Copyright © 1973, by Digital Equipment Corporation

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

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

1.1 THE PDP-11 CONCEPT ................................................... 1-1

1.2 COMPUTERS .............................................................. 1-2
   1.2.1 PDP-11/05 & PDP-11/10 ......................................... 1-2
   1.2.2 PDP-11/35 & PDP-11/40 ......................................... 1-3

1.3 PERIPHERALS/OPTIONS ................................................. 1-4
   1.3.1 I/O Devices .................................................... 1-4
   1.3.2 Storage Devices ............................................... 1-4

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

1.5 NUMBER SYSTEMS ........................................................ 1-6

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

2.1 UNIBUS ................................................................... 2-1
   2.1.1 Bidirectional Lines .............................................. 2-1
   2.1.2 Master-Slave Relation .......................................... 2-1
   2.1.3 Interlocked Communication .................................... 2-2

2.2 CENTRAL PROCESSOR ................................................. 2-2
   2.2.1 General Registers ............................................... 2-2
   2.2.2 Instruction Set .................................................. 2-3
   2.2.3 Processor Status Word ......................................... 2-4
   2.2.4 Stacks ................................................................ 2-5

2.3 PDP-11/05 & 11/10 INTERNAL CPU EQUIPMENT .................. 2-5

2.4 PDP-11/35 & 11/40 EQUIPMENT ....................................... 2-6

2.5 MEMORY .................................................................. 2-7

2.6 AUTOMATIC PRIORITY INTERRUPTS ................................ 2-9
   2.6.1 Using the Interrupts .............................................. 2-11
   2.6.2 Interrupt Procedure ............................................. 2-11
   2.6.3 Interrupt Servicing .............................................. 2-12

## CHAPTER 3  ADDRESSING MODES ........................................... 3-1

3.1 SINGLE OPERAND ADDRESSING ....................................... 3-2

3.2 DOUBLE OPERAND ADDRESSING ...................................... 3-2

3.3 DIRECT ADDRESSING .................................................. 3-4
   3.3.1 Register Mode ................................................... 3-4
   3.3.2 Auto-increment Mode ............................................ 3-5
   3.3.3 Auto-decrement Mode .......................................... 3-7
   3.3.4 Index Mode ....................................................... 3-8

3.4 DEFERRED (INDIRECT) ADDRESSING ................................. 3-10

3.5 USE OF THE PC AS A GENERAL REGISTER ....................... 3-12
   3.5.1 Immediate Mode ................................................ 3-13
   3.5.2 Absolute Addressing .......................................... 3-13
   3.5.3 Relative Addressing .......................................... 3-14
   3.5.4 Relative Deferred Addressing ................................. 3-15
3.6 USE OF STACK pointer AS GENERAL REGISTER 3-16
3.7 SUMMARY OF ADDRESSING MODES 3-16
   3.7.1 General Register Addressing 3-16
   3.7.2 Program Counter Addressing 3-18

CHAPTER 4 INSTRUCTION SET 4-1
  4.1 INTRODUCTION 4-1
  4.2 INSTRUCTION FORMATS 4-2
  4.3 LIST OF INSTRUCTIONS 4-4
  4.4 SINGLE OPERAND INSTRUCTIONS 4-6
  4.5 DOUBLE OPERAND INSTRUCTIONS 4-22
  4.6 PROGRAM CONTROL INSTRUCTIONS 4-32
  4.7 MISCELLANEOUS 4-70

CHAPTER 5 PROGRAMMING TECHNIQUES 5-1
  5.1 THE STACK 5-1
  5.2 SUBROUTINE LINKAGE 5-5
     5.2.1 Subroutine Calls 5-5
     5.2.2 Argument Transmission 5-6
     5.2.3 Subroutine Return 5-9
     5.2.4 PDP-11 Subroutine Advantage 5-9
  5.3 INTERRUPTS 5-9
     5.3.1 General Principles 5-9
     5.3.2 Nesting 5-10
  5.4 REENTRANCY 5-13
  5.5 POSITION INDEPENDENT CODE 5-15
  5.6 CO-ROUTINES 5-16
  5.7 PROCESSOR TRAPS 5-17
     5.7.1 Power Failure 5-17
     5.7.2 Odd Addressing Errors 5-17
     5.7.3 Time-Out Errors 5-17
     5.7.4 Reserved Instructions 5-17
     5.7.5 Trap Handling 5-17

CHAPTER 6 MEMORY MANAGEMENT (FOR THE 11/35 & 11/40) 6-1
  6.1 GENERAL 6-1
     6.1.1 Options 6-1
     6.1.2 Programming 6-1
     6.1.3 Basic Addressing 6-2
     6.1.4 Active Page Registers 6-2
     6.1.5 Capabilities Provided by Memory Management 6-3
  6.2 RELOCATION 6-3
     6.2.1 Virtual Addressing 6-3
     6.2.2 Program Relocation 6-4
  6.3 PROTECTION 6-6
     6.3.1 Inaccessible Memory 6-6
     6.3.2 Read-Only Memory 6-6
     6.3.3 Multiple Address Space 6-7
  6.4 ACTIVE PAGE REGISTERS 6-7
6.4.1 Page Address Registers ........................................... 6-8
6.4.2 Page Descriptor Registers ....................................... 6-8
6.5 VIRTUAL & PHYSICAL ADDRESSES .............................. 6-13
  6.5.1 Construction of a Physical Address .................. 6-13
  6.5.2 Determining the Program Physical Address ....... 6-14
6.6 STATUS REGISTERS .................................................. 6-15
  6.6.1 Status Register 0 .............................................. 6-15
  6.6.2 Status Register 2 .............................................. 6-17
6.7 INSTRUCTIONS ....................................................... 6-17
6.8 STACK LIMIT OPTION ................................................ 6-23

CHAPTER 7 ARITHMETIC OPTIONS (FOR THE 11/35 & 11/40) 7-1
  7.1 GENERAL ............................................................ 7-1
  7.2 EIS OPTION .......................................................... 7-1
  7.3 FLOATING POINT OPTION ........................................... 7-6

CHAPTER 8 CONSOLE OPERATION ........................................ 8-1
  8.1 PDP-11/05 & 11/10 CONSOLE ................................... 8-1
    8.1.1 Console Elements ........................................ 8-1
    8.1.2 Console Switches ........................................ 8-1
    8.1.3 Indicators ................................................ 8-3
  8.2 PDP-11/35 & 11/40 CONSOLE ................................... 8-6
    8.2.1 Console Elements ........................................ 8-6
    8.2.2 Status Indicators ....................................... 8-6
    8.2.3 Console Switches ........................................ 8-7
    8.2.4 Displays .................................................. 8-8

CHAPTER 9 SPECIFICATIONS ............................................. 9-1
  9.1 CPU OPERATING SPECIFICATIONS ............................... 9-1
  9.2 PACKAGING .......................................................... 9-1
  9.3 MOUNTING INFORMATION ......................................... 9-7
  9.4 TABLE OF SPECIFICATIONS ...................................... 9-9
  9.5 PDP-11 FAMILY OF COMPUTERS ................................... 9-10

Appendix A Programming Differences Between PDP-11 CPU's ........ A-1
Appendix B Memory Map .................................................. B-1
Appendix C Instruction Timing .......................................... C-1
Appendix D Instruction Index and Numerical Op Code List ........ D-1
Appendix E Summary of PDP-11 Instructions ......................... E-1
1.1 THE PDP-11 CONCEPT
In 1970, the first PDP-11 was introduced, marking a break with traditional small computers. To achieve a superior product, an extensive study was made on how small computers are used and what type of programming was needed. Typically, an installation has a wide variety of peripherals and customer designed interfaces. To handle this situation, the PDP-11 uses a single communications path called a UNIBUS, that allows easy data transfer and control with fast or slow, simple or complex devices. This unique bus design eliminates system obsolescence! Any particular component may be replaced by a faster or more sophisticated one, including the central processor, since timing is asynchronous. This means a PDP-11 system can take advantage of new technological breakthroughs.

The programming aspect of the investigation turned up several significant facts. Most programs operate on structured data; arrays, lists, matrices, and characters, as opposed to isolated data elements. The PDP-11 addressing architecture has eight different modes for efficient manipulation of the data structures. Input/output programming is accomplished in an innovative manner resulting in low overhead and easy implementation. The unique concepts in the PDP-11 resulted in the award of three patents.

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

All PDP-11 computers have the following features:
• 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 without the need to rotate, swap, or mask.

• Asynchronous operation
  system components run at their highest possible speed, replacement with faster devices means faster operation without other hardware or software changes
• Modular component design
  extreme ease and flexibility in configuring systems

• Stack processing
  hardware sequential memory manipulation makes it easy to handle
  structured data, subroutines, and interrupts

• Direct Memory Access (DMA)
  inherent in the architecture is direct memory access for multiple
  devices

• 8 general-purpose registers
  fast integrated circuits used for accumulators or address generation

• Automatic Priority Interrupt
  four-line, multi-level system permits grouping of interrupt lines
  according to response requirements

• Vectored interrupts
  fast interrupt response without device polling

• Single & double operand instructions
  powerful and convenient set of programming instructions

• Power Fail & Automatic Restart
  hardware detection and software protection for fluctuations in the
  AC power

### 1.2 COMPUTERS

This Handbook describes two central processors. They are offered in
several packaging arrangements, and with different services and software
to fill the diverse needs of the End User and the Original Equipment
Manufacturer (OEM). Four basic models will be described: PDP-11/05, PDP-11/10, PDP-11/35, and PDP-11/40. Another processor, the
PDP-11/45, is described in a separate handbook.

A PDP-11 processor 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 CPU performs all arith-
metic and logical operations required in the system.

#### 1.2.1 PDP-11/05 & PDP-11/10

The PDP-11/05 and the PDP-11/10 central processors are electrically
the same. Digital Equipment Corporation (DEC) offers the PDP-11/05
for the Original Equipment Manufacturer (OEM). As such it is sold in
those configurations and with those services that are convenient for the
OEM. The PDP-11/10 is offered for the End User, and is sold in con-
figurations that optimize its use with our small system software. More
services and software are included with the PDP-11/10 for the End User.

Both central processors are housed in a 5½” or 10½” high assembly
unit that mounts in a standard 19” rack. The PDP-11/05 can accept
between 4K and 28K words of memory; the PDP-11/10 comes standard
with 8K of core memory.
The PDP-11/05 and 11/10 are full-fledged computers that can execute all the basic PDP-11 instructions. They are the small, economical central processors with the large capability. They enjoy all the advantages of being a true member of the PDP-11 family and being able to use all the extensive developed software and peripheral equipment. If there is ever a need to upgrade to a faster or more powerful central processor, the PDP-11/05 (or 11/10) can simply be replaced by a different PDP-11 CPU, and software and peripherals remain the same in the system.

The PDP-11/05 and 11/10 include as standard item, hardware equipment that would be either necessary or very desirable in a usable configuration, such as:

- Input/Output computer terminal interface control
  The serial line interface can be used to operate a Teletype, a DEC-writer (LA30, 30 character/sec printer and input keyboard), or an Alphanumeric CRT Terminal (VT05, 240 character/sec display and input keyboard).

- Programmer’s Console
  Switches and display for entering and verifying data as well as controlling basic computer operations. There is a Power/Panel Lock switch with a removable key.

- Line Frequency Clock
  An internal timing signal derived from the power source for keeping track of when events happen. (Equivalent to the KW11-L clock option)

- Pre-wired Connector Slots
  The PDP-11/05 & PDP-11/10 are prewired to accept extra memory, communication interfaces, and standard peripheral device controllers. The included CPU power supply has sufficient excess capacity to handle optional internal equipment.

1.2.2 PDP-11/35 & PDP-11/40

The PDP-11/35 and the PDP-11/40 central processors are functionally identical. The 11/40 is packaged in a 21” high front panel slide chassis, which in turn is mounted in a standard DEC 72” cabinet, allowing convenient access and expansion. The 11/35 is mounted in a 10½” high slide mounted chassis for compactness. The computers were designed to fit a broad range of applications, from simple situations where the computer consists of only 8K of memory and an I/O device, to large multi-user, multi-task environments requiring up to 124K of core memory. The machines provide a balance between high-speed processing and economy coupled with expandability. The processor assembly is pre-wired to accept a Floating Point option, and a Memory Management option for addressing over 28K of core memory. Memory Management also provides relocation and protection, especially useful in a multi-user operation.

Included with the basic 11/35 & 11/40 are:

- 8K of 900 nsec core memory
- Programmer Console with LED display and removable key for Power/Panel Lock
- Power supply with excess capacity to drive internal optional equipment
• Prewired to accept Floating Point and Memory Management hardware options

Table 1.1 highlights some of the significant differences and similarities of the computers.

DEC References

PDP-11 Peripherals and Interfacing Handbook
Introduction to Programming
Small Computer Handbook
PDP-11 Computer Manuals

1.3 PERIPHERALS/OPTIONS

Digital Equipment Corporation (DEC) designs and manufactures many of the peripheral devices offered with PDP-11’s. As a designer and manufacturer of peripherals, DEC can offer extremely reliable equipment, lower prices, more choice and quantity discounts.

1.3.1 I/O Devices

All PDP-11 systems can use a Teletype as the basis I/O device. However, I/O capabilities can be increased with high-speed paper tape reader-punches, line printers, card readers or alphanumerics 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:

Cassette, TA11
DECTerminal alphanumeric display, VT05
DECwriter teleprinter, LA30
High Speed Line Printers, LS11, LP11
High Speed Paper Tape Reader and Punch, PC11
Teletypes, LT33
Card Readers, CR11, CD11
Graphics Terminal, GT40
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 RP03 Disk Pack which can store up to 20 million words.

1-4
<table>
<thead>
<tr>
<th></th>
<th>PDP-11/05</th>
<th>PDP-11/10</th>
<th>PDP-11/35</th>
<th>PDP-11/40</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Market</td>
<td>OEM</td>
<td>End User</td>
<td>OEM</td>
<td>End User</td>
</tr>
<tr>
<td>Front Panel Height</td>
<td>5¼&quot; or 10¼&quot;</td>
<td>5¼&quot; or 10¼&quot;</td>
<td>10½&quot;</td>
<td>21&quot;</td>
</tr>
<tr>
<td>Central processor</td>
<td>KD11-B</td>
<td></td>
<td>KD11-A</td>
<td></td>
</tr>
<tr>
<td>Max Memory Size (words)</td>
<td>28K (K = 1024)</td>
<td></td>
<td>124K</td>
<td></td>
</tr>
<tr>
<td>Max Address Space (words)</td>
<td>32K</td>
<td></td>
<td>128K</td>
<td></td>
</tr>
<tr>
<td>Instructions</td>
<td>basic set</td>
<td></td>
<td>basic set + 5 extra</td>
<td></td>
</tr>
<tr>
<td>Programming Modes</td>
<td>1</td>
<td></td>
<td>1 (std), or 2 (optional)</td>
<td></td>
</tr>
<tr>
<td>EQUIPMENT:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating Point</td>
<td>(software only)</td>
<td></td>
<td>option</td>
<td></td>
</tr>
<tr>
<td>Memory Management</td>
<td>(not available)</td>
<td></td>
<td>option</td>
<td></td>
</tr>
<tr>
<td>I/O Serial Interface</td>
<td>included with CPU</td>
<td></td>
<td>option</td>
<td>included</td>
</tr>
<tr>
<td>Line Frequency Clock</td>
<td>included with CPU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core memory included</td>
<td>4K or 8K</td>
<td>8K</td>
<td>8K</td>
<td>8K</td>
</tr>
<tr>
<td>Teletype or DECwriter</td>
<td>optional</td>
<td>optional</td>
<td>optional</td>
<td>included</td>
</tr>
<tr>
<td>Cabinet</td>
<td>optional</td>
<td>optional</td>
<td>optional</td>
<td>included</td>
</tr>
<tr>
<td>SERVICES:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warranty</td>
<td>30 day, return to factory</td>
<td>90 day, on-site</td>
<td>30 day, return to factory</td>
<td>90 day, on-site</td>
</tr>
<tr>
<td>Installation</td>
<td>optional</td>
<td>on-site</td>
<td>optional</td>
<td>on-site</td>
</tr>
<tr>
<td>Papertape System</td>
<td>optional</td>
<td>included</td>
<td>optional</td>
<td>included</td>
</tr>
<tr>
<td>Software</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Papertape Diagnostic</td>
<td>with first system</td>
<td>included</td>
<td>optional</td>
<td>included</td>
</tr>
<tr>
<td>Software</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maintenance Manual</td>
<td>with first system</td>
<td>included</td>
<td>optional</td>
<td>included</td>
</tr>
<tr>
<td>Training</td>
<td>optional</td>
<td>optional</td>
<td>optional</td>
<td>included</td>
</tr>
</tbody>
</table>
PDP-11 storage devices include:

- DECtape, TU56
- Magtape, TU10
- 64K word fixed-head disk, RS64
- 256K word fixed-head disk, RS11
- 1.2M word moving-head disk, RK05
- 20M word moving-head disk, RP03

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

1.4.1 Paper Tape Software
The Paper Tape Software system includes:

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

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

- Text Editor (ED11)
- MACRO Assembler (MACRO-11)
- Linker (LINK11)
- File Utilities Packages (PIP)
- On Line Debugging Technique (ODT11)
- Librarian (LIBR11)

1.4.3 Higher Level Languages

BASIC
PDP-11 users needing an interactive conversational language can use BASIC which can be run on the paper tape software system with only 8,192 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**

![Octal representation diagram]

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

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

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Octal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000000</td>
</tr>
<tr>
<td>(2^{18} - 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>Sign Bit</td>
</tr>
<tr>
<td>+32,766</td>
<td>0</td>
</tr>
<tr>
<td>+1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>-2</td>
<td>1</td>
</tr>
<tr>
<td>-32,767</td>
<td>1</td>
</tr>
<tr>
<td>most negative -32,768</td>
<td>1</td>
</tr>
</tbody>
</table>
CHAPTER 2

SYSTEM ARCHITECTURE

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

![Figure 2-1 PDP-11 System Simplified Block Diagram](image)

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

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

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

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

2.1.3 Interlocked Communication

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

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.

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

2.2 CENTRAL PROCESSOR

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

2.2.1 General Registers

The central processor contains 8 general registers which can be used for a variety of purposes. The registers can be used as accumulators, index
registers, autoincrement registers, autodecrement registers, or as stack pointers for temporary storage of data. Chapter 3 on Addressing describes these uses of the general registers in more detail. Arithmetic operations can be from one general register to another, from one memory or device register to another, or between memory or a device register and a general register. Refer to Figure 2-2.

![General Registers Diagram]

Figure 2-2 The General Registers

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

The R6 register is normally used as the Stack Pointer indicating the last entry in the appropriate stack (a common temporary storage area with “Last-in First-Out” characteristics).

2.2.2 Instruction Set

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

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

**PDP-11 Approach**

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

**Conventional Approach**

LDA A  
; load contents of memory location A into AC

ADD B  
; add contents of memory location B to AC

STA B  
; store result at location B

**Addressing**

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

### 2.2.3 Processor Status Word

![Processor Status Word Diagram](image)

The Processor Status word (PS), at location 777776, contains information on the current status of the PDP-11. This information includes the current processor priority: current and previous 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.

**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.

### 2.2.4 Stacks
In the PDP-11, a stack is a temporary data storage area which allows a program to make efficient use of frequently accessed data. A program can add or delete words or bytes within the stack. The stack uses the "last-in, first-out" concept; that is, various items may be added to a stack in sequential order and retrieved or deleted from the stack in reverse order. On the PDP-11, a stack starts at the highest location reserved for it and expands linearly downward to the lowest address as items are added. The stack is used automatically by program interrupts, subroutine calls, and trap instructions. When the processor is interrupted, the central processor status word and the program counter are saved (pushed) onto the stack area, 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.

### 2.3 PDP-11/05 & 11/10 INTERNAL CPU EQUIPMENT

**SCL, Serial Communication Line Interface**
The SCL interface is contained on one of the CPU modules. The interface is program compatible with the DL11-A, DEC’s standard serial interface option, and can handle speeds up to 2,400 baud. Specifically it can control:

- DECwriter, LA30, up to 30 characters/sec
- Alpha-numeric Terminal, VT05, up to 240 characters/sec
- Teletype, up to 10 characters/sec

The SCL interface is not connected to the UNIBUS; it is connected to the CPU by an internal bus. This means that there can be no NPR transfers on the SCL. It also means that a parallel LA30 cannot be used as the local I/O device, only a serial LA30 or a Teletype.
LTC, Line Time Clock
The clock is contained on one of the CPU modules. It is program compatible with the KW11-L, DEC’s standard line clock option. The clock senses the 50 or 60 Hz line frequency for internal timing.

Power Supply
The power supply can be operated from either 115 VAC or 230 VAC by a simple change of the power control (within the power supply assembly). The power supply has enough capacity to handle the CPU, 8K of memory, plus additional memory and optional equipment that can mount within the CPU assembly unit.

2.4 PDP-11/35 & 11/40 EQUIPMENT
The central processor is prewired to accept the following options:

- Extended Instruction Set, KE11-E
- Floating Point, KE11-F
- Memory Management, KT11-D
- Programmable Stack Limit, KJ11-A
- Real Time Clock, KW11-I
- I/O Terminal Control, DL11 or LC11

Extended Instruction Set & Floating Point Options
The Extended Instruction Set (EIS) option 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 uses the EIS as a prerequisite. 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.

The Floating Point Unit functions as an integral part of the PDP-11 processor, not as a bus device.

Memory Management Option
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

With this option 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. A software operating system can assure 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.
Bits 12 through 15 of the Processor Status word, see Figure 2-3, are used with the Memory Management option. Mode information includes the present mode, either Kernel or User (bits 15,14) and the mode the machine was in prior to the last interrupt or trap (bits 13,12).

Stack Limit Option
This option allows program control of the lower limit for permissible stack addresses. The normal boundary without this option is \((400)\_8\). If the program attempts to exceed this limit for stack addresses, an indication is given to the program by means of a trap.

The Stack Limit option is included with the Memory Management option.

Power Supply
The power supply can be operated from either 115 VAC or 230 VAC, by making simple equipment changes. The power supply has enough capacity to handle the CPU, 8K of memory, plus additional memory and optional equipment that can mount within the CPU assembly unit.

Other Equipment
The PDP-11/40 is supplied with the following:

- I/O terminal interface control logic
- DECwriter terminal or teletype
- 72” high, standard 19” wide cabinet
- 8K words of core memory

2.5 MEMORY
Memory Organization
A memory can be viewed as a series of locations, with a number (address) assigned to each location. Thus an 8,192-word PDP-11 memory could be shown as in Figure 2-4.

![Memory Addresses Diagram](image)

Figure 2-4 Memory Addresses
Because PDP-11 memories are designed to accommodate both 16-bit words and 8-bit bytes, the total number of addresses does not correspond to the number of words. An 8K-word memory can contain 16K bytes and consist of 037777 octal locations. Words always start at even-numbered locations.

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

![Figure 2-5 High & Low Byte](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 2-6.

![Figure 2-6 Word and Byte Addresses](image)

Certain memory locations have been reserved by the system for interrupt and trap handling, processor stacks, general registers, and peripheral device registers. Addresses from 0 to 370₉ are always reserved and those to 777₉ are reserved on large system configurations for traps and interrupt handling.

A 16-bit word used for byte addressing can address a maximum of 32K words. However, the top 4,096 word locations are reserved for peripheral and register addresses and the user therefore has 28K of core to program. With the PDP-11/35 & 11/40, the user can expand above 28K with the Memory Management option. This device provides an 18-bit
effective memory address which permits addressing up to 124K words of actual memory.

If the Memory Management option is not used, an octal address between 160 000 and 177 777 is interpreted as 760 000 to 777 777. That is, if bit 15, 14 and 13 are 1's, then bits 17 and 16 (the extended address bits) are considered to be 1's, which relocates the last 4K words (8K bytes) to become the highest locations accessed by the UNIBUS. Refer to Section 1.5.

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

![UNIBUS Priority Diagram]

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 service hardware selects and begins executing the appropriate service routine after having automatically saved the status of the interrupted program segment.

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

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

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

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

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

The processor's priority can be set under program control to one of eight levels using bits 7, 6, and 5 in the processor status register. These bits set a priority level that inhibits granting of bus requests on lower levels or on the same level. When the processor's priority is set to a level, for example 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 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-BR4) can take full advantage of the Central Processor by requesting an interrupt. In this way, the entire instruction set is available for manipulating data and status registers.

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

**2.6.2 Interrupt Procedure**

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

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

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

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

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

5. The device service routine can cause the processor to resume the interrupted process by executing the Return from Interrupt instruction, described in Chapter 4, which pops the two top words from the current processor stack and uses them to load the PC and PS registers.
A device routine can be interrupted by a higher priority bus request any time after the new PC and PS have been loaded. If such an interrupt occurs, the PC and PS of the service routine are automatically stored in the temporary registers and then pushed onto the new current stack, and the new device routine is initiated.

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.

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

Power Fail and Restart
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.
CHAPTER 3

ADDRESSING MDES

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

the function (operation code)

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

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

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

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

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

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

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

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

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

R7 is used by the processor as its program counter (PC). It is recommended that R7 not be used as a stack pointer.
An important PDP-11 feature, which must be considered in conjunction with the addressing modes, is the register arrangement:

Six general purpose registers, (R0-R5)

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

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

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

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

![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 is set to indicate deferred (indirect) addressing.

3.2 DOUBLE OPERAND ADDRESSING
Operations which imply two operands (such as add, subtract, move and compare) are handled by instructions that specify two addresses. The first operand is called the source operand, the second the destination operand. Bit assignments in the source and destination address fields may specify different modes and different registers. The Instruction format for the double operand instruction is:
The source address field is used to select the source operand, the first operand. The destination is used similarly, and locates the second operand and the result. For example, the instruction ADD A, B adds the contents (source operand) of location A to the contents (destination operand) of location B. After execution B will contain the result of the addition and the contents of A will be unchanged.

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

<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

OPR Rn

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

\[
\begin{align*}
R0 &= %0 \\
R1 &= %1 \\
R2 &= %2, \text{ etc.}
\end{align*}
\]

Register mode syntax requires that a general register be defined as follows:

- \(R0 = %0\)
- \(R1 = %1\)
- \(R2 = %2, \text{ etc.}\)

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

   \[
   \begin{array}{c|c}
   \text{BEFORE} & \text{AFTER} \\
   \hline
   R2 & 00002 \\
   R4 & 00004 \\
   \end{array}
   \]

3. COMB R4
   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)

   \[
   \begin{array}{c|c}
   \text{BEFORE} & \text{AFTER} \\
   \hline
   R4 & 02222 \\
   \end{array}
   \]

3.3.2 Autoincrement Mode

   OPR (Rn) +

   This mode provides for automatic stepping of a pointer through sequential ele-
ments of a table of operands. It assumes the contents of the selected general reg-
ister 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 se-
quential location. The autoincrement mode is especially useful for array process-
ing 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.

**BEFORE ADDRESS SPACE**

20000 005025
30000 11 116

**REGISTER**

R5 030000
30000 000000

**AFTER ADDRESS SPACE**

20000 005025
30000 000000

**REGISTER**

R5 030002

---

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>Octal Code</th>
<th>Instruction Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLRB (R5) +</td>
<td>105025</td>
<td>Clear Byte</td>
</tr>
</tbody>
</table>

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

**BEFORE ADDRESS SPACE**

20000 105025
30000 111 116

**REGISTER**

R5 030000
30002 000

**AFTER ADDRESS SPACE**

20000 105025
30002 000

**REGISTER**

R5 030001

---

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>Octal Code</th>
<th>Instruction Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD (R2) + ,R4</td>
<td>062204</td>
<td>Add</td>
</tr>
</tbody>
</table>

**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.

**BEFORE ADDRESS SPACE**

10000 062204
10002 010000

**REGISTERS**

R2 100002
R4 010000

**AFTER ADDRESS SPACE**

10000 062204
10002 010000

**REGISTERS**

R2 100004
R4 020000

---

3-6
3.3.3 Autodecrement Mode

OPR−(Rn)

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

**Autodecrement Mode Examples**

<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.

![Before Address Space](image1)

**BEFORE ADDRESS SPACE**

| 1000 | 005240 |
| 17774 | 000000 |

**REGISTERS**

| R0 | 017776 |

**AFTER ADDRESS SPACE**

| 1000 | 005240 |
| 17774 | 000001 |

**REGISTER**

| R0 | 017774 |

2. INC−(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.

![Before Address Space](image2)

**BEFORE ADDRESS SPACE**

| 1000 | 105240 |
| 17774 | 000 000 |

**REGISTERS**

| R0 | 017776 |

**AFTER ADDRESS SPACE**

| 1000 | 105240 |
| 17774 | 001 000 |

**REGISTER**

| R0 | 017775 |

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>1. 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  
                   000200

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. \textbf{ADD 30(R2),20(R5) 066265 000030 000020} \hspace{1cm} \textbf{Add}

**Operation:**

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

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

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

<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.
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

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 instruction and its objects are moved in such a way that the relative distance between them is not altered, the same offset relative to the PC can be used in all positions in memory. Thus, PIC usually references locations relative to the current location. PIC is discussed in more detail in Chapter 5.

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

### 3.5.1 Immediate Mode

**OPR \#n,DD**

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

**Immediate Mode Example**

<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</td>
</tr>
<tr>
<td></td>
<td>000010</td>
<td></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 second word of the instruction) before being incremented by two to point to the next instruction.

![Diagram of Immediate Mode Operation](image)

### 3.5.2 Absolute Addressing

**OPR @#A**

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

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>Octal Code</th>
<th>Instruction Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR @ #1100</td>
<td>005037</td>
<td>Clear</td>
</tr>
<tr>
<td></td>
<td>001100</td>
<td></td>
</tr>
</tbody>
</table>

Operation: Clear the contents of location 1100.

BEFORE
ADDRESS SPACE

<table>
<thead>
<tr>
<th>20</th>
<th>005037</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>001100</td>
</tr>
<tr>
<td>1100</td>
<td>????</td>
</tr>
<tr>
<td>1102</td>
<td></td>
</tr>
</tbody>
</table>

AFTER
ADDRESS SPACE

<table>
<thead>
<tr>
<th>20</th>
<th>005037</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>001100</td>
</tr>
<tr>
<td>24</td>
<td>000000</td>
</tr>
<tr>
<td>1100</td>
<td></td>
</tr>
<tr>
<td>1102</td>
<td></td>
</tr>
</tbody>
</table>

2. ADD @ #2000,R3 063703 002000


BEFORE
ADDRESS SPACE

<table>
<thead>
<tr>
<th>20</th>
<th>063703</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>002000</td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>000300</td>
</tr>
</tbody>
</table>

REGISTER

| R3 | 000500 |

AFTER
ADDRESS SPACE

<table>
<thead>
<tr>
<th>20</th>
<th>063703</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>002000</td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>000300</td>
</tr>
</tbody>
</table>

REGISTER

| R3 | 001000 |

3.5.3 Relative Addressing

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

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

Symbolic | Octal Code | Instruction Name
--- | --- | ---
INC A | 005267, 000054 | Increment

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

3.5.4 Relative Deferred Addressing

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

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

Relative Deferred Mode Example

Symbolic | Octal Code | Instruction Name
--- | --- | ---
CLR @A | 005077, 000020 | Clear

Operation: Add second word of instruction to PC to produce address of address of operand. Clear operand.
3.6 USE OF STACK POINTER AS GENERAL REGISTER
The processor stack pointer (SP, Register 6) is in most cases the general register used for the stack operations related to program nesting. Autodecrement with Register 6 "pushes" data on to the stack and autoincrement with Register 6 "pops" data off the stack. Index mode with SP permits random access of items on the stack. Since the SP is used by the processor for interrupt handling, it has a special attribute: autoincrements and autodecrements are always done in steps of two. Byte operations using the SP in this way leave odd addresses unmodified.

3.7 SUMMARY OF ADDRESSING MODES

3.7.1 General Register Addressing

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

Mode 0
Register OPR R
R contains operand

Mode 1
Register deferred OPR (R)
R contains address

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

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

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

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

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

Register = 7

Mode 2  Immediate  OPR #n  Operand n follows instruction

Mode 3  Absolute  OPR @#A  Address A follows instruction

Mode 6  Relative  OPR A  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"
Λ = boolean AND
v = boolean OR
⊕ = exclusive OR
~ = boolean not
Reg or R = register
B = Byte

= \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
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:

![Diagram showing byte addresses]

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 instruction set is shown in the following sequence.

▲—A triangle indicates instructions not implemented in the PDP-11/05 and 11/10.

## SINGLE OPERAND

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Op Code</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td></td>
<td></td>
<td></td>
</tr>
<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>
<tr>
<td>Shift &amp; Rotate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASR(B)</td>
<td>arithmetic shift right</td>
<td>062DD</td>
<td>4-13</td>
</tr>
<tr>
<td>ASL(B)</td>
<td>arithmetic shift left</td>
<td>063DD</td>
<td>4-14</td>
</tr>
<tr>
<td>ROR(B)</td>
<td>rotate right</td>
<td>060DD</td>
<td>4-15</td>
</tr>
<tr>
<td>ROL(B)</td>
<td>rotate left</td>
<td>061DD</td>
<td>4-16</td>
</tr>
<tr>
<td>SWAB</td>
<td>swap bytes</td>
<td>0003DD</td>
<td>4-17</td>
</tr>
<tr>
<td>Multiple Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADC(B)</td>
<td>add carry</td>
<td>055DD</td>
<td>4-19</td>
</tr>
<tr>
<td>SBC(B)</td>
<td>subtract carry</td>
<td>056DD</td>
<td>4-20</td>
</tr>
<tr>
<td>▲ SXT</td>
<td>sign extend</td>
<td>0067DD</td>
<td>4-21</td>
</tr>
</tbody>
</table>

## DOUBLE OPERAND

<table>
<thead>
<tr>
<th>General</th>
<th></th>
<th>Op Code</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV(B)</td>
<td>move source to destination</td>
<td>1SSDD</td>
<td>4-23</td>
</tr>
<tr>
<td>CMP(B)</td>
<td>compare src to dst</td>
<td>2SSDD</td>
<td>4-24</td>
</tr>
<tr>
<td>ADD</td>
<td>add src to dst</td>
<td>06SSDD</td>
<td>4-25</td>
</tr>
<tr>
<td>SUB</td>
<td>subtract src from dst</td>
<td>16SSDD</td>
<td>4-26</td>
</tr>
<tr>
<td>Logical</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BIT(B)</td>
<td>bit test</td>
<td>3SSDD</td>
<td>4-28</td>
</tr>
<tr>
<td>BIC(B)</td>
<td>bit clear</td>
<td>4SSDD</td>
<td>4-29</td>
</tr>
<tr>
<td>BIS(B)</td>
<td>bit set</td>
<td>5SSDD</td>
<td>4-30</td>
</tr>
<tr>
<td>▲ XOR</td>
<td>exclusive OR</td>
<td>074RDD</td>
<td>4-31</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>BR</td>
<td>branch (unconditional)</td>
<td>000400</td>
<td>4-33</td>
</tr>
<tr>
<td>BNE</td>
<td>branch if not equal (to zero)</td>
<td>001000</td>
<td>4-34</td>
</tr>
<tr>
<td>BEQ</td>
<td>branch if equal (to zero)</td>
<td>001400</td>
<td>4-35</td>
</tr>
<tr>
<td>BPL</td>
<td>branch if plus</td>
<td>100000</td>
<td>4-36</td>
</tr>
<tr>
<td>BMI</td>
<td>branch if minus</td>
<td>100400</td>
<td>4-37</td>
</tr>
<tr>
<td>BVC</td>
<td>branch if overflow is clear</td>
<td>102000</td>
<td>4-38</td>
</tr>
<tr>
<td>BVS</td>
<td>branch if overflow is set</td>
<td>102400</td>
<td>4-39</td>
</tr>
<tr>
<td>BCC</td>
<td>branch if carry is clear</td>
<td>103000</td>
<td>4-40</td>
</tr>
<tr>
<td>BCS</td>
<td>branch if carry is set</td>
<td>103400</td>
<td>4-41</td>
</tr>
</tbody>
</table>

## Signed Conditional Branch

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Op Code or Base Code</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGE</td>
<td>branch if greater than or equal (to zero)</td>
<td>002000</td>
<td>4-43</td>
</tr>
<tr>
<td>BLT</td>
<td>branch if less than (zero)</td>
<td>002400</td>
<td>4-44</td>
</tr>
<tr>
<td>BGT</td>
<td>branch if greater than (zero)</td>
<td>003000</td>
<td>4-45</td>
</tr>
<tr>
<td>BLE</td>
<td>branch if less than or equal (to zero)</td>
<td>003400</td>
<td>4-46</td>
</tr>
</tbody>
</table>

## Unsigned Conditional Branch

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Op Code or Base Code</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BHI</td>
<td>branch if higher</td>
<td>101000</td>
<td>4-48</td>
</tr>
<tr>
<td>BLOS</td>
<td>branch if lower or same</td>
<td>101400</td>
<td>4-49</td>
</tr>
<tr>
<td>BHIS</td>
<td>branch if higher or same</td>
<td>103000</td>
<td>4-50</td>
</tr>
<tr>
<td>BLO</td>
<td>branch if lower</td>
<td>103400</td>
<td>4-51</td>
</tr>
</tbody>
</table>

## Jump & Subroutine

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Op Code or Base Code</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP</td>
<td>jump</td>
<td>0001DD</td>
<td>4-52</td>
</tr>
<tr>
<td>JSR</td>
<td>jump to subroutine</td>
<td>004RDD</td>
<td>4-54</td>
</tr>
<tr>
<td>RTS</td>
<td>return from subroutine</td>
<td>00020R</td>
<td>4-56</td>
</tr>
<tr>
<td>▲ MARK</td>
<td>mark</td>
<td>006400</td>
<td>4-57</td>
</tr>
<tr>
<td>▲ SOB</td>
<td>subtract one and branch (if ≠ 0)</td>
<td>077R00</td>
<td>4-59</td>
</tr>
</tbody>
</table>

## Trap & Interrupt

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Op Code or Base Code</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMT</td>
<td>emulator trap</td>
<td>104000—104377</td>
<td>4-61</td>
</tr>
<tr>
<td>TRAP</td>
<td>trap</td>
<td>104400—104777</td>
<td>4-62</td>
</tr>
<tr>
<td>BPT</td>
<td>breakpoint trap</td>
<td>000003</td>
<td>4-63</td>
</tr>
<tr>
<td>IOT</td>
<td>input/output trap</td>
<td>000004</td>
<td>4-64</td>
</tr>
<tr>
<td>RTI</td>
<td>return from interrupt</td>
<td>000002</td>
<td>4-65</td>
</tr>
<tr>
<td>▲ RTT</td>
<td>return from interrupt</td>
<td>000006</td>
<td>4-66</td>
</tr>
</tbody>
</table>

## MISCELLANEOUS

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Op Code or Base Code</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>HALT</td>
<td>halt</td>
<td>000000</td>
<td>4-70</td>
</tr>
<tr>
<td>WAIT</td>
<td>wait for interrupt</td>
<td>000001</td>
<td>4-71</td>
</tr>
<tr>
<td>RESET</td>
<td>reset external bus</td>
<td>000005</td>
<td>4-72</td>
</tr>
</tbody>
</table>

## Condition Code Operation

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Op Code or Base Code</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLC, CLV, CLZ, CLN, CCC</td>
<td>clear</td>
<td>000240</td>
<td>4-73</td>
</tr>
<tr>
<td>SEC, SEV, SEZ, SEN, SCC</td>
<td>set</td>
<td>000260</td>
<td>4-73</td>
</tr>
</tbody>
</table>
4.4 SINGLE OPERAND INSTRUCTIONS

CLR
CLRB

clear destination

\[ \begin{array}{cccccccc}
0/1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
15 & 6 & 5 & 0
\end{array} \]

Operation: \( \text{(dst)} \ll 0 \)

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

0/1 0 0 0 1 0 1 0 0 1 d d d d d d d

15 6 5 0

Operation: \((\text{dst})\leftarrow \sim(\text{dst})\)

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

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

Byte: Same

Example: COM R0

Before
(R0) = 013333

NZVC
0110

After
(R0) = 164444

NZVC
1001

4-7
INC
INCB

increment dst

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

Operation: \( (\text{dst}) \leftarrow (\text{dst}) + 1 \)

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

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

Example:

<table>
<thead>
<tr>
<th>Before</th>
<th>INC R2</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R2) = 000333</td>
<td>(R2) = 000334</td>
<td></td>
</tr>
</tbody>
</table>

\[ \begin{array}{cccccc}
N & Z & V & C \\
0 & 0 & 0 & 0 & 0 & 0
\end{array} \]
DEC
DECB

decrement dst

\[ \text{Operation: } (\text{dst}) \leftarrow (\text{dst}) - 1 \]

\text{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

\text{Description:}
Word: Subtract 1 from the contents of the destination
Byte: Same

\text{Example:}

\begin{align*}
\text{Before} & \quad \text{DEC R5} & \quad \text{After} \\
(R5) &= 000001 & \quad (R5) &= 000000 \\
N & \quad Z & \quad V & \quad C \\
1 & \quad 0 & \quad 0 & \quad 0 \\
1 & \quad 0 & \quad 0 & \quad 0
\end{align*}
NEG
NEGB

 negate dst

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

15 6 5 0

Operation: \((dst)\leftarrow -(dst)\)

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

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

Byte: Same

Example: NEG R0

Before
\((R0) = 000010\)

\[\begin{array}{cccc}
N & Z & V & C \\
0 & 0 & 0 & 0 \\
\end{array}\]

After
\((R0) = 177770\)

\[\begin{array}{cccc}
N & Z & V & C \\
1 & 0 & 0 & 1 \\
\end{array}\]
test dst

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

**Operation:** \((\text{dst}) \leftrightarrow \text{dst})\)

**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>(R1) = 012340</td>
<td></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

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

15  6   5   0

Operation: \((\text{dst}) \leftarrow (\text{dst})\) shifted one place to the right

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

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

Word:

Byte:

 Odd Address 8

 Even Address 0
ASL
ASLB

arithmetic shift left  063DD

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

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

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

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

Word:

Byte:
rotate right

\[ \text{060DD} \]

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

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

**Example:**

**Word:**

![Word Diagram](image1)

**Byte:**

![Byte Diagram](image2)
**ROL**

**ROLB**

rotate left

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

**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:**

![Word Diagram](image)

**Bytes:**

![Byte Diagram](image)
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:

\[
\begin{array}{c|c}
\text{Before} & \text{After} \\
\hline
\text{(R1)} = 077777 & \text{(R1)} = 177577 \\
\hline
\text{NZVC} & \text{NZVC} \\
\text{1111} & \text{0000}
\end{array}
\]
**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 showing 32-bit word and operand positions]

**Example:**

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

\[
-1 = 377777777777
\]

(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 377777777776 or \(-2\)
add carry

0/1 0 0 0 1 0 1 1 0 1  d  d  d  d  d  d  d  d
0 6 5

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 077777 and \((C)\) was 1; cleared otherwise
- C: set if \((dst)\) was 177777 and \((C)\) was 1; cleared otherwise

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

Byte: Same

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

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

subtract carry

0/1 0 0 0 1 0 1 1 1 1 0 d d d d d d d

Operation: \((dst)\rightarrow (dst)-(C)\)

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

(not in 11/05 & 11/10)

0067DD

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

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

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

Example:  

Before  
(A) = 012345  

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

After  
(A) = 177777  

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
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.
move source to destination

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

Operation: \((\text{dst}) \rightarrow (\text{src})\)

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

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

Example:

\text{MOV} \ XXX, R1 ; loads Register 1 with the contents of memory location; \(XXX\) represents a programmer-defined mnemonic used to represent a memory location

\text{MOV} \ #20, R0 ; loads the number 20 into Register 0; \# indicates that the value 20 is the operand

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

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

\text{MOV} \ R1, R3 ; performs an inter register transfer

\text{MOVB} \ @#177562, @#177566 ; moves a character from terminal keyboard buffer to terminal buffer
CMP
CMPB

compare src to dst

\[ \text{Operation: } (\text{src})-(\text{dst}) \] (in detail, \( (\text{src}) + \sim (\text{dst}) + 1 \))

\[ \text{Condition Codes: } \]
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

\[ \text{Description: } \]
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 \( (\text{src})-(\text{dst}) \), not \( (\text{dst})-(\text{src}) \).
ADD

add src to dst

\[
\begin{array}{cccccccccccccc}
0 & 1 & 1 & 0 & | & s & s & s & s & s & | & d & d & d & d & d & d & d & d & d & d & d & 0
\end{array}
\]

Operation: \( (dst) \leftarrow (src) + (dst) \)

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

Description: Adds the source operand to the destination operand and stores the result at the destination address. The original contents of the destination are lost. The contents of the source are not affected. Two's complement addition is performed.

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

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

subtract src from dst

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

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

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:

<table>
<thead>
<tr>
<th></th>
<th>SUB R1,R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before</td>
<td>After</td>
</tr>
<tr>
<td>(R1) = 011111</td>
<td>(R1) = 011111</td>
</tr>
<tr>
<td>(R2) = 012345</td>
<td>(R2) = 001234</td>
</tr>
<tr>
<td>N Z V C</td>
<td>N Z V C</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>
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

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

\[
\begin{array}{c|c|c|c|c|c|c|c}
15 & 12 & 11 & 6 & 5 & 4 & 3 & 0 \\
\hline
& & & & & & & \\
\end{array}
\]

Operation: \((\text{src}) \land (\text{dst})\)

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 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 \(\neq 30.R3\)

\(\text{test bits 3 and 4 of R3 to see if both are off}\)

\((30)_2=0 \quad 000 \quad 000 \quad 000 \quad 011 \quad 000\)
bit clear

\[ \begin{array}{cccccccc}
0/1 & 1 & 0 & 0 & s & s & s & s & d & d & d & d & d & d & 0 \\
15 & 12 & 11 & 6 & 5 & 4 & 3 & 2 & 1 & 0
\end{array} \]

**Operation:** \( (\text{dst}) \leftarrow \sim (\text{src}) \land (\text{dst}) \)

**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:** 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**

<table>
<thead>
<tr>
<th>R3</th>
<th>R4</th>
<th>NZVC</th>
<th>NZVC</th>
</tr>
</thead>
<tbody>
<tr>
<td>001234</td>
<td>001111</td>
<td>1111</td>
<td>0001</td>
</tr>
</tbody>
</table>

**After**

<table>
<thead>
<tr>
<th>R3</th>
<th>R4</th>
</tr>
</thead>
<tbody>
<tr>
<td>001234</td>
<td>000101</td>
</tr>
</tbody>
</table>

**Before:**

\[ \begin{array}{ccccccc}
(R3) = & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\
(R4) = & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1
\end{array} \]

**After:**

\[ \begin{array}{ccccccc}
(R4) = & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array} \]
BIS
BISB

bit set

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

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:

Before
\[
\begin{align*}
\text{(R0)} &= 001234 \\
\text{(R1)} &= 001111
\end{align*}
\]
\[
\begin{align*}
N &\quad Z &\quad V &\quad C \\
0 &\quad 0 &\quad 0 &\quad 0
\end{align*}
\]

After
\[
\begin{align*}
\text{(R0)} &= 001234 \\
\text{(R1)} &= 001335
\end{align*}
\]
\[
\begin{align*}
N &\quad Z &\quad V &\quad C \\
0 &\quad 0 &\quad 0 &\quad 0
\end{align*}
\]

Before:
\[
\begin{align*}
\text{(R0)} &= 0 \quad 000 \quad 001 \quad 010 \quad 011 \quad 100 \\
\text{(R1)} &= 0 \quad 000 \quad 001 \quad 001 \quad 001 \quad 001
\end{align*}
\]

After:
\[
\begin{align*}
\text{(R1)} &= 0 \quad 000 \quad 001 \quad 011 \quad 011 \quad 101
\end{align*}
\]
XOR

exclusive OR

0111 0 0 r r r d d d d d d
d 15 9 8 6 5
0

Operation: \((\text{dst}) \leftarrow \text{Rv}(\text{dst})\)

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:

**Before**

\[
(\text{R0}) = 001234 \\
(\text{R2}) = 001111
\]

**After**

\[
(\text{R0}) = 001234 \\
(\text{R2}) = 000325
\]

**Before:**

\[
(\text{R0}) = 0 000 001 010 011 100 \\
(\text{R2}) = 0 000 001 001 001 001
\]

**After:**

\[
(\text{R2}) = 0 000 000 011 010 101
\]
4.6 PROGRAM CONTROL INSTRUCTIONS

Branches

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

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

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

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

The 8-bit offset allows branching in the backward direction by 200, 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:

\[ Bxx \quad \text{loc} \]

Where \(Bxx\) is the branch instruction and \(\text{loc}\) 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.
branch (unconditional) 000400 Plus offset

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

Offset:

\[PC \leftarrow PC + (2 \times \text{offset})\]

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) 001000 Plus offset

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

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

\begin{align*}
\text{CMP} & \quad A, B \quad \text{; compare A and B} \\
\text{BNE} & \quad C \quad \text{; branch if they are not equal}
\end{align*}

will branch to C if \( A \neq B \)

and the sequence

\begin{align*}
\text{ADD} & \quad A, B \quad \text{; add A to B} \\
\text{BNE} & \quad C \quad \text{; Branch if the result is not equal to 0}
\end{align*}

will branch to C if \( A + B \neq 0 \)
BEQ

branch if equal (to zero) 001400 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>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<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>
<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:

\[
\text{CMP A,B } \quad : \text{compare A and B} \\
\text{BEQ C } \quad : \text{branch if they are equal} \\
\]

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

\[
\text{ADD A,B } \quad : \text{add A to B} \\
\text{BEQ C } \quad : \text{branch if the result = 0} \\
\]

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

branch if plus

100000 Plus offset

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

OFFSET

15 8 7 0

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

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

branch if minus

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

OFFSET

100400 Plus offset

**Operation:**

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

**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

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

| OFFSET |

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

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

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

102400 Plus offset

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

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

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

103400 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>1</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>

**Operation:** $PC \leftarrow PC + (2 \times \text{offset})$ if $C = 1$.

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

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

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

<table>
<thead>
<tr>
<th>Largest</th>
<th>077777</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>077776</td>
</tr>
<tr>
<td>Positive</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>000001</td>
</tr>
<tr>
<td></td>
<td>000000</td>
</tr>
<tr>
<td></td>
<td>177777</td>
</tr>
<tr>
<td></td>
<td>177776</td>
</tr>
<tr>
<td>Negative</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>100001</td>
</tr>
<tr>
<td>Smallest</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100000</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Highest</th>
<th>177777</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>000002</td>
</tr>
<tr>
<td></td>
<td>000001</td>
</tr>
<tr>
<td>Lowest</td>
<td></td>
</tr>
<tr>
<td></td>
<td>000000</td>
</tr>
</tbody>
</table>
branch if greater than or equal (to zero) 002000 Plus offset

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

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

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)

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 1 & 002400 \text{ Plus offset} \\
15 & 8 & 7 & \text{OFFSET} & 0
\end{array}
\]

**Operation:**

\[\text{PC} \leftarrow \text{PC} + (2 \times \text{offset}) \text{ if } N \oplus V = 1\]

**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

```
15 8 7 0
0 0 0 0 0 1 1 0
```

**Operation:** \( PC \leftarrow PC + (2 \times \text{offset}) \) if \( Z \lor (N \lor V) = 0 \)

**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

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

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

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

15  0  0  0  0  0  1  0

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

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

branch if lower or same

101400 Plus offset

<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>1</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>

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

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

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

Description:  BHIS is the same instruction as BCC. This mnemonic is included only for convenience.
branch if lower 103400 Plus offset

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

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

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

jump 0001DD

<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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</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 O. Execution of a jump with mode O 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>15</th>
<th>9</th>
<th>8</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r</td>
<td>r</td>
<td>r</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d</td>
<td>d</td>
<td>d</td>
</tr>
</tbody>
</table>

**Operation:**

- `(tmp)←(dst)`  *(tmp is an internal processor register)*
- `r(SP)←reg`    *(push reg contents onto processor stack)*
- `reg←PC`       *(PC holds location following JSR; this address now put in reg)*
- `PC←(tmp)`     *(PC now points to subroutine destination)*

**Description:**

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

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

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

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>R7</td>
</tr>
<tr>
<td>(SP) R6</td>
<td>R6</td>
</tr>
<tr>
<td></td>
<td>R5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>After:</th>
</tr>
</thead>
<tbody>
<tr>
<td>R7</td>
</tr>
<tr>
<td>R6</td>
</tr>
<tr>
<td>R5</td>
</tr>
</tbody>
</table>

Stack

PC

n

n-2

PC+2

DATA 0

DATA 0

#1

#1
RTS

return from subroutine 00020R

Operation: \[ PC = reg \]
\[ reg = (SP) + 1 \]

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

(SP) R6

R5

After:

R7

R6

R5

Stack

DATA 0

#1

DATA 0

#1
(not in 11/05 & 11/10)

mark 00 64 NN

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & n & n & n & n & n & n
\end{array}
\]

15 8 7 6 5 0

Operation:
- \( \text{SP} \leftarrow \text{SP} + 2 \times \text{nn} \)
- \( \text{PC} \leftarrow \text{R5} \)
- \( \text{R5} \leftarrow (\text{SP}) \land \)

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

At this point the stack is as follows:

<table>
<thead>
<tr>
<th>OLD R5</th>
<th>P1</th>
<th>PN</th>
<th>MARK N</th>
<th>OLD PC</th>
</tr>
</thead>
</table>

4-57
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 the old R5 to be popped into R5 thus completing the return from subroutine.
SOB

(not in the 11/05 & 11/10)

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

\[
\begin{array}{cccccc}
0 & 1 & 1 & 1 & 1 & 1 \\
5 & 9 & 8 & 6 & 5 & 0 \\
\end{array}
\]

077R00 Plus offset

Operation: \( R \leftarrow R - 1 \) if this result \( \neq 0 \) 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:

\[
\text{SOB} \quad R,A
\]

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

**emulator trap**

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**Operation:**
- \( (SP) \leftarrow PS \)
- \( (SP) \leftarrow PC \)
- \( PC \leftarrow (30) \)
- \( PS \leftarrow (32) \)

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

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

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

**Before:**
- PS
- R7, PC
- R6, SP

**After:**
- PS
- PC
- SP

4-61
TRAP

trap 104400—104777

10000110001
15 8 7 0

Operation:
\[(SP)\rightarrow PS\]
\[(SP)\rightarrow PC\]
\[PC \rightarrow (34)\]
\[PS \rightarrow (36)\]

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

Description:
Operation codes from 104400 to 104777 are TRAP instructions. TRAPs and EMTs are identical in operation, except that the trap vector for TRAP is at address 34.

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

breakpoint trap

000003

\[
\begin{array}{ccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1
\end{array}
\]

15 0

Operation:

\[
\begin{align*}
\&\  (SP) & \leftarrow PS \\
\&\  (SP) & \leftarrow PC \\
PC & \leftarrow (14) \\
PS & \leftarrow (16)
\end{align*}
\]

Condition Codes:

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

Description:
Performs a trap sequence with a trap vector address of 14.
Used to call debugging aids. The user is cautioned against employing code 000003 in programs run under these debugging aids.
(no information is transmitted in the low byte.)
IOT

input/output trap

000004

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0

15 0

Operation:

\( \forall (SP) \leftarrow PS \)
\( \forall (SP) \leftarrow PC \)
\( \text{PC} \leftarrow (20) \)
\( \text{PS} \leftarrow (22) \)

Condition Codes:

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

Description:

Performs a trap sequence with a trap vector address of 20. Used to call the I/O Executive routine IOX in the paper tape software system, and for error reporting in the Disk Operating System.
(no information is transmitted in the low byte)
return from interrupt

000002

Operation:

PC ← (SP) + 1
PS ← (SP) + 0

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

(not in 11/05 & 11/10)

return from interrupt

000006

0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0

15 0

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 trap, while RTI permits a trap. If a 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 a certain length of time. In general, these are caused by attempts to reference non-existent memory, and attempts to reference non-existent peripheral devices.

Bus error traps 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 exits 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 inter-
rupts) 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
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):

11/05 & 11/10
Odd Address
Timeout
Trap Instructions
Trace Trap
Power Failure
11/35 & 11/40
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 oper-
ational 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

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>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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Condition Codes: not affected

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

4-71
RESET

reset external bus 000005

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

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.
Condition Code Operators

**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.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
<th>OP Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLC</td>
<td>Clear C</td>
<td>000241</td>
</tr>
<tr>
<td>CLV</td>
<td>Clear V</td>
<td>000242</td>
</tr>
<tr>
<td>CLZ</td>
<td>Clear Z</td>
<td>000244</td>
</tr>
<tr>
<td>CLN</td>
<td>Clear N</td>
<td>000250</td>
</tr>
<tr>
<td>SEC</td>
<td>Set C</td>
<td>000261</td>
</tr>
<tr>
<td>SEV</td>
<td>Set V</td>
<td>000262</td>
</tr>
<tr>
<td>SEZ</td>
<td>Set Z</td>
<td>000264</td>
</tr>
<tr>
<td>SEN</td>
<td>Set N</td>
<td>000270</td>
</tr>
<tr>
<td>SCC</td>
<td>Set all CC's</td>
<td>000277</td>
</tr>
<tr>
<td>CCC</td>
<td>Clear all CC's</td>
<td>000257</td>
</tr>
<tr>
<td>NOP</td>
<td>No Operation</td>
<td>000240</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."

5-1
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.

![Word Stack Diagram](image)

![Byte Stack Diagram](image)

**Figure 5-2: Word and Byte Stacks**

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

This operation is accomplished as follows:

- **MOV Source,-(SP)** ;MOV Source Word onto the stack
  
  or

- **MOV B Source,-(SP)** ;MOV Source Byte onto the stack

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

5-2
To remove an item from stack the autoincrement addressing mode with the appropriate SP is employed. This is accomplished in the following manner:

\[
\text{MOV (SP) + , Destination } \quad ; \text{MOV Destination Word off the stack}
\]

or

\[
\text{MOVB (SP) + , Destination } \quad ; \text{MOVB Destination Byte off the stack}
\]

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

Figure 5-3: Illustration of Push and Pop Operations
As an example of stack usage consider this situation: a subroutine (SUBR) wants to use registers 1 and 2, but these registers must be returned to the calling program with their contents unchanged. The subroutine could be written as follows:

<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></td>
<td></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>000004</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></td>
<td></td>
<td></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 MOV B (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 MOV B since it would accomplish the task of eliminating the unwanted character from the stack by readjusting the stack pointer without the need for a destination location. Also, the stack pointer (SP) used in this example cannot be the system stack pointer (R6) because R6 may only point to word (even) locations.

5.2 SUBROUTINE LINKAGE

5.2.1 Subroutine Calls

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

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

- a general register (R) for linkage
- an entry location (SUBR) for the subroutine

```plaintext
JSR R, SUBR
```
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>JSRR5 SUBR</td>
<td>004567</td>
</tr>
<tr>
<td>001002</td>
<td>index constant for SUBR</td>
<td>000060</td>
</tr>
<tr>
<td>001064</td>
<td>SUBR: MOV A,B</td>
<td>01nnmm</td>
</tr>
</tbody>
</table>

Figure 5-7: JSR using R5

![Figure 5-7: JSR using R5](image)

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

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

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

Calling Program: MOV
       POINTER, R1
       JSR    PC, SUBR

SUBROUTINE
ADD (R1) + (R1) ; Add item #1 to item #2, place
       result in item #2. R1 points
to item #2 now
etc.
or
ADD (R1), 2(R1) ; Same effect as above except that
       R1 still points to item #1
etc.

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

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

![Figure 5.12: Shifting Indexed Base](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)
5.2.3 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 TE0 and TE1.

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

Figure 5.14: Nested Interrupt Service Routines and Subroutines

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

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

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

PDP-11 Approach

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

Conventional Approach

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

Figure 5-15: Reentrant Routines

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

Using reentrant routines, control of a given routine may be shared as illustrated in Figure 5-16.
Figure 5.16: Reentrant Routine Sharing

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

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

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

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

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

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

![Figure 5-17: Multiple R6 Stack](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:

```
MOV #PC2,-(R6)

JSR PC,@(R6) +
```

with the following results:

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

Routine #2 is operating, it then executes:

```
JSR PC, @(R6) +
```

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

Figure 5.18 - Co-Routine Interaction
5.7 PROCESSOR TRAPS
There are a series of errors and programming conditions which will cause the Central Processor to trap to a set of fixed locations. These include Power Failure, Odd Addressing Errors, Stack Errors, Timeout Errors, Memory Parity Errors, Memory Management Violations, 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.

5.7.1 Power Failure
Whenever AC power drops below 95 volts for 115v power (190 volts for 230v) or outside a limit of 47 to 63 Hz, 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.

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

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

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

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

5.7.5 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 saves the PC and PS on the Processor Stack and begins to execute the trap routine pointed to by the trap vector.
CHAPTER 6

MEMORY MANAGEMENT (FOR THE 11/35 & 11/40)

6.1 GENERAL

6.1.1 Options
This chapter describes the Memory Management option, which mounts in the 11/35 or 11/40 Central Processor assembly unit. The option provides the hardware facilities necessary for complete memory management and protection. It is designed to be a memory management facility for systems where the memory size is greater than 28K words and for multi-user, multi-programming systems where protection and relocation facilities are necessary.

The Stack Limit option, which is included with the Memory Management option, is described at the end of the chapter. The Stack Limit option allows dynamic adjustment of the lower limit of permissible stack addresses.

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

KT11-D Memory Management option
KJ11-A Stack Limit option

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

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

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

In a multi-programming environment several user programs would be resident in memory at any given time. The task of the supervisory program would be: control the execution of the various user programs, manage the allocation of memory and peripheral device resources, and safeguard the integrity of the system as a whole by careful control of each user program.
In a multi-programming system, the Management Unit provides the means for assigning pages (relocatable memory segments) to a user program and preventing that user from making any unauthorized access to those pages outside his assigned area. Thus, a user can effectively be prevented from accidental or willful destruction of any other user program or the system executive program.

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

6.1.3 Basic Addressing
The addresses generated by all PDP-11 Family Central Processor Units (CPUs) are 18-bit direct byte addresses. Although the PDP-11 Family word length 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 expanding memory references.

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

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

Memory Size (words): 124K, max (plus 4K for I/O & registers)

Address Space: Virtual (16 bits)
                Physical (18 bits)

Modes of Operation: Kernel & User

Stack Pointers: 2 (one for each mode)

Memory Relocation:
Number of Pages: 16 (8 for each mode)
Page Length: 32 to 4,096 words

Memory Protection: no access
                  read only
                  read/write

6.2 RELOCATION
6.2.1 Virtual Addressing

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

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

6-3
to be "re-linked"; it always appears to be at the same virtual location in memory.

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

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

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

**6.2.2 Program Relocation**
The page address registers are used to determine the starting address of each relocated program in physical memory. Figure 6-2 shows a simplified example of the relocation concept.

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

![Figure 6-2 Simplified Memory Relocation Example](image-url)
If the next processor virtual address is 2, the relocation constant will then cause physical address $6402_{16}$, which is the second item of Program A, to be accessed. When Program B is running, the relocation constant is changed to $100000_{16}$. Then, Program B virtual addresses starting at 0, are relocated to access physical addresses starting at $100000_{16}$. Using the active page address registers to provide relocation eliminates the need to “re-link” a program each time it is loaded into a different physical memory location. The program always appears to start at the same address.

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

The relocation example shown in Figure 6-3 illustrates several points about memory relocation.

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

b) Pages may be relocated to higher or lower physical addresses, with respect to their virtual address ranges. In the example Figure 6-3, page 1 is relocated to a higher range of physical addresses, page 4 is relocated to a lower range, and page 3 is not relocated at all (even though its relocation constant is non-zero).

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

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

**Memory Units**

<table>
<thead>
<tr>
<th>Block:</th>
<th>32 words</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page:</td>
<td>1 to 128 blocks (32 to 4,096 words)</td>
</tr>
<tr>
<td>No. of pages:</td>
<td>8 per mode</td>
</tr>
<tr>
<td>Size of relocatable memory:</td>
<td>32,768 words, max (8 x 4,096)</td>
</tr>
</tbody>
</table>

6-5
6.3 PROTECTION
A timesharing system performs multiprogramming; it allows several programs to reside in memory simultaneously, and to operate sequentially. Access to these programs, and the memory space they occupy, must be strictly defined and controlled. Several types of memory protection must be afforded a timesharing system. For example:

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

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

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

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

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

6.3.2 Read-Only Memory
The access control key for a page can be set to 2, which allows read (fetch) memory references to the page, but immediately aborts any attempt to write into that page. This read-only type of memory protection
can be afforded to pages that contain common data, subroutines, or shared algorithms. This type of memory protection allows the access rights to a given information module to be user-dependent. That is, the access right to a given information module may be varied for different users by altering the access control key.

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

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

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

<table>
<thead>
<tr>
<th>PS(bits15, 14)</th>
<th>PAR/PDR Set Enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Kernel mode</td>
</tr>
<tr>
<td>01</td>
<td>Illegal (all references aborted on access)</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>User mode</td>
</tr>
</tbody>
</table>

Thus, a User mode program is relocated by its own PAR/PDR set, as are Kernel programs. This makes it impossible for a program running in one mode to accidentally reference space allocated to another mode when the active page registers are set correctly. For example, a user cannot transfer to Kernel space. The Kernel mode address space may be reserved for resident system monitor functions, such as the basic Input/Output Control routines, memory management trap handlers, and timesharing scheduling modules. By dividing the types of timesharing system programs functionally between the Kernel and User modes, a minimum amount of space control housekeeping is required as the timeshared operating system sequences from one user program to the next. For example, only the User PAR/PDR set needs to be undated as each new user program is serviced. The two PAR/PDR sets implemented in the Memory Management Unit option are shown in Figure 6.1.

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

A specific processor I/O address is assigned to each PAR and PDR of each set. Table 6-1 is a complete list of address assignment.

**NOTE**

UNIBUS devices cannot access PARs or PDRs

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

<table>
<thead>
<tr>
<th>No.</th>
<th>PAR</th>
<th>PDR</th>
<th>No.</th>
<th>PAR</th>
<th>PDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>772340</td>
<td>772300</td>
<td>0</td>
<td>777640</td>
<td>777600</td>
</tr>
<tr>
<td>1</td>
<td>772342</td>
<td>772302</td>
<td>1</td>
<td>777642</td>
<td>777602</td>
</tr>
<tr>
<td>2</td>
<td>772344</td>
<td>772304</td>
<td>2</td>
<td>777644</td>
<td>777604</td>
</tr>
<tr>
<td>3</td>
<td>772346</td>
<td>772306</td>
<td>3</td>
<td>777646</td>
<td>777606</td>
</tr>
<tr>
<td>4</td>
<td>772350</td>
<td>772310</td>
<td>4</td>
<td>777650</td>
<td>777610</td>
</tr>
<tr>
<td>5</td>
<td>772352</td>
<td>772312</td>
<td>5</td>
<td>777652</td>
<td>777612</td>
</tr>
<tr>
<td>6</td>
<td>772354</td>
<td>772314</td>
<td>6</td>
<td>777654</td>
<td>777614</td>
</tr>
<tr>
<td>7</td>
<td>772356</td>
<td>772316</td>
<td>7</td>
<td>777656</td>
<td>777616</td>
</tr>
</tbody>
</table>

**6.4.1 Page Address Registers (PAR)**

The Page Address Register (PAR), shown in Figure 6-4, contains the 12-bit Page Address Field (PAF) that specifies the base address of the page.

![Figure 6-4 Page Address Register](image)

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

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

**6.4.2 Page Descriptor Registers (PDR)**

The Page Descriptor Register (PDR), shown in Figure 6-5, contains information relative to page expansion, page length, and access control.
Access Control Field (ACF)
This 2-bit field, bits 2 and 1, of the PDR describes the access rights to this particular page. The access codes or "keys" specify the manner in which a page may be accessed and whether or not a given access should result in an abort of the current operation. A memory reference that causes an abort is not completed and is terminated immediately.

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

In the context of access control, the term "write" is used to indicate the action of any instruction which modifies the contents of any addressable word. A "write" is synonymous with what is usually called a "store" or "modify" in many computer systems. Table 6-2 lists the ACF keys and their functions. The ACF is written into the PDR under program control.

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

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

When the expansion direction is downward (ED = 1), the page length is increased by adding blocks with lower relative addresses. Downward expansion is specified for stack pages so that more stack space can be added. An example of page expansion downward is shown in Figure 6-7.
NOTE:
To specify a block length of 42 for an upward expandable page, write highest authorized block no. directly into high byte of PDR. Bit 15 is not used because the highest allowable block number is $177_8$.

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

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

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

PLF for an Upward Expandable Page
When the page expands upward, the PLF must be set to one less than the intended number of blocks authorized for that page. For example, if 52 \((42_{10})\) blocks are authorized, the PLF is set to 51 \((41_{10})\) (Figure 6-6). The KT11-D hardware compares the virtual address block number, VA (bits 12-6) with the PLF to determine if the virtual address is within the authorized page length.

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

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

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

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

6-11
To specify page length for a downward expandable page, write complement of blocks required into high byte of PDR.

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

\[ 42_{10} = 52_8; \text{ two's complement } = 126_8. \]

Figure 6.7  Example of a Downward Expandable Page
The calculations for complementing the number of blocks required to obtain the PLF is as follows:

\[
\begin{array}{cccc}
\text{MAXIMUM BLOCK NO.} & \text{MINUS} & \text{REQUIRED LENGTH} & \text{EQUALS} & \text{PLF} \\
177_8 & - & 52_8 & = & 125_8 \\
127_{10} & - & 42_{10} & = & 85_{10}
\end{array}
\]

6.5 VIRTUAL & PHYSICAL ADDRESSES

The Memory Management Unit is located between the Central Processor Unit and the UNIBUS address lines. Once installed, the Processor ceases to supply address information to the Unibus. Instead, addresses are sent to the Memory Management Unit where they are either transferred without change or relocated by various constants computed within the Memory Management Unit.

6.5.1 Construction of a Physical Address

The basic information needed for the construction of a Physical Address (PA) comes from the Virtual Address (VA), which is illustrated in Figure 6-8, and the appropriate APR set.

![Figure 6-8 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 \((2^{13} = 8K \text{ bytes})\). The DF is further subdivided into two fields as shown in Figure 6-3.

![Figure 6-9 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.

The formation of a physical address takes 150 ns.

The formation of the Physical Address is illustrated in Figure 6-10.

![Figure 6-10 Construction of a Physical Address](image-url)

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

1. Select a set of Active Page Registers depending on current mode.
2. The Active Page Field of the Virtual Address is used to select an Active Page Register (APR0-APR7).
3. The Page Address Field of the selected Active Page Register contains the starting address of the currently active page as a block number in physical memory.
4. The Block Number from the Virtual Address is added to the block number from the Page Address Field to yield the number of the block in physical memory which will contain the Physical Address being constructed.
5. The Displacement in Block from the Displacement Field of the Virtual Address is joined to the Physical Block Number to yield a true 18-bit Physical Address.

6.5.2 Determining the Program Physical Address
A 16-bit virtual address can specify up to 32K words, in the range from 0 to 177776\(_\text{o}\) (word boundaries are even octal numbers). The three most significant virtual address bits designate the PAR/PDR set to be referenced during page address relocation. Table 6-3 lists the virtual address ranges that specify each of the PAR/PDR sets.
Table 6-3  Relating Virtual Address to PAR/PDR Set

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

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

6.6 STATUS 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 (SRO)
SRO contains abort error flags, memory management enable, plus other essential information required by an operating system to recover from an abort or service a memory management trap. The SRO format is shown in Figure 6-11. Its address is 777 572.

Figure 6-11 Format of Status Register #0 (SRO)

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

NOTE
Bit 15, 14, or 13, when set (abort conditions) cause the logic to freeze the contents of SRO bits 1 to 6 and status register SR2. This is done to facilitate recovery from the abort.
Bits 15-13 are enabled when an address is being relocated. This implies that either SR0, bit 0 is equal to 1 (KT11-D operating) or that SR0, bit 8, is equal to 1 and the memory reference is the final one of a destination calculation (maintenance/destination mode).

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

Abort-Nonresident
Bit 15 is the "Abort-Nonresident" bit. It is set by attempting to access a page with an access control field (ACF) key equal to 0 or 4 and setting PS (bits 15, 14) to an illegal mode.

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

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

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

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

Mode of Operation
Bits 5 and 6 indicate the CPU mode (User or Kernel) associated with the page causing the abort. (Kernel = 00, User = 11). These bits are controlled by the logic that decodes current and previous mode bits of the PS.

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

Enable KT11-D
Bit 0 is the "Enable KT11-D" bit. When it is set to 1, all addresses are
relocated and protected by the memory management unit. When bit 0 is set to 0, the memory management unit is disabled and addresses are neither relocated nor protected.

6.6.2 Status Register 2 (SR2)
SR2 is loaded with the 16-bit Virtual Address (VA) at the beginning of each instruction fetch but is not updated if the instruction fetch fails. SR2 is read only; a write attempt will not modify its contents. SR2 is the Virtual Address Program Counter. Upon an abort, the result of SR bits 15, 14, or 13 being set, will freeze SR2 until the SR abort flags are cleared. The address of SR2 is 777 576.

![Figure 6-12 Format of Status Register 2(SR2)](image)

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

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MFPI</td>
<td>move from previous instruction space</td>
<td>0065SS</td>
</tr>
<tr>
<td>MTP</td>
<td>move to previous instruction space</td>
<td>0066DD</td>
</tr>
</tbody>
</table>

These instructions are directly compatible with the larger 11 computer, the PDP-11/45.
MFPI
move from previous instruction space 0065SS

<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>00000110101</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation: (temp) ← (src)
↓ (SP) ← (temp)

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

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

Example:

MFPI @ (R2)

R2 = 1000
1000 = 37526

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

move to previous instruction space

0066DD

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

Example:

MTPI @ (R2)  
R2 = 1000  
1000 ← 37526

The execution of this instruction causes the top word of the current stack to get stored into the (relative) 37526 of the previous address space.
MTPI AND MFPI, MODE 0, REGISTER 6 ARE UNIQUE IN THAT THESE INSTRUCTIONS ENABLE COMMUNICATIONS TO AND FROM THE PREVIOUS USER STACK.

; MFPI, MODE 0, NOT REGISTER 6

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

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

; MFPI, MODE 0, REGISTER 6

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

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

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

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

The desired data from the user stack is now in the kernel stack and has replaced the user stack address.
; MTPI, MODE 0 NOT REGISTER 6

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

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

; MTPI, MODE 0, REGISTER 6

MOV  #UM+PUM, PSW ; user mode, Prev User
CLR  %6 ; set user SP=0 (R16)
MOV  #KM+PUM, PSW ; Kernel mode, prev user
MOV  #−1, −(6) ; MOVE −1 into K stack (R6)
INC  @ #SR0 ; Enable KT
MTPI  %6 ; %16 ←(6)+

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

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

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

The data desired is obtained from the kernel stack then the destination address is obtained from the kernel stack and relocated through the previous mode.
Mode Description
In Kernel mode the operating program has unrestricted use of the machine. The program can map users' programs anywhere in core and thus explicitly protect key areas (including the device registers and the Processor Status word) from the User operating environment.

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

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

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

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

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

User Processor Status (PS) operates as follows:

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

* Explicit operations can be made if the Processor Status is mapped in User space.
6.8 STACK LIMIT OPTION
This option allows program control of the lower limit for permissible stack addresses. This limit may be varied in increments of (400)\textsubscript{8} bytes or (200)\textsubscript{8} words, up to a maximum address of 177 400 (almost the top of a 32K memory).

The normal boundary for stack addresses is 400. The Stack Limit option allows this lower limit to be raised, providing more address space for interrupt vectors or other data that should not be destroyed by the program.

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

```
  15  8  7  0
```

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)\textsubscript{8} or (256)\textsubscript{10}.

Stack Limit Violations
When instructions cause a stack address to exceed (go slower 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 location 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)\textsubscript{8}. The determination of the violation zones is as follows:

- Yellow Zone = (SL) + (340 through 377)\textsubscript{8}  execute, then trap
- Red Zone \leq (SL) + (337)\textsubscript{8}  abort, then trap to location 4

If the Stack Limit Register contents were zero:
- Yellow Zone = 340 through 377
- Red Zone = 000 through 337

6-23
CHAPTER 7

ARITHMETIC OPTIONS (FOR THE 11/35 & 11/40)

7.1 GENERAL
This chapter describes 2 options which mount in the 11/35 or 11/40 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 options are contained on individual modules that plug into dedicated, prewired slots.

KE11-E EIS option
KE11-F Floating Point 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 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.
MUL

multiply

070RSS

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>r</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>6</td>
<td>s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>8</td>
<td>r</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

Operation: 
R, Rv1 ← R x(src)

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

Description:
The contents of the destination register and source taken as two’s complement integers are multiplied and stored in the destination register and the succeeding register (if R is even). If R is odd only the low order product is stored. Assembler syntax is: MUL S,R.
(Note that the actual destination is R,Rv1 which reduces to just R when R is odd.)

Example:
16-bit product (R is odd)

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

Before
(R1) = 000400

After
(R1) = 004000

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

7.2
DIV

Divide

Operation: \( R, Rv1 \leftarrow R, Rv1 \div (src) \)

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>(R1) = 000001</td>
<td>(R1) = 000001</td>
</tr>
</tbody>
</table>
ASH

shift arithmetically

| 0 | 1 | 1 | 1 | 0 | 1 | 0 | r | r | r | s | s | s | s | s | s | s | s | s | s |
| 15| 9 | 8 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0

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 a right shift and positive is a left shift.

6 LSB of source
- 011111
- 000001
- 111111
- 100000

Action in general register
- Shift left 31 places
- Shift left 1 place
- Shift right 1 place
- Shift right 32 places

Example: ASH R0, R3

Before
\((R3) = 001234\)
\((R0) = 000003\)

After
\((R3) = 012340\)
\((R0) = 000003\)
ASHC

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

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

Description: The contents of the register and the register OR'ed 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.

7-5
7.3 FLOATING POINT OPTION

The Floating Point instructions used with this option are unique to the PDP-11/35 & 40. However, the OP Codes used do not conflict with any other instructions.

<table>
<thead>
<tr>
<th>Mnenomic</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 subract</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:

```
 15  7  6  0
S  EXPONENT  FRACTION (HIGH PART)
```

```
 15  0
FRACTION (LOW PART)
```

```
 0
LOW ARGUMENT
```

S = sign of fraction; 0 for positive, 1 for negative

Exponent = 8 bits for the exponent, in excess (200)_8 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 ½ 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)_8 or (128)_10 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>

7-6
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**

\[ +12_{10} = +(1100)_2 \]

\[ = +(2^4)_10 \times (.11)_2 \]

\[ [16 \times (1/2 + 1/4) = 12] \]

<table>
<thead>
<tr>
<th>S</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>representation: 0</td>
<td>10 000 100</td>
<td>1000000 00000000000000000</td>
</tr>
</tbody>
</table>

**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:

\( (R) \) = High B argument address
\( (R)+2 \) = Low B argument address
\( (R)+4 \) = High A argument address
\( (R)+6 \) = Low A argument address

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

\( (R)+4 \) = address for High part of answer
\( (R)+6 \) = 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:

\( V = 1 \), if an error occurs
\( N = 1 \), if underflow or divide-by-zero
\( C = 1 \), 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>
</tbody>
</table>
| Divide by 0 | 1     | 1     | 1     | 0

7-7
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 near completion. The Program Counter will point to the aborted Floating instruction so that the interrupt will look transparent.

Assembler format is: OPR R

INSTRUCTIONS

**FADD**

floating add 07500R

```
01111010 0000000 r r r
```

**Operation:** 
\[ [(R)+4, (R)+6] ← [(R)+4, (R)+6] + [(R),(R)+2] \], if result \( ≥ 2^{-128} \); 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:** 
Add 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.

\[ A ← A + B \]

**FSUB**

floating subtract 07501R

```
01111010 1000000 r r r
```

**Operation:** 
\[ [(R)+4, (R)+6] ← [(R)+4, (R)+6] − [(R),(R)+2] \], if result \( ≥ 2^{-128} \); 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:** 
Subtracts the B Argument from the A Argument and stores the result in the A Argument position on the stack.

\[ A ← A − B \]
**FMUL**

Floating Multiply

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

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

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

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.

\(A \leftarrow A / B\)
CHAPTER 8

CONSOLE OPERATION

8.1  PDP-11/05 & 11/10 CONSOLE

8.1.1  Console Elements
The PDP-11/05 and 11/10 Operator's Console provides the following facilities:

- Power Switch (with a key lock)
- ADDRESS/DATA display (16 bits)
- Switch Register (16 switches)
- RUN status light

Control Switches
- LOAD ADRS (Load Address)
- EXAM (Examine)
- CONT (Continue)
- ENABLE/HALT
- START
- DEP (Deposit)

8.1.2  Console Switches

<table>
<thead>
<tr>
<th>Switch</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POWER</td>
<td>Power to the processor is off.</td>
</tr>
<tr>
<td></td>
<td>Power to the processor is on and all console switches function normally.</td>
</tr>
<tr>
<td></td>
<td>Power to the processor is on, but the Control Switches are disabled. The Switch Register is still functional.</td>
</tr>
<tr>
<td></td>
<td>Used to manually load data or an address into the processor.</td>
</tr>
</tbody>
</table>

Switch Register
- (Up = 1)
- (Down = 0)

Control Switches
- LOAD ADRS (depress to activate)
  - Transfers contents of the Switch Register to the processor.
  - The entered data is displayed in the ADDRESS/DATA lights, and provides an address for EXAM, DEP, and START.
EXAM
Causes the contents of the selected location to be displayed in the ADDRESS/DATA lights. While the EXAM switch is depressed, the address to be examined is displayed. The data itself is displayed when the switch is released.
If the EXAM switch is depressed again, the contents of the next sequential word location are displayed. (Bus Address is incremented automatically). If an odd address is specified, the next lower even address word will be displayed (except for the general registers, R0 to R7). If a non-existent memory address is specified, no UNIBUS operation will be performed, and the processor will have to be initialized by setting the ENABLE/HALT switch to HALT and then depressing the START switch.
If the CPU is in the RUN state, the EXAM switch has no effect.

CONT
(depress and release 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 after the current instruction. All interrupts and traps will be executed prior to halting. Depressing and then releasing the CONT switch will now cause execution of a single instruction.
If the program had stopped, depressing the START switch causes a System Reset signal to occur and loads the Program Counter with the address contained in the switches when LOAD ADRS was last depressed. The program will then continue only if the ENABLE/HALT switch is in ENABLE.

WARNING:
If the CPU is in the RUN state and the POWER switch is not in PANEL LOCK, the START switch will interrupt the program. The program may even have to be reloaded.

8-2
Deposits contents of the Switch Register into the selected location. While the DEP switch is raised, the address to be loaded is displayed. When the switch is released, the data deposited is displayed.

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 (except for the general registers, R0 to R7). If a non-existent memory address is specified, no UNIBUS operation will be completed and the processor will have to be initialized by setting the ENABLE/HALT switch to HALT and then depressing the START switch.

8.1.3 Indicators

RUN

Lights when the processor is executing instructions. It is off when the processor is halted. It is on during a WAIT instruction and UNIBUS cycles.

ADDRESS/DATA

Displays either addresses or data, as specified in Table 8.1.
<table>
<thead>
<tr>
<th>Condition</th>
<th>ADDRESS/DATA Display</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>POWER On</strong></td>
<td><strong>ENABLE/HALT in</strong></td>
</tr>
<tr>
<td></td>
<td><strong>HALT</strong></td>
</tr>
<tr>
<td></td>
<td><strong>ENABLE/HALT in</strong></td>
</tr>
<tr>
<td></td>
<td><strong>ENABLE</strong></td>
</tr>
<tr>
<td>Load Address</td>
<td><strong>LOAD ADRS switch</strong></td>
</tr>
<tr>
<td></td>
<td>is depressed</td>
</tr>
<tr>
<td>Examine</td>
<td><strong>EXAM switch is</strong></td>
</tr>
<tr>
<td></td>
<td><strong>depressed</strong></td>
</tr>
<tr>
<td></td>
<td><strong>EXAM switch is</strong></td>
</tr>
<tr>
<td></td>
<td><strong>released</strong></td>
</tr>
<tr>
<td>Deposit</td>
<td><strong>DEP switch is</strong></td>
</tr>
<tr>
<td></td>
<td><strong>raised</strong></td>
</tr>
<tr>
<td></td>
<td><strong>DEP switch is</strong></td>
</tr>
<tr>
<td></td>
<td><strong>released</strong></td>
</tr>
<tr>
<td><strong>RUN Light On</strong></td>
<td><strong>ENABLE/HALT in</strong></td>
</tr>
<tr>
<td>Program Halt</td>
<td><strong>HALT</strong></td>
</tr>
<tr>
<td></td>
<td><strong>HALT instruction</strong></td>
</tr>
<tr>
<td></td>
<td><strong>executed</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Double Bus Error</strong></td>
</tr>
<tr>
<td></td>
<td>(two successive attempts to access non-existent memory or improper odd byte address)</td>
</tr>
<tr>
<td>Program Execution</td>
<td><strong>START switch is</strong></td>
</tr>
<tr>
<td></td>
<td><strong>depressed</strong></td>
</tr>
<tr>
<td></td>
<td><strong>CONT switch is</strong></td>
</tr>
<tr>
<td></td>
<td><strong>depressed</strong></td>
</tr>
</tbody>
</table>
8.2 PDP-11/35 & 11/40 CONSOLE

8.2.1 Console Elements

The PDP-11/35 & 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)

8.2.2 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.
8.2.3 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

Control Switches

LOAD ADRS (depress to activate)
Transfers contents of the Switch Register to the Bus Address register.

EXAM (depress to activate)
Causes the contents of the location specified by the Bus Address to be displayed in the DATA Register. If the EXAM switch is depressed again, the contents of the next sequential word location are displayed. (Bus Address is incremented automatically). If an odd address is specified, the next lower even address word will be displayed. If a non-existent memory address is specified, no UNIBUS operation will be completed, and contents of the Switch Register address (777 570) will be displayed in the DATA register.

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 located 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. If a non-existent memory address is specified, no UNIBUS operation will be completed, and contents of the Switch Register address (777 570) will be displayed in the DATA register.

8.2.4 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 R0 contents are displayed. During Single Instruction operation, the Processor Status word (PS) is displayed.
CHAPTER 9

SPECIFICATIONS

9.1 CPU OPERATING SPECIFICATIONS

Temperature: \(+10^\circ\text{C} \text{ to } +50^\circ\text{C}\)

Relative Humidity: 20% to 95% (without condensation)

Input Power: 
115 VAC ± 10\%, 47 to 63 Hz 
or 230 VAC ± 10\%, 47 to 63 Hz

9.2 PACKAGING

All the PDP-11 CPU's are housed in slide chassis units that can be mounted in standard 19" racks. The included power supply has sufficient excess capacity to drive core memory modules and peripheral logic mounted within the unit. Module slots are prewired to accept some of the optional equipment.

11/05 & 11/10

The PDP-11/05 and 11/10 are housed in either a 5-1/4" or 10-1/2" high unit. The 10-1/2" unit can slide out and then tilt (5 positions), for convenient access to the internal equipment. The 11/05 and 11/10 have the same CPU, but are available in several arrangements, see Figures 9-1 and 9-2.

11/35

The PDP-11/35 is housed in a 10-1/2" high unit that can slide and tilt. The 11/35 is available in 3 versions; the first 2 have core memory mounted within the CPU assembly, and the third has the memory in an external unit that can hold between 8K and 24K, see Figure 9-3.

11/40

The PDP-11/40 is housed in a 21" high unit. The 11/40 and 11/35 have the same CPU. The power supply does not slide out, but stays mounted stably in the cabinet, which is included. See Figure 9-4.

Standard Cabinet

A standard 19" cabinet has two rows of mounting holes in the front, spaced 18-5/16" apart. The holes are located at 1/2" or 5/8" apart from each other, see Figure 9-5. Standard front panel increments are 1-3/4".

\[(5/8 + 5/8 + 1/2 = 1-3/4)\]

The standard PDP-11 cabinet is 72" high by 21" wide by 30" deep. It is recommended that a service area of at least 35" be allowed in the front and back of the cabinet. Each cabinet contains a power control so that all equipment within the cabinet (and other connected cabinets) can have their AC power turned on and off together.
Figure 9-1 5-1/4" PDP-11/05 & 11/10 Assembly Unit
Figure 9.2 10-1/2" PDP-11/05 & 11/10 Assembly Unit
Figure 9-3 PDP-11/35 Assembly Unit
Figure 9-4 PDP-11/40 Assembly Unit
Figure 9-5 Standard 19" Rack
9.3 MOUNTING INFORMATION
There are three basic mounting assemblies for the computers described in this handbook.

<table>
<thead>
<tr>
<th>Front Panel Height</th>
<th>Computers</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-1/4&quot;</td>
<td>PDP-11/05, 11/10</td>
<td>Figure 9-6</td>
</tr>
<tr>
<td>10-1/2&quot;</td>
<td>PDP-11/05, 11/10, 11/35</td>
<td>Figure 9-7</td>
</tr>
<tr>
<td>21&quot;</td>
<td>PDP-11/40</td>
<td>Figure 9-8</td>
</tr>
</tbody>
</table>

Figure 9-6 5-1/4" High Unit
Figure 9-7 10-1/2" High Unit

Figure 9-8 21" High Unit
<table>
<thead>
<tr>
<th></th>
<th>11/05</th>
<th>11/05</th>
<th>11/10</th>
<th>11/10</th>
<th>11/35</th>
<th>11/40</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front Panel Height</td>
<td>5-1/4&quot;</td>
<td>10-1/2&quot;</td>
<td>5-1/4&quot;</td>
<td>10-1/2&quot;</td>
<td>10-1/2&quot;</td>
<td>21&quot;</td>
</tr>
<tr>
<td>Max Internal Space</td>
<td>9 slots</td>
<td>5 SU</td>
<td>9 slots</td>
<td>5 SU</td>
<td>5 SU</td>
<td>9 SU</td>
</tr>
<tr>
<td>Min Memory Size (words)</td>
<td>4K</td>
<td>4K</td>
<td>8K</td>
<td>8K</td>
<td>8K</td>
<td>8K</td>
</tr>
<tr>
<td>Max Pre-wired Memory</td>
<td>16K</td>
<td>16K</td>
<td>8K</td>
<td>16K</td>
<td>24K</td>
<td>24K</td>
</tr>
<tr>
<td>Max Memory in CPU Assembly</td>
<td>16K</td>
<td>28K</td>
<td>8K</td>
<td>28K</td>
<td>32K</td>
<td>80K</td>
</tr>
<tr>
<td>Space for Options (or extra memory)</td>
<td>1 or 4 slots</td>
<td>3 SU</td>
<td>4 slots</td>
<td>3 SU</td>
<td>1 to 3 SU</td>
<td>5 SU</td>
</tr>
<tr>
<td>Weight (CPU)</td>
<td>50 lbs</td>
<td>110</td>
<td>50</td>
<td>110</td>
<td>120</td>
<td>200</td>
</tr>
<tr>
<td>Power (CPU &amp; 8K of memory)</td>
<td>5 amps</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Current at 115 VAC</td>
<td>500 W</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>700</td>
<td>700</td>
</tr>
<tr>
<td>Heat dissipation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Excess Current available after CPU &amp; 8K memory</td>
<td>6 amps</td>
<td>15</td>
<td>6</td>
<td>15</td>
<td>11</td>
<td>21</td>
</tr>
<tr>
<td>+5 V</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>-15 V</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Current required for each extra 8K memory</td>
<td>1.7 amps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+5 V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-15 V</td>
<td>0.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Central Processor</td>
<td>Main Market</td>
<td>Memory</td>
<td>Reg to Mem Transfer</td>
<td>Max Mem Size (words)</td>
<td>Max Address Space (words)</td>
<td>General Purpose Reg</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
<td>--------</td>
<td>---------------------</td>
<td>---------------------</td>
<td>--------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>OEM</td>
<td>End User</td>
<td>OEM</td>
<td>End User</td>
<td>OEM</td>
<td>OEM</td>
<td>OEM</td>
</tr>
<tr>
<td>Core</td>
<td>Core</td>
<td>Core</td>
<td>Core</td>
<td>Core</td>
<td>Core</td>
<td>Core</td>
</tr>
<tr>
<td>2.7 ms</td>
<td>2.3 ms</td>
<td>2.8K</td>
<td>2.8K</td>
<td>2.8K</td>
<td>2.8K</td>
<td>2.8K</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>400 (fixed)</td>
<td>400 (fixed)</td>
<td>400</td>
<td>400</td>
<td>400</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>Software only</td>
<td>Software only</td>
<td>Software only</td>
<td>Software only</td>
<td>Software only</td>
<td>Software only</td>
<td>Software only</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4 line</td>
<td>4 line</td>
<td>4 line</td>
<td>4 line</td>
<td>4 line</td>
<td>4 line</td>
<td>4 line</td>
</tr>
<tr>
<td>Multi-level</td>
<td>Multi-level</td>
<td>Multi-level</td>
<td>Multi-level</td>
<td>Multi-level</td>
<td>Multi-level</td>
<td>Multi-level</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

9-10
APPENDIX A

PROGRAMMING DIFFERENCES BETWEEN
PDP-11 CPU's

A.1 INTRODUCTION

There are a few minor differences in how the different central processors operate on certain instructions and programming situations. In almost all cases the differences represent an improvement over the first PDP-11 computer, the PDP-11/20, and the operations are more efficient but transparent to the user. In other cases, obscure combinations of addressing modes and registers cause slightly different results.

A.2 DIFFERENCES

The following Table shows the major differences between the CPU's described in this Handbook, and the original PDP-11/20. (The PDP-11/15 operates the same as the 11/20).
# TABLE OF PROGRAMMING DIFFERENCES

| A. | OPR %R, (R)+ or OPR %R, -(R) or OPR %R, @(R)+ or OPR %R, @-(R) (Using the same reg. as both source & destination). | Contents of R are incremented by 2 (or decremented by 2) before being used as the source operand. | Initial contents of R are used as the source operand. (same as 11/20) |
| B. | JMP (R)+ or JSR reg, (R)+ (Jump using auto-increment mode). | Contents of R are incremented by 2, then used as the new PC address. | (same as 11/20) |
| C. | MOV PC, @#A or MOV PC, A (Moving the incremented PC of the Move instruction +4). | Location A will contain the PC of the Move instruction +4. | Location A will contain PC+2. (same as 11/20) |
| D. | Stack Pointer (SP), R6 used for referencing. | Using the SP for pointing to odd addresses or non-existent memory causes a HALT (double bus error). (same as 11/20) | Odd address of non-existent memory references with SP cause a fatal trap, with a new stack created at locations 0 & 2. |
### TABLE OF PROGRAMMING DIFFERENCES (Cont.)

<table>
<thead>
<tr>
<th></th>
<th>11/20</th>
<th>11/05 &amp; 11/10</th>
<th>11/35 &amp; 11/40</th>
</tr>
</thead>
<tbody>
<tr>
<td>E. Stack Overflow</td>
<td>Stack limit fixed at 400 (octal). Overflow (going lower) checked after @—(R6), JSR, traps, and address modes 4 &amp; 5. Overflow serviced by an overflow trap. No red zone.</td>
<td>(same as 11/20)</td>
<td>Variable limit with Stack Limit option. Overflow checked after JSR, traps, and address modes 1, 2, 4, &amp; 6. Non-altering references to stack data is always allowed. There is a 16-word yellow (warning) zone. Red zone trap occurs if stack is 16 words below boundary; PS &amp; PC are saved at locations 0 &amp; 2.</td>
</tr>
</tbody>
</table>

### II. TRAPS & INTERRUPTS

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>(same as 11/20)</th>
<th>If RTI sets the T bit, the T bit trap is acknowledged immediately after the RTI instruction. First instruction after RTT is guaranteed to be executed. Acts like RTI on the 11/20 (same as 11/05)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. RTI instruction</td>
<td>First instruction after RTI is guaranteed to be executed.</td>
<td>(not implemented)</td>
<td>Only RTI, RTT, traps, and interrupts can load the T bit.</td>
</tr>
<tr>
<td>B. RTT instruction</td>
<td>(not implemented)</td>
<td>(not implemented)</td>
<td></td>
</tr>
<tr>
<td>C. Processor Status (PS)</td>
<td>Addressing odd byte of PS (bits 15-8) causes an odd address trap.</td>
<td>Odd byte of PS can be addressed without a trap.</td>
<td></td>
</tr>
<tr>
<td>odd byte at location</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>777 777</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D. T bits of PS</td>
<td>T bit can be loaded by direct address of PS, or from the console.</td>
<td>(same as 11/20)</td>
<td></td>
</tr>
</tbody>
</table>
### TABLE OF PROGRAMMING DIFFERENCES (Cont.)

<table>
<thead>
<tr>
<th></th>
<th>11/20</th>
<th>11/05 &amp; 11/10</th>
<th>11/35 &amp; 11/40</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>E. Interrupt service routine</strong></td>
<td>The first instruction in the routine is guaranteed to be executed.</td>
<td>The first instruction will not be executed if another interrupt occurs at a higher priority.</td>
<td>(same as 11/05)</td>
</tr>
<tr>
<td><strong>F. Priority order of traps &amp; interrupts</strong></td>
<td>Odd address &lt;br&gt;Timeout &lt;br&gt;HALT from console &lt;br&gt;Trap instructions &lt;br&gt;Trace trap &lt;br&gt;Stack overflow &lt;br&gt;Power fail</td>
<td>Odd address &lt;br&gt;Timeout &lt;br&gt;HALT instruction &lt;br&gt;Trap instructions &lt;br&gt;Trace Trap &lt;br&gt;Stack overflow &lt;br&gt;Power fail &lt;br&gt;HALT from console</td>
<td>Odd address &lt;br&gt;Stack overflow (red) &lt;br&gt;Timeout &lt;br&gt;Mem. Mgt. violation &lt;br&gt;HALT &lt;br&gt;Trap instructions &lt;br&gt;Trace trap &lt;br&gt;Stack overflow (yellow) &lt;br&gt;Power fail</td>
</tr>
</tbody>
</table>

### III. MISCELLANEOUS

| **A. SWAB and V bit** | SWAB instruction conditionally sets the V bit. | V bit is cleared. | (same as 11/05) |
| **B. Instruction set** | Basic set. | (same as 11/20) | Basic set + MARK, RTT, SOB, SXT, XOR. <br>EIS adds: MUL, DIV, ASH, ASHC. <br>Floating Point adds: FADD, FSUB, FMUL, FDIV. |
A.3 COMPATIBILITY

In order to stay compatible with all PDP-11 computers, avoid the following.

1. Using:
   
   OPR %R, (R)+
   OPR %R, -(R)
   OPR %R, @(R)+
   OPR%R, @-(R)
   
   JMP (R)+
   JMP %R

   JSR reg, (R)+
   JSR %R, %R

2. Testing the V bit after SWAB.

3. Using the T bit of the Processor Status word.
## Appendix B  Memory Map

### Interrupt Vectors.

<table>
<thead>
<tr>
<th>Memory Address</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>BPT 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>TTL 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>AFO11 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 Pen 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, LS11 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 IRQ</td>
</tr>
<tr>
<td>244</td>
<td>Floating Point Error</td>
</tr>
<tr>
<td>250</td>
<td>Segmentation Trap</td>
</tr>
<tr>
<td>254</td>
<td>RP11 Disk Pack CTRL-BR5</td>
</tr>
<tr>
<td>260</td>
<td>TA11 Cassette-BR6</td>
</tr>
<tr>
<td>264</td>
<td>User Reserved</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>

B-1
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 PIRO 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  DL11 TTY OUT DBR
777564  DL11 TTY OUT CSR
777562  DL11 TTY IN DBR
777560  DL11 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 RFDSC

777456  RC11 DISK RCDBR
777454  RC11 MAINTENANCE
777452  RC11 RCCAR
777450  RC11 RCWC
777446  RC11 RCCSR
777444  RC11 RCCSRI
777442  RC11 RCER
777440  RC11 RCLA
777434  DT11 BUS SWITCH # 7
777432  BUS SWITCH # 6
777430  BUS SWITCH # 5
777426  BUS SWITCH # 4
777424  BUS SWITCH # 3
777422  BUS SWITCH # 2
777420  BUS SWITCH # 1

777416  RKDB   RK11 DISK
777414  RKMRR
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 KGNU7 |
| 770774 | KGDBR7     |
| 770772 | KGBBC7     |
| 770770 | KGCSR7     |
| 770716 | KGNU1      |
| 770714 | KGBCC1     |
| 770712 | KGDBR1     |
| 770710 | KGCSR1     |
| 770706 | KGNU0      |
| 770704 | KGDBR0     |
| 770702 | KGBCC0     |
770700    KG11A KGCSR0
770676 TO 770500 16 LINE FOR DM11BB
770676    DM11BB #16
770674
770672
770670
770666    DM11BB #15
770664
770662
770660
770656    DM11BB #14
770654
770652
770650
770646    DM11BB #13
770644
770642
770640
770636    DM11BB #12
770634
770632
770630
770626    DM11BB #11
770624
770622
770620
770616    DM11BB #10
770614
770612
770610
770606    DM11BB #9
770604
770602
770600    DM11BB #8
770076    LATENCY TESTER
770074    LATENCY TESTER
770072    LATENCY TESTER
770070    LATENCY TESTER
770056 TO 770000 SPECIAL FACTORY BUS TESTERS
767776 TO 764000 FOR USFR 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
764000    START NORMAL USER ADDRESSES HERE AND ASSIGN UPWARD.
760004 TO 760000 RESERVED FOR DIAGNOSTIC - SHOULD NOT BE ASSIGNED

B-6
APPENDIX C

INSTRUCTION TIMING

C.1 PDP-11/05 & 11/10

INSTRUCTION EXECUTION TIME

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

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

Double Operand instructions require all 3 of these Times, Single Operand instructions require a Basic Time and a DST Time, and with all other instructions the Basic Time is the Instr Time.

All Timing information is in microseconds, unless otherwise noted. Times are typical; processor timing can vary \(\pm 10\%\).

SOURCE & DESTINATION ADDRESS TIMES

The SRC and DST Times apply directly to Word and Even Byte instructions. Odd Byte instructions take longer, see Notes following.

<table>
<thead>
<tr>
<th>Mode</th>
<th>SRC Time*</th>
<th>DST Time**</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0 (\mu)sec</td>
<td>0.0 (\mu)sec</td>
</tr>
<tr>
<td>1</td>
<td>0.9</td>
<td>2.4</td>
</tr>
<tr>
<td>2</td>
<td>0.9</td>
<td>2.4</td>
</tr>
<tr>
<td>3</td>
<td>2.4</td>
<td>3.4</td>
</tr>
<tr>
<td>4</td>
<td>0.9</td>
<td>2.4</td>
</tr>
<tr>
<td>5</td>
<td>2.4</td>
<td>3.4</td>
</tr>
<tr>
<td>6</td>
<td>2.4</td>
<td>3.4</td>
</tr>
<tr>
<td>7</td>
<td>3.4</td>
<td>4.7</td>
</tr>
</tbody>
</table>

NOTES:

—For SCR Time, add 1.3 \(\mu\)sec for Odd Byte addressing.

—For DST Time, and Odd Byte addressing:
  1. add 1.3 \(\mu\)sec for a non-modifying instruction (CMPB, BITB, TSTB).
  2. add 2.4 \(\mu\)sec for a modifying instruction.
## BASIC TIME

### Double Operand

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Basic Time</th>
<th>Instr Time = Basic Time + SRC Time + DST Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD, SUB, BIC, BIS</td>
<td>3.7 μsec</td>
<td>2.5 μsec</td>
</tr>
<tr>
<td>CMP, BIT</td>
<td>2.5</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>3.7</td>
<td>(3.1 μsec if Word instruction and mode 0)</td>
</tr>
</tbody>
</table>

### Single Operand

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Basic Time</th>
<th>Instr Time = Basic Time + DST Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR, COM, INC, DEC, NEG, ASR, ASL, ROR, ROL, ADC, SBC</td>
<td>3.4 μsec</td>
<td></td>
</tr>
<tr>
<td>TST</td>
<td>2.2</td>
<td></td>
</tr>
<tr>
<td>SWAB</td>
<td>4.3</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>
</tr>
</thead>
<tbody>
<tr>
<td>(all branches)</td>
<td>2.5 μsec</td>
<td>1.9 μsec</td>
</tr>
</tbody>
</table>

### Jump Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Basic Time</th>
<th>Instr Time = Basic Time + DST Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP</td>
<td>1.0 μsec</td>
<td></td>
</tr>
<tr>
<td>JSR</td>
<td>3.8</td>
<td></td>
</tr>
</tbody>
</table>

### Control, Trap & Misc Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instr Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTS</td>
<td>3.8</td>
</tr>
<tr>
<td>RTI</td>
<td>4.4</td>
</tr>
<tr>
<td>SET N,Z,V,C</td>
<td>2.5</td>
</tr>
<tr>
<td>CLR N,Z,V,C</td>
<td>2.5</td>
</tr>
<tr>
<td>HALT</td>
<td>1.8</td>
</tr>
<tr>
<td>WAIT</td>
<td>1.8</td>
</tr>
<tr>
<td>RESET</td>
<td>100 msec</td>
</tr>
<tr>
<td>IOT, EMT, TRAP, BPT</td>
<td>8.2 μsec</td>
</tr>
</tbody>
</table>

### LATENCY

NPR latency is 7 μsec, max.
C.2 PDP-11/35 & 11/40

INSTRUCTION EXECUTION TIME

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

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

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

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 MM11-L 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 \( \mu \text{sec} \) must be added for each memory cycle.

3. If the Memory Management (KT11-D) option is installed, instruction execution times increase by 0.15 \( \mu \text{sec} \) 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>0.00 μsec</td>
<td>0</td>
<td></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,</td>
<td>1</td>
<td>.78 ( .90)</td>
<td>1</td>
</tr>
<tr>
<td>and</td>
<td>2</td>
<td>.84 ( .90)</td>
<td>1</td>
</tr>
<tr>
<td>Double</td>
<td>3</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</td>
<td>5</td>
<td>1.74 (1.80)</td>
<td>2</td>
</tr>
<tr>
<td>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 (use with SRC Time &amp; DST Time)</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></td>
<td>EF Time</td>
<td>Mem Cyc</td>
<td>EF Time</td>
</tr>
<tr>
<td>ADD, CMP, BIT, BIC, BIS</td>
<td>0.99 μs</td>
<td>1</td>
<td>1.60 μs</td>
</tr>
<tr>
<td>SUB</td>
<td>.99</td>
<td>1</td>
<td>1.60</td>
</tr>
<tr>
<td>XOR</td>
<td>.99</td>
<td>1</td>
<td>—</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</td>
<td>0.90 µsec</td>
<td>1.80 µsec</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1 to 7</td>
<td>1.46</td>
<td></td>
<td>1.80</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0 to 7</td>
<td>2.42</td>
<td></td>
<td>2.56</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0 to 7</td>
<td>2.42</td>
<td></td>
<td>2.56</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0 to 7</td>
<td>3.18</td>
<td></td>
<td>3.32</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0 to 7</td>
<td>2.42</td>
<td></td>
<td>2.56</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0 to 7</td>
<td>3.18</td>
<td></td>
<td>3.32</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>2.84</td>
<td></td>
<td>2.98</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>1 to 7</td>
<td>3.18</td>
<td></td>
<td>3.32</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>3.68</td>
<td></td>
<td>3.82</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>1 to 7</td>
<td>4.02</td>
<td></td>
<td>4.16</td>
<td>4</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, WCS, BGE, BLT, BGT, BLE, BHI, BLOS, 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></td>
<td>1.80 μsec</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2.10</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>2.30</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>1.90</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>2.30</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>2.36</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>2.92</td>
<td>3</td>
</tr>
<tr>
<td>JMP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>2.94</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>3.24</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>3.44</td>
<td>3</td>
</tr>
<tr>
<td>JSR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>3.04</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>3.44</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>3.50</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>4.06</td>
<td>4</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 Mode</th>
<th>SRC Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.28 μsec</td>
</tr>
<tr>
<td>1</td>
<td>.78</td>
</tr>
<tr>
<td>2</td>
<td>.98</td>
</tr>
<tr>
<td>3</td>
<td>1.74</td>
</tr>
<tr>
<td>4</td>
<td>.98</td>
</tr>
<tr>
<td>5</td>
<td>1.74</td>
</tr>
<tr>
<td>6</td>
<td>1.74</td>
</tr>
<tr>
<td>7</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>—</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 $\mu$sec. Interrupt service time is 5.42 $\mu$sec, max. NPR latency is 3.50 $\mu$sec, max.
<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>7-4</td>
</tr>
<tr>
<td>ASHC</td>
<td>7-5</td>
</tr>
<tr>
<td>ASR(B)</td>
<td>4-13</td>
</tr>
<tr>
<td>BCC</td>
<td>4-40</td>
</tr>
<tr>
<td>BCS</td>
<td>4-41</td>
</tr>
<tr>
<td>BEQ</td>
<td>4-35</td>
</tr>
<tr>
<td>BGE</td>
<td>4-43</td>
</tr>
<tr>
<td>BGT</td>
<td>4-45</td>
</tr>
<tr>
<td>BHI</td>
<td>4-48</td>
</tr>
<tr>
<td>BHIS</td>
<td>4-50</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>4-44</td>
</tr>
<tr>
<td>BLE</td>
<td>4-46</td>
</tr>
<tr>
<td>BLO</td>
<td>4-51</td>
</tr>
<tr>
<td>BLOS</td>
<td>4-49</td>
</tr>
<tr>
<td>BMI</td>
<td>4-37</td>
</tr>
<tr>
<td>BNE</td>
<td>4-34</td>
</tr>
<tr>
<td>BPL</td>
<td>4-36</td>
</tr>
<tr>
<td>BPT</td>
<td>4-63</td>
</tr>
<tr>
<td>BR</td>
<td>4-33</td>
</tr>
<tr>
<td>BVC</td>
<td>4-38</td>
</tr>
<tr>
<td>BVS</td>
<td>4-39</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>4-73</td>
</tr>
<tr>
<td>DEC(B)</td>
<td>4-9</td>
</tr>
<tr>
<td>DIV</td>
<td>7-3</td>
</tr>
<tr>
<td>EMT</td>
<td>4-61</td>
</tr>
<tr>
<td>FADD</td>
<td>7-8</td>
</tr>
<tr>
<td>FDIV</td>
<td>7-9</td>
</tr>
<tr>
<td>FMUL</td>
<td>7-9</td>
</tr>
<tr>
<td>FSUB</td>
<td>7-8</td>
</tr>
<tr>
<td>HALT</td>
<td>4-70</td>
</tr>
<tr>
<td>INC(B)</td>
<td>4-8</td>
</tr>
<tr>
<td>IOT</td>
<td>4-64</td>
</tr>
<tr>
<td>JMP</td>
<td>4-52</td>
</tr>
<tr>
<td>JSR</td>
<td>4-54</td>
</tr>
<tr>
<td>MARK</td>
<td>4-57</td>
</tr>
<tr>
<td>MFPI</td>
<td>6-18</td>
</tr>
<tr>
<td>MOV(B)</td>
<td>4-23</td>
</tr>
<tr>
<td>MTPi</td>
<td>6-19</td>
</tr>
<tr>
<td>MUL</td>
<td>7-2</td>
</tr>
<tr>
<td>NEG(B)</td>
<td>4-10</td>
</tr>
<tr>
<td>NOP</td>
<td>4-73</td>
</tr>
<tr>
<td>RESET</td>
<td>4-72</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>4-65</td>
</tr>
<tr>
<td>RTS</td>
<td>4-56</td>
</tr>
<tr>
<td>RTT</td>
<td>4-66</td>
</tr>
<tr>
<td>SBC(B)</td>
<td>4-20</td>
</tr>
<tr>
<td>SOB</td>
<td>4-59</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>4-62</td>
</tr>
<tr>
<td>TST(B)</td>
<td>4-11</td>
</tr>
<tr>
<td>WAIT</td>
<td>4-71</td>
</tr>
<tr>
<td>XOR</td>
<td>4-31</td>
</tr>
</tbody>
</table>
### NUMERICAL OP CODE LIST

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00</td>
<td>HALT</td>
<td>00 60 DD</td>
<td>ROR</td>
<td>10 40 00</td>
<td>EMT</td>
</tr>
<tr>
<td>00 00 01</td>
<td>WAIT</td>
<td>00 61 DD</td>
<td>ROL</td>
<td>10 43 77</td>
<td></td>
</tr>
<tr>
<td>00 00 02</td>
<td>RTI</td>
<td>00 62 DD</td>
<td>ASR</td>
<td>10 44 00</td>
<td>TRAP</td>
</tr>
<tr>
<td>00 00 03</td>
<td>BPT</td>
<td>00 63 DD</td>
<td>ASL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 00 04</td>
<td>IOT</td>
<td>00 64 NN</td>
<td>MARK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 00 05</td>
<td>RESET</td>
<td>00 65 SS</td>
<td>MFPI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 00 06</td>
<td>RTT</td>
<td>00 66 DD</td>
<td>MTPI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 00 07</td>
<td>(unused)</td>
<td>00 67 DD</td>
<td>SXT</td>
<td>10 47 77</td>
<td></td>
</tr>
<tr>
<td>00 01 DD</td>
<td>JMP</td>
<td>00 70 00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 02 0R</td>
<td>RTS</td>
<td>00 77 77</td>
<td>(unused)</td>
<td>10 50 DD</td>
<td>CLRB</td>
</tr>
<tr>
<td>00 02 10</td>
<td>(unused)</td>
<td>01 77 77</td>
<td>MOV</td>
<td>10 51 DD</td>
<td>COMB</td>
</tr>
<tr>
<td>00 02 27</td>
<td></td>
<td>01 SS DD</td>
<td>MOV</td>
<td>10 52 DD</td>
<td>INCB</td>
</tr>
<tr>
<td>00 02 3N</td>
<td>SPL</td>
<td>02 SS DD</td>
<td>CMP</td>
<td>10 53 DD</td>
<td>DECB</td>
</tr>
<tr>
<td>00 02 40</td>
<td>NOP</td>
<td>03 SS DD</td>
<td>BIT</td>
<td>10 54 DD</td>
<td>NEGB</td>
</tr>
<tr>
<td>00 02 41</td>
<td></td>
<td>04 SS DD</td>
<td>BIC</td>
<td>10 55 DD</td>
<td>ADCB</td>
</tr>
<tr>
<td>00 02 77</td>
<td></td>
<td>05 SS DD</td>
<td>BIS</td>
<td>10 56 DD</td>
<td>SBCB</td>
</tr>
<tr>
<td>00 03 DD</td>
<td>SWAB</td>
<td>06 SS DD</td>
<td>ADD</td>
<td>10 57 DD</td>
<td>TSTB</td>
</tr>
<tr>
<td>00 04 XXX</td>
<td>BR</td>
<td>07 OR SS</td>
<td>MUL</td>
<td>10 60 DD</td>
<td>RORB</td>
</tr>
<tr>
<td>00 04 XXX</td>
<td>BR</td>
<td>07 1R SS</td>
<td>DIV</td>
<td>10 61 DD</td>
<td>ROLB</td>
</tr>
<tr>
<td>00 04 XXX</td>
<td>BR</td>
<td>07 2R SS</td>
<td>ASH</td>
<td>10 62 DD</td>
<td>ASRB</td>
</tr>
<tr>
<td>00 04 XXX</td>
<td>BR</td>
<td>07 3R SS</td>
<td>ASHC</td>
<td>10 63 DD</td>
<td>ASLB</td>
</tr>
<tr>
<td>00 04 0R</td>
<td></td>
<td>07 4R DD</td>
<td>XOR</td>
<td>10 64 77</td>
<td>(unused)</td>
</tr>
<tr>
<td>00 10 XXX</td>
<td>BNE</td>
<td>07 50 0R</td>
<td>FADD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 14 XXX</td>
<td>BEQ</td>
<td>07 50 1R</td>
<td>FSUB</td>
<td>10 65 SS</td>
<td>MFPD</td>
</tr>
<tr>
<td>00 20 XXX</td>
<td>BGE</td>
<td>07 50 2R</td>
<td>FMUL</td>
<td>10 66 DD</td>
<td>MTPD</td>
</tr>
<tr>
<td>00 24 XXX</td>
<td>BLT</td>
<td>07 50 3R</td>
<td>FDIV</td>
<td>10 67 00</td>
<td></td>
</tr>
<tr>
<td>00 30 XXX</td>
<td>BGT</td>
<td>07 50 40</td>
<td></td>
<td></td>
<td>(unused)</td>
</tr>
<tr>
<td>00 34 XXX</td>
<td>BLE</td>
<td>07 67 77</td>
<td>(unused)</td>
<td>10 77 77</td>
<td></td>
</tr>
<tr>
<td>00 4R DD</td>
<td>JSR</td>
<td>07 67 77</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 50 DD</td>
<td>CLR</td>
<td>07 7R NN</td>
<td>SOB</td>
<td>11 SS DD</td>
<td>MOVB</td>
</tr>
<tr>
<td>00 51 DD</td>
<td>COM</td>
<td>07 8R NN</td>
<td>CMOD</td>
<td>12 SS DD</td>
<td>CMPB</td>
</tr>
<tr>
<td>00 52 DD</td>
<td>INC</td>
<td>10 00 XXX</td>
<td>BPL</td>
<td>13 SS DD</td>
<td>BITB</td>
</tr>
<tr>
<td>00 53 DD</td>
<td>DEC</td>
<td>10 04 XXX</td>
<td>BMI</td>
<td>14 SS DD</td>
<td>BICB</td>
</tr>
<tr>
<td>00 54 DD</td>
<td>NEG</td>
<td>10 10 XXX</td>
<td>BHI</td>
<td>15 SS DD</td>
<td>BISB</td>
</tr>
<tr>
<td>00 55 DD</td>
<td>ADC</td>
<td>10 14 XXX</td>
<td>BLOS</td>
<td>16 SS DD</td>
<td>SUB</td>
</tr>
<tr>
<td>00 56 DD</td>
<td>SBC</td>
<td>10 20 XXX</td>
<td>BVC</td>
<td>17 00 00</td>
<td></td>
</tr>
<tr>
<td>00 57 DD</td>
<td>TST</td>
<td>10 24 XXX</td>
<td>BVS</td>
<td></td>
<td>floating</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 30 XXX</td>
<td>BCC, BHIS</td>
<td></td>
<td>point</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 34 XXX</td>
<td>BCS, BLO</td>
<td></td>
<td></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**

- 
- SS = source field (6 bits)
- DD = 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)

**Operations**

- ( ) = contents of
- s = contents of source
- d = contents of destination
- r = contents of register
- ← = becomes
- X = relative address
- % = register definition

**Boolean**

- ∧ = AND
- ∨ = inclusive OR
- ⊕ = exclusive OR
- ~ = NOT

**Condition Codes**

- * = conditionally set or cleared
- − = not affected
- 0 = cleared
- 1 = set

### NOTE:

- ▲ = Applies to the 11/35, 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 Result</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR(B)</td>
<td>050DD</td>
<td>clear</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>COM(B)</td>
<td>051DD</td>
<td>complement (1's)</td>
<td>~d</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>INC(B)</td>
<td>052DD</td>
<td>increment</td>
<td>d + 1</td>
<td>*</td>
<td>*</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>
<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>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>TST(B)</td>
<td>057DD</td>
<td>test</td>
<td>0</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Rotate & Shift

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Instruction</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ROR(B)</td>
<td>060DD</td>
<td>rotate right</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROL(B)</td>
<td>061DD</td>
<td>rotate left</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASR(B)</td>
<td>062DD</td>
<td>arith shift right</td>
<td>d/2</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>ASL(B)</td>
<td>063DD</td>
<td>arith shift left</td>
<td>2d</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>SWAB</td>
<td>0003DD</td>
<td>swap bytes</td>
<td></td>
<td>*</td>
<td>*</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

### Multiple Precision

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Instruction</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC(B)</td>
<td>055DD</td>
<td>add carry</td>
<td>d + C</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>SBC(B)</td>
<td>056DD</td>
<td>subtract carry</td>
<td>d - C</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>SXT</td>
<td>0067DD</td>
<td>sign extend</td>
<td>0 or -1</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

### DOUBLE OPERAND: OPR src,dst OPR scr,R or OPR R,dst

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Instruction</th>
<th>Operation</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV(B)</td>
<td>1SSDD</td>
<td>move</td>
<td>d ← s</td>
<td></td>
<td>*</td>
<td>*</td>
<td>0</td>
</tr>
<tr>
<td>CMP(B)</td>
<td>2SSDD</td>
<td>compare</td>
<td>s - d</td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>ADD</td>
<td>06SSDD</td>
<td>add</td>
<td>d ← s + d</td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>SUB</td>
<td>16SSDD</td>
<td>subtract</td>
<td>d ← d - s</td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

### Logical

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Instruction</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT(B)</td>
<td>3SSDD</td>
<td>bit test (AND)</td>
<td>s ∧ d</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>BIC(B)</td>
<td>4SSDD</td>
<td>bit clear</td>
<td>d ← (~s) ∧ d</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>BIS(B)</td>
<td>5SSDD</td>
<td>bit set (OR)</td>
<td>d ← s ∨ d</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>*</td>
</tr>
</tbody>
</table>

### Register

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Instruction</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL</td>
<td>070RSS</td>
<td>multiply</td>
<td>r ← r × s</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>DIV</td>
<td>071RSS</td>
<td>divide</td>
<td>r ← r / s</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>ASH</td>
<td>072RSS</td>
<td>shift arithmetically</td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>ASHC</td>
<td>073RSS</td>
<td>arith shift combined</td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>XOR</td>
<td>074RDD</td>
<td>exclusive OR</td>
<td>d ← r ⊕ d</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>*</td>
</tr>
</tbody>
</table>

---

*E-2*
### BRANCH

**B - location**

```
15       8     7     xxx  0
BASE CODE
```

If condition is satisfied:
Branch to location,
New PC ← Updated PC + (2 x offset)

\[ \text{Op Code} = \text{Base Code} + XXX \]

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Base Code</th>
<th>Instruction</th>
<th>Branch Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Branches</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<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>Base 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>( \geq 0 ) ( N \lor V = 0 )</td>
</tr>
<tr>
<td>BLT</td>
<td>002400</td>
<td>br if less than (0)</td>
<td>( &lt; 0 ) ( N \lor V = 1 )</td>
</tr>
<tr>
<td>BGT</td>
<td>003000</td>
<td>br if greater than (0)</td>
<td>( &gt; 0 ) ( Z \land (N \lor V) = 0 )</td>
</tr>
<tr>
<td>BLE</td>
<td>003400</td>
<td>br if less or equal (to 0)</td>
<td>( \leq 0 ) ( Z \land (N \lor V) = 1 )</td>
</tr>
</tbody>
</table>

**Unsigned Conditional Branches**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Base 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 \lor Z = 0 )</td>
</tr>
<tr>
<td>BLOS</td>
<td>101400</td>
<td>branch if lower or same</td>
<td>( C \lor 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><strong>JMP</strong></td>
<td>0001DD</td>
<td>jump</td>
<td>PC ← dst</td>
</tr>
<tr>
<td><strong>JSR</strong></td>
<td>004RDD</td>
<td>jump to subroutine</td>
<td>use same R</td>
</tr>
<tr>
<td><strong>RTS</strong></td>
<td>00020R</td>
<td>return from subroutine</td>
<td>aid in subr return (R) − 1, then if (R) ≠ 0:</td>
</tr>
<tr>
<td><strong>MARK</strong></td>
<td>0064NN</td>
<td>mark</td>
<td>PC ← Updated PC − (2 x NN)</td>
</tr>
<tr>
<td><strong>SOB</strong></td>
<td>077RNN</td>
<td>subtract 1 &amp; br (if ≠ 0)</td>
<td></td>
</tr>
</tbody>
</table>

E-3
### 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</td>
<td>emulator trap</td>
<td>PC at 30, PS at 32</td>
</tr>
<tr>
<td></td>
<td>to 104377</td>
<td>(not for general use)</td>
<td></td>
</tr>
<tr>
<td>TRAP</td>
<td>104400</td>
<td>trap</td>
<td>PC at 34, PS at 36</td>
</tr>
<tr>
<td></td>
<td>to 104777</td>
<td></td>
<td></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:

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Code</th>
<th>Instruction</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLC</td>
<td>000241</td>
<td>clear C</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>0</td>
</tr>
<tr>
<td>CLV</td>
<td>000242</td>
<td>clear V</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>1</td>
</tr>
<tr>
<td>CLZ</td>
<td>000244</td>
<td>clear Z</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>1</td>
</tr>
<tr>
<td>CLN</td>
<td>000250</td>
<td>clear N</td>
<td>0</td>
<td>_</td>
<td>_</td>
<td>_</td>
</tr>
<tr>
<td>CCC</td>
<td>000257</td>
<td>clear all cc bits</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SEC</td>
<td>000261</td>
<td>set C</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>1</td>
</tr>
<tr>
<td>SEV</td>
<td>000262</td>
<td>set V</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>1</td>
</tr>
<tr>
<td>SEZ</td>
<td>000264</td>
<td>set Z</td>
<td>1</td>
<td>_</td>
<td>_</td>
<td>_</td>
</tr>
<tr>
<td>SEN</td>
<td>000270</td>
<td>set N</td>
<td>1</td>
<td>_</td>
<td>_</td>
<td>_</td>
</tr>
<tr>
<td>SCC</td>
<td>000277</td>
<td>set all cc bits</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

---

E-4
### PDP11/40 FLOATING POINT UNIT:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Description</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>FADD</td>
<td>07500R</td>
<td>floating add</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FSUB</td>
<td>07501R</td>
<td>floating subtract</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FMUL</td>
<td>07502R</td>
<td>floating multiply</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FDIV</td>
<td>07503R</td>
<td>floating divide</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### DEVICE REGISTER ADDRESSES

<table>
<thead>
<tr>
<th>Device</th>
<th>Description</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</td>
<td>BR6</td>
</tr>
<tr>
<td>KW11-P</td>
<td>Real Time Clock control &amp; status counter</td>
<td>772 540, 772 544</td>
<td>772 542</td>
<td>104</td>
<td>BR6</td>
</tr>
<tr>
<td>LA30</td>
<td>DECwriter keyboard printer</td>
<td>777 560, 777 564</td>
<td>777 562, 777 566</td>
<td>60, 64</td>
<td>BR4</td>
</tr>
<tr>
<td>LP11</td>
<td>Line Printer</td>
<td>777 514</td>
<td>777 516</td>
<td>200</td>
<td>BR4</td>
</tr>
<tr>
<td>LT33</td>
<td>Teletype keyboard printer</td>
<td>777 560, 777 564</td>
<td>777 562, 777 566</td>
<td>60, 64</td>
<td>BR4</td>
</tr>
<tr>
<td>PC11</td>
<td>Paper Tape reader punch</td>
<td>777 550, 777 554</td>
<td>777 552, 777 556</td>
<td>70, 74</td>
<td>BR4</td>
</tr>
<tr>
<td>RC11/RS64</td>
<td>Disk (64K words) look ahead disk address error status command &amp; status word count current address maintenance</td>
<td>777 440, 777 442, 777 444, 777 446, 777 450, 777 452, 777 454</td>
<td>777 456</td>
<td>210</td>
<td>BR5</td>
</tr>
<tr>
<td>RF11/RS11</td>
<td>Disk (256K words) control status word count current mem addr disk address adrsh ext error maintenance segment address</td>
<td>777 460, 777 462, 777 464, 777 466, 777 470, 777 474, 777 476</td>
<td>777 472</td>
<td>204</td>
<td>BR5</td>
</tr>
</tbody>
</table>
RK11/RK05  Disk Cartridge  777 416 220  BR5
 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  777 350 214  BR6
 control  777 340
 command  777 342
 word count  777 344
 current address  777 346

TM11/TU10  Magtape  772 530 224  BR5
 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

General Registers
(console use only)
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)  240  PIRQ
004  Time Out & other errors  244  Floating Point
010  illegal & reserved instr  250  Memory Management
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>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>744</td>
<td>016 701</td>
<td>764</td>
<td>000 002</td>
</tr>
<tr>
<td>746</td>
<td>000 026</td>
<td>766</td>
<td>005 400</td>
</tr>
<tr>
<td>750</td>
<td>012 702</td>
<td>770</td>
<td>005 267</td>
</tr>
<tr>
<td>752</td>
<td>000 352</td>
<td>772</td>
<td>177 756</td>
</tr>
<tr>
<td>754</td>
<td>005 211</td>
<td>774</td>
<td>000 765</td>
</tr>
<tr>
<td>756</td>
<td>105 711</td>
<td>776</td>
<td>177 560 (KB) or 177 550 (PR)</td>
</tr>
<tr>
<td>760</td>
<td>100 376</td>
<td></td>
<td></td>
</tr>
<tr>
<td>762</td>
<td>116 162</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
INTERNATIONAL

NETHERLANDS
Digital Equipment N.V.
The Hague
Sir Winston Churchilllaan 370
Rijswijk/The Hague, Netherlands
Telephone: 070-995-160
Telex: 32533

BELGIUM
Digital Equipment N.V./S.A.
BRUSSELS
108 Rue De l’Arton
1040 Brussels, Belgium
Telephone: 02-138256
Telex: 25297

SWEDEN
Digital Equipment AB
STOCKHOLM
Engelsvägen 7, 171 41 Solna, Sweden
Telephone: 98 13 90
Telex: 170 50
Cable: Digital Stockholm

NORWAY
Digital Equipment Corp. A/S
OSLO
Tromsdammeveien 47
Oslo 5, Norway
Telephone: 02/68 34 40
Telex: 19705 DEC N

DENMARK
Digital Equipment Aktieselskab
COPENHAGEN
Helevopaeg 9
2900 Hellerup, Denmark

SWITZERLAND
Digital Equipment Corporation S.A.
GENEVA
81 Route de l’Aire
1211 Geneva Switzerland
Telephone: 42 79 50
Telex: 22 683

ZURICH
Schweizerstrasse 21
CH-8006 Zurich, Switzerland
Telephone: 01/60 35 95
Telex: 56039

ITALY
Digital Equipment S.p.A.
MILAN
Corso Garibaldi 49, 20121 Milano, Italy
Telephone: 827 746 694 349
Telex: 38165

SPAIN
Digital Equipment Corporation Ltd.
MADRID
Atos Ingenieros S.A., Enrique Larretas 12, Madrid 16
Telephone: 215 35 43
Telex: 27249

BARCELONA
Atos Ingenieros S.A., Ganduxer 16, Barcelona 6
Telephone: 22 44 55

CANADA
Digital Equipment of Canada, Ltd.
CANADIAN HEADQUARTERS
P.O. Box 11500
Ottawa, Ontario, Canada
KH2 9B3
Telephone: (613)-582-1111
Telex: 610-562-8732

CANADA (cont.)
OTTAWA
P.O. Box 1150
Ottawa, Ontario
KH2 779
Telephone: (613)-562-5120
Telex: 610-562-9070

TORONTO
230 Lakeshore Road East, Port Credit, Ontario
Telephone: (416)-274-1241
Telex: 610-492-4306

MONTREAL
596 Cote de Liesse Road
Dorval, Quebec, Canada J3H 7G9
Telephone: (514)-336-9339
Telex: 610-422-4124

CALGARY
Edmonton
Suite 140, 990 Fisher Road S.E.
Calgary, Alberta, Canada
Telephone: (403)-435-4691
Telex: 430-255-7400

VANCOUVER
2110 West 12th Avenue
Vancouver 9, British Columbia, Canada
Telephone: (604)-736-7616
Telex: 610-509-2006

GENERAL INTERNATIONAL SALES
REGIONAL OFFICE
146 Main Street, Maynard Massachusetts 01754
Telephone: (508)-867-5111
From Metropolitan Boston 646-6600
Telex: 710-347-0211

CABLE: DIGITAL MAYN
Telex: 94 9457

AUSTRALIA
Digital Equipment Australia Pty. Ltd.
SYDNEY
P.O. Box 491, Crown Nest
N.S.W. Australia 2065
Telephone: 439-2066
Telex: A230740
Cable: Digital, Sydney

MELBOURNE
60 Park Street, South Melbourne, Victoria, 3005
Telephone: 699-3988
Telex: AA44615

PERTH
643 Murray Street
West Perth, Western Australia 6005
Telephone: 24-4893
Telex: A201470

BRISBANE
138 Morville Street, South Brisbane
Queensland, Australia 4100
Telephone: 44-4047
Telex: A400161

ADELAIDE
5 Moreton Avenue
Norwood, South Australia 5067
Telephone: 63-1338
Telex: A080255

NEW ZEALAND
Digital Equipment Corporation Ltd.
AUCKLAND
Hilton House, 430 Queen Street, Box 2471
Auckland, New Zealand
Telephone: 720-323

JAPAN
Digital Equipment Corporation International
TOKYO
Kows Building No. 17, Second Floor
2-7 Nakano-azabu 1-Chome
Minato-Ku, Tokyo, Japan
Telephone: 404-5064/6
Telex: TK-6438

JAPAN (cont.)
Rikai Trading Co., Ltd. (seeks only)
Kozato-Kaikan Bldg.
No. 16-14, Nihonbashibashi 1-chome
Minato-Ku, Tokyo, Japan
Telephone: 505-3426
Telex: 781-4208

PUERTO RICO
Digital Equipment Corporation of Puerto Rico
MIRAMAR
American Airlines Bldg.
604 Ponce De Leon, Miramar, Puerto Rico
Telephone: 809-723-8661/7
Telex: 365-9056

ARGENTINA
BUENOS AIRES
Cossen S.A.
Virrey del Pino 4071, Buenos Aires
Telephone: 52-3185
Telex: 012-2584

BRAZIL
RIO DE JANEIRO — GB
Ambix S.A.
Rua Cesáreo, 104, 2. a e 3. a andares
Fones: 221-6856/4, 225-9732
Cables: RAIGCARDIO
SÃO PAULO — SP
Ambix S.A.
Rua Topy, 235
Fones: 51-9012, 52-8655, 52-7806
Cables: RAIGCARDIO
PORTO ALEGRE — RS
Ambix S.A.
Rua Cel. Vicente, 421, 1. a andar
Fones: 244-111, 24-1906

CHILE
SANTIAGO
Cossen Chile Ltda. (seeks only)
Casilla 1616, Correo 15, Santiago
Telephone: 286713
Cables: COACHIND

MEXICO
MEXICO CITY
Manzat, S.A.
Eugenia 408 Deptos. 1
Apoq, Postal 12-1013

MEXICO 12, D.F.
Telephone: (060)-536-0610

PHILIPPINES
MANILA
Stanford Computer Corporation
P.O. Box 1608
416 Damien St., Manila
Telephone: 746-9052
Telex: 742-0322

VENEZUELA
CARACAS
Coxwor, C.A.
Apartado 50939
Sabinas Grande No. 1, Caracas 105
Telephone: 72-6662, 72-6673
Cables: INSTRUENV