DOCUMENT RESUME

ED 177 020 SE 029 188

TITLE PDP-8 Introductory Minicomputer Laboratory Manual. Digital Systems Education Committee Instructional Tools Task Force.

INSTITUTION Pittsburgh Univ., Pa. Dept. of Electrical Engineering.

SPONS AGENCY National Science Foundation, Washington, D.C.
PUB DATE 76
GRANT NSF-GZ-2957
NOTE 117p.

EDRS PRICE MF01/PC05 Plus Postage.
DESCRIPTORS Computer Based Laboratories; Computer Programs; Computers; *Computer Science Education; *Digital Computers; *Engineering Education; Higher Education; *Independent Study; Instruction; Instructional Materials; Manuals; Programed Materials

ABSTRACT

This is a self-study manual designed for freshman or sophomore engineering students who have interest in the organizational and operational concepts of the digital computer, but little or no experience with such computers. The manual gives an introduction to a general purpose minicomputer, Digital Equipments Corporation's PDP-8. (MK)

IT-01

PDP-8 INTRODUCTORY MINICOMPUTER LABORATORY MANUAL

ELECTRICAL ENGINEERING DEPARTMENT
UNIVERSITY OF PITTSBURGH
PITTSBURGH
PENNSYLVANIA

DIGITAL SYSTEMS EDUCATION COMMITTEE
INSTRUCTIONAL TOOLS TASK FORCE

DISE is a project supported by the National Science Foundation
Grant No. GZ--2997

DISE PROJECT

The DISE (DIgital Systems Education) Project is sponsored by Grant #GZ-2997 from the National Science Foundation. The nucleus of the project is the DISE Advisory Committee, which is an inter-university, inter-industry working group with the specific charter of developing, coordinating the development of, and distributing educational/instructional materials in the digital systems area.

The specific goals of the project are: to assess Digital Systems Education, both to determine the types of curricula, course contents, lab structures, etc., in present programs and to determine how present programs are meeting the needs of industry and the students; to review existing educational/instructional materials in this area; to develop and/or coordinate the development of new materials; to provide a industry/university forum to foster the exchange of new technology; and to obtain widespread dissemination and use of newly developed or existing materials..

Project Structure

Advisory Committee. It is the responsibility of this group to determine the areas in which the educational/instructional materials will be solicited and developed and to act as a review board for proposed projects and completed materials. The committee will consist of between 15 and 20 members with the academic, industrial, and professional societies sectors represented. The present members of this committee are:

Dr. Wayne Black
Charles T. Main, Inc.

Professor Taylor Booth
University of Connecticut

Professor Thomas Brubaker
Colorado State University

Professor James T. Cain
University of Pittsburgh

Professor Yaohan Chu
University of Maryland

Professor Ben Coates
Purdue University

Professor Ronald G. Hoelzeman
University of Pittsburgh

Dr. Glen Langdon
IBM Research

Professor Arthur Lo.
Princeton University

Mr. Francis Lynch
National Semiconductor Corporation

Professor Larry McNamee
UCLA

Professor T.W. Sze
University of Pittsburgh

Professor H.C. Torng
Cornell University

Professor Raymond Voith
University of Toledo

Task Force Committees (Project Groups). These committees or groups represent the "manufacturing" division of DISE and constitute the working groups of people developing materials for dissemination. There are presently five project groups, although new groups will be formed due to changes in technology, demand, or interest levels.

For further information, contact:

DISE
Electrical Engineering Department
University of Pittsburgh
Pittsburgh, Pennsylvania 15261

PREFACE

This manual is designed for the freshman or sophomore level engineering student who has no knowledge of a computer system. It is to serve as an introduction to a Digital System (a general purpose minicomputer, Digital Equipment Corporation's PDP-8) from a programming or user's viewpoint. The intent of this manual is to present organizational and operational concepts of the digital computer to the student who has the interest in the subject, but little or no experience. This manual is self study in nature, and when used with the PDP-8, should allow the student to, on his own, master the operation of the computer as well as several basic digital system architecture concepts.

The solution to each exercise is given after each problem definition with the intent of demonstrating the subject material. There are also several questions in each chapter which the reader is asked to solve. The solution will usually follow directly from the section or exercise preceding the question(s). A general knowledge of binary and octal number systems will be helpful to the reader. However, there is an appendix with sufficient background information on these number systems for the reader to handle this booklet.

TABLE OF CONTENTS

CHAPTER 1

Front Panel

In this first section, the reader will sit down and perform some simple operations on the PDP-8/I computer, with the objectives of familiarization with the controls on the front panel, and what one can do from the panel.

In the lower left-hand corner of the panel is a key-type power switch. (On the PDP-8/S the power key slot is on the lower right, and the panel lock key slot is on the lower left.) Push the key in slightly and turn to the right to the "power" position. The computer is ready to use.

The first exercise will demonstrate how to manually load information into specific locations in the memory of the computer. The PDP-8 has 4,096 ($=2^{12}$) "slots" called words in its memory. The memory is the place where the computer stores or remembers information entered by the programmer, or calculated by the machine. The memory is called "random access"; any location is just as easily accessed as another. Information in the PDP-8 is organized into words of 12 binary digits, or "12 bits", which the computer interprets as instructions or as data to be operated on.

The twelve switches, beginning with the seventh switch from the left, are called the SWITCH REGISTER (Figure 1.1). The switch register is used to manually enter information into the machine. Note that each of the twelve switches of the switch register correspond to a particular binary digit in the display on the panel labeled the PROGRAM COUNTER, MEMORY ADDRESS, MEMORY BUFFER and ACCUMULATOR. The nineteenth through twenty-sixth switches are for some general functions. These switches and displays will be presented as the reader progresses.

EXERCISE 1.1

Load five binary numbers into five consecutive locations in the computer memory; then examine these locations and verify that these numbers have been

stored. The five binary numbers are:

000 000 000 001

000 000 000 111

000 000 111 111

000 111 111 111

111 111 111 111

Procedure: The programmer must first "tell" the computer where to store the first number. Suppose the location for the first number will be 001 000 000 000. Set the SWITCH REGISTER to this value; the "up" position of a switch is a '0' and "down" is a '1'. (On PDP-8/S switch register up = 1, down = 0.) Then, to tell the computer this is the address in memory you are interested in, press the LOAD ADD switch and release. Observe that the setting of the switch register is now shown in the display labeled PROGRAM COUNTER. (An "on" light is a '1' and an "off" light is a zero.) The program counter is a circuit which the computer uses to keep track for itself where in memory it is to "find" something to do next. But the computer has still done nothing to location 001 000 000 000. To place the first of the five numbers in this location, set the switch register now to 000 000 000 001, and lift DEP (deposit) switch to operate. The location in memory which is numbered 001 000 000 000 now holds in it the number 000 000 000 001. The programmer has a visual check of this fact by the displays on the panel labeled MEMORY ADDRESS and MEMORY BUFFER.

Memory Address shows the address of a location in memory and Memory Buffer shows what is contained in that particular memory location. Note that after lifting DEP, the Memory Address reads 001 000 000 000 and Memory Buffer reads 000 000 000 001. Also note that now the PROGRAM COUNTER reads 001 000 000 001; it has automatically incremented itself by one, to indicate the next consecutive memory location address. The programmer can now deposit the next number without

loading the next consecutive address. Therefore, set the Switch Register to 000 000 000 111 and lift DEP. Repeat this procedure for the next three numbers, observing that the Memory Buffer and Memory Address displays will verify the deposit operations.

Note that the program counter still incremented itself again after the fifth deposit operation.

Now, to go back and check these five locations, first, "tell" the computer the first address you are interested in (i.e., 001 000 000 000). Set the switch register to 001 000 000 000 and press LOAD ADD. To check the contents of this location, press and release the EXAM (examine switch). The Memory Address display will read 001 000 000 000 and the Memory Buffer will read 000 000 000 001. The program counter has again incremented itself to 001 000 000 001, and the next consecutive memory location can be displayed by pressing EXAM again. Examine the next three locations in the same way.

Note that pressing EXAM does not affect any memory location contents; it merely lets the programmer "look" into memory.

Summary: This exercise has presented a method for the programmer to load information into any memory location and to check and modify the contents, if necessary.

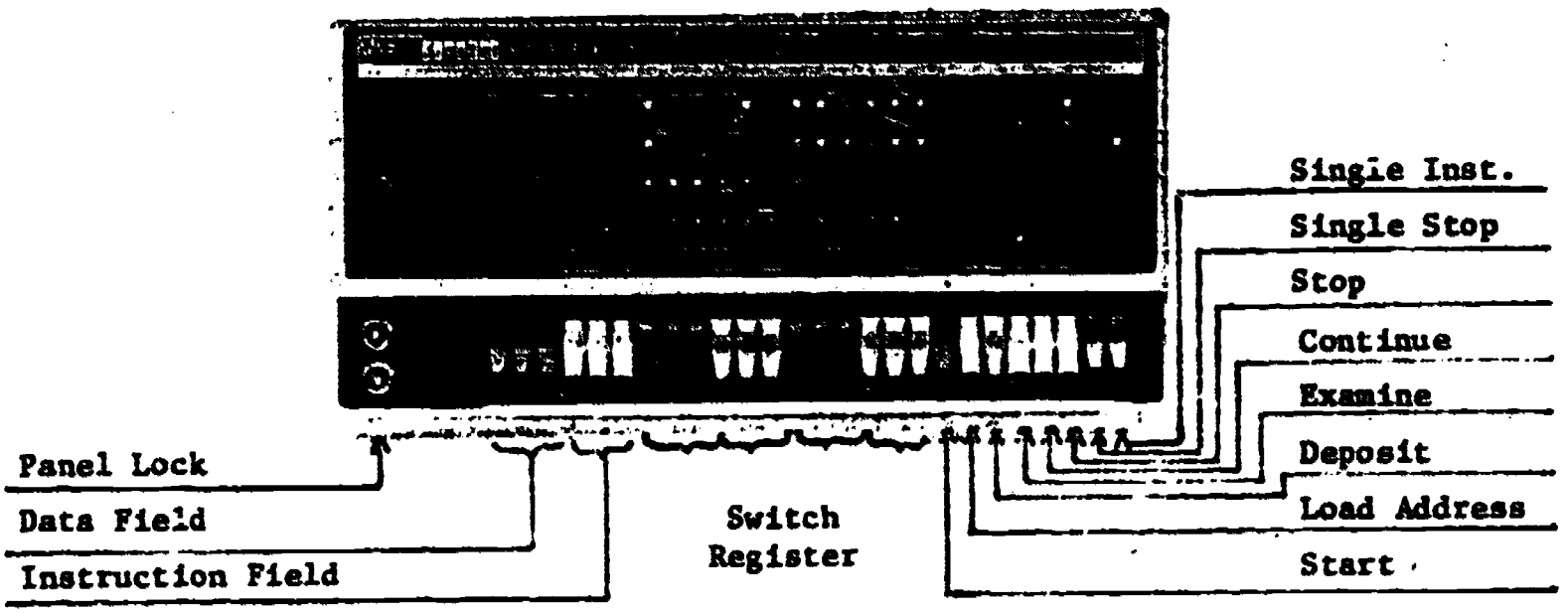


Figure 1.1

EXERCISE 1.2

Run a sample program to clear 200_8 (128_{10}) consecutive memory locations beginning with location 001 000 000 000. It is suggested that Exercise 1.1 be completed before this exercise. Because this exercise is meant to merely familiarize you with the controls of the computer, we postpone until later the explanation of why this particular series of ones and zeroes accomplishes what we say it does. For now just take it on faith.

Memory Location	Contents
000 000 101 000	111 110 000 000
000 000 101 001	011 100 101 111
000 000 101 010	010 000 101 111
000 000 101 011	010 000 101 110
000 000 101 100	101 000 101 001
000 000 101 101	111 100 000 010
000 000 101 110	111 110 000 000
000 000 101 111	001 000 000 000

Procedure: Deposit these instructions into the given locations using the procedure in Exercise 1.1. Note that instructions are actually coded in binary numbers. Also use the procedure from Exercise 1.1 to check that the instructions were loaded correctly. To run the program, set the starting address, i.e. 000 000 101 000 into the Switch Register, press LOAD ADD, and tell the computer to begin execution of the program by pressing START. The lights on the display will go out and stop at some different setting. If the memory buffer display reads 111 100 000 010, you know the program has stopped. This instruction is read by the machine as "halt execution." (In the PDP-8/S the memory buffer should read 000 000 000 000, and the memory address should be 56.)

Next examine the locations from 001 000 000 000 to 001 001 111 111 (1000_8 to 1177_8). They should all read 000 000 000 000.

Summary: Exercise 1.2 illustrates how to execute a program which is stored in memory; and again, to check results of the program operation.

QUESTION 1.1

How would you manually clear memory locations by using the panel instead of using a stored program to do it? (Hint: Exercise 1.1).

ANSWER TO QUESTION 1.1

First, set the switch register to the address of the desired memory location. Then press LOAD ADD. Next, set the switch register to all zeros (000 000 000 000) and press the DEP switch. This procedure deposits a word of all zeros into the desired memory location, i.e., that location has been "cleared." To clear successive memory locations, load the address of the first location to be cleared, deposit zeroes in it, and continue to press the DEP switch. Each "deposit" will clear the next successive memory location, because the computer increments the program counter after each deposit; the computer is then pointing to the next successive location.

CHAPTER 2

Memory

Chapter 1 introduced the memory of the PDP-8. The $4{,}096_{10}$ words of memory are in a structure called MAGNETIC CORE MEMORY. The core is composed of an array of small magnetic "donuts" interlinked by a series of criss-crossing wires. A logical '0' or a logical '1' will be set on a particular "donut" depending on which direction it is magnetized by the current passing through it. A 12 bit word is set in memory by magnetizing a group of 12 "donuts" in memory. The 4,096 word memory is thus composed of 12 x 4,096 = 49,152 magnetic "donuts."

To make it easier to handle the words in memory, the 4,096 words are subdivided into PAGES. Each page holds a set number of words just as a page in a book. In the PDP-8, there are 32_{10} pages, each containing 128_{10} words. There is actually no physical barrier between the pages, but just as in a page in a book, the machine "looks" at one page in memory at a time. The idea of accessing different pages by direct and indirect addressing techniques will be covered in later sections.

A digital computer operates on information stored in its memory by means of a section called the Arithmetic Unit. The most important part of the arithmetic unit of the PDP-8 is a 12 bit register called the ACCUMULATOR. A register is a temporary storage area of data obtained from memory where operations of the arithmetic unit are performed. The accumulator is connected to the memory. One can think of the accumulator as a "scratch pad"; it can retrieve information held in memory, perform operations on it, and return the result of these operations to memory. The programmer can "see" the contents of the accumulator by means of the display on the front panel labeled ACCUMULATOR. To realize how the accumulator is utilized during a program, run the following exercise on the computer.

EXERCISE 2.1

Have the computer move the five numbers in locations 010 000 000 000 to 010 000 000 100 (2000_8 to 2004_8), using the accumulator, into the locations 000 000 101 000 to 000 000 101 100 (50_8 to 54_8).

Procedure: First, load the five numbers into the locations given below, using the techniques from Exercise 1.1.

Memory Location	Contents
010 000 000 000 (2000_8)	101 101 101 101 (5555_8)
010 000 000 001 (2001_8)	010 010 010 010 (2222_8)
010 000 000 010 (2002_8)	001 001 001 001 (1111_8)
010 000 000 011 (2003_8)	000 000 000 000 (0000_8)
010 000 000 100 (2004_8)	110 101 110 101 (6565_8)

The binary code for the program to do this exercise is given below. Using the procedure in exercise 1.2, load the contents into the corresponding locations:

Memory Location	Contents
000 100 000 000 (0400_8)	111 011 000 000 (7300_8)
000 100 000 001 (0401_8)	001 010 001 100 (1214_8)
000 100 000 010 (0402_8)	011 010 001 101 (3215_8)
000 100 000 011 (0403_8)	001 110 001 110 (1616_8)
000 100 000 100 (0404_8)	111 100 000 010 (7402_8)
000 100 000 101 (0405_8)	011 110 001 111 (3617_8)
000 100 000 110 (0406_8)	010 010 001 110 (2216_8)
000 100 000 111 (1047_8)	010 010 001 111 (2217_8)
000 100 001 000 (0410_8)	010 010 001 101 (2215_8)

Memory Location	Contents
000 100 001 001 (0411_8)	101 010 000 011 (5203_8)
000 100 001 010 (0412_8)	111 100 000 010 (7402_8)
000 100 001 011 (0413_8)	101 010 001 010 (5212_8)
000 100 001 100 (0414_8)	111 111 111 011 (7773_8)
000 100 001 101 (0415_8)	000 000 000 000 (0000_8)
000 100 001 110 (0416_8)	010 000 000 000 (2000_8)
000 100 001 111 (0417_8)	000 000 101 000 (0050_8)

Read pages 12 and 13 before running the program.

Note: The numbers in the parentheses above are much easier to read than the corresponding binary strings. The four digit numbers in parentheses are the octal representations of the binary strings before them. The 12 binary digits of a word are arranged in four groups, each group containing three binary digits. Each of these four groups of binary digits can be converted into their equivalent octal number. (Also see appendix.)

EXAMPLE 2.1

111 001 010 101

(1x4)+(0x2)+(1x1) =5
(0x4)+(1x2)+(0x1) =2
(0x4)+(0x2)+(1x1) =1
(1x4)+(1x2)+(1x1) =7

then, 111 001 010 101 $=7125_8$

EXAMPLE 2.2

110 100 000 011 = 6403_8

6 4 0 3

QUESTION 2.1

What would be the octal representation of the following binary numbers?

001 010 011 100

011 101 110 000

000 000 111 010

110 101 111 001

010 001 011 111

QUESTION 2.2

What would be the binary representation of the following octal numbers?

6210

5114

1062

0047

With some practice, the reader should find that this octal representation is easier to handle rather than the binary strings. (Remember, however, that the computer still only understands the binary numbers.)

Check that the five numbers were properly loaded into locations 2000_8 to 2004_8, and also check that the program is loaded correctly. Briefly, the program takes the first of the five numbers, "moves" it to the accumulator, finds the address where it is to be deposited in memory, and then deposits it in the memory location. The program then "erases" (clears) the accumulator and gets the next number to be moved.

Two "halt" commands have been inserted in the program. The first will allow the programmer to see that the numbers in locations 2000_8 to 2004_8 are first loaded into the accumulator before they are transferred to the memory

locations 0050_8 to 0054_8. The second "halt" command will signify the end of the program.

Once the program has been loaded into the computer, set the switch register to 0400_8 (starting address of the program), LOAD ADD and press START. The program will stop when it comes to the first "halt" command. Note that the accumulator display on the front panel will contain 101 101 101 101 (5555_8), the contents of location 2000_8. To resume continuation of the program with the next instruction after the "halt", press the CONT (continue) switch on the front panel. The accumulator will now contain the contents of location 2001_8, which is 2222_8. (Before 2222_8 was moved to the accumulator, 5555_8 was moved to location 0050_8 so that the accumulator could accept the next number.) When the CONT button has been pressed for the fifth time, all five numbers will have been moved to locations 0050_8 to 0054_8. At this point, the program counter will contain 0413_8, the memory address will contain 0412_8, and the memory buffer will contain 7402_8. (Memory address has 413_8 and the memory buffer has 5212_8 in the in the PDP-8/S.) The program run has been completed.

To verify that the five numbers have been transferred, load 0050_8 into the switch register, and press LOAD ADD. To examine the contents of locations 0050_8 to 0054_8 press the EXAM button five consecutive times.

Summary: This example demonstrates that information can be entered into the accumulator from one memory location, and then can be transferred to another location in memory. Also, the idea that octal representation is easier to handle than binary, should be noted.

ANSWERS TO CHAPTER 2 QUESTIONS

Question 2.1 a) 001 010 011 100

(1x4) + (0x2) + (0x1) = 4

(0x4) + (1x2) + (1x1) = 3

(0x4) + (1x2) + (0x2) = 2

(0x4) + (0x2) + (1x1) = 1

then, 001 010 011 100 = 1234_8

Similarly b) 011 101 110 000 = 3560_8

c) 000 000 111 010 = 0072_8

d) 110 101 111 001 = 6571_8

e) 010 001 011 111 = 2137_8

Question 2.2 a) 6210_8

6_8 = 110

2 = 010

1 = 001

0 = 000

then, 6210_8 = 110 010 001 000

b) 5114_8 = 101 001 001 100

c) 1062 = 001 000 110 010

d) 0047 = 000 000 100 111

e) 0327 = 000 011 010 111

CHAPTER 3

Instructions

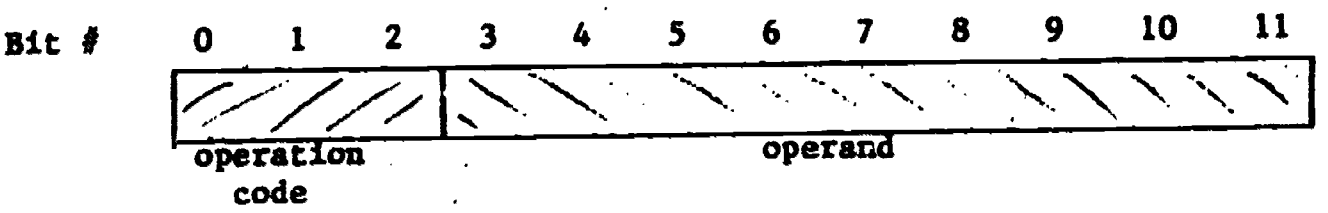
The previous material presented some short example programs which the reader would execute. This section will show how the computer takes these sequences of octal numbers (remember, they are actually binary strings) and interprets them to be "command" or "data" words.

Excluding an equipment failure, the computer can only do what it is "told" to do. The programmer tells the computer what to do by use of one or more 12-bit instruction words. An instruction word specifies to the machine what operation to perform and/or where to find the data upon which to carry out this operation.

The first major class of instruction words is the Memory Reference Instructions. They provide a means for the programmer to have the computer access, and operate on, data which is stored in memory. There are two parts to a memory reference instruction:

1. The operation code
2. The operand

The Operation Code is an octal number (actually, a three-bit binary number) which the computer translates into a command. On the PDP-8, the operation code is located in the three left-most bits of the instruction word. The remaining nine bits are the Operand. The operand does the memory referencing; it tells the computer the address of the data word, which the translated instruction will work on:



The PDP-8 has a set of six memory reference instructions. The first column of the table below gives the three-letter mnemonics, which make it easier to remember the instructions:

TABLE 3.1

	binary code	octal code	
AND	000	0	Logical AND of a word in memory with the accumulator
TAD	001	1	Two's complement ADd a word in memory to the accumulator
ISZ	010	2	Increment a word in memory and Skip next step if result is Zero
DCA	011	3	Deposit into memory and Clear Accumulator
JMS	100	4	JuMp to Subroutine
JMP	101	4	JuMP

In order to simplify the explanation of these instructions, the concept of "pages" in memory (first introduced in Chapter 2) must be expanded.

The operand part of an instruction word is divided into three sections:

1. Address Mode Bit -- 1 bit
2. "Page" Bit -- 1 bit
3. Page Address Bits -- 7 bits

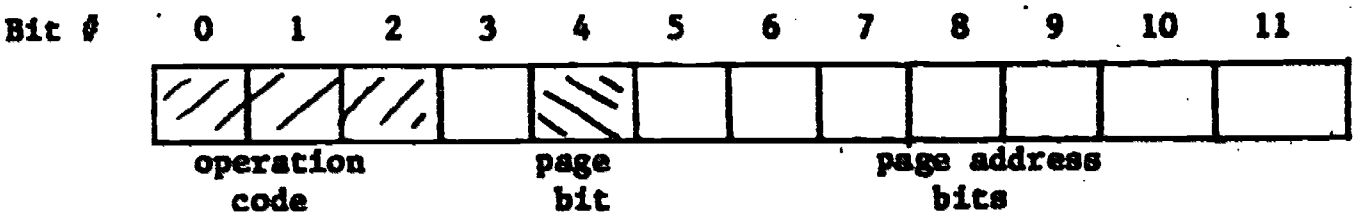
9 bits total in length

The 7 page address bits can "call out," or address 200_8 (128_{10}) locations in memory, i.e., 000_8 to 177_8.

But as stated before, the PDP-8 has 4,096 memory locations. How, then, can the computer access all of this available memory? This is where the memory pages come in. The 4,096 memory locations are divided into 32_{10} "pages" each of length 200_8 (128_{10}) locations. Table 3.2 gives the page breakdown of the memory locations.

When executing a program, the computer can only "look" directly at the memory in one of two ways:

It can look at the first page, called page zero, or at the page from which it is presently getting its instructions, the "current page". The section of the operand, called the page bit, tells the computer which page the address specified by the 7 page address bits is referring to: either "page 0", if the page bit is a '0', or "current page" if the bit is a '1'.



The purpose of bit #3 will be discussed later in the section on Indirect Addressing.

EXERCISE 3.1

Translate the following octal instructions words: 5050, 5250. Assume that the current page is page 4.

Solution:

a) 5050_8 ----- 101 0 0 0 101 000 (the 12 binary bits)

5=JMP "page 0" "location 50_8"

The instruction is: JuMP to location 50_8 on page "0". (That is, absolute location 50_8, since page 0 is the first page.)

TABLE 3.2

Memory Page	Octal Memory Locations	Memory Page	Octal Memory Locations
0	0–177	20	4000–4177
1	200–377	21	4200–4377
2	400–577	22	4400–4577
3	600–777	23	4600–4777
4	1000–1177	24	5000–5177
5	1200–1377	25	5200–5377
6	1400–1577	26	5400–5577
7	1600–1777	27	5600–5777
10	2000–2177	30	6000–6177
11	2200–2377	31	6200–6377
12	2400–2577	32	6400–6577
13	2600–2777	33	6600–6777
14	3000–3177	34	7000–7177
15	3200–3377	35	7200–7377
16	3400–3577	36	7400–7577
17	3600–3777	37	7600–7777

b) 5250_8 ——— 101 (5=JMP) 0 1 0 ("current page") 101 000 ("location 50_8") (the 12 binary bits)

The instruction is JuMP to location 50_8 on the current page. If the instruction was stored on page 4 of memory (which is locations 1000_8 to 1177_8) this instruction would tell the computer to jump to location ($1000_8 + 50_8$) = absolute location 1050_8 (JMP 1050_8).

EXERCISE 3.2

Translate the following octal instruction words. Assume that the current page is page 5.

1001

1301

5201

5355

2177

1377

Solution: Using the same format as in Exercise 3.1:

a) 1001_8 = 001 0 0 0 000 001 TAD 1
1=TAD "page 0" "location 1"

The instruction is: "Add the number which is in location 1 on page zero to the accumulator."

b) 1301_8 = 001 0 1 1 000 001 TAD 1201
1=TAD "current page" "location 101"

The instruction is: "Add the number which is in location 101 on the current page to the accumulator."

c) 5201_8 = 101 0 1 0 000 001 JMP 1201
5=JMP "current page" "location 1"

The instruction is: "Jump to location 1 on the current page."

d) 5355_8 = 101 0 1 1 101 101 JMP 1355
5=JMP "current page" "location 155"

The instruction is: "Jump to location 155 on the current page."

e) 2177_8 = 010 0 0 1 111 111 ISZ 177
2=ISZ "page 0" "location 177"

The instruction is: "Increment the number in location 177 on page 0, and see if that number is zero or not."

f) 1377_8 = 001 0 1 1 111 111 TAD 1377

1=TAD "current page" "location 177"

The instruction is: "Add the number which is in location 177 on the current page to the accumulator."

QUESTION 3.1

Translate the following octal instruction codes. Assume that the current page is page 6.

1301

2354

2254

1020

3120

5377

QUESTION 3.2

Translate the following "English" instructions to their octal. Assume that the current page is page 7.

TAD 55 (page zero)

TAD 1655 (current page)

DCA 102 (page zero)

JMP 1777 (current page)

DCA 120 (page zero)

TAD 1620 (current page)

ISZ 1654 (current page)

ISZ 1754 (current page)

Table 3.1 listed the six memory reference instructions for the PDP-8 and gave their three-letter codes called mnemonics. The mnemonics present an easier way for the programmer to develop a set of instructions for the computer. Instead of looking at lists of octal numbers, the programmer can read these short code words and follow the flow of the instructions. Once he has written up the complete instruction set in mnemonics, he can then translate to octal (actually binary) which can then be entered into memory locations from the front panel, and execute this program. However, this hand translation of "letter codes" to the binary "machine language" instructions can become quite time consuming for very long programs. A program can be written to have the computer carry out this translation process. This program must be written in machine language and is called the assembler; it "assembles" the programmer's letter codes into the binary instructions which the computer then executes. The examples and exercises that will follow will be short enough so that the translation process can be carried out by hand.

The following exercises will now begin to use the Memory Reference Instructions in some simple programs, in order to demonstrate their use.

EXERCISE 3.3

Write a program (in mnemonics) to add two numbers in memory together and store the answer in a third memory location. Also code it into the octal (binary) instruction words.

Solution: The programmer cannot tell the PDP-8 to "Add 'A' to 'B' and call the sum 'C'" in one, or two instructions. He must first think out all the steps the machine will have to carry out to do this problem.

The PDP-8 uses the register called the accumulator (introduced in Chapter 2) for all transfers of data words to and from the arithmetic unit, and to do the

arithmetic operations. The "English" for this program would look like the following:

Step 1: "Clear out the accumulator in case there is something in it."

Step 2: "Add the first number 'A' to the accumulator."

Step 3: "Add the second number 'B' to the accumulator."

Step 4: "Return the answer 'C' to memory."

Step 5: "End of program."

The "addition" command is the memory reference instruction "TAD", or "Two's Complement Add." Two's complement means that negative integers are stored in the "Two's complement" form. This will be covered in a later example. The mnemonics for the program can now be written. Assume the number 'A' is in location 50_8 and the number 'B' is in location 51_8, and the sum will be stored in location 52_8. Begin the program at location 30_8 (note that all these locations are on page zero).

Octal Memory Location	Mnemonic	Meaning
30	CLA	Clear out the accumulator
31	TAD 50	Add the number in location 50_8 to the accumulator
32	TAD 51	Add the number in location 51_8 to the accumulator
33	DCA 52	Deposit the answer which is in the accumulator into location 52_8 and clear out the accumulator
34	HLT	End of program
50	Octal value of A	
51	Octal value of B	
52	Octal value of answer C	

Note that the TAD instruction adds a copy of the contents of the specified location to the accumulator, that is, the number 'A' is still in location 50_8 after the machine does "TAD 50".

Also, when the computer executed "DCA 52", the present value of the accumulator becomes the new contents of location 52_8, and the old contents are lost; then the new value of the accumulator is zero. (The instructions CLA and HLT are called microinstructions and will be covered in Chapter 5.)

The program can now be coded into octal (binary) instructions which the computer can understand; the codes for CLA and HLT are given:

Memory Location	Mnemonic	Octal Code
30	CLA	7600
31	TAD 50	1050
32	TAD 51	1051
33	DCA 52	3052
34	HLT	7402

The reader should now load the octal code of the program into the PDP-8, and run the program (using the techniques from Exercise 1.2), for the following sets of 'A' and 'B'. After each execution, examine the answer 'C' which is in the location 52_8, verifying that the machine did add the two octal numbers. Note that this addition has been performed in octal rather than our familiar decimal addition.

	Trial 1	Trial 2	Trial 3	Trial 4
A (=loc. 50)	0707	1111	1324	3666
B (=loc. 51)	0070	1111	0175	2666
C (=loc. 52)	0777	2222	1521	6554

DO NOT load the answers in location 52 before each trial!

Summary: The purpose of this exercise is to show how a program can be written in mnemonics instead of the 12-bit binary coding.

QUESTION 3.3

Take the program in Exercise 3.3 and translate to the octal code noting that now the program will begin at location 1000_8 and that 'A' is in location 1050_8, 'B' is in location 1051_8, and 'C' is in location 1052_8.

The next exercise will repeat Exercise 3.3, except that the concept of setting the page bit is emphasized.

EXERCISE 3.4

Repeat the problem from Exercise 3.3, except that now begin the program at location 400_8, and the number 'A' is in location 450_8, and 'B' is in location 451_8, and the answer 'C' will be placed in location 452_8. (All these locations are on page 2.)

Solution: The page the computer will be working on is not "page 0", but is the "current page". The page bit in the memory reference instructions used in the program will now be set to '1' (Bit #4). 'A' is to be in location 450_8, but this is the same as location 50_8, "current page". For example, the instruction "TAD 450" in the program will be coded as follows:

001	0	1	0	101	000	=	001	010	101	000
TAD		"current page"		loc. 50		=	1	2	5	0_8

(Note that for Exercise 3.3, "TAD 50" was coded as 1050_8.)

Following this pattern, the program will be coded as follows:

Memory Location	Mnemonic	Octal Code
400	CLA	7600
401	TAD 450	1250
402	TAD 451	1251
403	DCA 452	3252
404	HLT	7402
450	(octal value of A)	
451	(octal value of B)	
452	(octal value of C)	

Using the console, load this program code into the given locations and repeat trials 1 through 4 for Exercise 3.3, verifying that the results are the same. Note that if the programmer did not set the "page bit" to '1' in the memory reference instructions used in the program, the computer would have added the contents of location 50_8 and 51_8 on "page zero" which are the absolute locations 50_8 and 51_8.

QUESTION 3.4

Write the octal code for the same program as above, starting at location 400_8, except that now add the numbers 'A' and 'B' stored at absolute location 50_8 and 51_8 (on page zero) and store the result in absolute location 52_8.

The major advantage of using the computer is to perform the same operations many times. This is referred to as looping. The programmer can, at his discretion, execute various portions of his program many times without repeatedly writing the group of instructions for that particular part of the program.

Because the programmer can accomplish this with just a few instructions, this is one of the most powerful tools at his disposal.

There are two things that a programmer must know when creating a loop: (1) the portion of the program that he wants to be repeated, and (2) the number of times that portion of the program is to be repeated. The programmer keeps track of the number of times the loop is to be executed by means of a counter. This counter is usually a number set by the programmer and stored in memory.

For ease of programming, the PDP-8 has two instructions to perform the task of looping: the memory reference instructions ISZ and JMP. In the PDP-8, it is easier to compare a number of value against zero than to compare that number or value to some other constant. The ISZ command increments the specified memory location and compares the result to zero. Usually the counter is stored as its negative value (2's complement) so that when the ISZ is executed, the contents of the memory location holding the counter will approach zero. For example, if the programmer wanted to loop a particular part of his program five times, the octal number 7773_8 ($=-5_8$) would be stored in some memory location as the counter.

EXAMPLE 3.1

Obtain the negative value, in eight's complement form, of the octal numbers: 0005, 0060, 0525, 0001.

the octal number	0005_8	
its seven's complement	7772	
"add 1"	+ 1	
eight's complement	7773	(negative 5_8)

Note that finding the eight's complement of a number is the same as finding the two's complement of the equivalent binary number:

the binary number	000	000	000	101	
its one's complement	111	111	111	010	
"add 1"	+			1	
two's complement	111	111	111	011	= 7773_8

Similarly:

0060_8	0525_8	0001_8
7717	7252	7776
+ 1	+ 1	+ 1
7720 (-60_8)	7253 (-525_8)	7777 (-1_8)

Note that most assemblers (discussed later) allow entry of a negative number directly. For example, octal 7771 can be specified as -7.

The JMP command is then used with the ISZ command allowing the programmer the ability to transfer control back to the beginning of the loop. These ideas are covered in the following example.

EXAMPLE 3.2

The following program will add the octal number 1010_8 stored in location 450_8 to the accumulator five times, and will store the result in location 451_8. Note that negative five will be stored as the counter in location 452_8.

Octal Memory Location	Mnemonic	Meaning
400	CLA	"Clear the accumulator"
401	TAD 450	"Add 1010 to the accumulator"
402	ISZ 452	"Increment the contents of location 452_8 and compare the result to zero" (If result = 0, skip next instruction; if result ≠ 0, execute next instruction)
403	JMP .-2	"Jump to beginning of loop (location 401_8)"

Octal Memory Location	Mnemonic	Meaning
404	DCA 451	"Deposit the result in location 451_8"
405	HLT	"Stop execution"
450	1010	The number to be added
451	0000	A location to store the result
452	7773	"negative 5"

Note that the (.) refers to the present location of the program counter, that is, location 403_8. Thus (.-2) refers to location 401_8, which is the beginning of the loop. Then, "JMP .-2" transfers control to the instruction TAD 450.

The octal code for the program can be written as follows:

Octal Memory Location	Mnemonic	Octal Code
400	CLA	7600
401	TAD 450	1250
402	ISZ 452	2252
403	JMP .-2	5201
404	DCA 451	3251
405	HLT	7402
450	1010	1010
451	0000	0000
452	7773	7773

Notice that location 452, which initially contains a -5 would have to be re-initialized each time the program is to run since it ends up with a value of zero at the end of each run.

Solution: The following table lists the contents of the accumulator, location 452_8 which holds the counter, and location 451_8 where the result will be stored, on each successive pass through the loop.

	Accumulator	Loc.451_8	Loc. 452_8
Initially	0000	0000	7773
1st pass through the loop	1010	0000	7774
2nd pass	2020	0000	7775
3rd pass	3030	0000	7776
4th pass	4040	0000	7777
5th pass	0000	5050	0000

EXERCISE 3.5

Write a program that first clears the accumulator, then keeps adding '1' to the accumulator. Write a delay loop to slow down the computer so that one can "watch" the computer count by viewing the display labeled ACCUMULATOR on the front panel. (The octal code for the instruction Increment the ACcumulator is 7001_8 and the mnemonic is IAC; microinstructions will be covered in Chapter 5.)

Solution: A simple program to continuously increment the accumulator is written as:

Memory Location	Mnemonic	Octal Code
400	CLA	7600
401	IAC	7001
402	JMP .-1	5201

If this program is run on the computer, the accumulator will be incremented so fast that the display will appear blurred. This is because the execution time for the JMP instruction on the PDP-8/I minicomputer is 1.5 microseconds (a microsecond being equal to 10^{-6} seconds), which is too fast for the programmer to see on the display. (In the PDP-8/S JMP takes 28 microseconds.)

To slow down the time between each time the accumulator is incremented, a loop can be inserted within the program whose only purpose is to slow down the time for execution. This type of looping is called <u>delay looping</u>. Thus execution can be slowed down so that the programmer can "watch" the computer count on the display. A simple delay loop is added to the previous program:

Memory Location	Mnemonic	Octal Code
400	CLA	7600
401	IAC	7001
402	ISZ 405	2205
403	JMP .-1	5202
404	JMP .-3	5201
405	0000	0000

Since the execution time for the ISZ instruction is 3.0 microseconds (54 microseconds), the loop:

ISZ 405

JMP .-1

will take 4.5 microseconds altogether. This loop will be executed 4096_{10} (10000_8) times before the accumulator will again be incremented. Thus a delay of 18.432 milliseconds (335 milliseconds) will be present between each time the accumulator is incremented. Run this program on the computer and note that now

part of the accumulator will be blurred, and the other part will show the accumulator "counting" on the display.

We should expect this range in the speed of blinks. If we were asked to list all the integers in sequence from 0 to 999, or any positive integer, the left most digit would change the slowest while the units digit would change the fastest. In a computer the left most bit is called the Most Significant Bit, MSB, while the right most is the least, or LSB. We should therefore expect the MSB to change much slower than the LSB, which it does.

To further slow down the execution, an outer delay loop can be added, allowing the programmer to vary the speed at which the accumulator will count. Run the following program on the computer. (Note that now the delay loop will be approximately 8 x 18.432 milliseconds = .147 seconds.) The calculation for the PDP-8/S becomes 8(0.335872 seconds) = 2.686976 seconds.

Memory Location	Mnemonic	Octal Code
600	CLA	7600
601	TAD DELAY	1215
602	DCA 614	3214
603	TAD 616	1216
604	ISZ 613	2213
605	JMP .-1	5204
606	ISZ 614	2214
607	JMP .-3	5204
610	IAC	7001
611	DCA 616	3216
612	JMP .-11	5201

Memory Location	Mnemonic	Octal Code
613	0000	0000
614	0000	0000
615	DELAY, 7770	7770
616	0000	0000

To vary the speed of execution, the value of location 615 can be varied (remember to reinitialize location 616 to zero each time). Notice also the use of a label or name to reference the delay count rather than using its address.

EXERCISE 3.6

Set up the program for a delay of 10 seconds.

Solution: The loop at locations 604, 605 takes 18.432 seconds for the PDP-8/I (0.335872 seconds for the PDP-8/S). Therefore the outer loop must be run:

PDP-8/I 10/0.018432 times = 543_{10} times

PDP-8/S 10/0.335872 times = 30_{10} times

PDP-8/I Initiallize location 615 to $-543_{10} = -1037_8 = 6741$

PDP-8/S Initiallize location 615 to $-30_{10} = -36_8 = 7742$

ANSWERS TO CHAPTER 3 QUESTIONS

Question 3.1 a) 1301_8 = 001 0 1 1 000 001

"TAD" "current page" "location 101"

1301_8 = TAD 101 (current page) (TAD 1501)

b) 2354_8 = 010 011 101 100 = ISZ 154 (current page) (ISZ 1554)

c) 2254_8 = 010 010 101 100 = ISZ 54 (current page) (ISZ 1454)

d) 1010 = 001 000 010 000 = TAD 20 (page zero) (TAD 20)

e) 3120 = 011 001 010 000 = DCA 120 (current page) (DCA 120)

f) 5377 = 101 011 111 111 = JMP 177 (current page) (JMP 1577)

Question 3.2

TAD 55 = 001 0 0 0 101 101
"TAD" "page zero" "location 55"

= 001 000 101 101

= 1055_8

TAD 1655 = 001 0 1 0 101 101
"TAD" "current page" "location 55"

= 001 010 101 101

= 1255_8

Similarly, DCA 102 = 011 001 000 101

= 3102_8

JMP 1777 = 101 011 111 111

= 5377_8

DCA 120 = 011 001 010 000

= 3120_8

TAD 1620 = 001 010 010 000

= 1220_8

ISZ 1654 = 010 010 101 100

= 2254_8

ISZ 1754 = 010 011 101 100

= 2354_8

Question 3.3

Location	Mnemonic	Octal Code
1000	CLA	7600
1001	TAD 1050	1250
1002	TAD 1051	1251
1003	DCA 1052	3252
1004	HLT	7402

Location	Mnemonic
1050	Octal value of A
1051	Octal value of B
1052	Octal value of C

a) These locations are on memory page 4.

b) Compare the octal code for the instructions TAD 50, TAD 51, and DCA 52 above, with those same instructions as they are cited in Exercise 3.3.

Question 3.4

Location	Mnemonic	Octal Code
400	CLA	7600
401	TAD 50	1050
402	TAD 51	1051
403	DCA 52	3052
404	HLT	7402
50	Octal value of A	
51	Octal value of B	
52	Octal value of C	

CHAPTER 4

Indirect Addressing

It was previously mentioned that bit #3 of a memory reference instruction on the PDP-8 is called the Address Mode Bit. In all preceding examples, this bit was set to '0' in the memory reference instructions; this value of '0' declares the Address Mode to be "Direct Addressing". The address contained in the operand of the memory reference instruction is the location of the desired information to be operated on. If bit #3 is now set to '1', the address mode becomes Indirect Addressing. Now, the operand of the memory reference instruction holds an address, but it serves as a "pointer". It points to a memory location and that memory location contains the information the instruction is to operate on.

There are three main reasons why we may need indirect addressing.

(1) The most important is that there are not enough bit positions left in a Memory Reference Instruction to address any word in memory. Remember that we can only address "page 0" or the "current page". If, however, we had a full 12 bit word available to use as an address, then we could refer to any word in core. Since $2^{12}-1=4095$ and there are 4096 words in core (call the first word's address 0000), the 12 bit positions allow reference to any word. This is exactly what indirect addressing allows us to do.

(2) Indirect addressing must be used when using subroutines. This will be explained shortly.

(3) There may be times when we want to pick words sequentially from some list. We can fetch these very simply by incrementing a word which contains the address of the first word in the list and using indirect addressing.

EXAMPLE 4.1

If memory location 400_8 contains the instruction TAD I 450, where the "I" is the mnemonic symbol for "Indirect Addressing", (octal code = 1650), what number would be added to the accumulator when this instruction is executed?

Solution: The instruction would not add the contents of 450_8 as in a direct addressing instruction. The computer goes to location 450_8, reads the contents of location 450_8 as an address, and then goes to this new address and adds the contents of this new location to the accumulator. If the contents of memory location 450_8 is 2300_8, and memory location 2300_8 contained 1111, the computer would add the number 1111 (i.e., the contents of location 2300) to the accumulator, and not the number 2300_8.

EXAMPLE 4.2

If location 600_8 contains the instruction "JMP I 643, current page" (octal code = 5643) and location 643_8 contains the instruction "TAD I 745, current page" (octal code = 1745) and location 1745_8 contains the instruction "HLT" (octal code = 7402), what is the next instruction executed after the instruction "JMP I 643"?

Solution: The next instruction executed would be the "HLT" in memory location 1745. "JMP I 643" looks at the contents of location 643 as an address ("I" means indirect addressing). The contents of location 643 is not interpreted as an instruction, but rather as the address for the computer to "jump" to. Since location 643 contains the number 1745_8, the next instruction executed is the "HLT", which is the contents of 1745:

Location	Mnemonics	Octal Code
600	JMP I 643 (Current Page)	5643
.	.	.
.	.	.
.	.	.
643	TAD I 745 (Current Page)	1745
.	.	.
.	.	.
.	.	.
1745	HLT	7402

EXERCISE 4.1

What is the octal code for the following indirect addressed instructions: TAD I 130 ('page zero'), DCA I 250 ('current page')?

Solution:

a) TAD I 130 --- 001 1 0 1 011 000

"TAD" "indirect address" "page zero" 130_8

then, TAD I 130 (page zero) = 001 101 011 000

$= 1530_8$

b) DCA I 250 --- 011 1 1 0 101 000

"DCA" "indirect address" "current page" 50_8

then, DCA I 250 (current page) = 011 110 101 000

$= 3650_8$

QUESTION 4.1

What is the octal code for the following instructions: TAD 43, DCA I 500, ISZ I 413, JMP I 213, JMP I 20, DCA 100, TAD I 43?

EXERCISE 4.2

Go back to Exercise 2.1, and translate the octal code of the program into the mnemonics, and explain each instruction. Note that the program uses both indirect and direct addressing, and that some address references are by name or label rather than the address.

Solution:

Memory Location	Octal Code	Mnemonic	Explanation
400	7300	CLA CLL	Clear the accumulator (acc.) and the link bit.
401	1214	TAD MINUS5	Move "counter" in location 414 to the accumulator.
402	3215	DCA COUNTR	Move "counter" from acc. to location 415.
403	1616	TAD I 416	Move the contents of the address which is contained in loc. 416 to the acc.
404	7402	HLT	A "halt" to see the data transfer above.
405	3617	DCA I 417	Move it from the acc. to the address found in loc. 417 and clear the acc.
406	2216	ISZ 416	Increment the pointer which is in loc. 416. It now points to the next "source".
407	2217	ISZ 417	Increment the pointer which is in loc. 417. It now points to the next "destination".
410	2215	ISZ COUNTR	Increment the courter in loc. 415. Is it now zero?
411	5203	JMP 403	No, so jump to loc. 403, to move the next number.
412	7402	HLT	Yes, the counter in 415 is now zero, so halt the execution of the program: all 5 numbers have been moved.
413	5212	JMP 412	Jump to location 412.

Memory Location	Octal Code	Mnemonic	Explanation
414	7773	MINUS5, -5	The 8's complement form of negative 5 (-5); the "counter".
415	0000	COUNTR, 0	A location to store the "counter" and to add one to it every time a number is moved. When the counter reaches zero, all 5 numbers will have been moved.
416	2000	the address 2000	The address of the 1st of the 5 numbers to be moved; the "source".
417	0050	the address 50	The address of the location to move the 1st word to; the "destination".

EXERCISE 4.3

If the computer is at address 405_8 (page 2), how can it be programmed to JUMP to address 620 (page 3)?

Solution: This can only be done through the use of indirect addressing.

Memory Location	Octal Code	Mnemonic	Explanation
404		CLA	
405		JMP I .+1	JUMP indirectly to the location given in the next (.+1) memory location. Final result is to JUMP to location 0620.
406	0620		
.	.	.	
.	.	.	
.	.	.	
620		DCA .-1	Put contents of location 617 into the accumulator.

Note that in this solution, almost any instructions could have been used in place of the CLA and DCA. The main point is that the JMP I .+1 caused a jump to a location on another page. This procedure is not necessary if we are simply at the end of a page and want to go to the first location of the next page because this automatic single address advance is done by the Program Counter which is a 12 bit register in the computer.

Summary: Indirect addressing must be used when accessing a location not in the "current page", except when --

(1) the location to be assessed is in "page 0."

(2) the instruction being executed is the last word of the "current page" and the next instruction is the first word of the next page.

QUESTION 4.2

Suppose a program extends over several pages of memory, for example pages $1\text{-}10_8$. How would data on page 11_8 be accessed? How else could the data be stored to ease the access problem?

QUESTION 4.3

Write and execute a program to add seven octal numbers stored in locations 1000_8 to 1006_8 and store the result in location 640_8. Start the program at location 200_8. Use indirect addressing and looping in the program.

SUBROUTINES

In computer programming, the situation often arises where a certain group of operations will have to be carried out several times with, perhaps, different data. Instead of the programmer repeatedly writing out the instructions for this group each time it is needed, it would be more convenient (for the programmer) to only write the instruction group one time, and to have the computer "fetch" it any time it is needed. A group of instructions used in this manner is a sub-program called a SUBROUTINE. Subroutines are aside from the main program. Whenever they are needed, the main program "stops" what it is doing, transfers control to the sequence of operations in the subroutine, executes them and then picks up at the point in the main program immediately after the "call" for the subroutine. Each time a subroutine is called, the computer does this branching, execution of the subroutine, and then returns to the main program at the branch point.

If, for example, the main program had to add some two numbers together, instead of writing the sequence of addition instructions each time they were needed, the programmer could write a subroutine to do the addition. He would then have the main program "send" the two numbers to be added to this subroutine, which would carry out the addition and then return to the main program. The programmer has then saved himself some code writing, and also conserved memory space. In order to implement subroutines on the PDP-8, the programmer must use indirect addressing.

EXAMPLE 4.3

Write a program that calls a subroutine to add two numbers 'A' and 'B'. The result will be stored in 'C'.

Solution: The memory reference instruction to call a subroutine is the JMS instruction (JuMp to Subroutine). The JMS does two things:

1. The address of the next instruction after a JMS is stored in the first location of the subroutine. (Note then that the first memory location in a subroutine should not contain any instruction or data. It will be strictly a storage space.)
2. The address which is in the operand of the JMS instruction is increased by 1 and placed in the program counter. Therefore the computer is now ready to get the first instruction to be executed in the subroutine.

Indirect addressing is the means to return from the subroutine to the main program. The last instruction carried out in any subroutine is an indirect-addressed JMP. It jumps to the address which is stored in the first location of the subroutine. In other words, it jumps back to the next instruction after the point in the main program where it had left off to carry out the subroutine.

Octal Memory Location	Mnemonic	Octal Code	Meaning
400	CLA	7600	Clear the accumulator.
401	JMS 450	4250	Jump to the addition subroutine which starts in Loc. 50 current page ($=450_8$).
402	DCA 406	3206	After returning from the addition subroutine, store the sum in 'C' and clear the accumulator.
403	HLT	7402	Stop the program, addition completed.
404	(the value of 'A')	-	
405	(the value of 'B')	-	
406	(the location to store the sum 'C')	-	
450	0	0000	The addition subroutine; location 450 is the storage area to place the return address.
451	TAD 404	1204	Get the value of 'A' and put it into the accumulator.
452	TAD 405	1205	Get the value of 'B' and add it to the value of 'A' in the accumulator.
453	JMP I 450	5650	Get the return from subroutine - address in Loc. 450 and jump to it.

Remember that a JMS is a memory reference instruction and so we still have the limits of directly addressing only locations in page zero or the current page. If the subroutine is in another page, we must use indirect addressing to jump to it, therefore, when possible it may be convenient to put subroutines into page zero so that they may be addressed directly.

When the program executes the instruction in 401_8 (JMS 450), it will first store in location 450, 401 + 1 = 402, the next instruction to be executed when returning from the subroutine.

Then the computer gets the next instruction to be carried out by adding 1 to the address in the JMS instruction, i.e., 450 + 1 = 451. It then executes the instruction is 451 (TAD 404), then in 452 (TAD 405), and then it reaches the instruction to leave the subroutine (JMP I 450). It gets the address contained in location 450 (i.e., 402), and jumps to that location. Therefore, it jumps out of the subroutine and back to the main program at location 402. It deposits the results of the addition, and then stops (DCA 406, HLT).

Note again that the computer does <u>not</u> execute the instruction in the address of the JMS instruction. It stores in that location the address of the instruction <u>after</u> the JMS.

Note that indirect addressing is used to return from the subroutine to the main program.

The general structure of an m-instruction subroutine is:

Octal Memory Location	Contents
X	0
.	.
.	.
.	.
	(m instructions)
.	.
.	.
.	.
X+(m+1)	JMP I X

EXERCISE 4.4

Write a program which will call a subroutine that will find the numbers which are divisible by three (3) from the numbers 1 to 27_{10}, and then store these numbers starting in location 1000_8. Start the program in location 600_8.

Solution: To test a number A to see whether it is divisible by three, repeatedly subtract three from A until the result is either '0', which indicates the number is divisible by three, or the result is less than '0', indicating that the number is not divisible by three. The mnemonics for the program and subroutine and the corresponding octal codes are given as follows: (CLA, CIA, IAC, SPA, SNA, and SZA are microinstructions - see Chapter 5.)

Location	Mnemonic	Code	Meaning
600	CLA	7600	Clear accumulator
601	TAD K27	1215	Find the negative value
602	CIA	7041	of 27_{10} and use this
603	DCA COUNTR	3216	as the counter.
604, LOOP,	TAD TESTD	1221	Calculates next number
605	IAC	7001	to be tested and store
606	DCA TESTD	3221	in location 621.
607	TAD TESTD	1221	Add number to be tested to the accumulator.
610	JMS 650	4250	Jump (current page) to subroutine starting in location 650.
611	CLA	7600	Clear accumulator
612	ISZ COUNTR	2216	Increment counter and halt if zero.
613	JMP LOOP	5204	Return to beginning of loop
614	HLT	7402	Halt execution
615 K27	0033	0033	Constant $27_{10} = 33_8$

Location	Mnemonic	Code	Meaning
616	COUNTR, 0000	0000	Location to store counter
617	MINUS3, 7775	7775	8's complement of '3', or (-3)
620	1000	1000	Starting location of where numbers divisible by three are to be stored.
621	TESTD, 0000	0000	Contains number being tested
(The Subroutine)			
650	SUBRTN, 0000	0000	Storage location for return address.
651	TAD MINUS3	1217	Subtract '3' from number being tested, by adding (-3).
652	SPA SNA	7550	This microinstruction (see Chapter 5) will skip the next instruction if the contents of the accumulator is greater than '0'.
653	JMP 655	5255	Jump to the test for a '0' accumulator.
654	JMP 651	5251	Repeat loop until result is less than, or equal to, zero.
655	SZA	7440	This microinstruction will skip the next instruction if the contents of the accumulator is zero.
656	JMP I SUBRTN	5650	Jump back to main program to get next number.
657	TAD TESTD	1221	If divisible by three, store
660	DCA I 620	3620	in location starting at 1000_8.
661	ISZ 620	2220	Get location for next number to be stored
662	JMP 656	5256	Jump back to location 656 to exit subroutine.

Note again how the use of names or labels for certain addresses makes the program much easier to follow.

Autoindexing

When using indirect addressing, care must be taken when using memory locations 0010_8 through 0017_8. When one of these locations is addressed indirectly, the content of that location is incremented by one, rewritten into the same location, and used as the effective address of the current instruction. These registers can be used to gain access to a table of data without using the ISZ instruction to step through the table.

EXERCISE 4.5

Write a program using autoincrementing to clear locations 2000_8 to 2777_8.

Solution:

Location		Instruction	
		*10	
0010	INDEX,	0	/AUTOINDEX REGISTER
		*200	
0200	CLEAR,	CLA CLL	/CLEAR ACC AND LINK
0201		TAD CONST	/GET PERMANENT COUNTER
0202		DCA CONST	/STORE IT FOR USE
0203		TAD TTABLE	/GET ADDRESS OF LOC ONE
			/BEFORE TABLE BEGINS
0204		DCA INDEX	/STORE IT IN AUTOINDEX REG
0205		DCA I INDEX	/USE AUTOINCREMENTING
0206		ISZ COUNT	/UPDATE COUNT, SKIP IF ZERO
0207		JMP .-2	/LOOP BACK
0210		HLT	/HALT, FINISHED
0211	CONST,	-1000	/COUNTER
0212	COUNT,	0	/TEMPORARY COUNTER LOCATION

Location		Instruction	
0213	TTABLE,	TABLE-1	/TABLE ADDRESS -1
		*2000	
2000	TABLE,	(table values)	/TABLE STARTS HERE

.

.

.

$

In the above program, the symbols * and $ were used to tell the assembler program special things. These will be explained in a later chapter.

EXERCISE 4.6

Use autoindexing to search all of core for an occurrence of the number 1234_8.

Solution:

Location		Instruction	
		*0	
0000	NUMBER,	1234	/THIS IS THE NUMBER
		*10	
0010	ENTRY,	0	/USED FOR AUTOINDEXING
		*200	
0200	BEGIN,	CLA CLL	/CLEAR ACC AND LINK
0201		TAD NUMBER	/GET NUMBER
0202		CIA	/GET ITS NEGATIVE
0203		DCA COMPARE	/STORE IN COMPARE
0204		DCA ENTRY	/INITIALIZE AUTOINDEX REG
0205	REPEAT,	CLA CLL	/CLEAR ACC AND LINK

Location		Instruction	
0206		TAD I ENTRY	/GET FIRST VALUE
0207		TAD COMPARE	/ADD A -1234
0210		SZA	/SKIP IF 1234 IS FOUND
0211		JMP REPEAT	/RETURN IF 1234 IS NOT FOUND
0212		TAD ENTRY	/PUT ADDRESS OF 1234 IN ACC
0213		HLT	/STOP
0214	COMPARE,	0	/TEMPORARY LOCATION
		$	

Note that the program stops with the address of the location in which 1234 was found in the accumulator.

ANSWERS TO CHAPTER 4 QUESTIONS

Question 4.1

TAD 43 = 001 0 0 0 100 011

"TAD" "direct address" "page zero" "location 40_8"

= 001 000 100 011 = 1043_8

DCA I 500 = 011 1 1 1 000 000

"DCA" "I" "current page" 100_8

= 011 111 000 000 = 3700

ISZ I 413 = 010 1 1 0 001 011

= 010 110 001 011

= 2613_8

JMP I 213 = 101 1 1 0 001 011

= 101 110 001 011

= 5613_8

DCA 100 = 011 0 0 1 000 000

= 011 001 000 000

= 3100_8

TAD I 43 = 001 1 0 0 100 011

= 001 100 100 011

= 1443_8

Question 4.2

Indirect addressing would have to be used to access them because when the computer is executing instructions on some given memory page, it can directly access constants or variables only from the "current page" or from "page zero". If a program extends beyond one memory page in length, and, all the instructions need to access the same data directly, the data must be on a page which can be accessed by all pages, i.e., page zero. For example, memory pages 4 through 10 would not be able to directly access data "defined" on page 11, but they could access the data on page zero. Therefore if direct access were important it would be necessary to locate all the data for the long program on page zero. Of course, if the penalty of an extra word and the slower access time of indirect addressing are not critical, then the data may be stored on any page and accessed indirectly.

Note that when indirectly addressing data not on page 0, it is convenient to put the pointer to the data on page 0, so that the pointer may be used by any page without being repeated on that page.

Question 4.3

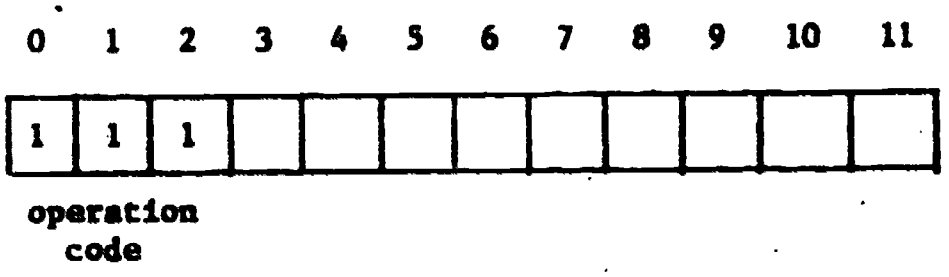
Location	Mnemonics	Octal Contents	
200_8	CLA	7600	Clear the acc.
201	TAD 211	1211	Set the counter in loc. 242 to negative 7 (-7)
202	DCA 212	3212	

Location	Mnemonics	Octal Contents	
203	TAD I 213	1613	Add the number, whose address is in loc. 213 to the acc.
204	ISZ 213	2213	Increment the pointer in loc. 213; it now points to the next number to be added.
205	ISZ 212	2212	Increment the counter in loc. 212; is it = zero?
206	JMP 203	5203	No; jump back and add the next number
207	DCA I 214	3614	Yes; deposit the sum in the location whose address is in loc. 214.
210	HLT	7402	Stop the program; finished
211	the number 7771	7771	The no. (-7) in 8's complement form.
212	0000	0000	The counter location
213	the address 1000	1000	The address of the 1st no. to be added.
214	the address 640	0640	The address of the location to store the sum.
1000		(Store the 7 octal numbers to be added in these locations.)	
1001			
1002			
1003			
1004			
1005			
1006			

CHAPTER 5

Microinstructions

Some of the previous exercises used several instructions that were not listed in Table 3.1, such as CLA, "clear the accumulator", and HLT, "halt". These instructions belong to a second class of PDP-8 instructions called Micro-instructions. They are called microinstructions because in a 12 bit instruction word single bits are each interpreted as an instruction, whereas in the Memory Reference Instructions, the entire 12-bit word is one instruction to the computer. Therefore, if the programmer writes some valid combination of bits set to '1' in a microinstruction, each '1' will be translated into an operation. The operations are determined by the positions of the '1' bits in the word (i.e., which bits are set). Microinstructions have a 3 bit operation code, just as the memory reference instructions do. To specify that an instruction word is a microinstruction, the first three bits are set to 111 ($=7_8$):

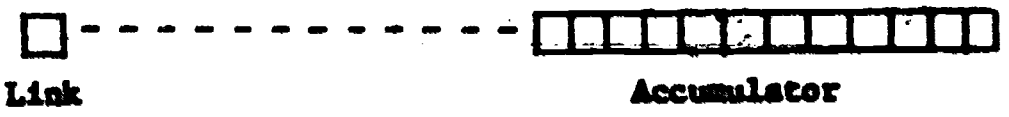


The general concept of having single particular bits in an n-bit instruction word representing different commands to the computer's control unit is called MICROPROGRAMMING.

There are two groups of microinstructions for the PDP-8:

Group 1	Manipulate the contents of the accumulator and the link
Group 2	Primarily for testing operations

The LINK is a register, like the accumulator, except that it is only one bit long. It acts as what could be called an "overflow" register for the accumulator. If, for example, an addition operation resulted in a 'carry' beyond 12 bits, the link would "catch" the overflow:



Link Accumulator

The microinstructions give the programmer a way of using the link to his advantage in some problem situation, such as the mentioned additon overflow.

The Group 1 Microinstructions:

Bits #0 through #2 are set to '1' as the microinstruction operation code. But unlike the memory reference instructions, the remaining nine bits do not specify an address in memroy. Bit #3 is set to '0' to indicate "Group 1". Then:

1. When bit #4 is set to '1', the instruction is Clear the Accumulator; the mnemonic is CLA. The accumulator is set to all zeros.

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

2. When bit #5 is set to '1' the instruction is Clear the Link; the mnemonic is CLL. The link bit is set to zero.

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

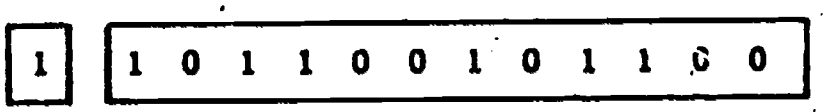
3. When bit #6 is set to '1', the instruction is Complement the Accumulator; the mnemonic is CMA. Any bit of the accumulator that was a '1' will be set to '0' and any bit that was a '0' will be set to '1'.

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

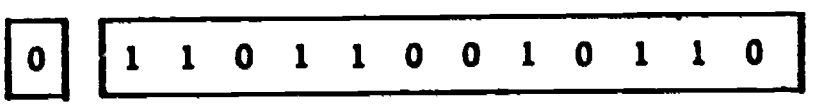
4. When bit #7 is set to '1', the instruction is Complement the Link; the mnemonic is CML. If the Link was a '1', it will be set to '0'; if it was a '0' it will be set to '1'.

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

5. When bit #8 is set to '1' and bit #10 is set to '0', the instruction is Rotate the Accumulator and Link Right; the mnemonic is RAR. A loop is formed between the accumulator and the link and all bits in the loop are shifted one position to the right. Example:



(Before "RAR)



(After "RAR)

6. When bit #8 is set to '1' and bit #10 is set to '1', the instruction is Rotate Accumulator and Link to the Right Twice; the mnemonic is RTR. The result is the same as doing "RAR" two times in a row.

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

7. When bit #9 is set to '1' and bit #10 is set to '0', the instruction is Rotate the Accumulator and Link Left; the mnemonic is RAL. The result is the same as an RAR, except that the bits of the accumulator and link are shifted one position to the left.

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

8. When bit #9 is set to '1' and bit #10 is set to '1', the instruction is Rotate the Accumulator and Link Left Twice; the mnemonic is RTL. The instruction is the same as doing RAL two times in a row.

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

9. When bit #11 is set to '1', the instruction is Increment the Accumulator; the mnemonic is IAC. The contents are increased by 1.

1 1 1 0 0 0 0 0 0 0 1

Table 5.1 below summarized the Group 1 microinstructions:

TABLE 5.1

Mnemonic	Octal Code	Meaning
CLA	7200	CLear Accumulator
CLL	7100	CLear Link
CMA	7040	CoMplement Accumulator
CML	7020	CoMplement Link
RAR	7010	Rotate Accumulator (and Link) Right
RTR	7012	Rotate Twice Right
RAL	7004	Rotate Accumulator (and Link) Left
RTL	7006	Rotate Twice Left
IAC	7001	Increment ACcumulator

There is an advantage to each bit in a microinstruction being a particular command. The programmer can form certain valid combinations of these instructions, i.e., he can set more than one bit to '1' in a given microinstruction word. One can then think of each of these valid combinations as a very small, incomplete "program".

EXAMPLE 5.1

Using Group 1 microinstructions, what is the binary (octal) coding of the instruction "Clear the accumulator and clear the link"? (The mnemonic is written CLA CLL).

Solution: The operation code is 111 ($=7_8$); for "Group 1", bit #3 is set to '0'. To command a CLA, bit #4 is set; to command a CLL, bit #5 is set:

CLA CLL =	111	0	1	1	000	000
	operation code	Group 1	CLA	CLL		

then, CLA CLL = 111 011 000 000 = 7300_8

EXAMPLE 5.2

What is the microinstruction coding for "Rotate the accumulator and link three places to the right"?

Solution: To rotate the accumulator and link once to the right the instruction would be:

bit #	0	1	2	3	4	5	6	7	8	9	10	11	
	1	1	1	0	0	0	0	0	1	0	0		= 7010_8

To rotate the accumulator and link two more places to the right, the instruction would be:

bit #	0	1	2	3	4	5	6	7	8	9	10	11	
	1	1	1	0	0	0	0	0	1	0	1	0	= 7012_8

Note that bit #10 would have to have the value of '1' and '0' at the same time in order to write the command "Rotate 3 places right" in one instruction word. Therefore this instruction is impossible to write in one 12-bit microinstruction word, and the correct code would be two words long:

RAR . . . . 7010_8

RTR . . . . 7012_8

The Group 2 Microinstructions:

In the PDP-8, octal numbers take on the range of 0000_8 to 7777_8. Both positive and negative numbers are represented within this range of octal numbers. The range of the positive and the negative octal numbers are as follows

(remember that negative octal numbers are obtained by taking the 8's complement of the positive octal numbers):

Positive	0000_8 to 3777_8	(0_{10} to 2047_{10})
Negative	7777_8 to 4000_8	((-1_{10}) to (-2048_{10}))

Note that the main difference between positive and negative octal numbers is that bit #0 of the negative numbers is set to a '1', and that bit #0 of the positive numbers is set to a '0'. Certain tests are provided for handling negative as well as positive numbers and are covered by the "Group 2" microinstructions.

The Group 2 microinstructions give the programmer the ability to make certain tests on the accumulator and also the link and to make a decision based on the results of the particular test. Whether the result of the test produces a "true" or "false" condition, will determine whether the <u>next</u> instruction following the microinstruction will be executed or will be skipped. This allows the programmer to branch control to other parts of the program depending on whether the accumulator is positive, negative, zero, or some other combination of these; or if the link is a '0' or '1'.

Bits #0 through #2 are set to '1' as the microinstruction operand code. Bit #3 is set to a '1' and bit #11 is set to a '0' to specify "Group 2". Then:

1. When bit #4 is set to a '1', the instruction is <u>Clear the Accumulator</u>; the mnemonic is <u>CLA</u>. The accumulator is set to all zeros.

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

2. When bit #5 is set to a '1' and bit #8 is set to a '0', the instruction is <u>Skip on Minus Accumulator</u>; the mnemonic is <u>SMA</u>. If bit #0 of the accumulator is a '1', which indicates a negative number, the next instruction will be skipped; otherwise, the next instruction will be executed.

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

3. When bit #6 is set to a '1' and bit #8 is set to a '0', the instruction is Skip on Zero Accumulator; the mnemonic is SZA. If the contents of the accumulator are zero, the next instruction will be skipped.

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

4. When bit #7 is set to a '1' and bit #8 is set to a '0', the instruction is Skip on Nonzero Link; the mnemonic is SNL. The link bit can have either one of two values, a '0' or a '1'. If the link bit is a '1', the next instruction will be skipped.

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

5. When bit #5 is set to a '1' and bit #8 is set to a '1' the instruction is Skip on Positive Accumulator; the mnemonic is SPA. If bit #0 of the accumulator is a '0' which indicates a positive number, the next instruction will be skipped.

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

6. When bit #6 is set to a '1' and bit #8 is set to a '1', the instruction is Skip on Nonzero Accumulator; the mnemonic is SNA. If any bit in the accumulator has the value of '1', the next instruction will be skipped.

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

7. When bit #7 is set to a '1' and bit #8 is set to a '1', the instruction is Skip on Zero Link; the mnemonic is SZL. If the link bit is '0', the next instruction will be skipped.

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

8. When bit #8 is set to a '1' and bits #5, #6, and #7 are all '0', the instruction will perform an Unconditional Skip; the mnemonic is SKP. A possible use for this instruction might be that the programmer may

decide that the instruction following the microinstruction is not to be performed any more, and instead of rewriting the program, a SKP instruction can be substituted:

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

9. When bit #9 is set to a '1', the instruction will perform an Inclusive OR of the Switch Register with the Accumulator; the mnemonic is OSR. The original contents of the accumulator will be replaced with the result of the OSR instruction.

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

Example:

001	100	101	101	(Accumulator)
101	001	100	110	(Switch Register Setting)
101	101	101	111	(After OSR. Result is in Acc.)

10. When bit #10 is set to a '1', the instruction will perform a Halt; the mnemonic is HLT. This instruction will actually halt the current execution of the program, and can be inserted anywhere within the program to signal an end to the program. Note that HLT commands may also be inserted anywhere in the program to provide an aid in "debugging", i.e., finding errors in the program during execution. (Show that the octal code for a HLT is 7402_8 as was used in the previous exercises).

Table 5.2 below summarizes Group 2 microinstructions:

TABLE 5.2

Mnemonic	Code	
CLA	7600	CLear the Accumulator
SMA	7500	Skip on Minus Accumulator
SZA	7440	Skip on Zero Accumulator

Mnemonic	Code	
SNL	7420	Skip on Nonzero Link
SPA	7510	Skip on Positive Accumulator
SZL	7430	Skip on Zero Link
SKP	7410	SKiP unconditionally
OSR	7404	inclusive OR, Switch Register with accumulator
HLT	7402	HaLT

In the following two exercises, take the time to verify that the given octal codes are correct, and then work through the programs by hand and then verify the results by running the programs on the PDP-8.

EXERCISE 5.1

Determine the contents of locations 427_8, 430, and 431 after execution of the following program:

Location	Mnemonic	Code
400	CLA CLL	7300
401	TAD 424	1224
402	RAL	7004
403	RTL	7006
404	SMA	7500
405	JMP .+2	5207
406	DCA 427	3227
407	SNL	7420
410	JMP .+6	5216
411	TAD 425	1225
412	RAR	7010
413	DCA 430	3230

Location	Mnemonic	Code
414	SZL	7430
415	HLT	7402
416	TAD 426	1226
417	CMA	7040
420	IAC	7001
421	DCA 431	3231
422	CML	7020
423	JMP .-7	5214
424	1725	1725
425	4266	4266
426	0015	0015
427	0000	0000
430	0000	0000
431	0000	0000

Remember that:

1. Rotate the accumulator and the link when performing the rotate instructions.
2. The instruction DCA 27 in location 406_8 will deposit the contents of the accumulator in location 427_8, clear the accumulator, but will not alter the contents of the link.
3. The instructions CMA, IAC, in sequence, will find the negative value of the contents of the accumulator, in the two's complement binary form. (8's complement octal)

Solution: After the program is executed,

Loc. 427_8 will contain 7250_8

Loc. 430 will contain 6133

Loc. 431 will contain 7763

QUESTION 5.1

Repeat the previous exercise substituting the following values:

Loc. 424_8 now contains 4123_8

Loc. 425 now contains 2744

Loc. 426 now contains 0050

EXERCISE 5.2

A group of ten (10) octal numbers are stored in location 3000_8 to 3011_8. Determine how many numbers are positive (not including zero), negative, or zero and store these three tallies in consecutive memory locations.

Location	Mnemonic	Code
600	CLA CLL	7300
601	TAD 624	1224
602	CMA	7040
603	IAC	7001
604	DCA 625	3225
605	TAD I 626	1626
606	SPA SNA	7550
607	JMP .+3	5212
610	ISZ 627	2227
611	JMP .+6	5217
612	SZA	7440
613	JMP .+3	5216

Location	Mnemonic	Code	
614	ISZ 631	2231	
615	JMP .+2	5217	
616	ISZ 630	2230	
617	CLA	7200	
620	ISZ 626	2226	
621	ISZ 625	2225	
622	JMP .-15	5205	
623	HLT	7402	
624	0012	0012	
625	0000	0000	
626	3000	3000	
627	0000	0000	The
630	0000	0000	Three
631	0000	0000	Tallies
3000	5212	5212	
3001	3014	3014	
3002	0025	0025	
3003	0000	0000	
3004	6625	6625	
3005	7200	7200	
3006	1210	1210	
3007	0000	0000	
3010	0000	0000	
3011	2567	2567	

(The 10 numbers to be checked)

- - - - - - - - - - - - - - -

It is possible to combine microinstructions in order to perform more than one operation at a time. When combining microinstructions there are two things that must be considered, (1) that the resulting group of microinstructions can be coded properly, and (2) that there is a definite order or sequence in which the microinstructions are performed during the execution of the program.

When working with combined microinstructions, it will be helpful to refer to the bit settings for the Group 1 and Group 2 microinstructions:

Group 1 microinstructions

bit #	0	1	2	3	4	5	6	7	8	9	10	11
	1	1	1	0	CLA	CLL	CMA	CML	RAR	RAL	0/1	IAC
	operation code = 7_8			zero specifies Group 1							0: rotate one place 1: rotate two places	

Group 2 microinstructions

bit #	0	1	2	3	4	5	6	7	8	9	10	11
	1	1	1	1	CLA	SMA / SPA	SZA / SNA	SNL / SZL	0/1 / SKP	OSR	HLT	0
	operation code = 7_8			one specifies Group 2			0: SMA, SZA, SNL 1: SPA, SNA, SZL				zero specifies Group 2	

Refer to the bit settings given for the Group 1 and Group 2 microinstructions and note that a Group 1 microinstruction cannot be combined with a Group 2 microinstruction. The reason for this is that bit #3 cannot be set to a '0' and a '1' at the same time. Thus, the instruction to "complement the accumulator and skip on a nonzero link" could not be written as one line of instructions:

```
CMA SNL    (illegal)
```

but instead would be written as two separate lines of instructions:

```
CMA    (7040₈)
SNL    (7450₈)
```

Microinstructions can be combined within a particular group (either Group 1 or Group 2) by setting the proper bits to a '1'. A common instruction which is used at the beginning of a program is "clear the accumulator and link bit". These two microinstructions can be written as one line of instructions because they are both Group 1 microinstructions. The code for the instruction can be written as:

CLA CLL (7300)

(Note that bit #3 is set to '0' to indicate a Group 1 microinstruction, bit #4 is set to '1' to "clear the accumulator", and bit #5 is set to '1' to "clear the link bit".)

EXAMPLE 5.3

Write the group of microinstructions to form the negative of a number presently in the accumulator (i.e., the 2's complement of the number).

Solution: To obtain the negative of a number, the procedure is to complement each bit in the accumulator (all 1's would be changed to 0's and all 0's would be changed to 1's), and then increment the accumulator, in that order. The instruction to do this would be the two Group 1 microinstructions, CMA and IAC. The instruction then could be written as:

CMA IAC (7041)

or more commonly as:

CIA (7041) "Complement and Increment Accumulator"

The problem that exists is whether the instruction CMA IAC is going to be interpreted by the computer as "complement and increment the accumulator" or as "increment the accumulator and then complement it". Thus order is important in the execution of the microinstructions and it is necessary for the programmer

to know in what order the combined microinstructions are to be executed. (Note that this is only necessary when two or more microinstructions are combined to form one line of instructions.) This is the idea behind SEQUENCING, whereby there is a definite order in which the microinstructions will be executed by the computer. Note that even though a group of microinstructions may be coded properly, the sequence of execution may be different from that which the programmer had intended. The logical sequences by which the computer will execute a group of microinstructions follows:

Group 1 Logical Sequences	Group 2 Logical Sequences
1. CLA, CLL	1. Either SMA or SZA or SNL. Both SPA and SMA and SZL.
2. CMA, CML	2. CLA
3. IAC	3. OSR
4. RAR, RAL, RTR, RTL	4. HLT

According to the logical sequence for execution of Group 1 microinstructions, the combined microinstruction CMA IAC (7041), will be interpreted correctly and will form the negative of the number presently in the accumulator. There are a few things to note about the Group 2 "skip" microinstructions. First, the "skip" microinstructions are divided into two groups, the logical OR group, and the logical AND group. Whenever two or three microinstructions are combined from the OR group, the next instruction will be skipped if any one of the conditions are met. If two or three microinstructions are combined from the AND group, the next instruction will be skipped only if all of the conditions are met. Secondly, microinstructions in the OR group cannot be combined with instructions from the AND group because bit #8 of the Group 2 microinstructions would have to be set to a '0' and a '1' at the same time.

In the following exercises, different examples will be given on combining microinstructions. It will be helpful to refer back to the different bit settings for the Group 1 and Group 2 microinstructions and also the logical sequences of execution for each group.

EXERCISE 5.3

Write the octal form for the following groups of microinstructions and also what each group will actually do. Identify any illegal combinations and explain why they are not possible.

1. CLA CLL CMA CML
2. CLL RTL HLT
3. SMA SZA CLA
4. SPA SNL
5. CLL SPA
6. CLA SMA SZA
7. RAR RTR
8. SNA SZL
9. CLA CLL IAC RAR
10. CLA SPA

Solution:

1. CLA CLL CMA CML

   111 011 110 000 = 7360_8

Clear the accumulator and clear the link, then complement the accumulator and the link. After execution the accumulator will contain 111 111 111 111 = 7777 or -1, and the link bit will be set to a 1.

2. CLL RTL HLT

An illegal combination because Group 1 and Group 2 microinstructions cannot be combined.

3. SMA SZA CLA

111 111 100 000 = 7740_8

Skip the next instruction if the contents of the accumulator is less than or equal to zero, and then clear the accumulator. (SMA and SZA belong to the OR group and also bit #8 of the Group 2 microinstructions is set to a '0').

4. SPA SNL

An illegal combination because the OR group and the AND group "skip" microinstructions cannot be combined.

5. CLL SPA

An illegal combination because Group 1 and Group 2 microinstructions cannot be combined.

6. CLA SMA SZA

111 111 100 000 = 7740_8

Same as (3).. The order of writing this combined microinstruction is not important and will not affect the order of execution of the microinstructions.

7. RAR RTR

An illegal combination because bit #10 of the Group 1 microinstructions would have to be set to a '0' and a '1' at the same time.

8. SNA SZL

111 100 111 000 = 7470_8

Skip the next instruction if the contents of the accumulator is not equal to zero and if the link is equal to zero. (SNA and SZA belong to the AND group and also bit #8 of the Group 2 microinstructions is set to a '1'.)

9. CLA CLL IAC RAR (Illegal on PDP-8/S).

111 011 001 001 = 7311_8

Clear the accumulator and clear the link, increment the accumulator, then rotate the contents of the accumulator and the link one bit position to the right. This instruction will clear the accumulator and move a '1' into the link bit. Note that this instruction could also have been written as:

CLA CLL CML = 7340_8

10. CLA SPA

111 111 001 000 = 7710_8

Skip the next instruction if the contents of the accumulator is greater than or equal to zero and then clear the accumulator. Note that the order this combined microinstruction is written in is of no importance.

QUESTION 5.2

Repeat the previous exercise for the following groups of microinstructions:

1. CLA IAC RTL
2. CLA SMA SZL
3. SNA CLA
4. CMA SZA
5. SMA SZA SNL CLA

EXERCISE 5.4

Write the set of microinstructions and their corresponding codes for each of the following sets of instructions:

1. Clear the accumulator and clear the link, increment the accumulator, and then complement the accumulator.
2. Clear the accumulator and then skip on a nonzero link.

3. Rotate the accumulator left, clear the accumulator, and then skip on a zero accumulator.

Solution:

1. CLA CLL IAC CMA would be wrong because this instruction would complement the accumulator before incrementing it. The correct set of instructions would be:

CLA CLL IAC (7301_8)

CMA (7040_8)

2. CLA SNL (7620_8)

Note that order of execution is not important for this instruction, but the actual result would be to first execute the microinstruction SNL and then clear the accumulator.

3. RAL (7004_8)

CLA (7200_8)

SZA (7430_8)

RAL and CLA cannot be combined, nor can CLA and SZA be combined.

The following is a list of combined microinstructions which are commonly used:

	Microinstructions	Code	Meaning
*	CLA CLL	7300	clear acc., clear link
*	CIA (CMA IAC)	7041	complement and increment the acc.
	LAS (CLA OSR)	7604	load the acc. with the value of the switch register
	STL (CLL CML)	7120	set link bit to a '1'

* Indicates that it is used frequently.

	Microinstructions	Code	Meaning
	GLK (CLA RAL)	7204	put link bit into bit #11 of acc.
	CLA IAC	7201	set acc. to '1'
	STA (CLA CMA)	7240	set acc. to '-1'
	CLL RAR	7110	shift positive number 1 right
	CLL RAL	7104	shift positive number 1 left
	CLL RTL	7106	clear link, rotate 2 left
	CLL RTR	7112	clear link, rotate 2 right
	SZA CLA	7640	skip if acc.=0, then clear acc.
	SZA SNL	7460	skip if acc.=0, or link is '1', or both
	SNA CLA	7650	skip if acc.≠0, then clear acc.
	SMA CLA	7700	skip if acc.<0, then clear acc.
	SMA SZA	7540	skip if acc.≤0, then clear acc.
	SMA SNL	7520	skip if acc.<0, or link is '1', or both
*	SPA SNA	7550	skip if acc.>0
	SPA SZL	7530	skip if acc.≥, and if link is '0'
	SPA CLA	7710	skip if acc.≥0, then clear acc.
	SNA SZL	7470	skip if acc.≠0, and link is '0'

ANSWERS TO CHAPTER 5 QUESTIONS

Question 5.1

After the program is executed:

Location 427 will contain 0000

Location 430 will contain 0000

Location 431 will contain 7730

* Indicates that it is used frequently.

Question 5.2

1. CLA IAC RTL

$111\ 010\ 000\ 111 = 7207_8$

Clear the accumulator, increment the accumulator (set the accumulator to '1'), and then rotate the contents of the accumulator and link bit two places to the left.

2. CLA SMA SZL

An illegal combination because the OR group and AND group "skip" microinstructions cannot be combined.

3. SNA CLA

$111\ 110\ 101\ 000 = 7650_8$

Skip the next instruction if the contents of the accumulator are nonzero, then clear the accumulator.

4. CMA SZA

An illegal combination because Group 1 and Group 2 microinstructions cannot be combined.

5. SMA SZA SNL CLA

$111\ 111\ 110\ 000 = 7760_8$

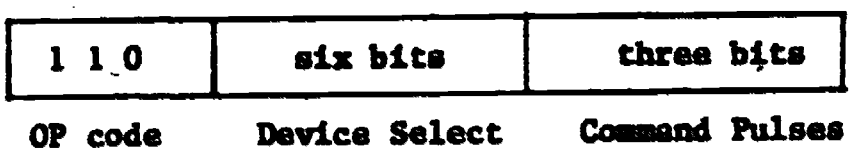
Skip the next instruction if either the contents of the accumulator are negative or zero or if the link bit is a '1', and then clear the accumulator.

CHAPTER 6

Input/Output

We will now learn how to get information into and out of the computer.

Until now the only way we could do this was by pushing the switches and reading the lights on the front of the computer. Obviously this is a tedious and very slow process. Fortunately there are instructions available which allow us to type the information onto a teletype and which allow the computer to type out its information. These are the input/output instructions (IOT instructions). The code for these instructions is as follows:



The OP code is octal 6. Six bits are used to identify the device for which an operation is to be performed. These bits are sent out to all devices, as are the three bits used for commands. Each device has a device selector which decodes the six bits of information and if the particular device has a code equal to the six bits, then the selector allows the three command bits to pass to the device. In this way only the selected device receives the commands. This is illustrated in the following figure.

Usually each device needs to communicate with the computer to indicate that it has data for the computer, or that it is ready to receive data from the computer. This is accomplished by use of a signal called a "flag". The flag is used to indicate that the device is either ready or busy. The computer has the capability to do instructions of the following sort:

1. Transfer data and/or operate the device
2. Test the status of the flag
3. Clear or set the flag

One instruction in particular is useful. This is the skip on flag instruction. This allows the computer to skip the next instruction if a particular device's flag is set. This is facilitated by the SKIP BUS on the PDP-8. If the bus sees a signal and a skip instruction is being executed then the next instruction is skipped. The use of this instruction will be illustrated in the next section. Note that the only device able to cause such a skip is the one being addressed by the six bits of the IOT instruction used for device select.

An illustration of the IOT instruction is communication between a teletype keyboard and the computer. Whenever a key is struck, the teletype prepares a series of high and low voltages, representing 1's and zeroes, puts these into a keyboard buffer register and then sends a signal to the computer (flag) that the buffer has data. If the computer is programmed to respond, the computer then puts this bit pattern into the accumulator register. What happens to the pattern from there depends on the user's program. Since the information must go through the accumulator register, we obviously need some mechanism to inform the computer that the information is coming so that the accumulator does not have some data from another program which would be lost when the accumulator gets the information from the teletype. This informing is accomplished by the setting and clearing of a keyboard "flag". The status of this flag is then

monitored by a specific set of IOT instructions called the Teletype Keyboard/ Reader Instructions. The flag being set indicates that the keyboard buffer contains information and is ready to send it. These instructions are easy to learn since there are only four of them.

Teletype Keyboard Instructions

Mnemonic	Octal	Meaning
KSF	6031	Keyboard Skip on Flag - Skip the next instruction if the keyboard flag is related (caused by the presence of information in the keyboard buffer register, i.e., a key has been pressed).
KCC	6032	Clear the accumulator and clear the keyboard flag.
KRS	6034	Move the bit pattern in the keyboard buffer to the accumulator register.
KRB	6036	Perform KCC and KRS.

An example routine which would read one typed character and store it in a location called TYPED is -

```
                .
                .
                .
                KCC             /CLEAR THE FLAG
                KSF             /SKIP IF FLAG IS SET
                JMP  .-1
                KRB             /READ KEYBOARD BUFFER
                DCA  TYPED
        TYPED,  0               /STORE TYPED CHARACTER HERE
                .
                .
                .
```

Since the flag will be cleared until a key is pushed, the computer will be in a continuous loop between the two instructions

```
KSF

JMP  .-1
```

until a key is pushed.

A nicely symmetrical concept and commands exist for sending information out of the computer. The "flag" now is a printer flag and it is "set" when the printing part of the teletype is ready to accept a character, while the flag is "lowered" or reset during the printing process. The repertoire for output is:

Teletype Printer Instructions

Mnemonic	Octal	Meaning
TSF	6041	Skip the next instruction if the printer flag is "set".
TCF	6042	Reset or clear the printer flag.
TPC	6044	Move the bit pattern from the accumulator register to the printer buffer register and print the character.
TLS	6046	Perform TPC and TCF.

Suppose that we have the bit pattern, called the ASCII code, for the letter "M" stored in a location we called "EM" and want it to be printed by the teletype. A short sequence of code to do that could be:

```
        .
        .
        .
EM, 315
        .
        .
        .
    CLA CLL
    TLS                 /RAISE FLAG INITIALLY
    TAD EM              /PUT ASCII CODE FOR M INTO ACC.
    TSF                 /WAIT FOR FLAG
    JMP .-1
    TLS                 /PRINT M
        .
        .
        .
```

Note that here there will be a continuous loop between

```
TSF
JMP  .-1
```

until the printer raises its flag indicating it's ready to type, then the TSF will be satisfied and therefore will skip the JMP .-1 instruction causing the letter M to be printed by the TLS instruction.

Again notice the basic difference between these I/O instructions and all the other instructions we have encountered previously. We now have the capability of having the running of a program be dependent on some external event, in this case the change in the status of an external device flag. Previously the only way we could have altered the running of a program once it started was to push the STOP button. This concept of having the running of a program be dependent on external events is the basis for "interrupt programming" or real-time computer usage as it is sometimes called.

Until now all the instructions we have asked the computer to execute have been linear in the sense that one instruction followed the other in a line of successive operations. We may have had JuMPs, but these were still linear in that:

a. a given instruction was executed;

b. the next instruction was a JuMP to some other location;

c. the next instruction to be executed after the JuMP was at the location which has been JuMPed to.

There are several situations where it would be very helpful if we could suddenly suspend, or interrupt, a program which is running, run another program for a while, then return to the previous program. This is exactly what INTERRUPTS do. There are at least three main situations where we want this ability.

1. Whenever the computer is communicating with another device, such as a teletype or magnetic tape unit, both the computer and the unit must be ready to send or accept data at the same instant so there must be some communication between them to establish this instant.

2. If a computer is being used to run or monitor some sort of process, such as an experiment in a lab or a steel mill, we would like the computer to respond to an emergency situation quickly rather than having to wait for some low priority program to finish running.

3. If a computer is being used in "timesharing", i.e., where several users are using the computer simultaneously, we would like the computer availability to go from user to user so quickly that all users feel they have the computer exclusively.

To make full usage of the speed of the computer we would like it to perform some useful calculations while waiting for external device flags rather than just looping endlessly as in the previous examples. This can be accomplished through two new instructions:

Mnemonic	Octal	Meaning
ION	6001	Turn the interrupt capability on.
IOF	6002	Turn the interrupt capability off.

When the interrupt capability is on, and a device flag is set, the computer will finish the single instruction it is executing but then instead of executing the next instruction of the program will automatically disable the interrupt system and execute a hardware JMS 0 instruction. From our previous descriptions you should remember that this will cause the location of the next instruction to be executed by the interrupted program to be placed into address 0 and will cause the execution of the instruction at address 1. It is the programmer's responsibility to write the interrupt service subroutine which begins at address 0.

The last instruction of this subroutine would naturally probably be a

JMP I 0

which would return control to the program which was running at the time the interrupt occurred since the JMS 0 stored the return address at location 0. Note again that the JMS 0 is hardwired or automatic, while the programmer must write the JMP I 0, to return from the interrupt. The basis of real time computing is therefore that while the computer is running some non-essential program, usually called a background program, it can be interrupted by a flag from an external device which causes the computer to devote itself to processing this interrupt. Upon completion of the interrupt processing, or foreground program, the computer resumes running the background program where it had left off.

Timesharing is possible because most I/O equipment is much slower than the computer. If a person is sitting at a teletype, the computer can normally execute thousands of instructions between successive key presses by the user. By allowing the user only a very short length of time, say 1/100 of a second, then moving to a second user, again for 1/100 of a second, then returning to the first user, the computer will then perform seemingly impossible tasks of servicing two users simultaneously! Since this switching between users is done via interrupts, there is no seeming interaction between the programs of the two users and each will feel that he (or she) has the full usage of the computer to herself (or himself). Of course, with large, fast computers, it is possible to service many users "simultaneously." A large computer may be reading/printing with 50 teletypes, be reading from several card readers and printing on several line printers all at once.

The following illustrates the use of the interrupt facility and interrupt programming.

Location	Instruction		Meaning
	*0		
0000	0		/LEFT FREE FOR RETURN ADDRESS
0001	JMP I SERV		/JUMP INDIRECTLY TO SERVICE ROUTINE
0002	SERVE		/ADDRESS OF SERVICE ROUTINE
	*1600		
1600	SERVE,	SKSCF	/SKIP ON SCOPE FLAG
1601		SKP	/SCOPE FLAG NOT RAISED
1602		JMP SRVSCP	/JUMP TO SCOPE ROUTINE
1603		KSF	/SKIP ON KEYBOARD FLAG
1604		SKP	/KEYBOARD FLAG NOT RAISED
1605		JMP SRVKEY	/JUMP TO KEYBOARD ROUTINE
1606		TSF	/SKIP ON PRINTER FLAG
1607		SKP	/PRINTER FLAG NOT RAISED
1610		JMP SRVPNT	/JUMP TO PRINTER ROUTINE
1611		.	
		.	
		.	

Note that location 1 has an indirect jump to SERV. SERV is the symbolic name of location 2 and location 2 has the symbolic name of location 1600 in it (SERVE).

Note that at location 1600 and following, there are a number of skip type instructions. These are used in the PDP-8 to determine which device has caused the interrupt. The PDP-8 has only one line to indicate that an interrupt has occurred. Therefore when an interrupt has occurred, it could have been caused by any device attached to the interrupt bus of the machine. The set of skips used above is called a skip chain. When the above chain determines which device has caused the interrupt, a JMP to the appropriate routine is caused, and the device is serviced. An example follows for the keyboard routine.

Location	Instruction		Meaning
	*1650		
1650	SRVKEY,	KRB	/READ DATA INTO ACC.
1651		DCA I LOC5	/STORE DATA INDIRECTLY
1652		INC LOC5	/UPDATE POINTER
1653		ION	/TURN ON INTERRUPT SYSTEM
1654		JMP I 0	/RETURN TO MAIN PROGRAM

In this service routine one datum is brought into the accumulator and stored in memory. The pointer is updated and then the interrupt system is turned back on. Remember that it was automatically turned off when the interrupt occurred. What would happen if another interrupt occurred just after the ION and before execution of the JMP I 0? The new interrupt would destroy the contents of location 0 disallowing return to the location in the main program from which the machine was initially interrupted. Since this is intolerable, the PDP-8 ION instruction does not actually turn the interrupt system on until after execution of of the instruction following the ION. This allows return from the current interrupt before another can be accepted.

SUMMARY

Input and output and control of devices by computer is usually accomplished by an interrupt cabability which allows a program to be interrupted, another program to be run and then followed by a return to the previous program.

CHAPTER 7

Assembler

It was mentioned in Chapter 3 that the programmer could use a program called the Assembler to translate his mnemonic-coded program into the binary coded instructions which the computer understands. There are several ways in which a programmer may prepare and submit his mnemonic program to the PDP-8. The instructions for preparing such a tape will be given later.

Refering to Exercise 3.3, the simple addition program would be submitted to the assembler in the following "file":

```
*30
CLA
TAD A
TAD B
DCA C
HLT
* 50
A, 0707
B, 0070
C, 0000
$
```

Note that symbolic names are now used to represent the memory locations which hold the two numbers to be added and the resulting sum. When the assembler translates the instruction "TAD A", it 'looks' for the definition of "A" in the file ("A, 0707") which means "A represents a memory location which holds the number 0707_8". "*50" is an instruction to the assembler to locate whatever follows in the "file", beginning at location 50:

Therefore: 'A' is in location 50 and has a value of 0707

'B' is in location 51 and has a value of 0070

'C' is in location 52 and has a value of 0000

Also, the assembler instruction "*30" tells the assembler to start the instructions of the program at location 30. Therefore, the instruction "CLA" will be in location 30 and the instruction "DCA C" will be in location (30+3)=33.

The dollar sign ($) is important! It must be at the end of the input to the assembler. It tells the assembler "end of the file to be assembled." If it is omitted, it could result in a "crash", a fatal programming error. When the assembler translates the file, it will result in the same binary (octal) coding that was developed by hand in Exercise 3.3. The procedure to use the assembler will be covered later in this chapter.

In the example above, 'A', 'B' and 'C' are referred to as symbolic addresses. The programmer does not have to figure out what absolute (or relative) numeric address they represent as in the examples in preceding chapters. They may be up to six (6) letters and numbers in length, but cannot begin with a number.

Examples:

Valid	Invalid
BUFFER	12X
A	7BP
X	
SUMS	

The symbols can be used anywhere in a program as long as they are defined.

Examples:

1.
```
        .
        .
        .
      JMP END        "END" is defined as the location containing
        .            the instruction "HLT". The computer, upon
        .            execution, will jump to the line of the pro-
        .            gram with the "HLT".
END,  HLT
```

2.
```
        .
        .
      DCA SUM        The computer will deposit the contents of the
        .            acc. into the location defined by the label
        .            "SUM", which initially is defined as contain-
        .            ing zero (0000).
SUM,  0000
```

Note that the comma (,) must appear after the definition of a symbolically addressed location:

```
SUM, 0000

END, HLT
```

NOTE: If a program had the instruction "JMP END" and the programmer forgot to 'define' a location with the name "END", an assembly error would occur and the program could not be assembled until "END" is defined, or, if "END" defined a location which contained a number instead of an instruction, "JMP END" would still jump to this defined location "END" and try to execute the number stored there. This would probably result in an error.

COMMENTS: The programmer can, optionally, attach a small 'note' explaining any program step if he desires. These notes are called comments and are ignored by the assembler if they are preceded by a slash (/). If the slash is not used, the assembler will try to translate the comments into instructions, and errors will occur. Examples:

```
        CLA             /Clear accumulator to start work
         .
         .
         .
        JMS ADD         /Jump to subroutine "ADD"
         .
         .
         .
END,    HLT             /End of program
ADD,    0000            /Begin subroutine "ADD"
         .
         .
         .
        JMP I ADD       /End of subroutine "ADD"
```

One of the greatest labor saving acts of the assembler is that it permits the programmer to do indirect addressing without having to do the tedious word counting. This may be accomplished in one of two ways:

1. Explicitly by use of the letter I after the instruction;
2. Intrinsically by use of the Pseudo-op "PAGE".

In the above example the effect of the "I" in JMP I ADD was to inform the assembler that we want to jump to ADD indirectly, i.e., we want to jump to the location in core whose address is stored in the word called ADD. Although in this example, the indirect addressing was for a return from a subroutine, the same usage of "I" will allow a jump or accessing of any location in core. Note that here the assembler does all the dirty work; it finds the relative address of ADD and encodes it into the instruction word at JMP I ADD.

The second method, that of using the Pseudo-op "PAGE", is even more powerful. "PAGE" is called a Pseudo-op (pseudo-operator), because it is not a computer instruction which we want the assembler to encode into machine language, but it is rather a command to the assembler. It says that we want no more instructions encoded into the current page and that the next instruction to be encoded should begin on the next page. If this instruction is given before the page is filled with code, there will be some unused words left. The assembler

will then use these words to store addresses for indirect references. Again, the assembler does all the dirty work. For example in:

```
NEXT,   IAC
          .
          .
          .
PAGE
          .
          .
          .
        JMP   NEXT
          .
          .
          .
PAGE
```

If the PAGE pseudo-op is used using CAP 98 (the cross assembler using the PDP-9) then the page desired must be explicitly identified. If it is not, then CAP 98 assumes that you mean page 0. In CAP 98 the PAGE pseudo-op is not needed to allow references to off page locations. The assembler will code the instruction using indirect addressing and uses the last locations on that page to code the full address, just as if you had reserved those locations with the PAGE Pseudo-op. **The cross assembler running on the PDP-11 does not allow use of the PAGE pseudo-op nor does it allow addressing directly any off page locations except page 0.**

The assembler will find the address of NEXT, store this address in the location following the last encoded instruction before the second "PAGE", and then encode the JMP instruction as an indirect jump. The assembler then allows easy access across page boundaries. The only concern of the programmer is that "PAGE" be declared to allow sufficient excess words on the current page for all indirect addresses. Since beginner programs usually do not need to be "tight" in the sense of not wasting core, a good rule is to issue "PAGE" after about every four paper pages of written code.

"PAGE" is only one of several pseudo-ops which make assembly language more convenient than it first appears. Details of these others may be found in any DEC literature which discusses the PAL-D Assembler, however, we have here provided enough to enable the reader to write meaningful programs now. All we have said can be summarized very briefly:

Summary of Rules for Writing Mnemonic Programs using the PAL-D Assembler:

1. The first line of code must be

*xxxx

where xxxx is the octal address where we want the first instruction encoded. Note: Do not use locations 200_8-204_8 or 7750_8-7751_8. This will be explained later.

2. All locations which we want to reference by name must be defined by the name followed by a comma. For example:

```
        .
        .
        .
SEVEN,7
        .
        .
        .
     TAD   SEVEN
        .
        .
        .
```

3. We may use indirect addressing by specifying it explicitly using the letter "I". For exampl :

```
        .
        .
        .
OTHER,5
     JMP I   OTHER
        .
        .
        .
```

4. We may tell the assembler to use indirect addressing for referencing locations in other pages by issuing PAGE before the page is filled with code.

5. We may comment on any line by using a slash (/).

6. The last line, physically, of our program must have only $.

Remember that we have been talking about input to the assembler. You may remember that we stated before that the input to the assembler is a punched paper tape. The directions for making the tape are in the next section. The normal sequence, then, for a beginner to use the PDP-8 is:

1. Write the program in MNEMONICS following the rules we have given for the assembler.

2. Use the EDITOR program to make a punched paper tape of the MNEMONIC program.

3. Use the ASSEMBLER to convert the mnemonic paper tape into a binary paper tape.

4. Use the LOADER to read the binary paper tape and load it into the core memory of the computer.

5. Run your program by putting its starting location into the PC register via the panel switches.

The Disk Operating System

A common feature of the PDP-8/I computer is a package of programs, including the assembler and other utility programs.

These programs are not kept in magnetic core memory, but are stored on a peripheral memory device called the DISK. A disk is a bulk storage device, consisting of a rotating magnetic disk on which "1"'s and "0"'s are recorded by magnetizing a small area of the disk's surface. The disk on the PDP-8 has a capacity of 831,488 12 bit words. (RK01 disk unit)

The user gets access to the disc and to any of the tasks for which programs have been written by use of a program called the MONITOR. The programmer tells the Monitor, by means of typing commands on the teletype, to 'fetch' from the disk the desired utility program(s), such as the assembler. This section will present three of the available utility programs to get the reader to begin using the disk system:

1. the Editor
2. the Assembler
3. the Loader

1. The Editor - a program which will accept the programmer's file (such as the example at the beginning of this chapter) and to punch a paper tape (called the source tape) of the file which can be submitted to the Assembler. It can also be used to correct typing errors, and to add or delete lines of instructions.

2. The Assembler - accepts the source tape created by the Editor, and translates it into binary instructions which the computer can execute; also will list any serious programming errors (such as illegal instructions); punches a binary-coded paper tape of the translated program to be submitted to the Loader.

3. The Loader - a program which reads in a binary-coded program tape and loads the instructions and/or data into locations in memory; the program would then be ready to execute by operating the front panel.

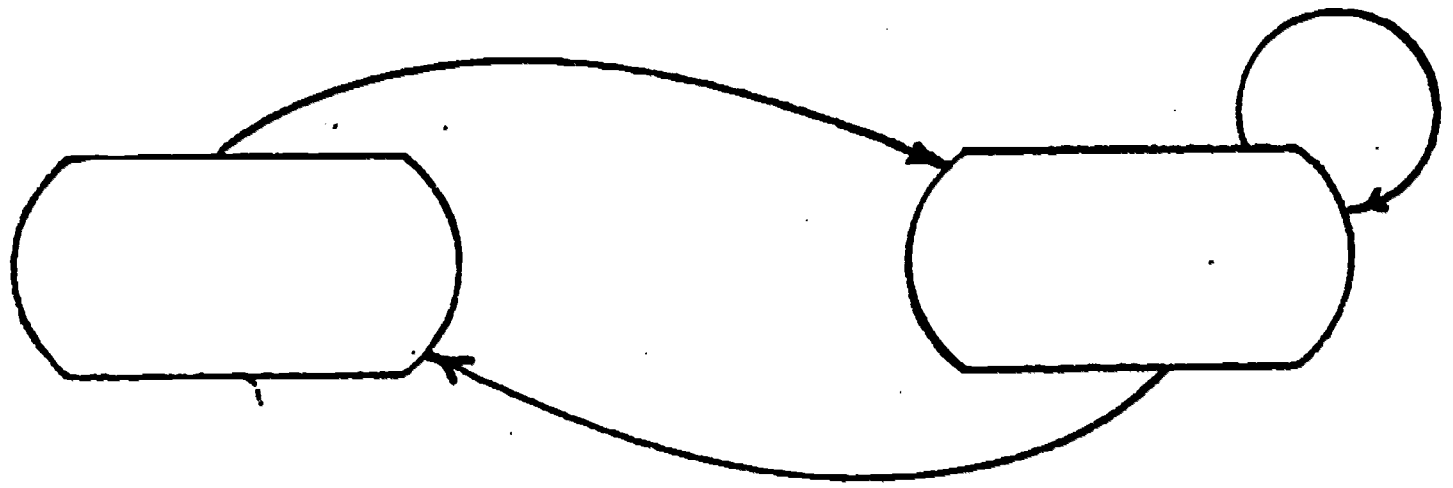
To prepare a program for execution on the PDP-8, using the disk system, the programmer will follow a certain sequence of steps. The sequence of steps is as follows:

As with the assembler, we present here only the minimum instructions to enable you to prepare a tape of a meaningful program. Full details of all options are in DEC literature describing the SYMBOLIC EDITOR program. The basic

idea of the editor is that it presents you with an empty "file" or scratch pad, correct any typing errors, and then tell the editor to punch out on paper tape the contents of the scratch pad. Therefore, whenever you type to the editor, your typing may be one of two distinct classes.

1. The material may be text which you wish to be written on your scratchpad.
2. The material may be a command to the editor.

To differentiate between these types, the editor lives in two different "modes", the text mode and the command mode. To change the editor's mode type the keys as shown here:



The most important editor commands are:

Command	Meaning
A	Append the following text, i.e., write it on the scratchpad.
L	List the buffer, i.e., type the entire contents of the scratchpad.
nL	List line n.
P	Punch the buffer, i.e., punch a paper tape of the contents of the entire scratchpad.

Command	Meaning
nD	Delete line n.
nI	Insert the next text before line n.

Again, ↵ represents the RETURN key. With these few commands, you should be able to type and correct a small program.

We have previously given the proper sequence of steps to running a small program. Assuming that you have written such a small program, we now give you step-by-step directions for doing so. Good luck!

1. Turn the teletype "on" (the knob on the lower right hand side of the teletype should be turned to the LINE position).

2. Load 7600_8 into the switch register and press LOAD ADD (7600 is the starting address of the Monitor), and then press START.

3. If the teletype responds with a ".", the monitor is ready to accept a command from the programmer. If the teletype does not respond with a ".", a procedure called "Bootstrapping the Monitor" must be done. The resident Monitor area, locations 7600 through 7777, may have been destroyed or altered and can be restored by manually loading into the computer the following bootstrap routine:

Location	Contents
0200	6603
0201	6622
0202	5201
0203	5604
0204	7600
7750	7576
7751	7576

After loading in the above program, set the switch register to 0200, press LOAD ADD, and then press START. The teletype should respond with a "."; the Monitor is ready to accept a command from the programmer.

Step 1: Prepare the tape for the Assembler by using the Editor (EDIT) program stored on disk. (Note: the doubly underlined portions will indicate characters typed out by the computer while your responses are singly underlined.

. EDIT ↵ (Call the editor program)

* OUT - T: ↵

*

* IN - T: ↵ (These commands tell the Monitor that input and output will be handled by the teletype)

* OPT - ↵ (No options)

* A ↵ ('A' is the Editor command APPEND which will append (or add) to the Editor buffer (at this time the buffer is empty; as the program is entered, the buffer will begin to fill))

Type in program to be assembled; be sure to end with a $ or a fatal error will result.

Press CONTROL and FORM keys at the same time to return Editor to command mode, which is denoted by an "*". Make any corrections necessary by using the given editor commands.

*

Turn the teletype to the LOCAL position, turn on the tape punch by pressing the ON button, press the HERE IS button on the teletype to punch a leader for the tape, and then return the teletype to the LINE position and press the OFF button on the punch.

P (Editor command to punch the entire contents of the buffer)

Turn on punch

(Editor will punch buffer after hitting carriage return)

Turn off punch

Press CONTROL and C Keys (Return to Monitor)

Step 2: Assemble the program by using the utility program PALD. The assembler must read your entire tape three times.

. PALD ↵

* OUT - T: ↵

*

* IN - T: ↵

*

* OPT - T ↵

↑ Place tape in tape reader with the control on STOP or FREE

Press CONTROL and P Keys

Turn on the reader

PASS 1

Turn off reader at end of tape

↑ Reload tape reader

Turn on PUNCH

Type CONTROL and P Keys - a leader is punched

Turn on reader when leader is completed

PASS 2 - BINARY TAPE IS PUNCHED

Turn off reader at end of tape

Trailer is punched

Turn off punch

Reload tape reader

Press CONTROL and P Keys

Turn on reader

PASS 3 - LISTING TYPES (the memory locations, their octal contents, and the program are all printed out)

Turn off tape reader at end of tape

(The computer will return to the Monitor), and then print a "."

If the assembler found any errors you must go back to the editor and make a new source tape.

Step 3: Load and execute BINARY tape using the utility program LOAD.

. LOAD ↵

* IN - T: ↵

* ↵

ST = ↵

↑

Press CONTROL and P Keys

Turn on tape reader

Machine will stop after tape is read; turn off tape reader.

Press CONT button on front panel of computer

↑

Computer will loop in resident monitor

Press CONTROL and P Keys, computer returns to monitor, prints a "."

To execute the program:

Press STOP button on front panel of computer

Set Switch Register to the starting address of the program

Press LOAD ADD, then press START

EXERCISE 7.1

Go back to Exercise 3.3 and use the disk monitor system to: create first the source tape, then the binary-coded tape, of the program, and load the program into memory; then execute the program.

Solution: The following page is a copy of the print-out from the teletype; it includes the commands typed by the programmer and the corresponding computer responses.

The Editor and Assembler (PALD) communications are shown. The Loader print-out is not shown.

Note that: (1) A dollar sign ($) must be at the end of the source tape that is submitted to the Assembler.

(2) When using the Assembler:

(a) The first time the source tape is read in, the teletype will print nothing, unless the assembler detects an error.

(b) The second time the tape is read, the teletype will print meaningless "garbage" (see following page)

(c) On the third read, the teletype prints a listing of the program along with the memory location and octal code of each instruction. Then it will print an alphabetical listing of all symbols in the program, and their octal locations (a symbol table).

```
·EDIT
*OUT-T:
*
IN-T:
*
*OPT-

*A
*3Ø
     CLA
     TAD A      /GET A
     TAD B      /ADD B TO IT
     DCA C      /STORE THE SUM
     HLT        /STOP
*5Ø
A,   Ø7Ø7
B,   ØØ7Ø
C,   ØØØØ      /STORE THE SUM HERE
$

*P
*3Ø
     CLA
     TAD A      /GET A
     TAD B      /ADD B TO IT
     DCA C      /STORE THE SUM
     HLT        /STOP
*5Ø
A,   Ø7Ø7
B,   ØØ7Ø
C,   ØØØØ      /STORE THE SUM HERE
$

·PALD
*OUT-T:
*
*IN-T:
*
*OPT-T
↑↑
          *3Ø
ØØ3Ø  72ØØ      CLA
ØØ31  1Ø5Ø      TAD A   /GET A
ØØ32  1Ø51      TAD B   /ADD B TO IT
ØØ33  3Ø52      DCA C   /STORE THE SUM
ØØ34  74Ø2      HLT     /STOP
          *5Ø
ØØ5Ø  Ø7Ø7  A,  Ø7Ø7
ØØ51  ØØ7Ø  B,  ØØ7Ø
ØØ52  ØØØØ  C,  ØØØØ    /STORE THE SUM HERE

A     ØØ5Ø
B     ØØ51
C     ØØ52
```

EXERCISE 7.2

Repeat Exercise 6.1, except use the program from Exercise 4.4.

Solution: The following five pages are a copy of the Editor and Assembler 'dialogue'. Note that the programmer made several typing errors when he created the file using the Editor. The errors are circled; he also forgot the instruction "HLT". The second and third following sheets show how the errors were corrected. The basic schemes are:

(1) Get the line which is in error.

(2) Delete that line.

(3) Insert the correct line.

OR

Insert the line(s) which was (were) missed.

Table 6.1, which follows the output from the teletype, lists the Editor commands.

NOTE: (1) When the "k th" line is deleted (*kD), the old (k+1)th line becomes the new k th line.

(2) When inserting a line (*jI), the new text is inserted before the 'j th' line, and the line count is adjusted. Also, after all the new text is typed in, hit CONTROL and FORM keys (together) to indicate 'end of new text' to the Editor, or anything else typed in will still be read by the Editor as 'new text'. After hitting CONTROL-FORM, the Editor will respond with a "*", it's ready for a new command.

(3) "*600" is counted as a line; it is line 1
"*650" is counted as a line; it is line 19
The line count is done in decimal, not octal.

After all the corrections were made, the programmer continued the processing of the program (punch the source tape, call PALD, etc.)

```
.EDIT
*OUT-T:
*
*IN-T:
*
*OPT-

*A
*6ØØ
        CLA
        TAD K27     /GET 27 DECI. AND
        CIB         /FORM -27
        DCA COUNT   /USE AS COUNTER
        TAD NUM     /GET LAST NO.
        IAC         /FORM NEXT 3 # TO TEST
        DCA NUMB    /AND STORE IT
        TAD NUM     /GET IT AGAIN
        JMS CHEK3   /JUMP TO SUBRT.
        CLA
        ISZ COUNT   /ADD 1 TO COUNTER.Ø?
        JMP LOOP    /NO ,JUMP
K27,    ØØ33
COUNT,ØØØØ
MIN3,  7775
WHERE,1ØØØ
NUM,   ØØØØ
*65Ø
CHEK2,ØØØØ
LP,     TAD MIN3    /SUBTRACT 3
        SPA SNA     /RESULT>Ø?
        JMP ZTEST   /NO
        JMP LP1     /YES ,JUMP TO LP1
ZTEST,  SZA         /ACC.=Ø?
END,     JMP ICHEK3/EXIT SUBRT.
         TAD NUM    /ACC.NOT ZERO,STORE #
        DCA I WHER  /AT ADDR. IN WHERE
        ISZ WHERE   /UPDATE STORING PTR.
        JMP END
$

*4L
        CIB         /FORM -27

*4D

*4I
        CIA         /FORM -27

*4L
        CIA         /FORM -27

*
```

```
*27L
END,  JMP ICHEK3/EXIT SUBRT.

*27D

*27I
END,  JMP I CHEK3 /EXIT SUR BRT.

*27L
END,  JMP I CHEK3 /EXIT SUBRT.

*29L
      DCA I WHER /AT ADDR. IN WHERE

*29D

*29I
      DCA I WHERE/AT ADDR. IN WHERE

*15L
K27,  0033

*15D

*15I
K27,  0033          /33 OCT.=27 DECI.

*18L
WHERE,1000

*18D

*18I
WHERE,1000          /FIND ADDR FOR STORING HERE

*
```

```
*6L
      TAD NUM     /GET LAST NO.

*6D

*6I
LOOP, TAD NUM     /GET LAST NO.

*6L
LOOP, TAD NUM     /GET LAST NO.

*8L
      DCA NUMB    /AND STORE IT

*8D

*8I
      DCA NUM     /AND STORE IT

*8L
      DCA NUM     /AND STORE IT

*14L
K27,  0033

*14I
      HLT         /YES,STOP

*14L
      HLT         /YES,STOP

*21L
CHEK2,0000

*21I
CHEK3,0000

*21L
CHEK3,0000

*22L
LP,   TAD MIN3    /SUBTRACT 3

*22D

*22I
LP1,  TAD MIN3    /SUBTRACT 3

*22L
LP1,  TAD MIN3    /SUBTRACT 3

*
```

```
*P
*6ØØ
      CLA
      TAD K27     /GET 27 DECI. AND
      CIA         /FORM -27
      DCA COUNT   /USE AS COUNTER
LOOP, TAD NUM     /GET LAST NO.
      IAC         /FORM NEXT # TO TEST
     DCA NUM    /AND STORE IT
      TAD NUM     /GET IT AGAIN
      JMS CHEK3   /JUMP TO SUBRT.
      CLA
      ISZ COUNT   /ADD 1 TO COUNTER.Ø?
      JMP LOOP    /NO ,JUMP
      HLT         /YES,STOP
K27,  ØØ33        /33 OCT.=27 CECI.
COUNT,ØØØØ
MIN3, 7775
WHERE,1ØØØ        /FIND ADDR FOR STORING HERE
NUM,  ØØØØ
*65Ø
CHEK3,ØØØØ
LP1,  TAD MIN3    /SUBTRACT 3
      SPA SNA     /RESULT>Ø?
      JMP ZTEST   /NO
      JMP LP1     /YES ,JUMP TO LP1
ZTEST, SZA        /ACC.=Ø?
END,  JMP I CHEK3 /EXIT SUBRT.
       TAD NUM    /ACC.NOT ZERO,STORE #
      DCA I WHERE/AT ADDR. IN WHERE
      ISZ WHERE   /UPDATE STORING PTR.
      JMP END
$
```

```
.PALD
*OUT-T:
*
*IN-T:
*
*OPT-T
↑↑
```

```
          *6ØØ
ØØ6Ø  72ØØ       CLA
ØØ61  1215       TAD K27     /GET 27 DECI. AND
ØØ62  7ØØ1       CIA         /FORM -27
ØØ63  3216       DCA COUNT   /USE AS COUNTER
ØØ64  1221  LOOP, TAD NUM    /GET LAST NO.
ØØ65  7ØØ1       IAC         /FORM NEXT # TO TEST
ØØ66  3221      DCA NUM    /AND STORE IT
ØØ67  1221       TAD NUM     /GET IT AGAIN
ØØ1Ø  425Ø       JMS CHEK3   /JUMP TO SUBRT.
ØØ11  72ØØ       CLA
ØØ12  2216       ISZ COUNT   /ADD 1 TO COUNTER.Ø?
ØØ13  52Ø4       JMP LOOP    /NO ,JUMP
ØØ14  74Ø2       HLT         /YES,STOP
ØØ15  ØØ33  K27, ØØ33        /33 OCT.=27 DECI.
ØØ16  ØØØØ  COUNT,ØØØØ
ØØ17  7775  MIN3,7775
ØØ2Ø  1ØØØ  WHERE,1ØØØ       /FIND ADDR FOR STORING HERE
ØØ21  ØØØØ  NUM, ØØØØ
          *65Ø
ØØ65Ø ØØØØ  CHEK3,ØØØØ
ØØ651 1217  LP1,  TAD MIN3   /SUBTRACT 3
ØØ652 755Ø       SPA SNA     /RESULT>Ø?
ØØ653 5255       JMP ZTEST   /NO
ØØ654 5251       JMP LP1     /YES ,JUMP TO LP1
ØØ655 744Ø  ZTEST, SZA        /ACC.=Ø?
ØØ656 565Ø  END,  JMP I CHEK3 /EXIT SUBRT.
ØØ657 1221         TAD NUM   /ACC.NOT ZERO,STORE #
ØØ66Ø 362Ø        DCA I WHERE/AT ADDR. IN WHERE
ØØ661 222Ø        ISZ WHERE /UPDATE STORING PTR.
ØØ662 5256        JMP END
```

```
CHEK3  ØØ65Ø
COUNT  ØØ616
END    ØØ656
K27    ØØ615
LOOP   ØØ64
LP1    ØØ651
MIN3   ØØ617
NUM    ØØ621
WHERE  ØØ62Ø
ZTEST  ØØ655
```

TABLE 7.2

Summary of Editor Commands

Command	Format(s)	Meaning
READ	R	Read incoming text and append to buffer until a form feed is encountered.
APPEND	A	Append incoming text to any already in the buffer until a form feed is encountered.
LIST	L	List the entire buffer.
	nL	List the line n.
	m,nL	List lines m through n.
PROCEED	P	Proceed and output the entire contents of the buffer and return to command mode.
	nP	Output line n, followed by a form feed.
	m,nP	Output lines m through n, followed by a form feed.
TRAILER	T	Punch four inches of trailer.
NEXT	N	Punch the entire buffer and a form feed; kill the buffer and read next page.
	nN	Repeat the above sequence n times.
KILL	K	Kill the buffer.
DELETE	nD	Delete line n.
	m,nD	Delete lines m through n.
INSERT	I	Insert before line one all text until a form feed is encountered.
	nI	Insert before line n until a form feed is encountered.
CHANGE	nC	Delete line n and replace it with any number of lines from the keyboard until a form feed is encountered
	m,nC	Delete lines m through n, replace from keyboard as above until form feed is encountered.
MOVE	m,n$kM	Move and insert lines m through n before line k.
GET	G	Get and list the next line beginning with a tag.
	nG	Get and list the next line after line n which begins with a tag.

TABLE 7.2 (continued)

Command	Format(s)	Meaning
SEARCH	S	Search the entire buffer for the character specified (but not echoed) after the carriage return; allow modification when found.
	nS	Search line n, as above, allow modification.
	m,nS	Search lines m through n, allow modification.
END FILE	E	Process the entire file (perform enough NEXT commands to pass the remaining input to the output file) and create an end-of-file indication; legal only for output to the system device. If the low-speed paper tape reader is used for input while performing an E command, the paper tape reader will eventually run out of tape, and at this point typing a form feed will allow the command to be completed.

CHAPTER 8

Overview

You now know all there is to know about digital computers. Yes, that is stretching the truth but you do now know how digital computers work. No matter how large or complicated the computer looks, the CPU just sits there and fetches one word at a time from memory, does something with it, then goes to the next instruction, etc., etc. You now understand how a computer can communicate with a person or many persons via teletypes or how a computer can monitor many events "simultaneously" through interrupts. You saw how it was necessary to have service programs to do anything useful with a computer because just operating it by its switches is hopelessly slow and tedious. So now you know what an operating system, just a collection of service programs, is. You saw how the Assembler, a program, can save you much of the dirty, tedious work in writing programs. By using the Assembler it is possible to write a still more complicated program called a Compiler and Viola! we have FORTRAN, BASIC, etc. These allow the user to be concerned even less with the internal workings of the computer. So now no matter how impressive a digital computer installation you see with tape drives and disc drives, and printers and punches and readers and teletypes and cathode-ray tubes and light pens and....and....., you know that, at the bottom of it all, in the panel behind the switches, the computer just fetches a single instruction from core, does something with it, goes to the next one, etc., etc. Of course, it may do this half a million times in the time it takes you to say -

The End.

APPENDIX

BINARY OCTAL - DECIMAL NUMBERS

Number Radix

All number systems have associated with them, a number called the radix or base. The radix is the number of symbols contained in the particular number system. In the decimal number system, the radix is 10 because it contains 10 symbols ranging from 0 - 9.

In the octal number system, the radix is 8 because it contains the symbols 0 - 7.

Example: What is the radix of the binary number system?

Solution: The binary number system as a radix of 2, the 2 symbols contained in it are 0 and 1.

Notice that the value of the radix in each number system is 1 greater than the highest possible value in that system.

In the binary number system, the highest possible value is 1. The radix is 1 greater than 1, or 2.

Example: What is the radix of the "number system" which contains the following symbols:

6 9 A Y % 0 # 1 5 ? F

Solution: Since there are 11 symbols in this particular number system, the radix is 11.

Radix Point

In all number systems, the radix point, or decimal point is the separation between the integers and fractional part of the number system. Because the

radix point is omitted in computer calculations, integers will be the subject of this appendix.

Subscripts

Because number systems share symbols, it is sometimes difficult to recognize the number system to which a particular number belongs.

For example, the number 7601 could belong to the decimal number system and to the octal number system. To clearly define which number system 7601 belongs to, the number is followed by its radix subscript, or subscript.

Therefore, 7601_8 belongs to the octal number system, and 7601_{10} belongs to the decimal number system.

Example: Show that the numbers 40, 101, 19 belong to the decimal number system.

Solution: 40_{10}, 101_{10}, 19_{10}

Example: Show that 40, 101, belong to the octal number system.

Solution: 40_8, 101_8

Example: Show that 40, 101 belong to the binary number system.

Solution: 101_2 -----The symbol "4" doesn't belong to the binary system, so the number 40 cannot be designated as belonging to the binary number system.

Number Position

Consider the decimal number 6947:

The 7 is said to be in the 0th or units position.

The 4 is said to be in the 1st or tens position.

The 9 is said to be in the 2nd or <u>hundreds</u> position.

The 6 is said to be in the 3rd or <u>thousands</u> position.

The rightmost position has a value of one (units position), and proceeding to the left, the next position has a value which is the radix (in this case, 10) times the preceeding positional value. The next position has a value 10 times the preceeding positional value, and so on.

These positional values can be expressed in powers of the radix value.

Also, the numbers occupying each position can be called the <u>coefficient</u> of that particular position.

<u>Example</u>: Express 6947_{10} in terms of its coefficients, radix, and positional values.

<u>Solution</u>: $6 \times 10^3 + 9 \times 10^2 + 4 \times 10^1 + 7 \times 10^0 =$

$6000 + 900 + 40 + 7 = 6947_{10}$

<u>Example</u>: Express 705_{10} in terms of its coefficients, radix, and positional values.

<u>Solution</u>: $7 \times 10^2 + 0 \times 10^1 + 5 \times 10^0 =$

$700 + 0 + 5 = 705_{10}$

<u>Example</u>: Express 097050_{10} in terms of its coefficients, radix, and positional values.

<u>Solution</u>: $0 \times 10^5 + 9 \times 10^4 + 7 \times 10^3 + 0 \times 10^2 + 5 \times 10^1 + 0 \times 10^0 =$

$0 + 90000 + 7000 + 0 + 50 + 0 = 97050_{10}$

Notice in the previous example, the 0 value coefficients contributed nothing to the value of the number.

Decimal Equivalent of Binary Numbers

Just as decimal numbers can be represented in this form, so can binary and octal numbers.

In the binary number system, the position values are based on the powers of two. Then, the 0th position has a value of 1, the 1st position has a value of 2 times the 0th position, or 2, the 2nd position has a value of 2 times the 2nd position, or 4, and so on.

In this way, any number in any number system can be expressed in its decimal equivalent.

Example: What is the decimal equivalent of the binary number 10110?

Solution:

Position #: 4 3 2 1 0
Number: $1\ 0\ 1\ 1\ 0_2$

$(1 \times 2^4) + (0) + (1 \times 2^2) + (1 \times 2^1) + (0) = 16 + 4 + 2 = 22_{10}$

Whereas, $2^4 = 16$, $2^3 = 8$, $2^2 = 4$, $2^1 = 2$, $2^0 = 1$

Example: What is the decimal equivalent of 1100_2?

Solution:

Position #: 9 8 7 6 5 4 3 2 1 0
Number: 0 0 0 0 1 0 0 1 0 0

$(1 \times 2^5) + (1 \times 2^2) = 32 + 4 = 36_{10}$

Decimal Equivalent of Octal Numbers

In the octal number system, the position values are based on powers of 8. The 0th position has a value of 1 (or $8^0 = 1$), the 1st position has a value of $8 \times 1 = 8$ (or $8^1 = 8$), the 2nd position has a value of $8 \times 8 = 64$ (or $8^8 = 64$), and so on.

Example: What is the decimal equivalent of the octal number 715?

Solution:

Position #: 2 1 0
Coefficient: 7 1 5

$(7 \times 8^2) + (1 \times 8^1) + (5 \times 8^0) =$

$448 + 8 + 5 = 461_{10}$

Example: What is the decimal equivalent of the number 6103_8?

Solution:

Position #: 3 2 1 0
Coefficient: 6 1 0 3

$(6 \times 8^3) + (1 \times 8^2) + (0) + (3 \times 8^0) =$

$3072 + 64 + 0 + 3 = 3139_{10}$

Example: Express the number 001010_8 in its decimal equivalent.

Solution:

Position #: 5 4 3 2 1 0
Coefficient: 0 0 1 0 1 0

$(0) + (0) + (1 \times 8^3) + (0) + (1 \times 8^1) + (0) =$

$0 + 0 + 512 + 0 + 8 + 0 = 520_{10}$

Notice in this example that 001010 could also have been considered a binary number, 001010_2. However, the decimal equivalent of 1010_2 is 10_{10} which greatly differs from 520_{10}.

In all of the previous examples, conversion was from the particular number system to the decimal number system which everyone is familiar with.

Conversion from the decimal number system to a particular number system will now be considered.

Binary to Decimal Conversion

Example: Convert 37_{10} to its binary equivalent. To accomplish this, repeatedly divide the decimal number by the radix of the number system being considered. In this example, begin by dividing 37 by 2:

Solution:			
	37 ÷ 2 = 18	Remainder = 1	0th position
	18 ÷ 2 = 9	Remainder = 0	1st position
	9 ÷ 2 = 4	Remainder = 1	2nd position
	4 ÷ 2 = 2	Remainder = 0	3rd position
	2 ÷ 2 = 1	Remainder = 0	4th position
	2 ÷ 1 = 0	Remainder = 1	5th position

Therefore, $37_{10} = 100101_2$

In this conversion, 37 was divided by 2 which is 18 and a remainder of 1. This first remainder fills in the 0th position of the binary number. 18 was then divided by 2 to get 9 and a 0 remainder, this remainder filled in the next position, the 1st. 9 was next divided by 2 to get 4 and a remainder of 1. The 1 filled up the 2nd position. 4 was divided by 2 to get 2 and a remainder of 0 which filled the 3rd position. 2 was divided by 2 to get 1 and a 0 remainder which went into position 4. 1 was divided by 2 to get 0 and a 1 remainder which was put in position 5. Repeated divisions by 2 will yield zeroes, which means that the power of two does not exist for those particular positions.

Example: What is the binary equivalent of 03_{10}?

Solution:			
	03 ÷ 2 = 1	with 1 remainder	0th position
	1 ÷ 2 = 0	with 1 remainder	1st position

$$03_{10} = 11_2$$

Example: 10_{10} has what binary equivalent?

Solution:			
	10 ÷ 2 = 5	with remainder of 0	0th position
	5 ÷ 2 = 2	with remainder of 1	1st position
	2 ÷ 2 = 1	with remainder of 0	2nd position
	1 ÷ 2 = 0	with remainder of 1	3rd position

$$10_{10} = 1010_2$$

Decimal to Octal Conversion

Decimal to octal conversion is accomplished in the same manner, using 8 as the conversion radix.

Example: What is the octal equivalent of 59_{10}?

Solution:			
	59 ÷ 8 = 7	with 3 remainder	0th position
	7 ÷ 8 = 0	with 7 remainder	1st position

$$59_{10} = 73_8$$

Example: What octal number is represented by 0991_{10}?

Solution:			
	0991 ÷ 8 = 123	with remainder 7	0th position
	123 ÷ 8 = 15	with remainder 3	1st position
	15 ÷ 8 = 1	with remainder 7	2nd position
	1 ÷ 8 = 0	with remainder 1	3rd position

$$0991_{10} = 1737_8$$

Grouping - Binary to Octal Conversion

The computer uses the binary number system in its calculations. But writing out long rows of binary numbers is very tiresome (write 1000101010011100110 five times to get an idea). Binary to octal conversion simplifies the handling of binary numbers.

First, the binary number is grouped into threes starting in the 0th position.

Example: Group 1000111010111011010 into threes.

Solution: 100 011 101 011 011 010

Because zeroes do not add value to a number, then can be included so that there are always three binary numbers to each group which is called a triad. This "addition" always takes place on the left side.

Example: Group 1011 into triads.

Solution: 001 011

Next, place the decimal value of each group below it and group the numbers together.

Example: Convert 101110001_2 to octal.

Solution:
101 110 001
5 6 1
$101110001_2 = 561_8$

Where $101 = 1 \times 2_2^2 + 1 \times 2_1^0 = 5$
$110 = 1 \times 2_2^2 + 1 \times 2^1 = 6$
$001 = 1 \times 2^0 = 1$

Example: Convert 1000_2 to octal.

Solution: 001 000
1 0
$1000_2 = 10_8$ $(=8_{10})$

Grouping - Octal to Binary Conversion

To convert a number from octal to binary reverse the above process.

Example: Convert 176_8 to binary:

Solution:
1 7 6
001 111 110

$176 = 001111110_2 = 1111110_2$

Zeroes were added to the octal '1' to form the left triad.

COMPLEMENTING

Complementing is important in a computer, because it allows the computer to subtract numbers "easier", by adding negative numbers.

1's Complement

The 1's complement of a binary number is a number formed by inverting all of the digits:

Example: What is the 1's complement of 10010_2?

Solution:

Number: 10010
Complement: 01101

Example: What is the 1's complement of 11_{10}?

Solution: Convert 11_{10} to Binary

$11_{10} = 1010_2$

Number: 1010

1's Complement: 0101

0101_2 is the binary 1's complement of decimal 11.

2's Complement

The 2's complement is found by adding '1' to the 1's complement.

Example: What is the 2's complement of 10010_2?

Solution:

Number:	10010
1's complement:	01101
	+ 1
2's complement:	01110

Example: What is the 2's complement of 11_{10}?

Solution: Since $11_{10} = 1011_2$

Number:	1011
1's complement:	0100
	+ 1
2's complement:	0101

"R's" Complements

The R's complement of an octal number can be found by subtracting the given number from the next highest power of 8.

Example: What is the R's complement of 17560_8?

Solution: Because 17560_8 contains 5 digits, the next power of 8 above 17560 is 100000_8 or 8^5.

Power of 8: 100000 Note that arithmetic is octal.

Number: -17560

R's complement: 60220

Example: What is the R's complement of 77_8?

Solution: There are 2 digits in 77, so the next highest power of 8 is 8^2 or 100.

Power of 8: 100

Number: -77

R's complement: 1

"R-1" Complement

The R-1 complement is found by subtracting 1 from the R's complement.

Example: What is the R-1 complement of 17560_8?

Solution: Power of 8: 100000

Number: -17560

R's complement: 60220

- 1

R-1 complement: 60217

The R's complement is analogous to the binary 2's complement.

The R's complement is analogous to the binary 1's complement.

Another way to find the R and R-1 complements is to convert the specified octal number to binary. Form that numbers 2's and 1's complements and then convert back to octal.

Example: Find the R and R-1 complement of 17560_8.

Solution:

	1	7	5	6	0	
	001	111	101	110	000	
Number:	001	111	101	110	000	
1's complement:	110	000	010	001	111	
2's complement:	110	000	010	010	000	
1's complement:	110	000	010	001	111	
R-1 complement:	6	0	2	1	7	$= 60217_8$
2's complement:	110	000	010	010	000	
R's complement:	6	0	2	2	0	$= 60220_8$

NOTE: The R's complement is often referred to as the eight's complement and the R-1 complement as the seven's complement.