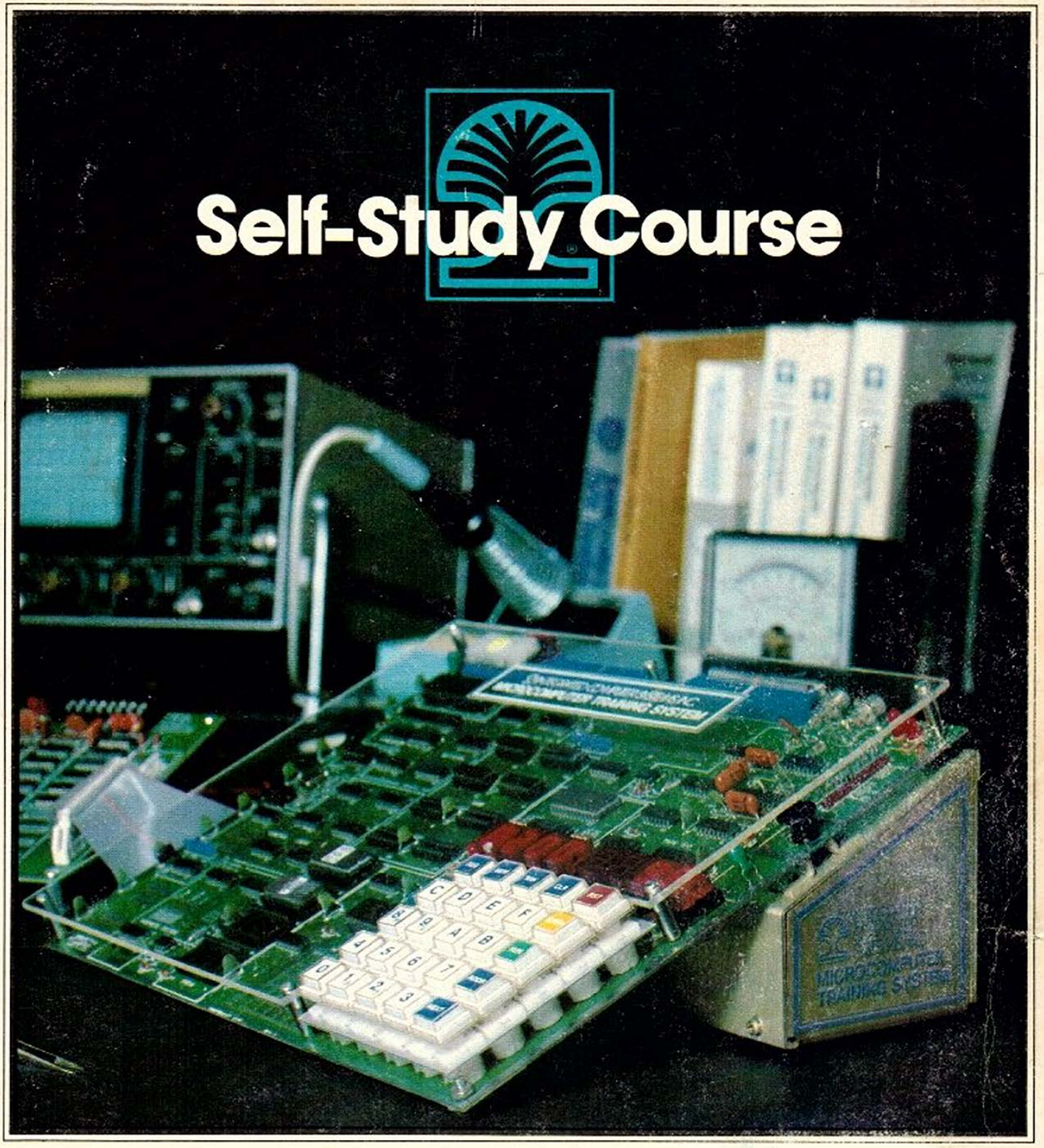
Self-Study Course



MICROPROCESSOR SOFTWARE & HARDWARE

Workbook/Text

Volume 1

Self-Study Course

Course 525A:

MICROPROCESSOR SOFTWARE & HARDWARE

Workbook/Text — Volume I

DEVELOPED & PUBLISHED BY:
INTEGRATED COMPUTER SYSTEMS
Course Development Division

SENIOR AUTHOR:
Edward Dillingham, M.E., M.S.E.E.

ASSISTED BY:
Dr. Daniel M. Forsyth
Dr. Rudolf Hirschmann
Ms. Ruth H. Savoie
Dr. David C. Collins

EDUCATION IS OUR BUSINESS™

MICROPROCESSOR SOFTWARE & HARDWARE

Two Volumes
ISBN 0-89438-009-5

Volume I
ISBN 0-89438-010-9

Volume II
ISBN 0-89438-011-7

TABLE OF CONTENTS

VOLUME I

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

VOLUME II

LIST OF ILLUSTRATIONS

VOLUME I

LIST OF ILLUSTRATIONS

VOLUME II

MICROCOMPUTER TRAINING WORKBOOK

INSTRUCTIONS

SYSTEM SETUP AND TEST PROCEDURE

MICROCOMPUTER TRAINING SYSTEM SETUP AND TEST PROCEDURE

I.1 RECEIVING INSPECTION

Upon receipt of your Microcomputer Training System, unpack it and inspect for any apparent shipping damage. If the equipment is damaged, or if any of the items listed below is missing, telephone Integrated Computer Systems for advice.

Items Supplied

MTS Circuit Board

Power Supply

Microcomputer Training Workbook

Pad of Coding Sheets

I.2 ASSEMBLY

Place the power supply on a table or desk with the sloping face towards the user. Mount the computer to the power supply by placing its lower edge on the table and its upper edge at the top of the sloping surface of the power supply. Reach under the plastic cover and push the two black plastic devices into mounting holes on the power supply.

I.3 POWER CONNECTION

Plug the multiconductor cable from the power supply into the socket at the upper left corner of the circuit board. Plug the power cord into a power outlet.

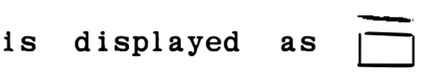
I.4 INITIAL TEST

Turn on the power switch at the back of the power supply. The numeric display above the keyboard should show 8200 in the four left hand digits. The next two digits should be blank, and the remaining digits may contain any data. No further testing should be required at this point, and the beginning user should now start reading the course material. If any problems are encountered that appear to be due to faulty hardware, it is recommended that the tests in the following sections be performed before calling Integrated Computer Systems for advice.

I.5 KEYBOARD TEST

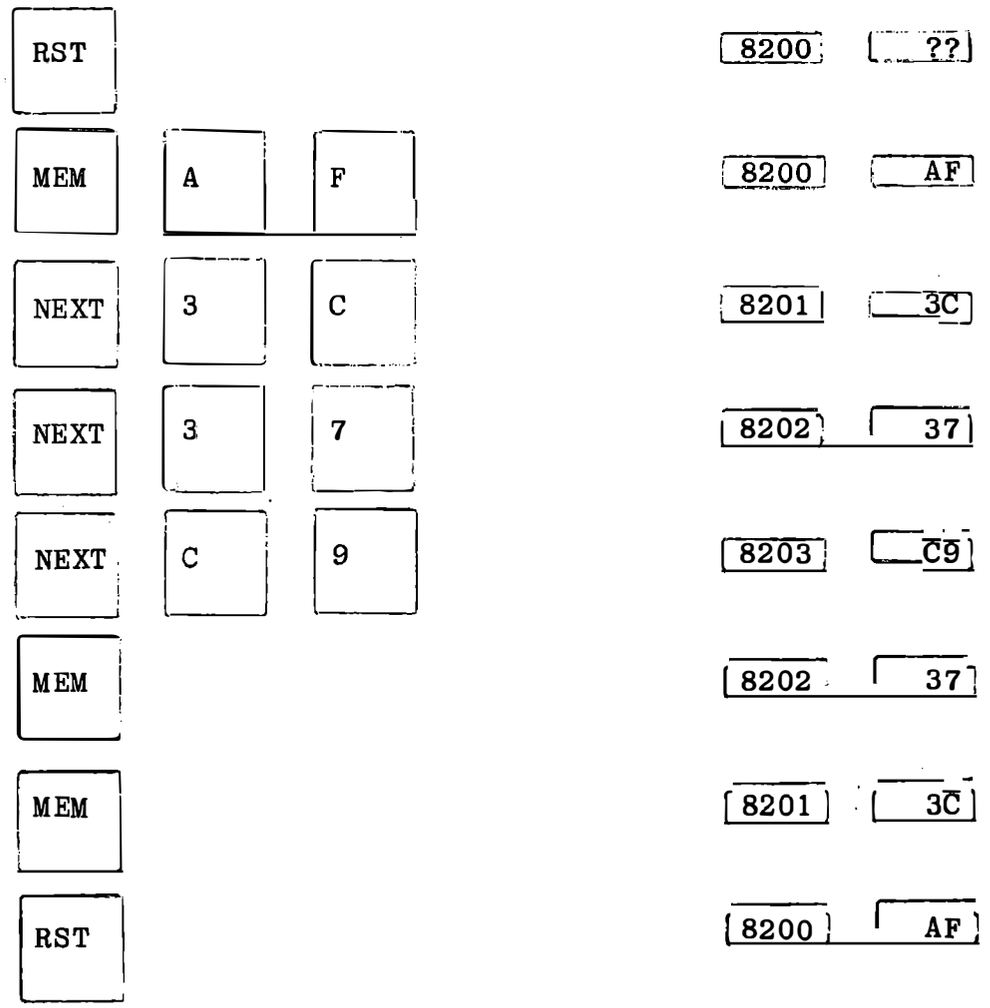
Press the following keys in the sequence shown. The displays that should appear are shown at the right. (?? indicates that the display is unpredictable.)

MEM	8200	??
0	8200	00
1	8200	01
2	8200	12
3	8200	23
4	8200	34

Proceed through the remaining white keys, 5 through F. Note that B is displayed as  to avoid confusion with 6, and D appears as  .

1.6 PROGRAM LOADING TEST

Load this simple test program by pressing keys in the sequence given below.



This program is used in the following test.

I.7 SINGLE STEP TEST

Load the program given in the preceding section.

In the middle of the left side of the circuit board a red- handled toggle switch projects slightly from under the plastic cover. Switch it toward the bottom of the board, to the STEP position. Press the following keys, and observe the display and the two red indicators (LED's) just to the left of the numeric display.

RST	8200 AF
STEP	8201 3C

The LED indicator lamp to the left of the display labeled ZERO should be on. The other indicator (labeled CARRY) should be off.

STEP	8202 37

Both indicators should now be off.

STEP	8203 C9

The indicator labeled CARRY should be on.

RST	8200 AF

This test has demonstrated that the single step function of the MTS is operating correctly, and has also tested the Zero and Carry indicators.

I.8 PROM CHECKSUM TEST

Set the red toggle switch to AUTO, and press the following keys in sequence.

ADDR	0	0	MEM		0000	31
ADDR	4	0	0	BRK	0400	BP.
REG	C	1			8200	C-01
ADDR	0	3	7	3	0373	21
RUN						

The display will be blank for a brief period, and then it will show:

0382 C-01

BRK	0400	BP.AA

The value displayed at the right hand two digits is a check sum for the content of the PROM memory. It should be AA for all versions of the monitor. Check the monitor version number by:

ADDR	0	2	8	0	0280	27

The number shown at the right indicates that your MTS is equipped with monitor version 2.7.

I.9 READ-WRITE MEMORY TEST

Load the program shown on the following page according to the following procedure.

RST					8200	??
ADDR	8	0	0	0	8000	??
MEM	F	3			8000	F3
NEXT	2	1			8001	21

Continue with the NEXT followed by two hex keys from the column headed CODE on the coding sheet until address 8015 has been loaded. Review the program by

ADDR	8	0	0	0	8000	F3
NEXT					8001	21
NEXT					8002	15

etc.

CODING SHEET

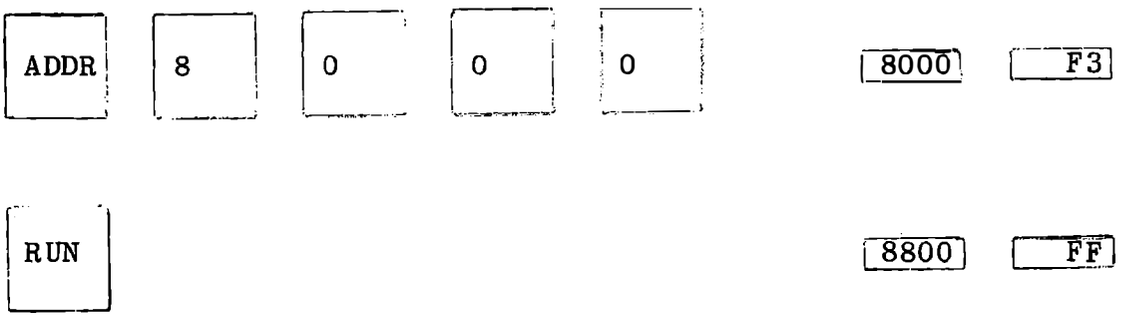
MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

READ-WRITE MEMORY TEST

ADDR	CODE		
8000	F3		DI
8001	21		LXI H, 8015
8002	15		
8003	80		
8004	23		INX H
8005	7E		MOV A, M
8006	2F		CMA
8007	77		MOV M, A
8008	BE		CMP M
8009	C2		JNZ 8012
800A	12		
800B	80		
800C	2F		CMA
800D	77		MOV M, A
800E	AE		XRA M
800F	CA		JZ 8004
8010	04		
8011	80		
8012	CD		CALL DMWD
8013	CE		
8014	02		
8015	76		HLT
6			
7			
8			
9			
A			
B			
C			
D			
E			
F			
8 0			
1			
2			
3			
4			
5			
6			
7			
8			

Now run the program by:



The program stops and displays a memory address at which it could not write and read data. This is the next address beyond the memory installed; 8800 if the MTS is equipped with 2048 bytes of memory. Any other address indicates a memory failure.

After testing each byte the program restores the previous value, so this test program may be run even when you have another program loaded.

I.10 SYSTEM EXPANSION

The Microcomputer Training System can be expanded in four ways:

a) An additional 2048 bytes of Read-Write memory can be plugged into the circuit board, giving a total of 4K bytes of RAM. Purchase Intel 2114 (or equivalent) 1024 x 4 static RAM chips and insert them in the empty sockets.

b) An additional 3K bytes of PROM can be plugged into the circuit board for programs that you have developed and want to keep permanently available. Also, by cutting and replacing some circuit board traces it is possible to replace the 1K PROM chips with 2K PROM chips, for a total PROM capacity of 8K bytes. Additional PROM chips will be offered by ICS in the future to provide additional built-in programs. Contact ICS for details.

c) The ICS Interface Training System can be connected to the MTS through a cable connector at the upper edge of the MTS circuit board. This training system includes additional input/output ports, interval timers, a power driver, digital/ analog/digital converter, and an extensive training course workbook covering the use of these devices, real time programming, interrupt handling, and closed loop control.

d) The MTS can be connected to an S-100 system to give access to a full 64K memory, Teletype or CRT terminal, printer, floppy disc, and other system devices. An interface cable and adapter board are available from ICS to plug directly into the S-100 bus. Such a system can support BASIC, FORTRAN, PLM and other high level programming languages.

This page intentionally left blank

MICROCOMPUTER TRAINING WORKBOOK

CHAPTER 1

HARDWARE AND SOFTWARE FUNDAMENTALS

INTRODUCTION TO CHAPTER 1

This chapter serves as the foundation upon which subsequent chapters are based. The basic structure of computer systems is described, principles of the binary number system are developed, the functional organization of memory and the central processing unit is introduced and the execution of several computer instructions is presented in some detail.

By writing and loading simple programs of your own, you will learn to use the Microcomputer Training System keyboard and display. You will observe first-hand the dynamics of program execution by watching, step-by-step, the results of executing individual instructions on your own computer.

If you are familiar with some of the topics covered here, skim but do not skip the material. The basic concepts are related to the structure and operation of the Microcomputer Training System.

After completing this chapter you will have a clear comprehension of the basic fundamentals of computer hardware and software. Most importantly, your knowledge will be rooted in hands-on usage of your MTS computer system.

1.1 BASIC CONCEPTS

1.1.1 Definition of a Computer

A computer is an electronic system which performs arithmetic and logical operations on data according to a sequence of instructions. The system consists of both hardware (physical devices) and software (sequences of instructions).

> HARDWARE: The electromechanical components of a computer system.

1.1.2 Basic Hardware Structure of a Computer

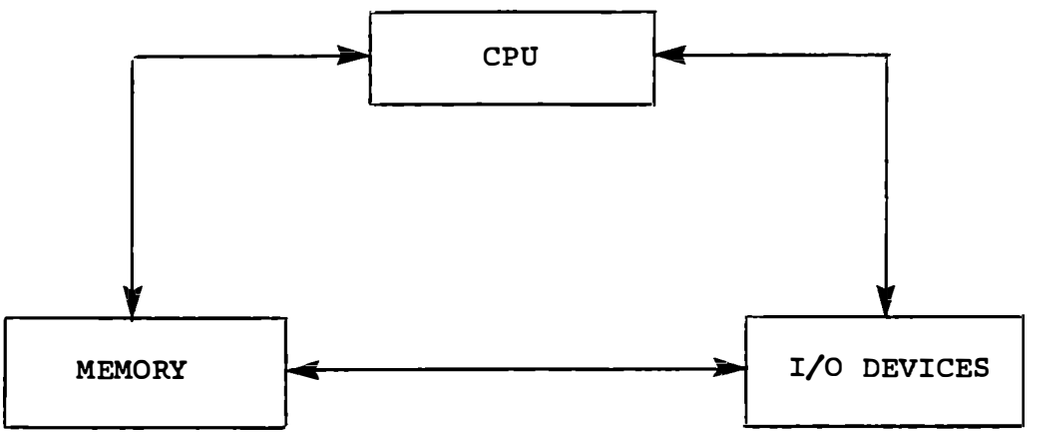
A computer has three principal hardware subsystems: a Central Processing Unit (CPU), a memory, and Input/Output (I/O) devices.

> CPU: The central processing unit, a set of elements which perform the actual arithmetic and logical operations. The CPU also provides the central control function of the computer system.

> MEMORY: A physical device in which data and instructions are stored for subsequent processing

I/O DEVICES:	Electro-mechanical devices that provide input of data and/or instructions to the system and output of results. Usually input devices are separate from output devices, e.g., a keyboard for input and a CRT display for output. Sometimes one device can combine both functions, e.g., a Teletypewriter can be used to input information and print output information.

These three subsystems are interconnected such that each one can communicate with the other two:



The model for computer operation is as follows:

1. Instructions are input via an INPUT DEVICE and stored in MEMORY.
2. Data are input via an INPUT DEVICE and stored in MEMORY.
3. The data are processed in a sequence and manner specified by the instructions.
4. The results of the data processing are output via an OUTPUT DEVICE.

In Figure 1-1, showing the layout of the MTS computer, the principal subsystems have been identified: The CPU, Memory, and Keyboard and Display. We will look at these in more detail later in the chapter.

INTEGRATED COMPUTER SYSTEMS, INC.

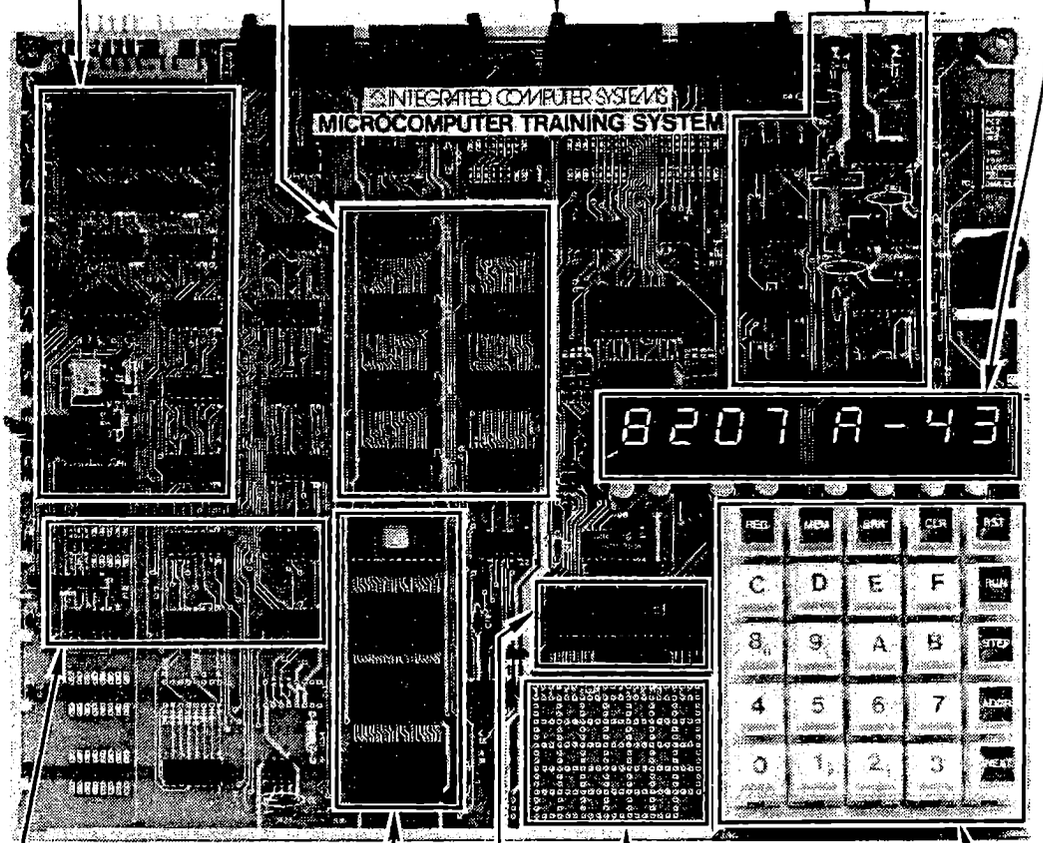
FULLY ASSEMBLED AND TESTED MICROCOMPUTER AND POWER SUPPLY

PROCESSOR HARDWARE
8080A Microprocessor and Control Logic

RAM MEMORY
2048 Bytes of RAM Memory for Programs and Data. Expandable On-Board to 4K Bytes.

AUDIO CASSETTE INTERFACE
Audio Cassette Interface and Associated Software for Easy Program Storage and Retrieval

DISPLAY
On-Board 8-Digit LED Display



DMA
Direct Memory Access (DMA) Channel

PROM MEMORY
Eraseable PROM Memory (containing the Educational Monitor Program) - 1024 Bytes. Expandable On-Board to 8K Bytes.

FREE AREA
Space for User's Hardware Additions

KEYBOARD
On-Board Keyboard with 25 Keys for Program and Data Entry.

PROGRAMMABLE PERIPHERAL INTERFACE
Programmable I/O Device Including Three 8-Bit Ports.

MTS Board Layout
Figure 1-1

1.1.3 Basic Software Concepts

The computer performs its functions under the control of a sequence of instructions. As an illustration, consider using a computer to convert miles to kilometers using the approximation that there are eight kilometers in five miles. The rule, as it might appear in a textbook, would say "Multiply the number of miles by eight and divide by five to obtain the answer in kilometers." The computer will need more detailed instructions than this. First assuming that the computer has been set up for the conversion by storing appropriate instructions in memory, it will also require that data be stored in memory. In this case the data are:

a. The number of miles to be converted.
b. The number 8.
c. The number 5.

Then, the sequence of operation might go as follows:

a. START.
b. Retrieve (miles) from memory.
c. Retrieve (8) from memory.
d. Multiply (miles) by (8).
e. Store result in memory under (temporary).
f. Retrieve (temporary) from memory when ready for next operation.
g. Retrieve (5) from memory.
h. Divide (temporary) by (5).
i. Store result in memory under (result).
j. Output/Display (result) and STOP

A sequence of instructions which performs such a calculation (or computation) is called a program.

> PROGRAM: A sequence of instructions which performs a specific calculation, computation or set of logical operations.

Programs may be specified which perform a vast and varied number of functions, including mathematical calculations, symbol manipulation, word processing and the detailed control and sequencing of I/O devices. A collection of such programs is referred to as software.

> SOFTWARE: 1) A collection of programs which perform many different functions; 2) The program component of a computer system in general, as distinguished from the hardware or physical component.

This page intentionally left blank.

1.1.4 The ICS Self-Study Microcomputer Training Course

This course is designed to provide you with the basic knowledge and practical experience which will give you the capability to:

- -Specify and write programs for performing a wide variety of different functions,
- -Enter programs and data into the Training Computer.
- -Verify that your programs operate correctly and, when they do not, modify them until they do.
- -Learn design techniques by actually connecting I/O devices to the Training Computer and controlling them with your own programs.
- -Explore the many hardware/software interrelationships, learn the cost-effective use of each, and design complete systems of your own.

In the succeeding chapters of this book you will be given, in step-by-step fashion, a sound foundation in both software and hardware techniques. You will progress from the simplified concepts of this introduction to a thorough understanding of these techniques as you "learn by doing", implementing each new concept yourself on your own computer.

1.2 NUMBER SYSTEMS AND REPRESENTATIONS

1.2.1 The Representation of Numbers

Physical representation of a decimal number requires an element with ten possible states, one for each of the decimal digits 0-9. Such a representation is found, for example, in the cog wheels of mechanical calculators. Elements with more than ten states are also common, for example in clocks.

Anyone having experience in solid state devices used in electronic circuits will know that substantial variability of characteristics exists for nominally identical devices. These characteristics are also usually a function of temperature. To stabilize such devices and to hold tolerances tight enough to distinguish unambiguously between multiple states would involve complex circuitry and would reduce reliability. Fortunately, the solid state devices are ideally suited for two-state operation in switching circuits, where an ON-state and an OFF-state can be readily distinguished. Thus, in the long run it is cheaper, simpler, and more reliable to work in terms of two-valued states, which are often two voltage levels, but can be - for example - positive or negative polarity of a magnetic element. In all cases, however, the computer operates on these two states in terms of logic TRUE and FALSE. This is equivalent to using a two-state or binary number system in which TRUE = 1 and FALSE = 0.

BINARY NUMBER SYSTEM: A two-valued number system using only the digits 0 and 1.

In most applications with which we will be concerned, the ON or HIGH voltage level will be equated to TRUE or 1, and the OFF or LOW voltage level (usually near ground potential) will be equated to FALSE or 0. This constitutes a POSITIVE LOGIC SYSTEM. Sometimes a NEGATIVE LOGIC SYSTEM is used, for ease of design in certain applications. In the latter system ON or HIGH is equated to FALSE or 0, and OFF or LOW is equated to TRUE or 1. Unless otherwise stated, we will use the POSITIVE LOGIC SYSTEM, which simply means that when considering a binary system using only the digits 0 and 1, the 0-level is low and the 1-level is HIGH.

To understand the basic principles of computer operation, it is essential to know something about digital logic and number systems. If you need a review of the former, then please see Appendix F, "A Primer on Digital Logic." We think you'll enjoy it. Now we will turn our attention to number systems in general and binary numbers in particular.

1.2.2 The Decimal Number System

Consider the following four ways of representing the decimal number 8192:

(1)	(2)	(3)	(4)
8000	8 x 1000	8 x 10 x 10 x 10	8×10^3
100	1 x 100	1 x 10 x 10	1×10^2
90	9 x 10	9 x 10	9×10^1
2	2 x 1	2 x 1	2×10^0
8192	8192	8192	8192

All of these representations are familiar. Column (1) indicates that the number 8192 can be represented as the sum of four different numbers. Columns (2) - (4) go further by illustrating that 8192 can be represented as the sum of four products. Column (4), however, exemplifies the basic principle of all number systems: each product can be obtained by multiplying a digit (in decimal the symbols 0-9) times a base (in decimal the number 10) raised to a power (see column 4 above).

> DIGIT: One of the symbols used in a number system.

> BASE: The number of different symbols used in a number system.

> POWER: The number of times that a base is multiplied by itself to form a product.

The decimal number system has ten digits or symbols; therefore the decimal number system has a base of ten, and in the example each product is obtained by multiplying a digit times the base ten raised to a power. The power to which the base is raised can be seen to be a natural progression from the least significant digit (rightmost) to the most significant (leftmost). The value of a base raised to a power is thus a function of its position in a string of digits, where position is counted from right to left starting with zero. In the following table we call the quantity of a base raised to its positional power a "multiplier". This number is multiplied by a digit to provide the final product:

POSITION	3	2	1	0
MULTI-PLIER	10^3 (1000)	10^2 (100)	10^1 (10)	10^0 (1)
DIGIT	8	1	9	2
PRODUCT	8000	100	90	2

Tables such as the above can be used to express the magnitude of a number in a system with any arbitrary base. The binary number system will be considered next.

1.2.3 The Binary Number System

The choice of base for a number system may be accidental or deliberate. The decimal system doubtless became widespread because of the ease of counting on ten fingers. Nonetheless, the Babylonians used a base of sixty and the Mayans, a base of twenty. The binary number system, which is most appropriate for computers, uses a base of two, and the digits 0 and 1.

Consider the following binary number:

11011

Had we lived from birth with a binary number system, we would immediately grasp its magnitude. As we have not, it is useful to convert it to its decimal equivalent.

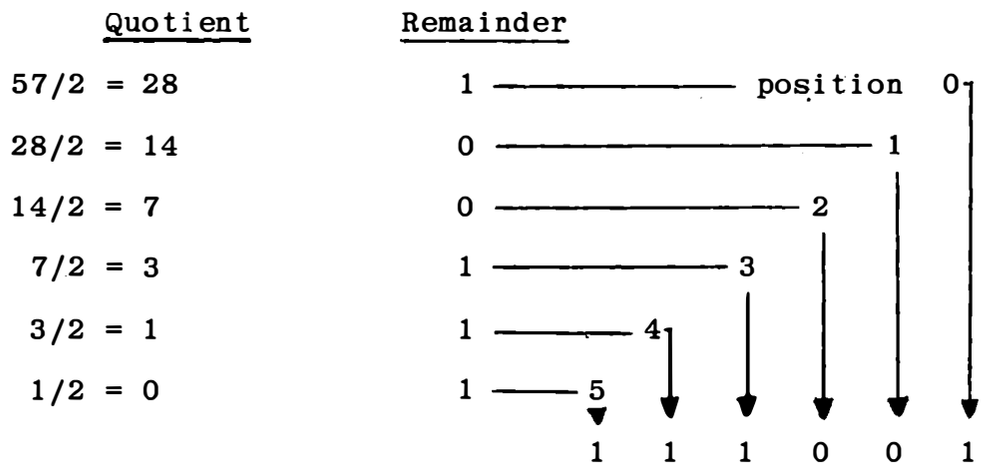
Knowing that binary numbers have a base of two, we can construct a table similar to that for decimal numbers. The table converts binary numbers to their decimal equivalent in the following fashion:

POSITION	4	3	2	1	0
MULTI-PLIER	2^4 (16)	2^3 (8)	2^2 (4)	2^1 (2)	2^0 (1)
DIGIT	1	1	0	1	1
PRODUCT	16	8	0	2	1

Thus 11011 (binary) = (16 x 1) + (8 x 1) + (4 x 0) + (2 x 1) + (1 x 1) = 27 (decimal). Larger tables may be constructed for converting longer strings of binary numbers.

Looking at the table again, it can be seen that the multiplier of each digit position is exactly twice the value of the position preceding it. Using this property, it is easy to calculate the products which are to be summed.

Conversion from decimal to binary could also be accomplished by using a table, but it is easier to use a process called "remaindering". Dividing an even decimal number by two will produce a quotient with a remainder of zero; dividing an odd decimal number by two will produce a quotient with a remainder of one. The remainders are used to construct the binary number, in the following example for decimal 57:



Decimal 57 is the equivalent of binary 111001. We may check this by writing down the products, counting from position: (1 x 1) + (2 x 0) + (4 x 0) + (8 x 1) + (16 x 1) + (32 x 1), which sum to 57.

1.2.4 Binary Addition and Counting

The rules for binary addition are very simple:

0 + 0 = 0

0 + 1 = 1

1 + 0 = 1

1 + 1 = 10

In performing the final addition, we would say to ourselves "One plus one equals zero and carry one". The rule for carries in binary is similar to that in decimal but much simpler, as there are only two symbols to worry about instead of ten. In both systems, symbols cycle (are successively incremented by 1) thru a digit position until all have been used. The next higher position is then incremented and the cycle is repeated.

The following addition tables illustrate counting rules for binary and decimal numbers:

0 + 0 =	0	0 + 0 = 0
0 + 1 =	1	0 + 1 = 1
1 + 1 =	10	1 + 1 = 2
10 + 1 =	11	2 + 1 = 3
11 + 1 =	100	3 + 1 = 4
100 + 1 =	101	4 + 1 = 5
101 + 1 =	110	5 + 1 = 6
110 + 1 =	111	6 + 1 = 7
111 + 1 =	1000	7 + 1 = 8
1000 + 1 =	1001	8 + 1 = 9
1001 + 1 =	1010	9 + 1 = 10

The binary portion of this table provides a graphic illustration of the relationship between a digit's position in a string and the power to which the base is raised at that position. In the "zero" position, note that that 0's and 1's cycle. In the "one" position, two 0's cycle with two 1's. In the "two" position, four 0's will cycle with four 1's. Each cycle is twice (base two) the length of the previous cycle. For decimal numbers each cycle will be ten times (base ten) the length of the previous cycle.

Subtraction, multiplication, division and the representation of negative binary numbers will be discussed in a subsequent chapter, but keep in mind that these operations are all derivatives of the basic operation of addition - which in turn is ~~drived~~ derived from counting.

When using more than one number system, their representations can often become confusing. To avoid this problem, a number may be subscripted to indicate its base:

11_2	(three)
11_{10}	(eleven)

In this manual whenever a number is not apparent from context, it will be subscripted or labelled appropriately.

A number of nomenclature conventions are important to introduce at this time: bit, string, bit position, most significant bit, and least significant bit.

BIT: An abbreviation for binary digit.

BIT STRING: A sequence of bits.

BIT POSITION: The location of a bit in a bit string.

MOST SIGNIFICANT BIT: The leftmost bit of a bit string.

LEAST SIGNIFICANT BIT: The rightmost bit of a bit string.

1.2.5 Hexadecimal Representation

We have seen that binary numbers are ideally suited to machine representation, and that they are easily added. Subtraction, multiplication and division are also simple operations in binary. There is in fact only one drawback to the use of binary numbers: they are difficult to perceive and describe if there are more than a few bits in a number. Consider, for example, the binary number:

1011000100001001

It is almost impossible to look at such a number and remember the digit in each bit position. There needs to be a way of encoding and naming such numbers so that they may be more easily comprehended, while at the same time preserving the underlying binary notion. A conventional arrangement is to separate the binary number into four bit groups.

A group of four bits can represent one of 16 numbers ranging from 0000 to 1111, or from 0 to 15. What we need is a set of sixteen symbols to represent each of the different numbers. We use the ten numerals 0-9 and the six letters A-F, as indicated in the following table. These correspond to the 16 white keys on the MTS keyboard.

0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

Returning to the original sixteen bit example,

1011	0001	0000	1001
B	1	0	9

it can be seen that this notation is much easier to read and remember. The introduction of a sixteen-symbol convention to represent groups of four binary digits is for the convenience of the user only. It can be seen, however, that we have in fact introduced a new number system with a base of 16_{10} , and which is called the hexadecimal number system (abbreviated hex).

> HEXADECIMAL NUMBER SYSTEM: A sixteen-valued number system using the symbols 0 - 9, A - F.

While it is possible to add hex numbers and construct tables for converting hex to decimal and decimal to hex, we will not consider these operations in any detail. The use of hex notation will be limited solely to the representation of four-bit groups of binary numbers, and is used only to facilitate describing them. The use of numbers such as $3C_{16}$, $82FF_{16}$ etc. will always be understood as a simple encoding of binary numbers. For practice, convert the following hexadecimal numbers to binary.

00

02

08

10

14

63

7A

9F

8200

83F8

023D

1.3 THE ORGANIZATION OF MEMORY

1.3.1 Memory Words

Data and instructions, represented as binary numbers, are stored in the computer's memory. The fundamental units of memory are words, each of which has a word size.

WORD: The basic unit of storage in a computer memory.

WORD SIZE: The number of bits contained in a word.

bit(N-1)..... bit 0 A word with word size N.

The word size of memory varies with the size of the computer system. Very large computers have word sizes from 32 to 64 bits. Mini-computers typically have word sizes of 16 or 24 bits. Micro-computers usually have a word size of 8 bits, which is the size of the MTS memory word. One factor is common to most - the word size is divisible by eight. This has led to the adoption of a special term for a a string of 8 bits.

BYTE: An 8-bit word. More generally, an 8-bit string, which can be part of a larger word.

1 0 1 1 0 1 0 1

A byte representing 181 decimal or B5 hex.

Each word in a memory has a location which is identified by a memory address.

MEMORY LOCATION: The position of a word in a memory.

MEMORY ADDRESS: A number specifying the exact location of a memory word.

A memory's size is equal to the number of words in a memory.

MEMORY SIZE: The total number of words in a memory.

An address size is the number of bits used to specify a memory address.

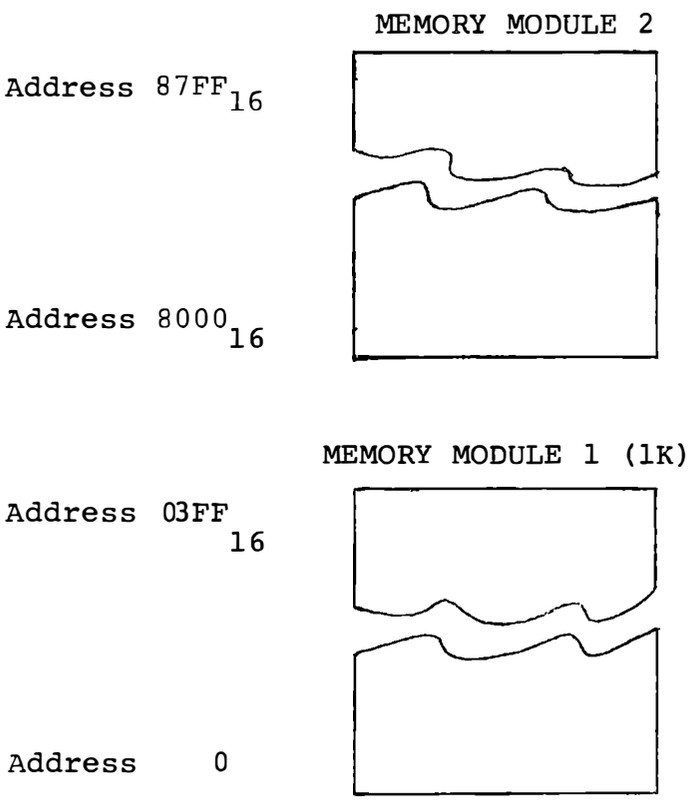
ADDRESS SIZE: The total number of bits which may be used to specify a memory address.

1.3.2 Memory Module

At first glance it might appear that memory size and address size are directly related. For example, a computer with an address size of eight bits can address 256 words; with an address size of sixteen bits, 65,536 words can be addressed. However, the capability of addressing words does not imply that the memory must contain that many words. Most computers, in fact, have far fewer memory words available than they are capable of addressing. This is possible because memory is usually available in modules, with each module containing a few hundred or a few thousand words. The same CPU can thus be used in a variety of configurations, with the size of memory used dictated by the application for which the system has been designed.

> MEMORY MODULE: A unit of memory containing a fixed number of words.

Memory modules contain a number of words or bytes which is generally expressed as some factor of the quantity 1024 = 2^{10}. This is such a convenient unit for describing memory size that the number 1024 has been given the symbol K. A memory module containing 4096 bytes is referred to as a 4K memory; one with 512 bytes, a .5K memory. These concepts may be illustrated by the diagram on the following page:



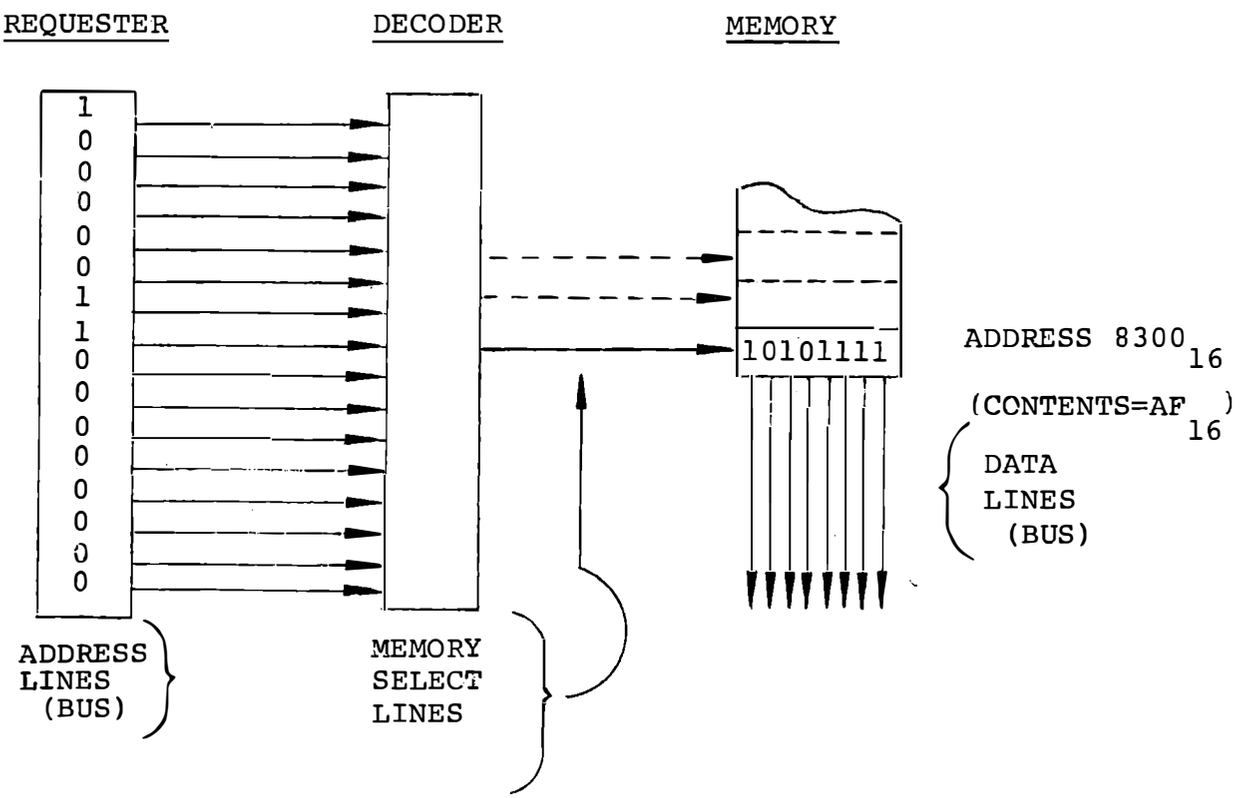
The diagram describes the memory structure of a system with a word size of eight bits, an address size of sixteen bits (Why are sixteen bits required?), and a memory size of 3K words. It is in fact the memory structure of a minimum MTS computer system. Two important properties of memory organization are illustrated here. 1) Within a memory module, addresses are numbered sequentially; 2) If two or more modules are used, the first address of the second module is independent of the last address of the first module (although for ease of implementation it is usually some multiple of 1K). This independence is made possible by the fact that the two modules are "wired in"; the addresses of available words are determined by the hardware of the system.

1.3.3 Memory Access

The process by means of which a request is made to access a memory word is conceptually simple. The requestor (the CPU or, in some instances, an I/O device) outputs the requested address on parallel address lines, one line for each bit of the address. This signal is interpreted by an address decoder, which then selects the single lead which will access the desired memory word. The contents of the word will then be made available on the data lines.

> DECODER: A device containing a switching matrix which responds to the pattern of a set of input signals and outputs a signal determined by that pattern. Usually the output takes the form of activating a particular output line.

The diagram on the following page illustrates the process:



The memory select lines are essentially internal to the memory itself. The address lines and data lines serve as the communication channels between the CPU and its memories and I/O devices, and they have special names: address bus and data bus.

ADDRESS BUS: The set of lines carrying address information. The number of lines in the bus will be equal to the address size of the system.

DATA BUS: The set of lines carrying data. The number of lines will be equal to the word size of the system.

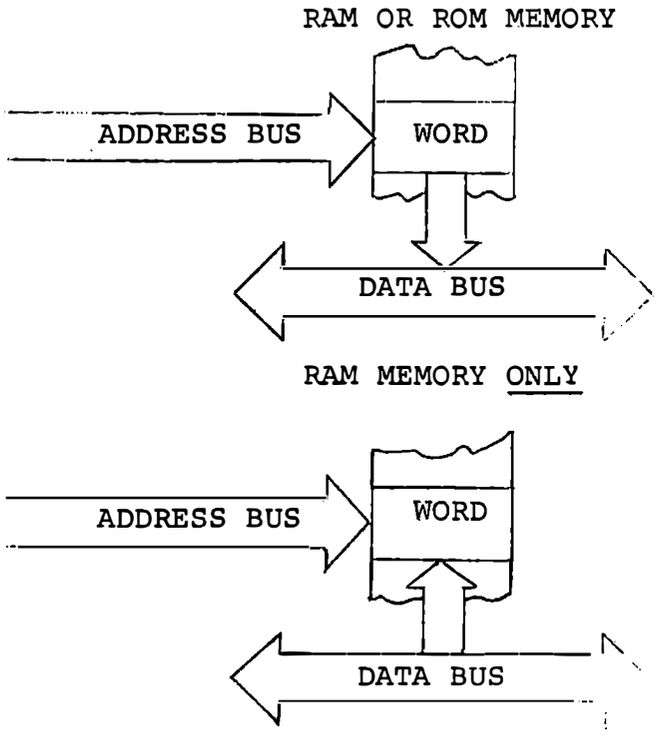
1.3.4 Varieties of Memory

There are two types of memory in your MTS computer system: Random Access Memory (RAM), which may be read or written, and Read Only Memory (ROM), from which data may be read but not written into. To read data from memory, the address bus is used to select a word whose contents can then be read out onto the data bus. To write data into memory, the address bus is used to select a word whose contents are then changed to that which is being sent on the data bus. Reading the contents of a word leaves the word unchanged.

RAM: Random Access Memory which may be both read and written.

ROM: Read Only Memory which may be read but not written.

Read and write operations are illustrated in the following diagram:



Read operations put the contents of a word onto the data bus.

Write operations put the information on the data bus into a word.

In Figure 1-2 the RAM and ROM of your MTS system are indicated. There are 2048 words of RAM and 1024 words of ROM. Your ROM contains a set of programs called the MONITOR, designed to assist you in learning the system. The functions of the MONITOR will be defined step-by-step as you progress through this manual. The RAM will be used to store the different programs which you will write yourself. ROMs are used for programs which do not need to be changed, and are protected against inadvertent modification. RAMs are used for program development (these programs can then be placed in a ROM, but special equipment is required) and for storage of transient data in actual applications. Some of the RAM in your MTS is required for use by the MONITOR and is not available for user programs. This will be discussed later.

INTEGRATED COMPUTER SYSTEMS, INC.

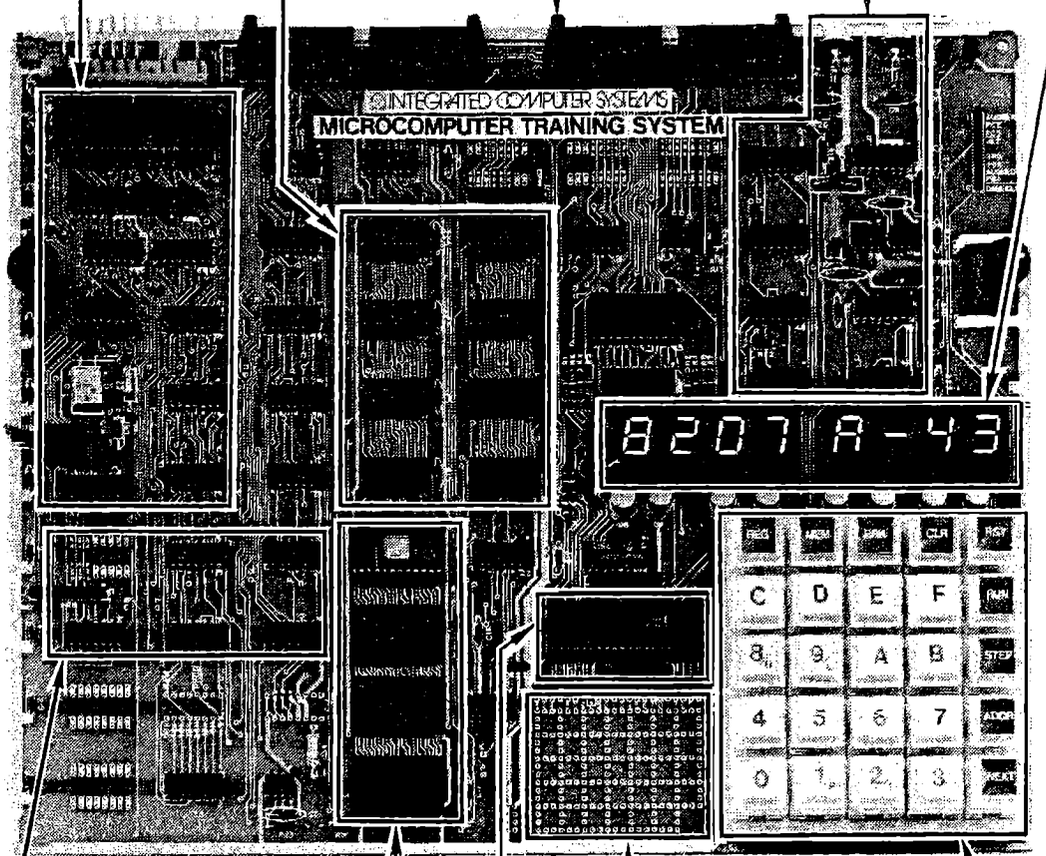
FULLY ASSEMBLED AND TESTED MICROCOMPUTER AND POWER SUPPLY

PROCESSOR HARDWARE
8080A Microprocessor and Control Logic

RAM MEMORY
2048 Bytes of RAM Memory for Programs and Data. Expandable On-Board to 4K Bytes.

AUDIO CASSETTE INTERFACE
Audio Cassette Interface and Associated Software for Easy Program Storage and Retrieval

DISPLAY
On-Board 8-Digit LED Display



DMA
Direct Memory Access (DMA) Channel

PROM MEMORY
Eraseable PROM Memory (containing the Educational Monitor Program) - 1024 Bytes. Expandable On-Board to 8K Bytes.

FREE AREA
Space for User's Hardware Additions

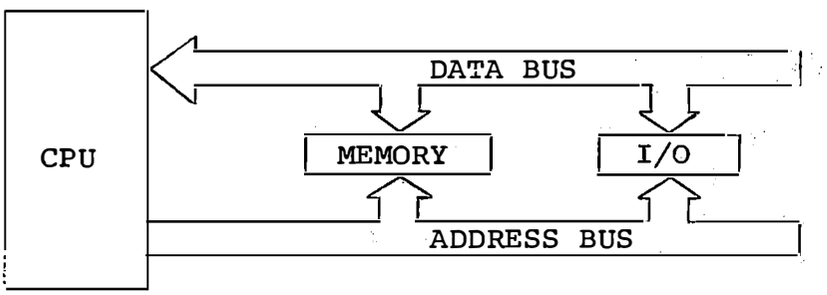
KEYBOARD
On-Board Keyboard with 25 Keys for Program and Data Entry.

PROGRAMMABLE PERIPHERAL INTERFACE
Programmable I/O Device Including Three 8-Bit Ports.

MTS Board Layout
Figure 1-2

1.4 STRUCTURE OF THE CPU

On the first page of this chapter, the CPU was described as a set of elements which perform the arithmetical and logical operations and also serve as the central controlling elements of a computer system. We will look at some of these operations in more detail, but first let us review the structure of the system including the data bus and address bus:

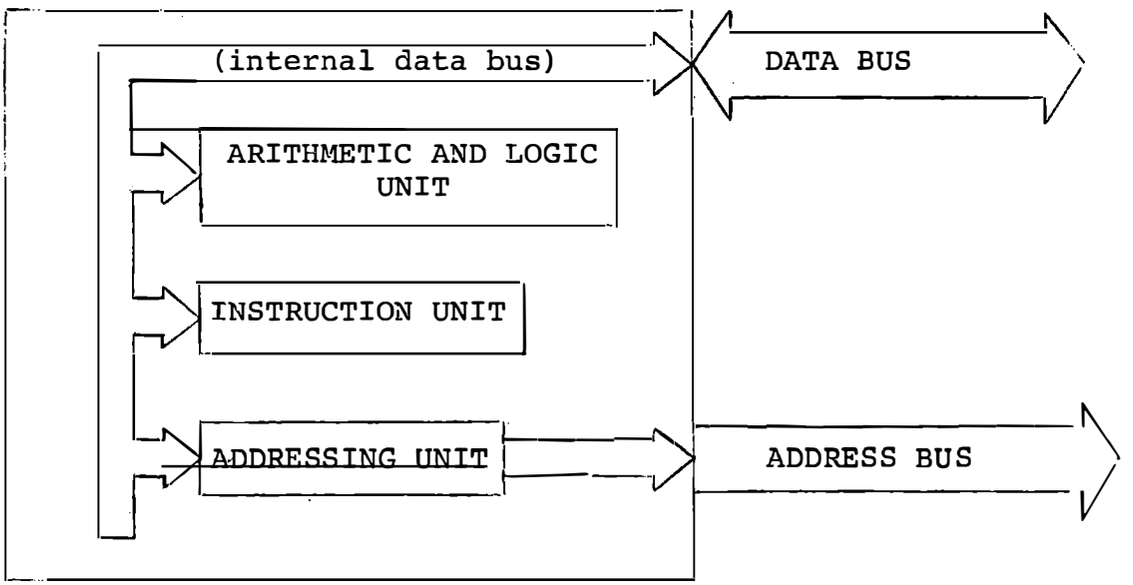


The CPU may send or receive data along the data bus which is bidirectional. The CPU sends memory addresses out on the address bus, but does not receive from the address bus.

1.4.1 Functional Units

Internally, the CPU consists of three primary functional units. One is concerned principally with addressing functions, selecting addresses which will be sent out on the address bus. A second unit is concerned with interpreting and decoding the instructions which are stored in memory. The third is the Arithmetic and Logical Unit (ALU), in which all arithmetic and logical functions are performed. These units are able to communicate with each other over an internal

data bus, which is the fourth functional component of the CPU. The following diagram schematically outlines this organization:



CPU ORGANIZATION

The internal data bus is illustrated here only to indicate that there is a physical pathway between the various internal units of the CPU. The term data bus will always refer to the main (external) data bus, to avoid confusion.

Each of the internal units of the CPU has one or more registers, one or two byte storage elements which are similar to memory locations but which are used for temporary storage, for holding the results of a calculation, or for other dynamic purposes. The nature and function of each register will be described as its use is first encountered.

> REGISTER: A one or two byte storage location used by the CPU for temporary storage or other dynamic purposes.

1.4.2 The Execution of Instructions

A computer is a system which performs operations on data according to a sequence of instructions called a program. A program is created by a user (programmer) to cause the computer to fulfill a particular task. An instruction is the smallest element of the program that conveys a complete meaning; it is similar to (and often represented by) a command in human language such as ADD B to A. To be stored in the computer's memory and handled by its electronic circuits, the

instruction must be represented as a binary number. This representation is called a code, and a program in binary code ready for use by the computer is said to be in machine language.

INSTRUCTION:	The smallest element of a computer language that directs the computer to perform a specific operation.

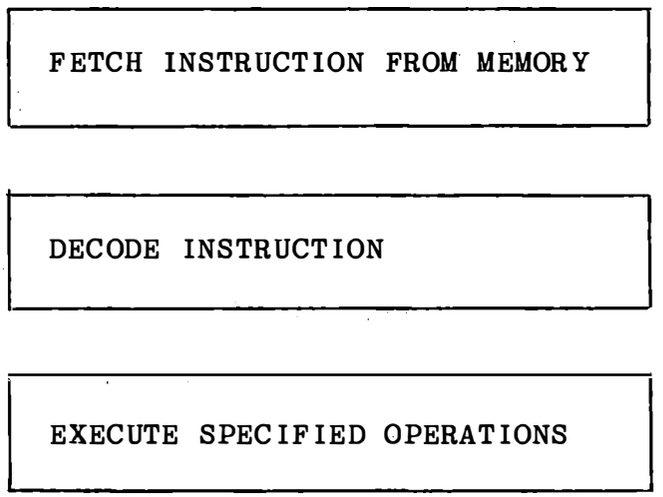
Each execution of an instruction will perform one small step in the calculation or process which the program is designed to accomplish. In turn, the execution of each instruction is broken up into a number of steps which are performed one after another.

1.4.3 Instruction Cycles

The program will be stored in memory; therefore the execution of each instruction will have to start with the transfer of an instruction from memory to one of the registers of the CPU. Then the instruction will be decoded (interpreted) and the operations specified will be carried out. The total time taken to fetch and execute an instruction is called an instruction cycle. The length of an instruction cycle varies considerably, depending upon the operations which must be performed. Every instruction cycle, however, begins with an instruction fetch.

INSTRUCTION CYCLE: The total time taken to fetch and execute an instruction.

The basic sequence of events during an instruction cycle is:

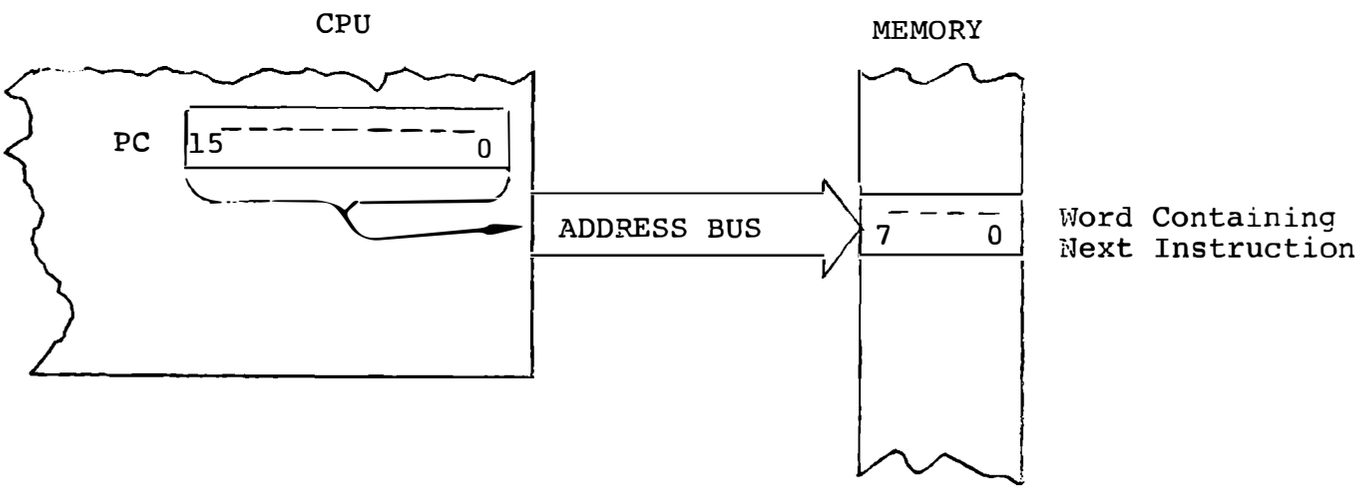


1.4.4 The Program Counter

To fetch an instruction from memory requires a memory address. The address from which an instruction is to be fetched is always contained in a CPU register called the Program Counter (PC). There are two strong implications in this statement: there must be a way to initialize the PC with the address of the first instruction in a program, and there must be a way to modify the PC after each instruction cycle so that it will contain the proper address for the next instruction to be fetched.

PROGRAM COUNTER:	A register in the CPU which contains the address of the next instruction to be fetched.

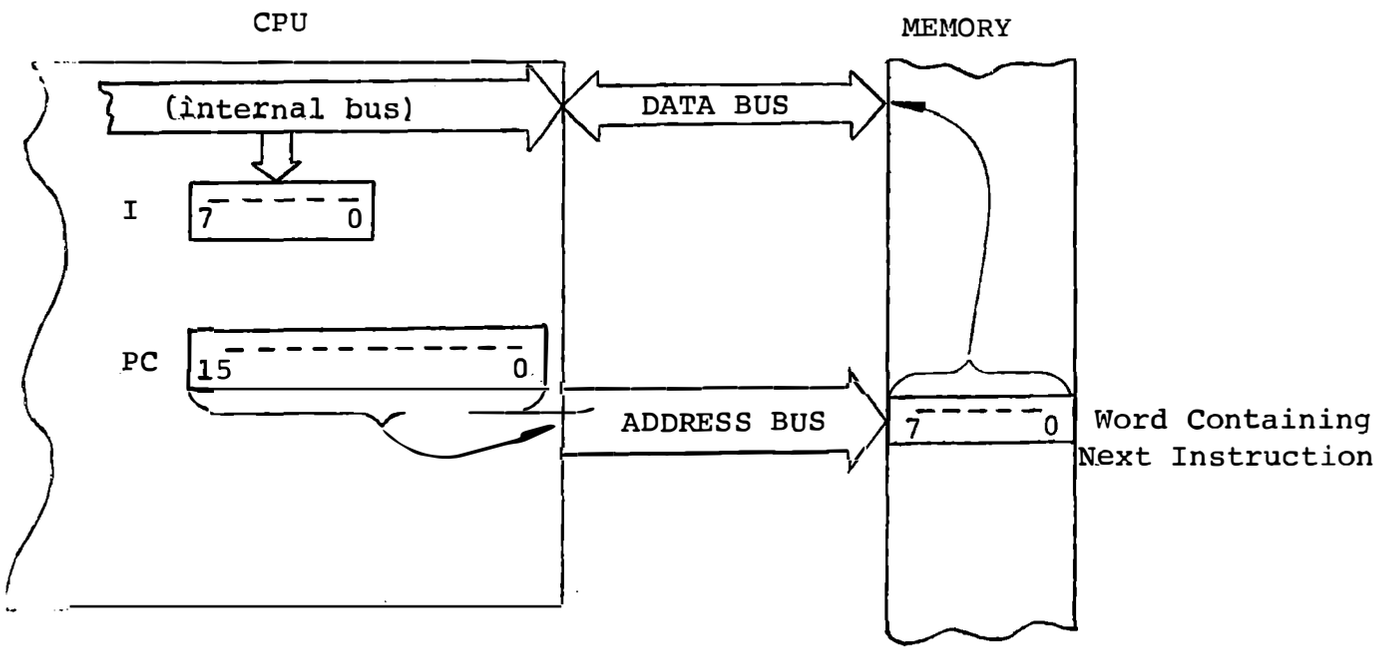
Use of the PC is illustrated below:



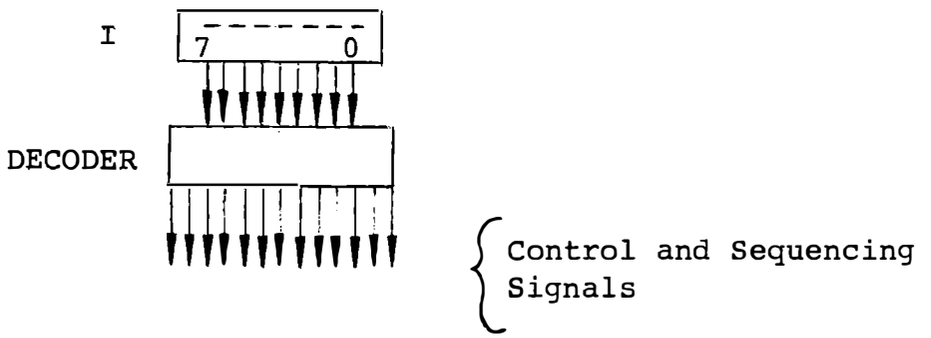
1.4.5 The Instruction Register

When a memory word has been selected by the PC, its contents will be gated onto the data bus and placed in a CPU register called the Instruction Register (I).

> INSTRUCTION REGISTER: A register in the CPU containing the instruction currently being executed.



After the instruction has been loaded in I it is fed to the instruction decoder. The instruction decoder looks at a pattern of input binary signals and outputs a pattern of signals which will sequence and control all of the steps required to execute the instruction.



1.4.6 The Accumulator

The program counter is one of the registers contained in the addressing unit. The instruction register is in the instruction unit. The final register which we will define at this point is called the Accumulator (A), an eight bit register in the arithmetic and logic unit. It is the register most actively used by programs because it contains the results of most arithmetic and logical instructions executed by the system.

1.4.7 The Clock

The computer operates in a sequential fashion, a step at a time. There must be no confusion or overlapping. Signals must be available on the appropriate lines at the right time. Many circuits are involved, each with inherent delays. Although the delays are short, on the order of nanoseconds, it does take time to access a particular device, e.g. memory, and get the response to the location required.

These delays ultimately limit the speed of operation of the computer. To ensure that each step is carried out in an orderly fashion, the process is controlled by a clock. It outputs a series of regularly spaced pulses that time all computer events. The clock frequency must be high enough to ensure rapid processing.

The upper frequency limit is set by the inherent device delays. If the frequency is too high, confusion will result because required signals will not appear in time for a particular operation. In the MTS system, there is an 8224 clock generator that uses an 8801 clock generator crystal specifically selected for the MTS 8080A microprocessor. The crystal frequency is 18.432 MHz (+0.005%). This is counted down by a factor of 9, to produce pulses at intervals of 488 nanoseconds. Thus the time for a single step in the MTS system is 488nS. Since a complete instruction may comprise about ten steps or clock periods, on the average, we arrive at an average time for an INSTRUCTION to be implemented of about 5 microseconds.

We will shortly begin active use of the Microcomputer Training System, but before doing so the system monitor provided with the MTS must be described briefly.

This page intentionally left blank.

1.5 THE MTS MONITOR

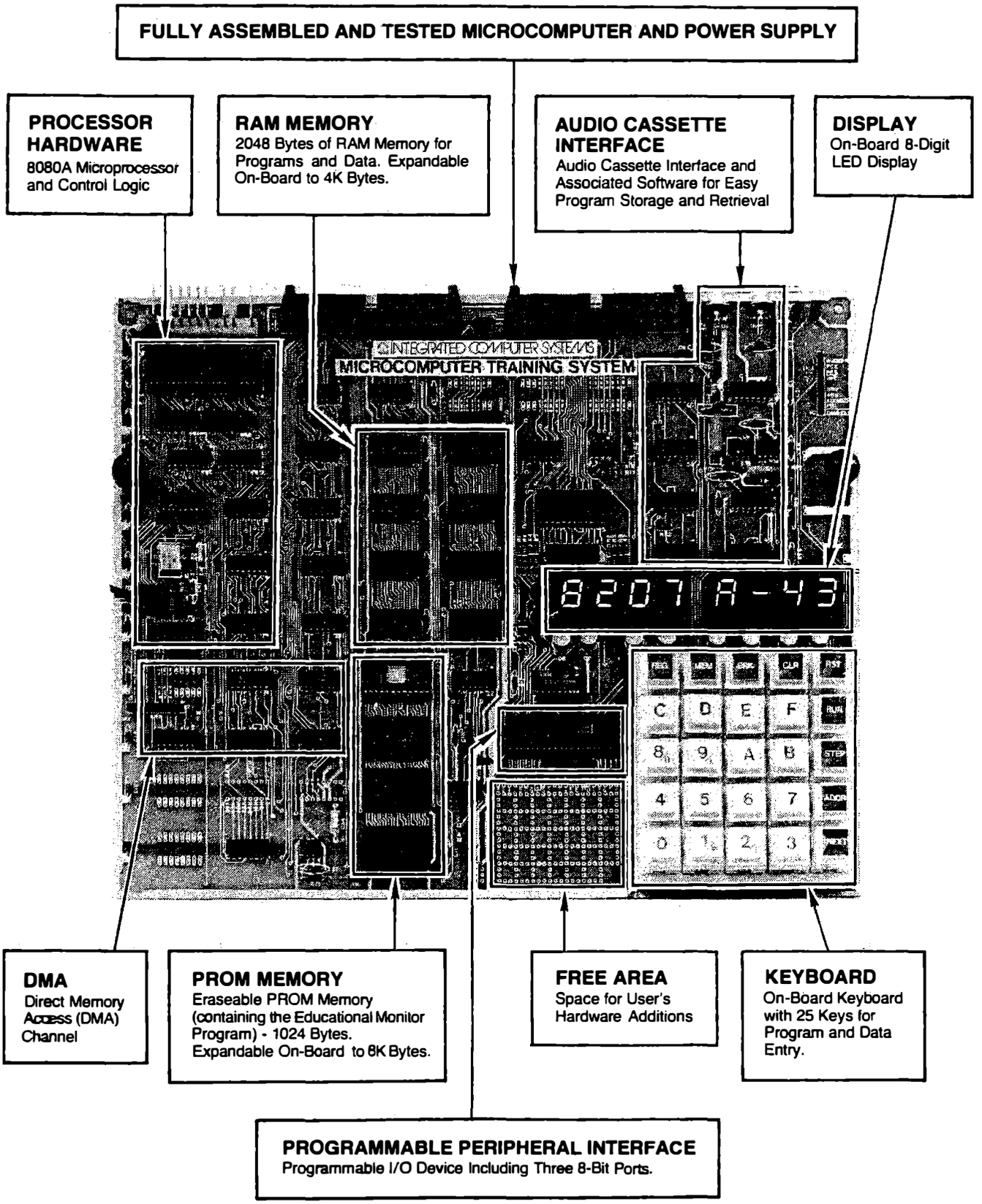
1.5.1 Monitor Software

The Microcomputer Training System has a CPU, memory (2K of RAM, 1K of ROM) and two I/O devices, a keyboard and a display (see Figure 1-3). In addition to its hardware, the MTS also has a set of programs which are stored in read-only memory. This built-in software allows you to load your own programs into the RAM memory, and to control and observe the execution of your programs. This observation function is called "monitoring", and the built-in programs in ROM memory are collectively called the Monitor.

> MONITOR: A set of programs stored in Read Only Memory, which provide for:
>
> a) Loading programs into RAM
> b) Controlling and observing the execution of programs
> c) Receiving data from the keyboard
> d) Displaying data in the eight digit display

While the monitor provides these facilities to enable you to use the MTS immediately, in later chapters you will learn to write programs for controlling the keyboard and display yourself.

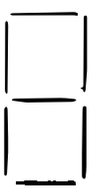
INTEGRATED COMPUTER SYSTEMS, INC.



MTS Board Layout
Figure 1-3

1.5.2 The MTS Keyboard and Display

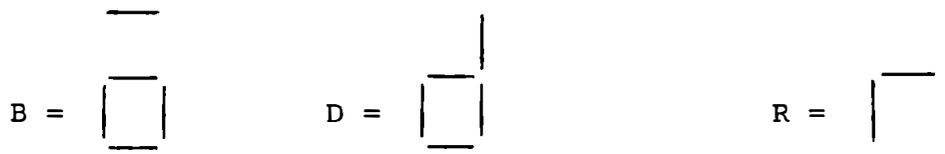
The MTS keyboard and display are shown in Figure 1-3. The display, located in the upper-right corner of the MTS, consists of two sets of four characters each. The characters are formed by sets of light-emitting diodes (LEDs). In each character position, there are eight LED elements arranged in the following fashion:



By activating one or more of the LEDs in a character position a character is formed, for example "A":

We will use initially a character set consisting of 0-9, A-F, and R. With a seven segment display, however, there are several ambiguities. The ten decimal digits are easily created, but "B" would be the same as "8", and "b" the same as "6". Also "D" would be the same as "0" and "R" the same as "A".

These characters are, therefore, represented by:



The keyboard is a five by five array. The upper row and right column of this array are command keys, each of which requests the monitor to perform a particular function. The remaining keys constitute the hex characters 0-9, A-F. For the moment we will ignore the alpha characters which appear on the 1, 2, 8 and 9 keys.

Using the keyboard and display, you will be able to:

-Inspect the contents of a memory word

-Change the contents of a memory word

-Inspect the contents of the program counter (PC)

-Change the contents of the program counter

-Inspect the contents of a register (e.g. A)

-Change the contents of a register

-Execute an instruction contained in a memory word

-Execute a program contained in memory

1.5.3 Using the MTS

When you use the monitor to control and observe execution of your programs you will be able to display and alter the content of the registers and program counter. Since the monitor is a program running in the same computer that you are using, it uses the program counter and registers itself. The information displayed has actually been stored in memory by the monitor; only when you press STEP or RUN is this information actually placed in the program counter and registers. When we refer to the program counter or to a register in this text we will generally be speaking of the values applicable to your program.

When power is turned on, the monitor will set the content of your PC to 8200, which is in RAM memory, and display this number in the left four digits of the display panel. The content of location 8200 will be displayed in the rightmost two digits. The monitor will then wait for you to depress one of the keys on the keyboard. Initially, the content of 8200 will be undefined; the contents of RAM memory are not preserved when power is turned off, and will be random when power is turned on. For convenience in writing, therefore, whenever a number is undefined we shall represent it with question marks. When power is turned on, your display will read:

Remember, the display will not actually contain question marks; it will simply be a number which the author of this manual cannot predict!

1.5.4 Inspecting Memory Contents

Having turned on the MTS, take one of the blank coding sheets provided. Note the columns labeled ADDRESS and CODE. Enter 8200 in the first column, and its content (the two rightmost digits) in the second column. We will now continue to examine the contents of the first ten words of memory. To look at the content of 8201, press the command key labeled NEXT

The display should now read: 8201 ??

Write 8201 in the first column, and its content in the second. Press NEXT again, and write down the address (8202) and its content. Continue in this fashion until the display reads 8209. You should now know the contents of the first ten words of your memory, in whatever random condition they may be.

The command key RST (for RESTART) has the same effect as turning power on: the user's PC will be set to 8200, memory address 8200 will appear in the left four digits of the display and the content of 8200 will be displayed in the rightmost two digits. If you have made an error, press RST and start over.

This page intentionally left blank.

1.5.5 Changing Memory Contents

We will now consider changing the contents of a memory word. Press [RST] . The display will read: [8200] [??]

Now press key 1 . The display will show Err . The monitor demands a command before it will accept hexadecimal data, because otherwise it does not know what was intended. By pressing the MEM (for MEMORY) key, you command the monitor to accept data from the keyboard and store it at the memory location whose address is displayed. Press [MEM] , then hex key [1] ; the display will read:

[8200] [.01]

Notice the decimal point to the left of the memory content. This indicates that data can be entered to memory. If it is not on, the monitor will not accept the data.

Press hex key [2] ; the display will read:

[8200] [.12]

Press hex key [3] ; the display will read:

[8200] [.23]

Each time a hex key is pressed, the right digit is shifted to the left, displacing whatever was there, and the new digit is entered in the rightmost position. Remember, a memory word can store only two hex characters (one byte). The monitor will allow you to press as many hex keys as you desire, but only the last two will be stored. This capability allows you to correct keying errors without the necessity of pressing another command key. To see what all of the

hex characters look like on the display, continue pressing the keys until you have seen the entire set. Finally, press hex keys [0] and [1] so that the display reads:

[8200] [.01]

Now press [NEXT] followed by hex keys [2] and [3]. The display will read:

[8201] [.23]

Pressing NEXT allows you to enter data in consecutive memory addresses, provided that MEM has already been pressed. The decimal point reminds you that MEM has been pressed.

NEXT increments by one the address displayed. After the first time you press MEM, pressing MEM again will decrement the address by one and display the memory content. This makes it easy to back up and correct an error. Try incrementing and decrementing the address with NEXT and MEM.

1.6 PREPARING A PROGRAM

You are now ready to prepare your first simple program. First, we will define the instructions which will be used. Next we will write the program down on paper. Then the program will be entered at the keyboard and verified. Finally, the program will be executed one instruction at a time, and the sequence of operations within the system will be detailed for each instruction.

Instruction codes are one-byte, 8-bit binary words represented by two hex characters. Neither the binary word nor its hex equivalent has an intrinsic meaning, so for each instruction a short two, three or four character mnemonic has been assigned. The mnemonic is a shorthand representation of the meaning or functional description of the instruction.

1.6.1 Instructions to be Used

The first instruction we will use is defined as follows:

BINARY CODE:	00000000
HEX CODE:	00
MNEMONIC:	NOP
MEANING:	No Operation. This is an instruction which does nothing at all. Its execution has no effect on any memory location or CPU register.

The chief purpose of NOP is to leave a space open in case you have to fix something - like leaving a spare pin on the edge connector of a printed circuit board. This instruction appears in the instruction set of almost every computer on the market, from huge IBM installations to microprocessors such as the one in your MTS. It is in effect a non-instruction; when a pattern of all zeroes is presented to the instruction decoder, no operation is specified.

Register A (the Accumulator) is the most important register in the CPU from the programmer's point of view, and there are a number of instructions which manipulate its contents. It is logical to consider next an instruction which sets the contents of Register A to zero.

BINARY CODE:	10101111
HEX CODE:	AF
MNEMONIC:	XRA A
MEANING:	Clear the contents of Register A (set to zero)

The mnemonic for this instruction will appear a bit strange. This is actually one of a set of logical instructions operating on the A register. The full significance of the mnemonic will become apparent when the other instructions are considered. The third instruction which will be used in your first program is one which increments (adds one) to the contents of the A register.

BINARY CODE:	00111100
HEX CODE:	3C
MNEMONIC:	INR A
MEANING:	Increment Register A (add one to the contents of Register A)

With these three instructions, you can write a program which initializes Register A with a value of zero and then successively adds one to A until it contains a specified value. Although a very simple routine, it will introduce and clarify some of the basic concepts of instruction and program execution.

1.6.2 Program Specification

Writing a program is a very structured exercise, and from the beginning you are urged to be methodical and precise about it. All programs should originate in a program specification, a written definition of what the program should accomplish. The specification for your first program is:

"Write a program which begins with a "no operation" code, then sets Register A to an initial value of zero and then, by successive increments of one, ends with the number seven in Register A."

1.6.3 Writing (Coding) the Program

The next step is to write the program down on paper, using the same notation which was used when you inspected the contents of the first ten locations of your memory. An important addition to that format, however, will be a column for comments. Programming mnemonics are so terse that simply looking at a sequence of hex codes or mnemonics will not convey the function, goal or intent of the program. Comments are used to convey this information. Writing a program is often called "coding", as it is a translation from a natural language to computer code.

Your first program, written in the recommended format, should look like Figure 1-4

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

ADDR	CODE								
8200	0	00		NOP					Dummy Operation
8201	1	AF		XRA	A				Clear Register A
8202	2	3C		INR	A				Count to 1
8203	3	3C		INR	A				2
8204	4	3C		INR	A				3
8205	5	3C		INR	A				4
8206	6	3C		INR	A				5
8207	7	3C		INR	A				6
8208	8	3C		INR	A				Count to 7
	9								
	A								
	B								
	C								
	D								
	E								
	F								
8	0								
	1								
	2								
	3								
	4								
	5								
	6								
	7								
	8								
	9								
	A								
	B								
	C								
	D								
	E								
	F								
3	0								
	1								
	2								
	3								
	4								
	5								
	6								
	7								
	8								

Figure 1-4

Remember, comments are used so that you will be able to look at a program you wrote weeks or months ago and understand what it is your program is doing. Even more important, when you are working as part of a team, they help someone else understand what your program is doing.

1.6.4 Loading Your Program in the MTS

Now that your program is committed to paper, it is time to load it in the MTS memory. First, initialize the system by pressing RST, which will establish the first entry point at 8200. The scenario should be as follows:

RST

8200 ??

Set in write mode to enter data:

MEM

8200 .??

Enter first instruction:

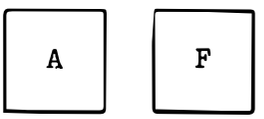
0 0

8200 .00

Advance to next instruction:

8201 .??

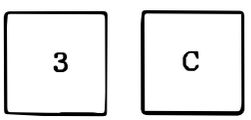
Enter second instruction.



8201 .AF

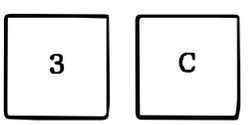
Advance to next memory address.

8202 .??



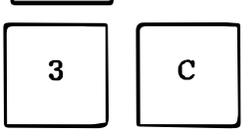
8202 .3C

8203 .??



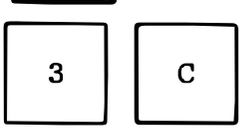
8203 .3C

8204 .??

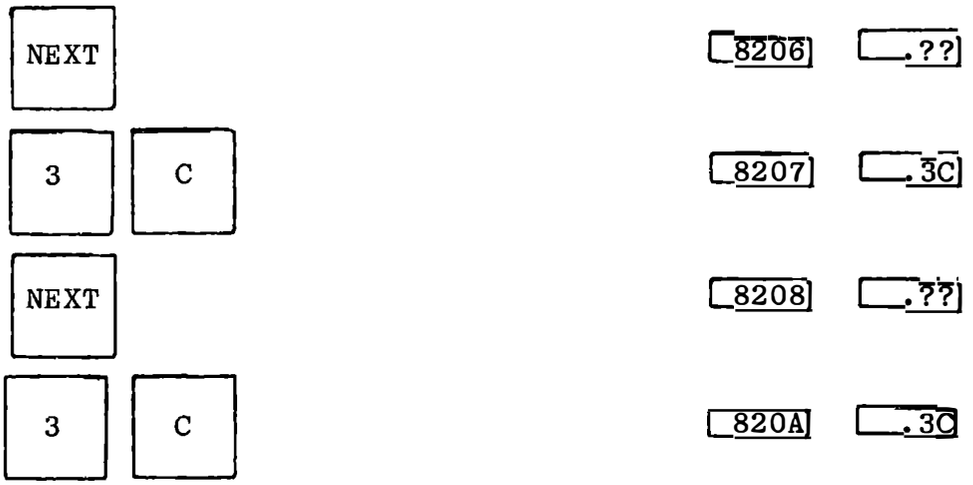


8204 .3C

8205 .??



8205 .3C



Your program has now been entered in memory.

1.6.5 Verifying and Correcting the Stored Program

Now that you have loaded your program, it will be helpful to you to verify it. It is easy to make a mistake at the keyboard, and the computer is absolutely intolerant of mistakes in the sense that it will do exactly what you tell it to do.

To be sure that your entries are correct, press RST and then, using the NEXT command, check the the contents of memory against your written coding sheet. If you detect an incorrect code in a word, it can be easily corrected, e.g.

The entry at 8205 should have been 3C. To correct it,

Corrects the error.

Inspect the next memory byte, then continue.

When you are satisfied that the program is correct according to your coding sheet, you are ready to execute the program.

1.6.6 Executing Your Program

To execute your program and follow the results of its operation on a step-by-step basis, three new commands must be introduced. These are REG , STEP and ADDR . The REG command causes the right four digits of your display to present a register name and its contents. To use the REG command, therefore, it is necessary to follow it by pressing a hex key which is the name of the register you wish to see. For the current program, we are interested only in Register A. Using the protocol developed above:

REG A 8200 A-??

The command REG followed by the hex character A leaves the address at 8200, but the right four digits identify the register (A) and its contents (undefined at this point). All of the registers will be represented in the right four digits according to the format: register name/dash/register contents.

The STEP command executes the instruction contained in the location designated by the left four-digit display (the PC). After each STEP command, the display will present the address of the next instruction. If the command REG A has been given putting the system in the "display register" mode, the contents of A will also be displayed after each instruction has been executed.

Follow this scenario on your MTS. Use your coding sheet as a guide:

RST 8200 00

Set PC to 8200 and display contents (NOP). Now display Register A.

REG A 8200 A-??

Before going on, be sure that the toggle switch at the left side of the MTS is set to STEP. Now press the STEP key.

STEP 8201 A-??

The NOP instruction has been executed and the PC has been incremented. Nothing has been done, so the content of A is still undefined.

ADDR 8201 AF

ADDR displays the current program counter and the instruction at that location. 8201 contains the instruction XRA A, clear Register A.

STEP 8202 A-00

Register A has now been cleared (it may have been empty before).

STEP 8203 A-01

Register A has been incremented. Look at your coding sheet. The instruction at 8203 is INR A.

Press STEP to execute it:

STEP 8204 A-02

Continue stepping through your program in this fashion until the PC is set at 8209. At this point, Register A should contain the number 7. If it does not, you have made a mistake either in entering your program or in pressing the command keys to execute it. If you have finished with the wrong value, inspect the memory to make sure it agrees with your coding sheet, then go through the above procedure again.

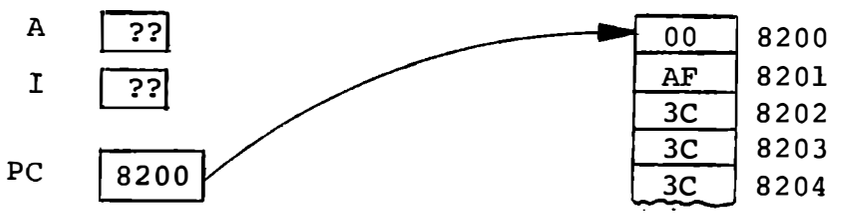
Anytime we wish to see the memory contents at a particular address, we can use ADDR . Following this by STEP causes the memory contents at that particular address to be treated as an instruction, which is carried out. The display we get depends on whether or not REG has been previously operated. If it has, we have just seen that we revert to the appropriate register display, the register contents being updated, if necessary, by the execution of the instruction carried out by the STEP command. If REG had not been operated, we would display the next instruction.

1.6.7 Instruction Execution: A Detailed Examination

We will now look at the three different instructions used in your program, describing what happens to the PC, and Registers A and I at each stage of instruction execution. Initialize the system:

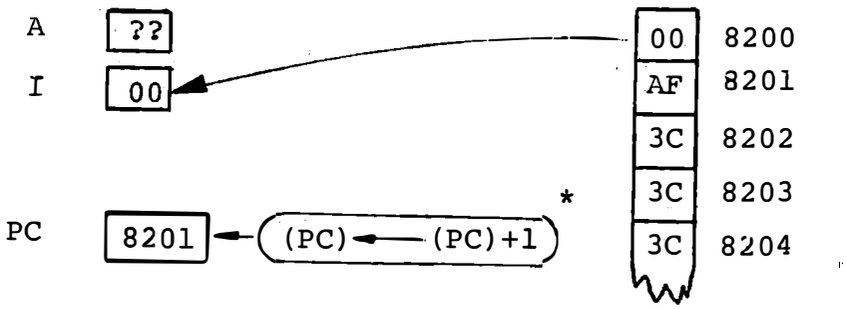
When the command STEP is issued, the following operations will occur:

1) The processor sends the contents of (PC) to memory, selecting address 8200.



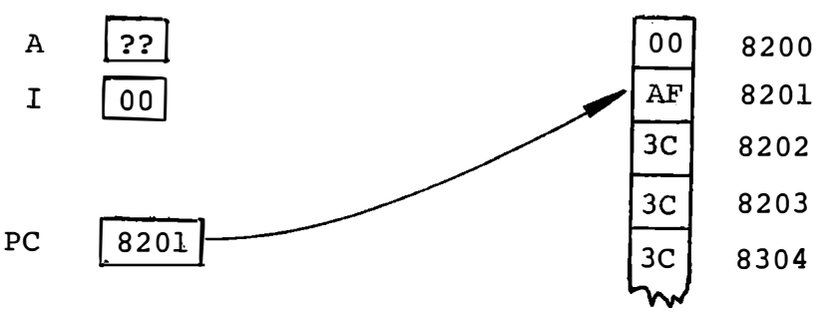
The contents of A and I are not yet defined.

2) Next, the memory sends the contents of address 8200 to the I register and PC is incremented by 1.



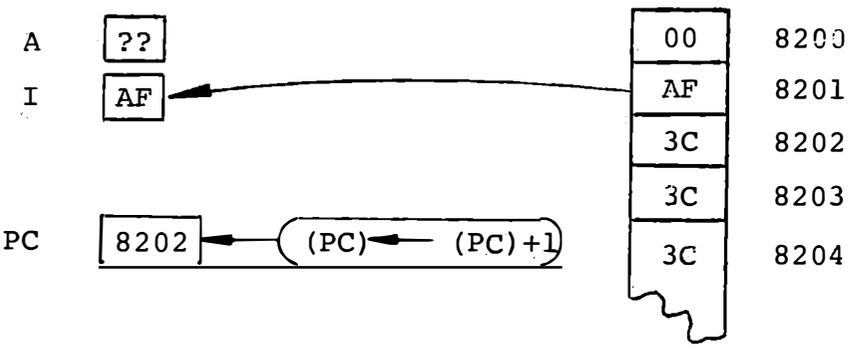
The contents of A are still undefined. The instruction is executed and as it is a NOP, the instruction cycle is completed. The next instruction will clear Register A:

1) The processor sends the contents of (PC) to the memory, selecting address 8201:

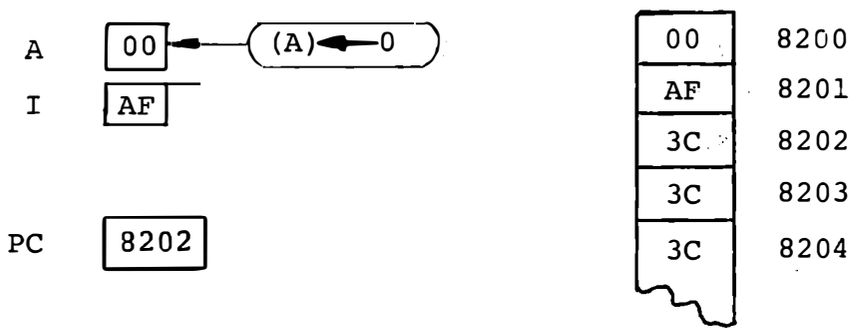


* The backward arrow (←) in an expression should be read as "is replaced by". Thus this expression reads: "The contents of PC are replaced by the contents of PC added to one".

2) The memory sends the contents of address 8201 to Register I, and the PC is incremented.

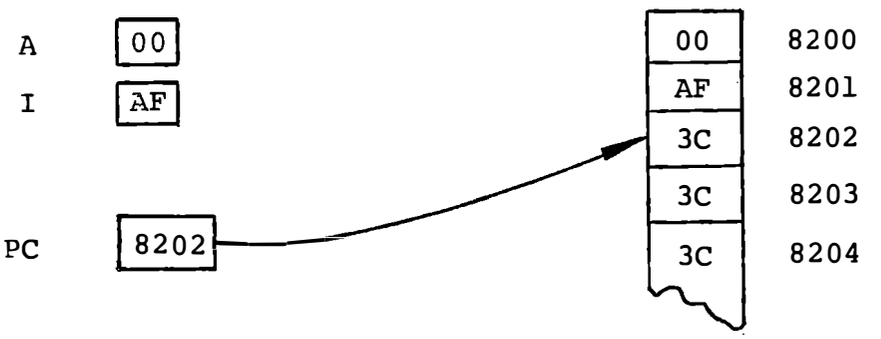


3) The instruction is executed and Register A is set to zero.

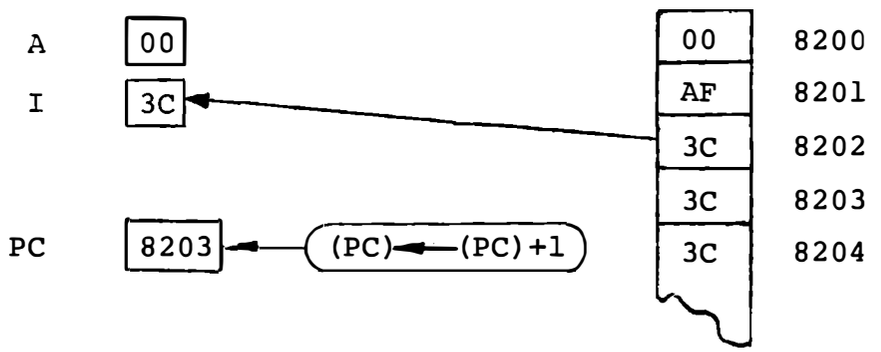


The next instruction will increment Register A:

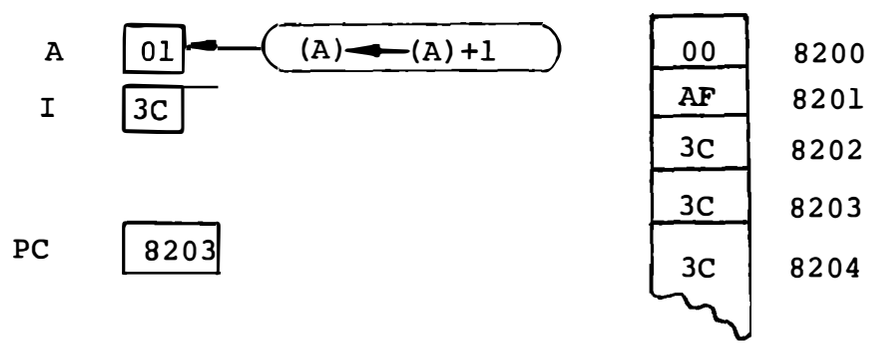
1) The processor sends the contents of PC to the memory, selecting address 8202.



2) The memory sends the contents of address 8202 to Register I, and the PC is incremented.



3) The instruction is executed and Register A is incremented by 1.



1.7 SUMMARY

This chapter has covered some very important basic concepts, both of hardware organization and function and software preparation, loading and executing. If you feel uncomfortable with any of the materials presented, go back over the relevant sections. You should now understand the functions of the following command keys. Define each of them mentally and then look at the following page.

ADDR

NEXT

MEM

REG

STEP

RST

ADDR	Displays the content of your program counter, and the hex code of the instruction addressed. It permits you to enter another address, by following ADDR with four (or more) hex keys.
NEXT	Advances to the next address for display of the memory content. NEXT does not affect your program counter.
MEM	Enables entry of data to the memory location displayed. The memory content display indicates that data entry is enabled. NEXT will advance to the next location, and data entry is still enabled. Pressing MEM repeatedly decrements the memory address. MEM does not affect your program counter.
REG	Followed by the name of a register (such as A) displays the content of that register.
STEP	Causes execution of the instruction addressed by your program counter. If STEP follows the entry of a new address by (ADDRxxxx) then that address is entered into your program counter, and the instruction located there is executed.
RST	Returns the computer to a standard condition. Your program counter is set to 8200.

MICROCOMPUTER TRAINING WORKBOOK

CHAPTER 2

TWO AND THREE BYTE INSTRUCTIONS

2.1 PROGRAM EXERCISE #2

In your first program, all of the instructions used (NOP, XRA A, INR A) were one byte instructions, fetched from memory and executed with no further memory accesses required. Many instructions comprise two or three bytes and require more than one memory access. In your next program two such instructions will be considered. Additional memory accesses are required whenever an instruction operates on data which is stored in memory, or when the results of an operation must be stored in memory.

2.1.1 The ADI Instruction

A number of instructions have the effect of adding a number to the contents of the Accumulator (A). One of these is "Add Immediate", which translates to: "Add to the Accumulator the contents of byte two of the instruction". Thus if the instruction is contained in address m, the contents of m + 1 would be added to A.

BINARY CODE:	11000110
HEX CODE:	C6
SECOND BYTE:	Data
MNEMONIC:	ADI
MEANING:	Add to the Accumulator the contents of the next memory address.

The ADI instruction requires two memory fetches, the first to get the instruction and the second to get the contents of the following word. Each memory access which is required during an instruction cycle is called a machine cycle. The instruction INR A takes one machine cycle; the instruction ADI takes two machine cycles.

MACHINE CYCLE:	The operation of accessing an address, either for reading from or writing to that address.

2.1.2 The STA Instruction

To transfer data from the Accumulator to a memory location takes even more machine cycles (before reading further, close the manual and try to determine by yourself how many cycles are required). The instruction to store the Accumulator is a three byte instruction. Bytes two and three contain the address in which the data is to be stored:

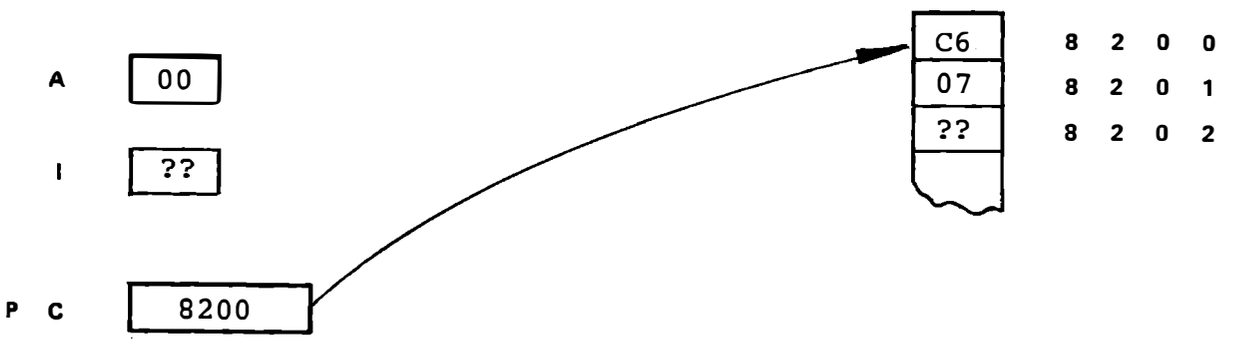
BINARY CODE:	00110010
HEX CODE:	32
BYTE TWO:	Low-order part of storage address
BYTE THREE:	High-order part of storage address
MNEMONIC:	STA
MEANING:	Store the contents of the Accumulator (A) at the address which is contained in the following two memory locations.

ADI is a two-byte instruction, STA is a three byte instruction. Their execution is more complex than the execution of the single byte instructions used in the previous program, so we will look at them in detail before using them.

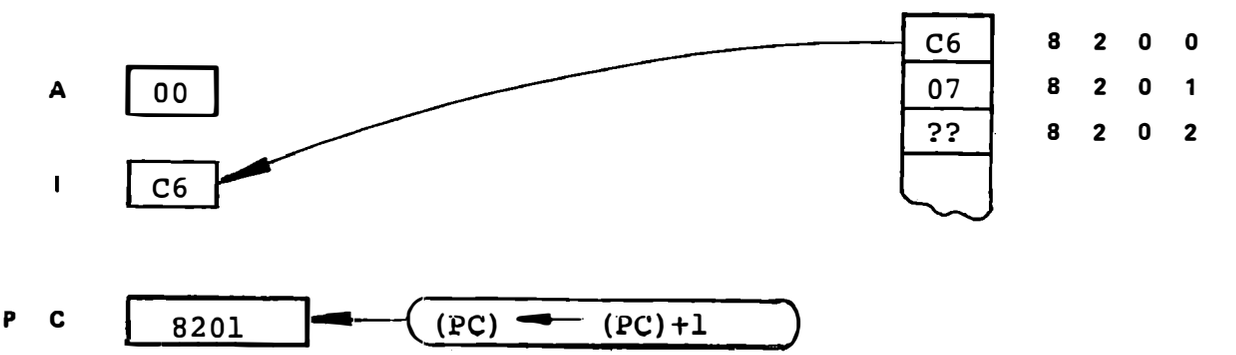
2.1.3 Instruction Execution Details

When the ADI code is fetched from memory and decoded, the logic determines that a second memory read operation is required, and that the data read is to be added into Register A. The operation looks like this:

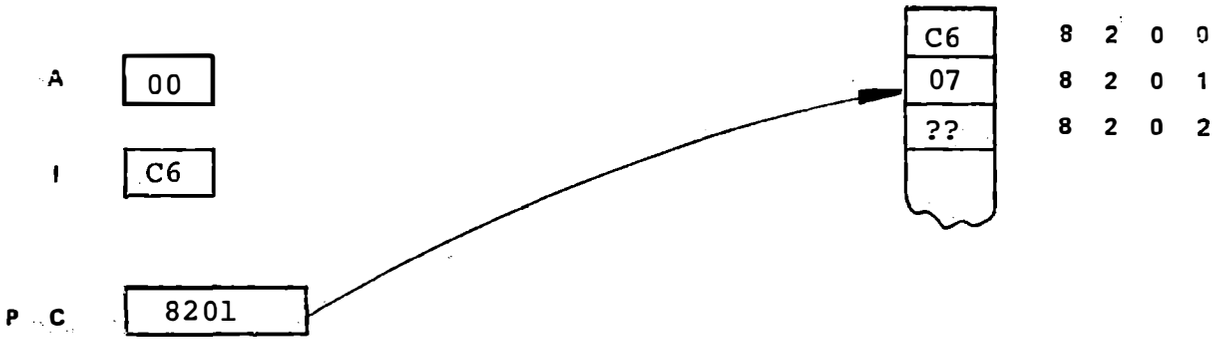
1) The processor sends (PC) to memory,
   selecting address 8200 (for this example)



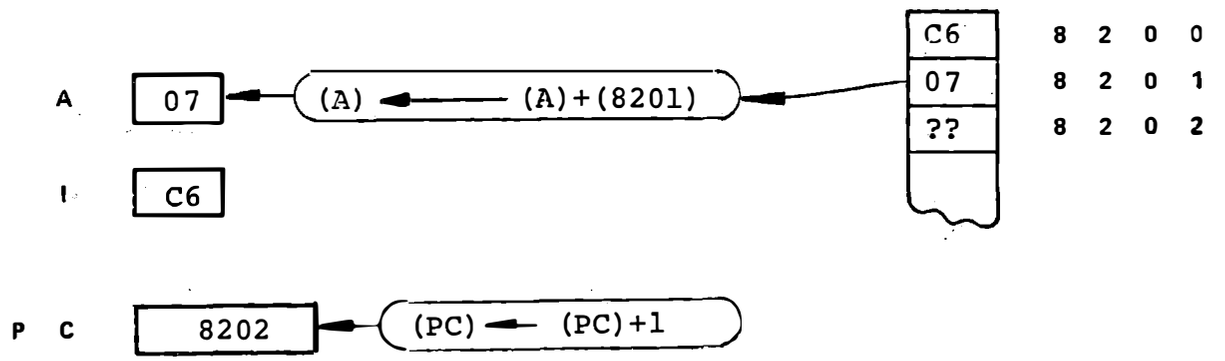
2) The memory sends the contents of address 8200
   to the I register and (PC) is incremented by 1.



3) The instruction is decoded, and the processor again sends (PC) to memory, selecting address 8201.



4) The memory sends the contents of address 8201, which is added to the contents of Register A, and (PC) is incremented by 1.



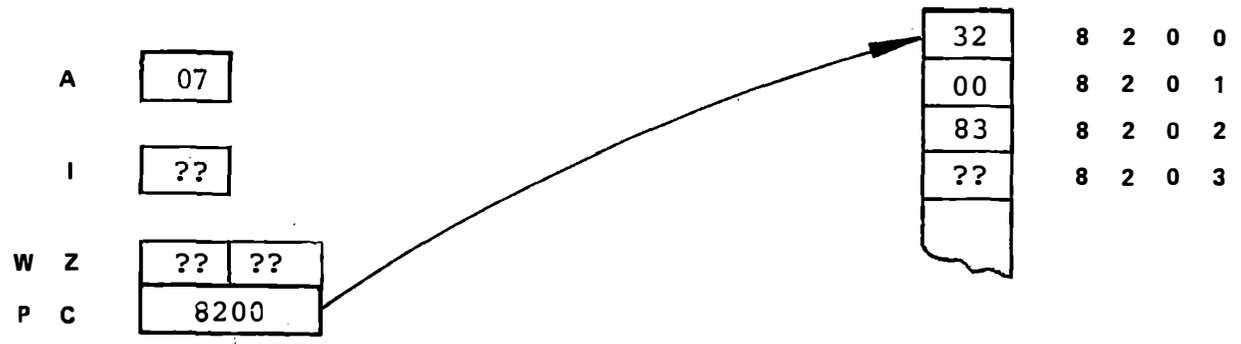
5) The instruction is completed. The memory has been accessed twice (two machine cycles), and (PC) has been incremented twice.

When the STA instruction is decoded, the logic "recognizes" that an address must be obtained from memory before the instruction can be completed, as the operation commanded is to store the contents of A in that address. The contents of the two memory words following the instruction STA must be read and stored temporarily in the processor so that they may be used. This is accomplished by the use of two registers which are called W and Z. The high-order bits of the address (most significant eight bits) are stored in W and the low order bits (least significant eight bits) are stored in Z. The sixteen bit quantity W, Z is then the address in which the contents of A will be stored. Like Register I, Registers W and Z are for internal use by the processor only and no instruction explicitly refers to them.

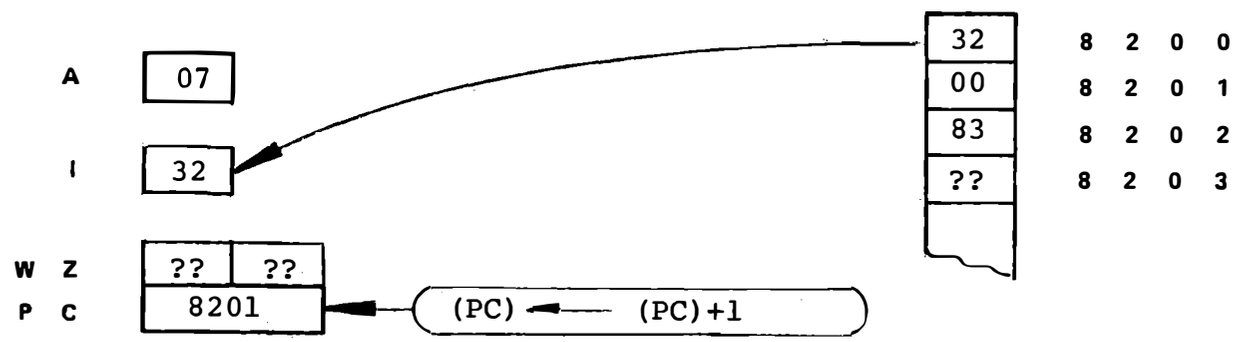
W, Z REGISTERS:	A temporary register pair in the address logic used during internal execution of instructions.

The details of execution are:

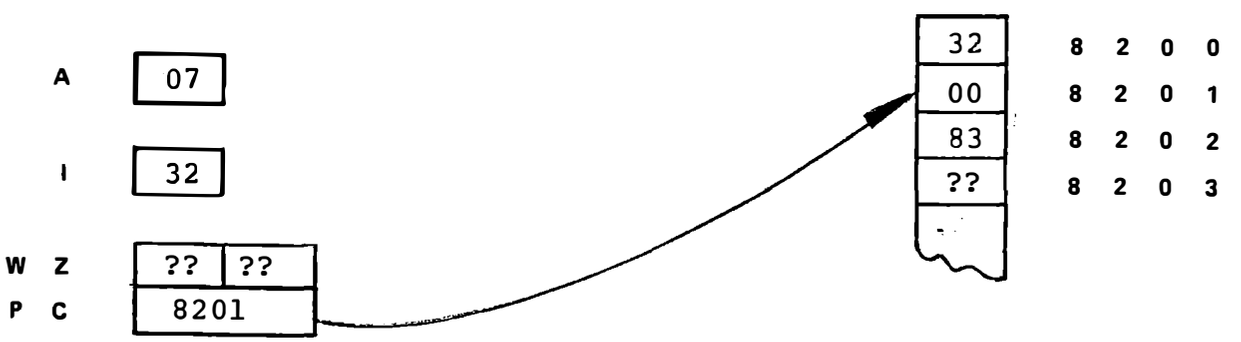
1) The processor sends (PC) to memory, selecting address 8200 (for this example):



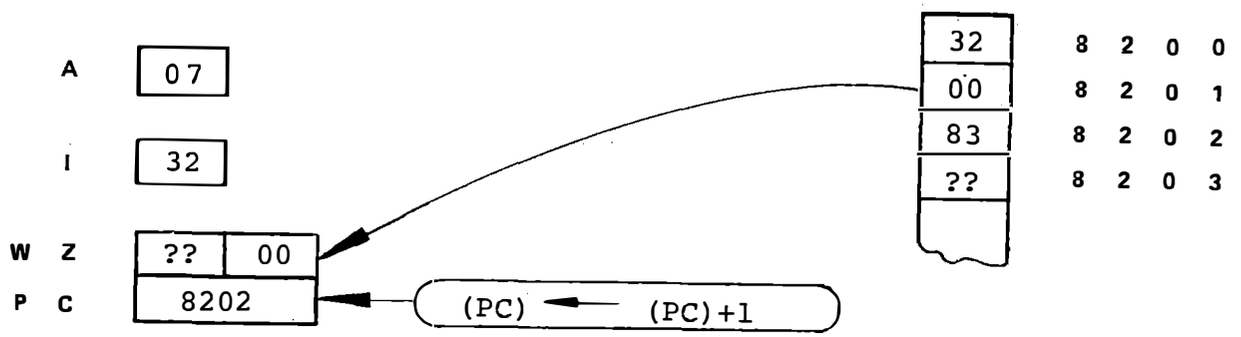
2) The memory sends the contents of 8200 to Register I and (PC) is incremented by 1.



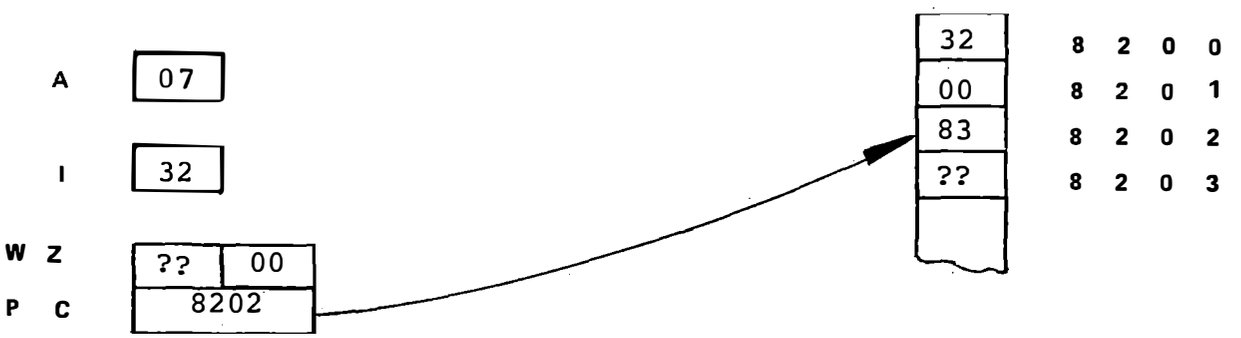
3) The instruction is decoded, and the processor sends (PC) to memory, selecting address 8201.



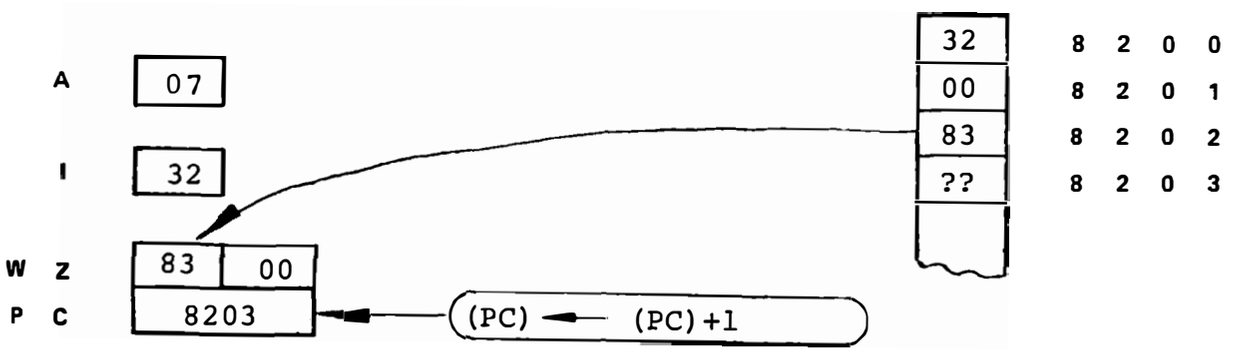
4) The memory sends the contents of 8201 to Register Z and (PC) is incremented by 1. Now Z contains the low order part of the address in which the contents of A will be stored. The design of the processor requires that the low order part of the address be stored immediately after the instruction code, followed by the high order portion.



5) Again the processor sends (PC) to memory, selecting address 8202.

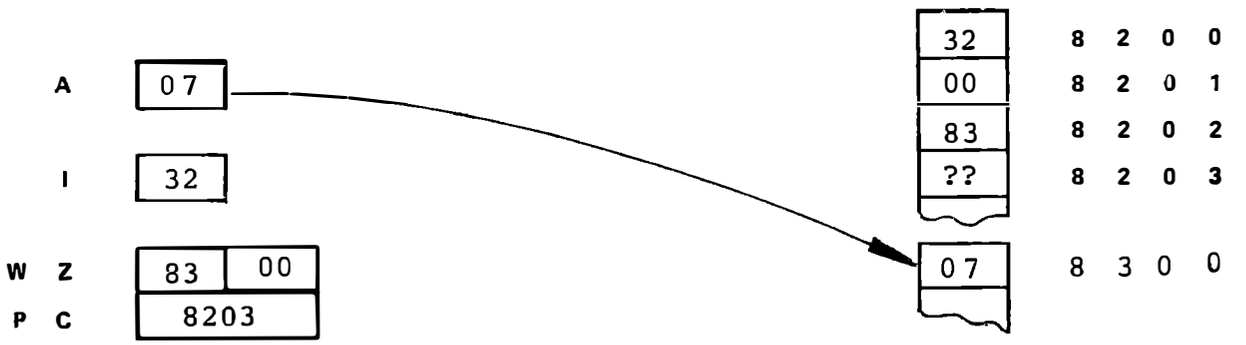


6) The memory sends the contents of 8202 to Register W, and (PC) is incremented by 1. The complete address in which the contents of A are to be stored is now available.



7) The contents of W, Z are sent to memory, selecting address 8300:

8) The processor sends the contents of Register A to address 8300 and the instruction is completed.



The execution of STA has required four machine cycles: an instruction fetch, two memory reads, and the one memory write. Do not be confused by the fact that the high and low order parts of the address in this three-byte instruction (and all similar instructions) are reversed. The arrangement was adopted by the microprocessor's designers to simplify parts of the internal circuitry.

Notice that throughout the execution of STA, the content of Register A did not change. It was duplicated in the memory location at address 8300 and remains in Register A as well.

2.1.4 Writing the Program

You are now ready to observe the behavior of these instructions in a program. As before, we start with a program specification:

> "Write a program which sets the Accumulator to an initial value of seven and then, by successive increments of one, doubles the initial value. Store the result in location 8300."

Before looking closely at the model coding sheet which follows, try to write the program by yourself.

ADDRESS	HEX	MNEMONIC	COMMENTS
8200	00	NOP	Dummy operation
8201	AF	XRA A	Clear A
8202	C6	ADI	Add immediate to A the number
8203	07		-- contained in this location
8204	3C	INR A	Increment Register A
8205	3C	INR A	
8206	3C	INR A	
8207	3C	INR A	-- continue to increment
8208	3C	INR A	
8209	3C	INR A	
820A	3C	INR A	Until (A) = $14_{10} = E_{16}$
820B	32	STA	Store result in
820C	00		location
820D	83		8300
820E	00	NOP	Dummy operation.

Note that we have included two NOP instructions that were not in the program specification. We will not normally write these into the specification but will assume that the programmer will insert them wherever he thinks it necessary, i.e., when he thinks space should be left for future program amendment.

The instruction in location 8201 clears A. This is required because ADI adds the contents of the next memory byte to A. STA operates to replace the contents of 8300 with the new value. Adding and replacing are both common operations, and the beginning programmer must be careful to distinguish them.

2.1.5 Loading and Executing the Program

Review the directions for loading a program, then enter your new program in the MTS memory. Do not forget to verify it! Before executing your program, we need to look at memory address 8300. In order to do so the command key ADDR must be introduced. Pressing ADDR will display the address contained in the PC and the contents of that address. Since RST always sets your program counter to 8200, you should see:

ADDR 8200 00

If ADDR is followed by four hex keys, the address specified by those keys will be displayed with its contents:

ADDR 8 3 0 0 8300 ??

If this sequence is now followed by [MEM] the address is now a memory address and data may be entered. As this is the address which your program will use to store a result, it would be instructive to set some arbitrary initial value, so:

[MEM] [7] [7] [8300] [77]

Memory location 8300 now contains 77, and we are ready to execute your program. Although we have addressed 8300, the program counter still contains 8200. You can test this by:

[ADDR] [8200] [00]

Only the [STEP] and [RUN] commands, or execution of your program, can change the program counter. [ADDR] always displays the current value of the program counter.

[REG] [A] [8200] [A-??]

The contents of A are undefined here.

STEP 8201 A-??

The instruction in 8200 was NOP; only (PC) changes.

STEP 8202 A-00

Looking at the coding sheet, we see that XRA A has cleared Register A.

STEP 8204 A-07

The (PC) has been stepped by two, and A contains the results of the ADI instruction.

STEP 8205 A-08

The first of the INR A instructions adds 1 to the contents of A.

STEP 8206 A-09

STEP 8207 A-0A

STEP 8208 A-0B

STEP → 8209 A-0C

STEP → 820A A-0D

STEP → 820B A-0E

Now A contains $OE_{16} = 14_{10}$; the next instruction will store this result in 8300:

STEP → 820E A-0E

The (PC) has been stepped by three and the program has been executed.

Now take a look at location 8300:

ADDR 8 3 0 0 → 8300 0E

If at any point your program execution did not produce the results described above, correct the bad instruction in your memory (If there's an error, there's a bad instruction!) and start over.

2.2 DATA STORAGE CONVENTIONS

You may have wondered why 8300 was selected as the storage location for this result. While it is somewhat arbitrary, the basic requirement is to keep programs and data separated. It would have been quite possible, for example, to store the results in location 820F. The program would execute exactly as before, except that the results would be placed in a different memory word. Suppose, however, that you wished to modify the program, to add instructions to achieve some different purpose? The program could not utilize additional consecutive addresses without changing the initial storage address. In the example, only one such address was used, but in a complex program with many storage addresses, the problem becomes acute. Data addresses are therefore chosen to leave lots of space between program and data areas.

N.B. As the monitor is stored in read-only memory, it requires part of the RAM for temporary storage of data. Sixty four bytes of RAM, addresses 83C0 through 83FF, are allocated to the monitor; care should be taken not to modify these memory locations.

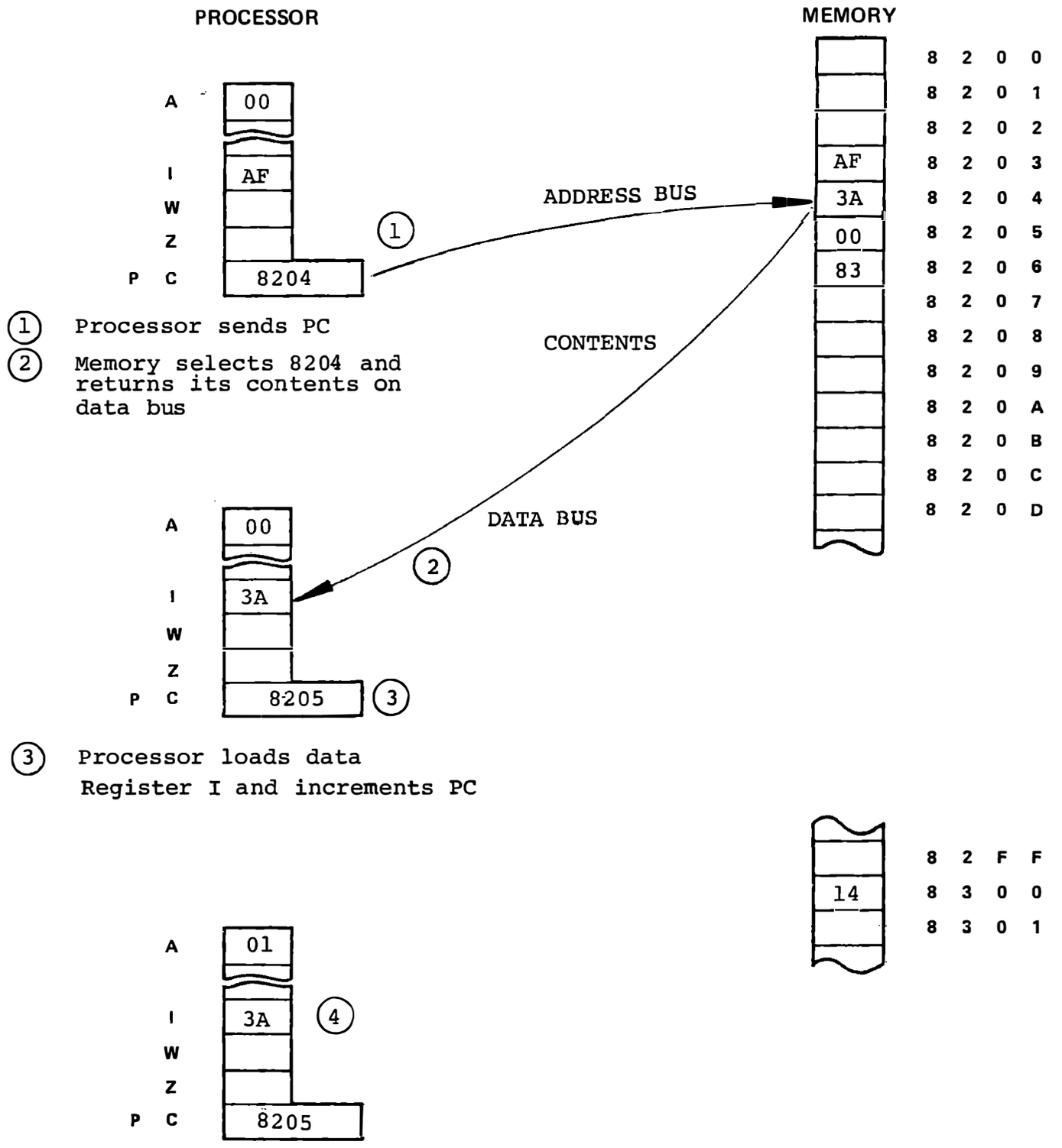
2.3 PROGRAM EXERCISE #3

2.3.1 The LDA Instructions

An instruction similar to STA has the effect of transferring data from memory to the Accumulator:

BINARY CODE:	00111010
HEX CODE:	3A
BYTE TWO:	Low-order part of address.
BYTE THREE:	High-order part of address.
MNEMONIC:	LDA
MEANING:	Load the Accumulator with the contents of the word whose address is contained in the following two memory locations.

The detailed instruction cycle for LDA is shown in Figures 2-1, 2-2, and 2-3. In these figures note the mention of the address bus and data bus. Review Section 1.3.3 and be sure you understand these buses and their functions.



(1) Processor sends PC

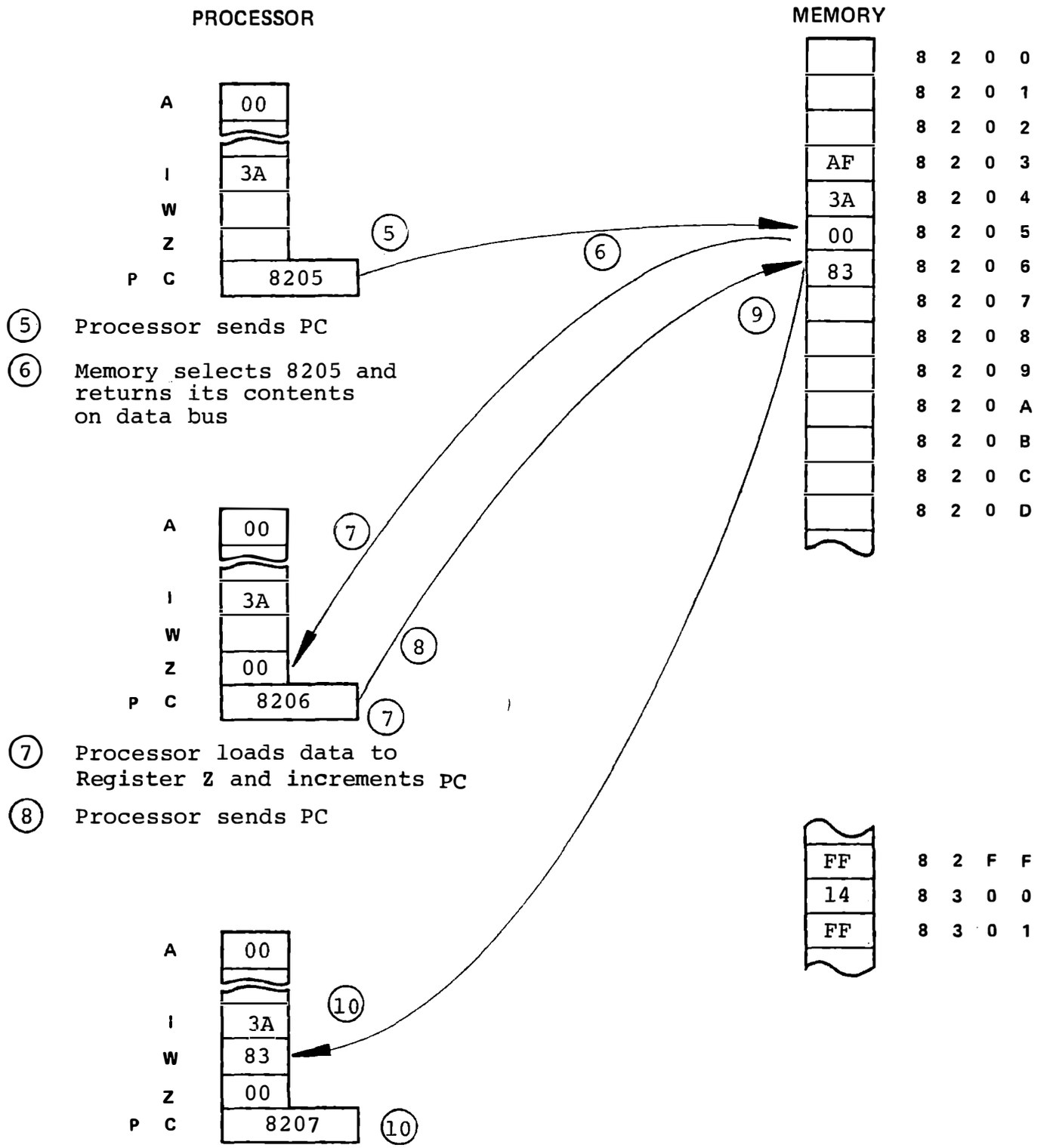
(2) Memory selects 8204 and returns its contents on data bus

(3) Processor loads data Register I and increments PC

(4) Processor interprets 3A as a three byte instruction

LDA Instruction Cycle

Figure 2-1



(5) Processor sends PC

(6) Memory selects 8205 and returns its contents on data bus

(7) Processor loads data to Register Z and increments PC

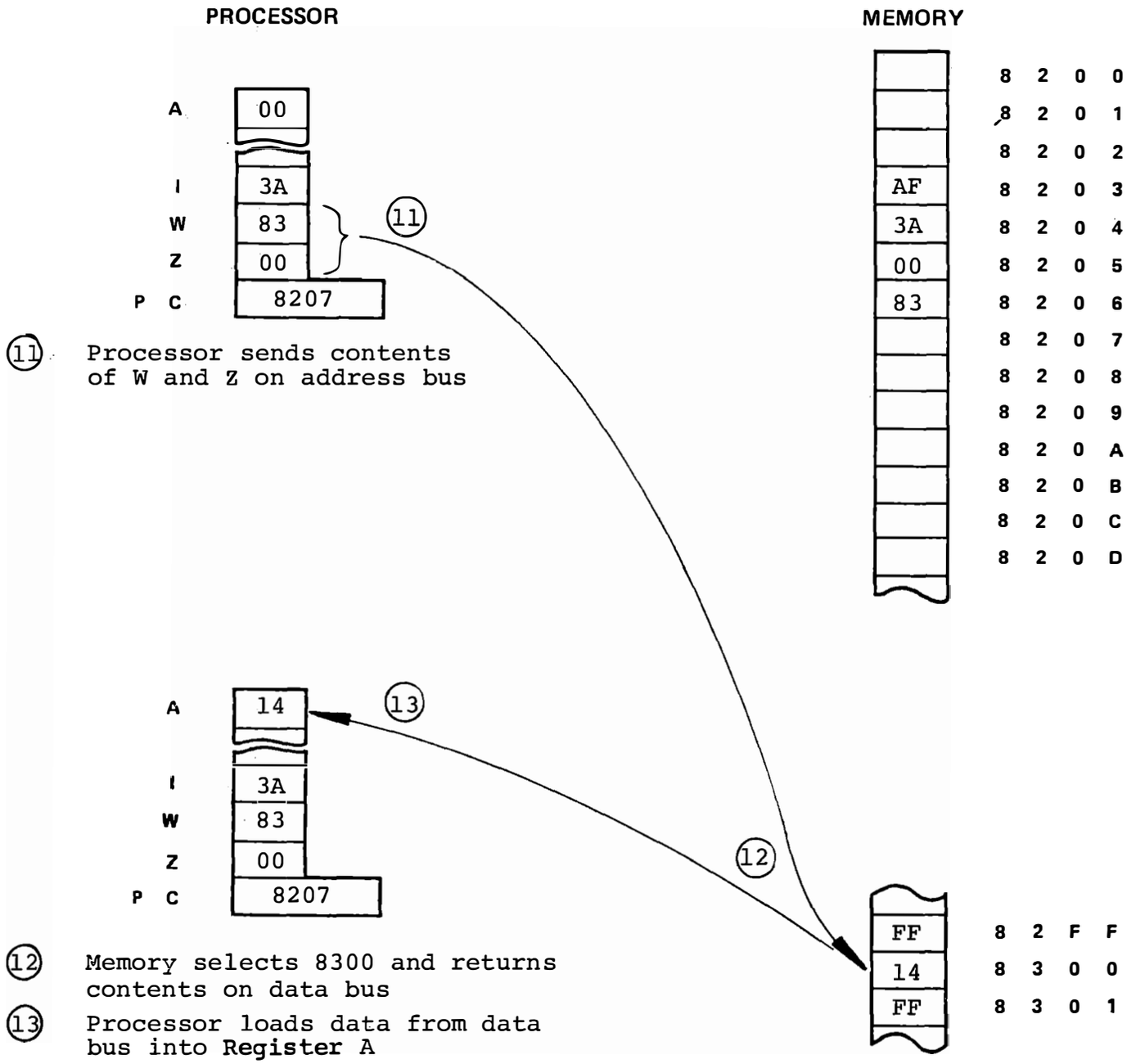
(8) Processor sends PC

(9) Memory selects 8206 and returns its contents on data bus

(10) Processor loads data to Register W and increments PC

LDA Instruction Cycle (continued)

Figure 2-2



(11) Processor sends contents of W and Z on address bus

(12) Memory selects 8300 and returns contents on data bus

(13) Processor loads data from data bus into Register A

LDA Instruction Cycle (continued)

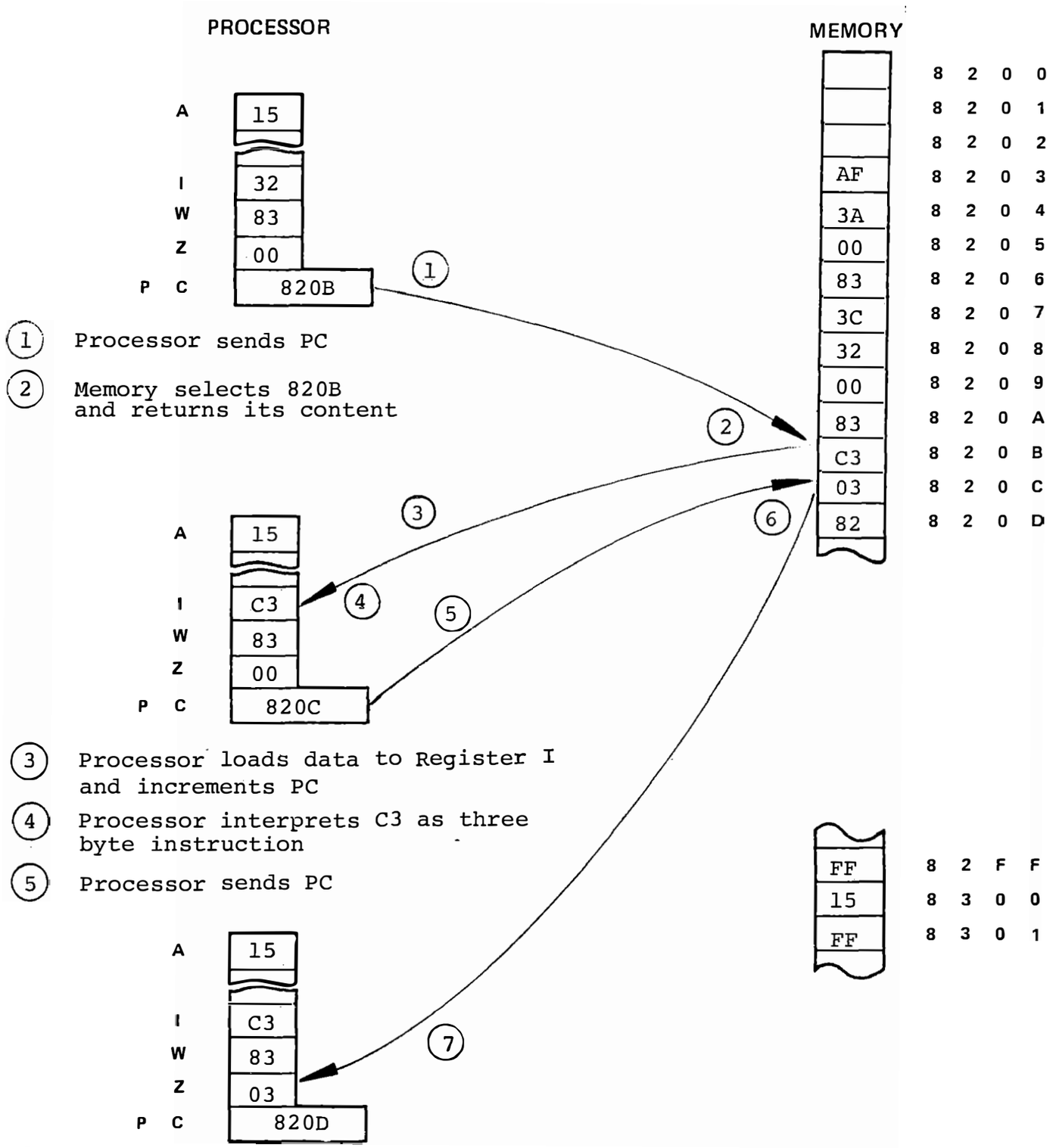
Figure 2-3

2.3.2 The JMP Instruction

To this point we have used instructions which perform an operation and advance the program counter so that it points to the address of the next sequential instruction. A very important class of instructions allows a program to branch or "jump" to an instruction at an arbitrary address. One of these instructions is JMP:

BINARY CODE:	11000011
HEX CODE:	C3
BYTE TWO:	Low-order part of address.
BYTE THREE:	High-order part of address.
MNEMONIC:	JMP
MEANING:	Load the PC with address contained in the following two memory locations.

The Execution cycle of the JMP instruction is shown in Figures 2-4 and 2-5.



(1) Processor sends PC

(2) Memory selects 820B and returns its content

(3) Processor loads data to Register I and increments PC

(4) Processor interprets C3 as three byte instruction

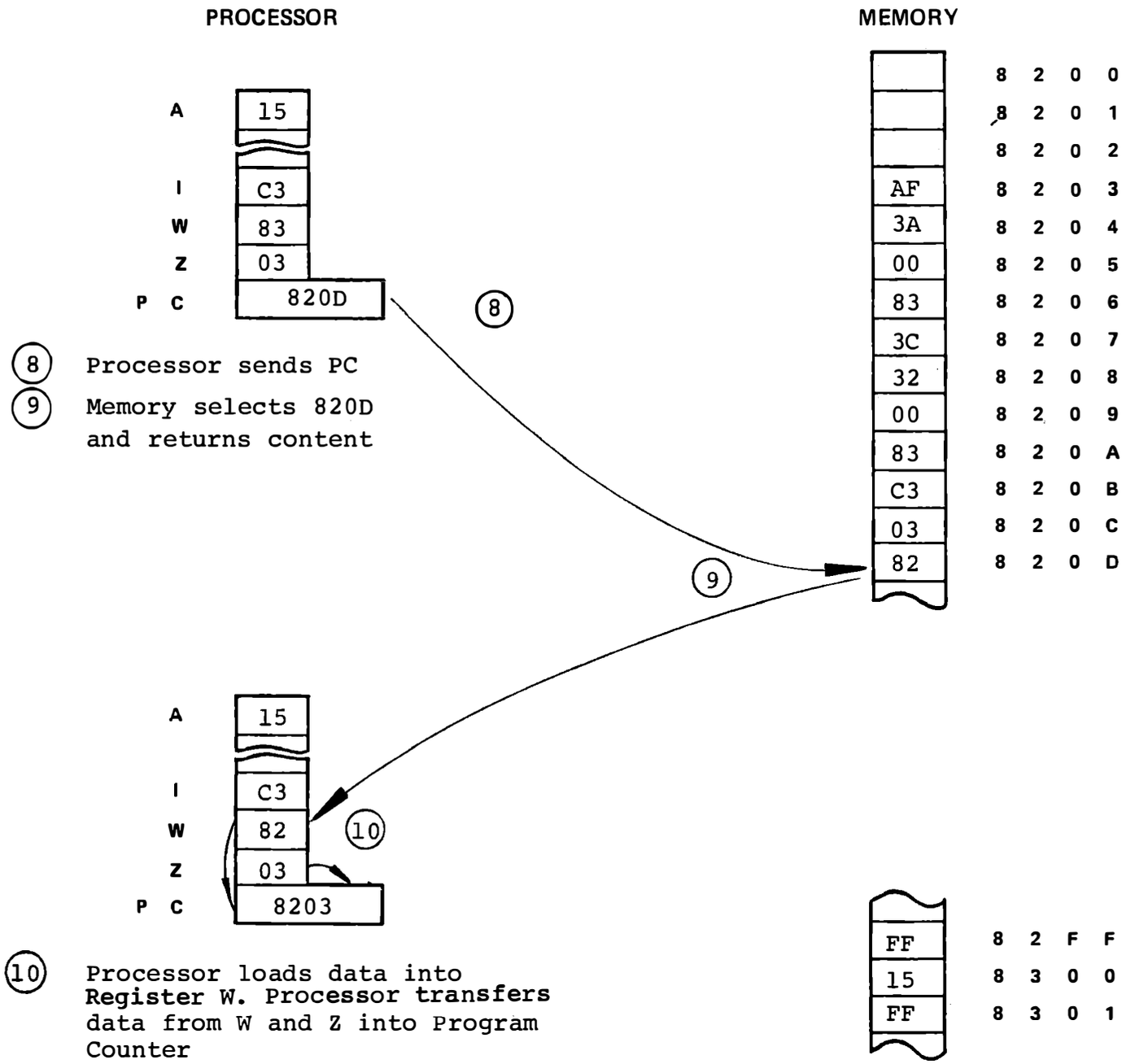
(5) Processor sends PC

(6) Memory selects 820C and returns its content on data bus

(7) Processor loads data to **Register Z** and increments PC

JMP Instruction Cycle

Figure 2-4



JMP Instruction Cycle (continued)

Figure 2-5

2.3.3 Writing the Program

Program specification:

> "Write a program which will clear the Accumulator, load it with the contents of 8300, increment this number by one, and store the result in 8300. Loop through this sequence repeatedly."

The program below starts with three consecutive NOPs, a convention which would permit entering a three-byte instruction here, should we wish to change the program later:

ADDR	HEX	MNEMONIC	COMMENTS
8200	00	NOP	Dummy
01	00	NOP	
02	00	NOP	
03	AF	XRA A	Clear A
04	3A	LDA 8300	Load A from 8300
05	00		
06	83		
07	3C	INR A	Increment A
08	32	STA 8300	Store A in 8300
09	00		
0A	83		
0B	C3	JMP 8203	Jump back to Start
0C	03		
0D	82		
8300	14		Arbitrary Data

Load and verify the program, press RST to set (PC) to 8200, then press STEP:

STEP executes the first NOP instruction and displays the next one:

Two more STEP's get us to the Clear A instruction, AF, at 8203. Execute this instruction.

We have executed Clear A. The next instruction is LDA. (3A at location 8204)

We cannot see the internal steps. The three byte instruction LDA occupies addresses 8204, 8205 and 8206. It has been executed and now the INR A instruction at 8207 is displayed.

Execute the INR A instruction.

8208 32

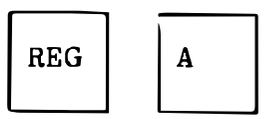
This is STA, another three byte instruction.

820B C3

We have come to the JMP instruction.

8203 AF

And now we are back to the start. Examine Register A.



8203 A-15

The program loaded 14 from 8300, incremented it and stored the new value. Register A still holds that value. Execute the Clear A instruction at 8203.

8204 A-00

Now Register A has been cleared.

8207 A-15

Now the LDA has reloaded from 8300.

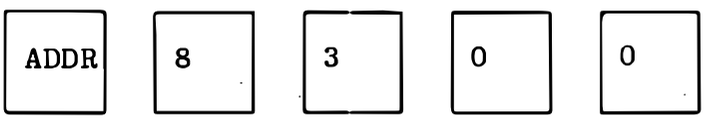
8207 3C

ADDR displays the instruction

8208 A-16

Step executes it and again displays the register we last examined.

Let's examine the memory location.



8300 15

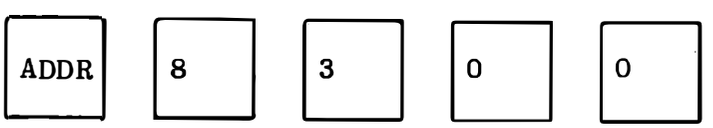
The new value has not been stored yet. DO NOT PRESS STEP NOW - The computer would execute from location 8300. Use ADDR to recall the current program counter.

8208 32

Then STEP.

820B A-16

And look again at 8300:



8300 16

Now the new value has been stored.

8300 .16

MEM tells the monitor you did not intend to change the program counter, but only the memory address. Therefore you can now use STEP. The PC contained 820B, addressing the Jump instruction.

8203 AF

So we jumped. Using the MEM key disposed of Register A display. The memory address we last requested is still there, so pressing MEM will fetch it back again.

8300 .16

We have introduced four new instructions and looked at the details of their execution cycles. The instructions are summarized in Section 2.4, and the command key functions are reviewed in Section 2.5. In Chapter 3 we will begin to develop some fundamental concepts of programming.

2.4 SUMMARY OF INSTRUCTIONS

3C	INR A	Increment Register A
		One byte
		One machine cycle
AF	XRA A	Clear Register A
		One byte
		One machine cycle
C6	ADI	Add immediate
xx	data	Two bytes
		Two machine cycles
32	STA	Store Register A
xx	low address	Three bytes
xx	high address	Four machine cycles
3A	LDA	Load Register A
xx	low address	Three bytes
xx	high address	Four machine cycles
C3	JMP	Jump
xx	low address	Three bytes
xx	high address	Three machine cycles

2.5 REVIEW OF COMMAND KEYS

ADDR Display Program Counter and Instruction. This instruction will be executed when you press STEP. Permits entry of another memory address to be examined or executed.

STEP Executes one instruction. If STEP immediately follows entry of an address, that address is entered into the program counter.

REG Must be followed by a register name (e.g. A). Displays the content of that register, and allows a new value to be entered from the keyboard.

MEM Enables entry of data to a memory location. Lights a decimal point to indicate that data entry is enabled.

If MEM directly follows ADDR, the contents of the program counter become the addressed memory location.

If MEM follows entry of an address that becomes the addressed memory location.

If MEM follows NEXT and data entry was not previously enabled, the displayed address becomes enabled for data entry.

If data entry was already enabled, MEM decrements the address.

If MEM follows REG or STEP it recalls the previously displayed memory address.

NEXT — If a memory address and its content are displayed, NEXT increments the address and stores it as the address to be recovered by MEM. NEXT does not enable or disable data entry. NEXT has other functions in monitor display modes.

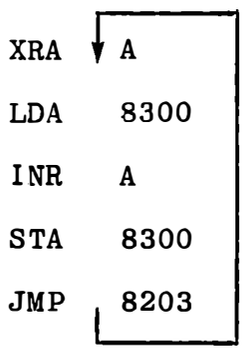
MICROCOMPUTER TRAINING WORKBOOK

CHAPTER 3

PROGRAM LOOPS

3.1 PROGRAM LOOPS AND FLOW CHARTS

The program we used in Chapter 2 was a loop:



Short loops of this kind are very common in computer programs, but they always include some means of exit from the loop. Otherwise the program would simply recycle through the loop forever, doing nothing useful.

3.1.1 The Monitor Run Command

To this point you have used the STEP command to execute your programs. Each time STEP is pressed, the instruction pointed to by your PC is executed, after which the monitor is re-entered so that it may activate the display and wait for your next command.

When the RUN command is issued, the monitor is also re-entered after your instruction is executed. However, instead of waiting for your command, it immediately allows your next instruction to be executed. To demonstrate this, make sure that your program loop is still in memory.

If you press RUN to execute this loop, the display will disappear and nothing more will happen. Internally, the count at

location 8300 is being incremented again and again, but you have no way of knowing what is happening. The keyboard is dead. Only the RESET key (or the power cord) can interfere. There must be some means of leaving such a closed loop.

In a sense, all computer programs are loops: they must somehow return and repeat the same instructions, but operating on different data, producing different outputs, and sometimes executing different sections of the program depending on the data.

This chapter presents the conditional jump, an instruction that alters the program flow as a function of the data. This is the most common way of exiting from a short loop. The flow chart is introduced, which describes the problem flow and is the principal design tool for programming. Finally, another method of entering the monitor for input and output will be provided.

3.1.2 The Conditional Jump

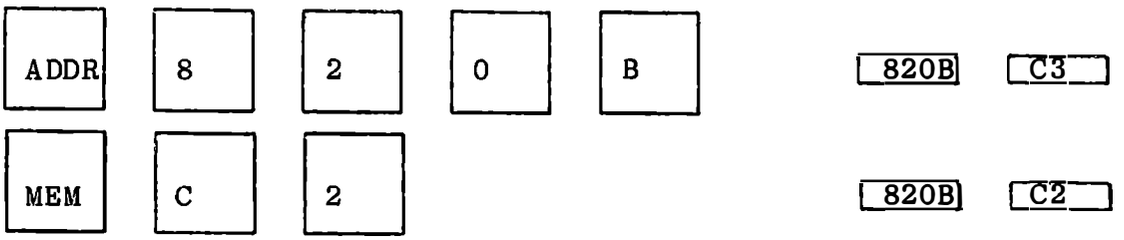
When certain instructions generate a zero result, a special "Flag" flip flop is set. This condition is displayed by the bottom LED labeled "Z" at the left of the numeric display. You will have seen this turn on each time XRA A was executed in the previous exercises (if not, try it now). When INR A causes a non-zero result, this LED is turned off. In the program loop above, Register A is repeatedly incremented. Once every 256 loops the content of A goes from FF to 00, setting the Zero flag. During the other 255 loops, the Zero flag is not set. The condition of this flag can be sensed and acted upon by the instruction "Jump if Not Zero".

BINARY CODE:	11000010
HEX CODE:	C2
BYTE TWO:	Low-order part of address.
BYTE THREE:	High-order part of address.
MNEMONIC:	JNZ
MEANING:	Jump to the address contained in the following two words if the result of the last counting, arithmetic or logical operation was not zero.

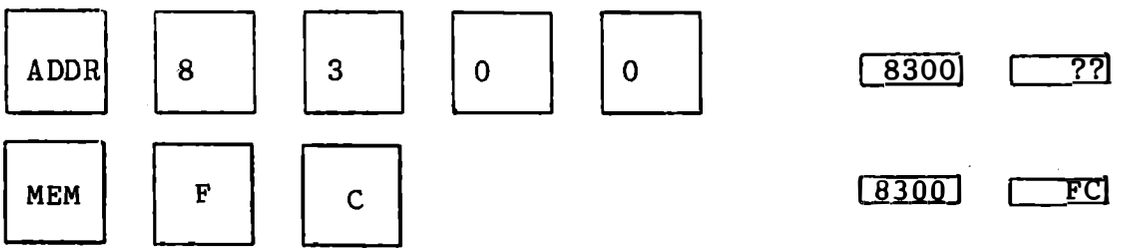
We will now modify the program loop above by replacing the jump instruction with the conditional jump, as follows:

8203	AF	XRA	A
8204	3A	LDA	8300
8205	00		
8206	83		
8207	3C	INR	A
8208	32	STA	8300
8209	00		
820A	83		
820B	C2	JNZ	8203
820C	03		
820D	82		

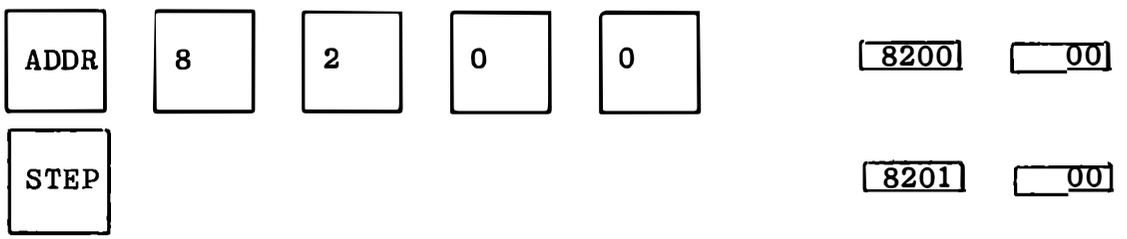
Change the jump instruction by pressing:



Since the jump address for the JNZ instruction is the same as for the old JMP, it need not be reentered. To avoid going through the loop many times, set a high value, say FC, into address 8300. Then step through the program:



Now go back to the beginning and step.



Request display of Register A,

and step through the program, watching Register A.

STEP 8202 A-??

STEP 8203 A-??

STEP 8204 A-00

THE XRA A instruction at 8203 has cleared A. The Zero flag should now be set.

STEP 8207 A-FC

The LDA instruction at 8204 has loaded A with the data from 8300. The Zero flag does not change.

STEP 8208 A-FD

INR A done. The result was non-zero, so now the Zero flag is cleared.

STEP 820B A-FD

(STA done)

STEP 8203 A-FD

(JNZ done)

Continue stepping until you see:

8207 A-FF

(LDA done)

8208 A-00

INR A done. Register A has been incremented from FF to 00. The Zero flag is now set, indicating that when you reach the JNZ it will not be executed.

820B A-00

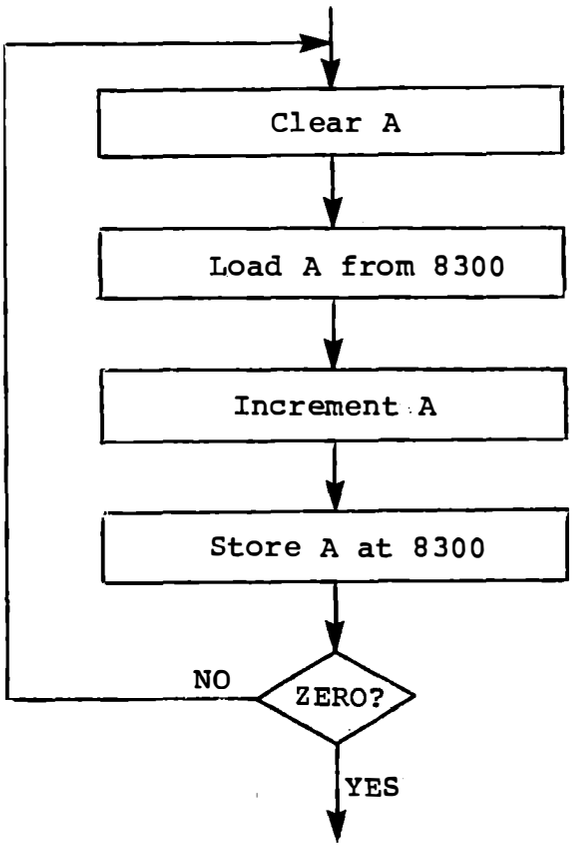
(STA done)

820E A-00

Since the INR A instruction at 8207 has incremented the value to 00, the JNZ instruction at 820B did not result in a jump. The three machine cycles were still performed, loading I, Z and W with the three bytes of the instruction and incrementing the program counter three times. At the final step, however, the logic unit tests for zero and sees that the condition for jumping is not met -- the result was zero -- and so does not transfer W and Z into the program counter. Execution continues from the previously incremented contents of the program counter to the next sequential instruction.

3.1.3 Flow Charts

A flow chart shows this operation in the following fashion:



The diamond shape represents a program branch conditioned by data. The branch to be followed depends on the results of the previous operations.

Flow charts represent the design of computer programs; they may be considered the equivalent of schematics in electronic design. Writing the final program is akin to the circuit board layout - the

function is fully defined but there is still some degree of freedom for the designer. From here on, each exercise will either include a flow chart or ask you to prepare one.

FLOW CHART:	A symbolic representation of the logical steps of a program, detailing control and sequencing of the flow of data, procedures to be followed, computations to be performed, and input/output operations.

The flow chart above shows an incomplete program. If you continue to step after passing the JNZ instruction, you will execute an unintended instruction at location 820E. A closed loop such as we started with has no value since it accomplishes nothing but merely repeats itself. An open loop is intolerable because it will have unintended results.

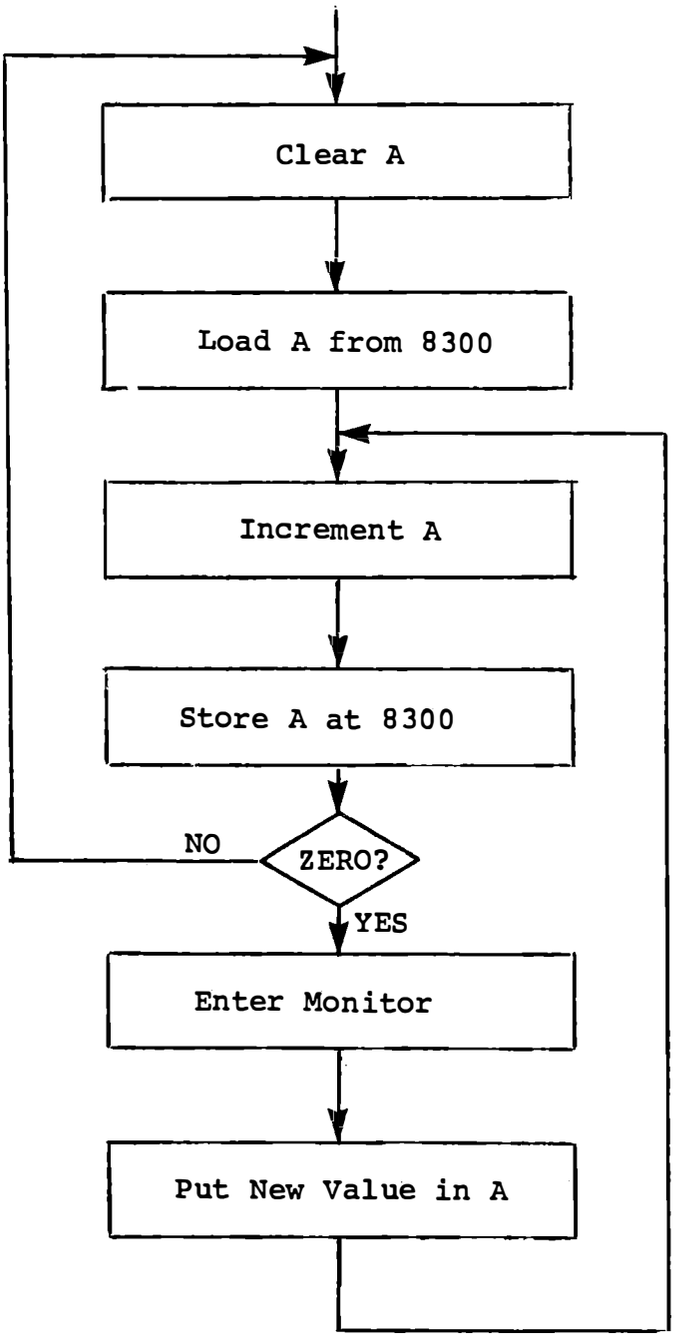
The purpose of the computer is to provide outputs depending on inputs. We have been obtaining outputs by looking at Register A contents after each step. You provided one input by loading data to address 8300. You could also change the data in the A register by a monitor command, but this is only effective at certain points in the program, since Clear A and Load A will destroy anything you enter. What we need is a means of entering data only at a certain position in the program.

3.2 PROGRAMMED MONITOR ENTRY

It is possible to activate the monitor from your program, instead of from the keyboard. The command is:

BINARY CODE:	11100111
HEX CODE:	E7
MNEMONIC:	RST4
MEANING:	Restart the monitor at entry point four.

When this command is executed, all of the monitor functions become available to you. This allows you to use the RUN command, but permits your program to enter the monitor where you wish it to do so. Now you can modify your program to provide additional inputs. Consider the revised flow chart in Figure 3-1.



Conditional Jumps

Figure 3-1

To implement the program, make the following changes to your code:

820E	E7	RST4	Enter the monitor
820F	C3	JMP	Jump to the "INR A"
8210	07		instruction.
8211	82		

Once again load a large value at 8300, then set the address to 8200 and step through the program.

When the address display shows:

0020 F3

(or) 0020 A-??

you have entered the monitor. Step again and your jump instruction will appear. Now try RUN . Each time you press RUN the display will go blank briefly while the computer counts to FF and 00, and then it will re-enter the monitor. Now press:

REG A 820F A-00

Register A has reached 00, the zero flag is set, and the program counter points to the jump instruction.

F 0 820F A-F0

you have entered a large value to Register A.

RUN 820F A-00

This time the display should barely blink, because the program only looped 16 times instead of 256.

This exercise illustrates the way in which timed delays may be implemented using program loops, a feature which is common in many process control operations.

3.3 ADDITION BY COUNTING

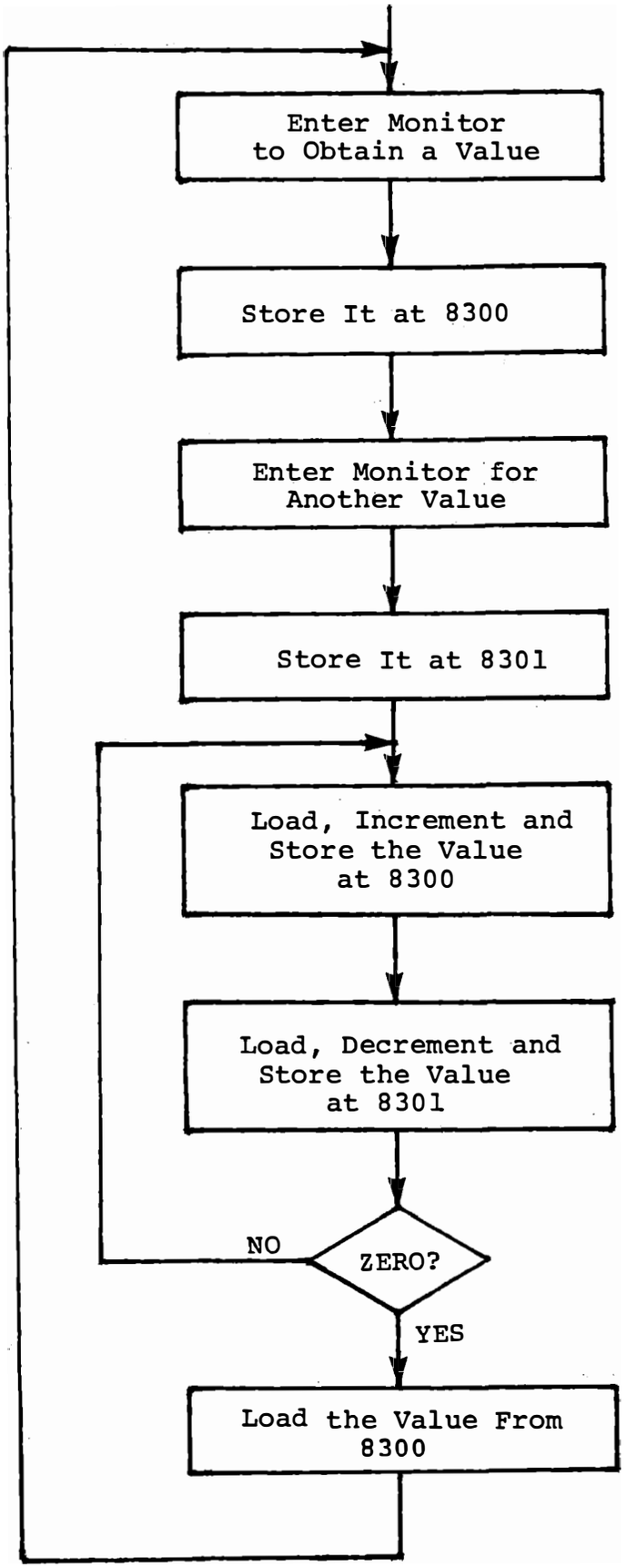
The next program exercise will demonstrate finding the sum of two numbers by the basic principle of counting. The program specification is:

> "Write a program which will form the sum of two numbers by successively incrementing the first number and decrementing the second, until the second reaches a value of zero."

To implement this program a new instruction will be required:

BINARY CODE:	00111101
HEX CODE:	3D
MNEMONIC:	DCR A
MEANING:	Decrement Register A

A flow chart for the program will be helpful and one is presented in Figure 3-2. Before looking at the coding sheet (Figure 3-3) try to write this program all by yourself, then match it against the one provided.



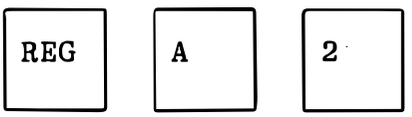
Addition by Counting

Figure 3-2

CODING SHEET — MICROCOMPUTER TRAINING SYSTEM — INTEGRATED COMPUTER SYSTEMS

ADDR	CODE										
8 2 0 0	0	0	N	O	P						Save three bytes
1	0	0	N	O	P						for a future change
2	0	0	N	O	P						
3	E	7	R	S	T	4					Enter monitor and
4	3	2	S	T	A		8	3	0	0	save the value
5	0	0									returned in A at 8300
6	8	3									
7	E	7	R	S	T	4					Enter monitor and
8	3	2	S	T	A		8	3	0	1	save the value
9	0	1									returned in A at 8301
A	8	3									
B	3	A	L	D	A		8	3	0	0	Begin loop
C	0	0									Load first value
D	8	3									
E	3	C	I	N	R		A				Increment and
F	3	2	S	T	A		8	3	0	0	store the first
8 2 1 0	0	0									value
1	8	3									
2	3	A	L	D	A		8	3	0	1	Load the second
3	0	1									value
4	8	3									
5	3	D	D	C	R		A				Decrement and
6	3	2	S	T	A		8	3	0	1	store the
7	0	1									second value
8	8	3									
9	C	2	J	N	Z		8	2	0	B	Loop until second
A	0	B									value is zero
B	8	2									
C	3	A	L	D	A		8	3	0	0	Exit from loop
D	0	0									Load the first
E	8	3									value and
F	C	3	J	M	P		8	2	0	3	go back to
8 2 2 0	0	3									monitor to
1	8	2									display it.
2											
3											
4											
5											
6											
7											
8											Figure 3-3

Before stepping through your program, press RST and then enter a small value in A:



8200 A-02

8201 A-02

Now press STEP repeatedly:	8202	A-02
	8203	A-02
You have just entered the monitor.	0020	A-02
Continue to STEP:	8204	A-02
	8207	A-02
You have entered the monitor again.	0020	A-02
Continue to STEP.	8208	A-02
This is the beginning	820B	A-02
of the loop. Continue to step.	820E	A-02
You have done the first INR A.	820F	A-03
The first value has been stored.	8212	A-03
The second value, also 2, has been loaded,	8215	A-02
decremented	8216	A-01

and stored.	8219	A-01
The program is now at JNZ, the result is not zero, and the jump occurs.	820B	A-01
The first value is loaded,	820E	A-03
incremented,	820F	A-04
stored,	8212	A-04
the second value is loaded,	8215	A-01
decremented (and the Zero flag is set),	8216	A-00
stored. The program is again at JNZ but the jump does not occur.	8219	A-00
	821C	A-00
The first value is loaded and now the jump	821F	A-04
back to the beginning occurs.	8203	A-04
The monitor again.	0020	A-04
Step again. Back to your program with A unchanged.	8204	A-04

As the initial value placed in A (2) became the value of both the first and second numbers, we can verify that the result (4) is in fact their sum.

Now press RST and run your program for various pairs of numbers. Remember each instruction takes only a few microseconds; the display will not even blink. Press RUN, then REG A (PC will be 8204) and enter the first number. Press RUN, REG A (PC will be 8208) and enter the second number. Press RUN again. The result will be displayed, and you can key in a new pair. Any two numbers whose sum is less than or equal to 255 (=FF hex) can be added in Register A.

3.4 EXERCISE

The program we have developed enters the monitor twice to accept two numbers to be added together. The sum is displayed (in Register A) and two more numbers are entered. Modify the flow chart of Figure 3-2 so that after a sum is displayed only one new number is entered, and that number is added to the previous sum. With the modified program you can sum a column of numbers.

3.5 SUMMARY

In this chapter several new instructions have been introduced, the use of RUN and programmed monitor entry has been shown, and the important concept of flow charts has been presented. All of the instructions used so far are summarized in Section 3.6. You may wish to write a program of your own at this point, for practice. If you do, follow the rules:

a) Specify the program
b) Draw the flow chart
c) Select memory areas for the program and for data (Do not use locations 83C0 - 83FF)
d) Write the code, with comments
e) Key in the code and verify it
f) Step through the program to check it, then run it

3.6 SUMMARY OF INSTRUCTIONS

00	NOP	Do nothing
AF	XRA A	Clear Register A
3C	INR A	Increment Register A
3D	DCR A	Decrement Register A
3A	LDA	Load Register A
XX	low address	with the data stored
XX	high address	in the memory location whose address is in the second and third bytes.
32	STA	Store the contents of
XX	low address	Register A in
XX	high address	the memory location whose address is in the second and third bytes.
C3	JMP	Jump to the location
XX	low address	whose address is in the
XX	high address	second and third bytes.

C2	JNZ	Jump if the result of
XX	low address	the last arithmetic
XX	high address	operation was not zero; otherwise continue to the next sequential instruction.
E7	RST4	Enter the monitor.

MICROCOMPUTER TRAINING WORKBOOK

CHAPTER 4

THE OTHER REGISTERS

4. THE OTHER REGISTERS AND MEMORY ADDRESSING

In this chapter we introduce the general purpose Registers B, C, D, E, H and L. These registers are used for:

1) Temporary data storage
2) Storing operands for arithmetic and logical operations
3) Counting
4) Memory addressing

For temporary data storage and counting, the general purpose registers are equivalent to Register A. There are instructions for all seven registers permitting data to be moved among them, moving data into them from memory, moving data from them into memory, incrementing and decrementing their contents. They are not identical in all functions, however, and each has certain unique features. Register A, or accumulator, is very different in that the results of most arithmetic and logical operations are stored in Register A. Similarly, input/output instructions use Register A.

4.1 THE MOV INSTRUCTIONS

It is often necessary to move data into one register from another. The instruction to do this has the form "MOV destination, source". Such an instruction exists for each possible pairing of registers. For instance:

BINARY CODE:	01001111
HEX CODE:	4F
MNEMONIC:	MOV C, A
MEANING:	Move into C the content of A

The data remain unchanged in the source register and are copied into the destination register, whose old content is lost. Note that in the mnemonic the destination is listed first, then the source register. Interchanging these is a common source of error, so be careful. Think of the instruction as "move into C from A" or "set C equal to A". The table below contains a summary of the MOV instructions. Note that the table is complete, including the useless MOV A,A; MOV B,B; etc. These are totally valueless to the user, but because of internal procedures in the microprocessor it would have added complexity to omit them or to use the wasted instruction codes for other purposes.

Inter-Register MOV Instructions:

	Source Register						
	A	B	C	D	E	H	L
MOV A,s	7F	78	79	7A	7B	7C	7D
MOV B,s	47	40	41	42	43	44	45
MOV C,s	4F	48	49	4A	4B	4C	4D
MOV D,s	57	50	51	52	53	54	55
MOV E,s	5F	58	59	5A	5B	5C	5D
MOV H,s	67	60	61	62	63	64	65
MOV L,s	6F	68	69	6A	6B	6C	6D

As an example we might need to copy data from some memory location into Register C:

```
3A        LDA 8300
00
83
4F        MOV C,A
```

The content of memory location 8300 is loaded into Register A and then copied into Register C. Both A and C now contain the same data as memory location 8300.

4.2 THE ADD INSTRUCTION

The program of Chapter 3 performed addition by counting. This is inefficient in terms of both program space and execution time. A single instruction will perform this function, now that we have a way to put one operand into another register:

BINARY CODE:	10000001
HEX CODE:	81
MNEMONIC:	ADD C
MEANING:	Add into A the content of C

Any register content may be added to A, with the result always being placed in A.

	HEX
ADD A	87
ADD B	80
ADD C	81
ADD D	82
ADD E	83
ADD H	84
ADD L	85

We can replace the repetitive loop in the program of Figures 3-2 and 3-3 with the ADD instruction.

```
8200    00    NOP
8201    00    NOP
8202    00    NOP
8203    E7    RST4        Enter Monitor
8204    32    STA 8300    Store Value Returned
8205    00
8206    83
8207    E7    RST4        Enter Monitor Again
8208    32    STA 8301    Store Value Returned
8209    01
820A    83
820B    3A    LDA 8300    Load First Value
820C    00
820D    83
820E    4F    MOV C,A     First Value to C
820F    3A    LDA 8301    Load Second Value
8210    01
8211    83
8212    81    ADD C       Add First Value
8213    C3    JMP 8204    Go Back to Store and Display Sum
8214    04
8215    82
```

This program is equivalent to the modified program of exercise 3.4. After finding a sum (by ADD C), we loop back to store the sum (STA 8300); enter the monitor to display the sum and accept a new number (RST4). After the first sum is displayed in this program, we only take one new number each time, and always add it to the old sum.

There is an important difference between this program and the "addition by counting" program, in its effect on the Carry flag.

4.3 THE CARRY AND ZERO FLAGS

In Chapter 3 we introduced the Zero flag and the conditional instruction Jump if Not Zero (JNZ). There are several other flags, and conditional instructions. Different instructions affect different flags, and some of the rules are fairly complicated. However, there are some simple general rules which may be defined before proceeding.

a. Data Transfer instructions never affect any flags. These include LDA, STA, MOV, and other similar instructions.

b. Counting (incrementing or decrementing) in any single register (A, B, C, D, E, H, L) sets the zero flag if the result of that count is zero. The condition of this flag at any given time does not necessarily mean that the register contains zero, however. Once the flag is set, a data transfer instruction may load the register without changing the flag.

c. Jump and conditional jump instructions never affect any flags.

4.3.1 Carry

If two numbers are added whose sum is greater than FF, there should be a Carry from the addition, e.g.:

```
    75   (HEX)
+   94   (HEX)
-------
=  109   (HEX)
```

This Carry is generated by the ADD instruction, among others, and sets a condition flag called Carry. Like the zero flag, Carry can be tested to cause a conditional jump to occur, but it can also be used in various arithmetic operations. Before discussing these, we will step through the program of Section 4.2 and observe Carry. It is indicated to the left of the numeric display by the top LED, labelled "CY". (In this description, keys to be pressed are shown at the left. The displays to be expected are shown at the right. (CY) and (Z) are shown where those flags are set. Until the first ADD, their states are unknown.)

RESET		8200	00
RUN	(until RST4)	8204	32
REG	A	8204	A-??
6	8 (enter a number)	8204	A-68
RUN	(until RST4)	8208	A-68
2	0 (another number)	8208	A-20
STEP		820B	A-20

The two values have been stored and we will now load the first value.

STEP		820E	A-68
STEP		820F	A-68
REG	C	820F	C-68

The first value has been copied to Register C and we will load the second value.

REG	A	820F	A-68
STEP		8212	A-20
STEP	(execute ADD C)	8213	A-88

We have added the two values. Note that both LED's left of the numeric display are off. The result of the addition was not zero, and did not generate a Carry.

STEP		8204	A-88
RUN	(until RST4 done)	8208	A-88

The old result has been stored at 8300, and the monitor is waiting for a new value, to be stored at 8301.

9	8	(enter a number)		8208	A-98
STEP		(store it)		820B	A-98
STEP		(load the old result)		820E	A-88
STEP		(move it to C)		820F	A-88
STEP		(load the new number)		8212	A-98

Now the content of 8300 has been copied to register C and the content of 8301 has been loaded into A. The next step will add these values. The hexadecimal result should be:

```
   88
+  98
------
= 120
```

The sum is greater than FF, so a Carry will result and will be shown in the upper LED to the left of the display.

STEP		(CY)	8213	A-20
RUN	(until RST4)	(CY)	8208	A-20

Note that the jump and store instructions have not affected the Carry flag. The value 20 (HEX) has been stored at 8300.

6	0	(enter new number)	(CY)	8208	A-60
RUN				8208	A-80

We have added 20 + 60. The Carry flag is cleared, because the result was not greater than FF.

Now we shall allow 80 (HEX) to be used for both values.

```
RUN                                        (CY)(Z)  8208    A-00
```

A Carry was generated by adding 80 + 80, and the numeric result is zero, so both Carry and Zero are set.

Use this program to add the column of numbers below. Write in the result of each addition and note if the Carry is set.

		Carry	Sum
First Number	04		
Second Number	44	____	____
	60	____	____
	95	____	____
	32	____	____
	A1	____	____
	F0	____	____
	C2	____	____
	C2	____	____
	80	____	____
	44	____	____
	60	____	____
	FF	____	____
	FF	____	____
	0A	____	____
	60	____	____

We have seen how the Carry flag is set or reset by the addition. Note that with the ADD instruction any previous Carry was lost and did not affect a further result. In the next section we shall see how the Carry flag can be used in addition.

4.3.2 Multiple Precision - The ADC Instruction

A single byte of data in memory or in a register can represent an integer value from 00 to FF (255 decimal). Obviously many computer programs need to represent numbers much larger than this, so more than one byte is used to represent such numbers. This is just like the use of multiple digits to represent numbers greater than 9 in decimal arithmetic.

Definitions:

MULTIPLE PRECISION:	The use of two or more bytes to represent an integer greater than FF (255 decimal).
DOUBLE PRECISION:	The use of exactly two bytes to represent an integer value from 0000 to FFFF (65535 decimal).

These definitions apply only to computers whose word size is 8 bits, and only in the context of unsigned integer values. The phrases convey similar ideas but with more complicated definitions in other contexts.

When we perform multi-digit addition the low order digits are added without regard to Carry, but for all higher digits a Carry must be considered.

```
Carry          1 0 1 X
                7 6 3 9  (decimal)
             + 1 5 4 3  (decimal)
             = 9 1 8 2  (decimal)
```

Similarly the computer can add low order bytes without regard to Carry, and then include the Carry for higher bytes using an ADC (add with Carry) instruction.

Example: ADC B

BINARY CODE:	10001000
HEX CODE:	88
MNEMONIC:	ADC B
MEANING:	Add the content of B to the content of A. If Carry was set before the addition, increase the result by 1. Place the result into Register A. If the addition generates Carry, set the Carry flag; otherwise reset it. If the result of the addition is zero, set the Zero flag; otherwise reset it.

Note that ADD and ADC both set or reset Carry and Zero in exactly the same way. The difference lies in the inclusion of Carry in the addition. A full set of ADC instructions exists.

	HEX CODE
ADC A	8F
ADC B	88
ADC C	89
ADC D	8A
ADC E	8B
ADC H	8C
ADC L	8D

Example: Add the content of Registers B and C to the content of Registers D and E. Here we consider C and E to contain the low order bytes to be added; B and D the high order bytes. The result is to be placed in D and E. Load this program.

```
8200    7B    MOV  A,E
8201    81    ADD  C
8202    5F    MOV  E,A
8203    7A    MOV  A,D
8204    88    ADC  B
8205    57    MOV  D,A
8206    E7    RST4
8207    C3    JMP  8200
8208    00
8209    82
```

Before stepping through the program place a two byte number (four HEX digits) into Registers B and C, and another number into Registers D and E.

REG	B	4	5		8200	B-45
REG	C	8	5		8200	C-85
REG	D	5	2		8200	D-52
REG	E	A	7		8200	E-A7

The numbers to be added are:

B, C	4585
D, E	52A7

The sum should be: 982C

Now step through the program.

ADDR					8200	7B
REG	A				8200	A-??
STEP					8201	A-A7
STEP				(CY)	8202	A-2C

The low bytes (A7 and 85) have been added, resulting in the low byte of the sum in Register A. Carry is set.

STEP				(CY)	8203	A-2C
STEP				(CY)	8204	A-52

We are about to add the high bytes (52 and 45) with Carry, which is set.

STEP					8205	A-98

The sum of 52 and 45 has been augmented by the Carry. No Carry resulted from this addition, so the Carry flag is clear.

```
STEP                    8206    A-98
STEP                    0020    A-98
STEP                    8207    A-98
```

We reentered the monitor at 0020 and are now at 8207 where we will jump back to the beginning. Examine the registers.

```
REG     B               8207    B-45
NEXT                    8207    C-85
NEXT                    8207    D-98
NEXT                    8207    E-2C
```

The content of Registers B and C has not changed. Registers D and E contain the sum, 982C.

We can again add the content of B and C to this sum merely by pressing RUN.

```
RUN                     8207    E-B1
REG     D               8207    D-DD
```

The new sum is DDB1.

Before doing this again, predict the next sum and carry.

B, C = 4 5 8 5

D, E = D D B 1

Carry _____ Sum _____

```
RUN                                    8207    D-??
NEXT                                   8207    E-??
```

Does the result agree with your prediction? It should be Carry, 2336.

4.3.3 Exercise

Rewrite the program we have just used to add the content of Registers B and C to the content of Registers H and L, placing the result in Registers H and L.

The solution is given in Figure 4-1.

DOUBLE PRECISION ADDITION

CODING SHEET

ADDR		CODE												
8	20	0	7	D		MOV		A,	L					
		1	8	1		ADD		C						
		2	6	F		MOV		L,	A					
		3	7	C		MOV		A,	H					
		4	8	8		ADC		B						
		5	6	7		MOV		H,	A					
		6	E	7		RST	4							
		7	C	3		JMP			8200					
		8	0	0										
		9	8	2										
		A												
		B												
		C				ADDS		THE		C	ONTENT OF			
		D				REGISTERS				B AND C				
		E				INTO		THE		C	ONTENT OF			
		F				REGISTERS				H AND L				
8		0												
		1												
		2												
		3												
		4												
		5												
		6												
		7												
		8												
		9												
		A												
		B												
		C												
		D												
		E												
		F												
3		0												
		1												
		2												
		3												
		4												
		5												
		6												
		7												
		8												

Figure 4-1

4.3.4 Subtraction - SUB and SBB

Subtraction is defined as the inverse of addition:

If A = B + C

Then C = A - B

We can show that this rule applies in the computer as well as in elementary school arithmetic. The 8080 has a set of subtract instructions; for example:

BINARY CODE:	10010000
HEX CODE:	90
MNEMONIC:	SUB B
MEANING:	Subtract the content of Register B from the content of Register A. Place the result in Register A. If the result is zero, set the Zero flag; otherwise reset the Zero flag. If the content of Register B was greater than the original content of Register A, set the Carry flag; otherwise reset the Carry flag.

To test the definition enter this program:

```
8200    78    MOV A,B
8201    81    ADD C
8202    90    SUB B
8203    C3    JMP 8200
8204    00
8205    82
```

Now enter data into B and C, and step through the program observing A.

```
REG     B       8       6                       8200    B-86
REG     C       1       2                       8200    C-12
REG     A                                       8200    A-??
STEP                                            8201    A-86
STEP                                            8202    A-98
STEP                                            8203    A-12
```

Adding 86 plus 12 gave 98; subtracting 86 gave 12. The rule still holds even if the sum is greater than FF.

```
STEP                                            8200    A-12
REG     C       9       0                       8200    C-90
REG     A                                       8200    A-12
STEP    (move into A from B)                    8201    A-86
STEP    (add C, 86 + 90)                (CY)    8202    A-16
STEP    (subtract B, 16 - 86)           (CY)    8203    A-90
```

Although the Carry flag was set when a sum greater than FF was generated, this Carry was ignored by the SUB instruction. It was set again by SUB when we subtracted 86 from 16.

As in addition, the Carry flag is used for multiple precision arithmetic. The SBB (subtract with borrow) instructions are used for this purpose. Note that although this name speaks of a "borrow" rather than a "carry" it is represented by the same flag in the 8080 microprocessor. The 8080 does not distinguish whether it resulted from an ADD or SUB instruction.

BINARY CODE:	1001 1000
HEX CODE:	98
MNEMONIC:	SBB B
MEANING:	If the Carry flag is set, reduce the value in Register A by 1. Subtract the content of Register B from the content of Register A. Place the result in Register A. If the result is zero, set the Zero flag; otherwise reset Zero. If the content of Register B was greater than the content of Register A minus CY, set Carry; otherwise reset Carry.

SUB and SBB exist for all registers:

97	SUB	A	9F	SBB	A
90	SUB	B	98	SBB	B
91	SUB	C	99	SBB	C
92	SUB	D	9A	SBB	D
93	SUB	E	9B	SBB	E
94	SUB	H	9C	SBB	H
95	SUB	L	9D	SBB	L

The double precision addition we programmed in Section 4.3.2 can readily be converted to a double precision subtraction, using SUB and SBB in place of ADD and ADC. Refer to Section 4.3.2 and write a program to subtract the content of Registers B and C from the content of Registers D and E. A solution is given in Figure 4-2.

From this point on we shall omit the binary codes when new instructions are defined, showing only the hex codes. Binary codes have been shown to stress that the computer recognizes binary patterns, not hex characters. If you translate into binary the hex codes above, and those for the MOV, ADD and ADC instructions given previously, you can see the patterns recognized by the computer. These are discussed in Chapter 11.

DOUBLE PRECISION SUBTRACTION

CODING SHEET

ADDR		CODE													
8 20	0	7	B		M	O	V		A	,	E				Subtract
	1	9	1		S	U	B		C						E - C
	2	5	F		M	O	V		E	,	A				Result to F
	3	7	A		M	O	V		A	,	D				Subtract
	4	9	8		S	B	B		B						D - B - CY
	5	5	7		M	O	V		D	,	A				Result to D
	6	E	7		R	S	T	4							
	7	C	3		J	M	P			8	2	0	0		
	8	0	0												
	9	8	2												
	A														
	B														
	C														
	D														
	E														
	F														
8	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														
	9														
	A														
	B														
	C														
	D														
	E														
	F														
3	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure 4-2

4.3.5 Review and Self Test

In Sections 4.1, 4.2 and 4.3 we have introduced a number of instructions that involve using registers to store data, provide operands, and count. Test your knowledge by answering the questions below. Each question refers to the section in which it is answered. The correct answers are given on the reverse side of this page.

1) What is the other name for Register A? (Section 4.0)_____

2) Name the other general purpose registers. (Section 4.0)_____

3) Which register receives results from arithmetic operations? (Section 4.0) _____

4) Which register has its content changed by the instruction MOV E,C? (Section 4.1)_____

5) Which register has its content changed by the instruction ADD B? (Section 4.2) _____

6) Which of the flags are affected by each of the following instructions? (Section 4.3)

		ZERO	CARRY
MOV	E,C	_____	_____
ADD	B	_____	_____
LDA	8300	_____	_____
INR	A	_____	_____
DCR	C	_____	_____
SBB	D	_____	_____

Answers to Self-Test, Section 4.3.4

1) Register A is also called the Accumulator.

2) The other registers are B, C, D, E, H, L.

3) Register A receives the results of arithmetic and logic operations.

4) MOV E, Cmoves into E the content of C. Register E is affected; Register C is unchanged.

5) ADD B adds the content of B to the content of A and places the result in A. Register B is unchanged.

6) MOV E, Caffects no flags.

   ADD B affects all flags.

   LDA affects no flags.

   INR A affects Zero.
   does not affect Carry.

   DCR C affects Zero.
   does not affect Carry.

   SBB D affects all flags.

4.4 IMMEDIATE INSTRUCTIONS

Although we have distinguished program memory from data memory, it is common to include some data in the program memory. Tables of fixed values such as values of functions (e.g. trigonometric) or calibration data are often stored at the end of a program. Some instructions include data in the second, or second and third bytes of the instruction. These are known as "immediate data" and the instructions are called "immediate instructions". Such an instruction (ADI) was presented in the second chapter.

A very common requirement is to load a register with some fixed value.

4.4.1 Move Immediate Instructions (MVI r)

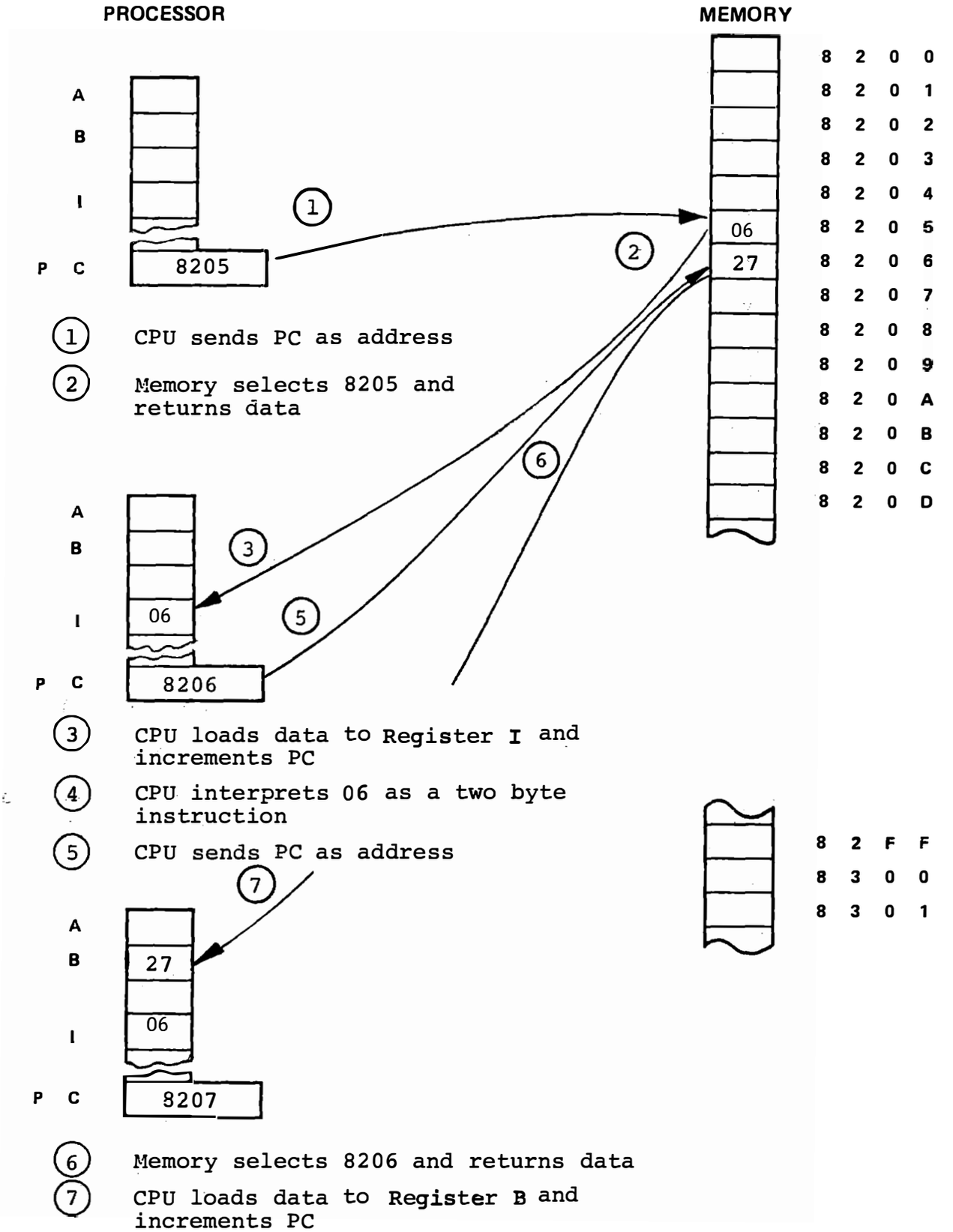
The MOV instruction has a complete set of MVI counterparts. The general MVI instruction looks like this:

MNEMONIC:	MVI r
SECOND BYTE:	Data
MEANING:	Move the value contained in the immediately following byte into Register r.

Following is the complete set of MVI instructions:

MNEMONIC	HEX CODE
MVI A	3E
MVI B	06
MVI C	0E
MVI D	16
MVI E	1E
MVI H	26
MVI L	2E

The MVI instruction is often used to initialize a counter. For example, in serial data communications it is necessary to transmit the eight bits of one byte sequentially. The counter is initialized at 8 and successively decremented (using DCR) to detect completion of the transmission. Then a JNZ instruction at the end of the loop causes repetition until the counter reaches zero. The instruction cycle for the MVI is shown in Figure 4-3.



MVI Instruction Cycle

Figure 4-3

4.4.2 Immediate Arithmetic Instructions

It is sometimes necessary to add a fixed value to a number - for instance one might want to count by threes. Although this could be done by placing the desired value in a register and adding the register content to Register A, the 8080 provides two instructions to perform the function directly: ADI data (add immediate) and ACI data (add with Carry immediate). We met the ADI instruction in Chapter 1; ACI is defined here.

HEX CODE:	CE
SECOND BYTE:	Data
MNEMONIC:	Add the value contained in the immediately following byte to the content of A. If Carry was set before the addition, increase the result by 1. Place the result in Register A. Set or reset the Carry and Zero flags according to the result.

Similarly there exist immediate counterparts for SUB and SBB. Thus we have:

```
C6 data          ADI     data

CE data          ACI     data

D6 data          SUI     data

DE data          SBI     data
```

Probably the most common use of the ACI instruction occurs when an arithmetic operation is required to generate a result with more bytes than the numbers being added. In the example of Section 4.3.2 we repeatedly added the content of B and C to a value in Registers D and E. When the sum exceeded FFFF a Carry occurred from the multi-byte addition, but was lost when we repeated the addition again. If we had provided for an additional byte in the result (say in Register L) the Carries could have been added into that byte by:

```
MOV     A,L
ACI     OO
MOV     L,A
```

This technique is used in multiplication or when a column of numbers is to be added. The next exercise demonstrates this.

4.4.3 Multiplication by Repetitive Addition

The process of multiplication that we use in decimal arithmetic is exactly equivalent to repetitive addition.

$$3 \times 8 = 8 + 8 + 8 = 24 \quad \text{(decimal)}$$

The same is true in binary (or hexadecimal) arithmetic in a computer. One way of performing multiplication is to add the multiplicand (8 in the above example) into the product (initially set to zero) repeatedly, multiplier times.

Definition:

MULTIPLICAND:	A number which is to be multiplied by another number, called a MULTIPLIER to generate a PRODUCT.

Although the multiplier and multiplicand should be interchangeable without affecting the result, the distinction is useful in describing the process. Load and test this:

```
8200    06      MVI   B,08    Place in Register B
8201    08                    The multiplicand
8202    0E      MVI   C,03    Place in Register C
8203    03                    The multiplier
8204    1E      MVI   E,00    Clear the product
8205    00                    to zero
8206    7B      MOV   A,E     Add to product
8207    80      ADD   B       The multiplicand
8208    5F      MOV   E,A     Save partial product
8209    0D      DCR   C       Count multiplier
820A    C2      JNZ   8206    down to zero
820B    06
820C    82
820D    E7      RST   4       Re-enter monitor
820E    C3      JMP   8200    Repeat
820F    00
8210    82
```

The result (in Register E) is 18 HEX (= 24 decimal). The program works since the product does not exceed FF, and so can be stored in a single byte. What happens for larger values of multiplicand or multiplier? If the immediate value for the multiplicand (at address 8201) is set to 70, then the final addition results in a Carry.

```
Initial Product          = 00
Add Multiplicand         + 70
Partial Product          = 70     No Carry
Add Multiplicand         + 70
Partial Product          = E0     No Carry
Add Multiplicand         + 70
Product                  = 50     Carry Set
```

Since the Carry is preserved, indicating a product of 150 (HEX) this might be acceptable. If the multiplicand were 90, this process would occur:

Initial Product	= 00	
Add Multiplicand	+ 90	
Partial Product	= 90	No Carry
Add Multiplicand	+ 90	
Partial Product	= 20	Carry
Add Multiplicand	+ 90	
Product	= B0	No Carry

The intermediate carry is lost. The result should have been 1B0, not B0. If the multiplicand and multiplier were each set to FF, the product would be FE01, a two byte number.

We can fix the program above by using two bytes for the product (say D and E). Both must be cleared initially. Then the multiplicand is added to the low byte of the product. If a Carry results it must be added into the high byte of the product. This is done with the ACI 00 instruction as shown below:

Program For Multiplication by Repetitive Addition

```
8200    06    MVI  B,FF  Place in Register B
8201    FF               the multiplicand
8202    0E    MVI  C,FF  Place in Register C
8203    FF               the multiplier
8204    1E    MVI  E,00  Clear product
8205    00               low byte
8206    16    MVI  D,00  high byte
8207    00
8208    7B    MOV  A,E   Product low byte
8209    80    ADD  B     Add multiplicand
820A    5F    MOV  E,A
820B    7A    MOV  A,D   Product high byte
820C    CE    ACI  00    Add Carry
820D    00
820E    57    MOV  D,A
820F    0D    DCR  C     Count multiplier
8210    C2    JNZ  8208  down to zero
8211    08
8212    82
8213    E7    RST  4     Enter monitor
8214    C3    JMP  8200  Repeat
8215    00
8216    82
```

Step through this program for a few loops, observing Register A and Carry. Then run it and look at the result in Registers D and E. Is Carry set or cleared at the end?

4.4.4 Multiplication - Exercise

When we perform multiplication with pencil and paper, the number of digits in the product depends on the sizes of the two numbers:

```
   22          99
 X 14        X 99
-----       -----
  308        9801
```

We express the answers this way because we always discard leading zeros, and assume that any higher order digits not shown must be zero. In the computer, however, storage must be provided for as many bytes as might be generated with the maximum values of multiplier and multiplicand that are permitted by the program.

The product of two numbers may occupy as many bytes as the sum of the number of bytes being multiplied. For example, a two byte number multiplied by a one byte number generates a three byte result.

FFFF X FF = FEFF01

(in decimal, 65535 X 255 = 16711425)

Write a program to multiply a two byte multiplicand by a one byte multiplier. Take the multiplier from memory location 8300. Take the low byte of the multiplicand from memory location 8301 and the high byte from memory location 8302. Store the three byte result in memory locations 8303 (low byte) to 8305 (high byte).

Try to write this program by yourself, using the instructions listed in Section 4.4.5. Remember to clear the product before starting the repetitive additions. Hint: It is usually more efficient to use registers for data than to load and store numbers in memory repetitively. Make a table of memory and register assignments.

Meaning of Data	Memory Location	Register
Multiplier	8300	_____
Multiplicand (low byte)	8301	_____
Multiplicand (high byte)	8302	_____
Product (low byte)	8303	_____
Product (mid byte)	8304	_____
Product (high byte)	8305	_____

A solution is given in Figure 4-4, following the list of instructions in Section 4.4.5.

4.4.5 Table of Instructions

Re-enter Monitor

E7	RST 4	(applies to ICS Microcomputer Training System only)

Jump and Conditional Jump Instructions

C3	JMP Address	C2	JNZ Address
XX	(low address)	XX	(low address)
XX	(high address)	XX	(high address)

Data Transfer Instructions

3A	LDA Address	32	STA address
XX	(low address)	XX	(low address)
XX	(high address)	XX	(high address)
78	MOV A,B	47	MOV B,A
79	MOV A,C	4F	MOV C,A
7A	MOV A,D	57	MOV D,A
7B	MOV A,E	5F	MOV E,A
7C	MOV A,H	67	MOV H,A
7D	MOV A,L	6F	MOV L,A

(Other register-to-register MOV instructions are tabulated on Page 4-3.)

Immediate Data Transfer Instructions

3E	MVI	A,	data	
data				
06	MVI	B,	data	
data				
OE	MVI	C,	data	
data				
16	MVI	D,	data	
data				
1E	MVI	E,	data	
data				
26	MVI	H,	data	
data				
2E	MVI	L,	data	
data				

None of the above instructions affect any flags.

Counting Instructions

These counting instructions set or reset Zero. The Carry Flag is not affected.

3C	INR	A	3D	DCR	A
04	INR	B	05	DCR	B
0C	INR	C	0D	DCR	C
14	INR	D	15	DCR	D
1C	INR	E	1D	DCR	E
24	INR	H	25	DCR	H
2C	INR	L	2D	DCR	L

Arithmetic Instructions

Zero and Carry are set or reset by these instructions.

87	ADD	A	8F	ADC	A
80	ADD	B	88	ADC	B
81	ADD	C	89	ADC	C
82	ADD	D	8A	ADC	D
83	ADD	E	8B	ADC	E
84	ADD	H	8C	ADC	H
85	ADD	L	8D	ADC	L
C6 data	ADI	data	CE data	ACI	data
97	SUB	A	9F	SBB	A
90	SUB	B	98	SBB	B
91	SUB	C	99	SBB	C
92	SUB	D	9A	SBB	D
93	SUB	E	9B	SBB	E
94	SUB	H	9C	SBB	H
95	SUB	L	9D	SBB	L
D6 data	SUI	data	DE data	SBI	data

MULTIPLICATION BY REPETIVE ADDITION

CODING SHEET

ADDR		CODE												
8 20	0	1	6		M	V	I		D,	0	0			Use D H, L for three byte product
	1	0	0											
	2	2	6		M	V	I		H,	0	0			
	3	0	0											
	4	2	E		M	V	I		L,	0	0			
	5	0	0											
	6	3	A		L	D	A			8	3	0	0	Load Multiplier
	7	0	0											
	8	8	3											
	9	5	F		M	O	V		E,	A				to Register E
	A	3	A		L	D	A			8	3	0	1	
	B	0	1											
	C	8	3											
	D	4	F		M	O	V		C,	A				Copy Multiplicand to Registers B and C
	E	3	A		L	D	A			8	3	0	2	
	F	0	2											
8 21	0	8	3											
	1	4	7		M	O	V		B,	A				
	2	7	D		M	O	V		A,	L				Low byte of product
	3	8	1		A	D	D		C					Add multiplicand
	4	6	F		M	O	V		L,	A				
	5	7	C		M	O	V		A,	H				Mid byte of product
	6	8	8		A	D	C		B					Add multiplicand
	7	6	7		M	O	V		H,	A				
	8	7	A		M	O	V		A,	D				High byte of product
	9	C	E		A	C	I		0	0				Add Carry
	A	0	0											
	B	5	7		M	O	V		D,	A				
	C	1	D		D	C	R		E					Count multiplier down to zero
	D	C	2		J	N	Z			8	2	1	2	
	E	1	2											
	F	8	2											
3	0													
	1				(	C	O	N	T	I	N	U	E	D NEXT PAGE)
	2													
	3													
	4													
	5													
	6													
	7													
	8													

Figure 4-4a

ADDR		CODE													
8	22	0	7D		MOV			A,	L						Copy Product
		1	32		STA				8303						into memory
		2	03												
		3	83												
		4	7C		MOV			A,	H						
		5	32		STA				8304						
		6	04												
		7	83												
		8	7A		MOV			A,	D						
		9	32		STA				8305						
		A	05												
		B	83												
		C	E7		RST	4									Enter Monitor
		D	C3		JMP				8200						Back to Start
		E	00												
		F	82												
8		0													
		1													
		2													
		3													
		4													
		5													
		6													
		7													
		8													
		9													
		A													
		B													
		C													
		D													
		E													
		F													
3		0													
		1													
		2													
		3													
		4													
		5													
		6													
		7													
		8													

Figure 4-4b

4.5 CONDITIONAL JUMPS

In Sections 4.3 and 4.4 we used the Carry flag in addition (with ADC or ACI) and in subtraction (SBB or SBI). This flag can also be controlled in several ways other than by addition and subtraction. Moreover, the Carry flag can be used to control execution of a conditional jump just as the Zero flag has done in our programs thus far.

Before proceding with this subject, let us review that single register counting instructions (INR and DCR) affect the Zero flag, but not the Carry flag. If the result of the count is zero, the Zero flag is set; otherwise it is cleared.

Arithmetic and logical instructions, on the other hand, affect both Zero and Carry. If the result of the operation is a zero in the accumulator, the Zero flag is set; otherwise it is cleared. If the operation generates a carry out of the highest bit the Carry flag is set, otherwise it is cleared. Conditional jumps can be made with tests for the set or clear state of each flag:

HEX CODE	MNEMONIC	MEANING
C2 xxxx	JNZ address	Jump if not Zero
CA xxxx	JZ address	Jump if Zero
D2 xxxx	JNC address	Jump if not Carry
DA xxxx	JC address	Jump if Carry

All of these are three byte instructions. For instance:

```
8218    D2      JNC     821C
8219    1C
821A    82
821B    14      INR     D
```

If Carry is not set when the JNC instruction is executed, the jump to 821C is made. If Carry is set, the program continues at 821B. The instruction cycle is similar to that for JMP. The entire instruction is read, with the address being copied into temporary Registers W and Z; the flag determines whether that address is copied into the program counter.

The procedure shown above is another means of adding the Carry into the high byte of a sum or product, which can be used instead of the ACI 00 instruction. If the two instructions above are substituted for

```
MOV  A,D
ACI  00
MOV  D,A
```

in the given solution for exercise 4.4.4 (Figure 4-4) the same product will be found. When no Carry is generated by the mid-byte addition, the JNC instruction passes over the INR D. When a Carry is generated, the JNC is not executed, so Register D is incremented, just as though the Carry had been added by ACI 00.

There is one effect of the revised program different from the original version. Since the multiplication of one byte times two bytes cannot exceed the capacity of the three byte product, ACI 00, in the program of Figure 4-4, never generates a Carry. Therefore, the original version of this program always finishes with no Carry. In the version using JNC, if the mid byte addition ADC B generates a Carry on the final loop, that Carry remains at the end because following JNC, INR D, DCR E instructions in that program do not affect the Carry flag. With arbitrary multiplicand and multiplier we cannot predict the state of the Carry at the end, and it conveys no useful information. Therefore, the ACI 00 technique is generally preferred in arithmetic programs, unless its very slightly slower execution is important. JNC and JC were introduced here because typical programs much more often use the Carry flag for decision making than for arithmetic.

4.6 TRANSFER NOTATION

A number of new instructions have been introduced. Most of these are members of sets that perform similar functions using different registers as a source and destination for data.

For convenience in describing instructions, we shall now introduce "transfer notation". A capital letter designates a specfic register or a flag; a lower case letter refers to a register which will be identified in the instruction. Parentheses imply "the content of". Thus:

```
ADD   r        (A) <- (A) + (r).
```

states that the content of Register r is added to the content of Register A and the result is placed in Register A.

4.6.1 Instruction Definitions

The instructions used so far in the course are described below using transfer notation. Their effects on the Carry and Zero flags are also indicated. (The other three flags of the 8080 are treated in Chapters 10 and 11.) Review all of the instructions shown here to be sure that you understand them.

LDA address	(A) <- (address) Regiser A is loaded with the content of the memory location whose address is given in bytes 2 and 3 of the instruction. No flags are affected.
STA address	(address) <- (A) The content of Register A is stored at the memory location whose address is given in bytes 2 and 3 of the instruction. No flags are affected.
JMP address	(PC) <- address The address in bytes 2 and 3 of the instruction is loaded into the program counter. Program execution continues from that address. No flags are affected.

JNZ address	If Zero flag is clear (PC) <- address Otherwise program execution continues at the next sequential instruction. No flags are affected.
JZ address	If Zero flag is set (PC) <- address Otherwise program execution continues at the next sequential instruction. No flags are affected.
JNC address	If Carry flag is clear (PC) <- address Otherwise program execution continues at the next sequential instruction. No flags are affected.
JC address	If Carry flag is set (PC) <- address Otherwise program execution continues at the next sequential instruction. No flags are affected.
MOV d,s	(d) <- (s) The content of source Register s is copied into destination Register d. No flags are affected.

MVI r, data — (r) <- data

Register r is loaded with the data contained in byte 2 of the instruction. No flags are affected.

INR r — (r) <- (r) + 1

Register r is incremented. Zero is set or reset. Carry is not affected.

DCR r — (r) <- (r) - 1

Register r is decremented. Zero is set or reset. Carry is not affected.

ADD r — (A) <- (A) + (r)

Zero is set or reset. Carry is set or reset.

ADC r — (A) <- (A) + (r) + (CY)

Zero is set or reset. Carry is set or reset.

ADI data — (A) <- (A) + data

Zero is set or reset. Carry is set or reset.

ACI data — (A) <- (A) + data + (CY)

Zero is set or reset. Carry is set or reset.

SUB r	(A) <- (A) - (r) Zero is set or reset. Carry is set or reset.
SBB r	(A) <- (A) - (r) - (CY) Zero is set or reset. Carry is set or reset.
SUI data	(A) <- (A) - data Zero is set or reset. Carry is set or reset.
SBI data	(A) <- (A) - data - (CY) Zero is set or reset. Carry is set or reset.
XRA A	(A) <- 00 Zero is set. Carry is reset. (Note: XRA A is a member of a set of logic instructions which will be introduced later. The above definition applies to XRA A only).
RST 4	Enter monitor This applies to the ICS Microcomputer Training System only.

4.6.2 Review and Self Test

In the preceding sections we have used data transfer instructions, arithmetic and counting instructions, and immediate instructions. Test your knowledge by answering the questions below. Correct answers are on Page 4-51.

1) Use transfer notation to describe these instructions: (Section 4.5)

MOV C,E	_____
SUB r	_____
MVI D,13	_____
ADC E	_____
ACI 00	_____

2) What instruction is described by each of the following statements in transfer notation? (Section 4.5)

(8300) <- (A)	_____
(PC) <- address	_____
(r) <- (r) - 1	_____
(A) <- (A) + data + (CY)	_____

3) What instruction usually appears at the end of a repetitive loop controlled by counting? (Section 4.4.1)

_____

4) Identify the register and flags affected by each of these instructions. (Section 4.5)

	Register	Zero	Carry
INR D	_____	____	____
MOV B,A	_____	____	____
STA 8300	_____	____	____
ADC E	_____	____	____

This page intentionally left blank.

Answers to Self Test, Section 4.6.2

1)	MOV	C,E	(C) <- (E)
	SUB	r	(A) <- (A) - (r)
	MVI	D, 13	(D) <- 13
	ADC	E	(A) <- (A) + (E) + (CY)
	ACI	00	(A) <- (A) + (CY)
2)	STA	8300	(8300) <- (A)
	JMP	address	(PC) <-address
	DCR	r	(r) <- (r) - 1
	ACI	data	(A) <- (A) + data + (CY)

3) A repetitive loop controlled by counting usually ends with JNZ

4)

		Register	Zero	Carry
INR	D	D	X	
MOV	B,A	B		
STA	8300	None		
ADC	E	A	X	X

Bit Pattern	Display
0000 0000	Off
0000 0001	(Top Horizontal)
0000 0010	(Upper Right)
0000 0100	(Lower Right)
0000 1000	(Bottom Horizontal)
0001 0000	(Lower Left)
0010 0000	(Upper Left)
0100 0000	(Middle Horizontal)
1000 0000	(Decimal Point)
1111 0111	(All Except Bottom Horizontal)

Bit Patterns for MTS Display

Figure 4-5

4.7 THE MTS DISPLAY

Until this point the only means we have used for input of data and output of results has been to enter the monitor and look at registers and memory locations. Now we will output directly to the display. The hardware used in this process is described in Chapter 5; for the moment simply accept the following functional description. Later we will explain the external process.

4.7.1 Displaying a Bit Pattern

If you store a pattern of bits in a cetain memory location, that pattern will be reproduced in one of the display digits. Note that the bit pattern is not interpreted as a number, but reproduced as a pattern. Figure 4-5 shows the segments illuminated by each bit. If only one bit in the pattern is a 1 and all others are 0, then exactly one segment will be illuminated. If two bits are 1's, then two segments will be illuminated. The last pattern in Figure 4-5 shows seven bits set to 1; only the bottom horizontal is left off. Try this with the following program.

```
8200    32     STA 83F8
8201    F8
8202    83
8203    C3     JMP 8203
8204    03
8205    82
```

Before running this program, enter a value into register A.

```
REG    A      4      0                    8200    A-40
RUN                                       -
RESET                                     8200     32
REG    A      F      7                    8200    A-F7
RUN                                       A.
```

The bit pattern you enter into Register A is reproduced in the left hand digit. The monitor destroys the pattern you have displayed, so here we cannot reenter the monitor automatically, nor step through the program. Instead the program ends with an instruction that jumps to itself. The program waits here indefinitely, simply repeating that jump, until you press RESET. Therefore, we can now write programs that have output functions but no input. Until we learn of other means of input (in Chapter 6) we are limited to generating displays that change only according to values built into the program, or values entered before running the program. The following exercise uses such a procedure.

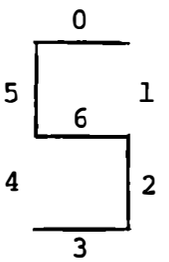
4.7.2 Display Digit Addresses

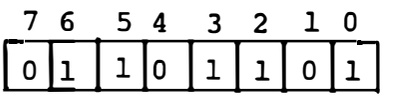
You saw above that a pattern stored at memory location 83F8 appears in the left digit. The next digit is controlled by 83F9, the third by 83FA, etc. The right hand digit is controlled by the bit pattern stored at 83FF.

We can load the display with a fixed pattern by a series of instructions like:

```
MVI   A, xx
STA   xxxx
```

To create the bit pattern for a desired display, draw the pattern in seven segment format, and mark the bit numbers. For example:





If the segment is to be illuminated, enter a 1 for that bit position into the pattern; otherwise enter a 0. Translate the bit pattern into hexadecimal and use that value in a MVI A, data instuction. The above example gives a HEX value of 6D, so the instruction is

```
3E    MVI   A, 6D
6D
```

For example, the series below will display ICS.

```
8200    3E    MVI A,'I'
8201    06
8202    32    STA 83F8
8203    F8
8204    83
8205    3E    MVI A,'C'
8206    39
8207    32    STA 83F9
8208    F9
8209    83
820A    3E    MVI A,'S'
820B    6D
820C    32    STA 83FA
820D    FA
820E    83
820F    C3    JMP 820F
8210    0F
8211    82
```

Exercise: Convert your own initials or name into characters, using the patterns from Figure 4-5, and make a display that pleases you.

4.8 REGISTER PAIRS AND MEMORY ADDRESSING

In the examples and exercises of Sections 4.3 and 4.4 we often used two registers to store a 16 bit number (and once, three registers for a 24 bit number). The general purpose registers (B, C, D, E, H, L) are equivalent to each other for the instructions used so far. They store data, provide operands for arithmetic and logical instructions, and count either up or down. When we stored a multiplicand in Registers B and C we could equally well have chosen any other two registers, or we could have reversed the order, using B for the low byte and C for the high byte.

Many instructions of the 8080 treat the general purpose registers as pairs, to hold sixteen bit numbers, in much the way we have been using them:

Register Pair B	B contains high byte
	C contains low byte
Register Pair D	D contains high byte
	E contains low byte
Register Pair H	H contains high byte
	L contains low byte

Their arrangement is like that of Registers W and Z, and for the same reason: a pair of eight bit registers is able to store a 16-bit memory address.

A number of instructions use register pairs for addressing the data memory. There are several reasons for addressing the memory this way. The least important (but not trivial) reason is efficiency. If the same address is to be accessed repeatedly, it takes less program space and running time to load the address into a register pair than

to repeatedly load the memory address from the program memory into W,Z. More importantly, if the same operation is to be performed on data in a series of adjacent memory locations, that operation can be performed in a repetitive loop, with the address being modified by incrementing (or decrementing) the register pair.

In many applications a memory address is calculated from variable data, or loaded from another memory location.

4.8.1 The LDAX and STAX Instructions

Register pairs B,C and D,E are used for addressing by the LDAX and STAX instructions. These correspond to the LDA and STA instructions, differing only in the source of address information. As is the case in all instructions using register pairs, the name of the first register is used to identify the pair, as in LDAX B:

HEX CODE:	OA
MNEMONIC:	LDAX B
MEANING:	Load Register A with the content of the memory location whose address is contained in register pair B,C. No flags are affected.

This is called an indirect instruction, and is expressed as: " Load A indirect from B". The term "indirect" means simply that the content of the designated register is not to be loaded; rather, its content is the address of a location to be loaded. The address is obtained indirectly, rather than by directly specifying it as the LDA instruction would have done.

The other instructions in this set are:

1A	LDAX	D	Load A indirect from D (A) <- ((DE))

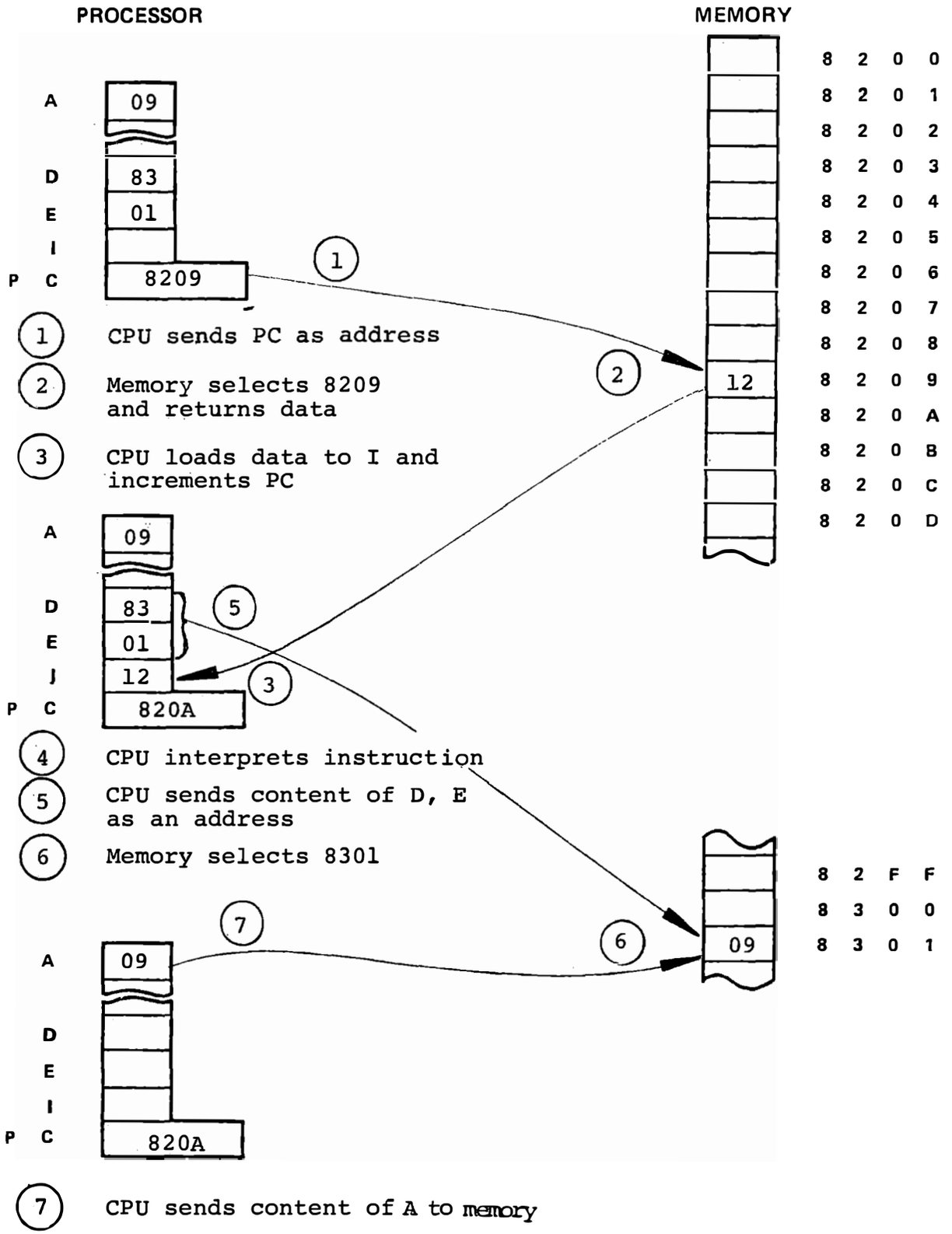
The STAX instructions similarly provide for storing data:

02	STAX	B	Store A indirect at B ((BC)) <- (A)
12	STAX	D	Store A indirect at D ((DE)) <- (A)

The content of A is stored in the memory location whose address is contained in the named register pair. Note that double parentheses such as ((BC)) imply the content of the memory location whose address is contained in register pair B,C.

Figure 4-6 illustrates the instruction cycle for STAX D, which typifies this usage of register pairs.

Note the absence of LDAX H and STAX H. The register pair HL is used to address memory in an even more powerful way, which will be introduced in Section 4.9.



Instruction Cycle for STAX D Instruction

Figure 4-6

A	77	a	
B		b	7C
C	39	c	58
D		d	5E
E	79	e	
F	71	f	
G	3D	g	6F
H	76	h	34
I	06	i	04
J	1E	j	
K		k	
L	38	l	06
M	use N,N	m	use n,n
N	37	n	54
O	7F	o	5C
P	73	p	
Q		q	67
R		r	50
S	6D	s	
T		t	78
U	3E	u	1C
V	use U	v	use u
W	use U,U	w	use u,u
X		x	
Y	6E	y	
Z	5B	z	

HEX Codes and Characters
Figure 4-7

4.8.2 Copy a List to Display - Exercise

With the LDAX and STAX instructions it becomes easy to access data in successive memory locations. In this exercise we will create a sequence of characters translated into bit patterns and place this sequence into memory as we load the program. Then the program will copy the characters into the display.

Figure 4-7 gives HEX codes that can be used for most characters. Unfortunately K and X are impossible, M and W require double characters, and several others are not very good representations because of the physical limitations of a 7-segment display. Use this table to generate a list of characters to be displayed, and store the list starting at address 8300. For example:

```
8300    73
8301    5C
8302    06
8303    04
8304    58
8305    79
```

Now write a program, using MVI instructions to load register pair BC with the address of your list (8300); pair DE with the address of the display (83F8), and Register L with the number of characters. Use these addresses to copy the list into the display.

```
OA      LDAX B      Load Character
12      STAX D      Copy to display
```

Increment the addresses in Register C and Register E; (the high bytes in B and D should not change). Count down in Register L and repeat (use JNZ) until the required number of characters have been copied. Finally jump back to the starting location (8200).

Write and code your program. Step through the program to test the program flow, but do not expect to see any results in the display while you are stepping. The monitor program uses the same display by writing to the same memory locations you are using. After the first time the JNZ instruction is executed, look at the registers to make sure they contain the correct addresses and count.

```
REG     B                       820A    B-83
NEXT                            820A    C-01
NEXT                            820A    D-83
NEXT                            820A    E-F9
REG     L                       820A    L-05(?)
```

The count in Register L should now be one less than the number of characters, since it has counted down once. The given solution (Figure 4-8) has six characters. Your program may have fewer or more, but not more than eight, since that is the size of our display.

COPY LIST TO DISPLAY

ADDR		CODE														
8 20	0	06		MVI				B,	83							Address list with BC
	1	83														
	2	0E		MVI				C,	00							
	3	00														
	4	16		MVI				D,	83							Address display with DE
	5	83														
	6	1E		MVI				E,	F8							
	7	F8														
	8	2E		MVI				L,	06							Character Count
	9	06														
820	A	0A		LDAX				B								Character from list to Display
	B	12		STAX				D								
	C	0C		INR				C								Next Character
	D	1C		INR				E								Next Digit
	E	2D		DCR				L								Count
	F	C2		JNZ					820A							Repeat until finished
8 21	0	0A														
	1	82														
	2	C3		JMP					8200							Start again
	3	00														
	4	82														
	5															
	6															
	7															
	8															
	9															
	A															
	B															
	C															
	D															
	E				LIST				OF			CH				ARACTERS
	F															
3 30	0	73														
	1	5C														
	2	06														
	3	04														
	4	58														
	5	79														
	6															
	7															
	8															

Figure 4-8

4.8.3 Display of Eight Characters

If you display exactly eight characters in the preceding program you can make use of the fact that the final display location is 83FF. When the display has been fully loaded, the INR E instruction will count to 00, setting the zero flag. In your program, replace the DCR L instruction with NOP (HEX code 00). Now exactly eight characters will be displayed. If you want any blank characters, put zeros in the table to turn off all segments.

This page intentionally left blank.

4.8.4 Register Pair Loading - LXI

Because it is so common to use register pairs for addressing memory, the 8080 includes special load immediate and counting instructions for register pairs.

```
01      LXI  B, address
xx      (low byte of address - to Register C)
xx      (high byte of address - to Register B)

11      LXI  D, address
xx      (low byte of address - to Register E)
xx      (high byte of address - to Register D)

21      LXI  H, address
xx      (low byte of address - to Register L)
xx      (high byte of address - to Register H)
```

These instructions are similar to the MVI instructions, except that two bytes of data follow the op-code and two registers are loaded. Note that we will write the addresses in a mnemonic instruction in the conventional way, with high byte first:

```
LXI  D,  8300
```

When this is translated into 8080 machine language we must follow the 8080 convention (as in JMP instructions) with low byte first, then high byte:

```
11      LXI  D,  8300
00      (low byte of address)
83      (high byte of address)
```

In transfer notation we use the abbreviation rp to designate any one of the register pairs. The LXI instructions can then be defined as:

LXI rp, address

(low register of pair) <- (byte 2)

(high register of pair) <- (byte 3)

No flags are affected.

In your program for copying a list to the display, replace the MVI instructions with LXI instructions.

Change These		To These	
06	MVI B,83	01	LXI B, 8300
83		00	
0E	MVI C,00	83	
00		00	NOP
16	MVI D,83	11	LXI D, 83F8
83		F8	
1E	MVI E,F8	83	
F8		00	NOP

The program operation will be unchanged.

4.8.5 Register Pair Counting - INX, DCX

In the program for copying a list to the display we started the list at 8300, so for eight characters it ended at 8307. Suppose the list were to start at 82FF. Then the first INR C instruction would advance Register C to 00, but Register B would not be affected and the address in B,C would be 8200. The 8080 includes register pair counting instructions, which will count a sixteen bit number in a pair.

03	INX	B	OB	DCX	B
13	INX	D	1B	DCX	D
23	INX	H	2B	DCX	H

Again using rp to designate a register pair:

```
INX  rp    (rp) <- (rp) + 1
           No flags are affected
DCX  rp    (rp) <- (rp) - 1
           No flags are affected
```

Note that the register pair counting instructions do <u>not</u> affect any flags. In the modified "Copy List to Display" program, using the count of Register E to terminate the loop, we must continue to use INR E, since INX D would fail to terminate the loop at 8400. We can use INX B to address the list, and we are then not constrained to start the list at any particular place. Figure 4-9 shows the fully modified version of the given "Copy List to Display" program.

COPY LIST TO DISPLAY

ADDR		CODE													
8200	0	01		LXI		B	,	82	FF						Address List
	1	FF													(C) ← FF
	2	82													(B) ← 82
	3	11		LXI		D	,	83	F8						Address Display
	4	F8													(E) ← F8
	5	83													(D) ← 83
	6	00		NOP											Spare bytes
	7	00		NOP											
	8	00		NOP											Discard original
	9	00		NOP											MVI L
820A	A	0A		LDAX		B									Character from List
	B	12		STAX		D									Copy to Display
	C	03		INX		B									Next Character
	D	1C		INR		E									Next Digit
	E	00		NOP											
	F	C2		JNZ				820A							Repeat until finished
8210	0	0A													
	1	82													
	2	C3		JMP				8200							Back to start
	3	00													
	4	82													
	5														
	6														
	7														
	8														
	9														
	A														
	B														
	C														
	D														
	E			LIST				OF				CH			ARACTERS
82F	F	00		BLANK											
8300	0	73		P											
	1	5C		O											
	2	06		L											
	3	04		I											
	4	58		C											
	5	79		E											
	6	00		BLANK											
	7	00		BLANK											
	8														

Figure 4-9

4.8.6 Delay Loops

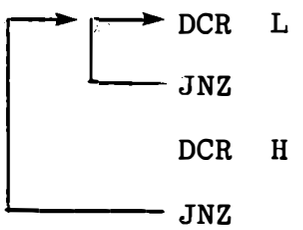
Although most of the operations we have performed with the computer appear to happen instantaneously, in fact each step in the computer takes a defined time to occur. If a delay of a specific length of time is desired it is easy to achieve, provided that the computer has nothing else to do. The trick is to perform some simple operation a very large number of times.

We will cause the display we created in the previous exercises to appear gradually by inserting a delay loop between characters. The program description becomes:

1)	Address List	(BC) <- 82FF
2)	Address Display	(DE) <- 83F8
3)	Copy one character to display	((DE)) <- ((BC))
4)	Set Delay	(HL) <- 0400
5)	Count Delay down to zero	
6)	Next List Addresses	(BC) <- (BC) + 1
7)	Next Display Digit	(E) <- (E) + 1
8)	Repeat from 3 until finished	
9)	Clear the display	
10)	Repeat from start	

This will load the display as before but with a delay between characters. Once loaded, the display will be turned off by writing zero into all the display locations, and the process will be repeated.

Steps 1, 2, 3 and 6, 7, 8 are the same steps we have been using. Step 4 uses another LXI instruction (LXI H,0400). The delay sequence is:



Register L repeatedly counts down from 00, FF, FE --- 01, 00. The final count sets the Zero flag and register H is counted once. Then L is counted down from 00 again, and so on until Registers H and L have both reached zero. (Be sure you understand this - study the sequence above carefully). For an 8080 running at normal speed this delay loop takes 3855 clocks or .001882 second for each count in Register H. Since we started with a count of 4 in Register H, the delay would be only 7.5 milliseconds (.0075 second) at full speed, still an imperceptible time. Because we are using the MTS monitor your program is executed much more slowly, and the value given is suitable for our purpose. The slow operation is explained in the next section.

Note that we have placed the address incrementing instructions (INX B, INR E) after the delay. The delay count uses the Zero flag, so the INR E instruction must follow the delay so that it can terminate the loop for displaying digits.

To clear the display we can again load its address into (DE) and write zeros into all eight locations.

Write this program yourself, referring to the program description. Then compare your results with our solution. (Figure 4-10). The next section describes a new technique for testing the program flow.

GRADUAL DISPLAY WITH CLEAR

ADDR		CODE													
8200	0	01		LXI		B,	82FF								Address List
	1	FF													
	2	82													
	3	11		LXI		D,	83F8								Address Display
	4	F8													
	5	83													
8206	6	0A		LDAX		B									
	7	12		STAX		D									
	8	21		LXI		H,	0400								Set Delay
	9	00													
	A	04													
820D	B	2D		DCR		L									Delay Loop
	C	C2		JNZ			820B								
	D	0B													
	E	82													
	F	25		DCR		H									
8210	0	C2		JNZ			820B								
	1	0B													
	2	82													
	3	03		INX		B									Next Character
	4	1C		INR		E									Next Digit
	5	C2		JNZ			8206								Repeat until finished
	6	06													
	7	82													
	8	11		LXI		D,	83F8								Address Display for clearing
	9	F8													
	A	83													
	B	AF		XRA		A									(A) ← 00
821C	C	12		STAX		D									Digit ← 00
	D	1C		INR		E									Next Digit
	E	C2		JNZ			821C								Repeat until clear
	F	1C													
8220	0	82													
	1	C3		JMP			8200								Back to start
	2	00													
	3	82													
	4														
	5			SAME		DATA									LIST AT 82FF
	6			AS		FIGURE									4-9
	7														
	8														

Figure 4-10

4.8.7 Breakpoints

We have used the MTS monitor to step through programs to test the program flow and look for errors. In a program that has short repetitive loops this is a little tiresome; when a loop such as the delay in this program is repeated a large number of times it is impractical to step through it. You would have to press STEP more than 16,000 times to step all the way through this program.

The monitor has a powerful feature that avoids repeated stepping, yet allows you to test program flow thoroughly.

Using the program solution given in Figure 4-10 we shall demonstrate the breakpoint ability of the monitor. (Be sure that the toggle switch at the left of the circuit board is in the "single step" position.)

RESET					8200	01
ADDR	8	2	0	6	8206	0A
BRK					8206	BP.

Do not press RESET again after the next steps.

We have set a breakpoint at 8206, the LDAX B instruction.

```
        ADDR                                    8200    01
```

This displays the present program address, at present the start of the program.

```
        RUN                                     8206    0A
```

Your program was executed until it reached the instruction whose address you entered as a breakpoint. This instruction has not yet been executed.

```
        STEP                                    8207    12
        STEP                                    8208    21
        STEP                                    820B    2D
        STEP                                    820C    C2
        STEP                                    820B    2D
```

We have now started the long countdown in Register L. We have 255 steps to go.

```
        REG     L                               820B    L-FF
```

Now we know that this piece of the program is operating. Enter another breakpoint after this loop, at DCR H.

```
ADDR    8       2       0       F               820F      25
BRK                                             820F    BP.
RUN                                       (Z)   820F    L-00
```

The first segment of the delay loop has been executed and we have reached the breakpoint at 820F. The last register we displayed is shown again, just as though we had stepped 255 times. Register L has counted down to zero (note that the zero flag is set) and we are ready to count in Register H.

```
REG     H                                 (Z)   820F    H-04
STEP                                            8210    H-03
STEP                                            820B    H-03
RUN                                       (Z)   820F    H-03
RUN                                       (Z)   820F    H-02
RUN                                       (Z)   820F    H-01
```

Note that we are always seeing the Zero flag set from counting down in Register L.

The program is stopped before we execute the DCR H. Now we are about to count H down to zero.

```
STEP                          (Z)  8210   H-00
ADDR                          (Z)  8210     C2
STEP                          (Z)  8213   H-00
```

The JNZ (C2) instruction was not executed. We can watch the addresses being incremented.

```
REG     C                     (Z)  8213   C-FF
STEP                          (Z)  8214   C-00
```

The Zero flag is still set from the previous DCR H. The next time around we shall see that when (BC) becomes 8301 the Zero flag is not affected.

Now we should STEP to be sure that the next untested instructions are correct.

```
REG     E                     (Z)  8214   E-F8
STEP                               8215   E-F9
STEP                               8206   E-F9
ADDR                               8206     0A
```

This is the LDAX B instruction.

```
REG     A                          8206   A-00?
```

Register A still contains the first character of the list.

```
STEP                               8207   A-73?
```

We have loaded the second display character. When we press RUN that character will appear momentarily on the display before we reach the breakpoint at 820F.

RUN				P
		(Z)	820F	A-73
REG	H	(Z)	820F	H-04
RUN		(Z)	820F	H-03
RUN		(Z)	820F	H-02
RUN		(Z)	820F	H-01

We are about to leave the delay loop.

STEP	(Z)	8210	H-00
STEP	(Z)	8213	H-00

Watch REG C and the Zero flag.

REG	C	(Z)	8213	C-00
STEP		(Z)	8214	C-01

As promised, INX B did not affect Zero. We need not continue to observe this part of the program, but we might want to see each character displayed.

We shall clear the breakpoint at 820F, but leave the breakpoint at 8206. Press BRK to display the breakpoint:

```
BRK                                    (Z)  820F   BP.00
CLR                                    (Z)  8206   BP.00
```

Clear removes the breakpoint displayed and shows the other one.

```
ADDR                                   (Z)  8214     1C
RUN                                         8206   C-01
RUN
```

The third character appeared momentarily and we are about to send the fourth. If you are now satisfied that this part of the program works we can clear the breakpoint at 8206, and insert a new breakpoint at 8218, just before the display is cleared.

```
BRK                                         8206   BP.00
CLR                                                BP.
```

No breakpoints remain.

```
ADDR   8      2      1      8                8218     11
BRK                                         8218   BP.
ADDR   MEM                                  8206    .0A
```

Pressing ADDR shows the instruction. MEM tells the monitor to display the instruction instead of a register.

Now run the remainder of the program.

```
        RUN                                                    licE

                                                    (Z)  8218      11
```

After the rest of the message was shown we reached the breakpoint at 8218 when Register E counted to zero.

Step through the display clearing loop once, and then practice what you have learned by setting breakpoints at the JNZ and JMP instructions. After a couple of times through the clearing loop, remove the breakpoint at JNZ, and watch the program stop at the JMP.

Finally, remove all breakpoints by pressing RESET, and run the whole program.

It was pointed out in Section 4.8.6 that your program executes slowly because of the MTS monitor. Before each of your instructions is executed the monitor looks in its list of breakpoints to see whether your program counter has reached one of them. This is done by the 8080 executing the monitor program. For every one of your instructions that is executed the 8080 executes at least 68 instructions of the monitor program. When you have entered breakpoints some of these must be executed in repetitive loops, making the process even slower. You can make your program run at full speed, after it is tested and operates correctly, by switching the monitor off. At the left edge of the MTS circuit board there is a switch. In its low position (STEP) the monitor is active; in AUTO the monitor is inactive.

4.8.8 Review and Self Test

This section has introduced register pairs, and used them to address memory. We have practiced the use of the MTS Display and used repetitive loops to address successive locations in memory and to generate a time delay. Monitor breakpoints were introduced. Test your knowledge with this quiz.

1) Identify the three register pairs, and tell which register is used for the high byte and the low byte of an address stored in the pair. (Sections 4.8, 4.8.1)

Register Pair Name	High Byte	Low Byte
_____	_____	_____
_____	_____	_____
_____	_____	_____

2) Describe the following instructions using transfer notation. (Sections 4.8.1, 4.8.4, 4.8.5)

LXI	B,	address	_____
INX	D		_____
LDAX	D		_____
STAX	B		_____

3) Which flags, if any, are affected by each of the above instructions? (Sections 4.8.1, 4.8.4, 4.8.5) _____

4) Give the MTS key sequence to set a breakpoint at address 8218. (Section 4.8.7) _____

5) Create a bit pattern to display the numeral 3, and translate it into hexadecimal. (Section 4.7.1) _____

6) Give the two instructions to display a 3 in a digit addressed by (DE). _____ _____

7) What hexadecimal value should be written to a display location for a blank digit? (Section 4.7.1) _____

Answers to Self Test - Section 4.7.8

1) The register pairs are:

B	B Stores the high byte C Stores the low byte
D	D Stores the high byte E Stores the low byte
H	H Stores the high byte L Stores the low byte

2) LXI B, address (BC) <- address
or (C) <- byte 2 of instruction (low address)
(B) <- byte 3 of instruction (high address)

INX D (DE) <- (DE) + 1

LDAX D (A) <- ((DE))

STAX B ((BC)) <- (A)

3) None of the above instructions affects any flags.

4) To set a breakpoint press

ADDR 8 2 1 8 BRK

5) Bit Pattern for 3 = 01001111 = 4F

6) To display 3 at (DE)

```
MVI  A,  4F
STAX D
```

7) Hexadecimal 00 gives a blank.

4.9 USE OF A MEMORY LOCATION AS A REGISTER

Register pair HL is primarily intended for addressing memory, and the memory location addressed by (HL) is available to the CPU as though it were another register. All of the register reference instructions (MOV, MVI, INR, DCR, ADD, ADC, SUB, SBB, and others not yet presented) have counterparts that perform the same function using the memory location addressed by (HL). The flags are affected as though the memory location were a general purpose register.

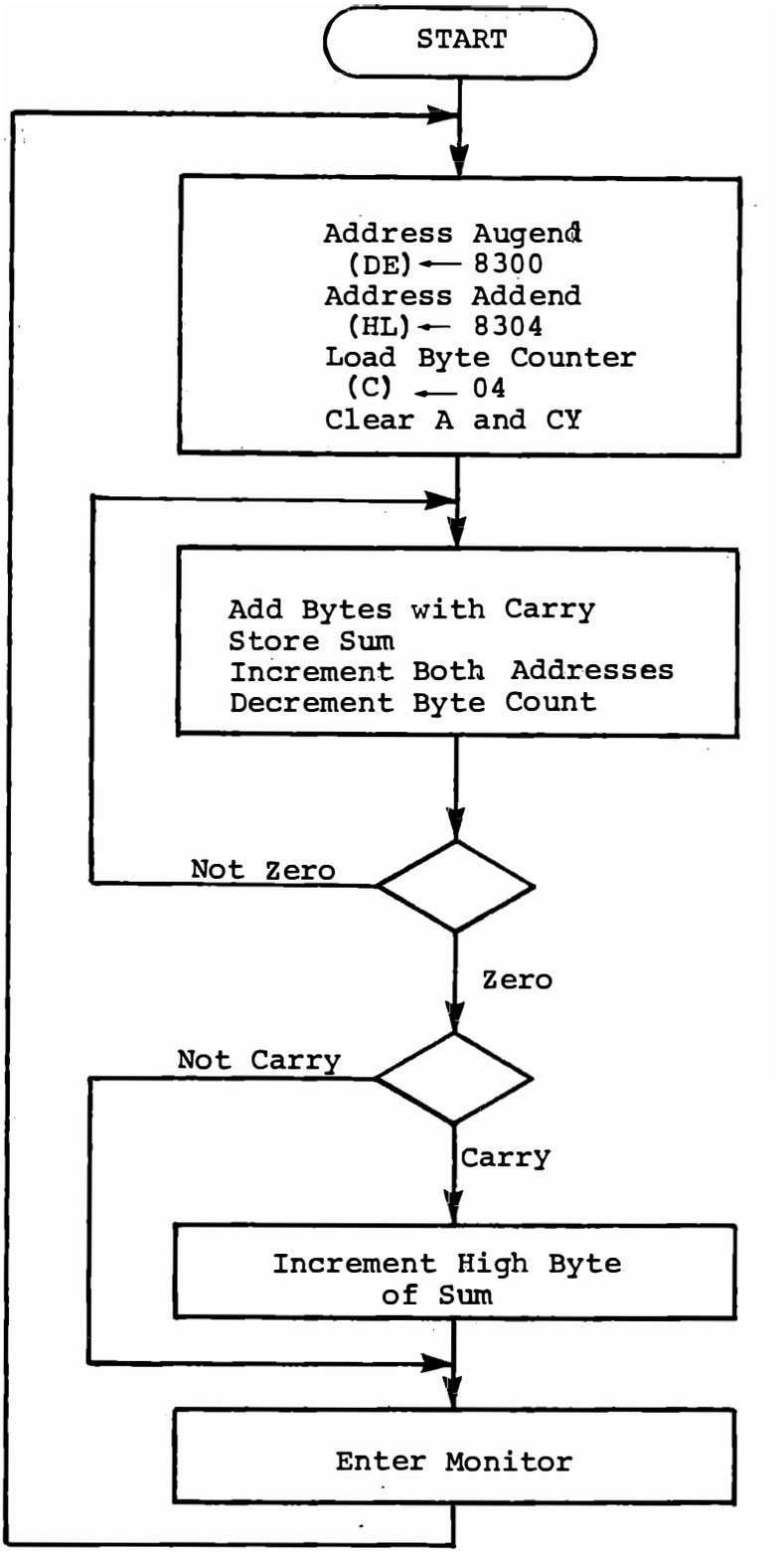
Before carrying out an exercise involving this type of memory addressing, we will formally define some instructions involving memory reference. Note that in transfer notation parentheses mean "the content of", so (HL) refers to the content of register pair HL. Doubled parentheses such as ((HL)) mean "the content of the memory location addressed by the content of register pair HL". In memory reference instructions that treat this memory location as a register, we use M to designate the register. For example: INR M. Thus (M) is always equal to ((HL)). Instead of LDAX H and STAX H we have equivalent instructions.

7E	MOV	A,M	(A) <- ((HL))
77	MOV	M,A	((HL)) <- (A)

4.9.1 Memory Reference Instructions

HEX	MNEMONIC	MEANING
34	INR M	Increment Memory ((HL)) <- ((HL)) + 1 Increment the content of the memory location addressed by the content of register pair HL. If ((HL)) becomes 0 then (Z)<-1 else (Z)<-0 The Carry flag is not affected.
35	DCR M	Decrement Memory ((HL)) <- ((HL)) - 1 Decrement the content of the memory location addressed by the content of register pair HL. If ((HL)) becomes 0 then (Z)<-1 else (Z)<-0 The Carry flag is not affected.
	MOV M,s (See Section 4.11.4 for hex codes)	Move into memory from register ((HL)) <- (s) The memory location addressed by the register pair HL is loaded with the content of source register s. The flags are not affected. The content of s is not affected.
	MOV d,M (See Section 4.11.4 for hex codes)	Move into register from memory (d) <- ((HL)) Destination register d is loaded with the content of the memory location The flags are not affected. The content of the memory location is not affected.
36 xx	MVI M,data	Move immediate data into memory ((HL)) <- (byte 2) The memory location addressed by register pair HL is loaded with the content of byte 2 of the instruction. The flags are not affected.

86	ADD M	Add memory to accumulator (A) <- (A) + ((HL)) The content of the memory location addressed by register pair HL is added to the content of Register A and the result is placed in Register A. The content of the memory location is not affected. If (A) becomes 0 then (Z) <- 1 else (Z) <- 0 If the result of the addition is greater than FF (ie a carry occurs) then (CY) <- 1 else (CY) <- 0
8E	ADC M	Add memory to accumulator with Carry. (A) <- (A) + ((HL)) + CY Flags are affected as in ADD M
96	SUB M	Subtract memory from accumulator (A) <- (A) - ((HL)) The content of the memory location addressed by (HL) is subtracted from the content of register A and the result is placed in register A. The content of the memory location is not affected. If the result is zero the Zero flag is set. Otherwise the Zero flag is reset. If the content of the memory location was greater than the original content of A then Carry is set to indicate a borrow. Otherwise Carry is reset.
9E	SBB M	Subtract memory from accumulator with borrow. (A) <- (A) - ((HL)) - CY Flags are affected as in SUB M.



Four Byte Addition in Memory

Figure 4-11

4.9.2 Four Byte Addition Exercise

The use of ((HL)) as a register makes it easy to do arithmetic with numbers that are too large (i.e., require too many bytes) to be kept in the working registers. For example: add two numbers of four bytes each and replace one of them (called the addend) with the sum. Allow the sum to occupy five bytes (since it might be as great as 01FFFFFFFE). Figure 4-11 is a flow chart for the program. We shall use Register C for a byte counter; DE for the address of the augend (the number to be added to the addend) and HL to address the addend. The augend is stored at 8300 - 8303; the addend and sum at 8304 - 8308.

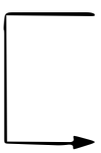
Because we shall do the multi-byte addition in a loop, we must use the ADC addition instruction. Carry must be cleared before the first addition. We have peviously used:

```
AF      XRA     A
```

to clear Register A; the same instruction also clears the Carry flag. The addition loop is:

```
        XRA     A
LOOP    LDAX    D
        ADC     M
        MOV     M,A
        INX     D
        INX     H
        DCR     C
        JNZ     LOOP
```

At the end of this loop Carry is set if the sum is too great for four bytes. Then either of these techniques can be used:



```
JNC                 MOV     A,M
INR  M              ACI     00
RST  4              MOV     M,A
JMP  START          RST 4
                    JMP     START
```

We have used ADC M and INR M, treating M or ((HL)) as though it were a register. A program solution is given in Figure 4-12.

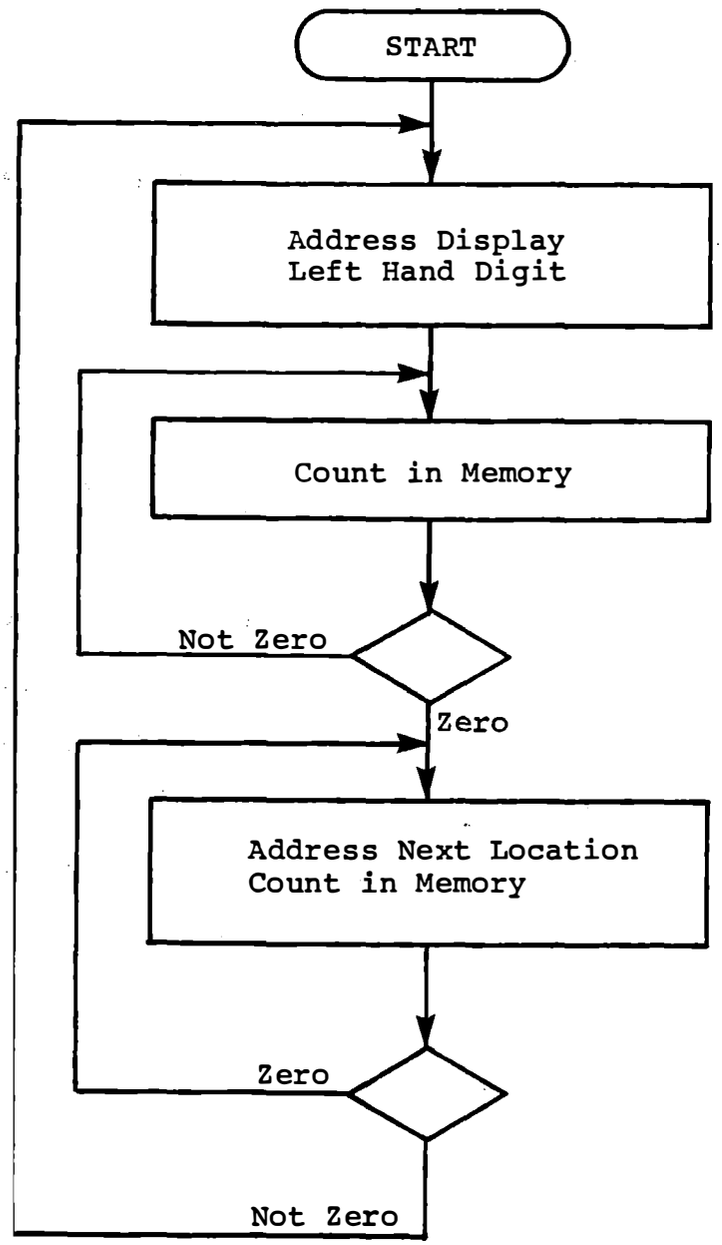
For additional practice, convert this into a multiplication program.

FOUR BYTE ADDITION IN MEMORY

CODING SHEET

ADDR		CODE													
8 20	0	11		L	X	I		D	,	8	3	0	0		Address (augend)
	1	00													
	2	83													
	3	21		L	X	I		H	,	8	3	0	4		Address (addend)
	4	04													
	5	83													
	6	0E		M	V	I		C	,	0	4				Load byte Counter
	7	04													
	8	AF		X	R	A		A							Clear A and CY
820	9	1A		L	D	A	X		D						
	A	8E		A	D	C		M							
	B	77		M	O	V		M	,	A					
	C	13		I	N	X		D							
	D	23		I	N	X		H							
	E	0D		D	C	R		C							
	F	C2		J	N	Z			8	2	0	9			
8 21	0	09													
	1	82													
	2	D2		J	N	C			8	2	1	6			At exit from loop
	3	16													Jump if no carry
	4	82													LH Carry set
	5	34		I	N	R		M							Add 1 into high byte
821	6	E7		R	S	T	4								Enter Monitor
	7	C3		J	M	P			8	2	0	0			
	8	00													
	9	82													
	A														
	B														
	C														
	D														
	E														
	F														
3	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure 4-12

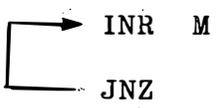


Counting in the Display

Figure 4-13

4.9.3 Counting in the Display - Exercise

A trivial but amusing use of the INR M instruction allows us to view a counting operation as it occurs. Since the display is controlled by eight specified memory locations, we can count in those locations and see the effect on the display. Figure 4-13 shows the program flow chart. The left hand digit of the display memory counts very rapidly, using only two instructions:



With the monitor disabled (set the STEP/AUTO switch to AUTO) this loop is executed once in 10 microseconds. A full cycle in that digit takes about .00256 second. The second digit counts 256 times more slowly; allowing for the extra instructions, but a clock rate slightly greater than 2 MHZ, a full cycle in the second digit takes 0.646 second, and the third completes a cycle in 165 seconds. How long will it be before the display is all blank again?

4.10 INDIRECT ADDRESSING

We have previously described LDAX B (or MOV A,M) as "indirect addressing". This is Intel usage of the phrase, but more conventionally indirect addressing implies taking an address from one location in memory to point to another memory location. This can be done in two ways in the 8080.

```
LXI     H,      8300
MOV     C,M
INX     H
MOV     B,M
```

Now register pair BC contains an address which was (and still is) stored in memory locations 8300 and 8301. This technique is very powerful, as we shall see in the later exercises. A program can store an address in memory, and later use that address to find desired data.

4.10.1 Load and Store HL Direct

In order to use a memory location as a working register, its address must be in register pair HL. We can load an address into pair BC as above and then copy it to HL by using MOV L,C and MOV H,B. It is so important to be able to do this kind of function that the 8080 provides an instruction to do it:

2A	LHLD Address	Load H and L Direct
xx	(low address)	(L) <- (address)
xx	(high address)	(H) <- (address + 1)
		No flags are affected.

This is a three byte instruction similar to LDA address, but it loads two bytes of data from memory. The byte stored at "address" is copied into Register L, and the following byte is copied into Register H. Be sure to understand the difference between LXI H address and LHLD address.

LXI H,	8300	(L) <- 00
		(H) <- 83
LHLD	8300	(L) <- (8300)
		(H) <- (8301)

The reverse function is also available:

22	SHLD Address	Store H and L Direct
xx	(low address)	(address) <- (L)
xx	(high address)	(address + 1) <- (H)

Note that these are called "direct" instructions because the program provides the address where the data are stored. Their principal use is for indirect addressing; having loaded H and L directly, we now use the information we loaded as an address to find other data.

```
LHLD    8300
MOV     A,M
```

We have loaded Register A from memory, using another pair of memory locations (8300, 8301) to provide an address.

4.10.2 LHLD and SHLD - Example

To make these instructions more clear, enter and step through this program:

```
8200    AF    XRA A       Clear A
8201    21    LXI H,8400  An address for data
8202    00
8203    84
8204    77    MOV M,A     Store datum
8205    22    SHLD  8300  Store address
8206    00
8207    83
8208    21    LXI H,FFFF  Discard the address
8209    FF                to prove it has
820A    FF                been stored
820B    7D    MOV A,L     Discard the datum
820C    2A    LHLD  8300  Recover the address
820D    00
820E    83
820F    7E    MOV A,M     Recover the datum
8210    23    INX H       Next address
8211    3C    INR A       Next datum
8212    C2    JNZ   8204  Repeat 256 times
8213    04
8214    82
8215    E7    RST 4       Enter monitor
8216    C3    JMP   8200
8217    00
8218    82
```

The following pages describe the results of this program as you step through it.

This program will store the content of A (MOV M,A) at the address contained in HL. At the beginning it stores 00 at address 8400, and stores 8400 at 8300 and 8301. Set the STEP/AUTO switch to step and go through the first six instructions.

RST	8200	AF
STEP	8201	21
STEP	8204	77
STEP	8205	22
STEP	8208	21
STEP	820B	7D
STEP	820C	2A

Now inspect the registers and memory locations.

REG	A				820C	A-FF
REG	H				820C	H-FF
NEXT	(the next register)				820C	L-FF
ADDR	8	4	0	0	8400	00
ADDR	8	3	0	0	8300	00
NEXT	(the next memory location)				8301	84

The registers contain garbage, but the initial value of A is stored at 8400 and that address is stored at 8300, 8301.

ADDR		820C	2A

This is the LHLD instruction. Watch H.

REG	H	820C	H-FF
STEP		820F	H-84
REG	L	820F	L-00

The address has been recovered by LHLD 8300, and (HL) now contains 8400.

REG	A	820F	A-FF

Register A still contains garbage but the next instruction (MOV A,M) will recover the data from (8400).

STEP		8210	A-00

Place a breakpoint here (at 8210) and step through the next several instructions.

ADDR	BRK	8210	BP.
STEP		8211	A-00
STEP		8212	A-01

Register pair H now contains the next address for data storage (8401) and Register A contains the next datum.

```
        STEP                                        8204    A-01
```

The new value in A will be stored at 8401 by MOV M,A and the new address will be stored at 8300, 8301.

```
        RUN                                         8210    A-01
```

We have gone through the store and recover instructions, so once again the address and datum have been recovered from memory by LHLD 8300 and MOV A,M.

```
        REG     H                                   8210    H-84
        NEXT                                        8210    L-01
        ADDR    8       4       0       1           8401     01
```

Register pair HL points to memory location 8401, which contains the datum 01, which we have already loaded into A.

We shall continue stepping through this program in the next section.

4.10.3 Examining a Register Pair

The MTS monitor provides a convenient means of examining a register pair and the memory location addressed by the register pair. Note that key 8 is also labelled H. This refers to register pair H.

```
        ADDR    H       MEM                         8401    HL.01
```

The monitor is now addressing the same memory location that is addressed by (HL).

```
        NEXT                                        8402      ??
```

Next displays the next memory location. It does not affect the contents of H and L.

```
        ADDR    H       MEM                         8401    HL.01
```

Run through the loop again.

```
        RUN                                         8210      23
        ADDR    H       MEM                         8402    HL.02
```

We can look backward in memory by pressing:

```
        MEM                                         8401     .01
        MEM                                         8400     .00
```

Repeat this a few more times.

```
        RUN                                          8210      23
        ADDR    H       MEM                          8403    HL.03
        RUN                                          8210      23
        RUN                                          8210      23
        ADDR    H       MEM                          8405    HL.05
```

Remove the breakpoint and run all the way.

```
        BRK                                          8210    BP.00
        CLR                                                  BP.
        RUN                                    (Z)   8216      C3
```

The program has run all the way and is ready to start over.

```
        REG     A                                    8216    A- 00
        ADDR    H       MEM                          8500    HL.??
```

Now look through the memory.

```
        ADDR    8400                                 8400      00
        NEXT                                         8401      01
        NEXT                                         8402      02
```

Review what has been done. During each loop we stored a data byte in a memory location (8400, then 8401, then 8402, etc.) and stored the address of that memory location in a pair of other memory locations. Then we used the registers for some other undefined purpose. Then we recovered the address and the data byte, incremented both, and repeated. The important points here are the storage of an address in memory so that it could be found later, and indirectly loading data from the addressed location.

4.10.4 Review and Self Test

The use of registers pairs for addressing memory, and the use of the memory location addressed by (HL) as a working register are extremely important features of the 8080 microprocessor. The next two exercises use these features. Before going on, test your knowledge.

1) Assume (for the program below) that memory contains:

8300	03
8301	83
8302	03
8303	06
8304	0A
8305	6F
8306	FF
8307	84

For each step in the following program indicate which registers and/or memory locations are affected, and give the content of the register or memory location after execution of the instruction.

			Register or Memory Location	Content
8200	LXI B,	8302		
8203	LDAX B			
8204	LXI H,	8304		
8207	ADD M			
8208	LHLD	8306		
820B	INX H			
820C	MOV M,A			
820D	SHLD	8306		
8210	LHLD	8300		
8213	DCR M			

2) Which instructions in the program above affect the Zero flag?

_____ _____

3) Which instructions affect Carry?

_____ _____

4) If you press the following keys after the instruction at 8213 has been executed, what will be displayed?

ADDR H MEM _____ _____

5) Neither of the following instructions exists in the 8080. What equivalent instructions do exist?

LDAX H _____

STAX H _____

6) There is no instruction to load BC or DE in the same way that LHLD loads HL. There are several ways to accomplish the same function with three or four instructions. Give three ways to load register pair B with the data stored at addresses 8300 and 8301. Which takes the fewest bytes of program memory?

Answers to Self Test, Section 4.10.4

1)

After execution of:				Register or Memory Location	Content
8200	LXI	B,	8302	C	02
				B	83
8203	LDAX	B		A	03
8204	LXI	H,	8304	L	04
				H	83
8207	ADD	M		A	0D
8208	LHLD		8306	L	FF
				H	84
820B	INX	H		L	00
				H	85
820C	MOV	M,A		8500	0D
820D	SHLD		8306	8306	00
				8307	85
8210	LHLD		8300	L	03
				H	83
8213	DCR	M		8303	05

2) The Zero flag is reset by:

8207 ADD M (result = OD)

8213 DCR M (result = 05)

3) The Carry flag is reset by:

8207 ADD M

4) ADDR H MEM displays 8303 HL.05

5) Instead of LDAX H use MOV A,M

   Instead of STAX H use MOV M,A

6) To load BC with data from memory locations 8300 and 8301:

   a) LDA 8301
      MOV C,A
      LDA 8300
      MOV B,A
      This takes 8 bytes of program memory.

   b) LXI H, 8300
      MOV C,M
      INX H
      MOV B,M
      This takes 6 bytes of program memory.

   c) LHLD 8300
      MOV C,L
      MOV B,H
      This takes 5 bytes of program memory.

4.11 COMPARISONS AND CONDITIONAL JUMPS

We have repeatedly used the Zero flag in counting and repeating a loop (DCR, JNZ).

We have used the Carry flag in arithmetic in several exercises: setting or resetting the flag by ADD and SUB; using it in ADC or ACI and SBB or SBI; and demonstrated a conditional jump (JNC) in one arithmetic program. There are a number of other ways to set or reset the flags, and they are most often used with the conditional jumps.

Review the four conditional instructions that have been introduced so far:

C2 xxxx	JNZ	address	Jump if not zero
CA xxxx	JZ	address	Jump if zero
D2 xxxx	JNC	address	Jump if not carry
DA xxxx	JC	address	Jump if carry

Recall that both Zero and Carry are affected by arithmetic and logic instructions. Zero is affected by single register counting instructions (INR, DCR) but not by register pair counting (INX, DCX). Carry is not affected by counting. Data movement instructions and jump instructions do not affect any flags.

4.11.1 Comparison Instructions - CMP

In addition and subtraction the Carry and Zero flags were set or reset as a result of the arithmetic operation. There is a set of comparison instructions whose only function is to affect the flags. These instructions permit a program to determine whether the content of Register A is greater than, equal to, or less than the content of any specified general purpose register (including M). The operation is identical to subtraction except that the numeric result is discarded instead of being placed in Register A.

For comparing Register C with Register A the instruction is:

HEX CODE:	B9
MNEMONIC:	CMP C
MEANING:	Subtract the content of C from the content of A and set the flags accordingly. The content of A is not changed.

This sets or clears the Zero and Carry flags as follows:

	Zero	Carry
A greater than C	Cleared	Cleared
A equal to C	Set	Cleared
A less than C	Cleared	Set

4.11.2 Compare Immediate Instruction - CPI

The CPI instruction compares the content of the immediately following data byte with the content of A.

HEX CODE:	FE
SECOND BYTE:	Data
MNEMONIC:	CPI
MEANING:	Subtract the value in the immediately following byte from the content of A. Set or reset all flags to reflect the The content of of A is not changed.

For all of the arithmetic and logical instructions that operate on data in Register A and one general purpose register, there are corresponding immediate instructions. These may be thought of as referring to a phantom register, created just to provide a desired data byte.

4.11.3 Moving Message - Exericse

In our previous display exercises we have been limited by the eight digit display. Here we shall output a longer message, shifting it across the display. The message will be terminated by a character with a period (decimal point) and then it will start again. Recall that the decimal point in a display digit is controlled by bit 7 in the byte written to the display memory, so:

79 = "E"

F9 = "E."

We can test for the decimal point by:

```
CPI  80
```

Any character that does not have a period or decimal point is less than 80 (see Figure 4-7) so CPI 80 must set Carry unless a period is present. Thus we can continue a loop to shift the display as long as this instruction sets Carry; when the period appears we will restart the message.

The procedure to be used is this:

1) Copy 8 bytes of message to display. If the end of the message is reached, continue from the start of the message until the display is filled.

2) Delay

3) Examine the character displayed at the left. If it contains a period, address the start of the message. If not, address the next following character in the message table.

4) Repeat from (1).

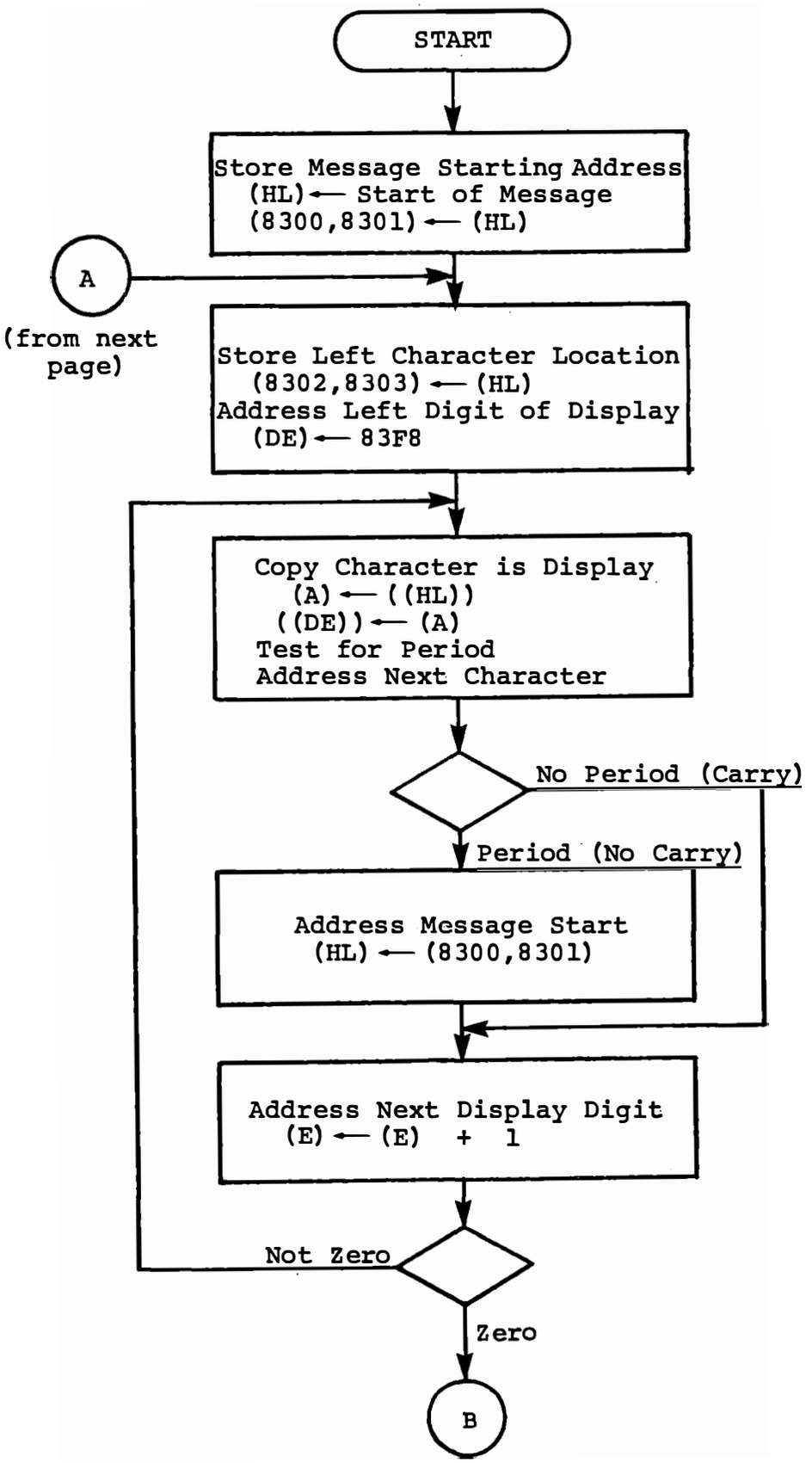
We need to keep track of two message addresses - the start of the message and the message location most recently displayed at the left. During Step 1 we will increment the message address eight times and then discard the final address. The starting location and the most recent left hand location will be kept in memory.

8300, 8301	Message start location
8302, 8303	Most recent left character location

A program flow chart is shown in Figure 4-14.

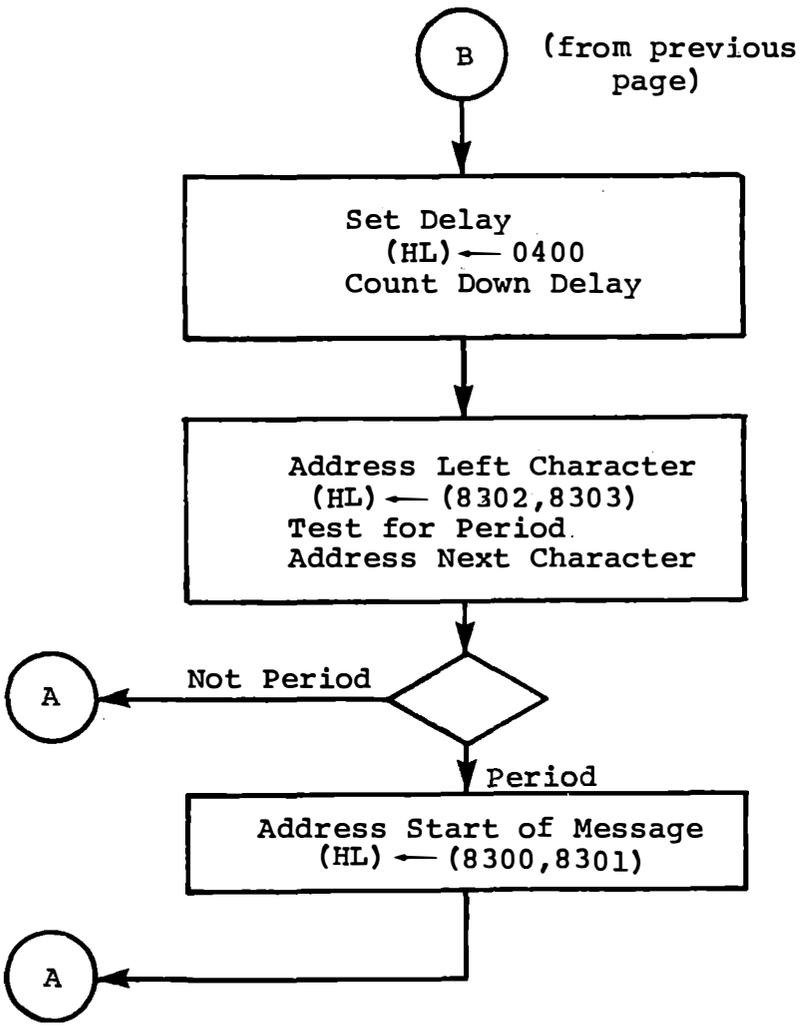
Write and code this program yourself. The next section lists all of the instructions we have introduced so far. Generate a message using the characters from Figure 4-7 and store it in memory, or else use one of two character tables that are in Read Only Memory -- at 02B3 or 0326. One of these displays the HEX characters and the other displays the register names, followed by some garbage characters. Your own message can be more interesting.

A program solution is given in Figure 4-15, following Section 4.11.4.



Moving Message

Figure 4-14a



Moving Message (continued)

Figure 4-14b

4.11.4 List of Instructions

Re-enter Monitor

E7	RST4	(applies to ICS Microcomputer Training System only.)

Jump and Conditional Jump Instructions

C3	JMP address	Unconditional Jump
xx	(low address)	
xx	(high address)	
C2	JNZ address	Jump if Not Zero
xx	(low address)	xx
xx	(high address)	xx
CA	JZ address	Jump if Zero
xx	(low address)	xx
xx	(high address)	xx
D2	JNC address	Jump if Not Carry
xx	(low address)	
xx	(high address)	
DA	JC address	Jump if Carry
xx	(low address)	
xx	(high address)	

Data Transfer Instructions

3A	LDA address	32	STA address
xx	(low address)	xx	(low address)
xx	(high address)	xx	(high address)
0A	LDAX B	02	STAX B
1A	LDAX D	12	STAX D
78	MOV A,B	47	MOV B,A
79	MOV A,C	4F	MOV C,A
7A	MOV A,D	57	MOV D,A
7B	MOV A,E	5F	MOV E,A
7C	MOV A,H	67	MOV H,A
7D	MOV A,L	6F	MOV L,A
7E	MOV A,M	77	MOV M,A

Other register-to-register MOV instructions are tabulated below.

	SOURCE REGISTER							
	A	B	C	D	E	H	L	M
MOV A,s	7F	78	79	7A	7B	7C	7D	7E
MOV B,s	47	40	42	42	43	44	45	46
MOV C,s	4F	48	49	4A	4B	4C	4D	4E
MOV D,s	57	50	51	52	53	54	55	56
MOV E,s	5F	58	59	5A	5B	5C	5D	5E
MOV H,s	67	60	61	62	63	64	65	66
MOV L,s	6F	68	69	6A	6B	6C	6D	6E
MOV M,s	77	70	71	72	73	74	75	-

Immediate Data Transfer

3Exx	MVI	A,	data
06xx	MVI	B,	data
0Exx	MVI	C,	data
16xx	MVI	D,	data
1Exx	MVI	E,	data
26xx	MVI	H,	data
2Exx	MVI	L,	data
36xx	MVI	M,	data

None of the above instructions affect any flags.

Register Pair Data Transfer Instructions

```
01      LXI  B  address
xx      (low address)
xx      (high address)

11      LXI  D, address
xx      (low address)
xx      (high address)

21      LXI  H, address
xx      (low address)
xx      (high address)

2A      LHLD    address
xx      (low address)
xx      (high address)

22      SHLD    address
xx      (low address)
xx      (high address)
```

Register Pair Counting Instructions

03	INX B	OB	DCX B
13	INX D	1B	DCX D
23	INX H	2B	DCX H

None of the above affect any flags.

Counting Instructions

These counting instructions set or reset Zero. The Carry flag is not affected.

3C	INR	A	3D	DCR	A
04	INR	B	05	DCR	B
0C	INR	C	0D	DCR	C
14	INR	D	15	DCR	D
1C	INR	E	1D	DCR	E
24	INR	H	25	DCR	H
2C	INR	L	2D	DCR	L
34	INR	M	35	DCR	M

Arithmetic Instructions

Zero and Carry are set or reset by these instructions.

87	ADD	A	8F	ADC	A
80	ADD	B	88	ADC	B
81	ADD	C	89	ADC	C
82	ADD	D	8A	ADC	D
83	ADD	E	8B	ADC	E
84	ADD	H	8C	ADC	H
85	ADD	L	8D	ADC	L
86	ADD	M	8E	ADC	M
C6	ADI	data	CE	ACI	data
data			data		
97	SUB	A	9F	SBB	A
90	SUB	B	98	SBB	B
91	SUB	C	99	SBB	C
92	SUB	D	9A	SBB	D
93	SUB	E	9B	SBB	E
94	SUB	H	9C	SBB	H
95	SUB	L	9D	SBB	L
96	SUB	M	9E	SBB	M
D6	SUI	data	DE	SBJ	data
data			data		

B8	CMP	A
B9	CMP	B
BA	CMP	C
BB	CMP	D
BC	CMP	E
BD	CMP	H
BE	CMP	L
BF	CMP	M
FE	CPI	data
data		

MOVING MESSAGE

ADDR		CODE													
8 20	0	21		L	X	I		H	,	02	B3				Start of Message
	1	B3													1234567 - - - FF4.
	2	02													(or - use 0326)
	3	22		S	H	L	D		83	00					Store start of
	4	00													message
	5	83													
820	6	22		S	H	L	D		83	02					Store as next
	7	02													character location
	8	83													
	9	11		L	X	I		D	,	83	F8				
	A	F8													
	B	83													
820	C	7E		M	O	V		A	,	M					Display Loop
	D	12		S	T	A	X		D						(Display) ← Char
	E	FE		C	P	I		80							Test for period
	F	80													
8 21	0	23		I	N	X		H							Address next char
	1	DA		J	C				82	17					Jump if not period
	2	17													
	3	82													
	4	2A		L	H	L	D		83	00					H period address
	5	00													Start of message
	6	83													
821	7	1C		I	N	R		E							Next digit
	8	C2		J	N	Z			82	0C					Loop ti display
	9	0C													eight digits!
	A	82													
	B	21		L	X	I		H	,	10	00				Set Delay Time
	C	00													
	D	10													
821	E	2D		D	C	R		L							Count Delay
	F	C2		J	N	Z			82	1E					
3 22	0	1E													
	1	82													
	2	25		D	C	R		H							
	3	C2		J	N	Z			82	1E					
	4	1E													
822	5	82													
	6														
	7			(	C	O	N	T	I	N	U	E	D	)	
	8														

Figure 4-15a

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

ADDR		CODE														
8	0															
	1															
	2															
	3															
	4															
	5															
822	6	2	A		L	H	L	D		8	3	0	2			Address left character
	7	0	2													
	8	8	3													
	9	7	E		M	O	V		A	,	M					Test for period
	A	F	E		C	P	I		8	'0						
	B	8	0													
	C	2	3		I	N	X		H							Address next char
	D	D	A		J	C				8	2	0	6			Go to display unless period
	E	0	6													
	F	8	2													
823	0	2	A		L	H	L	D		8	3	0	0			Address start of message
	1	0	0													
	2	8	3													
	3	C	3		J	M	P			8	2	0	6			Go to display
	4	0	6													
	5	8	2													
	6															
	7															
	8															
	9															
	A															
	B															
	C															
	D															
	E															
	F															
3	0															
	1															
	2															
	3															
	4															
	5															
	6															
	7															
	8															

Figure 4-15b

This page intentionally left blank.

4.12 SENSOR CORRECTION EXERCISE, VERSION 1

This exercise introduces a more complete and realistic problem than any we have dealt with previously. It has four purposes:

1) to suggest the kind of task that a microcomputer may perform in a measurement or control application;

2) to bring in the idea of a data structure;

3) to demonstrate table lookup and calculating an address; and

4) to give you practice in using the instructions that you have learned.

4.12.1 Sensor Characteristics

A sensor is a device for measuring a physical variable such as temperature, pressure, sound intensity, light, etc. With our nerve cells we detect coldness and warmth; the familiar mercury thermometer converts that same physical variable into the length of a column of mercury; a semiconductor device called a thermistor converts that variable into a resistance that can be detected electrically.

The computer itself cannot measure resistance. External circuits must be attached to convert the variable resistance of a thermistor into a number that can be handled by the computer. This process is part of what is called "interfacing" -- connecting a computer to the external world. We shall not treat that subject here, but assume that our computer receives a number representing a measurement. We must process the number, perhaps to display or record a temperature or control a heater.

Suppose that we had an unmarked thermometer. To measure temperature in inches or millimeters of mercury would be meaningless, because the relationship depends on the size of the well of mercury at the bottom of the thermometer and the inside diameter of the glass tube. We could immerse the thermometer in a pot of melting ice, to give one repeatable temperature, and mark the point on the tube that the mercury reached. Then if we placed the thermometer in a pot of boiling water we could mark another point on the tube. Such a procedure is called "calibration". If we label the two points 0 and 100, and mark off equal spaces between them, we have calibrated our thermometer to the Celsius scale of temperature.

Similarly, if we have a sensor and an interfacing system connected to the computer, we can relate numbers we receive to known temperatures (or other physical qualities). Generally some arithmetic must be done to relate the electronically generated numbers to a familiar scale; this is similar to the procedure of converting a temperature measured in Fahrenheit to a Celsius temperature:

$$C = (F-32)\ (5/9)$$

Since Fahrenheit measurements relate to the same physical sensing device as Celsius measurements, this formula applies to any Fahrenheit thermometer. When we use a fundamentaly different sensing device such as a thermister, we have a more difficult problem. This is partly because the manufacturer of these devices is less consistent; each device may need a different offset and a different scaling factor.

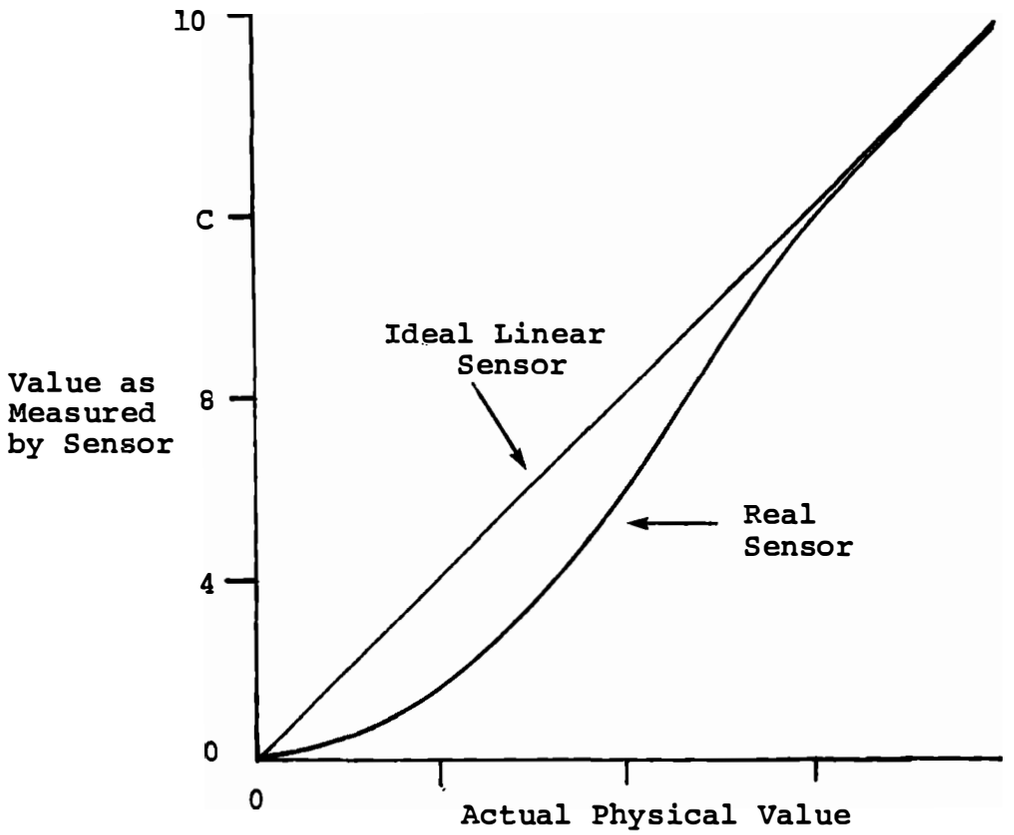
C = (measured value-offset) (scaling factor)

An additional problem arises with many electronic sensors: non-linearity. A formula such as that above may give correct answers over a limited range of measurement, but be increasingly in error outside of that range. Provided that the device gives consistent measurements, the measurements can still be converted to a standard scale, but simple arithmetic may not be sufficient. We may have to calibrate the device by making measurements at many known temperatures instead of just two. For each measurement we record the number received by the computer, and the known temperature. The resulting list is called a calibration table. Then in normal

operation, when we receive a new measurement we can look in the table to find the correct value. Such a procedure is part of the sensor correction exercise.

If all possible measurements are recorded in the table, it is easy to address the table and obtain the final result required. Sometimes, however, we can conserve memory space by including only a partial table. Suppose that we have a sensor which is linear over most of the range of measurements we are interested in, but at one end of the scale it has significant departure from linearity. Such behavior is suggested in the curve of Figure 4-16. If we had an ideal linear sensor, it would give a straight line in this plot, from 0 up to FF (if this is the possible range). Our real sensor is linear everywhere above about OC, but at the low end we have measured different values. These measurements are tabulated below.

SENSOR CALIBRATION TABLE	
Sensor Value	Corrected Value
0	0
1	3
2	4
3	5
4	6
5	7
6	8
7	9
8	9
9	A
A	B
B	B
> B	Linear



Sensor Calibration Curves

Figure 4-16

With this table we can correct the real sensor input to be equivalent to that of an ideal sensor. If the input is greater than OB, no adjustment is required; if less than that we must obtain an adjusted value from the table. There is no offset here --0 input means 0 true -- but we will have to multiply the actual or adjusted measurement by a scaling factor.

4.12.2 Organizing the Data Structure

We shall develop a program to adjust a non-linear sensor input value by table look-up, and multiply the result by a scaling factor. The adjusted values will be listed in a table, with one entry for each possible measurement up to the point where the sensor becomes linear.

Because the same program may be used for a different sensor, which may have a different linear point and a different scaling factor, these values will also be stored in the table. Such a combination of related but different kinds of values is called a "data structure".

The data structure will have this form:

8308	Scaling Factor
8309	Linear Point
830A	Adjusted value for input = 00
830B	Adjusted value for input = 01
830C	Adjusted value for input = 02
	(more adjusted values up to the linear point)

We shall see later how we can use an identical data structure, but with different information, to describe a second sensor which is also processed by the computer.

4.12.3 Organizing the Program

This program requires both input and output - obtain a value, correct it, and display it. We shall use a single programmed entry to the monitor (RST4) to accomplish the output from one calculation and the input to the next calculation. Each time the monitor is entered (after the first) Register A will contain a result. We shall display Register A to see this, enter a new input data byte to Register A, and press RUN to perform the next calculation.

At this point we must perform the following tasks:

1) Address the data structure and load the scaling factor into register E.
2) Increment the address and compare the input value with the linear point.

If the input value is equal to or greater than the linear point, skip the next three steps. Otherwise:

3) Increment the address to reach the adjusted value corresponding to a zero input.
4) Add the input into the address to reach the adjusted value corresponding to the actual input.
5) Replace the input value (A) with the adjusted value from memory.

Register A now contains either an input value which is in the linear range or an adjusted value.

6) Copy the (adjusted) input value to Register C from A.

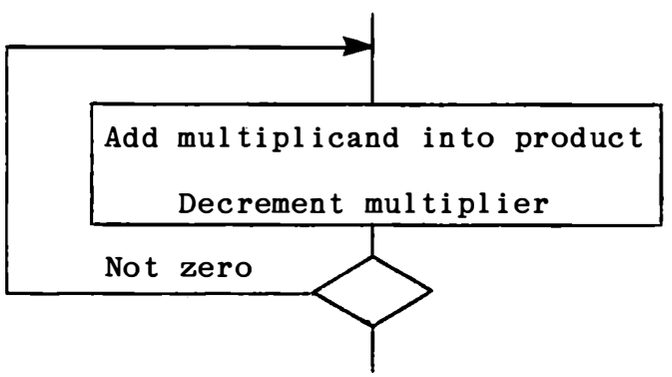
7) Clear register pair HL for the product (C) * (E).

8) Perform the multiplication. (see Section 4.4.3)

9) Jump back to enter the monitor again, with Register A containing the high byte of the result.

Note that using (HL) to address memory gives us two advantages here. We can move the scaling factor directly into Register E from memory, without disturbing Register A where we have the input, and we can compare (A) with memory to test whether the input is linear.

We have located the data structure starting at address 8308. Write the program yourself; then compare it with the solution given in Figure 4-17a. Copy the data structure from Figure 4-17b. Note that a scaling factor of 00 is given there. If you perform multiplication by repetitive addition, without special precautions, a zero multiplier does not result in a zero product.



If the multiplier is initially 01, the first decrement will set the Zero flag and end the process after the multiplicand has been added in once. If the multiplier is initially 00, the first decrement will make it FF, not zero, and the loop will be repeated 256 times. This technique does not admit the existance of multiplication by zero; instead it takes 00 in the multiplier (but not in the multiplicand) to mean 100 HEX.

We shall use this feature as a convenience here. An input of (say) 36 will be multiplied by 100 HEX, giving 3600 as a product. The high byte remains in (A) at the end of the multiplication, and is to be displayed. For initial testing of this program it will be easier if the adjusted result has not been scaled but merely shows the data from the table. A multiplier of 00 does this. Later we shall change to a different scaling factor.

SENSOR CORRECTION

CODING SHEET

ADDR		CODE													
8	20	0	00		NOP										
		1	00		NOP										
		2	00		NOP										
		3	AF		XRA		A								Clear Result
	820	4	E7		RST	4									Output and Input
		5	21		LXI		H	,	8	3	0	8			Address Data
		6	08												Structure
		7	83												
		8	5E		MOV		E	,	M						(E) ← Scaling Factor
		9	23		INX		H								Address linear point
		A	BE		CMP		M								Compare input
		B	D2		JNC				8	2	1	2			skip table lookup
		C	12												if input ≥ linear
		D	82												
		E	23		INX		H								Address table
		F	85		ADD		L								Add input to address
8	210	0	6F		MOV		L	,	A						Point to adjusted
		1	7E		MOV		A	,	M						value and load it
	821	2	4F		MOV		C	,	A						(C) ← (adjusted) input
		3	21		LXI		H	,	0	●	0	0			Clear product
		4	00												
		5	00												
	821	6	7D		MOV		A	,	L						Multiply by
		7	81		ADD		C								repetitive
		8	6F		MOV		L	,	A						addition
		9	7C		MOV		A	,	H						
		A	CE		ACI		00								
		B	00												
		C	67		MOV		H	,	A						
		D	1D		DCR		E								
		E	C2		JNZ				8	2	1	6			
		F	16												
8	22	0	82												
		1	C3		JMP				8	2	0	4			Back to monitor
		2	04												to display result
		3	82												
		4													
		5													
		6													
		7													
		8													

Figure 4-17a

ADDR		CODE			
8	0				
	1				
	2				
	3				
	4				
	5				
	6				
	7				
830	8	00	*	SCALING FACTOR	
	9	0C		LINEAR POINT	
	A	00		INPUT = 00	ADJUSTED VALUE
	B	03		INPUT = 01	
	C	04		INPUT = 02	
	D	05		INPUT = 03	
	E	06		INPUT = 04	
	F	07		INPUT = 05	
831	0	08		INPUT = 06	
	1	09		INPUT = 07	
	2	09		INPUT = 08	
	3	0A		INPUT = 09	
	4	0B		INPUT = 0A	
	5	0B		INPUT = 0B	
	6				
	7				
	8				
	9				
	A				
	B		*	SCALING FACTOR TO BE	
	C			CHANGED LATER	
	D				
	E				
	F				
8	0				
	1				
	2				
	3				
	4				
	5				
	6				
	7				
	8				

Figure 4-17b

4.12.4 Testing Sensor Correction

After writing your program and comparing it with Figure 4-17, you can test it by entering data and observing results. First, however, you should step through it to be sure there are no mistakes. If your program is different from Figure 4-17, follow the procedure below approximately, taking into account the differences.

RST				8200	00
RUN				8205	21

We have entered the monitor. Now it is time to enter data.

REG	A			8205	A-00
3				8205	A-03
STEP				8208	A-03
STEP				8209	A-03
STEP				820A	A-03

We should now have copied the scaling factor into (E) and addressed the linear point.

REG	E			820A	E-00
ADDR	8/H	MEM		8309	HL.0C
STEP			(CY)	820B	D2

The linear point (0C) is greater than the input value (03), so Carry was set by CMP M at 820A. The JNC will not be executed.

```
STEP                                    820E    23
STEP                                    820F    85
```

Now we shall add the input value to the table address.

```
REG     A                               820F    A-03
STEP                                    8210    A-0D
STEP                                    8211    A-0D
ADDR    8/H     MEM                     830D    HL.05
```

We have addressed the table for an input value of 03. The adjusted value is 05.

```
STEP                                    8212    4F
STEP                                    8213    21
STEP                                    8216    7D
```

All of the registers have been prepared for the multiplication.

```
REG     C       (multiplicand)          8216    C-05
REG     E       (multiplier)            8216    E-00
REG     H       (product)               8216    H-00
NEXT                                    8216    L-00
NEXT                                    8216    A-05
```

Step through the multiplication loop once or twice and then press RUN.

```
        RUN                                          8205    A-05
```

Multiplication by 100 HEX made the high byte of the product equal to the multiplicand.

```
        ADDR    8/H     MEM                          0500    HL.??
```

Test the program with each of the non-linear values (00 through 0B) and see that the results agree with the tabulated values. Switch to AUTO mode to speed up the lengthy multiplication.

Now change the scaling factor at 8308 to 88. Then try these input values and see if your results agree.

Input	Result
00	00
01	01
02	02
03	02
04	03
05	03
06	04
09	05
0D	06
10	08
20	11
30	19
40	22
80	44
C0	66

4.12.5 Review

In this exercise we have introduced the idea of a data structure -- a combination of related but different kinds of values. Often the arrangement of the data structure has an important effect on the efficiency of a program. If we had placed the scaling factor after the table of adjusted values, instead of before, we could still have found it but with several more program steps. In any program with variable data that can be structured, the data organization should be an early step in program development.

The table lookup in this program is a typical requirement in real measurement and control systems. Adding a physical quantity to an address seems peculiar on the surface -- like adding the number of passengers on a train to its speed, the numbers do not have the same dimensions. Adding a physical value to an address is only meaningful in the context of a data structure or table.

We have seen here the use of addressing memory with the register pair HL, thereby making a memory location available to be treated as a register. We used a comparison and the Carry flag to make a decision -- to adjust or not to adjust.

In the next exercise, which is a continuation of this one, we shall see further use of table lookup using HL memory addressing, and more decision making.

4.13 MULTIPLE TABLES WITH A DIRECTORY

In the sensor correction exercise of Section 4.12 we had a single sensor whose characteristics were described by the contents of a data structure. We shall now extend that program to handle multiple sensors. Both a sensor number and a physical measurement will be taken as inputs. The sensors, although similar in kind, will have different scaling factors, linear points and adjustment tables.

We shall add a second set of data with the same data structure as the existing one. The content of this second copy of the data structure will be:

```
8316    C8    Scaling factor
8317    08    Linear point
8318    00    Adjusted value, input = 00
8319    02                    input = 01
831A    04                    input = 02
831B    04                    input = 03
831C    05                    input = 04
831D    06                    input = 05
831E    07                    input = 06
831F    07                    input = 07
```

Now on the basis of the sensor number the program must select the appropriate table. Although we have specified addresses for the two tables in this example, the program must be written in a general way that permits more sensors, each having its own copy of the data structure with different data. In writing the program, then, we

shall assume that the number of sensors and the lengths of their tables are unknown; they are to be provided as initial information later on. For simplicity we shall allow not more than seven sensors, numbered from 1 to 7; and require that all of the sensor data will fit within 120 (decimal) bytes, from 8308 through 837F.

4.13.1 Directory to Data Structures

To find the address for the data relating to a particular sensor, we shall create an additional, different, data structure called a "directory". This is a different data structure in that the data contained in it do not have the same meanings as those for the individual sensors. The directory contains a list of the addresses of sensor data structures. It also contains, as its first entry, the highest sensor number for which data is stored in memory. The directory is to be located at 8300 - 8307.

8300	02	Highest existing sensor number
8301	08	Address for sensor number 1
8302	16	Address for sensor number 2
8303	00	Not used
8304	00	
8305	00	
8306	00	
8307	00	

Since we have only two sensors in this exercise, there are no tables for 3 through 7, and their positions in the directory are empty. Because we have specified that all of the data are in page 83xx, we have stored only the low byte of each data structure address.

4.13.2 Organizing the Program

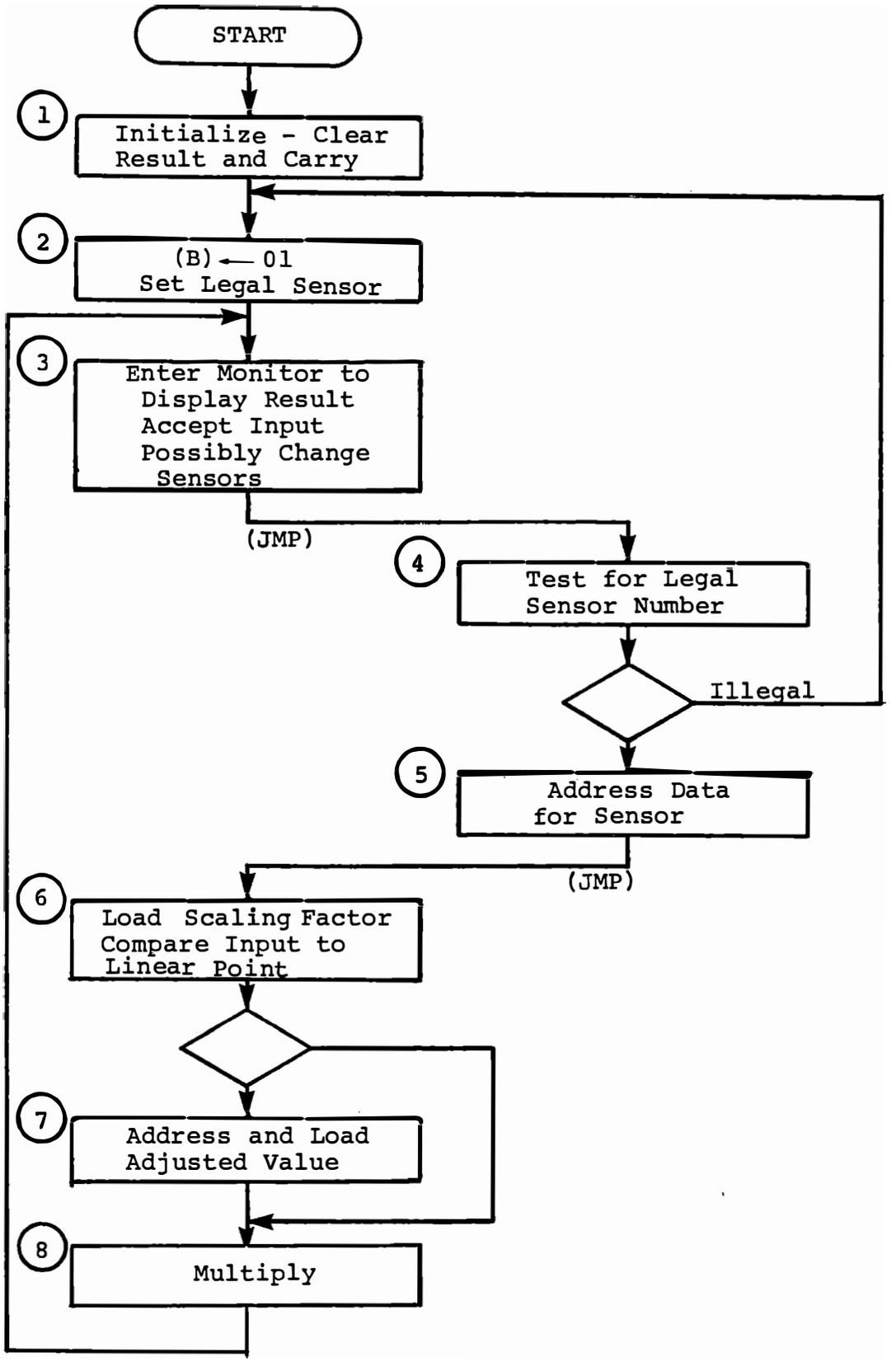
We shall accept sensor number as an input to the program at the same time that we accept the measured data. The sensor number probably will be one for which a data table is in the memory (in our example, 1 or 2). A wise programmer protects against errors, so we shall test for illegal sensor numbers -- 00 is forbidden, and any number greater than the first entry in the directory is forbidden.

We shall again use a monitor entry (RST4) to accept inputs. To avoid having to enter a sensor number every time, we shall keep the sensor number in Register B and allow but not require that it be changed. Thus you can test the program for one sensor at a time, without touching Register B.

Let us list the steps required in the program.

1) Clear the result (A) <- 00
2) Set a legal sensor number (B) <- 01
3) Enter the monitor to display the result and accept new data. Also accept sensor number if desired.
4) Test sensor number for a legal value -- not zero, and not greater than the highest sensor number in the directory. If illegal, take some special action, to be determined.
5) Use sensor number with directory to address the data structure for the sensor.
6) Load the scaling factor into register E.
7) Test data input:

   If less than linear point, address the adjustment table; find and load the adjusted value.
8) Multiply the (adjusted) value times the scaling factor.
9) Go to Step 3 and display the result.



Multiple Sensor Correction

Figure 4-18

Figure 4-18 shows the program as a flow diagram. The circled numbers correspond to the steps listed above. Reviewing these, steps 6 through 9 are identical to the program of Section 4.12. Steps 4 and 5 replace the LXI H, 8308 instruction which addressed the single data structure in the previous program. We can replace that LXI instruction with a JMP to some other location where we perform Steps 4 and 5; then jump back to Step 6 to finish the remaining program steps. This is shown in Figure 4-18. As indicated in the flow diagram, if an illegal sensor number is detected we shall go back to set a legal sensor number again.

4.13.3 Testing Sensor Number

At return from the monitor we have two bytes of data to be handled.

(A) = data input
(B) = sensor number

At this point we jump to another program segment to test the sensor number and find its data structue address.

We shall need Register A for making comparisons, so move the input data to another register. Then address the directory at 8300.

```
MOV     C,A
LXI H,  8300
```

Memory location 8300 contains the highest existing sensor number.

We are required to reject the input if the sensor number is greater than the highest existing number. Recall the way flags are set by a comparison (section 4.11.1).

CMP r	Zero	Carry
(A) greater than (r)	Cleared	Cleared
(A) equal to (r)	Set	Cleared
(A) less than (r)	Cleared	Set

To make the decision with a single conditional jump we must make the comparison by:

```
MOV   A,M        Highest existing sensor
CMP   B          Compare sensor number
```

This sets Carry if the sensor number is too great. Then a single JC will handle this error condition. If we used

```
MOV   A,B
CMP   M
```

then either Carry or Zero would indicate a legal sensor number, and two conditonal jumps would be needed.

We must also test for the other illegal condition, sensor number zero. This can be done by

```
MOV   A,B
ORA   A
```

which sets Zero if the sensor number is zero.

It would be convenient if the error condition were somehow indicated, and if the illegal sensor number were kept available for inspection at reentry to the monitor. When the sensor number is legal we go to the monitor with (A) = high byte of the multiplication result, and carry clear from the multiplication. Let us define the error result as follows:

Carry set
(A) = illegal sensor number
(B) = sensor number 1

The following procedure will do the testing and give the above result.

```
MOV     C,A             (C) <- Input Value
LXI     H, 8300         Address Directory
MOV     A,M             Highest Sensor Number
CMP     B               Test sensor number
MOV     A,B             (A) <- sensor number
JC      8202            To set (B) = 01 and
                        display (A) with Carry
ORA     A               Test for sensor = 0
STC                     Mark error
JZ      8202            To set (B) = 01 and
                        display (A) with Carry
```

If both tests are satisfied (the sensor number is legal) we must find the address of its data structure.

4.13.4 Using the Directory

Assuming that we have a legal sensor number, we shall now use it to look in the directory and address the data structure for this sensor.

In the table lookup of Section 4.12 we added the input value to a table address to find another address where desired data was stored. Here we do the same thing. Recall that the directory contains:

8300	02	Highest existing sensor number
8301	08	Data structure address for Sensor Number 1
8302	16	Data structure address for Sensor Number 2

Register pair HL contains 8300, and the sensor number is already in Register A.

Add the sensor number into the address:

```
ADD  L
MOV  L,A
```

and now (HL) contains either 8301 or 8302. (Since Register L contained 00 we could skip the ADD L, but that would only work with a directory starting at a page boundary such as 8300).

Now (HL) points to a memory location containing the address of another memory location. Since all of the data are in a single page we can finish the indirect addressing with only one more instruction:

```
MOV  L,M          Address data table
```

Note that we can load L with a data byte from a memory location addressed by HL. By the time Register L is affected we no longer need the old address in HL. If the directory entries were two byte addresses we would use a more conventional indirect addressing means.

We have now loaded HL with the address of the data structure for the given sensor number.

For Sensor Number 1, (HL) = 8308. We have replaced the instruction LXI H, 8308 that existed in the earlier program. One more step is required before going back to the original program: copy the input value back into Register A where it was placed originally.

```
MOV    A,C
JMP    8208
```

Now for Sensor Number 1 the program should behave exactly as it did with the program of Section 4.12. When you change the sensor number you will receive different results.

When you have loaded your program and the directory and second set of data, we shall step through the program. The addresses shown below refer to the given solution (Figure 4-19). Follow your own program through the same process.

ADDR		CODE															
8 20	0	3	E		M	V	I		A	,	0	0			Clear Result		
	1	0	0														
820	2	0	6		M	V	I		B	,	0	1			Set Legal Sensor Number		
	3	0	1														
820	4	E	7		R	S	T	4									
	5	C	3		J	M	P			8	2	3	0		Get address for Data Structure		
	6	3	0														
	7	8	2														
820	8	5	E		M	O	V		E	,	M						
	9	2	3		I	N	X		H								
	A	B	E		C	M	P		M								
	B	D	2		J	N	C			8	2	1	2		Identical to Figure 4-16a		
	C	1	2														
	D	8	2														
	E	2	3		I	N	X		H								
	F	8	5		A	D	D		L								
8 21	0	6	F		M	O	V		L	,	A						
	1	7	E		M	O	V		A	,	M						
821	2	4	F		M	O	V		C	,	A						
	3	2	1		L	X	I		H	,	0	0	0	0			
	4	0	0														
	5	0	0														
821	6	7	D		M	O	V		A	,	L						
	7	8	1		A	D	D		C								
	8	6	F		M	O	V		L	,	A						
	9	7	C		M	O	V		A	,	H						
	A	C	E		A	C	I		0	0							
	B	0	0														
	C	6	7		M	O	V		H	,	A						
	D	1	D		D	C	R		E								
	E	C	2		J	N	Z			8	2	1	6				
	F	1	6														
3 22	0	8	2														
	1	C	3		J	M	P			8	2	0	4				
	2	0	4														
	3	8	2														
	4																
	5																
	6																
	7																
	8														Figure 4-19a		

ADDR		CODE													
8 23	0	4	F		M	O	V		C	,	A				(C) ← Input Data
	1	2	1		L	X	I		H	,	8	3	0	●	Address Directory
	2	0	0												(highest existing
	3	8	3												sensor number)
	4	7	E		M	O	V		A	,	M				
	5	B	8		C	M	P		B	,					Set CY if illegal
	6	7	8		M	O	V		A	,	B				(A) ← Sensor Number
	7	D	A		J	C				8	2	0	2		If illegal go to
	8	0	2												Display Sensor
	9	8	2												number with Carry
	A	B	7		O	R	A		A						Test for zero
	B	3	7		S	T	C								
	C	C	A		J	Z				8	2	0	2		If sensor 0 go to
	D	0	2												display sensor
	E	8	2												number with Carry
	F	8	5		A	D	D		L						Add sensor number
8 24	0	6	F		M	O	V		L	,	A				into directory address
	1	6	E		M	O	V		L	,	M				Address data structure
	2	7	9		M	●	V		A	,	C				(A) ← Input Data
	3	C	3		J	M	P			8	2	0	8		Go to table
	4	0	8												lookup and
	5	8	2												multiply
	6														
	7														
	8														
	9														
	A														
	B														
	C														
	D														
	E														
	F														
3	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure 4-19b

ADDR		CODE			
8 30	0	02		HIGHEST	SENSOR NUMBER
	1	08		DATA	ADDRESS - SENSOR 1
	2	16		DATA	ADDRESS - SENSOR 2
	3	00		RESERVED	FOR
	4	00			ADDITIONAL SENSORS
	5	00			
	6	0			
	7	00			
830	8	88		SCALING	FACTOR - SENSOR 1
	9	0C		LINEAR	POINT
	A	00		ADJUSTED	VALUES
	B	03		FOR	01
	C	04			02
	D	05			03
	E	06			04
	F	07			05
8 31	0	08			06
	1	09			07
	2	09			08
	3	0A			09
	4	0B			0A
	5	0B			0B
831	6	C8		SCALING	FACTOR - SENSOR 2
	7	08		LINEAR	POINT
	8	00		ADJUSTED	VALUES
	9	02		FOR	01
	A	04			02
	B	04			03
	C	05			04
	D	06			05
	E	07			06
	F	07			07
3	0				
	1				
	2				
	3				
	4				
	5				
	6				
	7				
	8				

Figure 4-19c

4.13.5 Testing MULTIPLE SENSOR CORRECTION

First, let us try the program with an illegal sensor to check on the test:

RST				8200	3E
RUN				8205	C3
REG	B			8205	B-01
3				8205	B-03
REG	A	8		8205	A-08
RUN			(CY)	8205	A-03

The illegal sensor number is displayed with Carry set. B has been loaded with 01 again. Let 03 stay as an input value.

STEP		(CY)	8230	A-03
STEP		(CY)	8231	A-03
STEP		(CY)	8234	A-03
STEP		(CY)	8235	A-02

We have loaded the highest existing sensor number. Now the comparison (CMP B):

STEP		8236	A-02

Since Register B contains a legal number (01) Carry is reset. We move the sensor number into A, do <u>not</u> execute JNC, and test for zero.

STEP		8237	A-01
STEP		823A	A-01
STEP		823B	A-01

Neither Carry nor Zero is set by the test for zero (ORA A at 823A), but now the program sets Carry before the conditional jump, which will not be executed.

```
        STEP                                (CY)    823C    A-01
        STEP                                (CY)    823F    A-01
```

ADD L clears the Carry but, since (L) = 00 it changes nothing else.

```
        STEP                                        8240    A-01
        STEP                                        8241    A-01
```

We have now addressed the directory entry for Sensor Number 1.

```
        ADDR    8/H     MEM                         8301    HL.08
        ADDR                                        8241      6E
```

This is the MOV L,M instruction.

```
        STEP                                        8242      79
        ADDR    8/H     MEM                         8308    HL.88
```

We have addressed the scaling factor for Sensor Number 1.

```
        STEP                                        8243      C3
        REG     A                                   8243    A-03
        STEP                                        8208    A-03
```

We are ready for table lookup and multiply.

```
        RUN                                 (Z)     8205    A-02
```

Multiplication has set Zero (by DCR E) but left Carry clear.

Check the two byte result of the multiplication:

```
ADDR    8/H     MEM                         02A8    HL.??
```

The data in HL represent the product. Because this happens also to represent a memory address within the monitor, a data byte is shown, but it is meaningless here.

Now try the other sensor.

```
REG     B       2                           8205    B-02
REG     A       8                           8205    A-08
```

Set a breakpoint at the instruction after the JMP 8230.

```
ADDR    8       2       0       8   BRK     8208    BP.
RUN                                         8208    A-08
```

The input data has been restored. Check the sensor number and data structure address.

```
REG     B                                   8208    B-02
ADDR    8/H     MEM                         8316    HL.C8
```

We have addressed the data structure for sensor number 2.

```
RUN                                         8205      C3
REG     A                                   8205    A-06
```

The entry value (08) was not adjusted, but it was multiplied by C8. The two byte product is:

```
ADDR    8/H     MEM                         0640    HL.??
```

It all of this has worked, set AUTO mode to speed up the operation.

Try the following input data and check that your results agree. The inputs have been chosen to include some that give identical results.

Sensor (B)	Input (A)	Result (A)	Two Byte Product (HL)
01	00	00	0000
01	01	01	0198
01	04	03	0330
01	07	04	04C8
01	08	04	04C8
01	09	05	0550
01	0A	05	05D8
01	0B	05	05D8
01	0C	06	0660
01	80	44	4400
02	03	03	0320
02	06	05	0578
02	07	05	0578
02	08	06	0640
02	09	07	0708
02	0C	09	0960
02	80	64	6400

4.14 SUMMARY

In this chapter we have met many of the 8080 instructions. Registers have been used for temporary data storage, providing operands for ADD, SUB, CMP, etc., and for counting. Exercises have been used to introduce arithmetic, including double precision addition, subtraction and multiplication.

We have used register pairs to address memory, using LDAX and STAX, and using ((HL)) as a register. The concept and practice of indirect addressing was introduced, and we have used several methods of obtaining memory addresses from other memory locations.

The technique of operating the MTS display by storing data in certain memory locations was also used. Overall, then, this chapter has dealt extensively with memory. The next chapter teaches about memory hardware and how some of these addressing techniques work in a physical sense.

4.15 INSTRUCTION CHART

The instruction chart on the following page shows all of the 8080 instructions. Most of the data transfer, counting and arithmetic instructions have now been introduced, as well as a few of the branch instructions. Study the organization of this chart so that you can readily find an instruction when you need it. A hard copy of this chart is supplied for convenient reference.

HEX CODES FOR 8080 INSTRUCTIONS

DATA TRANSFER	SOURCE REGISTER A	B	C	D	E	H	L	M	SP	IMMEDIATE (DATA FROM PROGRAM)
MOV A,s	7F	78	79	7A	7B	7C	7D	7E		MVI A 3E
MOV B,s	47	40	41	42	43	44	45	46		MVI B 06
MOV C,s	4F	48	49	4A	4B	4C	4D	4E		MVI C 0E
MOV D,s	57	50	51	52	53	54	55	56		MVI D 16
MOV E,s	5F	58	59	5A	5B	5C	5D	5E		MVI E 1E
MOV H,s	67	60	61	62	63	64	65	66		MVI H 26
MOV L,s	6F	68	69	6A	6B	6C	6D	6E		MVI L 2E
MOV M,s	77	70	71	72	73	74	75	–		MVI M 36
LXI rp		01		11		21			31	2 DATA BYTES FROM PROGRAM
LDA addr	3A									ADDRESS FROM
STA addr	32									PROGRAM (2 BYTES)
LDAX rp		0A		1A						ADDRESS FROM
STAX rp		02		12						REGISTER PAIR
LHLD addr						2A				ADDRESS FROM
SHLD addr						22				PROGRAM (2 BYTES)
SPHL						F9				SP←HL
PCHL						E9				PC←HL (BRANCH)
XCHG						EB				DE↔HL
XTHL						E3				STACK TOP↔HL
PUSH rp		C5		D5		E5		PUSH PSW	F5	SP←SP – 2
POP rp		C1		D1		E1		POP PSW	F1	SP←SP + 2

COUNTING	A	B	C	D	E	H	L	M	SP	FLAGS AFFECTED
INR d	3C	04	0C	14	1C	24	2C	34		Z, S, P, AC
DCR d	3D	05	0D	15	1D	25	2D	35		Z, S, P, AC
INX rp		03		13		23			33	NONE
DCX rp		0B		1B		2B			3B	NONE

ARITH/LOGIC	A	B	C	D	E	H	L	M	SP	IMMEDIATE (DATA FROM PROGRAM)
DAD rp		09		19		29			39	
ADD s	87	80	81	82	83	84	85	86		ADI C6
ADC s	8F	88	89	8A	8B	8C	8D	8E		ACI CE
SUB s	97	90	91	92	93	94	95	96		SUI D6
SBB s	9F	98	99	9A	9B	9C	9D	9E		SBI DE
ANA s	A7	A0	A1	A2	A3	A4	A5	A6		ANI E6
XRA s	AF	A8	A9	AA	AB	AC	AD	AE		XRI EE
ORA s	B7	B0	B1	B2	B3	B4	B5	B6		ORI F6
CMP s	BF	B8	B9	BA	BB	BC	BD	BE		CPI FE

	INSTRUCTION								FLAGS
ACCUMULATOR AND CARRY	RLC 07	RRC 0F	RAL 17	RAR 1F	DAA 27	CMA 2F	STC 37	CMC 3F	ONLY THE CY FLAG IS AFFECTED EXCEPT: CMA NO FLAGS DAA ALL FLAGS
BRANCH	JMP	CALL	RET	PCHL	HLT	NOP			BRANCH AND IN/OUT INSTRUCTIONS DO NOT AFFECT ANY FLAGS
UNCOND	C3	CD	C9	E9	76	00			DATA TRANSFER INSTRUCTIONS DO NOT AFFECT ANY FLAGS EXCEPT: POP PSW AFFECTS ALL FLAGS
COND NZ	C2	C4	C0						ARITHMETIC/LOGIC INSTRUCTIONS AFFECT ALL FLAGS EXCEPT: DAD AFFECTS CY ONLY
Z	CA	CC	C8						INR AND DCR AFFECT ALL FLAGS EXCEPT: CY
NC	D2	D4	D0						INX AND DCX DO NOT AFFECT ANY FLAGS
C	DA	DC	D8						
PO	E2	E4	E0						
PE	EA	EC	E8						
PLUS	F2	F4	F0						
MINUS	FA	FC	F8						
INPUT/OUTPUT & INTERRUPT	IN DB	OUT D3	EI FB	DI F3					IN AND OUT ARE TWO BYTE INSTRUCTIONS WITH PORT ADDRESS
RESTART (CALL TO) HEX CODE	RST 0 0000 C7	RST 1 0008 CF	RST 2 0010 D7	RST 3 0018 DF	RST 4 0020 E7	RST 5 0028 EF	RST 6 0030 F7	RST 7 0038 FF	

This page intentionally left blank.

MICROCOMPUTER TRAINING WORKBOOK

CHAPTER 5

MEMORY HARDWARE

INTRODUCTION TO CHAPTER 5

Having explored (in Chapters 2 and 4) the ways that programs address the memory, we will now examine the physical addressing of the memory. This chapter discusses the following subjects:

Control Interface

Memory Technology - ROM and RAM

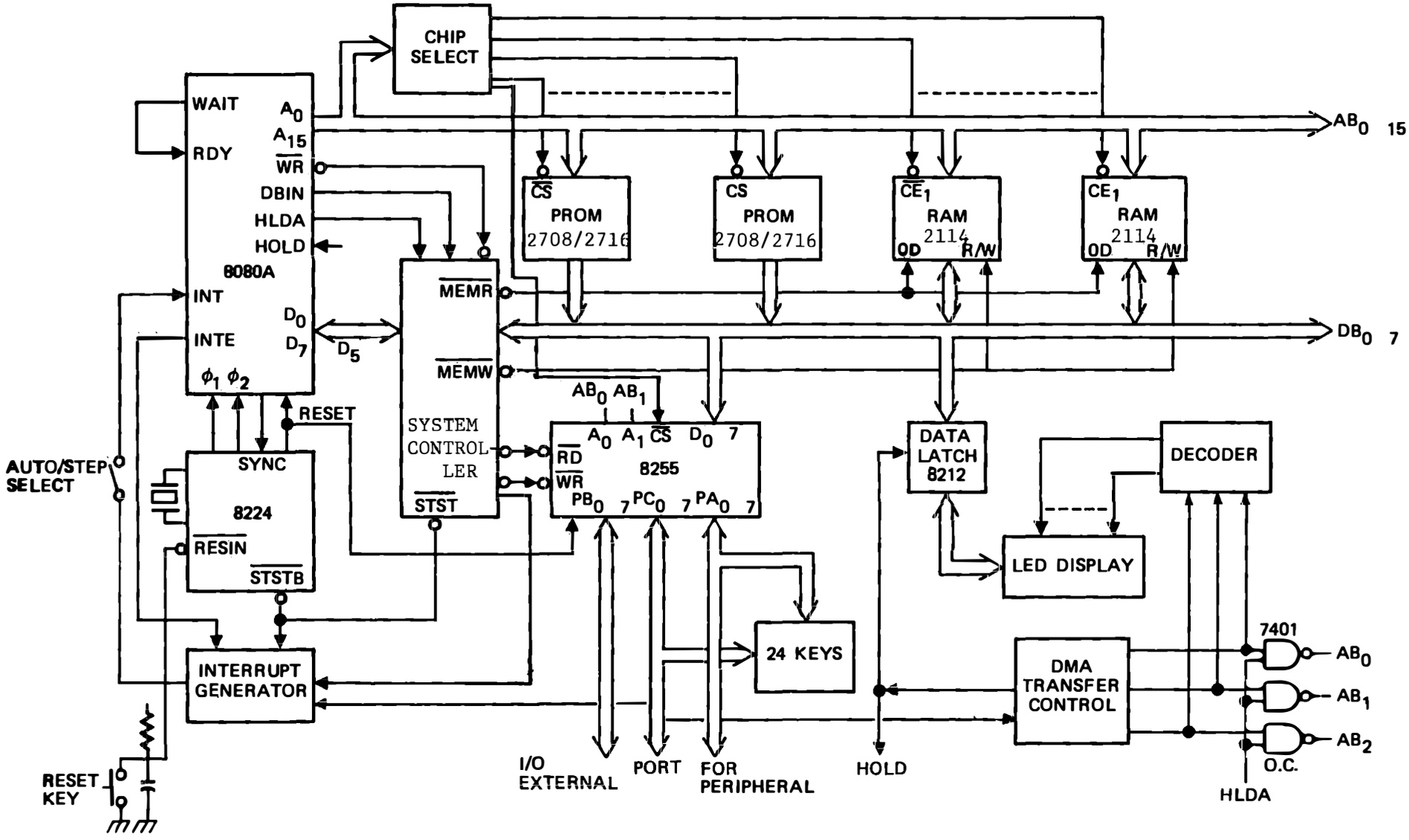
Memory Addressing and Address Decoding

Data Bus Connections and Tri-State Circuits

Direct Memory Access and Interrupt Inputs

Memory Signals and Timing

The principal purpose of this chapter is to discuss the connection of memory devices to the microprocessor. This requires a cursory understanding of the control signals between the CPU and the memory. For the sake of completeness Section 5.1 discusses these control signals in some detail, but it is suggested that the student skim much of this section now, and refer back to it when other control signals are brought up in later chapters.



Microcomputer Training System Configuration
Figure 5-1

5.1 SYSTEM CONTROLLER

A computer must include a CPU (central processing unit), memory, and input/output devices (Figure 5-1). The 8080 microprocessor demands additional hardware (the System Controller) to allow the necessary connections to memory and I/O, because of pin limitations. To overcome this limitation, some pins are bi-directional -- at some times they are inputs to the CPU, and at other times they are outputs.

The CPU controls the usage of the address and data buses, giving control signals to memory, I/O, and other external devices to indicate the functions to be performed. To further extend the functions of the limited number of pins, certain of the control signals are output on the data bus, and must be accepted and stored by the System Controller so that the data bus can be used to transfer other data. The control signals output via the data bus are referred to collectively as the "status byte".

5.1.1 Control Signals

In Chapters 1, 2 and 4 we described in some detail the series of steps required to execute each of several instructions. Such a series is an "instruction cycle". In general each of the steps is a "machine cycle", and in each step the address and data buses may be used differently. The control signals are largely concerned with defining the functions of the buses, controlling the operations of different external devices.

Some of the control signals contain timing information, and vary within a machine cycle. These signals have assigned pins on the 8080 chip. Other signals remain effective throughout one machine cycle. These are output on the data bus at the beginning of a machine cycle as the status byte and are latched by the System Controller.

The timing signals are:

SYNC	Designates status byte time.
DBIN	CPU will accept data from bus.
$\overline{WR}$	CPU places data on bus.
WAIT	Acknowledge "Not Ready".
HLDA	Acknowledge "Hold".

The two signals DBIN and $\overline{WR}$ are actually sufficient for a system that does not use interrupts, and which uses "memory mapped" input/output. (These I/O schemes are described in Chapter 8.)

When DBIN is true (high) the memory or input device addressed should deliver data onto the data bus to be read by the CPU. When $\overline{WR}$ is true (low) the memory or output device addressed should accept data placed on the data bus by the CPU. These signals do not distinguish memory from I/O devices.

If the memory or I/O device is too slow to deliver or accept data within the time available, it can give the 8080 a Not Ready input which will extend the time of DBIN or $\overline{WR}$ for one or more clock cycles. WAIT acknowledges this request.

If some other device needs to use the address and data buses, it may ask the CPU to suspend its operations and release the buses. HLDA is a signal that grants such a request.

SYNC actually extends both before and after the time that the status byte is present on the data bus. It must be gated with the phase 1 clock (a narrow pulse) to latch the status byte into the System Controller. This function is performed by the 8224 clock generator, which receives SYNC and outputs $\overline{\text{STSTB}}$, the narrow pulse.

5.1.2 Status Byte

The status byte output on the data bus at SYNC time is defined below. The major function of the System Controller is to latch (hold) the status byte and also decode it to give signals that are more convenient for use by the memory and I/O devices. The data bus line that carries each signal is designated in parentheses.

Some of the functions mentioned below have not been defined, and will not be discussed until later chapters. The student is urged to ignore them for now, and refer back to this chapter when appropriate. A detailed understanding of these controls is necessary for the hardware designer, but not for the programmer.

a. MEMR (D7) This machine cycle is to read from memory. The signal is true during instruction fetch,memory read, stack read, and halt machine cycles.

b. $\overline{WO}$ (D1) This machine cycle is to output from the CPU to memory or I/O. The signal is true (low) during memory write, stack write, and output machine cycles.

c. INP (D6) An IN instruction is being executed. The addressed input device should place data on the bus during DBIN.

d. OUT (D4) An OUT instruction is being executed. The addressed output device should accept data from the bus during $\overline{WR}$.

e. M1 (D5) An instruction fetch cycle is being executed. This is true only and always for the first machine cycle of every instruction cycle.

f. STACK (D2) The current address is from the stack pointer.

g. HLTA (D3) Indicates that the CPU is in a Halt state.

h. INTA (D0) Acknowledges an interrupt.

5.1.3 Decoded Control Signals

The System Controller gates the various status byte and timing signals to generate control signals that are convenient for memory and I/O devices. In subsequent discussion of memory and I/O hardware, we will refer to the following signals:

a. $\overline{\text{MEMW}}$ An active low signal indicating that the data bus content should be stored at the addressed memory location. It is true (low) during $\overline{\text{WR}}$ time if $\overline{\text{WO}}$ is true and OUT is false.

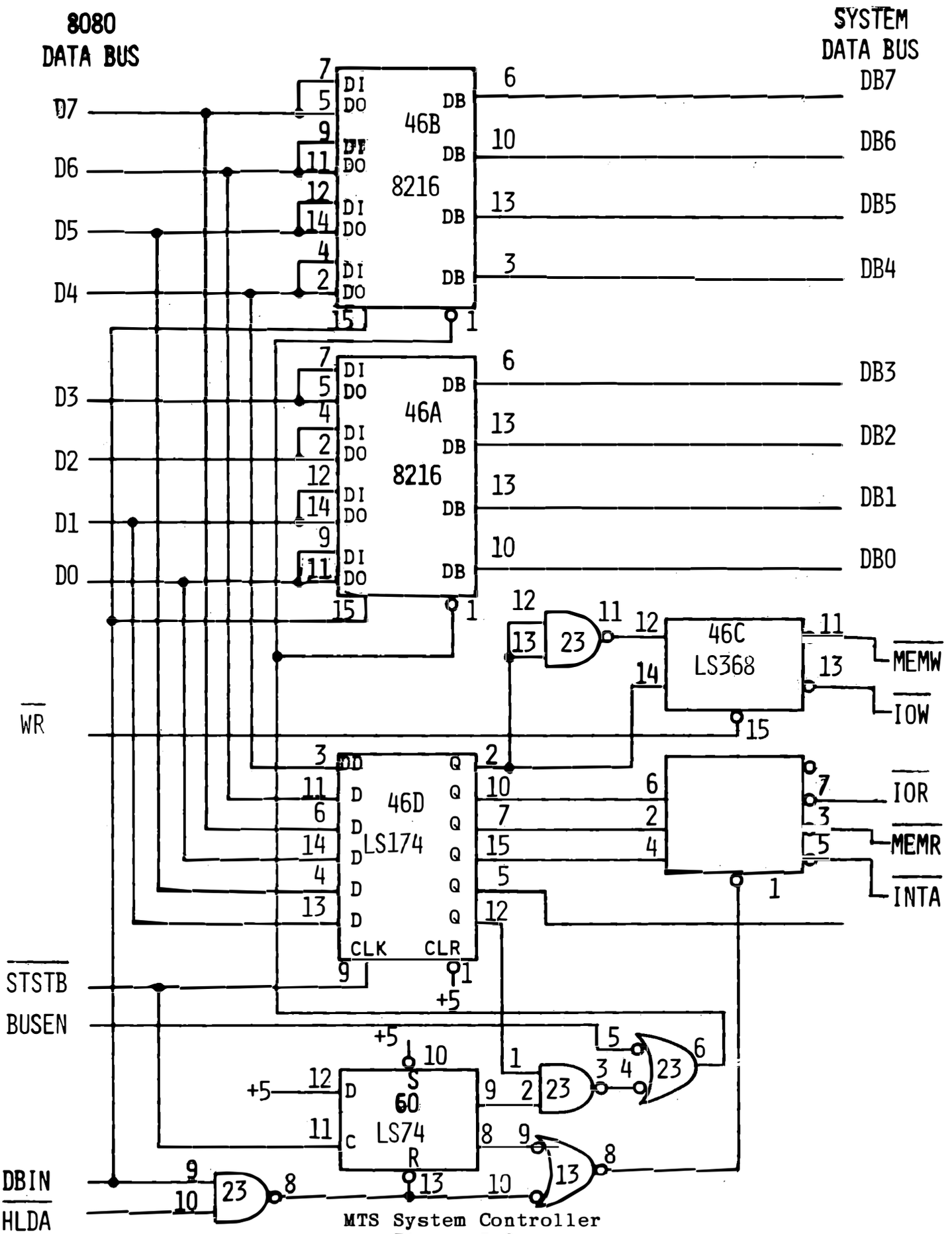
b. $\overline{\text{MEMR}}$ An active low signal indicating that data from the addressed memory location should be placed on the data bus. It is true (low) during DBIN time of an instruction fetch, memory read or stack read machine cycle.

c. $\overline{\text{IOW}}$ An active low signal indicating that the addressed output input device should accept data from the bus. It is true (low) during $\overline{\text{WR}}$ time if $\overline{\text{WO}}$ is true and OUT is true.

d. $\overline{\text{IOR}}$ An active low signal indicating that the addressed input device should place data on the bus. It is true during DBIN time of an input read machine cycle.

e. $\overline{\text{INTA}}$ An active low signal indicating that an interrupt has been acknowledged, and the interrupt instruction should be placed on the data bus. It is true during DBIN time if INTA of the status byte is true.

f. M1 An active high signal indicating that the current machine cycle is the first (or only) machine cycle of an instruction cycle. It is the latched value of M1 in the status byte.



MTS System Controller
Figure 5-2

5.1.4 MTS System Controller Logic

Figure 5-2 shows the detailed logic of the MTS system controller. The two 8216 bidirectional bus drivers provide electrical isolation of the 8080 data bus from the system bus. The 74LS174 six bit latch stores the required bits of the status bytes. (STACK and HLTA are not used.) The 74LS368 tri state buffer (upper section) generates either $\overline{\text{MEMW}}$ or $\overline{\text{IOW}}$ during $\overline{\text{WR}}$ time, depending on whether OUT is false or true. The lower section of the 368 generates $\overline{\text{IOR}}$, $\overline{\text{MEMR}}$ or $\overline{\text{INTA}}$ during DBIN time, depending on whether IN, MEMR, or INTA of the status byte was true. These signals are further qualified by the flip flop and gates at the bottom of the diagram, which have the effect of inhibiting the signals when a HOLD request has been given by the DMA channel and acknowledged by the 8080 on $\overline{\text{HLDA}}$.

5.1.5 Intel 8228 System Controller

All of the functions of the system controller can be provided by the Intel 8228. This is a 28 pin chip, is fairly inexpensive, and is used in most 8080 microcomputer systems. In fact, Intel refers to the 8080 microprocessor, 8224 clock generator and 8228 system controller as the "CPU Group".

In addition to latching and decoding the control signals, the 8228 isolates the system data bus from the 8080 data bus, providing additional power drive capability to support large memories and allowing certain data bus uses to overlap in time.

Although the 8228 is applicable in most microcomputer designs and is typically more economical than the several logic chips required to

replace it, the 8228 is unfortunately not compatible with the S-100 data bus. Therefore, the MTS was designed without the 8228 because its use would have precluded system expansion to the S-100 bus.

This incompatibility arose because the S-100 Bus was defined prior to the development of the 8228 by Intel. For good engineering reasons, the 8228 does not handle the status byte exactly as required for S-100 compatibility. In particular, the 8228 isolates the system data bus from the 8080 data bus during SYNC time, and does not place the status byte on the external bus. This has the advantage that an addressed memory or input device can place data on the bus prior to the DBIN signal, which slightly increases the effective memory speed. On the other hand, the S-100 Bus definition requires that the status byte be available on the data bus. Therefore, the 8228 cannot be used with an S-100 interface.

The 8228 has two additional functions that are useful in some interrupt systems, as will be described in Chapter 8. $\overline{\text{INTA}}$ is principally an output signal from the 8228, acknowledging an interrupt and indicating that an external device should enter an instruction to the 8080. If this pin is pulled up through a 1K resistor to +12 volts, the 8228 will supply the instruction code FF, which is RST7. (See Chapter 8.) In the MTS controller this function is accomplished by resistor pullups on the data bus.

The 8228 also recognizes a CALL instruction being placed on the data bus in response to $\overline{\text{INTA}}$, and controls the buses to accept from the external device the second and third bytes of the CALL.

5.2 MEMORY TECHNOLOGY

A memory device includes semiconductor circuits or elements to serve four functions:

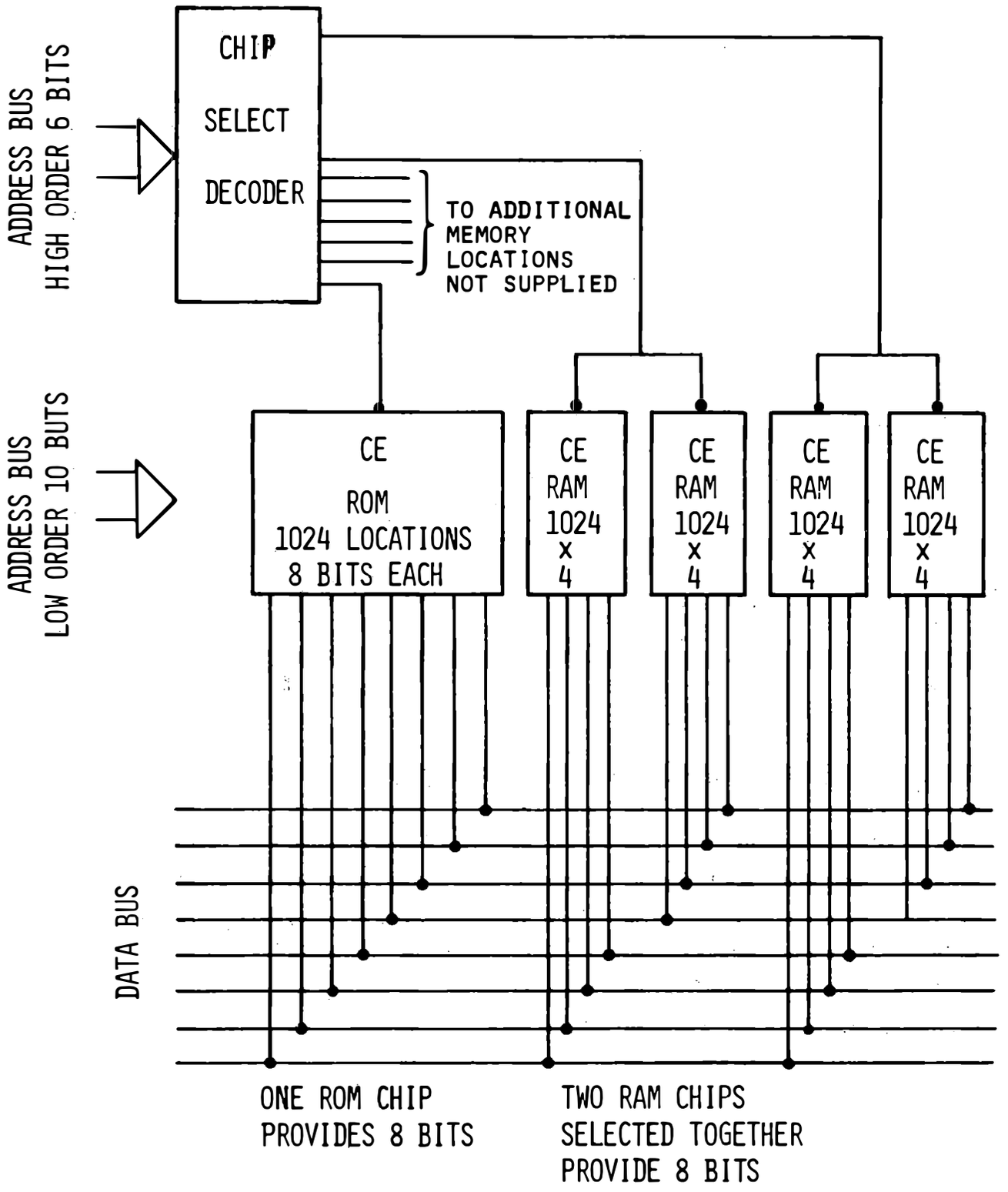
a) Store data in an ordered array

b) Decode the address inputs to select a certain location

c) Alter the stored data at the selected location upon command

d) Output the data from the selected location upon command

The memory devices used in the MTS each have 1024 locations, addressed by the low-order ten bits of the system address bus. The ROM and RAM memories of your MTS system are shown in Figure 5-3. The ROM devices store eight bits at each location. The RAM devices store four bits at each location, so two devices are used for the eight bits that must be stored for each address.

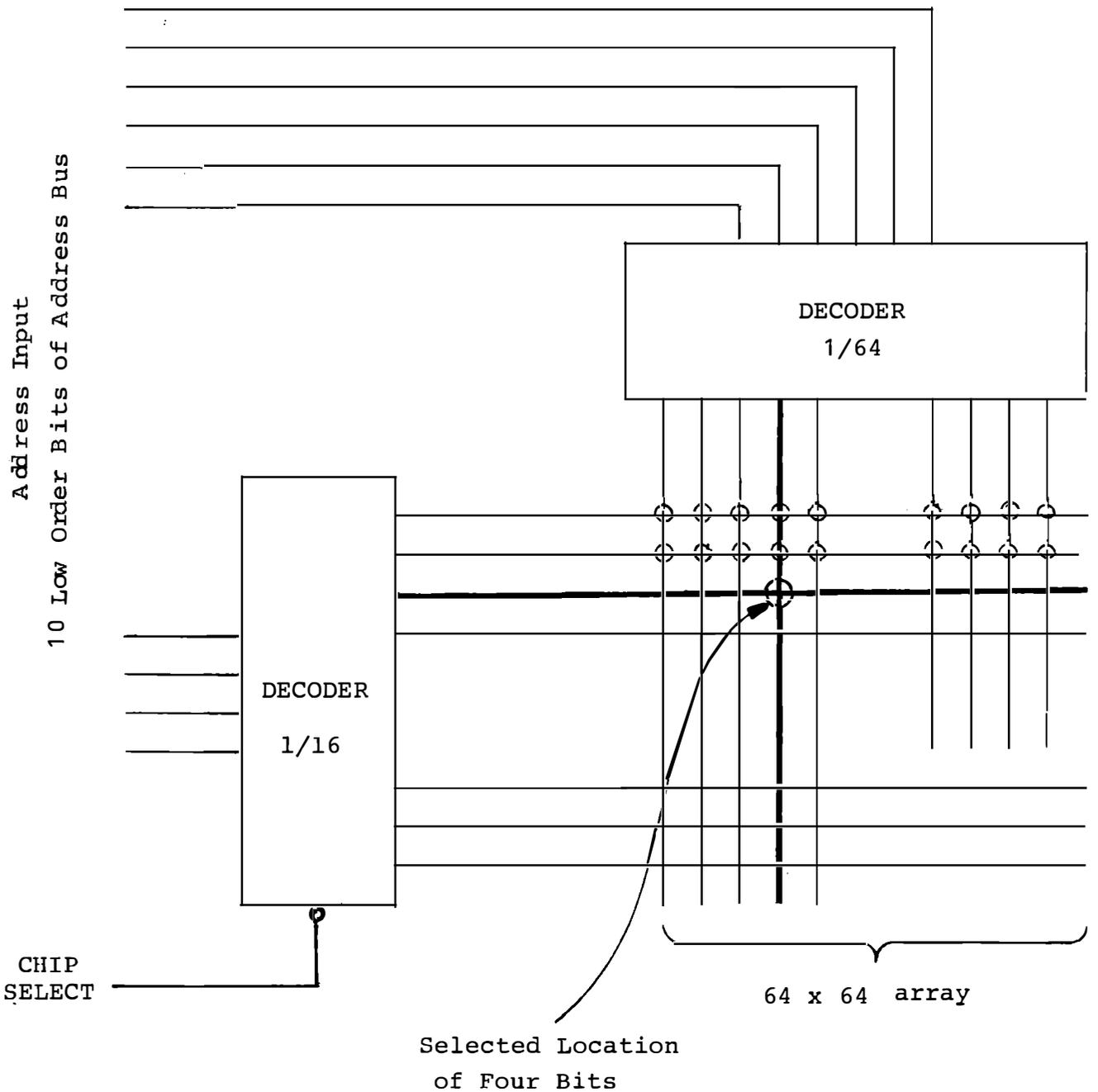


Memory Addressing

Figure 5-3

The electronic means of storing data depends on the kind of memory device used. Permanent (mask) Read Only Memory (ROM) has, for each bit, a transistor that is either created or destroyed during the semiconductor manufacturing process. In eraseable and Programmable Read Only Memory (PROM) devices, such as the MTS's 2708, a physical quality of the semiconductor material at each bit position is altered by a relatively high voltage pulse during programming. The change is reversible but non-volatile: it will remain indefinitely until a new programming operation is performed. The MTS has no facility for applying such high energy pulses, so data cannot be written to the PROM while it is in the circuit. The PROM can be rewritten by removing it from the circuit board, erasing it by exposure to intense ultraviolet light, and writing a new program with a special programming device.

In read-write memory the data are stored in the form of current or charge in transistors. Static RAMs, such as the MTS's 2114, include a flip flop circuit for each bit. Such a circuit has two stable states; one transistor conducts while a second is cut off. Dynamic RAMs store data in the form of a charge, which gradually leaks away and must be refreshed at approximately one millisecond intervals. Refreshing requires additional external circuits, which is not appropriate in small systems. However, many more bits can be stored in one dynamic device, which is desirable in large systems.



Internal Address Decoding in a Memory Device

Figure 5-4

The MTS read-write memory devices have an array of 4096 storage locations, arranged as a square 64 cells high and 64 cells wide. The ten address lines received by the device are divided into two groups, of six and four bits. The six lines are decoded to select one of 64 columns, as shown in Figure 5-4. The other four lines are decoded by a one-of-16 decoder to select four of the 64 rows, provided that the chip select input to the memory device is active. Thus a unique ten bit address, plus chip select, addresses a single set of four bits out of the 4096 bits stored in the memory device. These four bits are connected to control logic in the memory device to be read or written as required.

The PROM addressing is similar, except that these devices store 8192 bits, arranged as 1024 sets of eight bits.

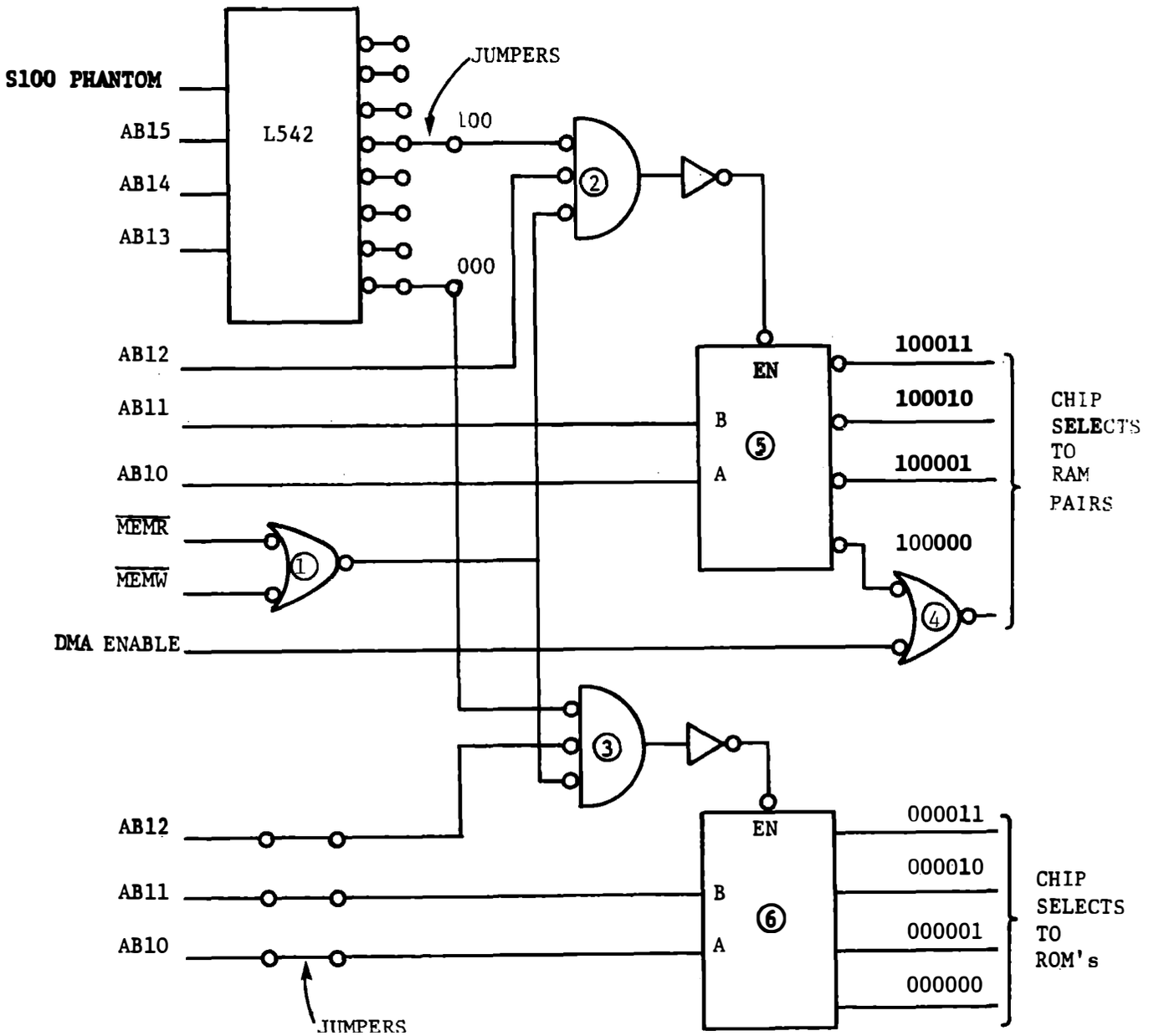
This page intentionally left blank.

5.3 CHIP SELECT LOGIC

The MTS provides for mounting four ROM (or PROM) chips and four pairs of RAM chips. It is supplied with one PROM device and two RAM chip pairs; the other locations are empty. Each memory device receives the ten low order lines of the address bus (AB0 through AB9) to select one byte (or half byte, in the RAM). The six high order address lines (AB10 through AB 15) are decoded externally to select one PROM or two RAM chips. These six lines can select among 64 possible positions of which only three are occupied and only eight can exist on the MTS circuit board. If one of the four PROM locations or one of the four RAM pair locations is addressed, decoding logic shown in Figure 5-5 will generate the appropriate chip select signal.

This is an active low signal, so one of eight chip select lines goes low.

In the following description it is assumed that the reader has at least a slight knowledge of TTL logic and conventional symbols. Readers lacking this knowledge should skip to Section 5.4.



NOTE: Circled numbers are for reference to text only.

Chip Select Logic

Figure 5-5

5.3.1 Memory Enabling

Two signals, $\overline{\text{MEMR}}$ and $\overline{\text{MEMW}}$, are derived by the system controller logic from data output by the microprocessor at the beginning of each machine cycle. If this cycle is to read from memory, $\overline{\text{MEMR}}$ becomes true (low). This occurs for an instruction fetch (the first machine cycle of every instruction cycle), and again to read the second and third bytes of multi-byte instructions or to load data from memory into the microprocessor.

If a data byte is to be written to memory (as in loading a program or in a STA instruction, for instance) $\overline{\text{MEMW}}$ becomes true. Either $\overline{\text{MEMR}}$ or $\overline{\text{MEMW}}$ implies that memory is to be addressed. Various other operations do not require access to the memory and neither of these signals is true. The negative OR gate (1) in Figure 5-5 recognizes that memory access is required and enables gates (2) and (3).

5.3.2 RAM Chip Selection

One pair of RAM memory chips (1024 bytes) will be selected by one of the output lines from the decoder (5). This occurs under the following conditions.

The S-100 PHANTOM is a signal derived from the S-100 bus that can inhibit the addressing of any of the memory on the MTS. This signal must be false. Then if the three high bits of the address bus contain 100, the 74LS42 decoder selects the output line labeled 100 in Figure 5-5, and gives a true (low) signal to gate (2). Gate (3) receives a false (high) signal from the line labeled 000, so its output will remain false.

Finally, address bus line 12 (AB12) must be low to make gate (2) have a true output and enable the decoder (5). Now this decoder selects among four RAM chip select lines according to AB11 and AB10. These lines are labeled with the six bits of the address bus that make them active. The bottom line of this group (100000) addresses the RAM pair for memory addresses 8000 - 83FF. These 1024 bytes include the display, monitor variable data, stack, and all the programs developed in this course. This leads to an important point for the design of small microcomputer systems. To address this 1024 byte RAM pair it would be sufficient to recognize only the high bit of the address bus if no other devices were addressed in the 8000 - FFFF memory area.

Gate (4) allows the selection of the 8000 - 83FF RAM pair in response to DMA ENABLE. This signal is generated during the repetitive accesses to memory to operate the display. At frequent intervals the 8080 processor stops its operations to allow the display circuits to obtain data for the seven segment displays. During this "Direct Memory Access" neither MEMR nor MEMW is active, so both decoders (5) and (6) are disabled, and the RAM chips are selected by the DMA ENABLE signal.

5.3.3 ROM Chip Selection

Now consider gate (3) and decoder (6). These select among the ROM or PROM chips. As for RAM chip selection, either MEMR or MEMW must be true. (In fact only MEMR should be true, since it is not possible to write to the ROM's, but the system hardware does not enforce this limitation.)

The 74LS42 decoder selects the lowest output line (000) if the three high bytes of the address bus contain 000. Now if AB12 is also 0, gate (3) output becomes true, and enables decoder (6). This selects among its four output lines according to AB11 and AB10, to enable one of the four ROM positions on the MTS. Since the monitor program occupies addresses 0000 through 03FF, only the lowest of these four lines will ever be active in normal operation of the MTS as supplied. You can read from a non-existing location:

```
ADDR    0400    MEM                              0400    .FF
```

Pullup resistors on the data bus force the bus content high when no other device drives it. If you now press a hex key the monitor program will attempt to write to this location. The monitor always tests after writing, and indicates an error if writing is not successful.

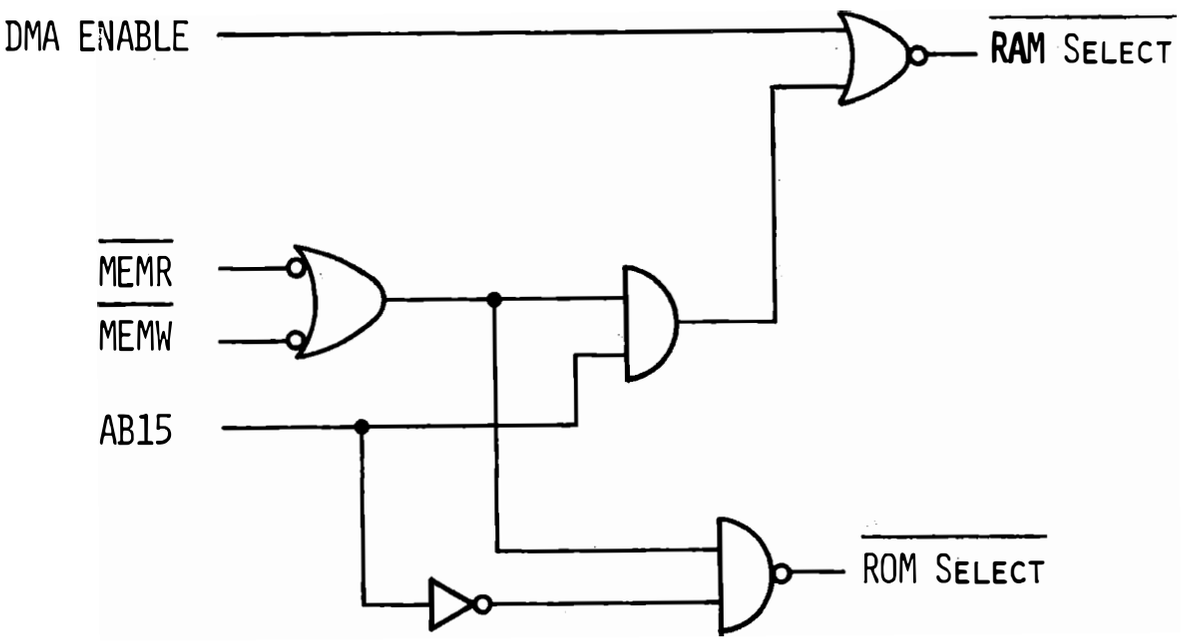
Address	AB15-AB10	Memory Selected
0000-03FF	000000	Monitor PROM
0400-07FF	000001	Empty ROM Position 1
0800-0BFF	000010	Empty ROM Position 2
0C00-0FFF	000011	Empty ROM Position 3
1000-7FFF	000100	No MTS Memory
	to	
	011111	
8000-83FF	100000	RAM Pair 0
8400-87FF	100001	RAM Pair 1
8800-8BFF	100010	Empty RAM Pair 2
8C00-8FFF	100011	Empty RAM Pair 3
9000-FFFF	100100	No MTS Memory
	to	
	111111	

MTS Memory Addresses

Figure 5-6

5.3.4 Partial Decoding

The memory locations that are addressed by the high six bits of the address bus (AB15-AB10) are tabulated in Figure 5-6. In the monitor and in the programs developed in this course only addresses 0000-03FF (the monitor) and 8000-83FF (RAM) are used. The logic of Figure 5-7 would be sufficient to select the RAM if AB15 = 1 and ROM if AB15 = 0. Such an arrangement is perfectly suitable for small microcomputer systems dedicated to well defined applications. With this arrangement, five bits of the address bus are ignored (AB14-AB10). Addresses 8000, 8400, 8800, 8C00, 9000, 9400, etc., are exactly equivalent, any of them reading or writing the same byte in memory. This is referred to as "partial decoding". Its only disadvantage is that it precludes expansion of the system. The MTS uses "full decoding", uniquely addressing each byte of memory, to permit expansion of the system through the S-100 bus interface.



Minimum Chip Select

Figure 5-7

5.3.5 Alternative Memory Addressing

Refer again to Figure 5-5, and note that provision is made for changing the address decoding. The jumpers between the 74LS42 decoder (Figure 5-5) and gates (2) and (3) allow the user to move the physical memory devices on the MTS circuit board to different logical addresses. This is not permissible with the MTS educational monitor, which must be located at addresses 0000-03FF and must have memory at 8000-83FF.

The jumpers between AB12-AB10 and gate (3) and decoder (6) may be reconfigured to permit use of ROM or PROM chips containing 2048 bytes instead of 1024 bytes each. Thus a total of 8192 bytes of ROM could be installed on the MTS for a large system.

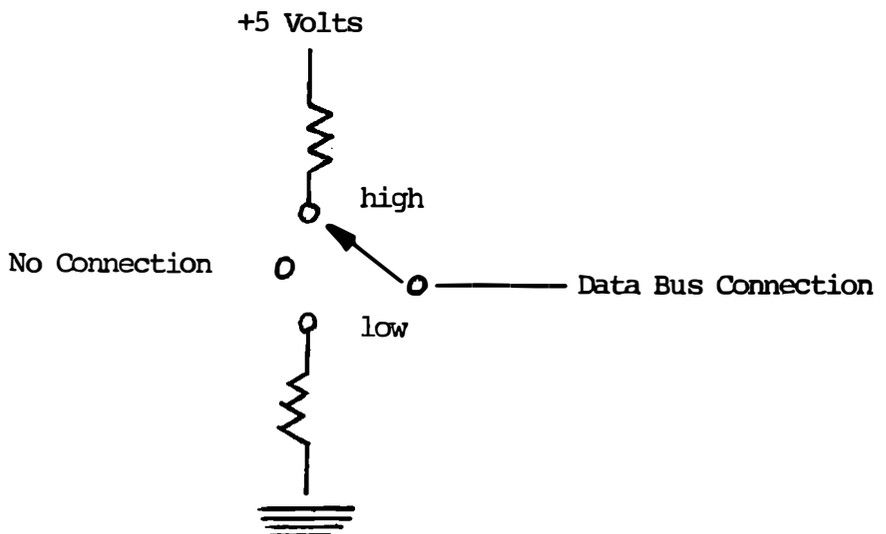
The S-100 bus defines the signal S-100 PHANTOM. If this is made true, all of the MTS memory is disabled. Suppose that you have developed a program which is ultimately to operate at memory locations 0000-07FF. You can use the MTS monitor to load this program into memory physically located in the S-100 system. Then by setting S-100 PHANTOM true you disable the MTS monitor and use the S-100 memory to run your program. Such operations are beyond the scope of this course, and this is mentioned solely to explain the PHANTOM signal.

5.4 DATA BUS CONNECTIONS

Figure 5-1 shows that the inputs and outputs of all the memory devices are connected to a common data bus. Only the chip (or pair of RAM chips) that has been enabled by the high address decoder is allowed to use the data bus: when the bus is active it is driven by one device (memory, CPU, or input) and it drives one device (memory, CPU, or output).

5.4.1 Tri-State Circuits

The device that is to receive data from the bus expects each line of the bus to be in a clearly defined state - one or zero. To achieve this the driving device either pulls the bus down to a voltage level close to 0 volts or pulls it up to a voltage level well above 0 volts - between about 2.5 and 5 volts. Other devices that are capable of driving the bus must not interfere with this operation. A semiconductor circuit for this purpose is called a Tri-State circuit: it has three output states, high, low, and off, and is analogous to a three-way on-off-on toggle switch.



Clearly we could connect many such switches to a data bus line and if exactly one switch is high or low the line will be in a well defined state. The circuit used in the memory uses MOS transistors. If the high transistor is turned on, the circuit delivers current to the line from the 5 volt supply. If the low transistor is turned on, the circuit sinks current to ground. If both are off, the circuit exhibits a high impedance to the line.

Tri-state circuits are used for all connections capable of driving the address bus or the data bus. This includes the 8080 CPU, the System Controller, each 2708 ROM and 2114 RAM (on the data bus only), and the 8255 Peripheral Interface.

5.4.2 Read-Write Control

In addition to allowing many devices to share the data bus, the tri-state circuit allows the individual device to use the same pins for input and output. When a device has been selected by the address bus decoder it observes the control lines from the system controller (the control bus), signals which are derived from the CPU.

A memory read operation causes the selected memory device to connect the outputs of the selected memory location to the system data bus by enabling the tri-state output to enter its high or low state.

When its tri-state circuits are in the high impedance state the device can sense data that the CPU has placed on the data bus. When a signal from the CPU commands a memory write operation, the selected device copies data from the bus to the inputs of the storage flip flops addressed by its internal decoder.

A similar operation occurs in the 8255 Peripheral Interface device when the CPU commands an input or output operation. On input the 8255 copies data from its external ports (from the keyboard, for instance) onto the data bus. On output the 8255 senses the data bus and copies the data to the output ports.

Some memory devices (such as the 2101) have separate input and output pins, but still include tri-state circuits controlled to permit both inputs and outputs to be connected to the data bus. Other memory devices (such as the 2102) do not permit such direct connection of outputs and inputs. Although the outputs have tri-state circuits, these are enabled whenever the chip is selected. Therefore a separate tri-state circuit must isolate the outputs from the data bus during memory write.

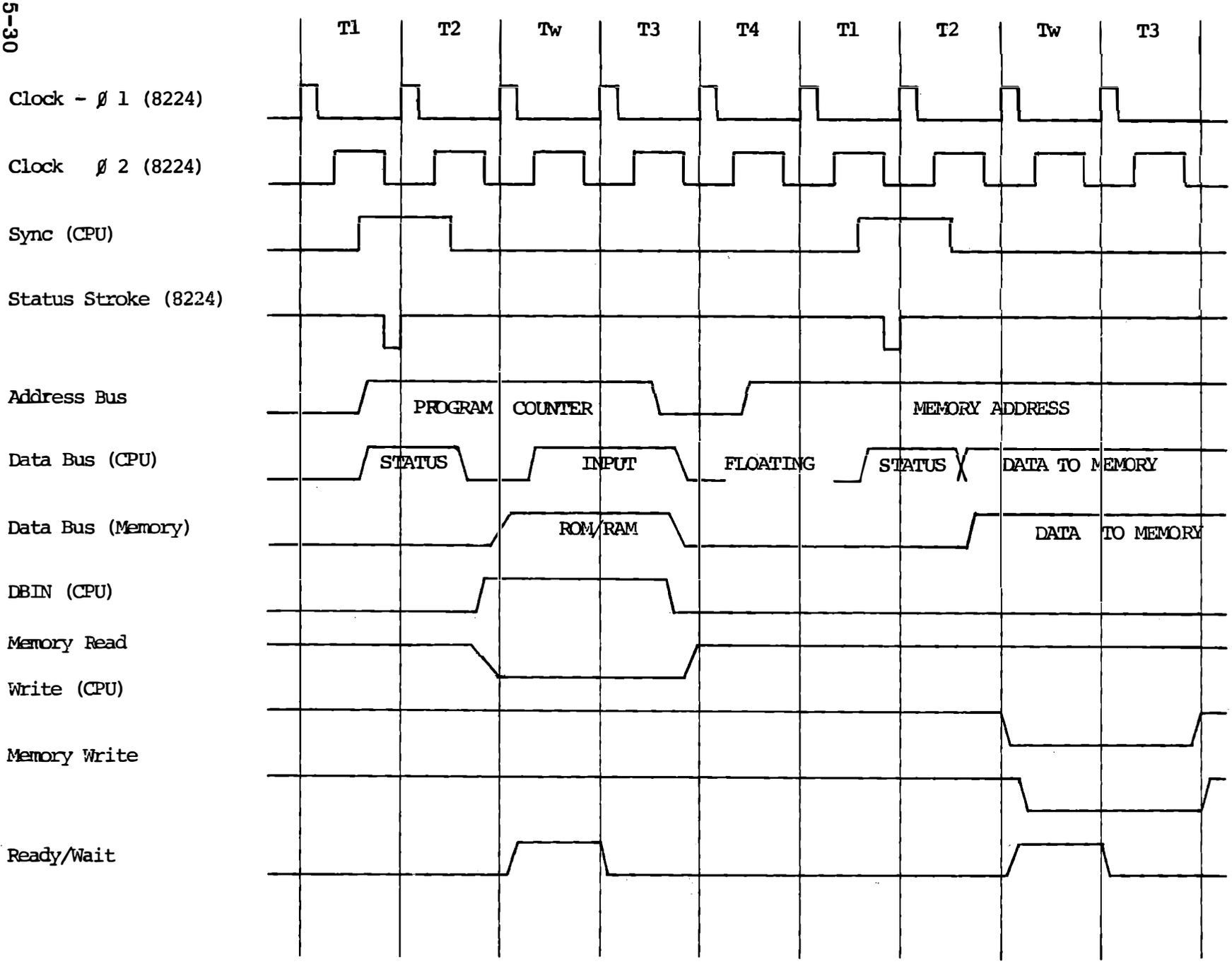
5.4.3 DMA and Interrupts - Introduction

The 8255 provides for programmed input and output. It sends data to the CPU from the external world when the program requests it, and it sends data to the external world when the program so specifies. There are two other means of input and output used in computers, and the MTS employs both of them. Direct Memory Access and Interrupts both provide for input or output on demand of an external device instead of on demand by a program. These subjects are discussed in detail in a later chapter; at the moment we are concerned with their relationship to memory and the buses.

Direct memory access permits an external device to read or write to the computer's memory without program control or CPU intervention.

When the device needs access to the memory it generates a signal to the CPU requesting a HOLD state. When the CPU finishes the current machine cycle it acknowledges the hold and relinquishes control of the memory, placing its address and data bus drivers into the high impedance condition. The external device -- the DMA channel -- now drives the address lines and the read and write control lines. If memory read is being requested, the selected memory device drives the data bus just as if the CPU had commanded a memory read - the memory does not know the difference. The DMA channel accepts the data from the bus, then returns control to the CPU by dropping the hold request.

The Interrupt method of externally controlled input and output involves only the data bus. An interrupt request is delivered to the CPU, which finishes the current instruction and relinquishes control of the buses. The interrupting device proceeds to place an instruction on the data bus, and the CPU treats this as though it were an instruction read from the program memory. Eight RST instructions are provided for this purpose. As you have seen, RST4 as an instruction in your program causes an entry to the monitor program. If it were entered by means of an external interrupt, exactly the same process would occur. Usually the interrupt initiates a programmed input or output operation; this is treated in Chapter 8.



Memory Access Timing

5.5 MEMORY SIGNALS AND TIMING

5.5.1 Machine States and Transitions

Figure 5-8 shows the signals involved in memory access during the MOV M,A instruction cycle. The system clock is driven by the 8224 clock generator, which includes an oscillator controlled by an external crystal. The oscillator is counted down and divided into a two phase clock: the ∅1 and ∅2 clocks, as shown. SYNC is generated by the CPU at the beginning of each machine cycle. The ∅1 clock period marks "states" of the processor. Each machine cycle has three or more states (clock periods). Each instruction cycle has one or more machine cycles. We will proceed along the time axis and explain the states as we meet them.

5.5.2 First State (T1)

During the last half of state T1 and the first half of state T2, the CPU generates a SYNC signal, and outputs on the data bus an eight-bit status word designating the kind of machine cycle that is being performed. In the first machine cycle of any instruction this is always an instruction FETCH.

The clock generator receives the SYNC signal and generates a status strobe in response: This is a narrow pulse which the system controller uses to latch the status data.

The CPU also connects its program counter outputs onto the address bus during the instruction FETCH machine cycle. This connection is retained through most of the machine cycle. All of the memory

devices receive the address (10 low-order bits) and decode it, and the external decoder selects one of the memory devices.

The system controller recognizes that this is an instruction FETCH cycle and generates the MEMORY READ signal. This is an active low signal; the near 0 volts condition tells the memory to read. It is timed by DBIN to ensure that the memory does not drive the data bus until the CPU has released the bus.

5.5.3 Second State (T2) and Wait (TW)

During state T2 a signal (DBIN) is raised to receive data. The DBIN signal is terminated during state T3. Some memory devices are too slow to deliver data to the CPU by this time, or if the memory is physically separated from the CPU the cables may introduce an excessive delay. To provide for this, if the READY signal to the CPU is low at the end of T2 the CPU enters a WAIT state, TW. The WAIT state is repeated until READY is high at the end of a clock period. Figure 5-8 shows one WAIT cycle with each memory access. This does not occur in the MTS when it operates with its own memory, but is required if it operates with S-100 memory. The READY signal can also be used during input or output to slow peripheral devices.

5.5.4 States T3, T4 and T5

During T3 the data bus is read by the CPU, and since this is an instruction FETCH it is loaded to Register I. The instruction is interpreted during T4, at the end of which a new machine cycle begins. The T5 state is available for certain instructions, but if not required T1 follows T4.

Since the instruction in Figure 5-8 is MOV M,A a MEMORY WRITE cycle is required. The CPU again outputs SYNC, Status and an address, but now the address is the content of (H,L). During T2 the CPU places the content of Register A on its data bus and the system controller passes it on to the system data bus. The CPU status indicates that a memory write cycle is required, so the system controller generates MEMW. Once again a WAIT state is shown. After TW the standard T3 state occurs. With fast memory the T3 state provides time enough for writing. The TW state doubles that time, while reducing the processor's speed by about 25%.

This page intentionally left blank.

MICROCOMPUTER TRAINING WORKBOOK

CHAPTER 6

MODULES, SUB-ROUTINES AND THE STACK

6.1 PROGRAM MODULES

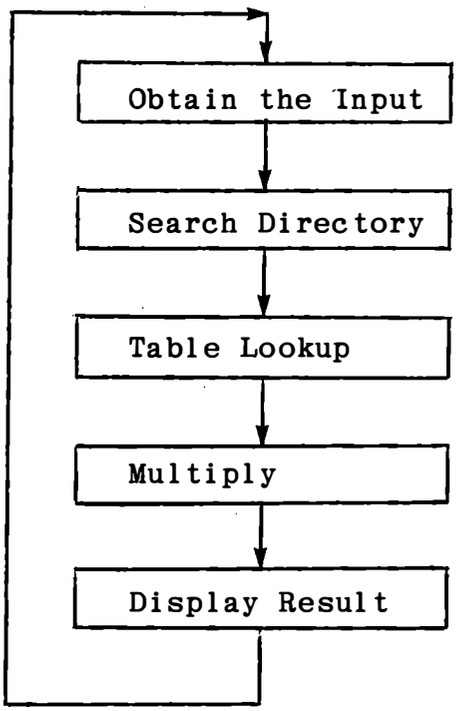
The design and hardware of a complex machine are always divided into modules, each having a limited function and a limited set of inputs and outputs. The purpose is to make each module comprehensible to the designer and to make it fit within a physically realizable structure (such as a circuit board). Often modules operate in parallel because their functions are separable but must or can overlap in time.

The design of a machine that uses a microprocessor is handled the same way. The microprocessor is part of a solution; it is surrounded by other hardware modules that relate to it. The program of the microprocessor is similarly divided into modules, which relate to each other and to the surrounding hardware. Your microcomputer training system and its monitor program include a clear example of this: when you press numeric keys they are displayed, but in the hardware there is no physical connection between the keyboard and display. There is a program module which services the keyboard and a program module which services the display. These operate independently, and other program modules determine their interactions, which vary with time and history. When you press a hexadecimal key it may be displayed in any of six positions depending on what command key and other hexadecimal keys you pressed before. (In a later chapter we will examine the design of the MTS and its input and output electronics and programming.)

6.1.1 In-Line Programming

Consider the sensor correction program of Chapter 4:

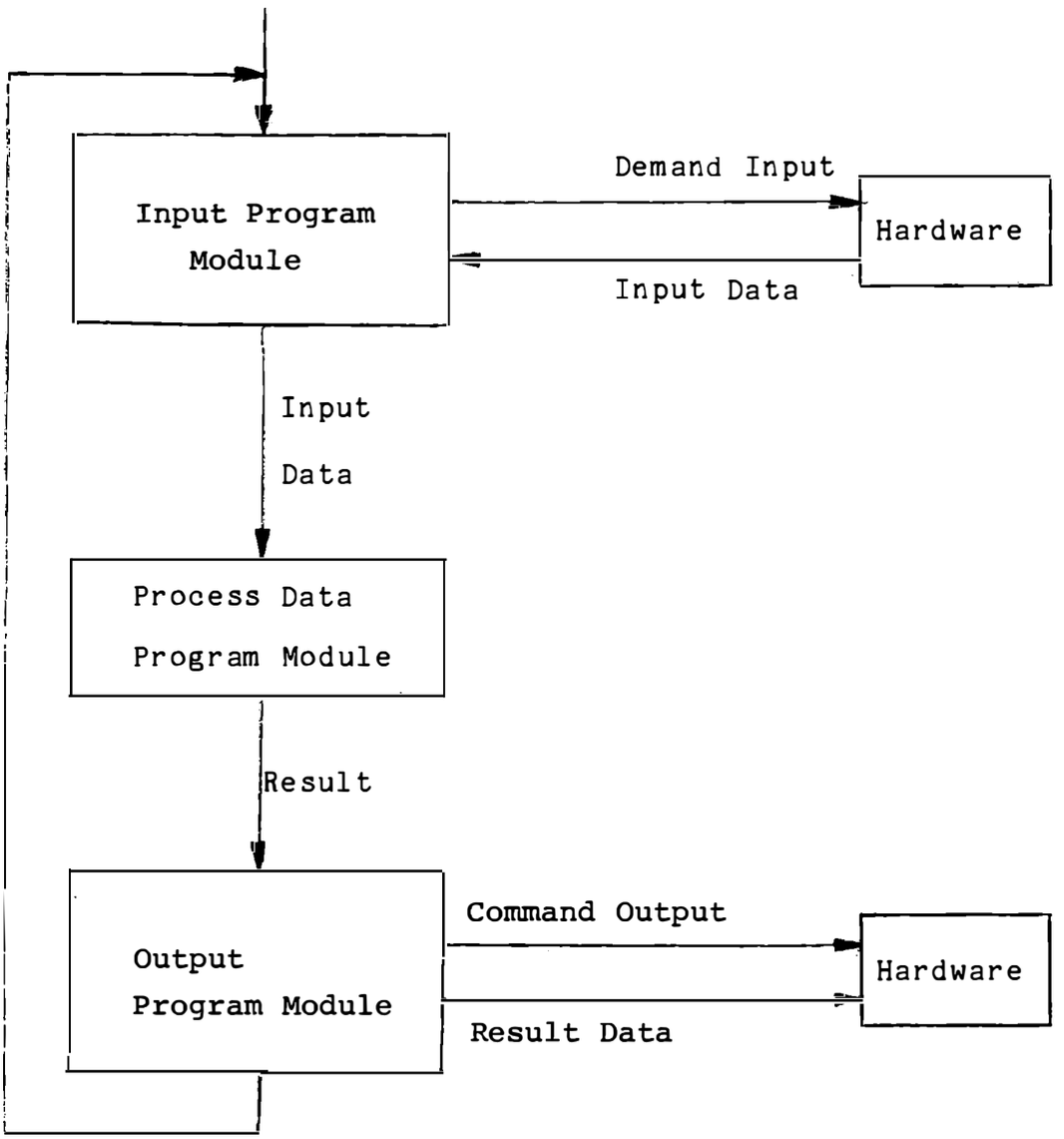
If the input and output functions were part of your program you might program them all "in-line", with a series of instructions to accept hexadecimal keys and display them (possibly with a loop for input of two or more keys), followed by the instructions for the directory search and table lookup for a linearized value, followed by the multiplication for scaling, then the commands to output the result, and finally a jump back to the beginning.



6.1.2 Creating Program Modules

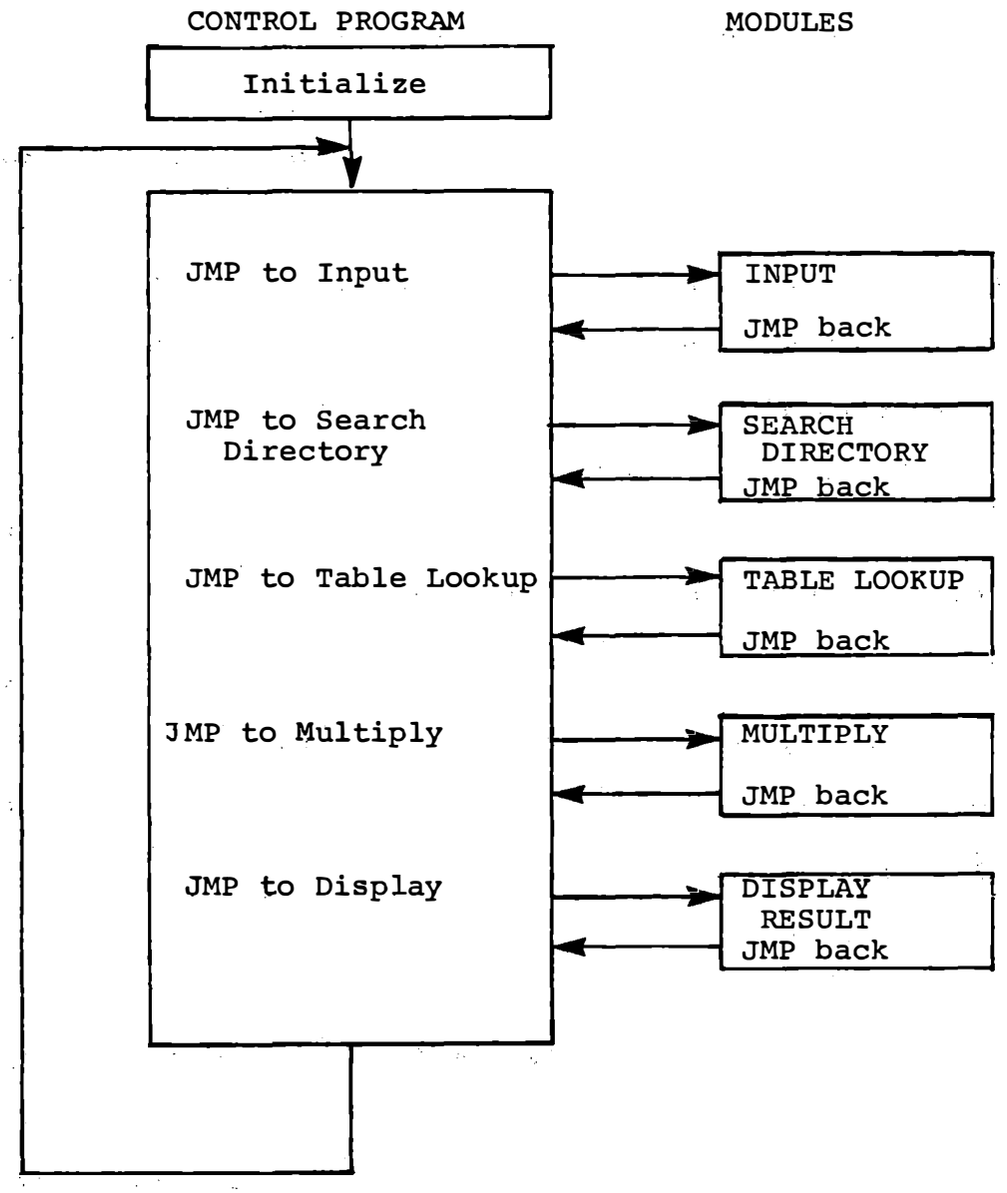
As these procedures become sufficiently complex, it is desirable to distinguish each of them as a separate module and develop it independently. This can be done with a subsequent integration of the several modules into an in-line program.

Consider an in-line procedure comprising input, process, and output.



The input may involve several data items (for instance, sensor number and data input), and the input program module retains control until the requisite data items have been obtained. There may be loops and decision points within the module, but control stays there until the task has been completed. Then some data processing occurs, which may involve loops, table lookup, and perhaps use of previous data. Again, control remains with this program module until its task is done. Finally results are passed to an output module which sends out the data. Such a procedure is exemplified by the sensor correction problem in Chapter 4, although we entered the monitor for input and output. (By the end of this chapter you will have learned ways to call upon the monitor for input and output as separate functions.)

Another way of organizing a program is to write the separate modules, locating them in different areas of program memory, and provide a control program that jumps to each module in turn. This is suggested in Figure 6-1. Why would we do this? In the sensor correction exercise of Section 4.12 we used a directory procedure that required all data tables to fit into a single page (8300 --) of memory. If we found later that more sensors or larger tables were needed, we might need a directory with two byte addresses. If the program were organized as Figure 6-1 we could rewrite the SEARCH DIRECTORY module with no effect on any other module. If we found it desirable to have the microprocessor select the sensor to be read instead of taking sensor number as an input, we would modify the input module, and possibly add a new module to select the sensor.



Program Modules with Control Program

Figure 6-1

As long as the overall function remains unchanged and no new modules are added, the main program retains the same jumps - one to the start of each module. Each module jumps back to the main program location following the instruction that jumped to the module. When each jump occurs, there usually is some information to be passed to the module or back to the main program: at least the inputs and results. These data may be in registers (the inputs and outputs, for instance) while other data might be in specified memory locations.

6.1.3 Module Specification

Now consider the program specification for each module. Suppose each were to be designed independently; what must its designer be given? Here are some of the important considerations:

Function:

Specify the "black box" algorithm for the module.

Entry:

The address to which the master program must jump.

Extent:

The range of program memory allotted to the module (starting and ending addresses or number of memory words used).

Inputs:

Identify the inputs to be given to the module. What are they, and where will they be? In what register or memory location? How many bytes? (Recall the specification of register assignments in Section 4.4.4.)

Outputs:

Identify the results the module is to generate. What are they, and where must the module place them?

Registers:

What registers are used or preserved? (Recall that we preserved sensor number in Register B.)

Constraints:

What memory areas may the module use for data storage, either temporary or permanent? Is the module permitted to use all of the registers, or must certain ones be preserved? How much time is permitted for the module's function?

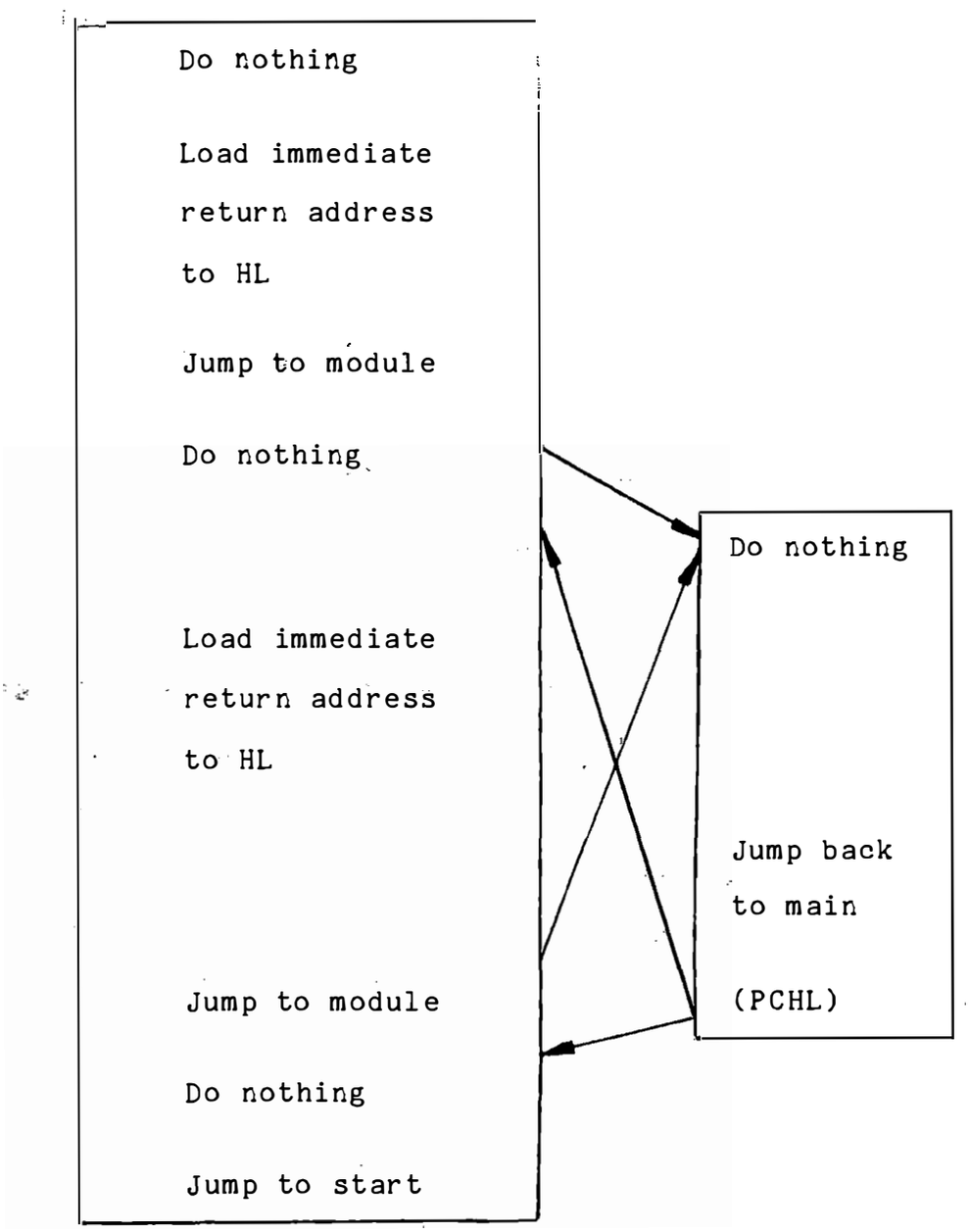
It may appear that the need to specify all of this (and often much more) makes the use of program modules a nuisance. In fact it is one of the best reasons for modular design: it forces a discipline that may otherwise be neglected. When such items are well-defined, many programming errors may be avoided.

Suppose that one module serves a function that is needed several times in the program - displaying data, for instance. In the sensor correction program it would be desirable to display the sensor number and the input data; later we display the result. If we jumped to the display module with an additional variable (perhaps in an unused register) indicating whether the entry is for input or result, the display module could test that variable and decide where to return. This would demand that the specification include two return addresses and a definition of the new control variable.

A much better procedure is for the main control program to pass the return address as a variable. Then we need a jump instruction that can use a variable address. We have such an instruction:

HEX CODE:	E9
MNEMONIC:	PCHL
MEANING:	Move the contents of register pair H,L into the program counter and continue program execution from that address.

To experiment with this we will write a trivial program that does nothing except load a variable, return address, and jump to a module, which does nothing except jump back. Figure 6-2 is a flow chart of the program shown in Figure 6-3. The return address to be loaded must be the address of the instruction following the jump into the module.



Do Nothing Program With Do Nothing Module

Figure 6-2

DO NOTHING PROGRAM

ADDR	CODE													
8200	00		NOP											
1	00													
2	00													
3	21		LXI		H,	8209								Address of next
4	09													NOP Instruction
5	82													
6	C3		JMP			8220								Jump to Module
7	20													
8	82													
9	00		NOP											
A	21		LXI		H,	8210								Address of next
B	10													NOP Instruction
C	82													
D	C3		JMP			8220								Jump to Module
E	20													
F	82													
8210	00		NOP											
1	C3		JMP			8200								Jump to start
2	00													
3	82													
4														
5														
6														
7														
8														
9														
A														
B														
C														
D														
E														
F														
8220	00		NOP											Module
1	E9		PCHL											Jump to address in HL
2														
3														
4														
5														
6														
7														
8														

Figure 6-3

When you have loaded the program, step though it. The program counter should show this sequence:

8200	00	NOP
8201	00	NOP
8202	00	NOP
8203	21	LXI H, 8209
8206	C3	JMP 8220
8220	00	NOP
8221	E9	PCHL
8209	00	NOP
820A	21	LXI H, 8210
820D	C3	JMP 8220
8220	00	NOP
8221	E9	PCHL
8210	00	NOP
8211	C3	JMP 8200
8200	00	NOP
8201	00	NOP
	etc.	

Of course if H,L were needed for other purposes we could have stored the return address in memory. In fact, the use of a variable return address is so common that the microprocessor has special jump instructions that do this for us automatically. When these are used the module becomes a subroutine.

6.2 SUBROUTINES

A subroutine is a program module that uses built-in features of the computer for entry to the module, and return from the module.

6.2.1 Subroutine Entry and Return

The entry to a subroutine is made by a special kind of jump instruction, CALL, which includes the address of the subroutine just as an ordinary jump instruction includes an address. The microprocessor automatically generates and saves an address for a subsequent jump back to the calling program, executed at a RETurn instruction.

SUBROUTINE:	A program module which is entered by means of a CALL instruction and which normally returns to the calling program by means of a RETurn instruction.

CALLING PROGRAM:	The program module which has called a subroutine. The calling program may be the main program or another subroutine.

The CALL instruction is:

HEX CODE:	CD
MNEMONIC:	CALL
SECOND BYTE:	Low address
THIRD BYTE:	High address
MEANING:	Save the address of the next following instruction, and jump to the subroutine whose first instruction is located at the address given in Bytes 2 and 3.

The CALL instruction executes a jump, but instead of discarding the present content of the program counter it stores (PC) in an assigned memory area called the stack.

STACK:	An area of memory assigned by the programmer for the temporary storage of return addresses or other data. It is addressed by a dedicated 16-bit counter called the Stack Pointer.

The jump back to the calling program is made by the RETurn instruction:

HEX CODE:	C9
MNEMONIC:	RET
MEANING:	Recover the address stored by CALL and jump to that location.

6.2.2 Tracing Subroutine Entry and Return

Revise the Do Nothing program (Figure 6-3) by replacing the following op-codes (the JMP addresses are not changed):

Address	Was	Change To
8206	C3 JMP	CD CALL
820D	C3 JMP	CD CALL
8221	E9 PCHL	C9 RET

Again trace the program flow and observe that the program counter sequence is the same; only the instructions change. The two LXI H instructions could be changed or removed with no effect. Now we will examine and define the CALL and RET instructions more thoroughly, and discuss the stack.

Use the "Do Nothing" program to follow this. Step through your program to 8206, the CALL:

```
STEP                                    8206    CD
```

The monitor can display the stack pointer as a register pair. Key 1 is also labelled P to designate the stack pointer.

```
ADDR    1/P     MEM                     83E0    SP.??
```

Now step once to execute the CALL instruction:

```
STEP                                    8220    00
```

Display the stack pointer again:

```
        ADDR    1/P     MEM                             83DE    SP09
```

The stack pointer contains the address in memory where the low byte of the return address (8209) is stored. The next memory location contains the high byte of the return address:

```
        NEXT                                            83DF     82
```

Any time that you display a register pair and the memory location it addresses you can see the following sequential memory location by pressing NEXT. In debugging programs you will more often be interested in the return address than the value of the stack pointer. Key 2 is labelled T to designate the stack top - two bytes in the stack.

```
        ADDR    2/T     MEM                             8209    ST00
```

The stack top contains the return address.

Now step twice to return to the main program:

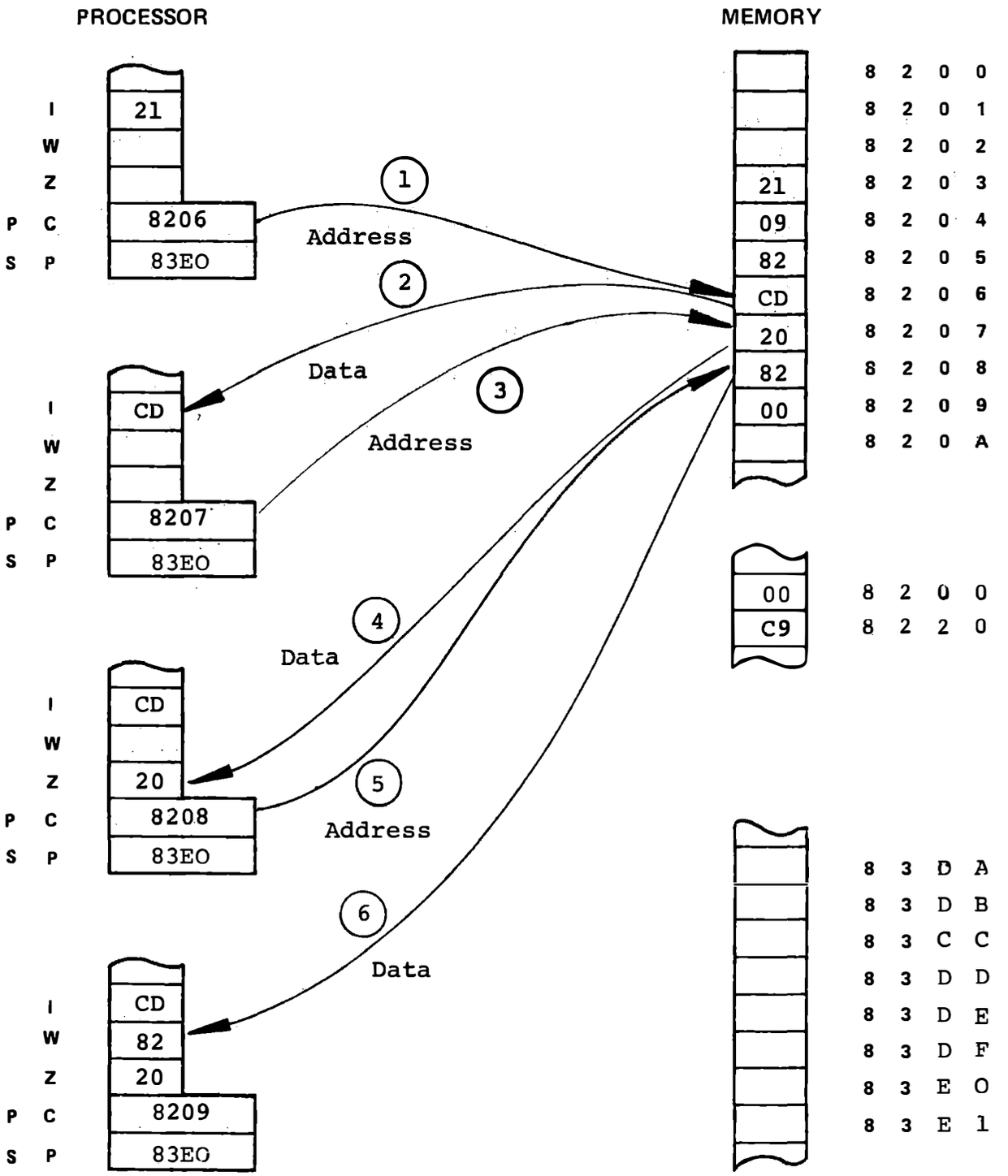
```
        STEP                                            8221     C9
        STEP                                            8209     00
```

The return address has been placed in the program counter.

6.2.3 CALL Execution

Figure 6-4 shows the program counter addressing 8206 and the CALL instruction being loaded into the instruction register. The program counter is incremented three times as the op code and the following two bytes are loaded into Registers I, Z and W respectively. So far the process is identical to that of a JMP instruction, as described in Chapter 2. We see that the program counter now addresses the next instruction following CALL, which is to be the return address. Registers W and Z contain the jump address. The stack pointer addresses a location (83E0) near the top of memory; this was loaded by the monitor program when power was turned on. (The description continues on the next page.)

CALL INSTRUCTIONS

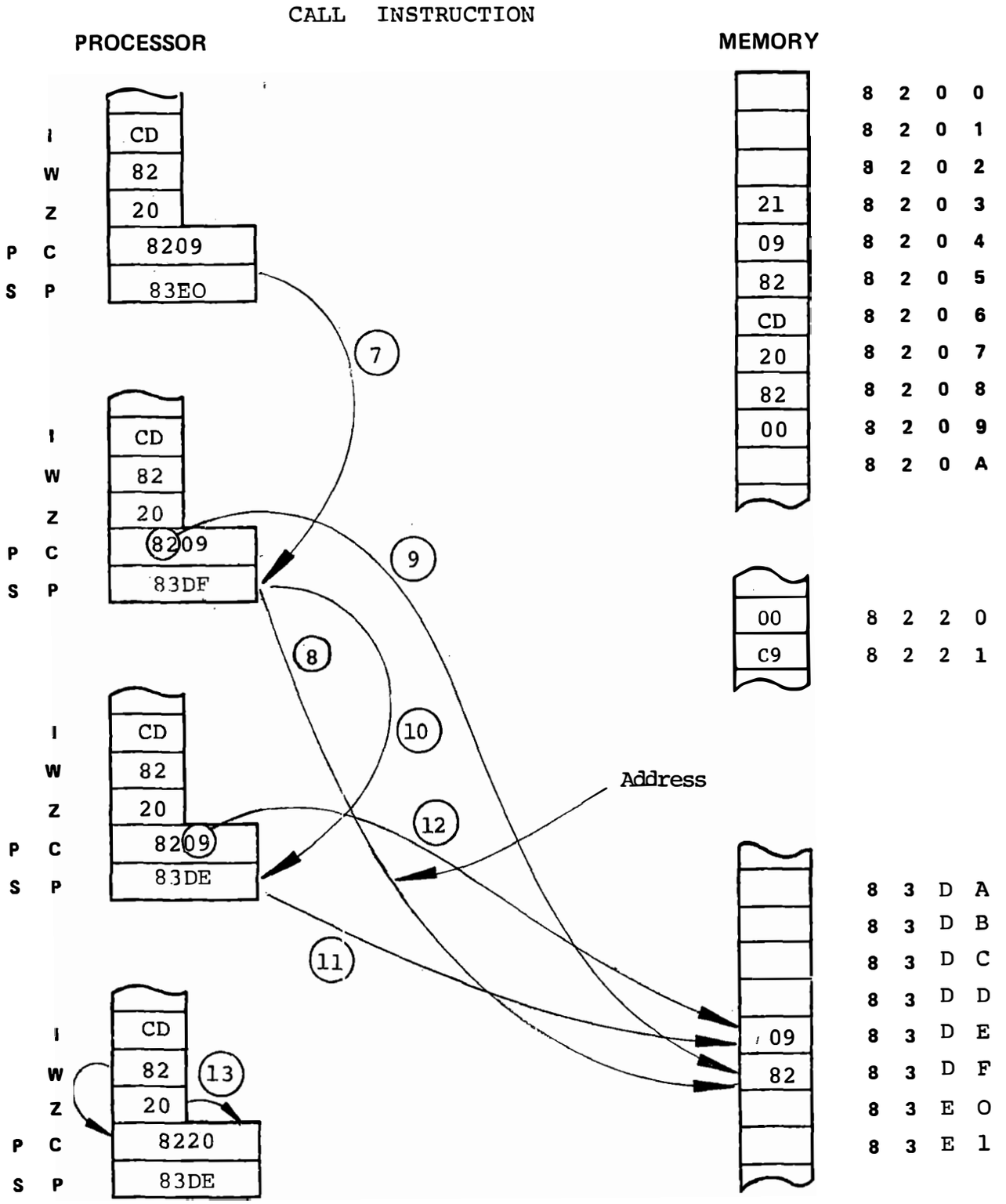


As in a jump instruction, the PC is used to address the instruction code and the two following bytes, which are loaded into I, Z and W respectively

Figure 6-4

Figure 6-5 shows the stack writing operation in a CALL instruction. The content of the stack pointer is decremented (7) and sent out on the address bus (8). The high byte of the program counter is sent out on the data bus (9) to be written to the selected location in the stack area of the memory. Now the stack pointer is decremented again (10) and the low byte of the program counter is written to the memory at the next location below the high byte (11, 12). Any 8080 instruction that stores an address places it in the same position sequence - low byte at the lower memory location.

Finally the subroutine address is moved (13) from Registers W and Z into the program counter, as in a normal jump, and program execution continues with the instruction there.



The stack pointer is decremented (7) and sent out as an address (8). The high byte of the program counter is sent on the data bus (9) and written to the addressed memory location. This is repeated for the low byte of the program counter (10,11,12). Then the content of W,Z, is moved to PC.

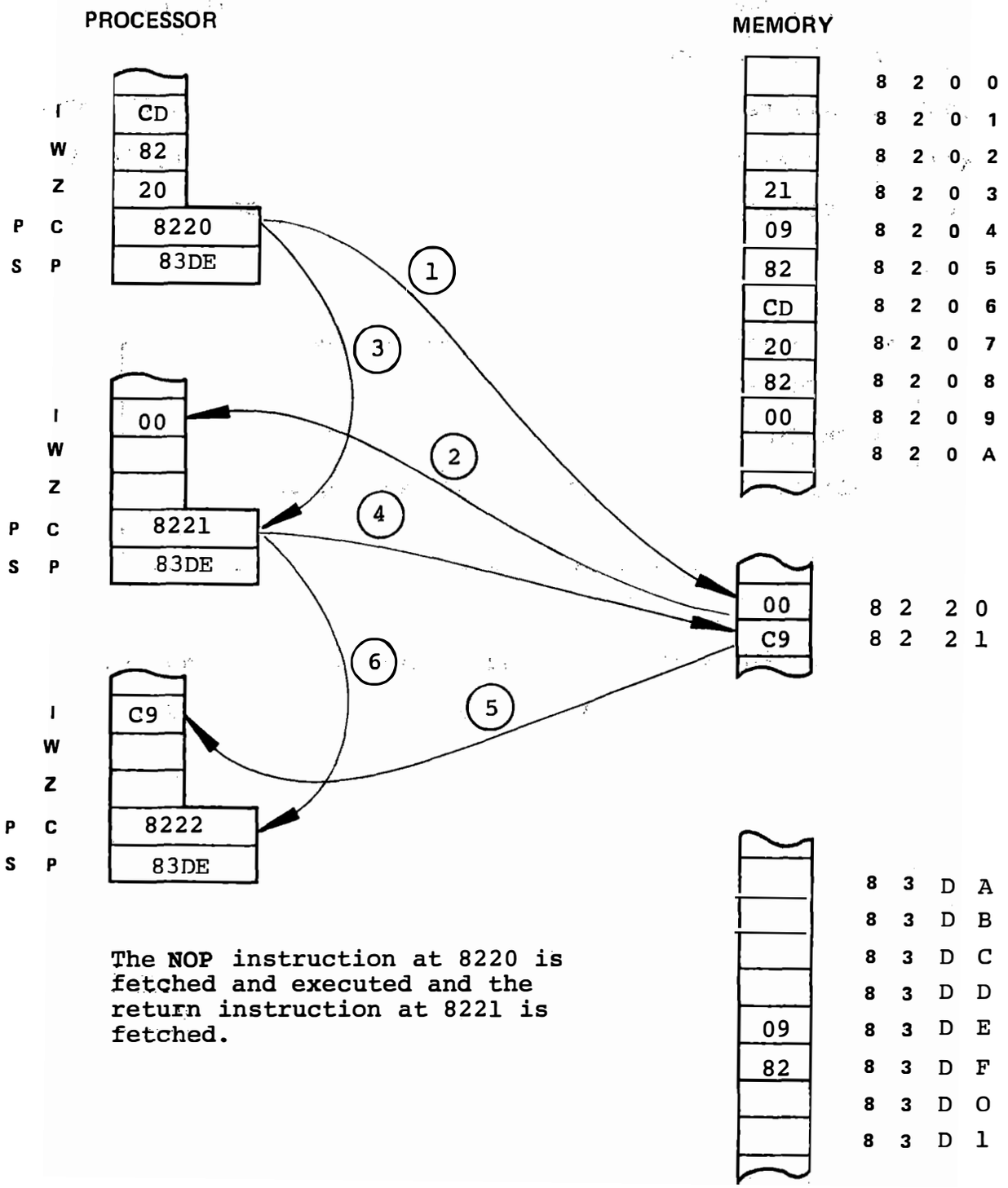
Figure 6-5

6.2.4 Return Instruction

The RET instruction recovers the last address entered in the stack and executes a jump to that address. Note that although RET is a jump it only requires one byte in the program (like PCHL) because the address to which it jumps is a variable stored by the CALL. The RET instruction cycle is shown in Figures 6-6 and 6-7.

HEX CODE:	C9
MNEMONIC:	RET
MEANING:	Return to the calling program.

Figure 6-6 shows the fetch and execution of the NOP instruction at 8220 and fetch of the RET instruction (C9) at 8221. Execution of the return is shown in Figure 6-7.

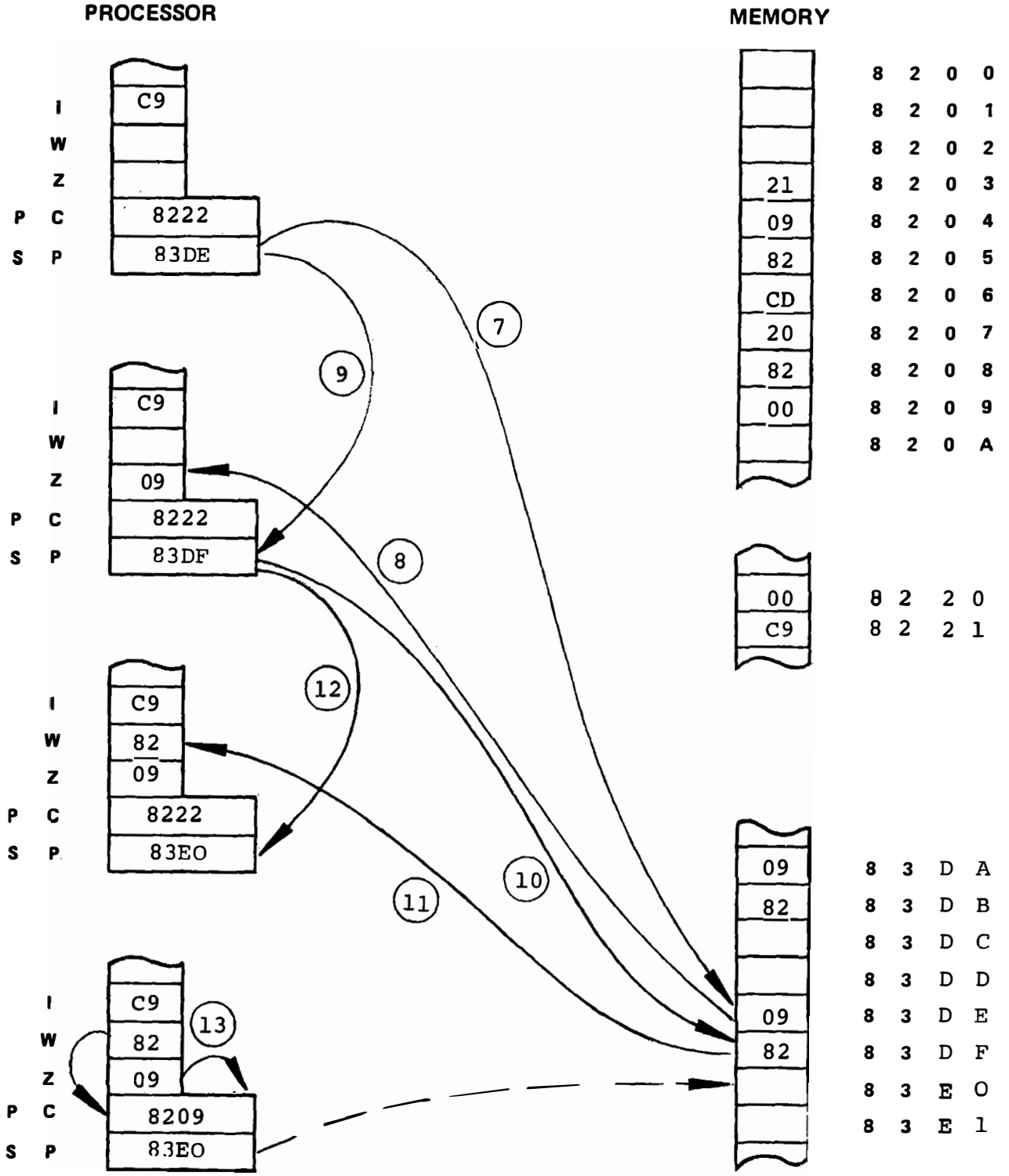


The **NOP** instruction at 8220 is fetched and executed and the return instruction at 8221 is fetched.

Figure 6-6

In Figure 6-6 we saw the RET instruction loaded to the I register. Its execution appears in Figure 6-7. The stack pointer provides a memory address (7) and the low byte of the return address is moved into Z (8). The stack pointer is incremented (9) to address the high byte (10), which is moved into W (11). The stack pointer is incremented again (12) and the content of W and Z is moved to the program counter to accomplish the jump (13). Notice that this process is identical to a normal jump except that after the instruction fetch, the stack pointer is used instead of the program counter to read the jump address.

RETURN INSTRUCTION Cont'd



The stack pointer addresses the low byte of the return address which is loaded to Z (7,8). The stack pointer is incremented (9) and the high byte is loaded to W (10,11). The stack pointer is incremented again (12) and the program counter is loaded from W and Z.

Figure 6-7

6.2.5 Subroutine Nesting

Why is the return address stored in memory? Since a 16 bit register exists (the stack pointer), why not simply place the return address in that register? In fact, this scheme was used in early computers, and still appears in such small microprocessors as the 4004 and 4040. The problem is that if only one register exists there can be only one level of subroutine: one subroutine cannot call another subroutine. The 4004 and 4040 have four return address registers, so that four levels of subroutines can be used.

This is still a noticeable limitation. Using a memory stack permits unlimited subroutine nesting. Figure 6-8 shows some nested subroutines. Note that there is no inherent "level" to a subroutine. Any subroutine can be called from the main program or from any other subroutine.

Load the program (Figure 6-9) and trace the program flow, as described on the following pages.

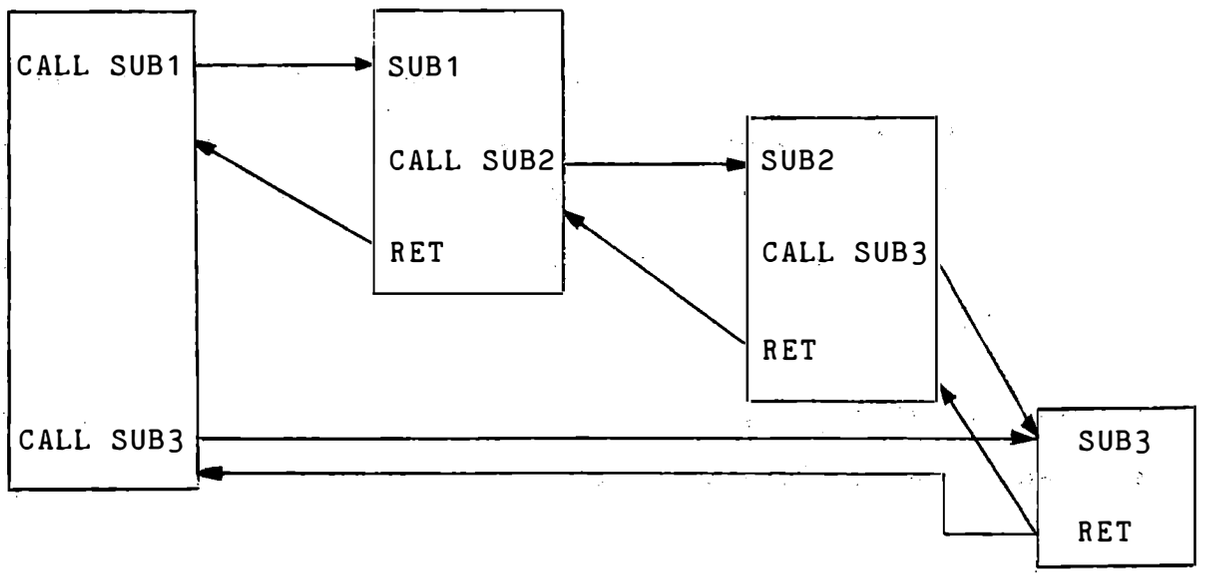

CALL SUB1
SUB1
CALL SUB2
RET
SUB2
CALL SUB3
RET
CALL SUB3
SUB3
RET

Figure 6-8

NESTED DO NOTHING SUBROUTINES

CODING SHEET

ADDR		CODE														
8 20	0	00		NOP												MAIN
	1	CD		CALL				SUB1								
	2	10														
	3	82														
	4	00		NOP												
	5	CD		CALL				SUB3								
	6	1C														
	7	82														
	8	00		NOP												
	9	C3		JMP				8200								
	A	00														
	B	82														↓
	C															
	D															
	E															
	F															
8 21	0	00		NOP												SUB1
	1	CD		CALL				SUB2								↓
	2	16														
	3	82														
	4	00		NOP												
	5	C9		RET												↓
	6	00		NOP												SUB2
	7	CD		CALL				SUB3								↓
	8	1C														
	9	82														
	A	00		NOP												
	B	C9		RET												↓
	C	00		NOP												SUB3
	D	C9		RET												↓
	E															
	F															
8	0															
	1															
	2															
	3															
	4															
	5															
	6															
	7															
	8															

Figure 6-9

Trace the program flow through the dummy subroutines of Figure 6-9. Step to address 821C.

RST	8200	00
STEP - - - - - STEP	821C	00

Display the stack pointer, and examine the stack.

ADDR 1/P MEM	83DA	SP1A	Return
NEXT	83DB	82	to SUB 2
NEXT	83DC	14	Return
NEXT	83DD	82	to SUB 1
NEXT	83DE	04	Return
NEXT	83DF	82	to MAIN

Now execute the NOP and RET intructions.

STEP	821D	C9
STEP (back in SUB 2)	821A	00

The stack pointer now addresses the return address that will take us back to SUB 1.

ADDR	1/P MEM	83DC	SP14
STEP		8213	C9
STEP	(back in SUB 1)	8214	00

The stack pointer now addresses the return address that will take us back to MAIN.

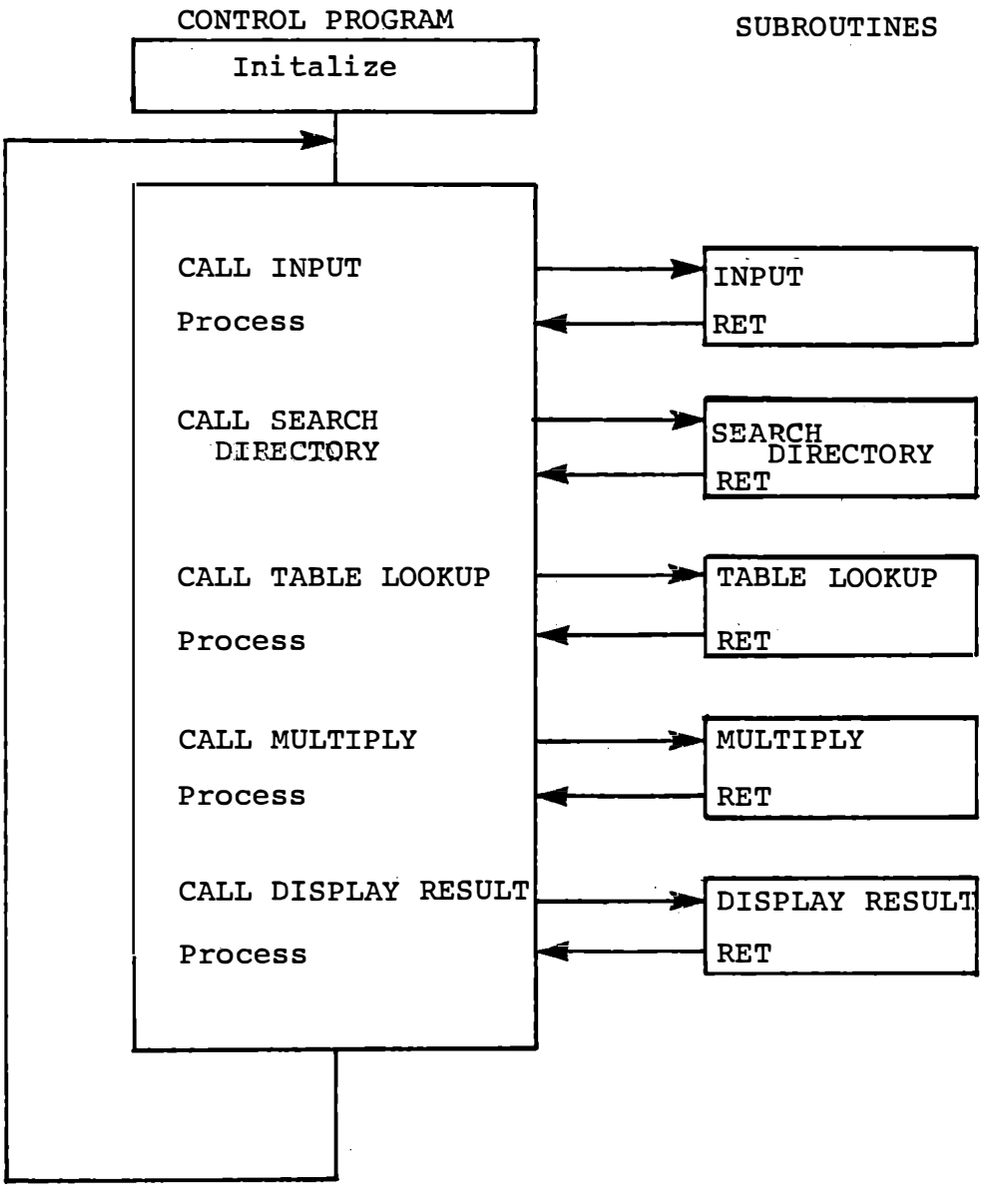
ADDR	1/P MEM	83DE	SP04
STEP		8215	C9
STEP	(back in MAIN)	8204	00
STEP	(call SUB 3)	8205	CD
STEP	(in SUB 3)	821C	00
STEP		821D	C9
ADDR	1/P MEM	83DE	SP08
STEP	(back in MAIN)	8208	00

6.3 SUBROUTINE SPECIFICATION

The central reason for writing modules as subroutines is to permit the same module to be called from various program locations; however, there are two extra advantages: The single byte RET saves program space, and it avoids the need to specify the return address during program design. Therefore most program modules are written as subroutines even if they are to be used only once.

We commonly give a name to a subroutine (INPUT, DISPLAY, SEARCH DIRECTORY, TABLELOOKUP, MULTIPLY). This is a convenience for the programmer, like the mnemonic names of instructions. It is much easier to remember a name than an address, and the name conveys some meaning. However, a subroutine has an address, the address of its first instruction. When you write the CALL instruction you must, of course, use the hexadecimal address of the subroutine, just as you would use an address in a jump instruction.

Figure 6-10 shows a flow chart for the sensor correction problem written as a series of subroutines and a main program. We shall briefly define all of the subroutines, and then develop them one at a time, with detailed specifications.



Sensor Correction with Subroutines
Figure 6-10

6.3.1 Program Development - Sensor Correction Problem

Developing a program generally involves these steps:

a) Define the problem
b) Conceive a program solution
c) Divide the solution into comprehensible and realizable program modules
d) Specify the modular functions
e) Specify the interfaces
f) Develop the main control program
g) Develop and test the modules
h) Integrate and test the system

In Chapter 4 we defined the sensor correction problem and conceived a solution. Now we have divided the program into modules. It remains to specify the functions and interfaces of the modules, to develop and integrate them. First we will give brief functional specifications. These will be developed more fully later.

Subroutines for Sensor Correction

Input:

Accept data input from the keyboard. Display the data as it is entered. On a specified command, change the sensor number. Return when a command is entered.

Search Directory:

Find the table address for the present sensor number.

Table Lookup:

Obtain the scaling factor and linearized value of the input from a data table

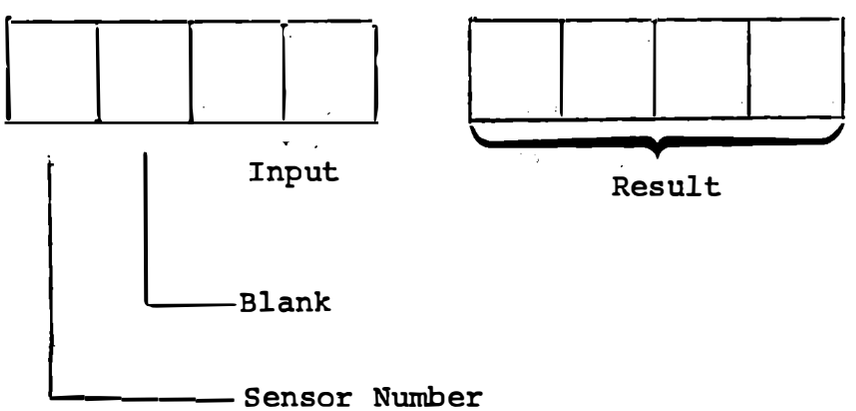
Multiply:

Generate the product of the scaling factor and the linearized value of the input as a double precision result

Display Result:

Display the double precision result.

We must also define the displays to be generated by this program. Data to be displayed are the sensor number, input byte, and result.



6.3.2 Main Program

A good procedure for developing a program that comprises a number of subroutines is to develop the main program first, using CALL instructions to call the various subroutines. At each subroutine location enter nothing but a RET instruction. You can then step through the main program to test the program flow, even though the subroutines do nothing. Then develop each subroutine in turn; as these are entered you can test them by running the main program. When all of the subroutines have been developed and tested, the entire program has also been integrated and tested. This approach is part of what is called "Top Down Programming" because you have started at the top (the main program) and worked down to the bottom.

Often a main program is required to load data, or move data around in registers, before calling a subroutine, and to store data returned by a subroutine. If you leave some space between the CALL statements it becomes easy to insert such functions into the main program later. The main program for sensor correction is shown in Figure 6-11. Three NOP's are left between CALL's. This is enough space for three MOV's or one LXI, LDA, STA, LHLD, or SHLD. If more manipulation is needed the three NOP's can be replaced by a CALL, and another subroutine can be created to load, store or manipulate the data as required. We have left three bytes at the beginning for initialization.

SENSOR CORRECTION - MAIN

ADDR		CODE														
8	20	0	00		NOP											For initialize
		1	00		NOP											
		2	00		NOP											
		3	CD		CALL				INPUT							Main Loop
		4	40													
		5	82													
		6	00		NOP											To manipulate
		7	00		NOP											input data
		8	00		NOP											
		9	CD		CALL				SEARCH							DIRECTORY
		A	60													
		B	82													
		C	00		NOP											
		D	00		NOP											
		E	00		NOP											
		F	CD		CALL				TABLE							LOOKUP
8	21	0	80													
		1	82													
		2	00		NOP											To manipulate
		3	00		NOP											table data
		4	00		NOP											
		5	CD		CALL				MULTIPLY							
		6	A0													
		7	82													
		8	00		NOP											To place result
		9	00		NOP											
		A	00		NOP											
		B	CD		CALL				DISPLAY							RESULT
		C	C0													
		D	82													
		E	C3		JMP				8203							Repeat Main Loop
		F	03													
8	22	0	82													
		1														
		2														
		3														
		4														
		5														
		6														
		7														
		8														

Figure 6-11

This is a very straightforward program. Most commonly the main program makes decisions, therefore including comparisons and conditional jumps. These should be designed in from the start, not patched in later. Programs, like machines, must be designed before they are built, or they are likely to fail. The spaces we have left are intended only for data movement, which is not fundamental to the design.

In Figure 6-11 we have arbitrarily placed the subroutines as follows:

```
8240   INPUT
8260   SEARCHDIRECTORY
8280   TABLELOOKUP
82A0   MULTIPLY
82C0   DISPLAYRESULT
```

Thus 32 bytes (20 hex) are allotted to each subroutine. If this is not enough we can easily relocate a subroutine and change the address in the main program.

Load the main program, and enter a RET instruction (C9) at each of the addresses above. Step through to make sure the program operates correctly. Note one of the advantages of "Top Down" programming - with only vague definitions of the program modules we have now established the relationships among them. This will help immensely in specifying the modules.

6.3.3 Input Subroutine

The definition for this subroutine was given as:

> Accept data input from the keyboard.
> Display the data as it is entered.
> On a specified command, change the sensor number.
> Return when a command is entered.

Nothing has been said here about register or memory assignments, and the mention of changing sensor number is vague indeed. A better definition is essential before we can design this module.

We shall switch temporarily from "Top Down" programming to "Bottom Up" programming. When you have no idea of how to accomplish a function, it is often much better to work out some details before proceeding with a design - just as we may experiment with a breadboard electronic circuit, or look in catalogs to see what is available, before specifying and designing hardware.

You do not yet have enough knowledge of the MTS hardware, nor of the 8080 instructions, to write a keyboard input subroutine. There is a built-in subroutine, GETKY, which you can use without understanding how it works just as you can buy and use an integrated circuit. This subroutine is used by the monitor when you key in a program or enter commands such as STEP or RUN. In fact, when you are using the monitor it spends almost all of its time in subroutine GETKY, waiting for you to press a key. The specification is given here:

6.3.3.1 Subroutine GETKY

Function:

Read the keyboard repeatedly until a key is pressed. Wait until the key is released; then return the value of the key and indicate whether it is a command or hex key.

Entry:

```
CD        CALL GETKY
3D
02
```

Inputs:

No data required at entry.

Returns:

The value of the key pressed, with Carry set if hex key; Carry cleared if command.

Registers:

```
(A) = (C) = Key Value
(B) = 00
```

All other registers are preserved. All flags are affected.

Note that this specification is not quite complete. No mention is made of the possibility of several keys being pressed at once, and there are some constraints that you need not worry about. We have not stated the values returned for the command keys; you will determine that by testing the subroutine.

6.3.3.2 Monitor Display Subroutine DBY2

Although you have operated the MTS display directly, by writing to memory locations 83F8 to 83FF, and you could develop your own display subroutine, it will be easier to use another monitor subroutine that displays a byte of data in two digits.

Subroutine DBY2

Function:

Display one byte of data in two specified digits of the MTS display.

Entry:

```
CD      CALL    DBY2
98
02
```

Inputs:

Byte to be displayed in Register A. Display address for low digit in register pair DE.

Outputs:

The byte displayed is duplicated in Registers A and C. The display address is decremented by two, pointing to the memory location below the left digit location.

Registers:

(A) = (C) = byte displayed
(DE) = Entry value of (DE) - 2
(B), (H), (L) preserved
Carry and Zero are cleared

Constraints:

For an effective display the entry value of (DE) must be in the range 83F9-83FF. No test is made on the address; DBY2 will store symbols for two digits at the address in (DE) and the next lower address. Only two memory locations and display digits are affected.

We can test both of these subroutines (GETKY and DBY2) within the context of the sensor correction subroutine INPUT. AT 8240, enter the calls and required input data for these two subroutines, followed by RET. Do this yourself, and then compare your work with Figure 6-12.

TEST GETKY AND DBY2

CODING SHEET

ADDR		CODE													
8024	0	CD		CALL				GETKY							(A) ← Key
	1	3D													
	2	02													
	3	11		LXI				D,	83	FB					Address for
	4	FB													display in digits
	5	83													3 and 4
	6	CD		CALL				DBY2							Display (A)
	7	98													
	8	02													
	9	C9		RET											
	A														
	B														
	C														
	D														
	E														
	F														
8	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														
	9														
	A														
	B														
	C														
	D														
	E														
	F														
8	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure 6-12

6.3.3.3 Testing GETKY and DBY2

These subroutines are guaranteed to work, so press RUN. The display will go blank. Press and release a key; its value will be displayed. With a display address of 83FB, the byte will appear in digits 3 and 4 of the display.

See that the hex keys of 0 - F are displayed as 00 - 0F. Make a list of the values returned by GETKY for commands.

REG	MEM	BRK	CLR	RST	
				STEP	
				RUN	
				ADDR	
				NEXT	

You will find that RST does not return a value from GETKY -- it resets the microcomputer. Electrically, RST is not part of the keyboard input circuit. Instead, it provides a direct input to the microprocessor and its function cannot be changed.

Place a breakpoint at the LXI D instruction, after the call to GETKY.

ADDR 8 2 4 3 BRK 8243 BP.

Enter arbitrary data into the registers:

```
REG    A   A                    8200      A-0A
NEXT   B                        8200      B-0B
NEXT   C                        8200      C-0C
NEXT   D                        8200      D-0D
NEXT   E                        8200      E-0E
NEXT   F                        8200      F-0F
NEXT   8                        8200      H-08
NEXT   9                        8200      L-09
NEXT                            8200      A-0A
RUN
```

The monitor blanks the display. You are now in subroutine GETKY. Press and release key 6. The program stops at the breakpoint.

```
6                          (CY) 8243      A-06
```

Examine the registers and note the Carry and Zero indicators. Confirm that GETKY returns (A) = (C) = key; (B) = 00; that D, E, H and L are preserved; that Carry was set by a hex key.

"Register" F actually displays the content of the five flags of the 8080; the only ones we are interested in are Carry and Zero, which appear in the LED indicators. The others will be described in later chapters.

Press RUN. The key you entered is displayed by DBY2 as before. Press RUN again. This time it is an entry to GETKY for your program. Again execution stops at 8243. Confirm that GETKY has returned (A) = (C) = 14 and (B) = 00. (DE) contains the value entered by your program decremented by 2, or (82F9. 83F9) This was returned by DBY2; GETKY has not disturbed it. Registers H and L are still preserved. Carry is cleared in response to the command key. Zero is also cleared in response to RUN. What key returns Zero set?

Now place a breakpoint at the RET instruction (8249), retaining the breakpoint at 8243. Run the program and press a key. When the program stops at 8243, enter arbitrary data into Registers B and C, and press RUN. At the 8249 breakpoint confirm that Register B has been preserved; (A) has been copied into Register C; and again (DE) = 83F9.

Be sure that you understand these two monitor subroutines before going on. Experiment further with them if you want.

6.3.3.4 Definition of Sensor Correction INPUT Subroutine

Now that we have some tools (the monitor subroutines GETKY and DBY2) we can define the INPUT subroutine for the sensor correction program. We want it to accept hex keys followed by a command, just as the monitor does, assembling two successive keys into a byte. We shall see how to do that in the next section. If some specified command key is entered, we are to "change" sensor number. The original definition was vague about this. What command key causes the change? Exactly what is meant by "change"? Is the user allowed to enter input data for the new sensor before making the change? If not, what is to be done with data entered before the change? Must new data be entered after the change?

You can make your own decisions about these questions. The solution given here is the simplest to program, but other approaches might be more interesting.

For simplicity we will use the following rules:

> Key MEM calls for a change in sensor number. (MEM returns Zero set from GETKY.)
>
> A data byte for the new sensor is to be entered before the change (MEM) command.
>
> If no hex key is entered, the input value returned will be zero.
>
> If only one hex key is entered, it will be taken as the low digit, and the high digit will be zero.

If two hex keys are entered, the earlier will be the high digit; the later will be the low digit.

If more than two hex keys are entered, the oldest will be discarded and the last two will be used to form the input data byte.

The change in sensor number will be to set the next higher allowable sensor number. The changes will be effected by another subroutine, NEXTSENSOR, which is called by INPUT in response to the MEM key. (Note that by defining another subroutine we are spared worrying about its details now. This is "Top Down" design again.)

Now we must also assign registers for data to be returned by INPUT, and decide whether it requires any input data from the main program.

The only input data that INPUT might need would be the sensor number. INPUT itself has no need for this; only SEARCHDIRECTORY and NEXTSENSOR use the sensor number. Let us say that it will be stored in memory, and leave the memory location to be defined later.

INPUT must return the data byte keyed in, and display it. Since GETKY and DBY2, between them, use Registers A, B, C, D and E but preserve H and L, we can only use Register H or L to accumulate the data as it is keyed in. Since NEXTSENSOR will surely need the Accumulator, it is probably easiest to return the data in one of these registers; we shall choose Register L. The specification for INPUT is given below.

Subroutine INPUT

Function:

Accept a byte of data from the keyboard, followed by a command. If the command is MEM, call NEXTSENSOR to set the next legal sensor number. If no hex keys are entered, return 00 for the data byte. Display the data byte in the third and fourth digits of the MTS display.

Entry:

```
CD      CALL    INPUT
40
82
```

Inputs:

None needed for INPUT.

Outputs:

Data byte entered from keyboard.

Registers:

A, B, C, D, E and L are used.
At return (L) = data byte entered.
Register H is preserved.

Constraints:

In response to MEM command calls NEXTSENSOR, which must preserve Registers H and L.

Processing of successive hex keys will be as defined in Section 6.3.3.4.

6.3.3.5 Design of Sensor Correction INPUT Subroutine

With a firm definition and the necessary subroutines we can now work out the program for INPUT. How can we combine two keys into one byte?

When the first hex key is entered, it is considered to be the low digit of the byte. When another hex key is entered, the earlier key becomes the high digit, and the later key the low digit. Recall that in a hexadecimal number the high digit has a value of 10 hex (16 decimal) times the number. That is:

10 = 1 x 10 (hex)

20 = 2 x 10 (hex)

30 = 3 x 10 (hex)

and F0 = F x 10 (hex)

With two non-zero digits, the value is 10 (hex) times the higher numeral, plus the value of the lower numeral.

24 = 2 x 10 (hex) + 4

To convert two digits into a byte, then, we must multiply the older digit by 10 (hex) or 16 (decimal). We could, of course, add the older digit into a product sixteen times, but there is a much easier procedure. Add the digit to itself once to get two times its value. Add that result to itself to get four times the digit value; again for eight times the digit value and once more for sixteen (10 hex) times. Now add in the low digit. Thus with the old digit in L and the new digit in C:

```
MOV  A,L        Old Digit
ADD  A          2 x Old Digit
ADD  A          4 x Old Digit
ADD  A          8 x Old Digit
ADD  A          10(16) x Old Digit
ADD  C          10(16) x Old + New
MOV  L,A        = Data Byte
```

Let us program this into our INPUT subroutine and test it. Start by entering a zero into Register L; call GETKY; test for a command key (Carry clear) and jump to the return if a command is entered. Otherwise do the process above; address 83FB and display the result, and jump back to call GETKY again. Try to program this yourself, then compare your program with Figure 6-13. We have not yet handled the call to NEXTSENSOR; this is covered in Section 6.3.4.

SENSOR CORRECTION - INPUT (NOT COMPLETE)

ADDR		CODE																		
8	24	0	2	E		M	V	I		L	,	0	0			Clear data byte
		1	0	0												
	824	2	C	D		C	A	L	L		G	E	T	K	Y	(A) ← (C) ← Key
		3	3	D												
		4	0	2												
		5	D	2		J	N	C			8	2	5	8		Jump if command
		6	5	8												
		7	8	2												
		8	7	D		M	O	V		A	,	L				Old digit
		9	8	7		A	D	D		A						2 × old digit
		A	8	7		A	D	D		A						4 × old digit
		B	8	7		A	D	D		A						8 × old digit
		C	8	7		A	D	D		A						10 × old digit
		D	8	1		A	D	D		C						10 × old + new
		E	6	F		M	O	V		L	,	A				(L) ← data byte
		F	1	1		L	X	I		D	,	8	3	F	B	(address) display
8	25	0	F	B												
		1	8	3												
		2	C	D		C	A	L	L		D	B	Y	2		Display data byte
		3	9	8												
		4	0	2												
		5	C	3		J	M	P			8	2	4	2		Loop
		6	4	2												
		7	8	2												
		8	C	9		R	E	T								Call to NEXTSENSOR
		9														to be added
		A														
		B														
		C														
		D														
		E														
		F														
3		0														
		1														
		2														
		3														
		4														
		5														
		6														
		7														
		8														

Figure 6-13

The main program of Figure 6-11 and this input subroutine can be run as we did the test of GETKY and DBY2. When you first enter a hex key it is displayed as the low digit, with a zero in the high digit. The next hex key shifts the old digit to the high position and enters the second key at the right. If you enter more hex keys the oldest one is lost. What happened to it? Review the multiplication by 10 hex. Place a breakpoint at the first ADD A (8249 in Figure 6-13) and run the program. Enter one hex key - 7. Program execution stops at the breakpoint.

```
RUN
7                (CY) 8249        87
REG C            (CY) 8249      C-07
REG A            (CY) 8249      A-00
```

Carry is set because a hex key was entered. We are about to multiply 00 x 10 (hex) and add 7. When you press RUN the result is displayed and the program waits for another key. The Carry indicator stays on.

```
RUN              (CY)             07
5                (CY) 8249      A-07
REG C            (CY) 8249      C-05
RUN              (CY)             75
8                (CY) 8249      C-08
REG A            (CY) 8249      A-75
```

The old value is 75 from the first two digits. or binary 0111 0101. Now step through the multiplication.

```
STEP                  8249      A-EA
```

Carry is now off. 2 x 75 = EA with no Carry. This can also be viewed as a left shift of the binary value.

Bit Positions	CY	7 6 5 4 3 2 1 0
Old Value (75)	1	0 1 1 1 0 1 0 1
2 x Old Value (EA)	0	1 1 1 0 1 0 1 0

The old Carry is lost. The high bit (0) has been shifted into Carry, and the other bits have shifted left. Now you can step three more times and see the hex values shown below.

4 x Old Value (D4)	1	1 1 0 1 0 1 0 0
8 x Old Value (A8)	1	1 0 1 0 1 0 0 0
10 x Old Value (50)	1	0 1 0 1 0 0 0 0

All four bits of the oldest key (7) have been shifted out of Register A. The next step will add the new key from (C).

10 x Old + New (58)	0	0 1 0 1 1 0 0 0

This addition clears Carry, so now all four bits of the oldest key are irretrievably lost.

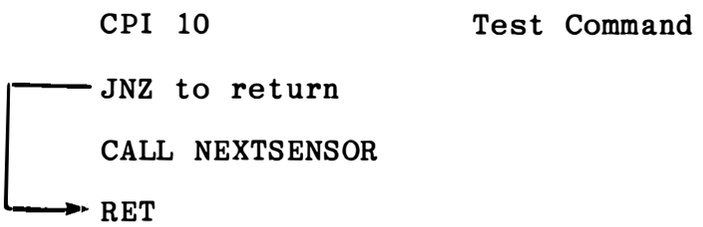
```
RUN                    58
```

The equivalence of a left shift to a multiplication by two is used in binary multiplication, as we shall see in Chapter 7.

When you test the subroutine, note any flaw you see in its operation, and correct the flaw.

6.3.4 Conditional Calls

We have still to handle the call to NEXTSENSOR in response to the MEM command. Subroutine INPUT (Figure 6-13) jumps to 8258 when any command key is pressed. There we must test the command key value and if it is MEM (= 10) then call NEXTSENSOR. Obviously this can be done by:



CALL and RET are special forms of JMP, and the 8080 provides the same conditional variations of CALL and RET as it does for JMP.

C3	JMP	CD	CALL	C9	RET
C2	JNZ	C4	CNZ	C0	RNZ
CA	JZ	CC	CZ	C8	RZ
D2	JNC	D4	CNC	D0	RNC
DA	JC	DC	CC	D8	RC

Four more variations of each, not listed above, also exist.

If the specified flag is set or reset, according to the instruction, execute the Jump, Call or Return. Otherwise continue program execution at the next sequential instruction. Call if Zero is defined in detail.

CC	CZ address	Call if Zero
xx	(low address)	
xx	(high address)	

Read the three byte instruction into Registers I, Z and W. If the Zero flag is set, save the program counter in the stack and move W and Z into the program counter. Otherwise proceed with program execution at the next location after the three byte CZ instruction.

No flags are affected.

6.3.4.1 Completion of Subroutine INPUT

With conditional call instructions we can avoid spending three bytes on a conditional jump instruction. Instead of JNZ, CALL, we shall use:

```
CPI 10
CZ  NEXTSENSOR
RET
```

As before, NEXTSENSOR is called if and only if the command key value is 10 (MEM). If you did not detect the flaw in the operation of INPUT, make this test. Run the program, key in a hex value and NEXT. The number is displayed. Now press NEXT again. According to the specification, pressing a command key with no preceding hex keys must return a value of zero, but the display shows the old value. What actually happened? Review the program and figure it out.

To correct this flaw, display the content of Register L after a command key is pressed. The solution given below in Figure 6-14 uses the same call to DBY2 both for hex keys and the command, thereby saving a little space in the program. This is not important - memory space is cheap. If your version of INPUT takes more than 20 (hex) bytes, relocate SEARCHDIRECTORY to 8270 instead of 8260. It will fit easily in 10 (hex) bytes. Since we have not done anything with it yet, the only change required is in the main program:

```
CD  CALL TABLELOOKUP
70
82
```

Remember to insert:

```
8270    C9      RET
```

6.3.4.2 Subroutine NEXTSENSOR Definition

This subroutine was not included in the original list of subroutines in Section 6.3.1, but we have described it in the course of developing INPUT (Section 6.3.3.4). We must assign a location for storage of the sensor number. We have two possibilities - in a register or in memory. In the sensor correction program of Chapter 4 we reserved Register B for the current sensor number, but here Register B has been affected by GETKY. INPUT preserved Register H, but this will be used in SEARCHDIRECTORY and MULTIPLY. Generally it is better to use memory to store a variable that must be retained indefinitely and changed only occasionally. We have previously said that the directory occupies 8300-8307 and the data tables 8308-837F;

let us now assign memory location 8380 for the current sensor number.

Assign memory locations 82E0 through 82FF to this subroutine.

Subroutine NEXTSENSOR

Function:

Select the next legal sensor number following the current sensor number. If the current sensor number is the highest allowable, set the sensor number equal to 1. Display the new sensor number in the left hand digit.

Entry Address: 82E0

(The call from INPUT will be CZ, but this is not a part of the subroutine specification.)

Inputs:

None required in registers. The following data must be in memory.

8380 Current Sensor Number
8300 Highest Existing Sensor Number

Outputs:

Memory location 8380 is updated to contain the new current sensor number.

Registers:

A, C, D and E are used. B, H and L are preserved.

Constraints:

The sensor number is to be displayed at the left by storing its display symbol at 83F8. The next display position (83F9) must be left blank. Memory location 83F7 must not be affected. (This location is reserved for use by the monitor, whose operation will be affected if you enter data there.)

6.3.4.3 Subroutine NEXTSENSOR Program

The function of this subroutine may be listed in six steps.

Load and increment the sensor number.

Test for a legal number (greater than zero; less than or equal to highest existing sensor number.)

Skip the next step if legal.

Set sensor number to 1.

Store the sensor number.

Display the sensor number.

You should be able to program all of this.

Remember that Registers H and L must be preserved. This does not forbid you to use them, but if you need them you must preserve their data by moving it elsewhere and restoring it to H and L before return.

The display function introduces a problem. We have been using DBY2 for display, but this subroutine displays a byte in two digits. We want to display the sensor number in the left hand digit (83F8) but we are required to leave the second digit (83F9) blank, and we are forbidden from disturbing memory location 83F7. Can you solve this problem?

(One helpful hint: The Read Only Memory contains a table of symbols for the numerals 0 - F, starting at 02B3.)

6.3.4.4 Testing INPUT and NEXTSENSOR

Once again, test the new subroutine using the main program to call it. When you enter hex keys they should be displayed; when you enter MEM a sensor number should be displayed. Your test should include not only checking the displays, but also making sure that the entire specification for each of these subroutines is met.

For this test to be successful you must have stored the highest allowable sensor number at memory location 8300. Try different values there. This may also be a convenient time to enter the directory and data tables. A complete version of INPUT and NEXTSENSOR, and the directory and data tables, are shown in Figures 6-14, 6-15 and 6-16.

SENSOR CORRECTION - SUBROUTINE INPUT

INTEGRATED COMPUTER SYSTEMS MICROCOMPUTER TRAINING SYSTEM CODING SHEET

ADD/R	CODE		Instruction	Comments
8 24 0	2	E	MVI L, 00	
1	0	0		
2	C	D	CALL GETKY	
3	3	D		
4	0	2		
5	4	7	MOV B, A	(B) ← Key
6	7	D	MOV A, L	(A) ← Old Keys)
7	D	2	JNC 8250	Jump if command
8	5	0		
9	8	2		
A	8	7	ADD A	2 x Old Keys)
B	8	7	ADD A	4 x old keys)
C	8	7	ADD A	8 x old keys)
D	8	7	ADD A	10 x old keys)
E	8	0	ADD B	10 x old + new
F	6	F	MOV L, A	(L) ← data input
8 25 0	1	1	LXI D, 83FB	Address digits 3 and 4
1	F	B		
2	8	3		
3	C	D	CALL DBY2	Display key value
4	9	8		
5	0	2		
6	7	8	MOV A, B	Most recent key
7	F	E	CPI 10	Test for key
8	1	0		
9	D	A	JC 8242	Loop if key
A	4	2		
B	8	2		
C	C	C	CZ NEXTSENSOR	
D	E	0		If command = MEM
E	8	2		
F	C	9	RET	
3 0				
1				
2				
3				
4				
5				
6				
7				
8				

Figure 6-14

SENSOR CORRECTION - SUBROUTINE NEXTSENSOR

ADDR		CODE													
8 2E	0	3	A		L	D	A		8	3	8	0			(A) ← Current
	1	8	0												Sensor Number
	2	8	3												
	3	3	C		I	N	R		A						Next Sensor Number
	4	C	A		J	Z				8	2	F	0		If zero set
	5	F	0												Sensor Number 1
	6	8	2												
	7	4	F		M	O	V		C	,	A				
	8	3	A		L	D	A		8	3	0	0			Highest allowable
	9	0	0												sensor number
	A	8	3												
	B	B	9		C	M	P		C						
	C	7	9		M	O	V		A	,	C				(A) ← Next sensor
	D	D	2		J	N	C			8	2	F	2		Jump if Legal
	E	F	2												0 < Sensor < Highest
	F	8	2												
8 2F	0	3	E		M	V	I		A	,	0	1			Set Sensor #1
	1	0	1												
8 2F	2	3	2		S	T	A		8	3	8	0			Store new
	3	8	0												sensor number
	4	8	3												
	5	1	1		L	X	I		D	,	0	2	B	3	Address symbols
	6	B	3												for 8-F
	7	0	2												
	8	8	3		A	D	D		E						Add sensor number
	9	5	F		M	O	V		E	,	A				Address and load
	A	1	A		L	D	A	X		D					symbol for sensor #
	B	3	2		S	T	A		8	3	F	8			Display at left
	C	F	8												
	D	8	3												
	E	C	9		R	E	T								
	F														
3	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure 6-15

SENSOR CORRECTION - DIRECTORY AND DATA

ADDR		CODE	
8 30	0	02	HIGHEST SENSOR NUMBER
	1	08	SENSOR 1 TABLE ADDRESS
	2	16	SENSOR 2 TABLE ADDRESS
	3	00	RESERVED FOR
	4	00	ADDITIONAL SENSORS
	5	00	
	6	00	
	7	00	
	8	88	SENSOR 1 SCALING FACTOR
	9	0B	LINEAR POINT
	A	00	CORRECTED INPUT = 00
	B	03	VALUES = 01
	C	04	= 02
	D	05	= 03
	E	06	= 04
	F	07	= 05
8 31	0	08	= 06
	1	09	= 07
	2	09	= 08
	3	0A	= 09
	4	0B	= 0A
	5	0B	= 0B
	6	C8	SENSOR 2 SCALING FACTOR
	7	07	LINEAR POINT
	8	00	CORRECTED INPUT = 00
	9	02	VALUES = 01
	A	04	= 02
	B	04	= 03
	C	05	= 04
	D	06	= 05
	E	07	= 06
	F	07	= 07
8	0		
	1		
	2		
	3		
	4		
	5		
	6		
	7		
	8		

Figure 6-16

6.3.5 Subroutine DISPLAYRESULT

It is often convenient to develop input and output subroutines for a program at an early stage, because these provide tools for testing other program modules. We now have the input subroutine with its own display, and we have a monitor subroutine that makes it easy to display the result. DBY2 only shows one byte; we want to display two bytes, but that merely involves two calls to DBY2, one for each byte. Remembering that DBY2 preserves the content of Registers H and L suggests that these registers can be used for the two byte number to be displayed.

Subroutine DISPLAYRESULT

Function:

Display two bytes of data in the four right hand digits.

Entry Address:

82C0

Inputs:

(L) = low byte to be displayed
(H) = high byte to be displayed

Outputs:

(Specification of the outputs is left as an exercise for the student. Review the specification of DBY2 in Section 6.3.3.2, and state what each register will contain at return from DISPLAYRESULT.)

To test DISPLAYRESULT, remember that INPUT places the entry data in Register L, and preserves Register H. Before running the program, use the monitor to load arbitrary data into H; this should be displayed every time. The data keyed in through INPUT should appear in digits 7 and 8 as well as digits 3 and 4.

Note that we are able to test each subroutine as we develop it, using the main program and earlier subroutines as testing tools.

Figure 6-17 gives a solution for subroutine DISPLAYRESULT.

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

ADDR		CODE													
8 2C	0	1	1		L	X	I		D	,	8	3	F	F	
	1	F	F												
	2	8	3												
	3	7	D		M	O	V		A	,	L				
	4	C	D		C	A	L	L		D	B	Y	2		
	5	9	8												Returns (DE) =
	6	0	2												83FF - 2 = 83FD
	7	7	C		M	O	V		A	,	H				
	8	C	D		C	A	L	L		D	B	Y	2		
	9	9	8												Returns (A) = (C) = (H)
	A	0	2												(DE) = 83FB
	B	C	9		R	E	T								
	C														
	D														
	E														
	F														
8	0				E	N	T	E	R		W	I	T	H	
	1					(	L	)	=	L	O	W		B	YTE OF RESULT
	2					(	H	)	=	H	I	G	H		BYTE
	3														
	4				R	E	T	U	R	N					
	5					(	B	)	,	(	H	)	,	(	L) PRESERVED
	6					(	A	)	=	(	C	)	=	(	H)
	7					(	D	E	)	=	8	3	F	B	
	8														
	9														
	A														
	B														
	C														
	D														
	E														
	F														
3	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure 6-17

6.3.6 Subroutine SEARCHDIRECTORY

This subroutine is to be used to return the address of the data table for a particular sensor - the one whose sensor number was stored at memory location 8380 by subroutine NEXTSENSOR. With the sensor number, directory and data tables all in a single page of memory (83xx) this subroutine can use single byte indirect addressing. It is further simplified by the assignments in the directory:

```
8301    Table address for sensor 1
8302    Table address for sensor 2
```

The indirect addressing then is merely:

```
(H)  < -   83
(L)  < -   (8380)
(L)  < -   ((HL))
```

This can be coded as:

```
LXI     H,8380          (H)  < -   83
MOV     L,M             (L)  < -   (8380)
MOV     L,M             (L)  < -   ((8380))
```

Remember, however, that at the return from INPUT we have the input data byte in Register L. This is why we provided NOP instructions in the main program - to make space for MOV instructions. Although we could specify that SEARCHDIRECTORY move the content of L to some other register, this is generally undesirable. Keep subroutines as nearly single purpose as possible in order to improve readability of the program and generality of the subroutine.

Subroutine SEARCHDIRECTORY

Function:

Load into register pair HL the address of the data table corresponding to the sensor number.

Entry Address: 8260

Inputs: Sensor number stored at 8380

Outputs: Data Table Address in (HL)

Registers: Only (H) and (L) are used

Constraints:

A directory must be stored in memory at 8301 - 8307. The data tables must also be in page 83xx.

Test SEARCHDIRECTORY using the main program, INPUT, NEXTSENSOR and DISPLAYRESULT. Since TABLELOOKUP and MULTIPLY do nothing yet, the address returned by SEARCHDIRECTORY will be displayed by DISPLAYRESULT. For Sensor Number 1 the address returned by SEARCHDIRECTORY should be 8308; for Sensor 2 it should be 8316.

This subroutine (Figure 6-18) is so short that it could easily be programmed in-line (i.e., in the main program) or it could be included in TABLELOOKUP. In another exercise we shall see reasons for not doing so.

SENSOR CORRECTION - SEARCHDIRECTORY

ADDR	CODE												
8 26 0	21		LXI		H,	8380							
1	80												
2	83												
3	6E		MOV		L,	M							
4	6E		MOV		L,	M							
5	C9		RET										
6													
7													
8													
9													
A													
B													
C													
D													
E													
F													
8 0													
1													
2													
3													
4													
5													
6													
7													
8													
9													
A													
B													
C													
D													
E													
F													
8 0													
1													
2													
3													
4													
5													
6													
7													
8													

Figure 6-18

6.3.7 Program Data Initialization

At this point we can see a need for setting initial values into the program data. In this program the only variable that is retained from one iteration of the main loop to the next is the sensor number. Recall that in Chapter 4 we always tested the sensor number before proceeding with the directory search and table lookup. Now we have delegated the task of testing sensor number to a subroutine that is only called in response to a user command. This implies the possibility of having an illegal sensor number stored when the program starts to run; hence making improper calculations. The risk is not immediately obvious, because we have already exercised subroutine NEXTSENSOR, thereby storing a legal sensor number at memory location 8380. Store an illegal number at that location and run the program already loaded, without pressing MEM. The address displayed will be neither 8308 nor 8316, which are the only proper table addresses. When you press MEM, thereby calling NEXTSENSOR, the table addresses become legal.

In the final program, if we accept data entry while an illegal sensor number is stored, the result will be meaningless. This must be forbidden. Also, of course, we want the sensor number displayed right from the start.

We can ensure that a legal sensor number is set and displayed by calling NEXTSENSOR as an initialization step. At the start of the main program, enter:

```
8200    CD          CALL NEXTSENSOR
8201    E0
8202    82
```

Test this. Either a 1 or 2 should appear at the left. Press MEM to change sensors. The only weakness is that on the first run you cannot predict which will appear. If this matters, an initial value must be stored before calling NEXTSENSOR.

6.3.7.1 Alternate Entry to Subroutine

There is another technique available which must be used with care. Examine the given solution for NEXTSENSOR (Figure 6-15). After incrementing the sensor number and finding it illegal (either 00 or greater than the highest allowable) the program reaches 82F0. The code there is:

```
82F0    MVI     A,01        Set Sensor 1
        STA     8380
        LXI     D,02B3      Address symbols
        ADD     E           Add sensor numer
        MOV     E,A         Address and load
        LDAX    D           Symbol for sensor
        STA     83F8        Display at left
        RET
```

The code above can be used as a subroutine by itself, to set and display Sensor Number 1. The initialization in the main program could be:

```
8200    CD      CALL 82F0
8201    F0
8202    82
```

If your NEXTSENSOR program is similar to Figure 6-15, you can use this procedure successfully. Address 82F0 is then an "Alternate Entry" to subroutine NEXTSENSOR.

Suppose now that a slightly more clever program had been written for NEXTSENSOR:

```
82E0    LXI     D,8380
82E3    LDAX    D
82E4    INR     A
82E5    JZ      82F1
82E8    MOV     C,A
82E9    LDA     8300
82EC    CMP     C
82ED    MOV     A,C
82EE    JNC     82F3
82F1    MVI     A,01
82F3    STAX    D
82F4    LXI     D,02B3
etcetera
```

This program is one byte shorter than the solution of Figure 6-15. If you were to call this at the MVI A,01 instruction, however, it would fail, because the STAX D instruction could store 01 anyplace - in the middle of your program, for instance. This is the danger of alternate entries to subroutines. If used without great care they can be disastrous.

The only safe way to use alternate entries is at the beginning of a subroutine. For instance, the display subroutine we have been using, DBY2, is actually an alternate entry to the monitor subroutine DBYTE, which starts at 0295 with LXI D,83FF. A call to DBYTE displays the byte in (A) in the two right hand digits; the alternate entry DBY2 allows you to select a different pair of display digits. It only bypasses the one instruction that loads a constant into the display address.

6.3.7.2 External Alternate Entry

In the discussion above we referred to address 82F0 as a possible alternate to NEXTSENSOR. The risk of using such an entry comes from the fact that it is inside the subroutine - hence it may be called an "internal alternate entry". We could avoid using an alternate entry by creating a separate initialization subroutine to be called by the main program.

```
        XRA     A                       Set Sensor = 0
        STA     8380
        CALL    NEXTSENSOR              Set Sensor = 1
        RET
```

This procedure is safe, because we are not relying on any specific coding of subroutine NEXTSENSOR. We can modify this to the following:

```
        XRA     A               Set Sensor = 0
        STA     8380
        JMP     NEXTSENSOR      Set Sensor = 1
```

This has an essentially identical effect. When it is called by main, a return address (8203) is placed in the stack. After setting sensor number equal to zero, it jumps to NEXTSENSOR to increment the number. When the RET instruction is encountered at the end of NEXTSENSOR, address 8203 is recovered from the stack so the return is directly to the main program instead of to another RET. This is called an "external alternate entry". We shall use this technique for initialization of sensor number.

Figure 6-19 shows the revised main program and subroutine INITIALIZE. Test that we now always start with Sensor Number 1 displayed, and that no improper table address occurs.

SENSOR CORRECTION - MAIN AND INITIALIZE

ADDR		CODE															
8200	0	CD			C	A	L	L		I	N	I	T	I	A	L	IZE
	1	21															
	2	82															
	3	CD			C	A	L	L		I	N	P	U	T			
	4	40															(L)← Input Data
	5	82															
	6	00			N	O	P										
	7	00			N	O	P										
	8	7D			M	O	V		A	,	L						(A)← Input Data
	9	CD			C	A	L	L		S	E	A	R	C	H	D	IRECTORY
	A	60															(HL)← Table Address
	B	82															
	C	00			N	O	P										
	D	00			N	O	P										
	E	00			N	O	P										
	F	CD			C	A	L	L		T	A	B	L	E	L	O	OKUP
8210	0	80															(E)← Scaling Factor
	1	82															(A)← Adjusted Input
	2	C0			N	O	P										
	3	00			N	O	P										
	4	00			N	O	P										
	5	CD			C	A	L	L		M	U	L	T	I	P	L	Y
	6	A0															(HL)← (E) * (A)
	7	82															
	8	00			N	O	P										
	9	00			N	O	P										
	A	00			N	O	P										
	B	CD			C	A	L	L		D	I	S	P	L	A	Y	RESULT
	C	C0															Display (HL)
	D	82															at Right
	E	C3			J	M	P			8	2	0	3				
	F	03															
8220	0	82															
8221	1	AF			X	R	A		A								INITIALIZE
	2	32			S	T	A		8	3	8	0					Set Sensor) #0
	3	80															
	4	83															
	5	C3			J	M	P			N	E	X	T	S	E	N	SOR
	6	E0															Set Sensor) #1
	7	82															
	8																

Figure 6-19

6.3.8 Subroutine TABLELOOKUP

This subroutine is specified to load data from the table whose address is supplied by SEARCHDIRECTORY. The scaling factor is loaded from the first entry in the table and the input data (in Register A) is compared with the linear point, the second item in the table.

```
MOV     E,M
INX     H
CMP     M
```

If the input data byte is equal to or greater than the linear point Carry is cleared by the comparison and no adjustment is necessary. Here we can use the conditional return, RNC, since the task of the subroutine is finished.

> Return if Not Carry
> Hex Code: D0
> Mnemonic: RNC
>
> If the Carry flag is clear, recover a return address from the stack and jump to that address.
>
> If Carry is set, continue program execution at the next sequential instruction, leaving the return address in the stack.

If the input value is less than the linear point (Carry is set) we must obtain an adjusted value from the table. In Chapter 4 we did this by:

```
        INX     H               Address table for 00 input
        ADD     L
        MOV     L,A
        MOV     A,M
```

Since Carry is set (else we would have returned) we can use a trick here: instead of INX H, ADD L we use ADC L. Adding in the Carry has the same effect, of adding table address + 1 plus input value.

Subroutine TABLELOOKUP

Entry Address: 8280

Entry Data: (A) = Measured Input

Return Data: (E) = Scaling Factor

If the input is greater than or equal to the linear point:

(A) preserved
(HL) addressing linear point

(A) = adjusted input value
(HL) addressing table location for the input value

Registers:

A, E, H and L Used
B, C, and D Preserved

To test this program we can again use our existing main program and subroutines. (Remember that MAIN must include MOV A,L before the call to SEARCHDIRECTORY.) Since we have not yet programmed the subroutine MULTIPLY, (HL) contains the address in the table, and this will be displayed. For a data input less than the linear point we should see the table address corresponding to the sensor number and input value. For greater inputs we should see the address of the linear point. Test your program in this mode, comparing inputs and results with the data tables of Figure 6-16.

6.3.9 Stubs for Subroutines

When we first entered the main program into the computer we placed a RET instruction at each subroutine location. Only one of these remains now (at MULTIPLY); all the others have been replaced by subroutines. Such a RET instruction is called a "stub" - it is a very short subroutine. Sometimes it is useful or necessary to have a stub that performs some reasonable substitute for the program module. For instance, if we did not yet have the data tables available, TABLELOOKUP could enter a fixed scaling factor into Register E, and do no adjustment on the input data. We could even think of our present version of TABLELOOKUP as a stub for a much more sophisticated program that might eventually provide for interpolation or some complex calculation.

The usual purpose of a stub is to permit other program modules to be tested in the absence of a module which has not yet been written. Somtimes a stub is substituted for a program module (even though that module may have been finished and tested) in order to make the test of a new module easier. Let us replace the existing stub of MULTIPLY (which has been simply RET) with a new stub which will cause the adjusted input and the scaling factor to be displayed.

```
82A0    67      MOV     H,A         (H)  < - Input
82A1    6B      MOV     L,E         (L)  < - Scaling Factor
82A2    C9      RET
```

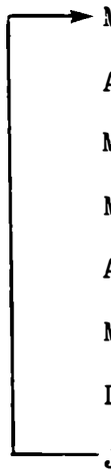
Now the program will display the results of TABLELOOKUP. This might discover some error in the data tables that otherwise would be concealed by the multiplication. Now for Sensor 1 we should always see the scaling factory (88) in the right hand digits, and the adjusted input in digits 5 and 6. Figure 6-20 shows TABLELOOKUP and this stub for MULTIPLY.

ADDR		CODE													
8 28	0	5	E		M	O	V		E	,	M				
	1	2	3		I	N	X		H						
	2	B	E		C	M	P		M						
	3	D	0		R	N	C								
	4	8	D		A	D	C		L						
	5	6	F		M	O	V		L	,	A				
	6	7	E		M	O	V		A	,	M				
	7	C	9		R	E	T								
	8														
	9														
	A														
	B														
	C														
	D														
	E														
	F														
8 2A	0	6	7		M	O	V		H	,	A				STUB for MULTIPLY
	1	6	B		M	O	V		L	,	E				(see text)
	2	C	9		R	E	T								
	3														
	4														
	5														
	6														
	7														
	8														
	9														
	A														
	B														
	C														
	D														
	E														
	F														
3	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure 6-20

6.3.10 Register Pair Addition

In Chapter 4 we used a repetitive double precision addition to perform multiplication.



```
        LXI     H,0000          Clear product
MOV     A,L                     Add multiplicand (C)
ADD     C                       into product (HL)
MOV     L,A
MOV     A,H
ACI     00
MOV     H,A
DCR     E                       Decrement multiplier
JNZ
```

The 8080 provides instructions that perform the double precision addition in a single step.

6.3.10.1 Double Precision Add - DAD

DAD rp Add the 16 bit content of register pair rp to the content of register pair HL, placing the result in HL.

(HL) < - (HL) + (rp)

If the result is greater than FFFF, set Carry. Otherwise clear Carry. No other flags are affected.

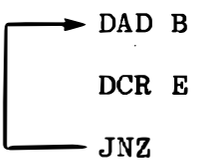
The hex codes for the DAD intructions are:

```
09      DAD B   (HL) < -  (HL) + (BC)
19      DAD D   (HL) < -  (HL) + (DE)
29      DAD H   (HL) < -  (HL) + (HL)
```

6.3.10.2 Subroutine MULTIPLY

If our sensor data tables were more extensive, and might cross page boundaries, we would have used a DAD instruction in TABLELOOKUP. Here we shall use it in MULTIPLY.

We must still clear (HL) for the product. To use DAD we must place the multiplicand in the low byte of a register pair, and clear the high byte of that pair. Then to duplicate the multiplication of Chapter 4 we would do:



As before, multiplication by zero would be equivalent to multiplication by 100 hex. Although that was convenient in Chapter 4 we will here use a technique that gives the correct result of 0000 if the scaling factor is 00. We can readily test a register content for zero by:

```
1C      INR E
1D      DCR E
```

The register content is restored and the Zero flag is set or reset according to the content. Now we can use a conditional return:

```
C8      RZ      Return if Multiplier Zero
```

If the multiplier was zero this returns before we have added the multiplicand the first time. Otherwise, execute DAD B; then jump back to DCR E, RZ.

Write, and load this final subroutine. Once again, the main program provides a test, described in Section 6.3.10.3.

SENSOR CORRECTION - MULTIPLY

ADDR		CODE													
8	2A	0	21		LXI		H,	0000							Clear Product
		1	00												
		2	00												
		3	4F		MOV		C,	A							(C) ← Multiplicand
		4	45		MOV		B,	H							(B) ← 00
		5	1C		INR		E								
	82A	6	1D		DCR		E								
		7	C8		RZ										
		8	09		DAD		B								
		9	C3		JMP			82A6							
		A	A6												
		B	82												
		C													
		D													
		E													
		F													
8		0													
		1													
		2													
		3													
		4													
		5													
		6													
		7													
		8													
		9													
		A													
		B													
		C													
		D													
		E													
		F													
8		0													
		1													
		2													
		3													
		4													
		5													
		6													
		7													
		8													

Figure 6-21

6.3.10.3 Final Test

With subroutine MULTIPLY written and entered we are ready for a final test. We shall use the same data that were used in Chapter 4.

Sensor	Input	Two Byte Product (HL)
1	00	0000
1	01	0198
1	04	0330
1	07	04C8
1	08	04C8
1	09	0550
1	0A	05D8
1	0B	05D8
1	0C	0660
1	80	4400
2	03	0320
2	06	0578
2	07	0578
2	08	0640
2	09	0708
2	0C	0960
2	80	6400

This test does not fully prove the MULTIPLY subroutine, since only two different multipliers (88 and C8) have been used. This is one

case where we should properly write a "driver" program to test the subroutine. Such a program would test MULTIPLY for all possible multipliers and multiplicands. The exercise of Section 6.7 involves writing a test driver for MULTIPLY.

6.3.11 Program Integration

Historically, every program module was written and tested separately, using "driver" programs to supply simulated input data and test the results. Then a giant task called "program integration" would bring all of the modules together, and find out why they did not work. Top down programming has brought us to a finished product when the last subroutine was written and tested. Program integration consists of listing the program in one place. (This listing appears at the beginning of Section 6.5, where some additional exercises are suggested.)

No special test programs to try out the modules were written - the main program tested each module. The only exception was the special stub for MULTIPLY, used for testing TABLELOOKUP. We also indicated the need to test MULTIPLY with a "driver" program.

This does not imply that final testing is not needed, but the purpose of the test should be to prove that the program handles all conditions - not to debug modules and their interfaces. Of course it is not this easy with a big program, but that is where top down programming really pays off.

6.4 REVIEW AND SELF TEST

This chapter has introduced the very important concepts of program modules and subroutines, and "top down" programming. We have used a main program with subroutines, and used stubs for subroutines that had not yet been written.

Section 6.2 described how the stack pointer works with the CALL and RET instructions, and we used the monitor to examine the stack pointer and the contents of the stack. We have also used monitor subroutines for input and output. Section 6.10 defines a number of additional monitor subroutines that you will use in this course; others appear in Appendix A, Volume II.

Review the new instructions that have been introduced in this chapter. You have already used six of these fourteen.

Double Precision Add

09	DAD B	(HL) < - (HL) + (BC)
19	DAD D	(HL) < - (HL) + (DE)
29	DAD H	(HL) < - (HL) + (HL)

These instructions set or reset Carry but do not affect Zero or any other flag.

Indirect Jump

E9	PCHL	(PC) < - (HL) Jump to the location whose address is in (HL)

Call and Return Instructions

CD	CALL	address	Unconditional Call
C4	CNZ	address	Call if Not Zero
CC	CZ	address	Call if Zero
D4	CNC	address	Call if Not Carry
DC	CC	address	Call if Carry

Calls are three byte instructions. The returns are single byte instructions.

C9	RET	Unconditional Return
C0	RNZ	Return if Not Zero
C8	RZ	Return if Zero
D0	RNC	Return if Not Carry
D8	RC	Return if Carry

Refresh your memory by answering the following questions.

1) What instructions are used to enter a subroutine? What supplies the subroutine address?

2) What instructions exit from a subroutine? What supplies the return address?

3) What is an internal alternate entry to a subroutine? Why is it undesirable? How can you avoid the difficulties?

4) What happens to the stack pointer when a CALL is executed? What datum is found in the memory location addressed by the stack pointer after the CALL?

5) What happens to the stack pointer when a RET is executed?

6) What happens to the stack pointer if the instruction RNZ is encountered when the Zero flag is set? What happens to the Zero flag?

7) Show the content of the three register pairs and the Carry and Zero flags after each instruction in the following program segment.

	CY	Z	BC	DE	HL
Starting Data	1	0	0654	83F8	6400
LXI H,2000					
MOV C,L					
MOV B,H					
LXI D,4000					
DAD B					
DAD D					
DAD H					

Answers to Self Test, Section 6.4

1) CALL and conditional calls enter a subroutine. Bytes 2 and 3 of the intruction supply the address.

2) RET and conditional returns exit from a subroutine. The return address is taken from the stack.

3) An internal alternate entry is a location within the body of a subroutine that may be called from another program module. It requires that the coding of the subroutine be designed to permit the alternate entry to a specific location. An external alternate entry avoids this requirement because it reaches the normal starting point of the subroutine.

4) A CALL instruction causes the stack pointer to be decremented twice. The high byte of the return address is stored after the first decrement; then the low byte is stored after the second decrement, so the stack pointer addresses the low byte of the return address.

5) A RET instruction recovers the return address from the stack, and in the process the stack pointer is incremented twice.

6) RNZ is not executed if the Zero flag is set. Therefore the stack and stack pointer are not changed. Call and return instructions do not affect any flags.

(The answers to question 7 are on the next page)

7) Show the content of the three register pairs and the Carry and Zero flags after each instruction in the following program segment.

	CY	Z	BC	DE	HL
Starting Data	1	0	0654	83F8	6400
LXI H,2000	1	0	0654	83F8	2000
MOV C,L	1	0	0600	83F8	2000
MOV B,H	1	0	2000	83F8	2000
LXI D,4000	1	0	2000	4000	2000
DAD B	0	0	2000	4000	4000
DAD D	0	0	2000	4000	8000
DAD H	1	0	2000	4000	0000

The first two DAD's clear carry. The final DAD H adds 8000 + 8000, giving a carry. Even though the result is 0000 the Zero flag is not affected.

6.5 ADDITIONAL EXERCISES

The following exercises will give you added experience in programming, but more importantly, in specifying subroutines. All of these involve changes to the sensor correction exercise, whose given solution is repeated here for convenience. Read the descriptions of all four changes. Then write new specificatons for INPUT and NEXTSENSOR. Revise and test the program after <u>each</u> change. Note how easy this is with a main program and subroutines.

INTEGRATED COMPUTER SYSTEMS MICROCOMPUTER TRAINING SYSTEM CODING SHEET

ADDR		CODE														
820	0	CD		CALL					INITIALIZE							
	1	21														
	2	82														
	3	CD		CALL					INPUT							
	4	40														(L)←Input Data
	5	82														
	6	00		NOP												
	7	00		NOP												
	8	7D		MOV			A,	L								(A)←Input Data
	9	CD		CALL					SEARCHDIRECTORY							
	A	60														(HL)←Table Address
	B	82														
	C	00		NOP												
	D	00		NOP												
	E	00		NOP												
	F	CD		CALL					TABLE							LOOKUP
821	0	80														(E)←Scaling Factor
	1	82														(A)←Adjusted Input
	2	00		NOP												
	3	00		NOP												
	4	00		NOP												
	5	CD		CALL					MULTIPLY							
	6	A0														(HL)←(E)*(A)
	7	82														
	8	00		NOP												
	9	00		NOP												
	A	00		NOP												
	B	CD		CALL					DISPLAY RESULT							
	C	C0														Display (HL)
	D	82														at Right
	E	C3		JMP					8203							
	F	03														
822	0	82														
822	1	AF		XRA			A									INITIALIZE
	2	32		STA			8380									Set Sensor #0
	3	80														
	4	83														
	5	C3		JMP					NEXTSENSOR							
	6	E0														Set Sensor #1
	7	82														
	8															

Figure 6-22a

SENSOR CORRECTION - SUBROUTINE INPUT

ADDR	CODE		LABEL	OPCODE	OPERAND	COMMENTS
8 24 0	2	E		MVI	L, 00	
1	0	0				
2	C	D		CALL	GETKY	
3	3	D				
4	0	2				
5	4	7		MOV	B, A	(B) ← Key
6	7	D		MOV	A, L	(A) ← Old Key
7	D	2		JNC	8250	Jump if command
8	5	0				
9	8	2				
A	8	7		ADD	A	2 x Old Key
B	8	7		ADD	A	4 x old key
C	8	7		ADD	A	8 x old key
D	8	7		ADD	A	10 x old key
E	8	0		ADD	B	10 x old + new
F	6	F		MOV	L, A	(L) ← data input
8 25 0	1	1		LXI	D, 83FB	Address digits
1	F	B				3 and 4
2	8	3				
3	C	D		CALL	DBY2	Display key value
4	9	8				
5	0	2				
6	7	8		MOV	A, B	Most recent key
7	F	E		CPI	10	Test for key
8	1	0				
9	D	A		JC	8242	Loop if key
A	4	2				
B	8	2				
C	C	C		CZ	NEXTSENSOR	
D	E	0				If command - MEM
E	8	2				
F	C	9		RET		
3 0						
1						
2						
3						
4						
5						
6						
7						
8						

Figure 6-22b

ADDR		CODE													
8	26	0	21			LXI			H,		8	3	8	0	
		1	80												
		2	83												
		3	6E			MOV			L,		M				
		4	6E			MOV			L,		M				
		5	C9			RET									
		6													
		7													
		8													
		9													
		A													
		B													
		C													
		D													
		E													
		F													
8		0													
		1													
		2													
		3													
		4													
		5													
		6													
		7													
		8													
		9													
		A													
		B													
		C													
		D													
		E													
		F													
3		0													
		1													
		2													
		3													
		4													
		5													
		6													
		7													
		8													

Figure 6-22c

ADDR		CODE												
8	28	0	5	E		M	O	V		E	,	M		
		1	2	3		I	N	X		H				
		2	B	E		C	M	P		M				
		3	D	0		R	N	C						
		4	8	D		A	D	C		L				
		5	6	F		M	O	V		L	,	A		
		6	7	E		M	O	V		A	,	M		
		7	C	9		R	E	T						
		8												
		9												
		A												
		B												
		C												
		D												
		E												
		F												
8		0												
		1												
		2												
		3												
		4												
		5												
		6												
		7												
		8												
		9												
		A												
		B												
		C												
		D												
		E												
		F												
3		0												
		1												
		2												
		3												
		4												
		5												
		6												
		7												
		8												

Figure 6-22d

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

ADDR	CODE													
8 2A 0	2	1		L	X	I		H,		0	0	0	0	Clear Product
1	0	0												
2	0	0												
3	4	F		M	O	V		C,		A				(C) ← Multiplicand
4	4	5		M	O	V		B,		H				(B) ← 00
5	1	C		I	N	R		E						
82A 6	1	D		D	C	R		E						
7	C	8		R	Z									
8	0	9		D	A	D		B						
9	C	3		J	M	P				8	2	A	6	
A	A	6												
B	8	2												
C														
D														
E														
F														
8 0														
1														
2														
3														
4														
5														
6														
7														
8														
9														
A														
B														
C														
D														
E														
F														
3 0														
1														
2														
3														
4														
5														
6														
7														
8														

Figure 6-22e

SENSOR CORRECTION - SUBR DISPLAY RESULT

ADDR		CODE													
8 2C	0	1	1		L	X	I		D	,	8	3	F	F	
	1	F	F												
	2	8	3												
	3	7	D		M	O	V		A	,	L				
	4	C	D		C	A	L	L		D	B	Y	2		
	5	9	8												Returns (DE) =
	6	0	2												83FF - 2 = 83FD
	7	7	C		M	O	V		A	,	H				
	8	C	D		C	A	L	L		D	B	Y	2		
	9	9	8												Returns (A) = (C) = (H)
	A	0	2												(DE) = 83FB
	B	C	9		R	E	T								
	C														
	D														
	E														
	F														
8	0				E	N	T	E	R		W	I	T	H	
	1					(	L	)	=	L	O	W		B	YTE OF RESULT
	2					(	H	)	=	H	I	G	H		BYTE
	3														
	4				R	E	T	U	R	N					
	5					(	B	)	,	(	H	)	,	(	L) PRESERVED
	6					(	A	)	=	(	C	)	=	(	H)
	7					(	D	E	)	=	8	3	F	D	
	8														
	9														
	A														
	B														
	C														
	D														
	E														
	F														
3	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure 6-22f

SENSOR CORRECTION - SUBROUTINE NEXTSENSOR

CODING SHEET

ADDR		CODE														
8 2E	0	3A			L	D	A		8	3	8	0				(A) ← Current
	1	80														Sensor Number
	2	83														
	3	3C			I	N	R		A							Next Sensor Number
	4	CA			J	Z				8	2	F	0			If zero set
	5	F0														Sensor Number 1
	6	82														
	7	4F			M	O	V		C	,	A					
	8	3A			L	D	A		8	3	0	0				Highest allowable
	9	00														sensor number
	A	83														
	B	B9			C	M	P		C							
	C	79			M	O	V		A	,	C					(A) ← Next sensor
	D	D2			J	N	C			8	2	F	2			Jump if legal
	E	F2														0 < Sensor < Highest
	F	82														
82F	0	3E			M	V	I		A	,	0	1				Set Sensor #1
	1	01														
82F	2	32			S	T	A		8	3	8	0				Store new
	3	80														sensor number
	4	83														
	5	11			L	X	I		D	,	0	2	B	3		Address symbols
	6	B3														for 8-F
	7	02														
	8	83			A	D	D		E							Add sensor number
	9	5F			M	O	V		E	,	A					Address and load
	A	1A			L	D	A	X		D						symbol for sensor #
	B	32			S	T	A		8	3	F	8				Display at left
	C	F8														
	D	83														
	E	C9			R	E	T									
	F															
3	0															
	1															
	2															
	3															
	4															
	5															
	6															
	7															
	8															

Figure 6-22g

SENSOR CORRECTION - DIRECTORY AND DATA

ADDR	CODE	
8 30 0	02	HIGHEST SENSOR NUMBER
1	08	SENSOR 1 TABLE ADDRESS
2	16	SENSOR 2 TABLE ADDRESS
3	00	RESERVED FOR
4	00	ADDITIONAL SENSORS
5	00	
6	00	
7	00	
8	88	SENSOR 1 SCALING FACTOR
9	0B	LINEAR POINT
A	00	CORRECTED } INPUT = 00
B	03	VALUES } = 01
C	04	= 02
D	05	= 03
E	06	= 04
F	07	= 05
8 31 0	08	= 06
1	09	= 07
2	09	= 08
3	0A	= 09
4	0B	= 0A
5	0B	= 0B
6	C8	SENSOR 2 SCALING FACTOR
7	07	LINEAR POINT
8	00	CORRECTED } INPUT = 00
9	02	VALUES } = 01
A	04	= 02
B	04	= 03
C	05	= 04
D	06	= 05
E	07	= 06
F	07	= 07
3 0		
1		
2		
3		
4		
5		
6		
7		
8		

Figure 6-22h

6.5.1 Clear Result Display

While a new data input is being entered, the old result still appears at the right. During this time the display is showing misleading data - an input at the left with a result at the right that does not correspond to the input being displayed. Revise the specification of INPUT to require that the right hand display be blanked as soon as a key is entered.

6.5.2 Store and Recover Table Address

The sensor correction main program calls subroutine SEARCHDIRECTORY every time we receive new input data, even though the address returned is always the same unless NEXTSENSOR has been called by INPUT. It would be more efficient to combine the two functions. Revise NEXTSENSOR to call SEARCHDIRECTORY; and require SEARCHDIRECTORY to store the sensor table address in memory. In MAIN, simply load the table address from memory.

Alternately, require that INPUT and NEXTSENSOR return Zero set if a MEM command has been entered; Not Zero for other commands. Then have MAIN call SEARCHDIRECTORY only after a MEM command.

Very often it is useful to have a subroutine preserve or restore the flags, especially if the subroutine is expected to be called conditionally. In this case NEXTSENSOR could set Zero (by XRA A or CMP A); then the above requirement would be met.

6.5.3 Two Byte Table Addresses

Revise the directory to include two byte addresses for the data tables. Since each entry will now require two bytes we cannot do the simplified indirect addressing previously used in SEARCHDIRECTORY. This was:

```
LXI, H, 8380          Address Sensor Number
MOV L,M               Address Directory
MOV L,M               Address Table
RET
```

To obtain a two byte address from the sensor number, you must double the sensor number and add it to a fixed value to generate the correct address. Be careful about selecting the fixed value.

6.5.4 Empty Sensor Numbers

The existing data table and directory include only Sensor Numbers 1 and 2. The program allows for higher sensor numbers, but there is an assumption that no gaps exist in the sequence. If the sensor number were greater than zero and less than or equal to the highest allowable, then it is legal, and the directory must have an entry for it.

Remove that constraint by testing for the existence of a valid directory entry as part of the new NEXTSENSOR subroutine. If a sensor does not exist, its directory entry should be 0000. Make sensor 1 non-existent and use its data table for sensor 3.

6.6 USING THE STACK FOR DATA

The stack can provide temporary storage of data as well as storage of return addresses. You have probably seen a spring loaded stack of dishes in a restaurant. The busboy puts clean dishes on top and their weight pushes them down. When one is taken from the top, the spring pops the next one up. The microprocessor has PUSH and POP instructions to place data into the stack, and recover it. Since the stack exists mainly to hold addresses, the data are entered and recovered two bytes at a time, from and to register pairs:

C5	PUSH B	Push data into the stack from
D5	PUSH D	register pair B, D or H
E5	PUSH H	
C1	POP B	Pop data into register pair B, D
D1	POP D	or H from the stack.
E1	POP H	

Suppose that a program needs to call MULTIPLY and DISPLAYRESULT but also needs to retain other data in HL. Since each of the registers is used in at least one of these subroutines, we must save the content of HL in memory. We could do this with SHLD and LHLD, but at the expense of three bytes for each instruction and two bytes in data memory at least partially dedicated to this purpose. PUSH H before the call to MULTIPLY and POP H after return from DISPLAYRESULT will save and recover the data. The content of any of the three register pairs can be saved in this manner.

6.6.1 Testing Stack Usage

Enter this program in order to observe the operations.

```
8200    01      LXI     B,0B0C          Load registers
8201    0C                              with easily
8202    0B                              recognized data
8203    11      LXI     D,0D0E
8204    0E
8205    0D
8206    21      LXI     H,0809
8207    09
8208    08
8209    E5      PUSH    H               Save HL
820A    D5      PUSH    D               Save DE
820B    C5      PUSH    B               Save BC
820C    CD      CALL    8215
820D    15
820E    82
820F    C1      POP     B               Restore BC
8210    D1      POP     D               Restore DE
8211    E1      POP     H               Restore HL
8212    C3      JMP     8209
8213    09
8214    82
8215    04      INR     B               Subroutine
8216    0C      INR     C
8217    14      INR     D
8218    1C      INR     E
8219    24      INR     H
821A    2C      INR     L
821B    C9      RET
```

Note that this program pushes the register pairs in the sequence H, D, B and pops them in reverse sequence. The last bytes pushed are the first bytes popped. We shall see this in operation. Step through the first three instructions and examine the registers.

```
RST     STEP    STEP    STEP            8209      E5
REG B                                   8209    B-0B
NEXT                                    8209    C-0C
NEXT                                    8209    D-0D
NEXT                                    8209    E-0E
```

```
NEXT     (Ignore F)                    8209     F-??
NEXT                                   8209     H-08
NEXT                                   8209     L-09
```

Examine the stack pointer.

```
ADDR     1/P     MEM                   83E0     SP.??
```

Now we shall execute PUSH H

```
ADDR                                   8209       E5
STEP                                   820A       D5
ADDR     1/P       MEM                 83DE     SP.09
NEXT                                   83DF       08
```

The contents of pair HL have been pushed into the stack. The stack pointer has been decremented by 2, and points to the location where the low byte (from L) has been stored. The next higher memory location contains the high byte (from H).

Execute the next two push intructions.

```
ADDR                                   820A       D5
STEP                                   820B       C5
STEP                                   820C       CD
ADDR     1/P       MEM                 83DA     SP.0C
NEXT                                   83DB       0B
NEXT                                   83DC       0E
NEXT                                   83DD       0D
NEXT                                   83DE       09
NEXT                                   83DF       08
```

The stack contains the six bytes we have saved. The top of the stack (the most recent two bytes stored) contains the data from register pair B, the last one pushed.

```
        ADDR    2/T     MEM                     0B0C    ST.??
```

The next instruction is the call to 8215.

```
        ADDR                                    820C      CD
        STEP                                    8215      04
        ADDR    1/P     MEM                     83D8    SP.0F
```

The stack pointer has been decremented two more times.
The stack top now contains the return address.

```
        ADDR    2/T     MEM                     820F    ST.C1
```

The registers have not been altered by any of these instructions. Step through the subroutine, which increments each of the six registers. Review the registers again to check that we now have:

(B) = 0C (C) = 0D (D) = 0E (E) = 0F (H) = 09 (L) = 0A

The stack still contains the original data.

Now execute the return and three POP instructions. When you reach 8212 check that the six registers have been restored. Also check the stack pointer.

```
        ADDR    1/P     MEM                     82E0    SP.??
```

The stack pointer is back to its original position, and the entire

stack is available for other uses.

Transfer notation for PUSH and POP refers to the "Stack Top" as the source or destination. This means two bytes in the stack. For example:

```
PUSH  B        (ST) <- (BC)
               (SP) <- (SP) - 2

POP   H        (HL) <- (ST)
               (SP) <- (SP) + 2
```

6.6.2 Using the Stack Inside a Subroutine

It is perfectly legitimate to use the stack for data outside of a subroutine, as we have just done, and also inside a subroutine. Replace the subroutine above with:

```
8215  C5        PUSH  B         (ST)          (BC)
8216  01        LXI   B,0000    (BC)          0000
8217  00
8218  00
8219  C1        POP   B         (BC)          (ST)
821A  C9        RET
```

Now step through the program again until you reach 8219. (Do not use a breakpoint.) Examine the stack. The stack pointer now contains 83D6, where (C) has been stored again. The register contents can be saved and restored by PUSH and POP either outside or inside the subroutine. It is crucial, however, that these not be mixed. The PUSH and POP instructions must be balanced in each program module.

What would happen if you executed a POP B inside the subroutine, without a preceding PUSH? The two bytes at the top of the stack would be copied into register pair B, and the stack pointer would be incremented twice. Now BC contains the return address, and a RET instruction will jump to the location found in the next two bytes of the stack - 0B0C in the program above. Test this by deleting the PUSH B at 8215 and stepping through the program.

6.6.3 Processor Status Word (PSW)

The content of the accumulator and flags can also be saved in the stack. For PUSH and POP only, Register A and the flags are treated as a register pair, called the "Processor Status Word".

```
F5      PUSH    PSW
F1      POP     PSW
```

These instructions save and restore the content of the accumulator and all five 8080 flags (Zero, Carry, and three others not yet described.)

Recall in the sensor correction subroutine INPUT (Figure 6-22b) we copy an input key to Register B, and after displaying the hex value we test (B) to determine whether a hex key, or command MEM, or some other command was entered.

```
CALL GETKY
MOV  B, A
     .
     .
     .
MOV  A,B
CPI  10
JC   8242
CZ   NEXTSENSOR
RET
```

The CPI 10 instruction was in fact done in GETKY, which returns Carry for hex keys; Not Carry, Zero for MEM, Not Carry, Not Zero for the other commands. Then the above sequence could have been:

```
CALL GETKY
PUSH  PSW
    .
    .
    .
POP   PSW
JC    8242
CZ    NEXTSENSOR
RET
```

It is fairly common to need the results of a test after some intervening operations that affect the flags; PUSH PSW and POP PSW provide this facility. In these instructions the flags are treated as the low byte of the pair (stored in the lower memory location of the stack) and the accumulator is treated as the high byte. PUSH PSW and POP PSW are the only instructions that treat the flags as a register, or as part of a register pair; there is no LXI PSW instruction.

6.6.4 Exchange Instructions

With two exceptions the data movement instructions of the 8080 are all one-way. MOV A,C copies into A the content of C; Register C is not affected. SHLD stores the contents of H and L; the registers are not affected. In each case the old content of the destination is lost.

The two exchange intructions are the exceptions.

6.6.4.1 Exchange (HL) with (DE)

EB XCHG (HL) < - > (DE)

The content of Register E is exchanged with the content of Register L.

The content of Register D is exchanged with the content of Register H.

Flags are not affected.

Here all four data bytes are preserved, but in different registers. This instruction is especially useful when two different memory locations are successively addressed, or when some following operation must use HL. It can also sometimes be used merely as a single instruction to substitute for MOV E,L; MOV D,H. For instance, to load four bytes from memory:

```
LHLD  8300
XCHG
LHLD  8302
```

There is no corresponding instruction involving pair B; the other 8080 exchange involves the stack.

6.6.4.2 Exchange HL with Stack Top

E3 XTHL (HL) < - > (ST)

The operation involves the stack pointer and the temporary Registers W and Z. The data byte addressed by the stack pointer is copied into Z and the stack pointer is incremented; the data byte now addressed by the stack pointer is copied into W. Register H is copied into the stack and the stack pointer is decremented; Register L is copied into the location so addressed. W and Z are copied into H and L. There is no net effect on the stack pointer; it ends up where it started.

The process could be shown as:

(WZ) < - (ST) (like a POP)
(ST) < - (HL) (like a PUSH)
(HL) < - (WZ) (like an LXI)

This powerful one byte instruction effectively adds one more register pair to the 8080 set. This is particularly useful where three memory locations are to be addressed and one register is wanted for a counter. To add two multibyte numbers and place the result in a separate location, for example:

```
        XRA   A
        LXI   H, address of sum
        PUSH  H
        LXI   H, address of augend
        LXI   D, address of addend
        MVI   C, byte count
LOOP:   LDAX  D
        ADC   M
        XTHL
        MOV   M,A
        INX   H
        XTHL
        INX   H
        INX   D
        DCR   C
        JNZ   LOOP
        POP   H
```

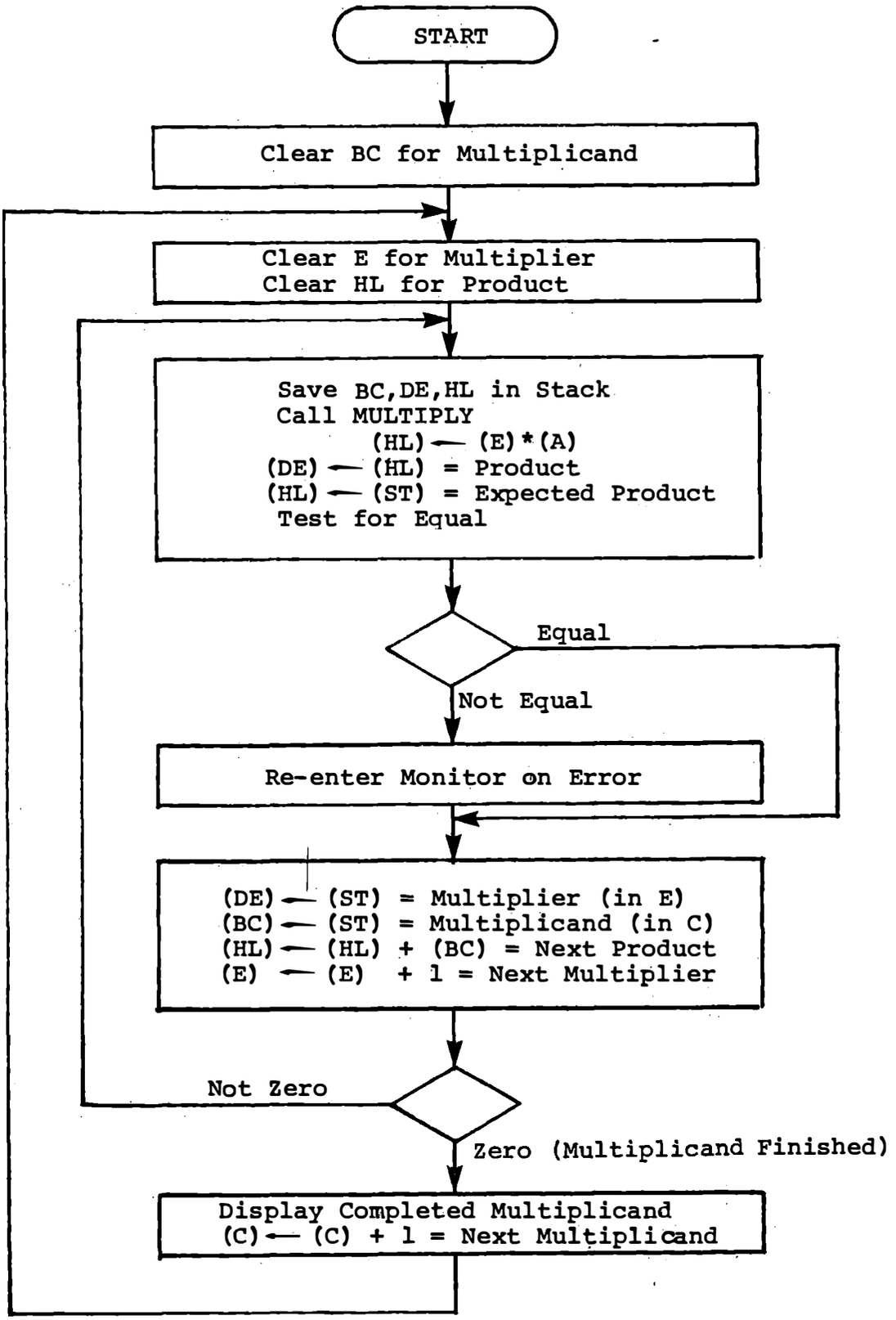
The XTHL instruction is also useful for doing arithmetic in registers. To multiply two numbers of two bytes each, giving a four byte result requires eight registers; BC, DE, HL and ST provide just enough.

6.7 TEST DRIVER FOR MULTIPLY-EXERCISE

We observed in Section 6.3.10 that subroutine MULTIPLY is not fully tested by the procedure we have used, since only a very small sample of all possible multiplicands (adjusted input values) and multipliers (scaling factors) have been used. One way of testing such a subroutine is to try either all possible values or a large random sample of possible values. Then each answer must be checked by some different calculation. We need 65536 tests to try all possible multipliers and multiplicands - a lengthy but reasonable task.

By sequentially testing all multipliers, starting at 00, it is easy to predict the correct result. The first product should be 0000; each following product should be the previous product plus the multiplicand. Figure 6-23 shows the test driver program. Note that when all multipliers have been tested with a given multiplicand we display that multiplicand; this is to provide assurance that the program is running. The test for each multiplicand, in AUTO mode, takes about half a second; in STEP mode more than 40 seconds.

Write the program, using PUSH, POP and XCHG instructions where appropriate. Step through one loop to test the program flow, then switch to AUTO mode and run the program for the full cycle of tests.



Test Driver for MULTIPLY

Figure 6-23

TEST DRIVER FOR MULTIPLY

ADDR		CODE														
8200	0	01		L	X	I		B	,	0	0	0	0		Clear BC for multiplicand
	1	00													
	2	00													
8203	3	AF		X	R	A		A							
	4	5F		M	O	V		E	,	A					Clear multiplier
	5	67		M	O	V		H	,	A					Clear expected
	6	6F		M	O	V		L	,	A					product
8207	7	C5		P	U	S	H		B						Save multiplicand
	8	D5		P	U	S	H		D						Save multiplier
	9	E5		P	U	S	H		H						Save expected product
	A	79		M	O	V		A	,	C					(A) ← multiplicand
	B	CD		C	A	L	L		M	U	L	T	I	PLY	
	C	A0													(HL) ← (E) * (A)
	D	82													
	E	EB		X	C	H	G								(DE) ← Product
	F	E1		P	O	P		H							(HL) ← Expected Product
8210	0	7B		M	O	V		A	,	E					Test low bytes
	1	BD		C	M	P		L							for equal
	2	C2		J	N	Z			8	2	1	A			If error skip
	3	1A													second test
	4	82													
	5	7A		M	O	V		A	,	D					Test high bytes
	6	BC		C	M	P		H							for equal
	7	CA		J	Z				8	2	1	B			Jump if equal
	8	1B													
	9	82													If error
8219	A	E7		R	S	T	4								re-enter monitor
8219	B	D1		P	O	P		D							(E) ← multiplier
	C	C1		P	O	P		B							(C) ← multiplicand
	D	09		D	A	D		B							Next expected product
	E	1C		I	N	R		E							Next multiplier
	F	C2		J	N	Z			8	2	0	7			Loop to multiply
3200	0	07													
	1	82													
	2			(	C	O	N	T	I	N	U	E	D	)	
	3														
	4														
	5														
	6														
	7														
	8														

Figure 6-24a

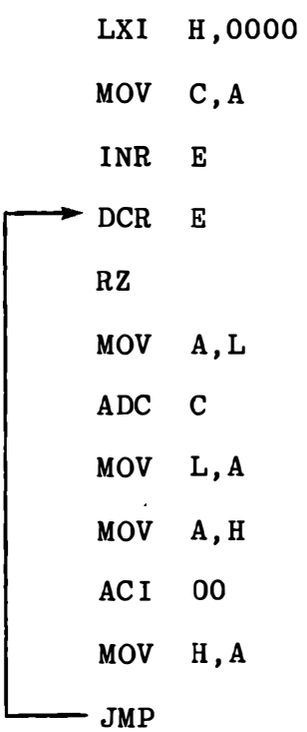
TEST DRIVER FOR MULTIPLY (continued)

ADDR	CODE												
8 0													
1													
822 2	7	9		MOV		A,	C						Display completed
3	1	1		LXI		D,	8	3	F	F			multiplicand
4	F	F											
5	8	3											
6	C	D		CALL		DBY2							
7	9	8											
8	0	2											
9	0	C		INR		C							Next multiplicand
A	C	3		JMP			8203						Loop to clear
B	0	3											multiplicand
C	8	2											product
D													
E													
F													
82A 0	2	1		LXI		H,	0	0	0	0			MULTIPLY
1	0	0											(Clear) Product
2	0	0											
3	4	F		MOV		C,	A						
4	4	5		MOV		B,	H						
5	1	C		INR		E							
82A 6	1	D		DCR		E							
7	C	8		RZ									
8	0	9		DAD		B							
9	C	3		JMP			82A6						
A	A	6											
B	8	2											
C													
D													
E													
F													
3 0													
1													
2													
3													
4													
5													
6													
7													
8													

Figure 6-24b

This is not a complete and perfect test because we have entered MULTIPLY with the flags and unassigned registers containing fixed information. For instance, the given version of MULTIPLY (Figure 6-24b) contains the instructions MOV B,L and MOV C,A to place the multiplicand in pair BC for the DAD instruction. If either of these were left out inadvertently the subroutine would be wrong, but our test program would not catch the error because the test program uses pair BC the same way.

Suppose the repetitive addition in MULTIPLY had been written like this:



Can you see the error? The test program will not find it, because the test for equality between the value returned by MULTIPLY and the known correct result will always clear the carry. Nevertheless, the

routine is wrong because if it were entered with carry set it would give a wrong answer. This error would not be detected in the sensor correction program either - TABLELOOKUP always returns carry cleared, just before the call to MULTIPLY. Imagine using such a subroutine successfully, believing you have tested it with a test driver, and some day copying it into a new program that occasionally calls it with carry set. Even then the error isn't obvious - it only affects the least significant bit of the result.

The design of test programs is extremely difficult - especially for testing your own programs. It is easy to test for errors that you can think of, but those are not the errors you make. If at all possible someone else should write the test, using only the module specification as a guide.

6.8 STACK POINTER INSTRUCTIONS AND RULES

6.8.1 Instructions that Affect Only the Stack Pointer

These intructions are defined for completeness. You are urged not to use them when working with MTS until you fully understnd the monitor program. The first, however, is a vital part of any real program:

```
31          LXI   SP          Load an initial
xx          low address       value to the
yy          high address      stack pointer.
```

This instruction must be executed before the stack can be used for data storage or for subroutine calls. Address 0000 to see it: it is the first instruction in the monitor, and initializes the stack at power-on or reset. Other instructions include:

```
33          INX   SP          Increment stack pointer
3B          DCX   SP          Decrement stack poiner
39          DAD   SP          (HL) < -  (HL) + (SP)
F9          SPHL              (SP) < -  (HL)
```

These manipulate the stack pointer. It may be incremented (with INX SP) to discard data or a return address that has been pushed into the stack, or decremented (with DCX SP) to recover data that has been pushed and popped.

The only way of finding the content of the stack pointer is this:

```
LXI     H,0000
DAD     SP
```

Now (HL) is equal to (SP). Using this together with "LXI SP, address" permits you to assign a different area in memory for the stack, and later restore the previous stack address.

```
LXI     H,0000          Get existing stack pointer
DAD     SP
LXI     SP, address     Address new stack
PUSH    H               Save old stack pointer


POP     H               Recover old stack pointer
SPHL                    Restore old stack pointer.
```

This page intentionally left blank.

6.8.2 Stack Operation Rules

There are some restrictions on use of the stack.

a) For every CALL there must be a RETURN. You must not jump into or out of a subroutine except by CALL and RETURN.

b) For every PUSH there must be a POP. You must not repeatedly push data onto the stack, or you will write into your program memory.

c) To restore registers saved by PUSH, the POP instructions must be in reverse order from the push instructions, because the last data entered is the first data returned.

d) PUSH and POP must be in the same program module. If a subroutine executes a POP with no preceding PUSH, the data recovered will be the return address.

These rules are not absolute: if you understand what you are doing you may use violations of the rules to good purpose. For instance, one program module might push data into the stack for retrieval by another module. This is referred to as unbalanced usage of the stack. It can lead to serious problems unless great care is utilized. (See Section 6.6.2.)

It may be desirable to jump from any of several subroutines to a special location in the main program when an error is detected. This is called an abnormal return. The error handling module may then return to the calling program, it may POP the return address to a register pair and discard it, or it may initialize the stack. Avoid

such procedures until you are reasonably expert.

6.8.3 Monitor Usage of the Stack

The MTS monitor program shares the stack with your program. You will not notice any effect from this except if you manipulate or examine the stack pointer. The monitor operates by "interrupting" your program before each of your instructions is executed. (The subject of interrupts is treated in Chapter 8.) The monitor program pushes your registers into the stack, and calls its own subroutines. When you display the register contents the monitor calculates their locations in the stack and displays the contents of those locations.

When you display the stack pointer, the monitor calculates the address that will be contained in the stack pointer before your next instruction is executed. To look into this, let us again use a program that places readily identified data in the registers.

```
8200    AF      XRA   A       Clear Carry, Set Zero
8201    3E      MVI   A,0A
8202    0A
8203    01      LXI   B,0B0C
8204    0C
8205    0B
8206    11      LXI   D,0D0E
8207    0E
8208    0D
8209    21      LXI   H,0809
820A    09
820B    08
820C    E5      PUSH  H
820D    C5      PUSH  B
820E    D5      PUSH  D
820F    F5      PUSH  PSW
8210    C3      JMP   8210
8211    10
8212    82
```

6.8.3.1 Examining the Monitor Stack

Step through this program to the JMP instruction at 8210. (Do not use breakpoints.) Check the register contents.

(A)=0A (B)=0B (C)=0C (D)=0D (E)=0E (F)=46 (H)=08 (L)=09

Look at your stack:

ADDR	1/P	MEM	83D8	SP.46
NEXT			83D9	0A
NEXT			83DA	0E
NEXT			83DB	0D
NEXT			83DC	0C
NEXT			83DD	0B
NEXT			83DE	09
NEXT			83DF	08

Now let us look into the monitor's part of the stack. The data shown depend on your following these steps exactly; a different key sequence could give different data in the first few bytes here.

ADDR	8	3	C	2	82C2	A2
NEXT					82C3	02

This is a return address within the subroutine DBY2, placed in the stack when DBY2 called another subroutine. It has since been used by a RET instruction. POP and RET do not remove the data or return address from the stack memory; they recover the data and increment the stack pointer. The contents of following locations can be seen by pressing NEXT.

The entire stack is listed here and on the following page:

The return address	83C2	A2
described above	83C3	02
The address previously	83C4	C2
displayed, from PUSH H	83C5	83
The return address	83C6	A2
into DBY2 again	83C7	02
The address of the byte	83C8	C7
previously displayed	83C9	83
Another return address	83CA	D1
for a display subroutine	83CB	02
A return address from	83CC	FA
the NEXT command	83CD	01
A return address to	83CE	A6
the main monitor program	83CF	00

PSW pushed by monitor	83D0	46
	83D1	0A
DE pushed by monitor	83D2	0E
	83D3	0D
BC pushed by monitor	83D4	0C
	83D5	0B
HL pushed by monitor	83D6	09
	83D7	08
PSW pushed by your program	83D8	46
	83D9	0A
DE pushed by your program	83DA	0E
	83DB	0D
BC pushed by your program	83DC	0C
	83DD	0B
HL pushed by your program	83DE	09
	83DF	08

The monitor has used 22 (decimal) bytes in the stack. Until breakpoints are set this is the most it uses.

There is no need for you to be familiar with the details above. In fact one of the great advantages of a stack is that you can use it, following some simple rules, without any concern over where a particular piece of data is stored. However, an understanding of the stack is very useful in troubleshooting programs that misbehave.

6.8.3.2 Breakpoints in the Stack

The MTS monitor breakpoint system also uses the stack, but in a special way. It moves all of the existing stack downward (to lower addresses) in memory, and places the breakpoint information above the stack. Four bytes are stored for each breakpoint. Press RST twice and then step to 8210 again. Now enter a breakpoint.

```
ADDR    BRK                    8210        BP.
```

This has moved the stack down four bytes.

```
ADDR    8   3   B   E          83BE          A2
NEXT                           83BF          02
```

Most of the same data we looked at before are again in the stack, but at locations four bytes lower. A few bytes are different because we have displayed different locations. Look at your stack pointer:

```
ADDR    1/P     MEM            83D4        SP.46
```

Your stack has also been moved down by four bytes. Examine the rest of your stack by pressing NEXT.

```
NEXT    (Register A)           83D5          0A
NEXT            (E)            83D6          0E
NEXT            (D)            83D7          0D
NEXT            (C)            83D8          0C
NEXT            (B)            83D9          0B
NEXT            (L)            83DA          09
NEXT            (H)            83DB          08
```

The next two bytes contain the value of your program counter the last time you pressed RST. Because you pressed it twice, this is 8200.

NEXT	(Program counter at RST)	83DC	00
NEXT		83DD	82

Now we find the breakpoint data.

NEXT	(The address, 8210)	83DE	10
		83DF	82
NEXT	(The data byte (JMP))	83D0	C3
NEXT	(An optional count)	83E1	00

Each breakpoint you enter occupies another four bytes in the stack.

6.8.4 The Growing Stack Problem

When you use the stack for data in complicated problems it is easy to make a mistake and have more PUSH instructions than POP instructions. If this occurs in a repetitive loop the stack will grow by two bytes each time through the loop, and eventually fill the memory with stack data until it destroys the program.

The monitor breakpoint system can be used to protect against a growing stack. In addition to stopping your program when the program counter reaches a breakpoint, the monitor will stop execution if the data stored at any breakpoint address is changed. This feature has two uses: to stop when a loop that is writing to various locations reaches some particular position; or to stop if your program writes in some specific but undesired location. If we choose a location somewhere between the lowest address the stack should ever reach, and the highest address (within page 83xx) that is occupied by variable data, we should expect no change in data at that location. By protecting it with a breakpoint we can detect a growing stack.

Try this disastrous program:

```
8330  21   LXI   H,1111
8331  11
8332  11
8333  E5   PUSH  H
8334  C3   JMP   8333
8335  33
8336  83
```

Be sure to set STEP mode, and set a breakpoint at 83A0 before running.

```
ADDR   8  3  A  0  BRK      83A0          BP.
ADDR   8  3  3  0           8330            21
RUN                         8334            C3
```

Now look at the stack:

```
ADDR    1/P   MEM          83AE        SP.11
NEXT                       83AF          11
```

You will find 11 in all locations up through 83DB. The unbalanced PUSH has wiped out the memory content in this area, but the breakpoint at 83A0 protected everything below 8398. The monitor detected the growing stack at the next instruction after your stack pointer reached 83AE, because the monitor itself had then written into 83A0. Then the monitor's display operations used another eight bytes of stack, down through 8398.

Now let us see what happens without the breakpoint protection.

```
RST
ADDR    8   3   3   0   RUN
```

The display goes blank (probably - depending on the garbage pushed into the stack other things could happen.) Push RST and look at the test program (8330 up). It has been destroyed by the repeated PUSH.

To protect your programs against such errors, follow these rules:

Avoid using memory locations between 8398 and 83FF, except for the stack and display.

Place a breakpoint at 83A0.

Operate the computer in STEP mode (rather than AUTO) until you are satisfied that your program is correct.

6.8.5 Review and Self Test

At this point you have completed two program developments in which you used subroutines that you wrote yourself, and also monitor subroutines for input and output. You have used the stack to store data, and seen how the monitor allows you to examine the stack pointer and the stack. The questions and problems below will help you to judge your understanding of the stack.

1) Identify the four PUSH instructions. Show their effects using transfer notation.

2) Identify the two exchange instructions, and show their effects in transfer notation. How do they affect the length of the stack?

3) How many bytes in the stack are used in the following program segment?

```
PUSH   B
PUSH   D
CALL   STUB  (just a return)
CALL   STUB
POP    D
POP    B
```

4) The monitor initializes the stack pointer, so you need not do so when using the ICS Microcomputer Training System. For almost any other machine your program must initialize the stack. What intruction would you use?

5) You are writing a main program, and intend to call a subroutine called QUIZ whose specification states:

Entry data: (DE) = Address for Data

Return data: (A) = Answer

Registers: All registers are used.

The data address you must pass to QUIZ is stored in memory locations 8300, 8301. Register pairs DE and HL presently contain data that you will need in subsequent operations. Write a program segment to save the data, load the address, call the subroutine, and recover the data.

6) Identify the serious flaw in this multiplication subroutine for (E) * (A), which is required to preserve (BC). Fix it without making the subroutine longer.

```
PUSH   B          (ST) < -   (BC)
LXI    H,0000     Clear Product
MOV    B,L        (BC) < -   Multiplicand
MOV    C,A
INR    E          Test multiplier
DCR    E          and exit if zero
RZ
DAD    B          Multiplication Loop
DCR    E
JNZ
POP    B          (BC) < -   (ST)
RET
```

7) How does your corrected version of the above multiplication subroutine affect Zero and Carry? Modify it to preserve the Carry flag, and return Zero set if the product is 0000. How many bytes in the stack are used when the subroutine is called?

Answers to Self Test, Section 6.8.5

1) The four PUSH instructions are:

C5	PUSH	B	(ST) < - (BC) (SP) < - (SP) - 2
D5	PUSH	D	(ST) < - (DE) (SP) < - (SP) - 2
E5	PUSH	H	(ST) < - (HL) (SP) < - (SP) - 2
F5	PUSH	PSW	(ST) < - (PSW) (SP) < - (SP) - 2

2) The two exchange instructions are:

EB	XCHG	(HL) < - > (DE)
E3	XTHL	(HL) < - > (ST)

XCHG does not use the stack. XTHL does not change the length of the stack, although it temporarily changes the stack pointer.

3) Each PUSH and each CALL uses two bytes in the stack, but the two CALL's use the same stack locations. Therefore the segment uses six bytes in the stack.

4) LXI SP, address initializes the stack pointer. You can also use LXI H, address; SPHL.

5) Program segment:

```
PUSH   D          (ST)  < -  (DE)
PUSH   H          (ST)  < -  (HL)
LHLD   8300       (HL)  < -  Address
XCHG              (DE)  < -  Address
CALL   QUIZ       (A)   < -  Answer
POP    H          (HL)  < -  (ST)
POP    D          (DE)  < -  (ST)
```

Equally good:

```
PUSH   H          (ST)  < -  (HL)
LHLD   8300       (HL)  < -  Address
PUSH   D          (ST)  < -  (DE)
XCHG              (DE)  < -  Address
CALL   QUIZ       (A)   < -  Answer
POP    D          (DE)  < -  (ST)
POP    H          (HL)  < -  (ST)
```

Equivalent, but two bytes longer:

```
PUSH   H          (ST)  < -  (HL)
PUSH   D          (ST)  < -  (DE)
LXI    H,8300     Address the address
MOV    E,M        (DE)  < -  Address
INX    H
MOV    D,M
CALL   QUIZ       (A)   < -  Answer
POP    D          (DE)  < -  (ST)
POP    H          (HL)  < -  (ST)
```

6) The bad multiplication subroutine attempts to exit for a zero multiplier with (BC) in the stack. The version shown below corrects the problem by testing for a zero multiplier before saving (BC) and placing the multiplicand there. This change goes not add or change any instructions.

```
        LXI     H,0000          Clear Product
        INR     E               Test multiplier
        DCR     E               and exit if zero
        RZ
        PUSH    B               (ST) < - (BC)
        MOV     B,L             (BC) < - multiplicand
        MOV     C,A
   ---> DAD     B               Multiplication Loop
   |    DCR     E
   ---- JNZ
        POP     B               (BC) < - (ST)
        RET
```

7) The given solution to 6 preserves Carry if the multiplier is zero; otherwise it returns Carry clear. It always returns Zero set. The following version meets the requirement stated.

```
        LXI     H,0000          Clear Product
        INR     E               Test multiplier
        DCR     E               and exit if zero
        RZ
        INR     A               Test multiplicand
        DCR     A               and exit if zero
        RZ
        PUSH    PSW             Save Carry, Not Zero
        PUSH    B               (ST) < - (BC)
   ---> DAD     B
   |    DCR     E
   ---- JNZ
        POP     B
        POP     PSW
        RET
```

Unless the product is zero, a call to this subroutine uses six bytes in the stack.

6.9 SUBROUTINE CLASSIFICATION

We will define four kinds of subroutines. These are not mutually exclusive.

Global Subroutines

Local Subroutines

Reentrant Subroutines

Interrupt Service Routines

6.9.1 Global Subroutines

A global subroutine is one which is available to be called from any other program module. Typically it serves a general purpose function such as input, output, multiplication, exponentiation, etc. It must be fully specified so that other programmers may use it. A number of restrictions are usually applied, although none are absolute:

a) It always returns to the calling program - it does not make abnormal returns.

b) Any use of the stack is balanced.

c) No data are preserved from one call to the next, except in memory locations specified by the calling program. The global subroutine may have memory areas reserved for its own use.

In the sensor correction problem, MULTIPLY and DISPLAYRESULT could be considered as global subroutines.

6.9.2 Local Subroutines

A local subroutine has restrictions that limit its use by other program modules. Typically it is a small or special purpose procedure. It may have restrictions on entry, abnormal returns, unbalanced stack usage, or it may preserve variable data in permanently assigned memory locations which are also used by other modules. In the sensor correction problem the subroutines that use the directory and data table are clearly local, because the data organization is highly specialized. INPUT could have been written as a global suroutine, but because it calls NEXTSENSOR it must be considered local to the sensor correction problem.

6.9.3 Re-Entrant Subroutines

A re-entrant suroutine is one that can be called even though it is already in use. A few of the monitor subroutines are re-entrant. Any subroutine that is subject to interrupts and which is called by an interrupt service routine must be re-entrant. Full discussion of this type of subroutine is beyond the scope of this text.

6.9.4 Interrupt Service Routine

An interrupt service routine is executed when an external interrupt occurs. There are very special requirements for interrupt servicing, which we will present in Chapter 8 with other input and output functions.

6.9.5 Subroutine Transparency

Transparency implies that a subroutine avoids changing register

contents except as necessary for returning results to the calling program. It is generally a desirable quality in a global subroutine, since the calling program is less likely to need PUSH and POP instructions. The monitor subroutine GETKY is a good example; it preserves D, E, H and L. The fact that the key value is returned in (BC) as well as in (A) is used by many programs that call GETKY. It also returns useful information in the Carry and Zero flags.

The display subroutines of the monitor are not as transparent as would be desirable. It would be sufficient to pass two bytes to DBY2: the data to be displayed and the display location; all other registers and the flags could be preserved since DBY2 has no useful information to be returned except the next display location. It is only convenient for DBY2 itself that the data displayed is copied into Register C; calling programs seldom if ever use that information. An earlier version was even worse; it destroyed the contents of Registers A and B.

The use of alternate entries to a subroutine tends to make it difficult to achieve transparency. This is especially true of internal alternate entries, since registers cannot be pushed into the stack at the beginning of the subroutine. Subroutine DBYTE, for example, loads (DE) with the address 83FF to display a byte in the right hand digit. It could save BC and DE in the stack, but the alternate entry DBY2 could not then be used to display data at other locations.

6.10 MONITOR SUBROUTINES

The remainder of this chapter describes monitor subroutines that are available to you. Others will be found in Appendix A. Timing data are given for some subroutines. These are in decimal count of clocks and include the time for the CALL to the subroutine (17 clocks). The MTS clock rate is 2048000 clocks per second. Operation of the monitor greatly extends the time for the display subroutines (by a factor of approximately 100). Operation of the display DMA channel very slightly extends the time, typically by about 0.1 percent.

6.10.1 Monitor Keyboard Scan Subroutine (SCAN)

Function

Scan the keyboard once, and if a key is pressed decode it and return with the key value in Register A, and the CY flag set. If no key is pressed return with CY clear.

CALL

```
CD          CALL SCAN
57
02
```

Extent

0257 through 0281

Inputs

Keyboard

Outputs

No key pressed: Cy clear, (A) = 00
Key pressed: Key value in A; CY set

Registers

A

Constraints

Uses Output Port C and Input Port A. Interface adaptor must be programmed for these. This is done by the monitor.

Leaves Output Port C loaded with different data depending on which key was pressed.

The monitor is disabled during operation and at return.

Timing

200 to 553 clocks, depending on input key. 457 clocks, if no key is pressed. Add 5432 clocks if the monitor is enabled.

6.10.2 Monitor Key Entry Subroutine (GETKY)

Function

Obtain one key input from the keyboard. Return when a key has been pressed and released.

Call

```
CD          CALL GETKY
3D
02
```

Extent

023D through 0256.
Calls SCAN

Inputs

Keyboard

Outputs

a) Value of the key entered, duplicated in Registers A and C. A hexadecimal key returns the hexadecimal value as the low four bits. Command keys return the following:

MEM	10
REG	11
ADDR	12
STEP	13
RUN	14
NEXT	15
BRK	16
CLR	17

RST causes a general reset to the processor and is not handled by the subroutine.

b) The Carry flag is cleared if a command key is entered; it is set if a hexadecimal key is entered.

Registers

Registers A, B and C are used. Register B is cleared. The contents of Registers D, E, H and L are preserved.

Constraints

a) Input Port A and Output Port C are used.

b) GETKY retains control until a key has been pressed and released. It delays until release has been continuously detected for 20 milliseconds (debouncing).

c) The monitor is disabled during key entry. At return the monitor, display, and keyboard are enabled.

6.10.3 Monitor Data Byte Input Subroutine (ENTBY)

Function

Accepts hexadecimal keys and one command key. Successive hexadecimal keys are combined into a byte and the last two keys pressed are displayed and returned in Register L. The preceding two keys (if any) are returned in Register H. Returns when a command key has been pressed, released and debounced, with the command key value in Registers A and C.

Call

```
CD          CALL  ENTBY
36
03
```

Extent

0336 through 0345.
Calls DBYTE and KEYS.

Inputs

Keyboard

Outputs

Command key in Registers A and C. Last two hexadecimal keys combined as a byte in L. Two preceding hexadecimal keys combined as a byte in H. Number of hexadecimal keys pressed in Register D. Register B is cleared. Zero is set if no hexadecimal keys were pressed. Carry is cleared.

Registers

A, B, C, D, H, L

Constraints

See GETKY Constraints.

6.10.4 Monitor Data Word Input Subroutine (ENTWD)

Function

Accepts hexadecimal keys and one command key. Successive hexadecimal keys are combined into two bytes, and the last four keys pressed are displayed and returned in Registers H and L. When four or more key have been pressed the content of the memory location addressed by those keys is displayed. Returns when a command key has been pressed, released and debounced, with the command key value in Registers A and C.

Call

```
CD      CALL  ENTWD
46
03
```

Alternate Entry (See Note)

```
CD CALL ENTW2
49
03
```

Extent

0346 through 0364
Calls DWORD, DMEM, CLEAR

Inputs

Keyboard

Outputs

Command key in Registers A and C. Last four hexadecimal keys in Registers H and L. Number of hexadecimal keys pressed in Register D. Zero set if no hexadecimal keys entered. Register B cleared.

Registers

A, B, C, D, H, L

Note

Register pair (HL) is cleared at entry ENTWD, so if no hexadecimal keys are pressed (HL) = 0000. If entry ENTW2 is used (HL) is preserved until a hexadecimal key is pressed; then the leading three digits are cleared.

Constraints

See GETKY Constraints.

6.10.5 Monitor Display Digit Subroutine (DISPR)

Function

Display one hexadecimal digit at a specified display position. The input is a hexadecimal value; the output to the display is encoded in the seven segment format.

Call

```
CD      CALL  DISPR
A6
02
```

Extent

02A6 through 02C2

Inputs

a) Hexadecimal value in Register A.

b) Display digit address stored in register pair D,E as follows:

(D,E)	
83F8	Left digit
83F9	Second digit
83FA	Third digit
83FB	Fourth digit
83FC	Fifth digit
83FD	Sixth digit
83FE	Seventh digit
83FF	Right digit

Outputs

a) The seven segment code for the hexadecimal input value is placed in the address provided. If the address is one of those listed above the value will be displayed by the DMA channel, provided that the channel has been turned on. (Note: the monitor leaves the DMA channel turned on, so unless you use other outputs this need not concern you.) If a different address is specified, the seven segment value will be stored there.

b) The address in Register D, E is decremented by one.

Registers

a) Registers A, C, D, E, H, L are used.
b) Only the memory location addressed by D,E is affected.
c) Register A is preserved and copied into Register C.
d) Zero and Carry flags are cleared.

Constraints

Hardware control outputs are not affected. For display to be effective the display must be enabled by a high output at PORTOC7.

Timing 82 clocks

6.10.6 Monitor Display Byte Subroutine - DMEM, DBYTE, DBY2

Function

Display a byte of data as two hexadecimal digits. The display is coded in seven segment format; decimal points are off.

Call

```
CD          CALL  DMEM
94          Display ((HL)) in right hand
02          digits

CD          CALL DBYTE
95          Display (A) in right hand digits
02

CD          Call DBY2
98          Display (A) at location ((DE))
02
```

Extent

0294 through 02A5
Calls DISPR and DIGHI

Inputs

DMEM - Memory address in H,L
DBYTE - Byte in A
DBY2 - Byte in A and memory address for display in DE.

DMEM and DBYTE initialize register pair DE to 83FF to display the byte in the right hand positions.

Outputs

Registers A and C contain byte displayed.

Register pair D,E is decremented by two.

Memory location addressed by contents of register pair DE (at entry) is loaded with the seven segment code for the low order four bits of the input byte.

The next lower memory location (DE) - 1 is loaded with the seven segment code for the high order four bits of the input byte.

Registers

Registers A, C, D, E are used

Registers B, H, L are preserved Register A is preserved except by DMEM.

Constraints

Successive calls to DBY2 will display bytes in successive pairs of digits. DBY2 does not test the address, so the codes may be stored in other memory locations. If data are stored in locations between 83C0 and 83F7 the monitor operation may be disrupted.

The monitor, display, and keyboard are enabled at exit.

Timing

DMEM	332 clocks
DBYTE	325 clocks
DBY2	315 clocks

6.10.7 Monitor Display Word Subroutine - DWORD DWD2

Function

Display two bytes of data as four hexadecimal digits.

Call

```
CD          CALL DWORD
D1          Displays content of
02          register pair H,L in
            four left digits.

CD          CALL  DWD2
D4          Displays content of
02          register pair H,L
            in specified digits
```

Extent

02D1 through 02DB
Calls DBY2

Inputs

a) Data to be displayed in (HL)
b) For DWD2 only, display address in register pair DE

Outputs

Registers A and C contain more significant byte of display. Register pair DE is decremented by 4 from the initial value provided by DWORD or at entry to DWD2.

Registers

Registers A, C, D and E are used. Registers B, H and L are preserved.

Constraints

Successive calls to DWD2 may be made without re-initializing (D,E), provided the first call addressed 83FF. The address supplied in DE is not tested, so the seven segment codes may be stored in other memory locations. If data are stored in locations between 83C0 and 83F7 the monitor operation may be disrupted.

Monitor interrupts, keyboard and display are enabled at exit.

Timing

660 clocks

6.10.8 Monitor Subroutine CLRGT, CLEAR, CLRLP

Function

Clear part or all of the display or memory.

Call

```
CD          CALL  CLRGT
82          Clears four right hand
02          display digits

CD          CALL  CLEAR
87          Clears entire display
02

CD          CALL  CLRLP
8C          Enter with number of bytes to be cleared
02          in (B) and highest address to be cleared
            in (HL)
```

Extent

0282 through 0293

Inputs

CLEAR, CLRGT - none
CLRLP - number of bytes in B
highest address in (H,L)

Outputs

Contents of display memory area starting at right are set to 0 (except for CLRLP)

(B) = 00
(HL) decremented by number of bytes cleared, addressing memory location below last byte cleared.

Registers

B, H, L are used. Zero is set. Carry is preserved.

Timing

CLEAR	284 clocks
CLRGT	174 clocks
CLRLP	27 clocks + 30 clocks for each byte cleared.

6.10.9 Monitor Subroutine DELAY, DELYA

Function

Wait in a loop for a defined time.

Call

```
CD              CALL DELAY
36              Wait for one millisecond
02

CD              CALL DELYA
38              Wait for a time
02              set in Register A
```

Extent

0236 through 023C

Input

DELAY - None
DELYA - Enter with a value in Register A, proportional to the delay desired.

Output

(A) = 00 Zero flag set. Carry preserved

Registers

A is used.

Timing for DELYA

Delay 15 clock times for each count in Register A, plus CALL and RET (27 clocks).

With the monitor enabled the delay is 1381 clocks for each count in Register A, plus 1393 clocks for CALL and RET.

Exact Timing for DELAY

1999 clocks = 0.976 milliseconds. With monitor enabled 182994 clocks = 375 milliseconds.

MICROCOMPUTER TRAINING WORKBOOK

CHAPTER 7

LOGIC AND BIT MANIPULATION

7. LOGIC AND BIT MANIPULATION

It is often necessary to perform functions that depend on individual bits in a byte. This is common, for example, in control problems, where data bits may represent discrete signals rather than numeric values.

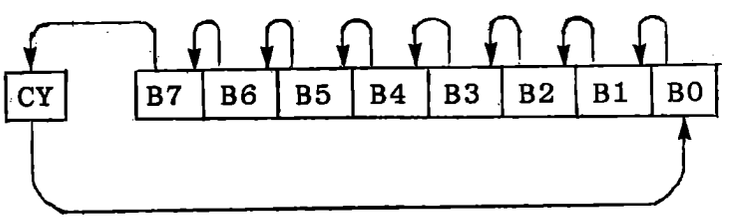
In this chapter two sets of instructions will be introduced: rotate commands, which work on the Accumulator and Carry flag only; and logical functions, which generally involve the Accumulator and another register.

7.1 ROTATE COMMANDS

Rotate is a command to move each bit in the Accumulator to an adjacent position.

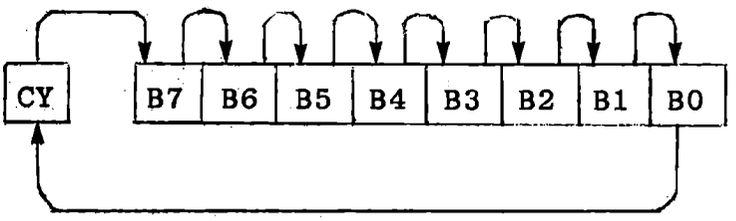
17 RAL Rotate Accumulator Left Through Carry

Move each bit in Register A to the next higher position. Move the most significant bit into the Carry flag. Move the contents of the Carry flag into the least significant bit. Carry is the only flag affected.



1F RAR Rotate Accumulator Right Through Carry

Move each bit in Register A to the next lower position. Move the least significant bit into the Carry flag. Move the content of the Carry flag into the most significant bit. Carry is the only flag affected.



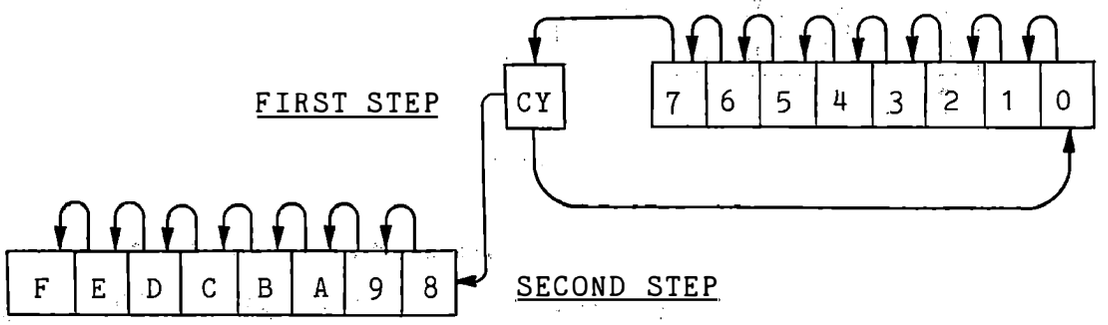
These two rotate commands are sometimes called "arithmetic shift" because they can be used to double or halve the value of the content of Register A and are used in multiplication and division. They can also be used to obtain access to an individual bit. To illustrate the arithmetic properties of rotate, consider the following simple binary numbers:

0000 0111 (=07) 0000 1110 (=0E, or 14 decimal)

The second number results from a left shift of the first, and as a result has been doubled in magnitude.

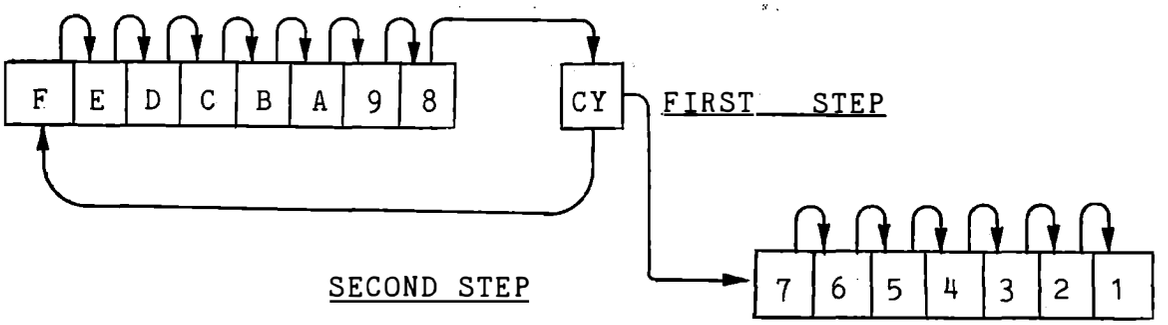
7.1.1 Rotate Exercise

A byte can be doubled by moving it into Register A, clearing the Carry, and rotating left. This places its most significant bit (MSB) in the Carry. To double a two byte value, perform this operation on the less significant byte (Register L), move the result back to L, and repeat on the more significant byte (Register H), but without clearing the Carry:



The result is that each bit in the sixteen bit word has been shifted left one position.

The word can be halved by the reverse process. It must start with the more significant byte and shift right:



In this exercise we shall use the rotate left and rotate right commands in two ways: to perform the arithmetic function of doubling or halving a two byte value, and to move specific bits of a command byte into the Carry so they can be tested. We shall use monitor subroutines to accept data and display the results. The result of the operation is to be preserved until a new command is entered, so that we can (according to the command) either use newly entered data or perform another operation on the previous result.

The result can be displayed by the subroutine DISPLAYRESULT from Chapter 6 if you still have that in memory. Otherwise use an almost identical monitor subroutine. DWD2 is an alternate entry to DWORD, which displays two bytes in the left hand four digits. To place the display at the right, preload (DE) with 83FF and call DWD2.

```
CD      CALL    DWD2    Display the content
D4                      of HL in the digits
02                      addressed by (DE).
```

To display in the left hand digits:

```
CD      CALL    DWORD   Display the content
D1                      of HL in the
02                      left hand digits.
```

We shall use the monitor subroutine ENTWD to obtain two data bytes and a command key, and act on the data word according to the command key entered.

```
CD      CALL ENTWD          (HL) < -    hex keys
46                          (A)  < -    command key
03
```

ENTWD displays the hex keys as they are entered, using the four left hand digits of the display. When four or more digits have been entered a byte is also displayed at the right. This is of no interest here; it is part of the function of ENTWD in the monitor when you press ADDR followed by four keys.

The arithmetic operations are to be performed by subroutines:

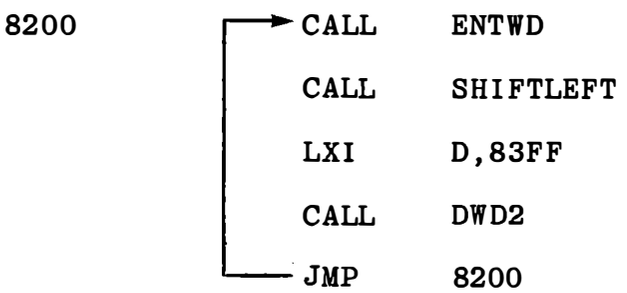
SHIFTRIGHT

Shift the content of register pair HL right one bit. Shift a zero into the high bit of (H). All flags and all other registers must be preserved.

SHIFTLEFT

Shift the content of register pair HL left one bit. Shift a zero into the low bit of (L). All flags and all other registers must be preserved.

Because we are using some new instructions and new monitor subroutines, it is desirable here to start with "bottom-up" programming. Use a simple test driver:



Write the subroutine SHIFTLEFT and test it. Then write SHIFTRIGHT and change the call in the test driver. Try the programs with simple numbers for data entry and observe that SHIFTLEFT doubles the value and SHIFTRIGHT halves the value.

ADDR		CODE													
8	20	C	D		C	A	L	L		E	N	T	W	D	
	1	4	6												
	2	0	3												
	3	C	D		C	A	L	L		S	H	I	F	T	LEFT
	4	4	0												Change to 8250
	5	8	2												for SHIFTRIGHT
	6	1	1		L	X	I		D,	8	3	F	F		
	7	F	F												
	8	8	3												
	9	C	D		C	A	L	L		D	W	D	2		
	A	D	4												
	B	0	2												
	C	C	3		J	M	P			8	2	0	0		
	D	0	0												
	E	8	2												
	F														
8	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														
	9														
	A														
	B														
	C														
	D														
	E														
	F														
3	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure 7-1

SHIFT SUBROUTINES

ADDR		CODE													
8 24	0	AF		XRA			A								SHIFTLEFT
	1	7D		MOV			A	,	L						
	2	17		RAL											
	3	6F		MOV			L	,	A						
	4	7C		MOV			A	,	H						
	5	17		RAL											
	6	67		MOV			H	,	A						
	7	C9		RET											
	8														
	9														
	A														
	B														
	C														
	D														
	E														
	F														
8 25	0	AF		XRA			A								SHIFTRIGHT
	1	7C		MOV			A	,	H						
	2	1F		RAR											
	3	67		MOV			H	,	A						
	4	7D		MOV			A	,	L						
	5	1F		RAR											
	6	6F		MOV			L	,	A						
	7	C9		RET											
	8														
	9														
	A														
	B														
	C														
	D														
	E														
	F														
3	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure 7-2

7.1.2 Rotate Instructions for Control Functions

In the final program the command keys are to be defined as follows:

MEM (=0001 0000)	Halve the new hex value
REG (=0001 0001)	Double the new hex value
ADDR (=0001 0010)	Halve the previous result
STEP (=0001 0011)	Double the previous result
RUN (=0001 0100)	Same as MEM
NEXT (=0001 0101)	Same as REG
BRK (=0001 0110)	Same as ADDR
CLR (=0001 0111)	Same as STEP

Thus the control is exercised according to the two low bits of the command key value. Bit 0 (the least significant bit) selects the arithmetic function; Bit 1 chooses between new data or an old result.

The command key definitions can be remembered easily if you use only the top row. The left keys (REG and MEM) use new data and the right keys (BRK and CLR) use the old result. The outside keys (REG and CLR) double the value and the inside keys halve it.

The main program must make all decisions and call subroutines as required. The decisions are based on the two low bits of the command character:

Bit 0	0 = Halve the data
	1 = Double the data
Bit 1	0 = Use new data
	1 = Use old result

The first decision depends on Bit 1. This can be moved into Carry, where it can control a conditional jump, by two RAR instructions. These also move Bit 0 into Bit 7.

The old result must be kept in memory, since ENTWD uses all registers except E. Let us assign 8300, 8301 for the result.

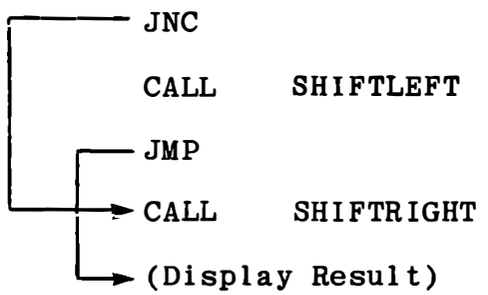
ENTWD will display newly entered data in the left digits. When an old result is to be used for the new calculation, it will be desirable to display it at the left. We can display it now, but must save the command character:

```
        RAR                 Bit 0 to Bit 7 and Bit 1 to CY
        RAR
        JNC                 If new data to be used
        LHLD  8300          (HL) < - Old Result
        PUSH  PSW           Save Command
        CALL  DWORD         Display Result at Left
        POP   PSW           Recover Command
        RAL                 (CY) < - Halve/Double
```

The RAL instruction moves the original Bit 0 of the command from Bit 7, where two RAR's put it, into Carry.

7.1.3 If - Then - Else Construct

With the Carry flag set to distinguish between SHIFTLEFT and SHIFTRIGHT we could do this:

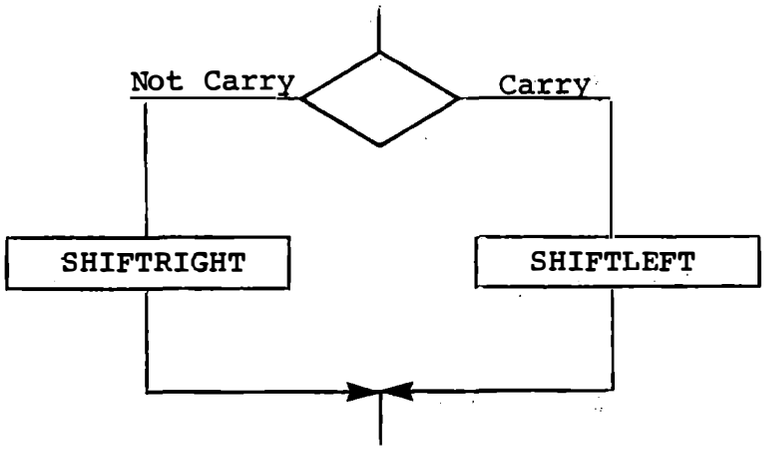


A more attractive way to do this is:

```
PUSH    PSW
CC      SHIFTLEFT
POP     PSW
CNC     SHIFTRIGHT
```

Because it has no jump instructions this has fewer bytes and fewer opportunities for mistakes. (It is slower by either 6 or 11 microseconds than the former arrangement.)

Either of these constructions is shown in a block diagram as:



It is described in words (in computereze) as:

If Carry Then Shiftleft

Else Shiftright

This is a very powerful construction (or "construct," in computereze) and the best computer languages (such as PASCAL) use it very commonly.

Now let us describe the main program as though we were writing it in a "higher level language" - a computer language that understands words instead of binary intructions.

1) Input Data and Command

2) If (Command Bit 1) = 1 then do the following:
   Replace Input Data with Old Result
   Display Old Result

3) If (Command Bit 0) = 1 then Shiftleft
   Else Shiftright

4) Store Result

5) Display Result

6) Go to step 1

The completed program solution is given in Figure 7-3, but for practice you should write and code it youself. Then experiment with numbers.

1 2 3 4 REG		
	1234	2468
CLR (2 x Old)	2468	48D0
CLR (2 x Old)	48D0	91A0
BRK (Old/2)	91A0	48D0
BRK	48D0	2468
BRK	2468	1234
BRK	1234	091A
BRK	091A	048D

Up to this point we can restore the previous value, because we have only shifted zeros out.

CLR (2 x Old)	048D	091A
BRK (Old/2)	091A	048D

One more shift right will lose the one in the least significant bit.

BRK	048D	0246
CLR (2 x Old)	0246	048C
CLR	048C	0918
CLR	0918	1230

Try other numbers to see where you lose data.

LEFT AND RIGHT SHIFT - MAIN PROGRAM

ADDR		CODE													
8 20	0	C	D		C	A	L	L		E	N	T	W	D	(HL) ← Input Data
	1	4	6												(A) ← Command
	2	0	3												
	3	1	F		R	A	R								(CY) ← Bit 1
	4	1	F		R	A	R								(Bit 7) ← Bit 0
	5	D	2		J	N	C			8	2	1	0		Jump if new data
	6	1	0												to be used
	7	8	2												
	8	2	A		L	H	L	D		8	3	0	0		(HL) ← Old Result
	9	0	0												
	A	8	3												
	B	F	5		P	U	S	H		P	S	W			Save shift-1 command
	C	C	D		C	A	L	L		D	W	O	R	D	Display old result
	D	D	1												in left digits
	E	0	2												
	F	F	1		P	O	P		P	S	W				Recover command
8 21	0	1	7		R	A	L								(CY) ← Bit 0
	1	F	5		P	U	S	H		P	S	W			Save Carry
	2	D	C		C	C			S	H	I	F	T	L	EFT
	3	4	0												If CY then new
	4	8	2												shift left
	5	F	1		P	O	P		P	S	W				Else shift right
	6	D	4		C	N	C		S	H	I	F	T	R	IGHT
	7	5	0												
	8	8	2												
	9	2	2		S	H	L	D		8	3	0	0		Store result
	A	0	0												
	B	8	3												
	C	1	1		L	X	I		D	,	8	3	F	F	Display result
	D	F	F												at right
	E	8	3												
	F	C	D		C	A	L	L		D	W	D	2		
8 22	0	D	4												
	1	0	2												
	2	C	3		J	M	P			8	2	0	0		
	3	0	0												
	4	8	2												
	5														
	6														
	7														
	8														

Figure 7-3a

ADDR		CODE													
8 24	0	AF		XRA			A								SHIFTLEFT
	1	7D		MOV			A	,	L						
	2	17		RAL											
	3	6F		MOV			L	,	A						
	4	7C		MOV			A	,	H						
	5	17		RAL											
	6	67		MOV			H	,	A						
	7	C9		RET											
	8														
	9														
	A														
	B														
	C														
	D														
	E														
	F														
8 25	0	AF		XRA			A								SHIFTRIGHT
	1	7C		MOV			A	,	H						
	2	1F		RAR											
	3	67		MOV			H	,	A						
	4	7D		MOV			A	,	L						
	5	1F		RAR											
	6	6F		MOV			L	,	A						
	7	C9		RET											
	8														
	9														
	A														
	B														
	C														
	D														
	E														
	F														
9	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure 7-3b

7.1.4 Arithmetic Substitutes for RAL

We have seen that RAL doubles the content of A. This can equally well be done by adding the content of A to itself by ADD A or ADC A.

```
87   ADD  A        (A) < -  (A) + (A)
8F   ADC  A        (A) < -  (A) + (A) + (CY)
```

ADD A discards the old content of Carry. Since the value is doubled it must result in an even number, with 0 in the least significant bit. ADC A adds the old Carry in, so it is identical to RAL in its numeric result. In SHIFTLEFT we can discard the XRA A, whose function was to clear Carry, and replace the first RAL with ADD A. Replace the second RAL with ADC A. Test to see that the result is identical.

These instructions differ from RAL in that all flags are affected, whereas RAL affects only the Carry flag. Sometimes one usage or the other is preferred because of the different effect on flags.

We also have available the double precision add instruction DAD H. This shifts left the 16 bit number in (HL), so we can replace the entire SHIFTLEFT subroutine by:

```
8240    29    DAD  H     (HL) < -  (HL) + (HL)
8241    C9    RET
```

Like RAL this affects only the Carry flag. Make the substitution and see that the program operation is unchanged.

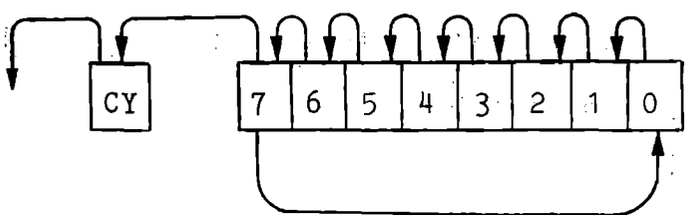
There is no arithmetic instruction equivalent to RAR.

7.1.5 Logical Rotate

Two other rotate commands are provided in the 8080, which are similar to RAL and RAR except for their handling of the Carry and the most and least significant bits.

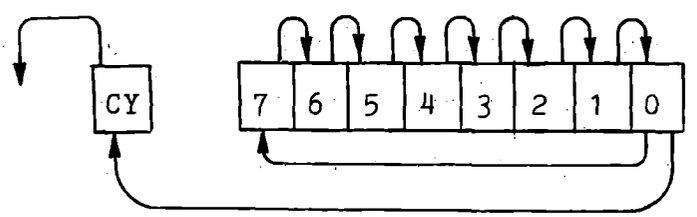
07 RLC Rotate Left into Carry

Move each bit in Register A to the next higher position. Move MSB into the Carry flag and into LSB. Only the Carry flag is affected.



0F RRC Rotate Right into Carry

Move each bit in Register A to the next lower position. Move LSB into the Carry flag and into MSB. Only the Carry flag is affected.



These two instructions are called logical rotate because they treat

the Accumulator as an eight bit ring in which MSB and LSB are conceptually juxtaposed. The operation does not have an arithmetic equivalent.

The logical shifts discard the old value of the Carry flag. If in the SHIFTLEFT and SHIFTRIGHT subroutines you replace both RAL commands (17) with RLC (07) and both RAR commands (1F) with RRC (0F) you will see that the two bytes are now independent of each other. If you enter two new bytes, using REG to shift left, and then BRK to shift the same data right, the input value will be restored. Now if you use either BRK or CLR eight times each byte will be shifted back to its original value. After four shifts in one direction the digits of each byte are interchanged:

1 2 3 4 REG	1234	2468
CLR	2468	48D0
CLR	48D0	90A1
CLR	90A1	2143

Another four shifts in either direction will restore the initial values.

Can you modify the SHIFTLEFT and SHIFTRIGHT subroutines to achieve sixteen bit logical rotates? This will preserve all bits, so that pressing BRK or CLR sixteen times will restore the initial value. Think of the solution before looking at Figure 7-4.

This page intentionally left blank.

SIXTEEN BIT LOGICAL ROTATE

CODING SHEET

ADDR	CODE															
8 24 0	7	C		M	O	V		A	,	H				SHIFTLEFT
1	1	7		R	A	L								
2	7	D		M	O	V		A	,	L				
3	1	7		R	A	L								
4	6	F		M	O	V		L	,	A				
5	7	C		M	O	V		A	,	H				
6	1	7		R	A	L								
7	6	7		M	O	V		H	,	A				
8	C	9		R	E	T								
9														
A														
B														
C														
D														
E														
F														
8 25 0	7	D		M	O	V		A	,	L				SHIFTRIGHT
1	1	F		R	A	R								
2	7	C		M	O	V		A	,	H				
3	1	F		R	A	R								
4	6	7		M	O	V		H	,	A				
5	7	D		M	O	V		A	,	L				
6	1	F		R	A	R								
7	6	F		M	O	V		L	,	A				
8	C	9		R	E	T								
9														
A														
B														
C														
D														
E														
F														
3 0														
1														
2														
3														
4														
5														
6														
7														
8														

Figure 7-4

7.2 BINARY ENTRY AND DISPLAY EXERCISE

In the preceding exercise we accepted hexadecimal keys and displayed hexadecimal values, using monitor subroutines. Now we shall use the display techniques learned in Chapter 4 to display a number in binary form. Monitor subroutine GETKY will be used to read in one key at a time, and distinguish commands from hex keys.

```
CD      CALL GETKY      (A) = (C)  < -   Key
3D                      Carry set if hex
02
```

At any moment we shall control one bit of the number being entered, and one digit of the display. This bit and display digit can be changed back and forth between 0 and 1. A command key will move on to the next bit and display digit.

Only the least significant bit of a hex key will be used. If it is zero, we shall put a zero into the bit being entered; if it is one, put a one into the bit being entered. Display 0 or 1 in the corresponding display digit, using 3F for 0, 06 for 1. Until a bit has been entered display a decimal point only (80) in the digit.

A convenient way of both testing and keeping the data bit entered uses the RRC intruction. This sets Carry if the least significant bit (Bit 0) is one, and also copies Bit 0 into Bit 7. Save this in Register L.

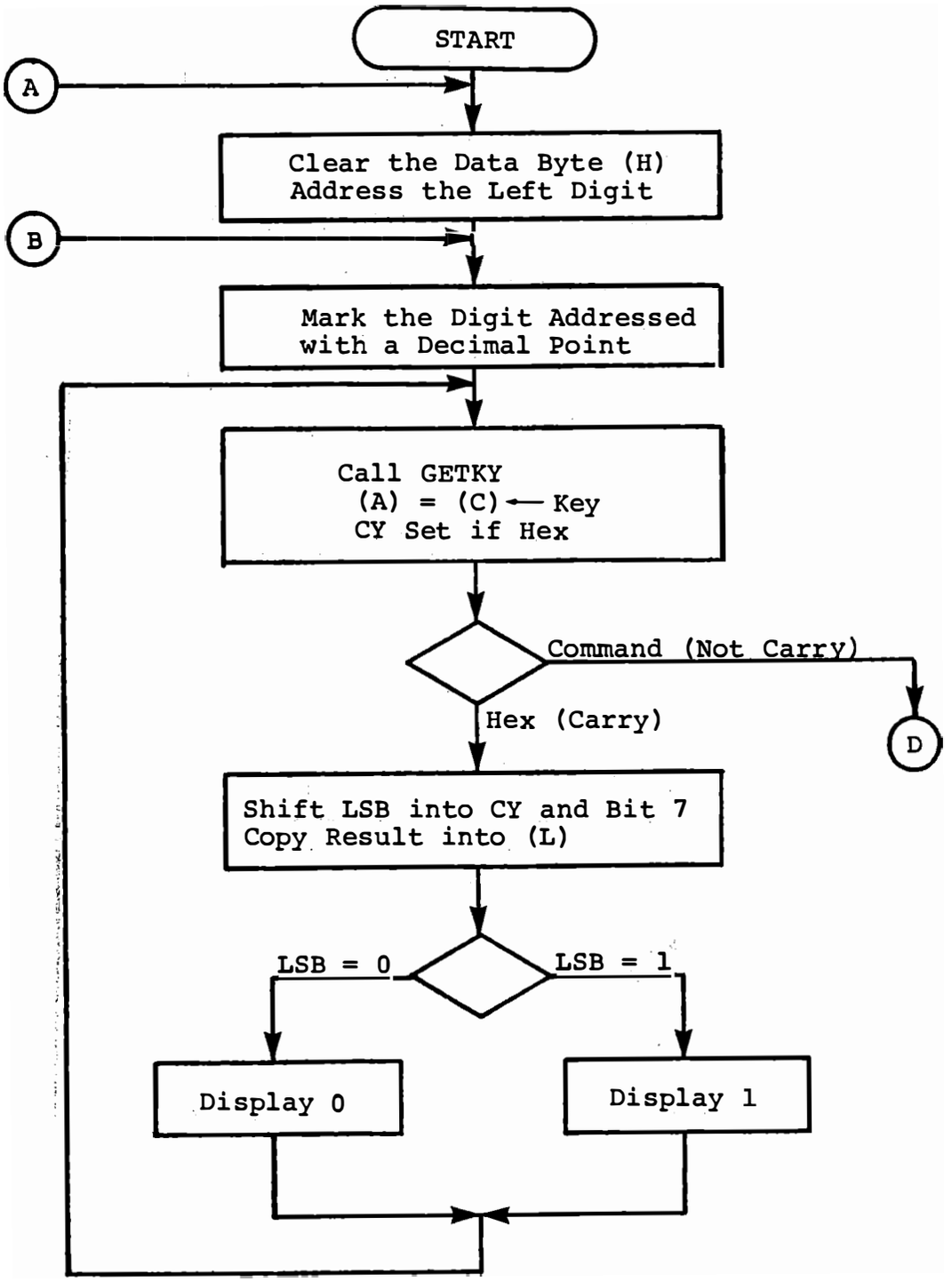
When any command key is entered, do the following:

If no hex key has yet been entered for the current position, enter and display a zero.

Shift the data bit entered for the current position into the least significant bit of a data byte, shifting preceding bits left.

Address the next digit of the display. If still within the eight digit display then loop to accept data for the next bit. At the end of the display, when eight bits have been entered, clear the binary display and show the eight bit value in hexadecimal.

Figure 7-5 shows the program as a flow diagram.



Binary Entry and Display Program

Figure 7-5a

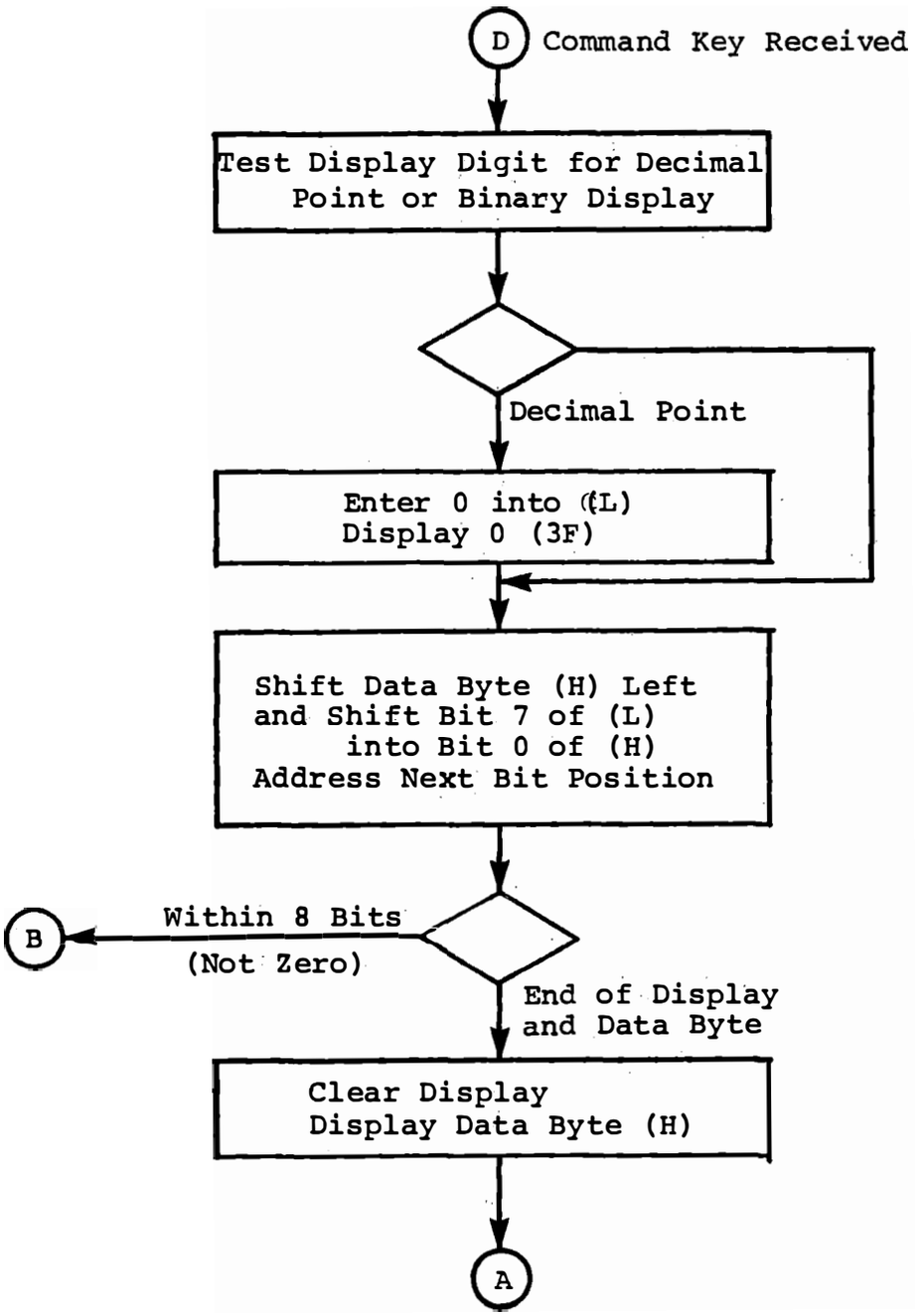

D
Command Key Received
Test Display Digit for Decimal Point or Binary Display
Decimal Point
Enter 0 into (L)
Display 0 (3F)
Shift Data Byte (H) Left
and Shift Bit 7 of (L)
into Bit 0 of (H)
Address Next Bit Position
Within 8 Bits
(Not Zero)
B
End of Display
and Data Byte
Clear Display
Display Data Byte (H)
A

Figure 7-5b

Since the display digit contains only a decimal point (80) until a hex key has been pressed, we can test for that value when a command is entered. If nothing has been entered, replace 80 by 3F to display 0. Also enter 00 into Register L.

Register H is used for the data byte entered; the high bit of Register L contains the new data bit. DAD H will shift the data byte and enter the new bit.

We have monitor display routines to clear the display and show the data byte (H) in hex.

```
CD      CALL    CLEAR   Clear the display
87                      Uses (B) and (HL)
02
```

Since CLEAR uses Register H but not A, precede this with MOV A, H. Then use:

```
CD      CALL    DBYTE   Display (A)
95                      at the right
02
```

Write the program and try entering binary values. The next exercise will use similar techniques.

BINARY ENTRY AND DISPLAY

ADDR	CODE												
8200	26		MVI		H,	00			(A) Clear data byte
1	00												
2	11		LXI		D,	83F8			Address left digit
3	F8												
4	83												
5	3E		MVI		A,	80			(B) Mark digit
6	80												addressed by (DE)
7	12		STAX		D				with decimal point
8	CD		CALL		GETKY				(C)
9	3D								
A	02								
B	D2		JNC		COMMAND				
C	20								
D	82								
E	0F		RRC						CY ← Bit 0 and
F	6F		MOV		L,	A			to Bit 7 of (L)
8210	3E		MVI		A,	3F			If LSB = 0
1	3F								Then display □
2	D2		JNC		8207				
3	07								
4	82								
5	3E		MVI		A,	06			Else display 1
6	06								
7	C3		JMP		8207				
8	07								
9	82								
A									
B									
C									
D									
E									
F									
30									
1									
2									
3									
4									
5									
6									
7									
8									

Figure 7-6a

BINARY ENTRY AND DISPLAY (continued)

ADDR		CODE													
8 22	0	1	A		L	D	A	X		D					Test display digit
	1	F	E		C	P	I		8	0					for desired point
	2	8	0												
	3	C	2		J	N	Z			8	2	2	B		Jump if binary
	4	2	B												display present
	5	8	2												
	6	2	E		M	V	I		L	,	0	0			Else enter 0
	7	0	0												to register L
	8	3	E		M	V	I		A	,	3	F			and display □
	9	3	F												
	A	1	2		S	T	A	X		D					
822	B	2	9		D	A	D		H						Shift list into (H)
	C	1	C		I	N	R		E						Address next
	D	C	2		J	N	Z			8	2	0	5		list and digit
	E	0	5												and continue
	F	8	2												until end
8 23	0	7	C		M	O	V		A	,	H				(A) ← data byte
	1	C	D		C	A	L	L		C	L	E	A	R	Clear binary
	2	8	7												display
	3	0	2												
	4	C	D		C	A	L	L		D	B	Y	T	E	Display data byte
	5	9	5												
	6	0	2												
	7	C	3		J	M	P			8	2	0	0		
	8	0	0												
	9	8	2												
	A														
	B														
	C														
	D														
	E														
	F														
3	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure 7-6b

7.3 LOGIC FUNCTIONS

Logic functions operate on individual bits or pairs of bits. The defined functions are:

Complement

AND

Inclusive OR

Exclusive OR

7.3.1 Complement (CMA)

If a bit is 0, its complement is 1; if a bit is 1, its complement is 0. The complement is often symbolized by a bar, read as NOT. Thus:

If X = 1, $\overline{X}$ = 0 (If X equals one, NOT X equals zero)

If X = 0, $\overline{X}$ = 1 (If X equals zero, NOT X equals one)

The complement of a byte is the byte comprising the complements of each of the bits of the original byte. For example:

$$\overline{01101100} = 10010011$$

or $\overline{6C} = 93$

This function is generated in the 8080 by the instruction:

2F CMA Complement Accumulator

(A) < - $\overline{(A)}$

No flags are affected.

The complement function is also involved in arithmetic, as you will see in later chapters.

7.3.2 AND (ANA)

The AND of two bits is 1 if and only if both bits are 1. The AND is symbolized by a dot, or by the intersection symbol $\cap$, or simply by placing two symbolic characters next to each other. Since we will be dealing with bytes for which multiplication is also defined, we will use $\cap$

$$X \cap Y \qquad (X) \text{ AND } (Y)$$

The operation of a logical function is often shown by a truth table.

X	Y	(X) $\cap$ (Y)
0	0	0
0	1	0
1	0	0
1	1	1

The AND of two bytes is the byte comprising the bits generated by the AND of corresponding bits in the two original bytes. For instance:

$$01101100 \cap 11101001 = 01101000$$
$$\text{or } 6C \cap E9 = 68$$

A logic function of two bytes expressed in hexadecimal is not obvious at a glance - one usually has to expand the bytes to binary representation.

The AND of the bytes in Register A and any other register (or M, the

memory location addressed by the content of register pair H) is generated, and the result placed in Register A, by:

ANA	r	And (r) with (A); place the result in A. (A) <- (A) $\frown$ (r) The Carry flag is cleared. Other flags are set or cleared according to the result.

7.3.3 Inclusive OR (ORA)

The inclusive OR of two bits is 1 if either of the bits is 1. The OR is symbolized by a + sign or the union symbol $\cup$. Again, since addition is defined for bytes, we use $\cup$:

X	Y	(X) $\cup$ (Y)
0	0	0
0	1	1
1	0	1
1	1	1

The OR of two bytes it the OR of corresponding bits:

01101100 $\cup$ 11101001 = 11101101
or 6C $\cup$ E9 = ED

The OR of the bytes in Register A and any other register (or M) is generated, and the result placed in Register A, by ORA r.

ORA	r	OR (r) with (A); place the result in A. (A) <- (A) $\cup$ (r) The Carry flag is cleared. Other flags are set or cleared according to the result.

Since $1 \cup 1 = 1$ and $0 \cup 0 = 0$, the function ORA A does not change the content of Register A, but sets the Zero flag if (A) = 0, and clears it otherwise. It similarly sets or clears the other flags which have not yet been defined. We have used it to clear the Carry flag.

7.3.4 Exclusive OR (XRA)

The Exclusive OR of two bits is 1 if one but not both of the bits is 1. The Exclusive OR, commonly referred to as XOR (sometimes EXOR), is symbolized $\oplus$.

X	Y	(X) $\oplus$ (Y)
0	0	0
0	1	1
1	0	1
1	1	0

The XOR of two bytes is the XOR of corresponding bits:

01101100	$\oplus$	11101001	=	10000101
or 6C	$\oplus$	E9	=	85

The XOR of the byte in Register A and any other register (or M) is generated, and the result placed in Register A, by XRA r.

XRA	r	XOR (r) with (A); place the result in A. (A) <- (A) ⊕ (r) The Carry flag is cleared. Other flags are set or cleared according to the result.

Recognize that since $1 \oplus 1 = 0$, and $0 \oplus 0 = 0$, then $(A) \oplus (A) = 0$. Therefore XRA A is used to clear Register A (SUB A could also be used.)

7.3.5 Immediate Logic Functions

For each of the logic functions except complement, there is a set of instructions using each of the registers (or the referenced memory location) as a source for the data byte. These instructions are tabulated in the instruction chart. As with the arithmetic instructions, there are also immediate versions of each:

E6	ANI	AND Immediate data
xx		with Register A.
F6	ORI	OR Immediate data
xx		with Register A.
EE	XRI	XOR Immediate data
xx		with Register A.

These generate the indicated logic function of the content of Register A with the content of Byte 2 of the instruction and place

the result in Register A. The Carry flag is cleared and other flags are set or cleared according to the result of the operation.

The instruction ANI is especially useful in masking unwanted data from the result of an input operation. For instance, if you are concerned with Bit 4 of an input byte and want to jump if it is one, it is more efficient to write:

```
ANI     10      (00010000)
JNZ
```

than to shift the data bit to the Carry flag and jump if Carry. Even more important, ANI can test for any of several bits:

```
ANI     58      (01011000)
JNZ
```

If Bit 3 or Bit 4 or Bit 6 of Register A is 1, the result is not zero.

7.3.6 Set and Complement Carry

These two instructions affect only Carry.

```
37      STC     Set Carry
3F      CMC     Complement Carry
```

We have seen several ways of clearing the Carry flag, but these also affect other flags. STC,CMC clears Carry with no effect on other flags.

7.4 LOGIC FUNCTION EXERCISE

Now we shall plan an exercise using bit shifting and masking techniques to demonstrate the logic functions. We shall accept eight data bits as a sequence of ones and zeros from the keyboard and display them as they are received, using the decimal point to mark the bit position, as in the exercise of Section 7.2. At the same time, we shall perform a logic function, combining the new data bit with one previously entered in the same bit position. The new bit, the old bit, and the result of the logic function will all be displayed together.

‾‾ Top Horizontal = Logic Function

— Middle Horizontal = Old Bit

_ Bottom Horizontal = New Bit

. Decimal Point = Bit Marker

A blank horizontal bar will represent 0 and an illuminated bar will represent 1.

The logic function of the old and new data bytes will be selectable by command keys. Define the commands in the top row of keys for this purpose.

REG = ORA

MEM = ANA

BRK = XRA

CLR = CMA

These commands are to be stored, so that whenever a bit is changed the function most recently selected can be generated and displayed.

Define the command keys at the right for moving data.

NEXT	Move to next bit position
ADDR	Ignore - has no purpose here
RUN	Replace old data with result of logic function
STEP	Replace old data with new data

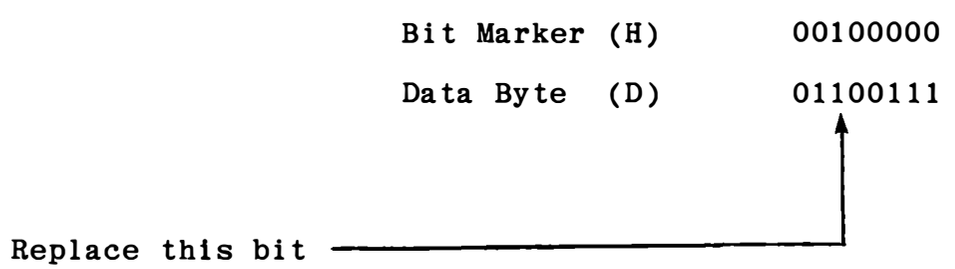
These commands are only executed when entered, so they need not be stored.

7.4.1 Data Byte and Bit Marker

In this new exercise we will not clear the data byte after entering eight bits, but wrap around to the most significant bit. When NEXT is pressed we shall move on to the next bit, preserving the existing bit, rather than inserting a zero. With this rule, the shifting technique we used for entering bits into a register in Section 7.2 is no longer suitable; instead we must use a masking technique. Use Register D for the data byte. In Register H keep a mark indicating position:

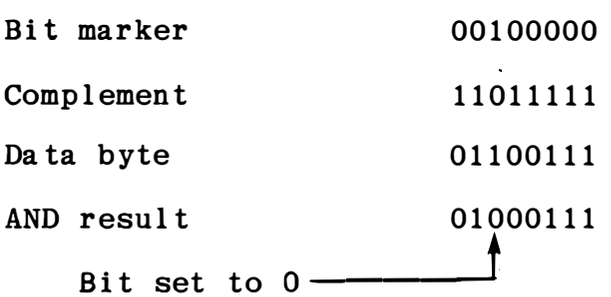
```
80 = Most significant bit (Bit 7)
40 = Bit 6
20 = Bit 5
10 = Bit 4
08 = Bit 3
04 = Bit 2
02 = Bit 1
01 = Bit 0
```

The bit marker keeps track of which bit is to be entered, and we will use it to modify individual bits. For example:

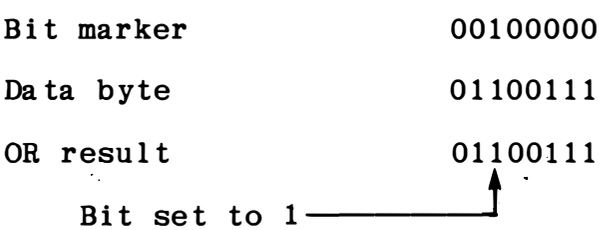


There are several ways of entering the new bit. One obvious way is to test the key (in the Carry after a shift right) and jump to one of two separate procedures:

Key is zero:



Key is one:



We shall consider alternative methods later.

The bit marker itself is changed when NEXT is pressed. It moves to the right by one bit position, until it is 00000001, marking the least significant bit. Now it must wrap around to the most significant bit. This is exactly the function of the RRC instruction, so the response to NEXT will be:

```
MOV  A,H
RRC
MOV  H,A
```

7.4.2 Keyboard Functions

Review the responses to the keyboard entries:

a) When a hex key is pressed, enter its least significant bit into Register D in the bit position marked by (H). Display the new data. Calculate the logic function and display the new result.

b) When NEXT is pressed, move the bit marker in (H). Display the bit marker.

c) When STEP is pressed, replace the old data byte with the new data byte. Display the newly replaced "old" data byte. Calculate the logic function and display the result.

d) When RUN is pressed, replace the old data byte with the result of the logic function. Display the newly replaced "old" data byte. Calculate the logic function again (now using different "old" data) and display the result.

e) When REG, MEM, BRK or CLR is pressed, replace the logic function selector. Calculate the new logic function and display the result.

From the above we can see that most keys require calculation and display of the logic function. Only NEXT and ADDR have no effect on the function result. Therefore it is reasonable to recalculate the logic function and display it after every key has been processed. Once displayed it need not be retained.

7.4.3 Register Assignments

We can keep all of the data for this program in registers, using the stack for temporary storage. In the main program the following assignments are convenient:

(D) = New Data

(E) = Old Data

(H) = Bit Mark

(L) = Logic Function Selector

These registers are preserved by GETKY, and other subroutines must affect them only as required by the data and command entries.

7.4.4 Subroutines for Logic Functions Exercise

Let us define the following subroutines to accept and process data and commands.

GETKY The monitor subroutine (at 023D) which accepts one key and returns:

(A) = (C) = key value

(B) = 00

Carry set for a hexadecimal key

Carry clear for a command key

D, E, H and L are preserved.

DATA A local subroutine to enter the least significant bit of a hex key into the new data byte (D) and display the byte.

COMMAND A local subroutine to interpret the commands. The logic function commands (REG, MEM, BRK, CLR) will be stored; the other commands will be processed immediately.

FUNCTION Generate the logic function selected by (L) (ORA, ANA, XRA, CMA), of new data (D) with old data (E). Return the result in Register A.

DISPLAY Display one byte of data which may be the new data, old data, logic function or bit marker as selected by the calling program. Enter with:

(A) = data to be displayed

(B) = symbol for data, as follows:

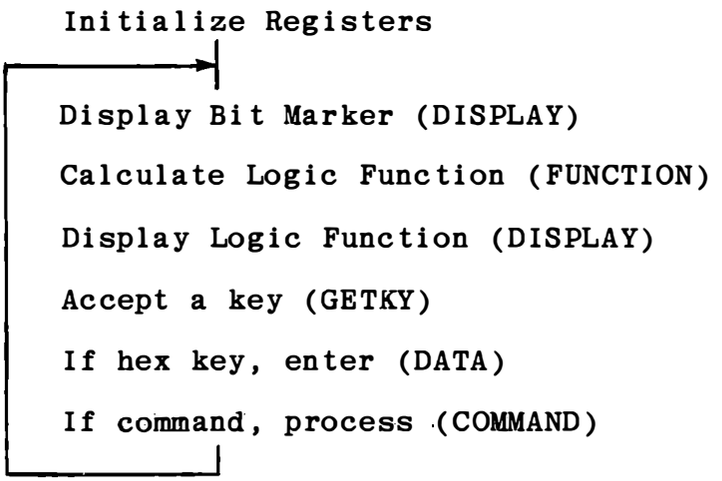
01 = Logic Function (Top Horizontal)
40 = Old Data (Middle Horizontal)
08 = New Data (Bottom Horizontal)
80 = Bit Marker (Decimal Point)

All segments of each display digit, except the segment designated by (B), must be preserved.

THIS PAGE INTENTIONALLY LEFT BLANK.

7.4.5 Main Program for Logic Function Exercise

Having assigned registers and identified subroutines we can now proceed to develop the program, again using the top-down approach.



The bit marker is displayed by a call from MAIN because we do not want to wait until a key is pressed to show the location. There are two reasons for placing the logic function display in MAIN rather than in FUNCTION. When the command key RUN is pressed we must calculate the function in order to replace the old data, but do not particularly want to display it. Second, FUNCTION will require jumps to each of the logical functions (ORA, ANA, XRA, CMA); the subroutine will be shorter if each of these can be followed by RET instead of calling DISPLAY.

Assign the following memory locations to the subroutines and write the main program.

8220	DISPLAY
8240	DATA
8260	COMMAND
82A0	FUNCTION

Use stubs (RET) for the subroutines. The registers should be initialized as follows:

(D)	< -	00	for new data byte
(E)	< -	00	for old data byte
(H)	< -	80	mark most significant bit
(L)	< -	17	for CMA function

LXI instructions can be used for these.

With no subroutines except GETKY the display will be blank and pressing keys will have no visible effect. Place a breakpoint after the return from GETKY and step through the program to be sure that DATA and COMMAND are called appropriately in response to hex and command keys.

7.4.6 Stubs for COMMAND and FUNCTION

These subroutines will be fairly complicated, but we must at least react to NEXT before we can enter data and observe the display. It will also be useful to have something returned by FUNCTION, for testing the display.

The stub for COMMAND can test for NEXT, and if the command is NEXT perform the logical rotate right on (H). If the command is not NEXT then ignore it.

A convenient stub for FUNCTION returns the complement of the new data. This is the same function that CLR will give us when COMMAND and FUNCTION are completed. Since DATA does not yet exist, register D will always contain 00, and FUNCTION will return FF. The stubs are shown in Figure 7-8.

Test the operation of COMMAND by setting a breakpoint at 8265. It should only be reached after a NEXT key. The content of H should be halved each time NEXT is pressed, until after 01 it becomes 80.

LOGIC FUNCTIONS - MAIN

ADDR		CODE														
8	20	0	1	1		L	X	I		D	,	0	0	0	0	Instructions
		1	0	0												(E) ← Old Data
		2	0	0												(D) ← New Data
		3	2	1		L	X	I		H	,	8	0	1	7	
		4	1	7												(L) ← CMA Selector
		5	8	0												(H) ← Bit Mark
		6	7	C		M	O	V		A	,	H				Display Bit Mark
8	20	7	0	6		M	V	I		B	,	8	0			Symbol for Bit Mark
		8	8	0												= decimal point
		9	C	D		C	A	L	L		D	I	S	P	L	AY
		A	2	0												
		B	8	2												
		C	C	D		C	A	L	L		F	U	N	C	T	ION
		D	A	0												(A) ← Function (L)
		E	8	2												of (D) with (E)
		F	0	6		M	V	I		B	,	0	1			Symbol for Function
8	21	0	0	1												= top horizontal
		1	C	D		C	A	L	L		D	I	S	P	L	AY
		2	2	0												Display Function
		3	8	2												
		4	C	D		C	A	L	L		G	E	T	K	Y	Accept data
		5	3	D												or command
		6	0	2												
		7	D	C		C	C				D	A	T	A		If key, enter
		8	4	0												
		9	8	2												
		A	D	4		C	N	C			C	O	M	M	A	ND
		B	6	0												If command,
		C	8	2												process or store
		D	C	3		J	M	P			8	2	0	6		
		E	0	6												
		F	8	2												
8		0														
		1														
		2														
		3														
		4														
		5														
		6														
		7														
		8														

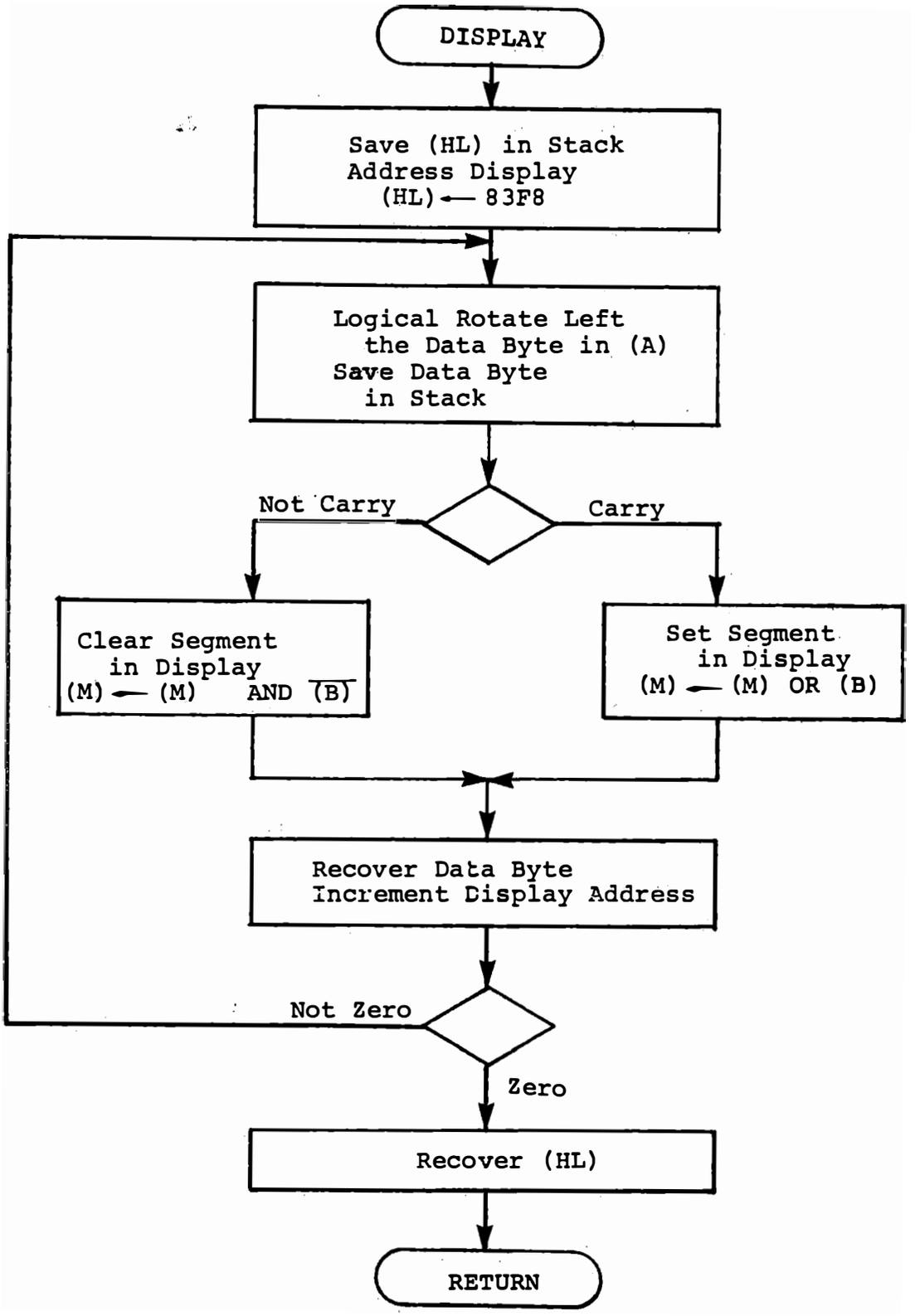
Figute 7-7

STUBS FOR COMMAND AND FUNCTION

CODING SHEET

ADDR		CODE														
8	26	0	FE			CPI			NEXT							COMMAND STUB
		1	15													
		2	C2			JNZ				8269						
		3	69													
		4	82													
		5	7C			MOV			A,	H						
		6	0F			RRC										
		7	67			MOV			H,	A						
		8	C9			RET										
82	6	9	C9			RET										
		A														
		B														
		C														
		D														
		E														
		F														
8	2A	0	7A			MOV			A,	D						FUNCTION STUB
		1	2F			CMA										
		2	C9			RET										
		3														
		4														
		5														
		6														
		7														
		8														
		9														
		A														
		B														
		C														
		D														
		E														
		F														
8		0														
		1														
		2														
		3														
		4														
		5														
		6														
		7														
		8														

Figure 7-8



Logic Functions Display Subroutine

Figure 7-9

7.4.7 Logic Functions DISPLAY Subroutine

Now create the subroutine DISPLAY, since this will be a necessary tool for the rest of the program. It will also give you some familiarity with the logic function commands.

In accordance with the description in Section 7.4.4, DISPLAY receives a byte of data in Register A and a symbol in Register B. It must preserve all segments of each display digit except the segment designed by (B). For each bit in (A) the designated segment of the corresponding display digit must be cleared or set. Figure 7-9 shows one design for DISPLAY. Although a shorter version could be written, this has the advantage of simplicity.

Entering the symbol (B) into a display digit is done by the OR function.

```
MOV  A,M
ORA  B
MOV  M,A
```

Entering a zero requires that (B) be complemented and used as a mask.

```
MOV  A,B
CMA
ANA  M
MOV  M,A
```

When (B) is complemented all bits except the designated symbol bit become 1, while the symbol bit becomes 0. The AND function then preserves all display segments except the one designated.

Stepping through DISPLAY is unrewarding because it takes existing data from the display, but this is upset by the monitor. The best way to debug a display subroutine is to substitute some different memory locations such as 82F8 - 82FF when stepping.

With DISPLAY and the stub for COMMAND you can observe the bit marker move in response to NEXT. Since we cannot yet enter data, the bottom segments will remain off, and the top segments will all be turned on because FUNCTION returns the complement of (D). Try a different initial value for D to see the effect.

LOGIC FUNCTIONS - SUBROUTINE DISPLAY

ADDR		CODE														
8	22 0	E	5		P	U	S	H		H						
	1	2	1		L	X	I		H	,	8	3	F	8		
	2	F	8													
	3	8	3													
8	22 4	0	7		R	L	C									
	5	F	5		P	U	S	H		P	S	W				
	6	D	A		J	C				8	2	3	0			
	7	3	0													
	8	8	2													
	9	7	8		M	O	V		A	,	B					Clear segment (B)
	A	2	F		C	M	A									All ones except (B)
	B	A	6		A	N	A		M							
	C	7	7		M	O	V		M	,	A					
	D	C	3		J	M	P			8	2	3	3			
	E	3	3													
	F	8	2													
8	23 0	7	8		M	O	V		A	,	B					Set segment (B)
	1	B	6		O	R	A		M							
	2	7	7		M	O	V		M	,	A					
8	23 3	F	1		P	O	P		P	S	W					
	4	2	C		I	N	R		L							
	5	C	2		J	N	Z			8	2	2	4			
	6	2	4													
	7	8	2													
	8	E	1		P	O	P		H							
	9	C	9		R	E	T									
	A															
	B															
	C															
	D															
	E															
	F															
3	0															
	1															
	2															
	3															
	4															
	5															
	6															
	7															
	8															

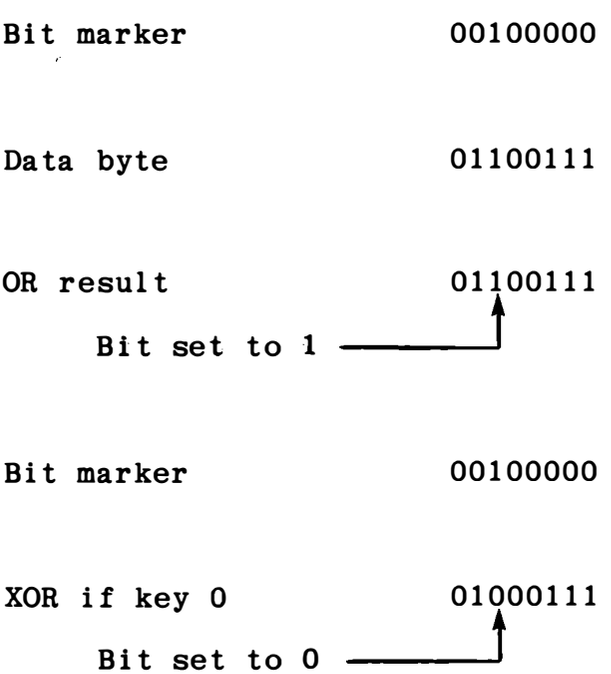
Figure 7-10

7.4.8 Logic Functions DATA Subroutine

This subroutine was defined as follows: Enter the least significant bit of a hex key into the new data byte (D) and display the byte.

The bit marker in (H) identifies the bit position in (D) where the bit is to be entered. We have used one method for entering a bit into a byte, in the DISPLAY subroutine. There we make a conditional jump; if the input bit is 1 we OR the symbol into the display digit; if the input bit is 0 we mask the display with the complemented symbol.

A possibly more efficient procedure is to force the bit to 1 by an OR, and then complement that bit by XOR with the bit marker if the key is zero (leaving the OR result if the key was one):



This procedure is efficient if we can make the decision after the

first OR operation. Since the logic instructions (except CMA) affect all flags, this is a little awkward.

The following procedure avoids any jump instructions, but requires use of an extra register. First, combine the input data with the bit mask:

```
                                    Key 0              Key 1
          RAR                  (Clears Carry)     (Sets Carry)

          SBB    A    (A) = 00000000             11111111
          ANA    H    (A) = 00000000             00010000
                                  |                  |
Location of Bit Marker ___________|__________________|
```

Save this result in another register, and create a reverse mask from the bit marker by complementing it.

```
          MOV   B,A   (B) = 00000000             00010000
          MOV   A,H   (A) = 00010000             00010000
          CMA         (A) = 11101111             11101111
```

AND this with the existing data byte to force the marked bit position to zero; OR the desired bit; and return the new byte to D.

```
                      (D) = 10110010             10110010
          ANA    D    (A) = 10100010             10100010
          ORA    B    (A) = 10100010             10110010
          MOV   D,A   (D) = 10100010             10110010
                                  |                  |
Bit reset or set  ________________|__________________|
```

The revised data byte is now in (D) where it is to be kept, and also in (A), ready to be displayed. Load (B) with the symbol for the new data byte and call DISPLAY.

Reviewing the MAIN program we can see an additional requirement to be placed on DATA. We used:

```
        CC      DATA
        CNC     COMMAND
```

If command is not to be called after a hex key, then DATA must return with Carry set. The 8080 provides an instruction to perform this:

```
37      STC             Set the Carry Flag
                        (CY) < - 1
                        No other flags or registers
                        are affected.
```

This can be placed just before the return from DATA, to inhibit the following CNC COMMAND in the main program.

Code and test the program. You can now enter data with hex keys and move the bit marker with NEXT. The stub for FUNCTION returns the complement of the data entered, so data entered appear in the bottom horizontals and the complements appear in the top.

Figure 7-11 gives the coded subroutine.

LOGICAL FUNCTIONS - SUBROUTINE DATA

ADDR	CODE														
8 24 0	1	F		R	A	R									(B) ← 1 or 0 in current bit position
1	9	F		S	B	B		A							
2	A	4		A	N	A		H							
3	4	7		M	O	V		B	,	A					
4	7	C		M	O	V		A	,	H					(A) ← Data Byte with 0 in current bit position
5	2	F		C	M	A									
6	A	2		A	N	A		D							
7	B	0		O	R	A		B							(A) ← New Data Byte
8	5	7		M	O	V		D	,	A					(D) ← New Data Byte
9	0	6		M	V	I		B	,	0 8					Symbol for New Data = bottom horizontal
A	0	8													
B	C	D		C	A	L	L		D	I	S	P	L	A Y	Display New Data Byte in bottom horizontal
C	2	0													
D	8	2													Mark as key Data
E	3	7		S	T	C									
F	C	9		R	E	T	U	R	N						
8 0															
1															
2															
3															
4															
5															
6															
7															
8															
9															
A															
B															
C															
D															
E															
F															
3 0															
1															
2															
3															
4															
5															
6															
7															
8															

Figure 7-11

7.4.9 Additional Specifications for DATA

We have previously indicated that the bit marker is to be moved only in response to NEXT. You might prefer to move it also whenever a hex key is pressed. If we allow NEXTCOMMAND (at 8280) as an internal alternate entry to COMMAND, it can be called by DATA. An alternative would be for DATA to enter a simulated NEXT command, and clear Carry to force a call to COMMAND.

The 8080 does not provide an explicit "clear Carry" command, but the logic instructions (ORA, ANA, XRA) all clear Carry. We have used XRA A to clear both Carry and the content of A; this works because the exclusive or of a bit with itself is always zero.

$$1 \oplus 1 = 0$$
$$0 \oplus 0 = 0$$

ORA A and ANA A have the effect of clearing Carry and preserving the content of A. They set or reset Zero (and the other flags) according to the content of A; in fact they are exactly equivalent to CPI 00.

Another way of controlling the flags is to compare (A) with itself. CMP A will clear Carry and set Zero without affecting the content of Register A.

Replace the STC instruction at the end of DATA with any of these intructions:

```
BF    CMP   A
B7    ORA   A
A7    ANA   A
```

followed by:

```
3E    MVI   A,15
15
C9    RET
```

Now the bit marker moves in response to hex keys as well as NEXT.

There was some purpose in the original design of DATA, that did not shift the bit marker after a hex key: observation of the effects of the several logic functions is more convenient if you can switch one bit back and forth easily. Since we are using only the least significant bit of a hex key as data, it is possible to define additional bits for other purposes.

0	Enter zero into current bit position
1	Enter one into current bit position
2	Enter zero into current bit position and shift bit marker to next position
3	Enter one into current bit position and shift bit marker to next position

Recall that GETKY returns the key value in both A and C, and neither DATA nor DISPLAY affects Register C (in the given solutions, at least; check your own program designs.)

Set or reset Carry according to bit 1 of the command.

```
MOV   A,C
RAR
RAR
```

This procedure makes Carry the opposite of what we wanted according to the definitions of the hex keys, since 2 and 3 will set Carry which inhibits the CNC command. Another 8080 instruction corrects this:

3F	CMC	Complement Carry (CY) < - $\overline{(CY)}$ No other flags or registers are affected.

The end of DATA then becomes

```
79    MOV  A,C
1F    RAR
1F    RAR
3F    CMC
3E    MVI  A,15
15
C9    RET
```

The completed program appears in Figure 7-12. Write a specification for the subroutine, indicating the function, entry data and return data.

ADDR		CODE	
8 24	0	1F	RAR
	1	9F	SBB A
	2	A4	ANA H
	3	47	MOV B, A
	4	7C	MOV A, H
	5	2F	CMA
	6	A2	ANA D
	7	B0	ORA B
	8	57	MOV D, A
	9	06	MVI B, 08
	A	08	
	B	CD	CALL DISPLAY
	C	20	
	D	82	
	E	79	MOV A, C
	F	1F	RAR
8 25	0	1F	RAR
	1	3F	CMC
	2	3E	MVI A, 15
	3	15	
	4	C9	RET
	5		
	6		
	7		
	8		
	9		REVISED VERSION OF
	A		SUBROUTINE "DATA"
	B		ADVANCES BIT POSITION
	C		IF HEX KEY = 2 OR 3
	D		
	E		
	F		
8	0		
	1		
	2		
	3		
	4		
	5		
	6		
	7		
	8		

Figure 7-12

7.4.10 Logic Functions COMMAND Subroutine

The command keys are interpreted according to the definitions below.

REG	(11)	Set Logic Function ORA
MEM	(10)	Set Logic Function ANA
BRK	(16)	Set Logic Function XRA
CLR	(17)	Set Logic Function CMA
STEP	(13)	Replace Old Data with New Data (E) < - (D)
RUN	(14)	Replace Old Data with Logic Function of Old Data (E) with New Data (D) selected according to (L)
ADDR	(12)	Ignore
NEXT	(15)	Rotate Bit Marker (H)

The sequence above reflects the physical arrangement of the keys. Numerically, keys of value greater than NEXT (15) or less than ADDR (12) are to be stored in (L) as logic function commands. Keys of value greater than ADDR but less than NEXT (STEP = 13 and RUN = 14) replace Old Data. This suggests that we can separate the key commands with three Compare Immediate instructions and five conditional instructions.

```
CPI 15          Set Zero if = NEXT
                Set Carry if < NEXT
CPI 12          Set Zero if = ADDR
                Set Carry if < ADDR
CPI 14          Set Zero if = RUN
```

The coding for subroutine COMMAND to make and act on these tests is indicated below:

```
CPI 15          NEXT
JZ              to Rotate Bit Marker
JNC             to Store Function Selector
CPI 12          ADDR
RZ              Ignore ADDR
JC              to Store Function Selector
CPI 14          RUN
MOV A,D         If not Run, A < - New data
CZ  FUNCTION    If Run, A < - Function


Replace Old Data and Display
MOV  E,A
MVI  B,40
JMP  DISPLAy

Store Function Selector
MOV  L,A
RET

Rotate Bit Marker
MOV  A,H
RRC
MOV  H,A
RET
```

Note that for the first test (CPI NEXT) there are two conditional jumps - each with a completed decision. If neither of these conditions is met, another test is made, followed by two conditional instructions (RZ, JC) for the two completed decisions. The final test (CPI RUN) is only testing for two possibilities - RUN or STEP - since all others have been eliminated. Here we make an assumption about the result - MOV A,D to copy the "new" data into A, to replace "old data" and display it, if the command was STEP. Now we can use

the conditional call (CZ FUNCTION) if in fact the command was RUN. This is permissible because FUNCTION has been defined to return the logic function in Register A, and does not use Register A for input data. Therefore we come to "Replace old Data and Display" with the appropriate value in Register A for either STEP or RUN. Note also that JMP DISPLAY is used instead of CALL DISPLAY, RET.

This subroutine has four exits - RZ for ADDR; JMP DISPLAY for STEP and RUN; RET after Store Function Selector and RET after Rotate Bit Marker. The multiple exits are efficient, because using a single return instruction would require three jumps to reach it. There is a disadvantage to this efficiency. Suppose that in the course of testing the entire program you should find that it occasionally "derails" - it fails to return to the main program. You might want to set a breakpoint at the return from COMMAND to examine the stack. With multiple exits you must enter multiple breakpoints. This is a minor nuisance here, but becomes a substantial problem with bigger programs, many subroutine calls, and extensive usage of the stack.

Write the COMMAND subroutine. A solution is given in Figure 7-13.

With COMMAND in place, we can see the effect of STEP and RUN as well as NEXT. STEP causes the middle horizontal segments to duplicate the bottom segments; RUN causes the middle to duplicate the top. We still have only the one logic function, CMA, in operation, so we cannot readily see the effect of the other command keys. One way to observe them is to store the function selector in memory and set a monitor breakpoint to detect a memory data change at that location. In the stub of FUNCTION, insert SHLD 8300. This will store the

function selector at 8300 and the bit marker at 8301. Set a breakpoint at 8300. Although data will be written there on every pass through the main program loop, the monitor will only detect a change - which occurs only when REG, MEM, BRK or CLR has been pressed. (It will also occur the first time the program is run.)

LOGICAL FUNCTIONS - SUBROUTINE COMMAND

ADDR		CODE														
8 26	0	F	E		C	P	I		N	E	X	T				
	1	1	5													
	2	C	A		J	Z				8	2	7	C			If Command = NEXT
	3	7	C													Go to rotate bit mark)
	4	8	2													
	5	D	2		J	N	C			8	2	7	A			If Command > NEXT
	6	7	A													(= BRK or CLR)
	7	8	2													go to select function
	8	F	E		C	P	I		A	D	D	R				
	9	1	2													
	A	C	8		R	Z										Ignore ADDR
	B	D	A		J	C				8	2	7	A			If Command < NEXT
	C	7	A													(= MEM or REG)
	D	8	2													go to select function
	E	F	E		C	P	I		R	U	N					
	F	1	4													
8 27	0	7	A		M	O	V		A	,	D					(A) ← New Data (STEP)
	1	C	C		C	Z			F	U	N	C	T	I	ON	
	2	A	0													(A) ← Function (RUN)
	3	8	2													
	4	5	F		M	O	V		E	,	A					Replace Old Data
	5	0	6		M	V	I		B	,	4	0				Display Replaced
	6	4	0													Old Data
	7	C	3		J	M	P		D	I	S	P	L	A	Y	
	8	2	0													
	9	8	2													
827	A	6	F		M	O	V		L	,	A					MEM REG BRK CLR
	B	C	9		R	E	T									Replace function selected
827	C	7	C		M	O	V		A	,	H					NEXT -
	D	0	F		R	R	C									Logical Rotate Right
	E	6	7		M	O	V		H	,	A					of Bit Marker
	F	C	9		R	E	T									
3	0															
	1															
	2															
	3															
	4															
	5															
	6															
	7															
	8															

Figure 7-13

7.4.11 Subroutine FUNCTION

Finally we come to the subroutine which performs the basic purpose of this entire exercise. FUNCTION must recognize the selector and perform one of the four logic functions - ORA, ANA, XRA or CMA.

At entry the registers contain:

(D) = new data

(E) = old data

(L) = function selector

Return the selected function of (D) with (E) in register A. Preserve all other registers.

The function selector in L is the key value used to select the function:

(L) = 10 = MEM = ANA

(L) = 11 = REG = ORA

(L) = 16 = BRK = XRA

(L) = 17 = CLR = CMA

Write the subroutine yourself. A solution is given in Figure 7-14, followed by an explanation of this solution.

LOGICAL FUNCTIONS - SUBROUTINE FUNCTION

ADDR		CODE														
82A	0	7D		M	O	L		A	,	L						(A) ← Selector
	1	E6		A	N	I		0	3							00 = MEM
	2	03														40 = REG
	3	0F		R	R	C										80 = BRK
	4	0F		R	R	C										C0 = CLR
	5	87		A	D	D		A								Set/Clear CY and Z
	6	7A		M	O	V		A	,	D						(A) ← New Data
	7	DA		J	C				8	2	B	1				Jump if BRK or CLR
	8	B1														
	9	82														
	A	C2		J	N	Z			8	2	A	F				MEM or REG
	B	AF														Jump if REG
	C	82														
	D	A3		A	N	A		E								MEM = ANA
	E	C9		R	E	T										
82A	F	B3		O	R	A		E								REG = ORA
82B	0	C9		R	E	T										
82B	1	C2		J	N	Z			8	2	B	6				BRK or CLR
	2	B6														Jump if CLR
	3	82														
	4	AB		X	R	A		E								BRK = XRA
	5	C9		R	E	T										
82B	6	2F		C	M	A										CLR = CMA
	7	C9		R	E	T										
	8															
	9															
	A															
	B			E	N	T	R	Y		D	A	T	A			
	C					(	D	)	=	N	E	W		D	A	T A
	D					(	E	)	=	O	L	D		D	A	T A
	E					(	L	)	=	F	U	N	C	T	I	ON SELECTOR
	F															
8	0			R	E	T	U	R	N	S						
	1					(	A	)	=	L	O	G	I	C		FUNCTION
	2															
	3															
	4															
	5															
	6															
	7															
	8															

Figure 7-14

The given solution for FUNCTION achieves its efficiency by setting two flags (Zero and Carry), to distinguish the four selector values. This permits loading A with the "new" data byte before making any jumps. By masking out the unwanted Bits 2 through 7, and rotating Bits 1 and 0 into Bits 7 and 6, the four selector values become:

```
00000000 = MEM = ANA
01000000 = REG = ORA
10000000 = BRK = XRA
11000000 = CLR = CMA
```

Now ADD A shifts Bit 7 into Carry, to distinguish ANA and ORA from XRA and CMA. It also leaves (A) = 00 and sets Zero for MEM and BRK; it leaves (A) = 80, so Not Zero, for ORA and CMA.

A conceivably useful feature is that it returns Carry set if the function is CMA, since that does not affect Carry while ORA, ANA and XRA clear Carry. This information is not used in the program.

This page intentionally left blank.

7.4.12 Exercising Logic Functions

Now with the final program in operation we can experiment with the logic functions and test your knowledge of them. It is particularly instructive to see the results of the functions on identical data bytes and on data bytes which are complementary. Enter some data - say 11000000. Store this value as the old data (STEP). Observe the functions:

REG	(ORA)	11000000
MEM	(ANA)	11000000
BRK	(XRA)	00000000
CLR	(CMA)	00111111

ORA and ANA duplicate the data bytes when the two bytes are identical; XRA gives a zero result. Recall the use of ORA A or ANA A to clear Carry and control Zero without changing the data, and XRA A to clear Register A.

Now store the complement of the data byte (CLR, RUN), and try the functions again.

REG	(ORA)	11111111
MEM	(ANA)	00000000
BRK	(XRA)	11111111

These same values will occur for any data if its complement is stored. Try entering other data, followed by CLR, RUN, REG, MEM, BRK.

We will use the program to test your knowledge of the logic functions.

Problem 1) Enter the data byte 01101100 and store it as old data by pressing STEP. Enter the new data byte 11010101. Before using REG, MEM and BRK, calculate the results yourself and fill in the blanks in Figure 7-15. Then use the program to check your results.

Problem 2) Store the result of XRA by pressing BRK, RUN. Enter new data 00010011. Calculate the next set of results, and again check your answers.

Problem 3) Store the result of XRA from Problem 2. Calculate a new data byte needed to generate the last three results in Figure 7-15.

1)	Enter	0 1 1 0 1 1 0 0
	Store Data (STEP)	
	New Data	1 1 0 1 0 1 0 1
	ORA (REG)	- - - - - - - -
	ANA (MEM)	- - - - - - - -
	XRA (BRK)	- - - - - - - -
2)	Replace old data with XRA (press BRK, RUN)	
	Old Data	- - - - - - - -
	New Data	0 0 0 1 0 0 1 1
	ORA (REG)	- - - - - - - -
	ANA (MEM)	- - - - - - - -
	XRA (BRK)	- - - - - - - -
3)	Replace old data with XRA (press BRK, RUN)	
	Old Data	- - - - - - - -
	New Data	- - - - - - - -
	ORA (REG)	1 0 1 1 1 1 1 1
	ANA (MEM)	1 0 0 0 0 0 1 0
	XRA (BRK)	0 0 1 1 1 1 0 1

Logic Functions - Self Test

Figure 7-15

7.5 FLOW CONTROL TECHNIQUES

In the logic functions exercise we saw two schemes to decide which of several possible actions to take, based on a data byte from the keyboard. In the COMMAND subroutine we used numeric comparisons:

```
CPI     NEXT
JZ      to rotate bit mark
JNC     to select logic function
CPI     ADDR
RZ
JC      to select logic function
CPI     RUN
```

In the FUNCTION subroutine we shifted control bits and used JC and JNZ instructions:

```
MOV     A,L
ANI     03
RRC
RRC
ADD     A
JC      to BRK or CLR
JNZ     to REG
```

These were reasonably efficient because the numeric values of the control bit patterns had convenient relationships. If the key definitions had been random it might have been necessary to use seven CPI, JZ segments in COMMAND.

It is possible to use a directory, or "dispatch table" instead of such a procedure. The command, or control pattern, is added to a table address. This locates a memory byte where we have stored another address. This is just like the directory procedure we used in the sensor correction programs of Chapters 4 and 6. In this case, however, we want to jump to the address obtained from the table, rather than using it to find more data.

If register pair HL is not in use for other data, it is very convenient to use it with a dispatch table, as we did with the sensor correction directory.

If all of the program segments to which we might jump are in the same memory page as the dispatch table, we can use single byte indirect addressing:

```
        LXI     H, TABLEADDRESS
        ADD     L
        MOV     L,A
        MOV     L,M
```

This has loaded into register pair HL the address to which we will jump. Recall the indirect jump instruction:

```
E9      PCHL        Jump to the address contained
                    in register pair HL.
                    (PC) < - (HL)
                    No flags are affected.
```

If we do not want to use register pair HL, but do have another pair available, we can use this technique:

```
        LXI     B, TABLEADDRESS
        ADD     C
        MOV     C,A
        LDAX    B
        MOV     C,A
```

This has loaded the jump address into register pair BC. There is no "PCBC" instruction, but we can use the stack.

```
PUSH    B               (ST) < - Address
RET                     Jump to (ST)
```

Here we place the address into the stack top, and a RET jumps to that address.

A third method uses HL and the stack.

```
        PUSH    H           Save (HL)
        LXI     H, TABLEADDRESS
        ADD     L
        MOV     L,A
        MOV     L,M
```

As in the first method, we have loaded the address into HL. Now we can recover the data that we saved, and put the jump address into the stack.

```
        XTHL            Exchange stack top with HL
        RET             Jump to (ST)
```

Any of these techniques can be used, with only slightly more

complexity, if two byte indirect addressing is needed.

When a dispatch table is used for MTS command keys, remember that these keys return the values 10-17. Therefore, we must either subtract 10 from the command before adding it to the table address or, more efficiently, load the register pair with an address 10 hex bytes below the actual table location.

Recall that subroutine GETKY returns with Register B cleared and the key in C as well as in Register A. This is designed to make the use of dispatch tables easy.

```
PUSH    H
LXI     H, DISPATCHTABLE -10
DAD     B
MOV     L,M
XTHL
RET
```

(Monitor subroutines ENTBY and ENTWD similarly return with Register B cleared and the command key in C as well as A.) This technique can be used in the logic functions program COMMAND subroutine. Change the specification of COMMAND to require that (B) = 00 and (C) = command key. (This change requires a change in DATA.) Rewrite COMMAND to use a dispatch table. A solution is shown in Figure 7-16.

REVISED SUBROUTINE DATA

ADDR	CODE				
824 0	1F		RAR		
1	9F		SBB	A	
2	A4		ANA	H	
3	47		MOV	B, A	
4	7C		MOV	A, H	
5	2F		CMA		
6	A2		ANA	D	
7	B0		ORA	B	
8	57		MOV	D, A	
9	06		MVI	B, 08	
A	08				
B	CD		CALL	DISPLAY	
C	20				
D	82				
E	79		MOV	A, C	
F	1F		RAR		
825 0	1F		RAR		
1	3F		CMC		
2	01	*	LXI	B, 0015	
3	15	*			
4	00	*			
5	C9	*	RET		
6					
7					
8					
9			REVISED VERSION		
A			SUBROUTINE "DATA"		
B			FOR LOGIC FUNTIONS		
C			PROGRAM OF FIGURE 7-7		
D					
E			PUTS "NEXT" COMMAND		
F			IN (BC) TO ADVANCE		
80			BIT POSITION		
1			IF HEX KEY = 2 OR 3		
2					
3		*	NOTE CHANGES		
4					
5					
6					
7					
8					

Figure 7016a

REVISED SUBROUTINE COMMAND

ADDR		CODE													
8	26	0	E5			F	U	S	H		H					
		1	21			L	X	I		H	,	D	I	S	P	ATCHTABLE-10
		2	59													
		3	82													
		4	09			D	A	D		B						
		5	6E			M	O	V		A	,	D				
		6	E3			X	T	H	L							
		7	7A			M	O	V		A	,	D				(A) ← New Data (STEP)
826		8	C9			R	E	T								
826		9	7A				M	E	M							DISPATCHTABLE
		A	7A				R	E	G							
		B	68				A	D	D	R						Ignore
		C	74				S	T	E	P						Old Data ← New Data
		D	71				R	U	N							Old Data ← Function
		E	7C				N	E	X	T						Shift Bit Mark
		F	7A				B	R	K							
8	27	0	7A				C	L	R							
827		1	CD	*		C	A	L	L		F	U	N	C	T	ION (RUN)
		2	A0													
		3	82													
827		4	5F			M	O	V		E	,	A				Replace old data
		5	06			M	V	I		B	,	40				
		6	40													
		7	C3			T	M	P			D	I	S	P	L	AY
		8	20													Display replaced
		9	82													Old Data
827		A	69	*		M	O	V		L	,	C				MEM REG BRK CLR
		B	C9			R	E	T								Replace Function Select
827		C	7C			M	O	V		A	,	H				NEXT
		D	0F			R	R	C								Logical Rotate Right
		E	67			M	O	V		H	,	A				of Bit Marker
		F	C9			R	E	T								
8		0														
		1														
		2				R	E	V	I	S	E	D		C	O	MMAND
		3					S	U	B	R	O	U	T	I	N	E FOR
		4					L	O	G	I	C		F	U	N	CTIONS
		5				E	N	T	E	R		W	I	T	H	
		6					(	C	)	=	C	O	M	M	A	ND
		7					(	B	)	=	00					
		8			*	N	O	T	E		C	H	A	N	G	ES

Figure 7-16b

7.6 REVIEW AND ADDITIONAL EXERCISES

The logic and bit manipulation techniques taught in this chapter are most commonly used for control operations and decision making. The additional exercises suggested in the following sections simulate some control applications.

We have introduced four types of instructions:

Arithmetic and Logical Rotate - RAR, RAL, RRC, and RLC, and the arithmetic intructions ADD A, ADC A and DAD H that have related properties.

Logic Functions - ORA, ANA and XRA, which combine two data bytes by the OR, AND and Exclusive OR rules; also CMA which complements (A) without involving another data byte.

Flag Control Instructions - STC and CMC, plus the logic and arithmetic instructions that can be used to control flags - ORA A, ANA A, XRA A, CMP A.

Masking - The use of ANI to mask (discard) unwanted bits in a byte used for control functions.

The exercises of this chapter have also given practice in important flow control techniques: the IF-THEN-ELSE construct; the use of conditional calls and returns; sequential testing procedures; and dispatch tables. We saw the use of making assumptions before executing a conditional jump, call or return.

Once again we saw the convenience of top-down programming and

subroutines, with stubs for incomplete subroutines. We passed arguments to subroutines. This is especially noticeable in the DISPLAY subroutine, where we placed various data bytes in (A) and a symbol in (B), but all of the subroutines in the exercise of Section 7.4 involved passing arguments.

Finally, we again used features that are specific to the ICS Microcomputer Training System -- the monitor subroutines DWORD and ENTWD in Section 7.1, and GETKY in the later exercises; the display system; and the use of a breakpoint to detect a change in memory content in Section 7.4.10.

It is recommended that you work out at least one of the exercises in the following four sections to obtain additional experience. Glance through all of the descriptions before choosing which you will pursue.

7.6.1 Traffic Control Exercise

Develop a simulator for a street intersection traffic light controller. This can use the same display subroutine and much of the same main program as the logic functions program.

Traffic lights are simulated by horizontal segments in the display. A top segment represents a red light, middle segment a yellow light, and a bottom segment a green light. Allow two lights to appear at the same time by initializing the bit marker (H) to 10000001 (81). Let (D) represent green lights and (E) represent yellow lights. Initialize (D) to 80, to start with one green light.

We no longer want to display the bit marker; it is convenient to display the green light where previously we displayed the bit marker. The display of the logic functions can be retained to display the red light instead.

Different subroutines are called for FUNCTION and COMMAND. These are defined as follows:

Subroutine REDS (replaces FUNCTION)

Function:

From given yellow and green lights, return other lights as red.

Entry Address: 82D0

Entry Data:

(D) = Green Lights (E) = Yellow Lights (H) = Light Positions

Return Data:

(A) = Red Lights

Registers:

All registers except (A) are preserved.

Constraints:

Entry of data to Register D without properly modifying the content of E may cause an improper condition of both lights being the same color.

Comment: The CC DATA instruction has been retained in the main program to permit forcing an error into Register D. Test your program initially without any error protection in subroutine REDS.

Subroutine SWITCH (replaces COMMAND)

Function:

Change any green light to yellow. If a light was previously yellow, change the other light to green, and turn off the yellow light.

Entry Address: 82C0

Entry Data:

(D) = green lights

(E) = yellow lights

(H) = light positions

Return Data:

(D) = new green lights

(E) = new yellow lights

Registers:

A, D and E are affected.

B, C, H and L are preserved.

Constraints:

It is assumed that the main program will display red and green lights.

Subroutines DISPLAY and DATA from the logic functions exercise are also required.

In this version of the program the lights only change in response to command keys. In Section 7.6.2 a timer will be introduced. It is suggested that you copy the changes of Figure 7-17a into the main program of Section 7.4, but develop subroutines SWITCH and REDS yourself.

TRAFFIC CONTROL - MAIN

ADDR		CODE													
8	200	0	11			LXI		D,	8000						Initialize
		1	00												(E) ← Yellow Lights
		2	80	*											(D) ← Green Lights
		3	21			LXI		H,	8117						TO
		4	17												(L) ← Time to Change
		5	81	*											(H) ← Light Positions
		6	7A	*		MOV		A,	D						Display Green Lights
		7	06			MVI		B,	08						in Position
		8	08	*											horizontal
		9	CD			CALL			DISPLAY						
		A	20												
		B	82												
		C	CD			CALL			REDS						
		D	D0	*											(A) ← Red Lights
		E	82												
		F	06			MVI		B,	01						Display Red Lights
8	210	0	01												in top horizontal
		1	CD			CALL			DISPLAY						
		2	20												
		3	82												
		4	CD			CALL			GETKY						
		5	3D												
		6	02												
		7	DC			CC			DATA						
		8	40												
		9	82												
		A	D4			CNC			SWITCH						
		B	C0	*											
		C	82												
		D	C3			JMP			8206						
		E	06												
		F	82												
8	3	0													
		1													
		2			*	CHANGES FROM									
		3				LOGIC FUNCTIONS PROGRAM									
		4													
		5													
		6													
		7													
		8													

Figure 7-17a

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

ADDR		CODE																
8 2C	0	7	B		M	O	V		A	,	E				(A) ← Yellow Lights
	1	5	A		M	O	V		E	,	D				Change green to yellow
	2	A	4		A	N	A		H						Were any yellow?
	3	C	A		J	Z				8	2	C	7		If not go to
	4	C	7												turn off all
	5	8	2												green lights
	6	A	C		X	R	A		H						Else change the
82C	7	5	7		M	O	V		D	,	A				reds to green
	8	7	B		M	O	V		A	,	E				
	9	0	6		M	V	I		B	,	4	0			
	A	4	0												
	B	C	D		C	A	L	L		D	I	S	P	L	A Y
	C	2	0												
	D	8	2												
	E	C	9		R	E	T								
	F														
8 2D	0	7	A		M	O	V		A	,	D				REDS – returns
	1	B	3		O	R	A		E						(A) = all that are
	2	A	C		X	R	A		H						neither green
	3	C	9		R	E	T								nor yellow
	4														
	5														
	6				N	O	T	E	:		D	I	S	P	LAY AND DATA
	7				S	U	B	R	O	U	T	I	N	E	S ARE ALSO
	8				R	E	Q	U	I	R	E	D.			SEE
	9						F	I	G	U	R	E		7	-10 DISPLAY
	A						F	I	G	U	R	E		7	-12 DATA
	B														
	C														
	D														
	E														
	F														
8	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure 7-17b

7.6.2 Extended Traffic Control Exercise

Elaborate the traffic control program of Section 7.6.1 in the following ways.

7.6.2.1

Revise subroutine REDS to protect against an error that sets both lights green at once. If such an error occurs, correct it by modifying the content of (D).

7.6.2.2

Replace the CALL GETKY instruction with a call to a time delay subroutine. This should set a relatively short delay for a yellow light; a longer delay for a green light. Review the discussion of time delays in Section 4.8.6 if necessary.

7.6.2.3

Replace the time delay subroutine with one that tests the keyboard during the time delay. If a key is pressed, call GETKY and return without completing the time delay. The monitor subroutine SCAN (0257) reads the keyboard once: if no key is pressed it returns Not Carry and (A) = 00; if a key is pressed it returns Carry set and the key value in Register A. SCAN takes a relatively long time; reduce your time delay count to compensate for this. This subroutine is shown in Figure 7-18. It permits you to change the lights at will, instead of waiting for the time delay.

7.6.2.4

Revise the traffic control program function to simulate a triggered traffic controller. This will normally keep the main street traffic light (the left hand digit) green, and the side street traffic light (the right hand digit) red. When a key is pressed, call SWITCH and a time delay four times, to allow side street traffic to flow. This can best be done by having the main program call a new subroutine intead of SWITCH.

TIMER AND KEYBOARD SCANNER

CODING SHEET

ADDR		CODE													
82E	0	01		L	X	I		B	,	1	0	0	0		Set Time for Yellow Light
	1	00													
	2	10													
	3	7B		M	O	V		A	,	E					Any yellow?
	4	A4		A	N	A		H							
	5	C2		J	N	Z			8	2	E	A			
	6	EA													
	7	82													
	8	06		M	V	I		B	,	4	0				Set Time for Green Light
	9	40													
82E	A	CD		C	A	L	L		S	C	A	N			Test for Key
	B	57													Sets CY if Key
	C	02													
	D	DA		J	C				8	2	F	7			If key pressed
	E	F7													go to read Key
	F	82													
82F	0	0B		D	C	X		B							
	1	78		M	O	V		A	,	B					
	2	B1		O	R	A		C							
	3	C2		J	N	Z			8	2	E	A			
	4	EA													
	5	82													
	6	C9		R	E	T									
82F	7	CD		C	A	L	L		G	E	T	K	Y		
	8	3D													
	9	02													
	A	C9		R	E	T									
	B														
	C														
	D			T	H	I	S		S	U	B	R	O		UTINE TO BE
	E			C	A	L	L	E	D		I	N	S		TEAD OF
	F			G	E	T	K	Y	,		A	T			824
8	0														
	1			I	F		N	O		K	E	Y			PRESSED
	2			R	E	T	U	R	N	S		A	F		TER TIMEOUT
	3			W	I	T	H		Z	E	R	O	,		NOT CARRY
	4			A	N	D		(	A	)	=	(	B	)	-(C) = 00
	5														
	6			I	F		K	E	Y		I	S			PRESSED
	7			R	E	T	U	R	N	S		A	S		GETKY
	8														

Figure 7-18

7.6.3 Fire and Burglar Alarm

Let the keys 0, 1, 2 and 3 represent two fire (or smoke) detectors and two burglar alarm sensors. If a fire is detected, flash the message FIRE in the display repeatedly. If a burglar is detected, flash the message POLICE. If both are detected, alternate the two displays.

Accept some sequence of the higher digits (4 through F) to simulate a combination lock used for an authorized entry, and turn off any alarm. If a wrong sequence is entered, or a long delay occurs between keys, call the police.

7.6.4 Model Railroad Simulator

If at this point you want to undertake a much more difficult program, simulate a model railroad in the display. Represent a train by a string of segments following each other around a track. Represent switches by the decimal point indicators. These can be set or reset by hex keys 0 through 7. The following rules are suggested for train control.

a) When a train is moving on the bottom track and encounters a switch which is set, it turns up to the middle track, where it resumes its previous direction.

b) When a train is moving on the middle track, and sees a switch set, it turns toward the bottom track where it resumes its previous direction.

c) If a train is moving on the top track it ignores the

indicated switches. If one of the hex keys 8 through F is being held down when the train reaches the corresponding position, then the train turns toward the bottom track. If it encounters a set switch, then it resumes its previous leftward or rightward direction. (This will reverse its clockwise or counter-clockwise direction.) If the train encounters a switch which is not set it must stop until the switch is set.

This program is difficult and lengthy. Do not undertake it unless you want a real challenge.

This page intentionally left blank.

INTEGRATED COMPUTER SYSTEMS

EDUCATION IS OUR BUSINESS™

NORTH AMERICAN HEADQUARTERS
Integrated Computer Systems, Inc.
3304 Pico Boulevard
P.O. Box 5339
Santa Monica, California 90405 USA
Telephone: (213) 450-2060
TWX: 910-343-6965

NORTH AMERICA – EASTERN REGION
Integrated Computer Systems, Inc.
300 North Washington Street
Suite 103
Alexandria, Virginia 22314 USA
Telephone: (703) 548-1333
TWX: 710-832-0045

EUROPEAN HEADQUARTERS
ICSP – U.K.
Pebblecoombe, Tadworth
Surrey KT20 7PA
England
Telephone: Leatherhead (03723) 79211
Telex: 915133

FRANCE
ICS France
90 Ave Albert 1er
92500 Rueil-Malmaison
France
Telephone: (01) 749 40 37
Telex: 204593

GERMANY
ICSD GmbH
Leonrodstrabe 54
8000 Munich 19
West Germany
Telephone: (089) 19 80 66
Telex: 5215508

SCANDINAVIA
ICSP Inc. – Scandinavia
Utbildningshuset AB
Box 1719
S-221 01 Lund, Sweden
Telephone: (046) 30 70 70
Telex: 33345