117
24K 137
28K 157
(or larger)

BOOTSTRAP LOADER

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>____</td>
<td>744</td>
</tr>
<tr>
<td>____</td>
<td>746</td>
</tr>
<tr>
<td>____</td>
<td>750</td>
</tr>
<tr>
<td>____</td>
<td>752</td>
</tr>
<tr>
<td>____</td>
<td>754</td>
</tr>
<tr>
<td>____</td>
<td>756</td>
</tr>
<tr>
<td>____</td>
<td>760</td>
</tr>
<tr>
<td>____</td>
<td>762</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>____</td>
<td>016 701</td>
</tr>
<tr>
<td>____</td>
<td>000 026</td>
</tr>
<tr>
<td>____</td>
<td>012 702</td>
</tr>
<tr>
<td>____</td>
<td>000 352</td>
</tr>
<tr>
<td>____</td>
<td>005 211</td>
</tr>
<tr>
<td>____</td>
<td>105 711</td>
</tr>
<tr>
<td>____</td>
<td>100 376</td>
</tr>
<tr>
<td>____</td>
<td>116 162</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>____</td>
<td>764</td>
</tr>
<tr>
<td>____</td>
<td>766</td>
</tr>
<tr>
<td>____</td>
<td>770</td>
</tr>
<tr>
<td>____</td>
<td>772</td>
</tr>
<tr>
<td>____</td>
<td>774</td>
</tr>
<tr>
<td>____</td>
<td>776</td>
</tr>
</tbody>
</table>

000 0002
005 267
177 756
005 765
177 560 (KB)
000 750 (PR)

or 177 550 (PR)