INTEGRATED COMPUTER SYSTEMS, INC.™

MICROCOMPUTER TRAINING SYSTEM

WORKBOOK

EDWARD DILLINGHAM, M.E., M.S.E.E.

ASSISTED BY

Dr. David C. Collins
Mr. Eric R. Garen
Dr. Daniel M. Forsyth

Published By

INTEGRATED COMPUTER SYSTEMS, INC.

LOS ANGELES BRUSSELS

Rev. A 6/78

TABLE OF CONTENTS

INTEGRATED COMPUTER SYSTEMS MICROCOMPUTER TRAINING SYSTEM CODING SHEET

ADDR	CODE	
8200		
01		
02		
03		
04		
05		
06		
07		
08		
09		
0A		
0B		
0C		
0D		
0E		
0F		
8210		
11		
12		
13		
14		
15		
16		
17		
18		
19		
1A		
1B		
1C		
1D		
1E		
1F		
8220		
21		
22		
23		
24		
25		
26		
27		
28		

INTEGRATED COMPUTER SYSTEMS MICROCOMPUTER TRAINING SYSTEM CODING SHEET

ADDR	CODE	
80		
81		
82		
83		
84		
85		
86		
87		
88		
89		
8A		
8B		
8C		
8D		
8E		
8F		
90		
91		
92		
93		
94		
95		
96		
97		
98		
99		
9A		
9B		
9C		
9D		
9E		
9F		
A0		
A1		
A2		
A3		
A4		
A5		
A6		
A7		
A8		

MICROCOMPUTER TRAINING WORKBOOK

CHAPTER 1

HARDWARE AND SOFTWARE FUNDAMENTALS

INTRODUCTION TO CHAPTER I

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

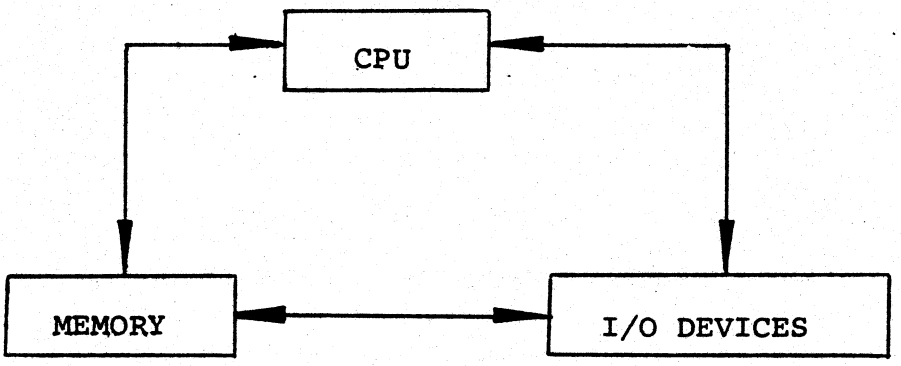
A computer has three principle hardware subsystems: a Central Processing Unit (CPU), a memory, and Input/Output (I/O) devices.

> CPU: The central processing unit, a set of elements which perform the actual arithmetic and logical operations. The CPU also serves 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 which provide input of data and instructions to the system and output of results, for example keyboards for input and displays for output.

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 I/O device and stored in memory.
2. Data are input via an I/O 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 I/O device.

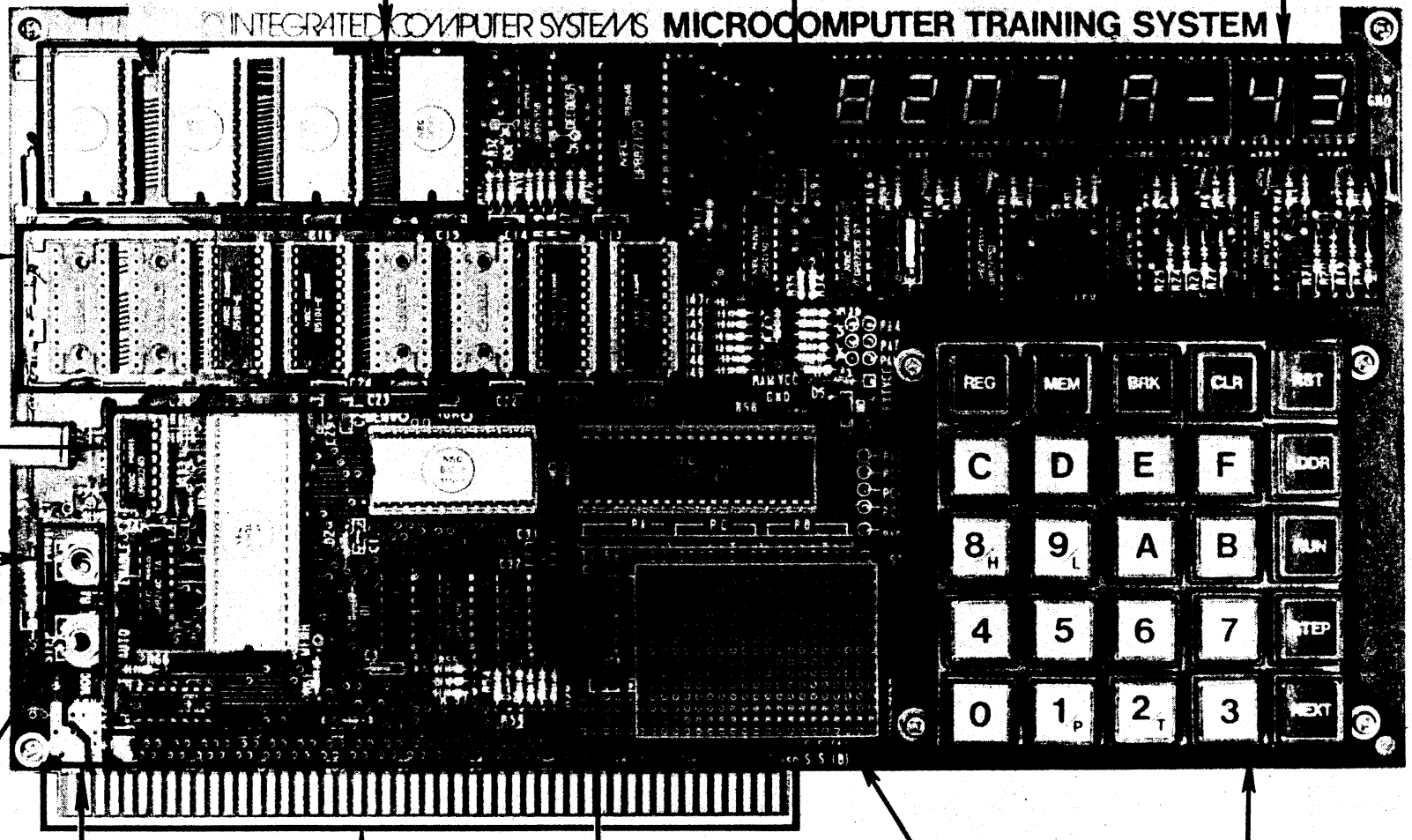
In Figure 1-1, showing the layout of the MTS computer, the principal subsystems have been identified: The CPU, Memory, and Keyboard and Display. We will look at these in more detail later in the chapter.

INTEGRATED COMPUTER SYSTEMS, INC.

MEMORY
1024 bytes of Electrically Eraseable PROM memory containing ICS Educational Monitor

DMA
Direct Memory Access (DMA) and timing circuits

DISPLAY
8-digit, 7-segment LED display



MEMORY
Space for 1024 bytes of CMOS RAM memory - 512 bytes provided with system

PROCESSOR HARDWARE
8080 microprocessor plus 8228 system controller and clock circuit

SWITCH (A)
provides the option to switch power supply mode to two user-supplied 1.5 volt dry cells. This permits retention of data in CMOS RAM memory.

SWITCH (B)
provides the option of operating the system in a hardware-generated single-step mode or in a free-running mode.

POWER SUPPLY CONNECTION
the system requires a simple external supply of +5 volts (at 1 amp) and +12 volts (at 0.2 amp) —user supplied

EDGE CONNECTOR
permits interfacing to external devices and expansion of memory (CPU address, data control buses are made available at board-edge pins)

PROGRAMMABLE PERIPHERAL INTERFACE
provides 3 programmable 8-bit I/O ports (can be programmed to provide two serial I/O ports for asynchronous transmit and receive - ICS Monitor handles all transmit/receive functions)

FREE AREA
provided for hardware additions by user

KEYBOARD
25-key keyboard (16 hex keys and 9 function keys)

Fig. 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, and the sequence might appear as follows:

```
START
INPUT NUMBER OF MILES TO BE CONVERTED
STORE IN MEMORY UNDER (MILES)
RETRIEVE (MILES) FROM MEMORY
RETRIEVE (8) FROM MEMORY
MULTIPLY (MILES) TIMES (8)
STORE IN MEMORY UNDER (TEMPORARY)
RETRIEVE (TEMPORARY) FROM MEMORY
RETRIEVE (5) FROM MEMORY
DIVIDE (TEMPORARY) BY (5)
STORE IN MEMORY UNDER (RESULT)
OUTPUT (RESULT)
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.

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 so.

-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

To physically represent 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.

For reasons of reliability and cost, such multi-state representations are impractical in the various types of electronic circuitry required by computer systems. A reliable and practical representation is a two-valued state, which may be realized by the use of two different voltage levels, by the state of a gate or flip-flop which is either open or closed, or by the positive or negative polarity of a magnetic element. In all cases, however, the computer operates on these two states logically as representing ones and zeros. Computers, therefore, use a two-state binary number system to represent numbers.

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

To understand the basic principles of computer operation, it is essential to know something about number systems in general, and about 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 quickly jot down the products which are to be summed.

Conversion from decimal to binary could also be accomplished by using a table, but it is much easier to use a process which we may call "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:

Quotient	Remainder
57/2 = 28	1 — position 0
28/2 = 14	0 — 1
14/2 = 7	0 — 2
7/2 = 3	1 — 3
3/2 = 1	1 — 4
1/2 = 0	1 — 5

1 1 1 0 0 1

Decimal 57 is the equivalent of binary 111001. We may check this by quickly jotting down the products, counting from position 0: (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

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 addition (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 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 <u>two</u>) the length of the previous cycle. For decimal numbers each cycle will be ten times (base <u>ten</u>) 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 really nothing more than 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 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 string 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. In the decimal system, digits are often grouped by threes, separated by commas (e.g. 862,249,101). Consider some possible groupings of the bits in our example:

10110001 00001001	(grouped by 8 bits)
1011 0001 0000 1001	(grouped by 4 bits)
10 11 00 01 00 00 10 01	(grouped by 2 bits)

A group of eight bits can represent one of 256 numbers ranging from 00000000_2 to 11111111_2, or from 0_{10} to 255_{10} (the reader is asked to verify that this is so by converting 11111111_2 to a decimal number). This is considerably less than the $65{,}536_{10}$ numbers which can be represented by a group of sixteen bits, but is still too large (256 different names?) to be useful. A two bit group, on the other hand, can represent only four numbers, and is too small to be useful. A four bit grouping, representing sixteen possible numbers, seems both visually satisfactory (look at the groupings again) and reasonable. What we need is a set of sixteen symbols to represent each of the different numbers, and these are given in the following table:

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

By adding the first six letters of the alphabet to the ten existing decimal symbols we are able to unambiguously name each unique group of four bits. 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.

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 which are contained in a word.

bit(N-1)..... bit 0

A memory 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 lead to the adoption of a special term for an 8-bit word or string, the byte.

BYTE: An 8-bit word. More generally, an 8-bit string.

1 0 1 1 0 1 0 1

A byte representing the number 181_{10} (or $B5_{16}$).

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

MEMORY LOCATION: The location 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.

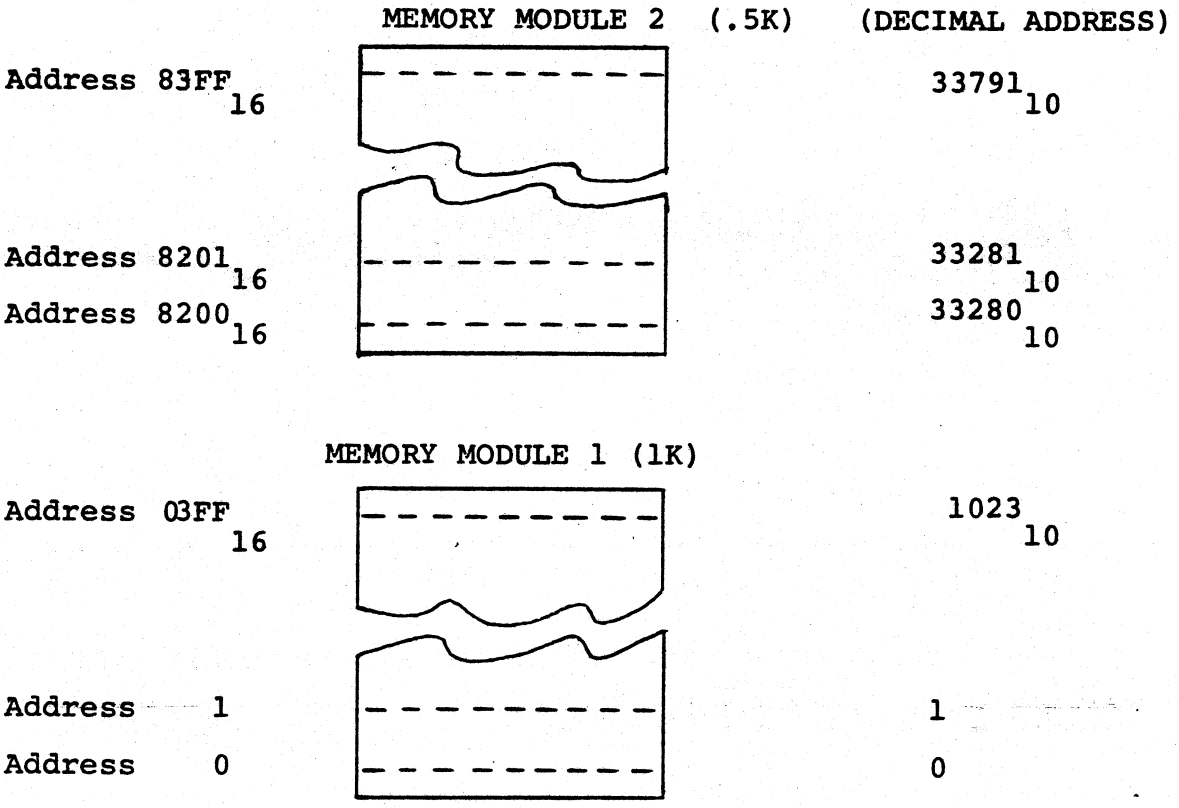
ADDRESS SIZE: The total number of bits which may be used to specify a memory address.

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_{10} (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:



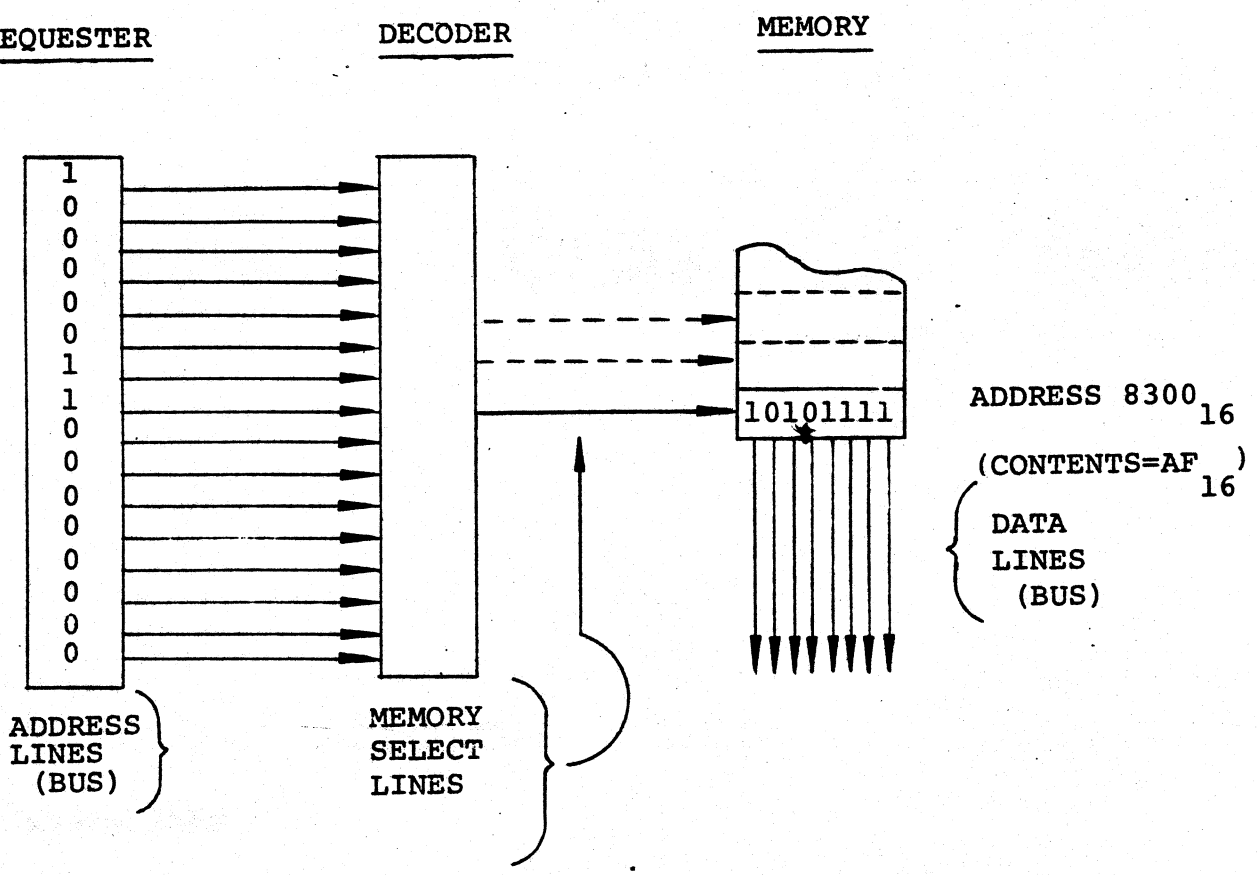
The diagram describes the memory structure of a system with a word size of eight bits, an address size of sixteen bits (Why are sixteen bits required?), and a memory size of 1.5K words. It is in fact the memory structure of your own MTS computer system. Two important properties of memory organization 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 looks at the pattern of a set of input signals and selects an output signal determined by that pattern.

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

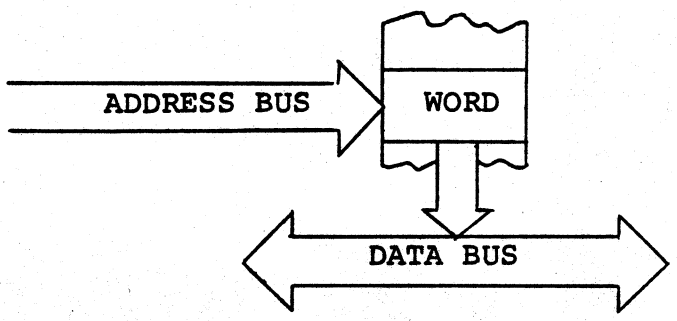
There are two types of memory in your MTS computer system: Random Access Memory (RAM), which may be read or written, and Read Only Memory (ROM), from which data may be read but not written into. To read data from memory, the address bus is used to select a word whose contents can then be read out onto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

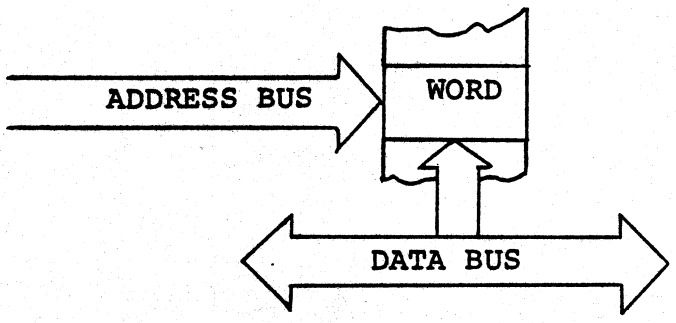
Read and write operations are illustrated in the following diagram:

RAM OR ROM MEMORY



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

RAM MEMORY ONLY



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

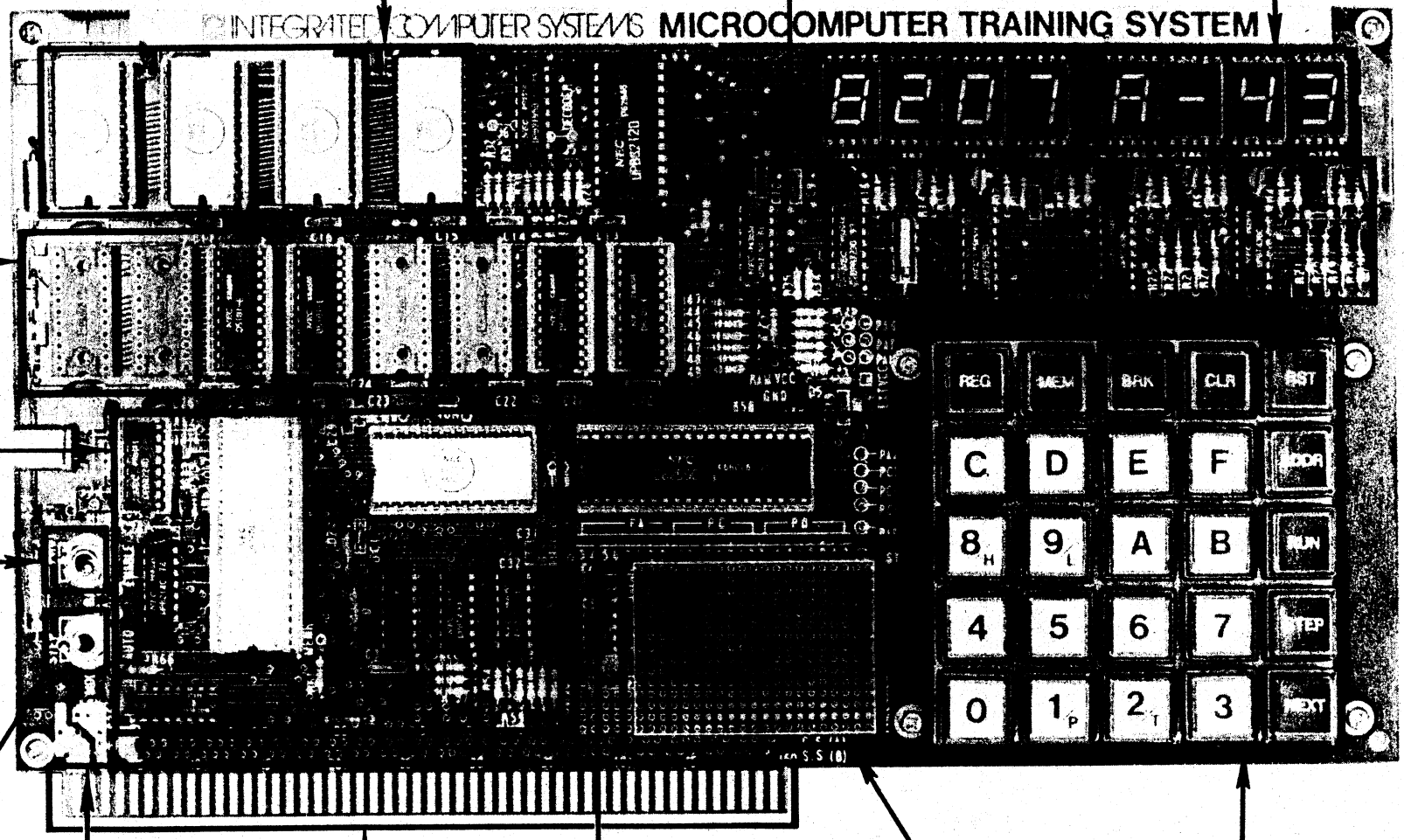
In Figure 1-2 the RAM and ROM memories of your MTS system are indicated. There are 512_{10} words of RAM and 1024_{10} words of ROM memory. Your ROM contains a set of programs called the MONITOR, designed to assist you in learning the system. The functions of the MONITOR will be defined step by step as you progress through this manual. The RAM memory will be used to store the different programs which you will write yourself. ROM memories are used for programs which do not need to be changed, and are protected against inadvertent modification. RAM memories are used for program development (these programs can then be placed in a ROM memory, but special equipment is required) and for storage of transient data in actual applications.

INTEGRATED COMPUTER SYSTEMS, INC.

MEMORY
1024 bytes of Electrically Eraseable PROM memory containing ICS Educational Monitor

DMA
Direct Memory Access (DMA) and timing circuits

DISPLAY
8-digit, 7-segment LED display



MEMORY
Space for 1024 bytes of CMOS RAM memory - 512 bytes provided with system

PROCESSOR HARDWARE
8080 microprocessor plus 8228 system controller and clock circuit

SWITCH (A)
provides the option to switch power supply mode to two user-supplied 1.5 volt dry cells. This permits retention of data in CMOS RAM memory.

SWITCH (B)
provides the option of operating the system in a hardware-generated single-step mode or in a free-running mode.

POWER SUPPLY CONNECTION
the system requires a simple external supply of +5 volts (at 1 amp) and +12 volts (at 0.2 amp) —user supplied

EDGE CONNECTOR
permits interfacing to external devices and expansion of memory (CPU address, data control buses are made available at board-edge pins)

PROGRAMMABLE PERIPHERAL INTERFACE
provides 3 programmable 8-bit I/O ports (can be programmed to provide two serial I/O ports for asynchronous transmit and receive - ICS Monitor handles all transmit/receive functions)

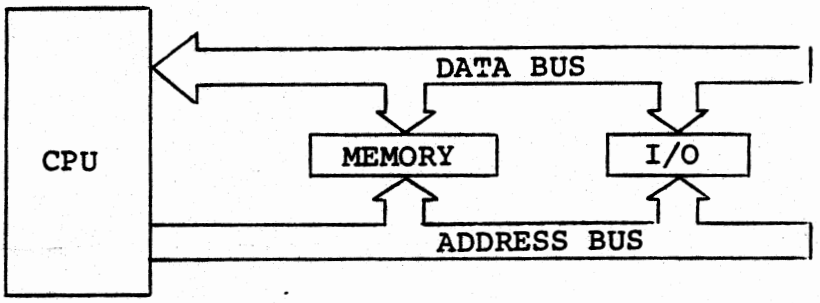
FREE AREA
provided for hardware additions by user

KEYBOARD
25-key keyboard (16 hex keys and 9 function keys)

Fig. 1

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 in this chapter, but first we may review the structure of the system including the data bus and address bus:

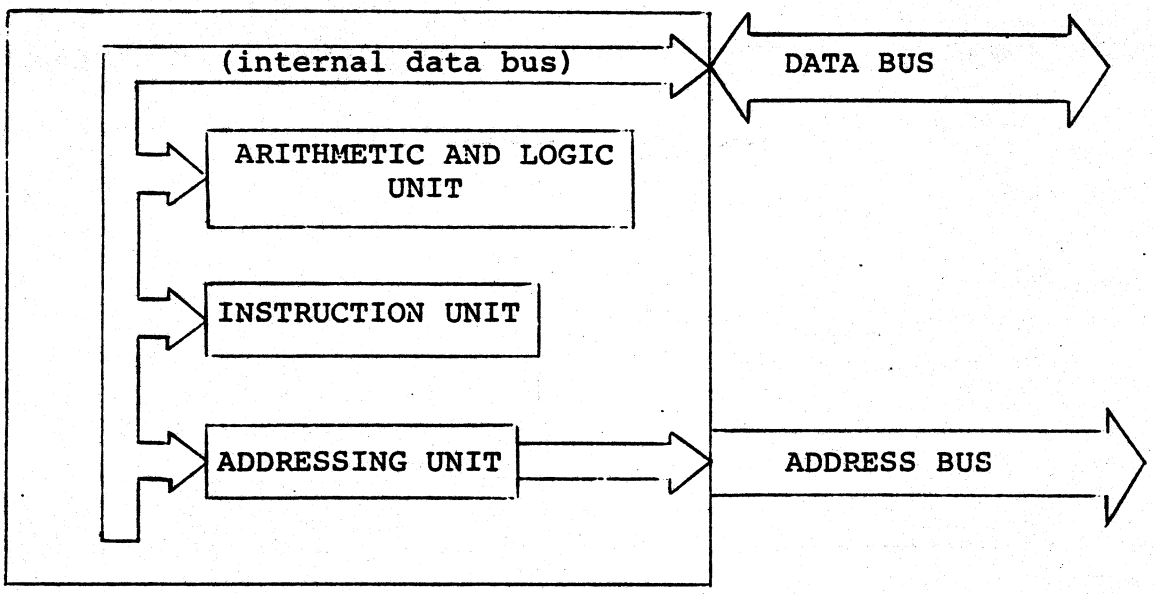


The CPU may send or receive data along the data bus (it is bidirectional), but no memory address is sent to the CPU along the address bus.

1.4.1 Functional Units

Internally, the CPU consists of four 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 Logic 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 words 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 register 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 instructs the computer to perform a specific operation.

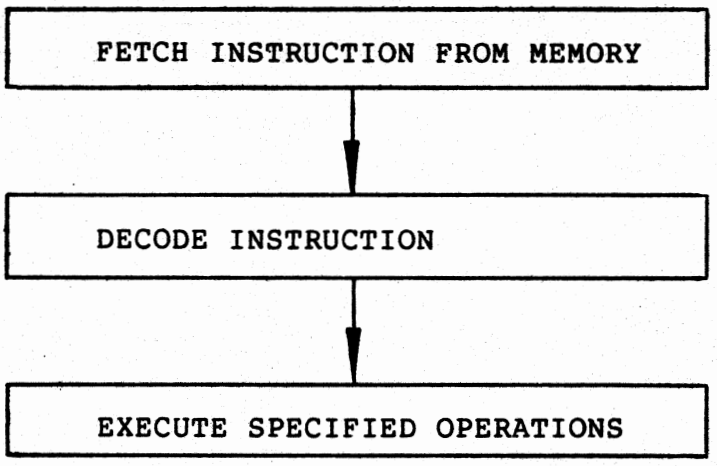
Each execution of an instruction will perform one small step in the calculation or process which the program is designed to accomplish. In turn, the execution of each instruction is broken up into a number of steps which are performed one after another.

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:

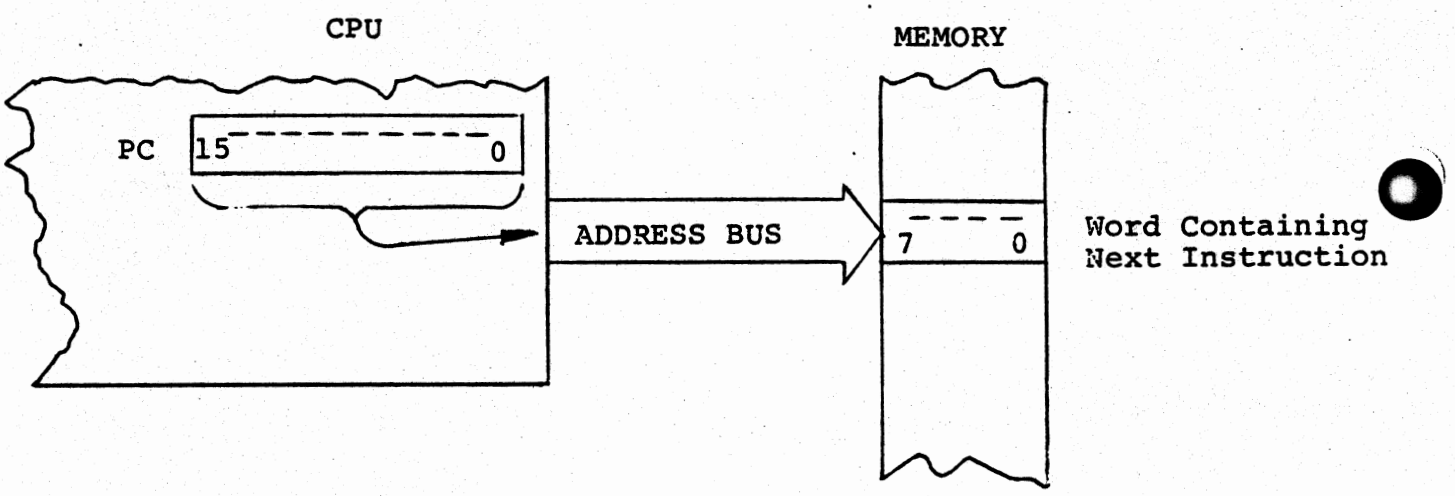


1.4.4 The Program Counter

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.

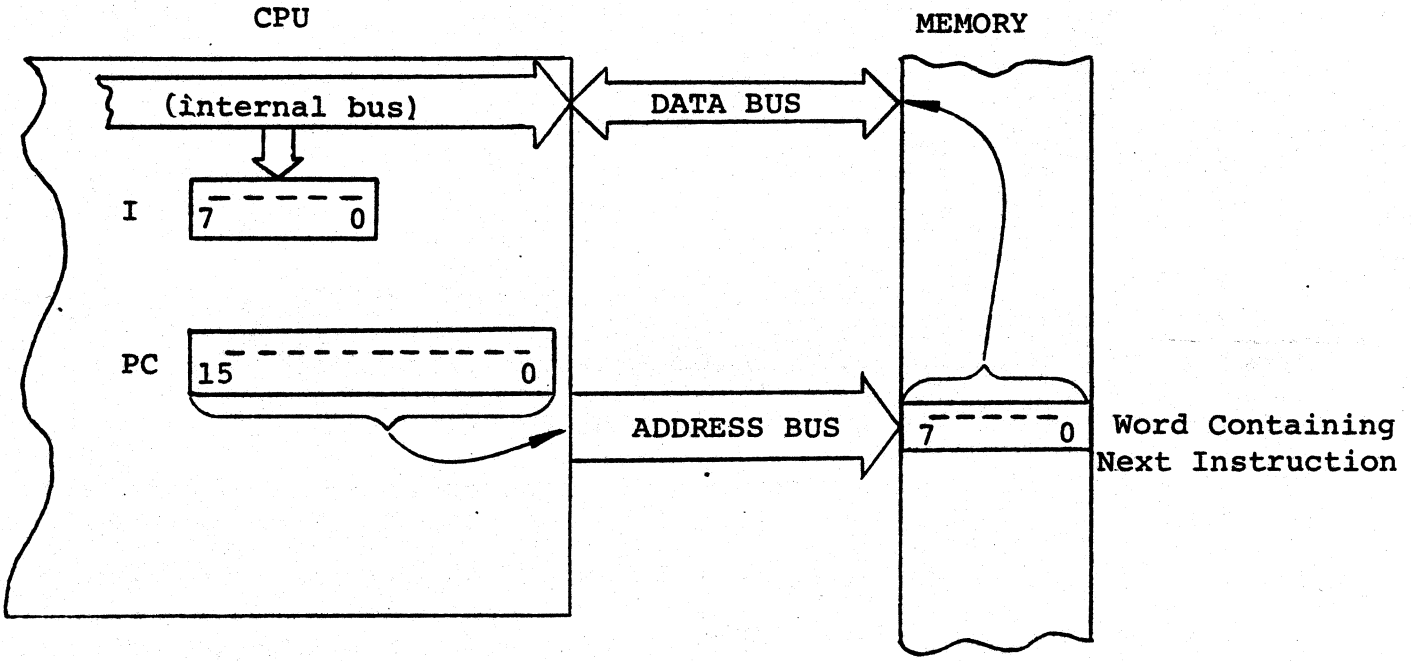
Use of the PC is illustrated below:



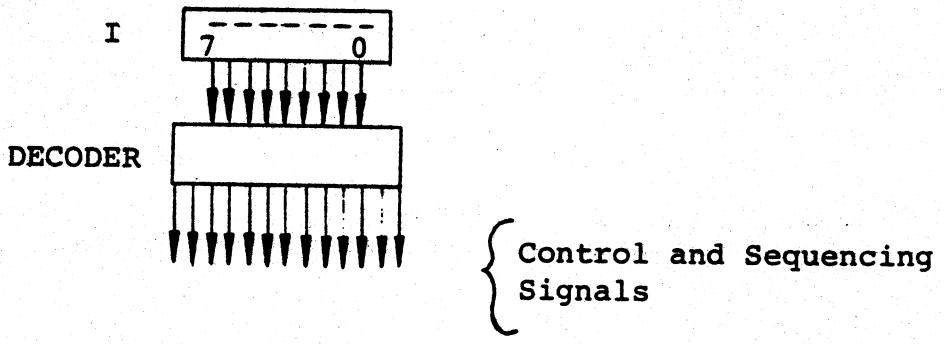
1.4.5 The Instruction Register

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 works much like the address decoder described earlier, looking at a pattern of input binary signals and outputing 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.

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.

1.5 THE MTS MONITOR

1.5.1 Monitor Software

The Microcomputer Training System has a CPU, memory (.5K 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 software is provided to assist you in

learning to use the MTS system, and is stored in ROM so that you will not inadvertently modify any of its instructions. While it would be possible for you to learn microprocessor principles without any software assistance at all, the learning process would take considerably longer. These programs are placed in the ROM memory at the factory and are ready to run as soon as power is supplied to the system.

The programs are collectively called the monitor. The monitor controls your input and output devices (keyboard and display), allows you to inspect and change the contents of memory, and performs other functions which will be described in detail as you progress through the course.

> MONITOR: A collection of programs which control I/O devices and provide various other functions for the user.

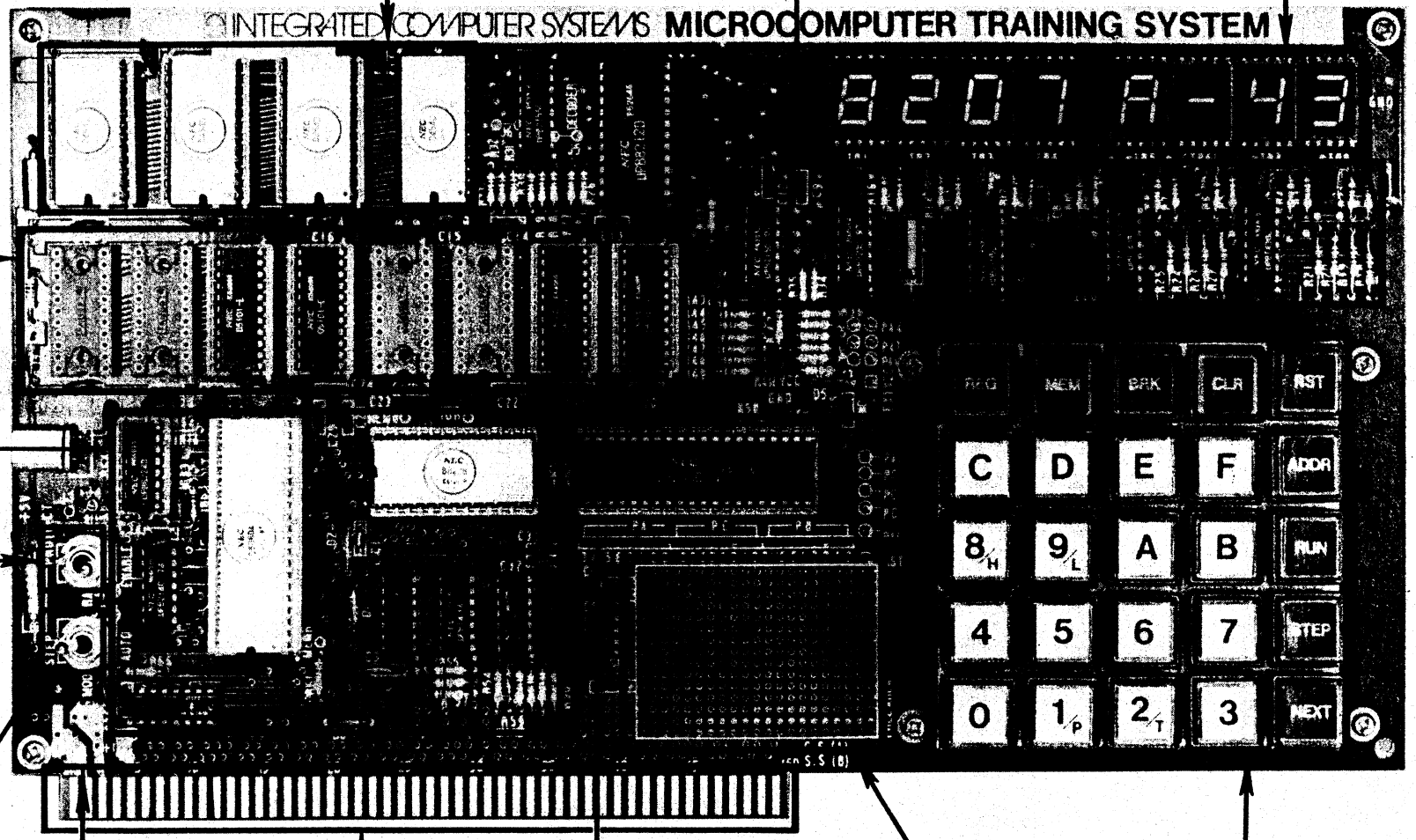
While the monitor provides these facilities to enable you to use the MTS immediately, in later chapters you will learn to write programs for controlling the keyboard and display yourself.

INTEGRATED COMPUTER SYSTEMS, INC.

MEMORY
1024 bytes of Electrically Eraseable PROM memory containing ICS Educational Monitor

DMA
Direct Memory Access (DMA) and timing circuits

DISPLAY
8-digit, 7-segment LED display



MEMORY
Space for 1024 bytes of CMOS RAM memory - 512 bytes provided with system

PROCESSOR HARDWARE
8080 microprocessor plus 8228 system controller and clock circuit

SWITCH (A)
provides the option to switch power supply mode to two user-supplied 1.5 volt dry cells. This permits retention of data in CMOS RAM memory.

SWITCH (B)
provides the option of operating the system in a hardware-generated single-step mode or in a free-running mode.

POWER SUPPLY CONNECTION
the system requires a simple external supply of +5 volts (at 1 amp) and +12 volts (at 0.2 amp) —user supplied

EDGE CONNECTOR
permits interfacing to external devices and expansion of memory (CPU address, data control buses are made available at board-edge pins)

PROGRAMMABLE PERIPHERAL INTERFACE
provides 3 programmable 8-bit I/O ports (can be programmed to provide two serial I/O ports for asynchronous transmit and receive - ICS Monitor handles all transmit/receive functions)

FREE AREA
provided for hardware additions by user

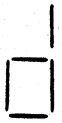
KEYBOARD
25-key keyboard (16 hex keys and 9 function keys)

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 seven 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 thus represented by:

B =  D =  R = 

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

The monitor is the silent and unseen servant that helps you accomplish all of the above functions. As it is a program, however, it uses all of the registers of the CPU, and you may be asking how your program and the

monitor programs can use the same registers without confusion. The answer is that the monitor "remembers" the contents of these registers (stores them in memory). This is possible because your program and the monitor programs are never being executed at the same time.

When the power is turned on, the monitor will set the contents of your PC to 8200_{16}, which is the first address of your RAM memory. This number will be displayed in the left four digits of the display. The contents of location 8200 will be displayed in the rightmost two digits of your display. The monitor will then wait for you to depress one of the keys on the keyboard. Initially, the contents of 8200 will be undefined - whatever is contained there is not a number which you put there. For convenience in writing, whenever a number is undefined we shall represent it with question marks. When power is turned on, then, your display will read:

8200 ??

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 a piece of paper and make two columns labeled ADDRESS and CONTENTS. Enter 8200 in the first column, and its contents (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 contents of 8201, press the command key labeled NEXT .

The display should now read:

8201	??

Write 8201 in the first column, and its contents in the second. Press NEXT again, and write down 8202 and its contents. 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 contents of 8200 will be displayed in the rightmost two digits. If you have made an error, press RST and start over.

1.5.5 Changing Memory Contents

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

8200	??

By pressing the MEM (for MEMORY) key, the monitor is commanded to accept data from the keyboard and store it in the displayed address. Press MEM , then hex key 1 ; the display will read:

8200	01

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.

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.

The A register (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 the instruction which sets the contents of the A register to zero:

BINARY CODE:	10101111
HEX CODE:	AF
MNEMONIC:	XRA A
MEANING:	Clear the contents of the A register (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 the A register (add one to the contents of the A register)

With these three instructions, you can write a program which initializes the A register 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 sets the A register to an initial value of zero and then, by successive increments of one, ends with the number seven in the A register."

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 this:

ADDRESS	HEX	MNEMONIC	COMMENTS
8200	00	NOP	Start with dummy operation
8201	AF	XRA A	Clear the A register
8202	3C	INR A	Increment the A register
8203	3C	INR A	
8204	3C	INR A	- continue to increment -
8205	3C	INR A	
8206	3C	INR A	
8207	3C	INR A	
8208	3C	INR A	- until A = 7 -

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:

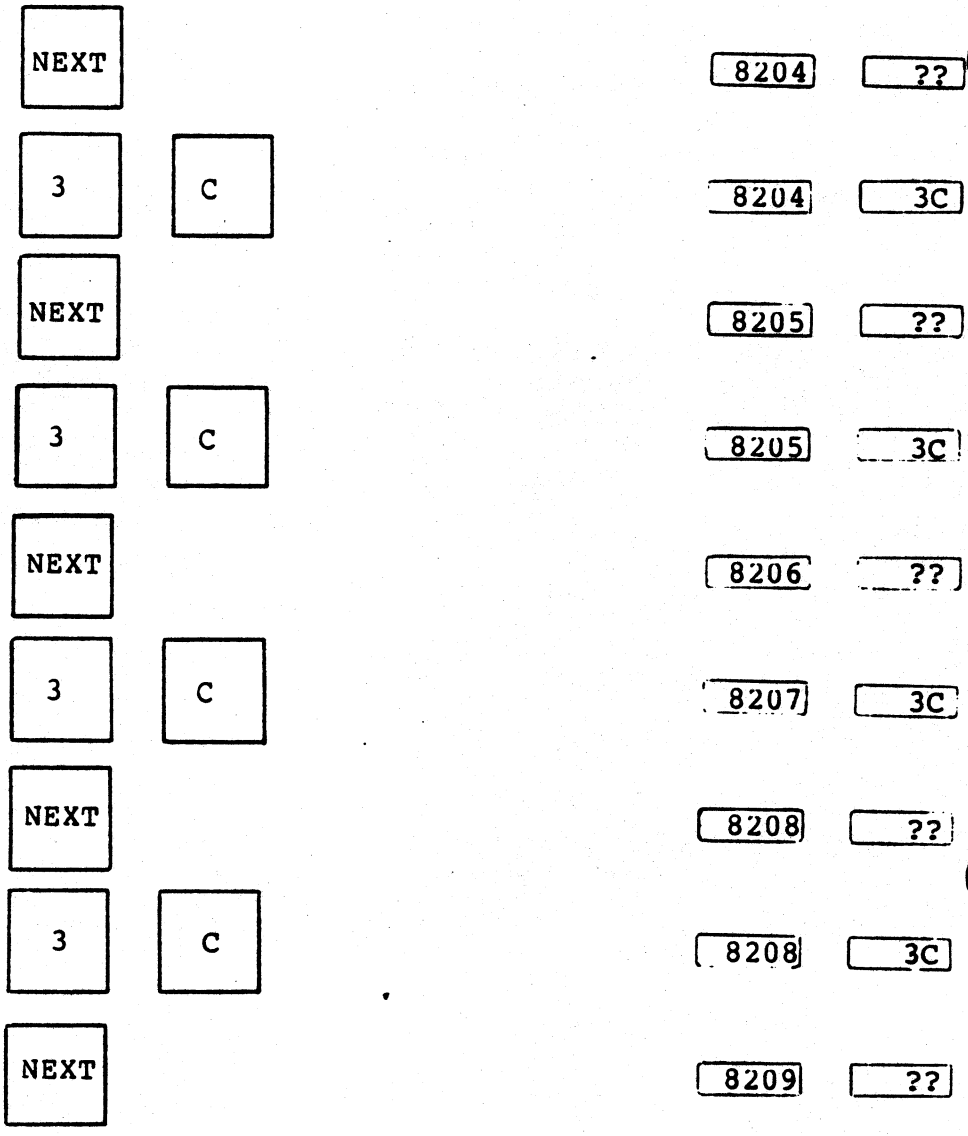
NEXT	8201	??

Enter second instruction.

A	F	8201	AF

Advance to next memory address.

NEXT		8202	??
3	C	8202	3C
NEXT		8203	??
3	C	8203	3C



Your program has now been entered in memory. Note that the final NEXT command is given to terminate your input string of characters.

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. It is trite but powerfully true that "garbage input, garbage output". To be sure that your entries are correct, press RST and then, using the NEXT command, check 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.

NEXT 8205 3D

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

3 C 8205 3C

Corrects the error.

NEXT 8206 3C

Inspect next register, 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 the A register. Using the protocol developed above:

REG A 8200 A-??

The command REG followed by the hex character A leaves the address at 8200, but in the right four digits identifies 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)

REG A 8200 A-??

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

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.

8201 AF

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

8202 A-00

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

8203 A-01

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

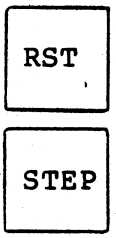
Press STEP to execute it:

8204 A-02

Continue stepping through your program in this fashion until the PC is set at 8209. At this point, the A register 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.

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, the A register and the I register at each stage of instruction execution. Initialize the system:



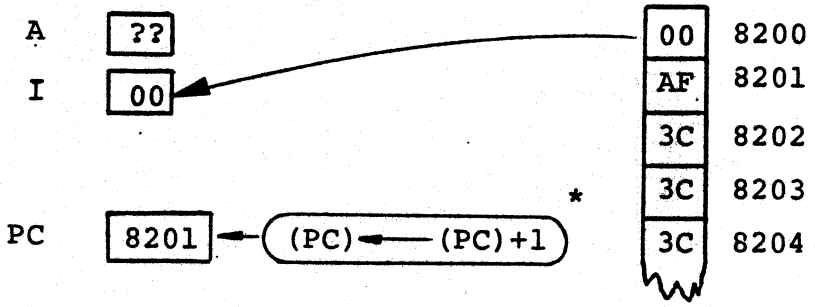
8200 00

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.



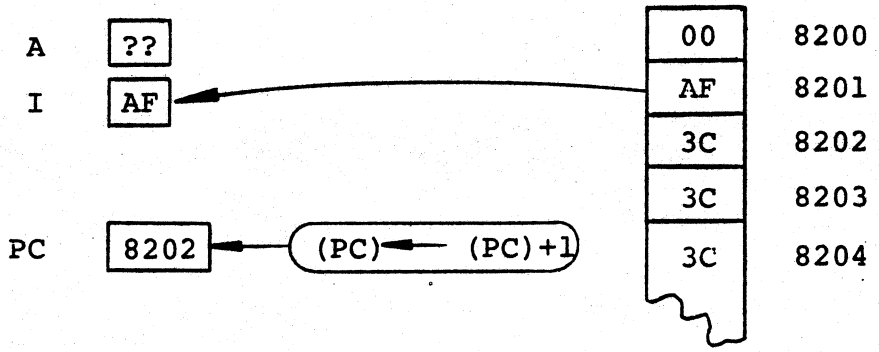
The contents of A are still undefined. The instruction is executed and as it is a NOP, the instruction cycle is completed.

* The 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".

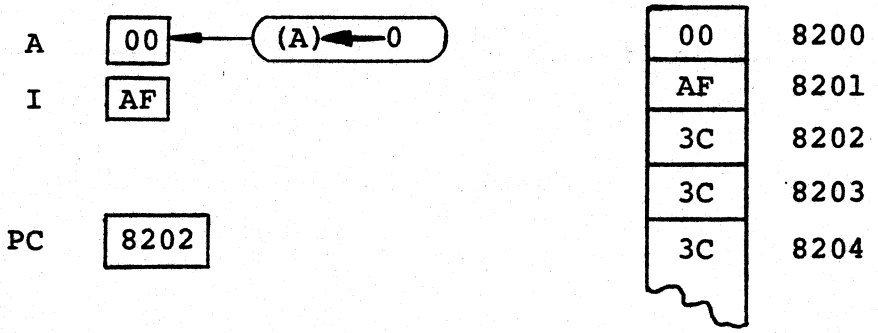
The next instruction will clear the A register:

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

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

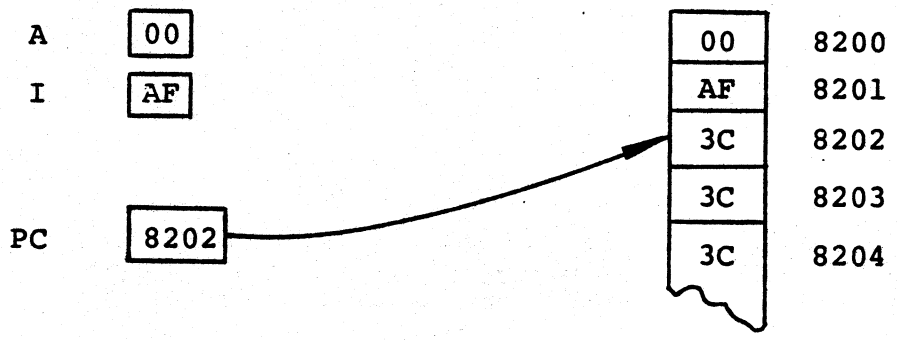


3) The instruction is executed and the A register is set to zero.

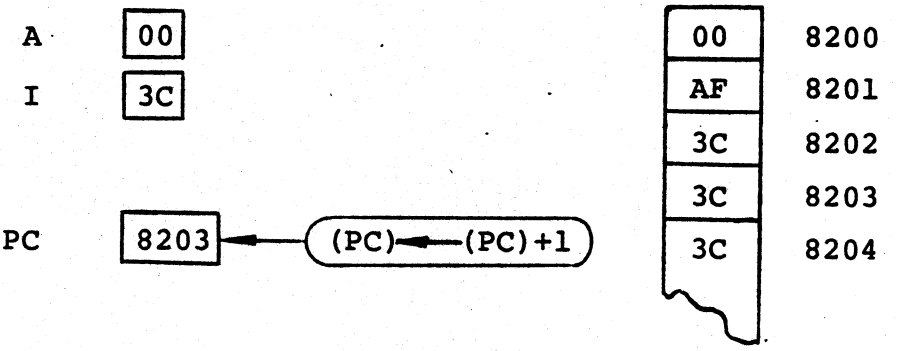


The next instruction will increment the A register:

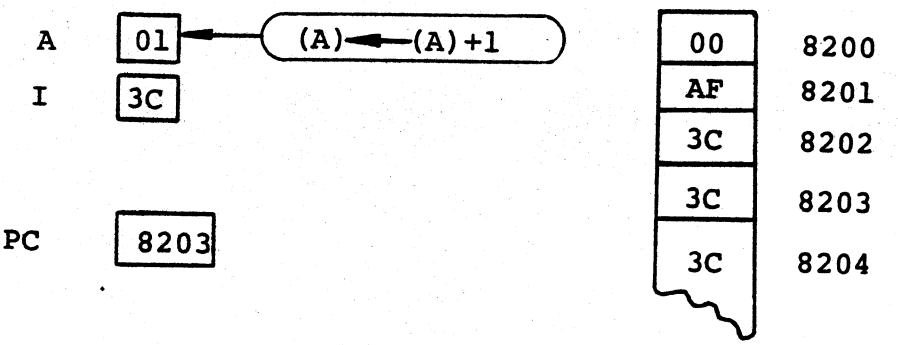
1) The processor sends the contents of (PC) to the memory, selecting address 8202.



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



3) The instruction is executed and the A register 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.

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 the second byte 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 an address 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:

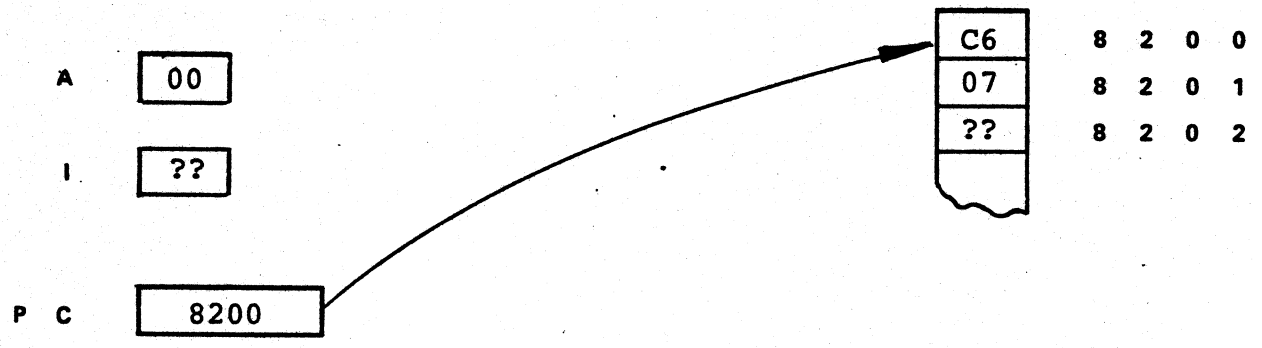
BINARY CODE:	00110010
HEX CODE:	32
BYTE TWO:	Low-order part of storage address
BYTE THREE:	High-order part of storage address
MNEMONIC:	STA
MEANING:	Store the contents of the accumulator (A) in the address which is contained in the following two memory addresses.

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.

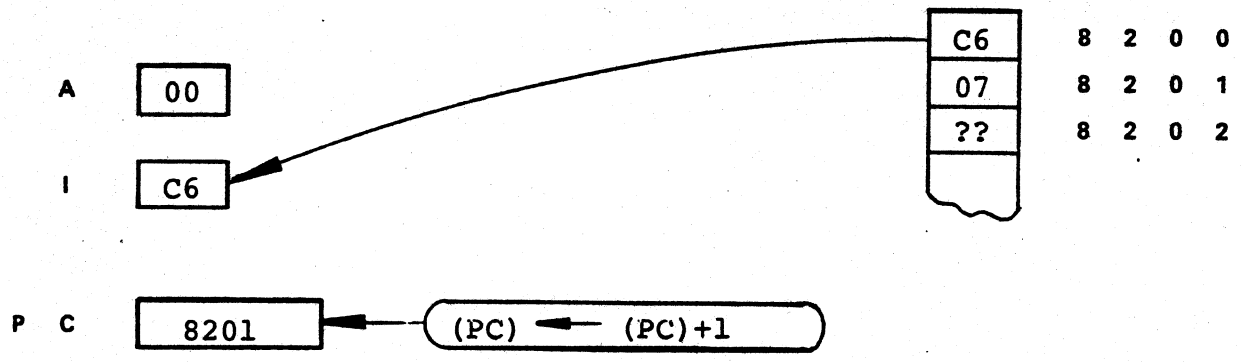
2.1.3 Instruction Execution Details

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 placed in the A register. The operation looks like this:

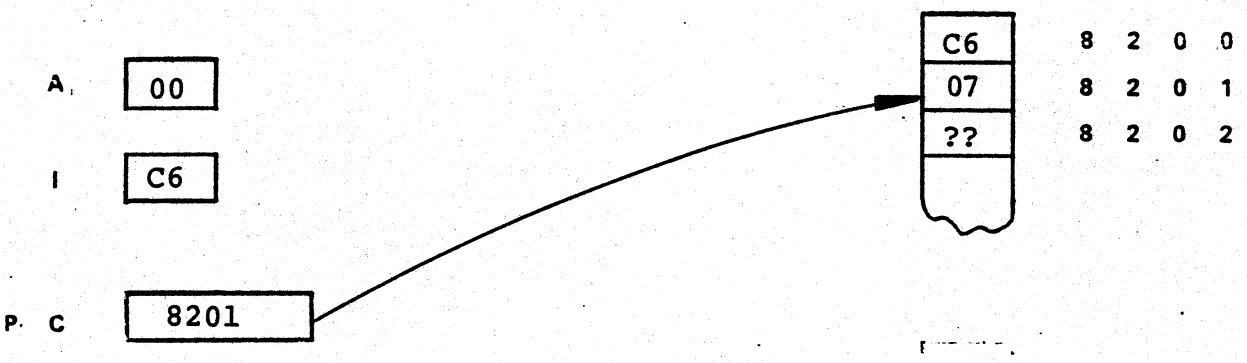
1) The processor sends the contents of (PC) to memory, selecting address 8200 (for this example)



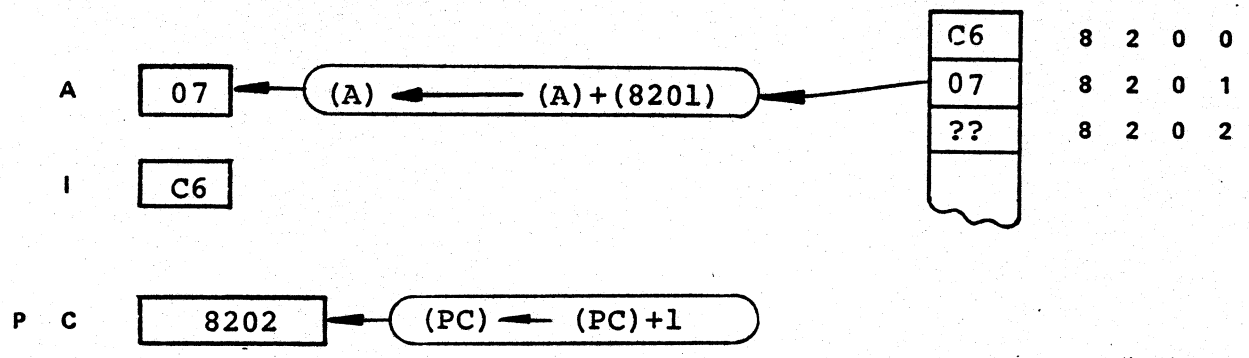
2) The memory sends the contents of address 8200 to the I register and (PC) is incremented by 1.



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



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



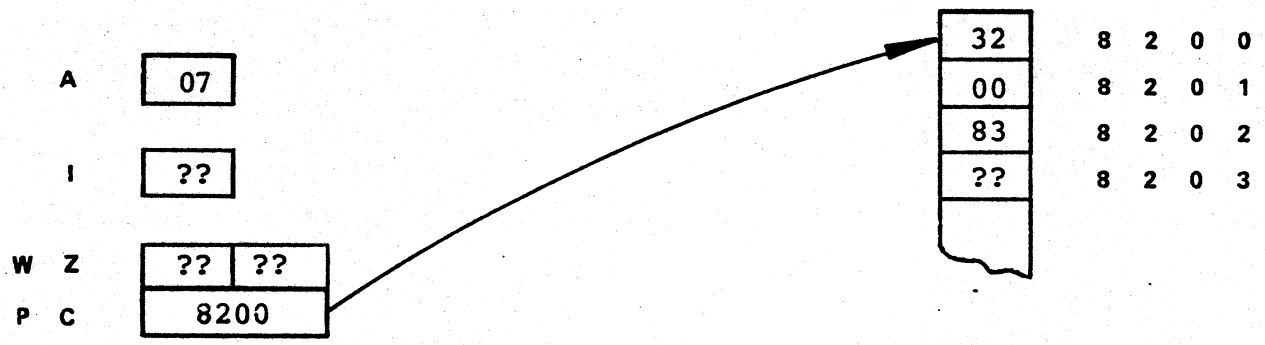
5) The instruction is completed. The memory has been accessed twice (two machine cycles), and (PC) has been incremented twice.

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 the I register, the W and Z registers are for internal use by the processor and no instruction explicitly refers to them.

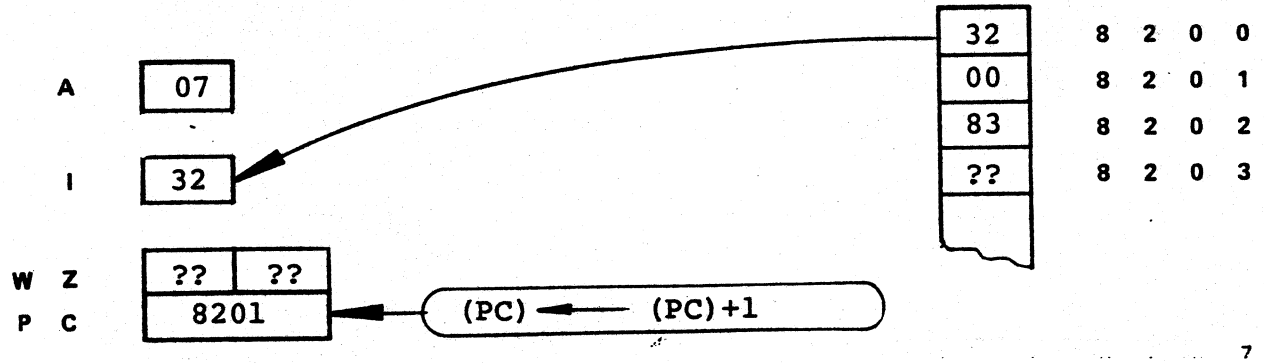
W,Z REGISTERS:	A temporary register pair in the address logic used during internal execution of instructions.

The details of execution are:

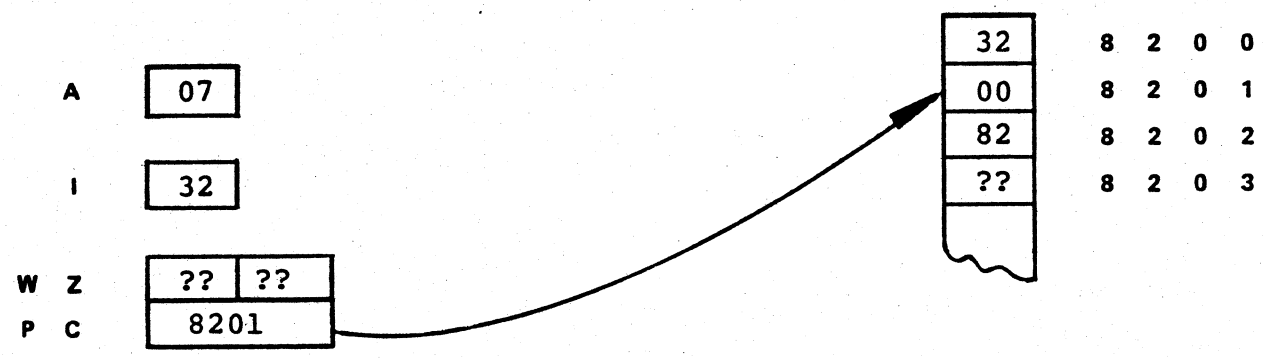
1) The processor sends the contents of (PC) to memory, selecting address 8200 (for this example):,



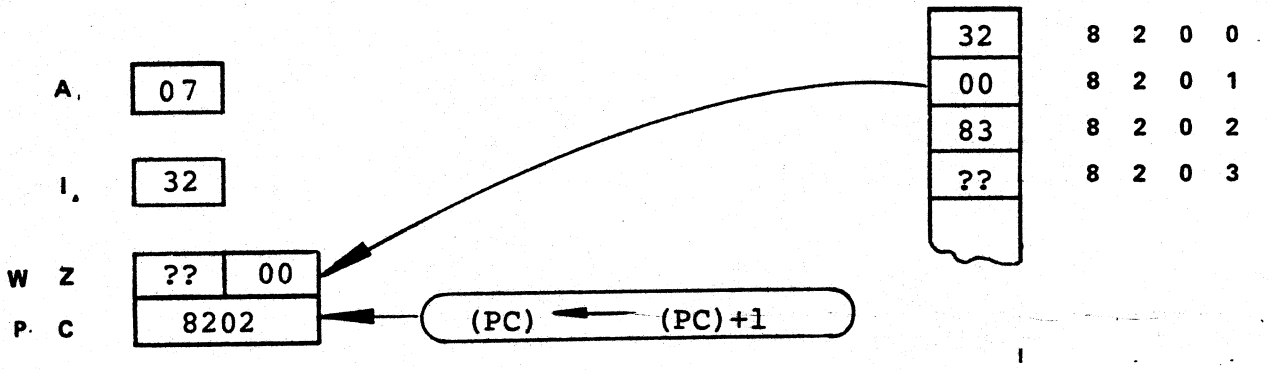
2) The memory sends the contents of 8200 to the I register and (PC) is incremented by 1.



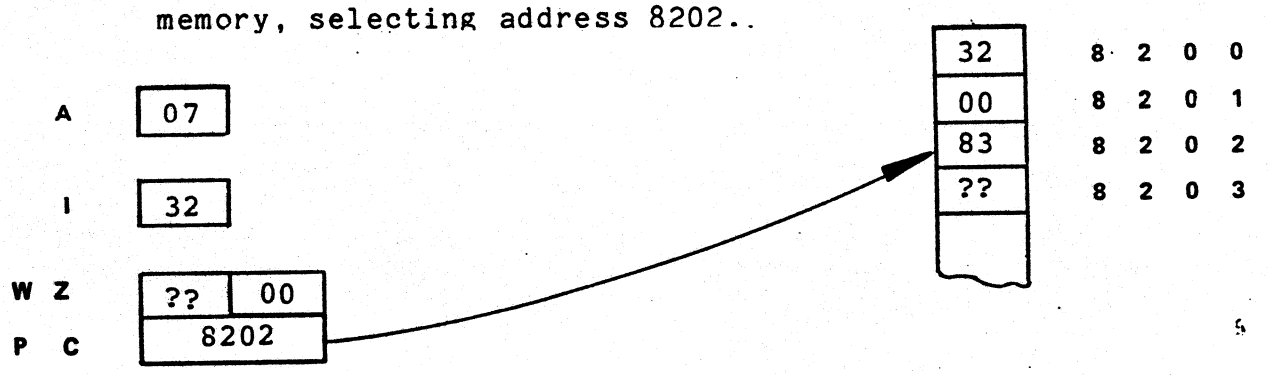
3) The instruction is decoded, and the processor sends the contents of (PC) to memory, selecting address 8201.



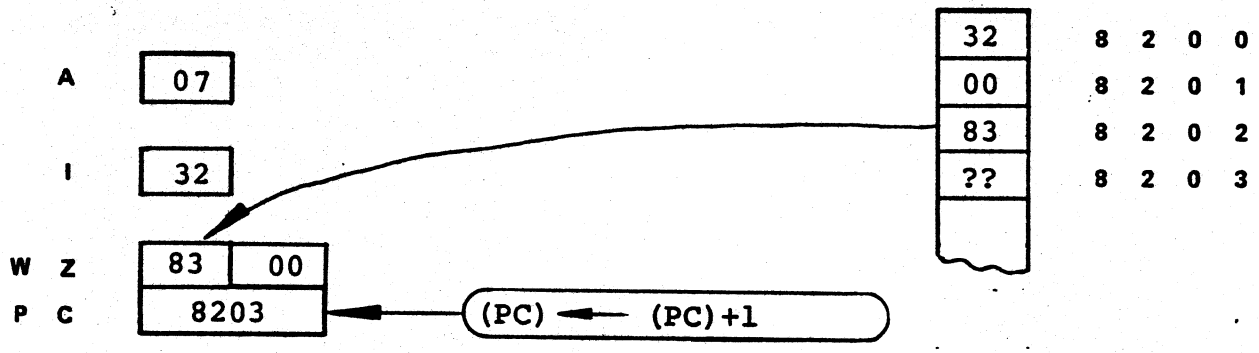
4) The memory sends the contents of 8201 to the Z register and (PC) is incremented by 1. Now Z contains the low order part of the address in which the contents of A will be stored. The design of the processor requires that the low order part of the address be stored immediately after the instruction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



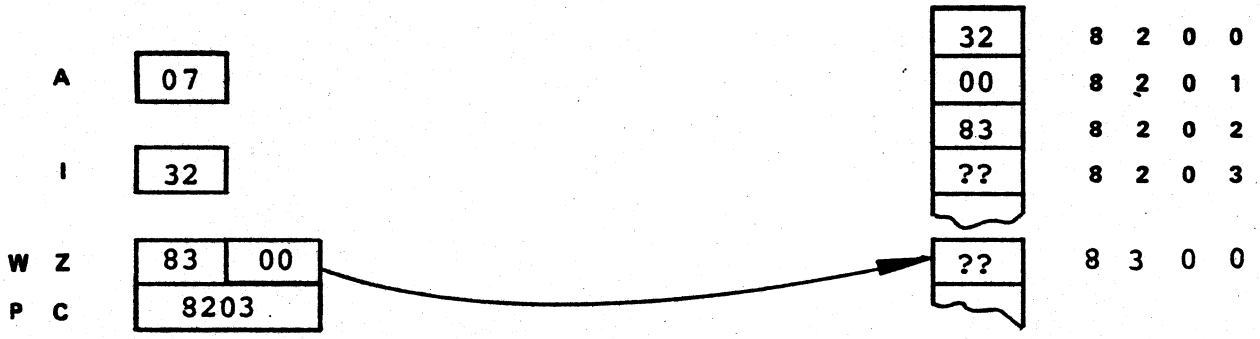
5) Again the processor sends the contents of (PC) to memory, selecting address 8202..



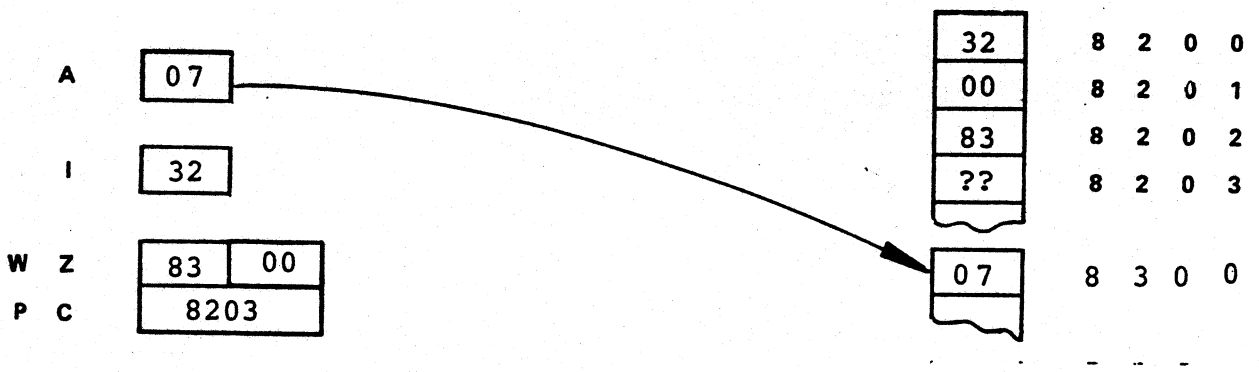
6) The memory sends the contents of 8202 to the W register, and (PC) is incremented by 1. The complete address in which the contents of A are to be stored is now available.



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



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



The execution of STA has required four machine cycles: an instruction fetch, two memory reads, and 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.

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 the A register
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		Dummy operation.

Note that 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 (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. If ADDR had been followed by STEP instead of MEM the (PC) would have been changed. However, (PC) should still be set at 8200, so your program can be executed as follows:

ADDR 8200 00

(PC) and contents of 8200.

REG A 8200 A-??

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 the A register.

STEP 8204 A-07

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

STEP 8205 A-08

First of the INR A instructions adds 1 to the contents of A.

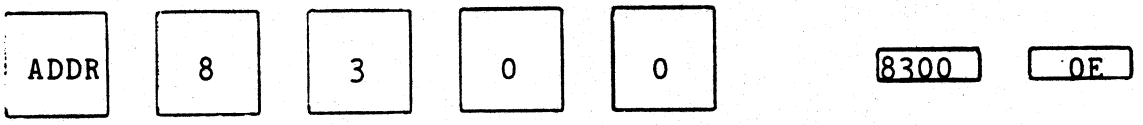
STEP	8206	A-09
STEP	8207	A-0A
STEP	8208	A-0B
STEP	8209	A-0C
STEP	820A	A-0D
STEP	820B	A-0E

Now A contains $0E_{16} = 14_{10}$; 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:



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. You should satisfy yourself that 8300 is the first word of the top half of your .5K RAM memory.

N.B. As the monitor is stored in read-only memory, it requires part of the RAM for temporary storage of data. The top 96 bytes of RAM, addresses 83A0 through 83FF, are allocated to the monitor; care should be taken not to modify these memory locations.

2.3 PROGRAM EXERCISE #3

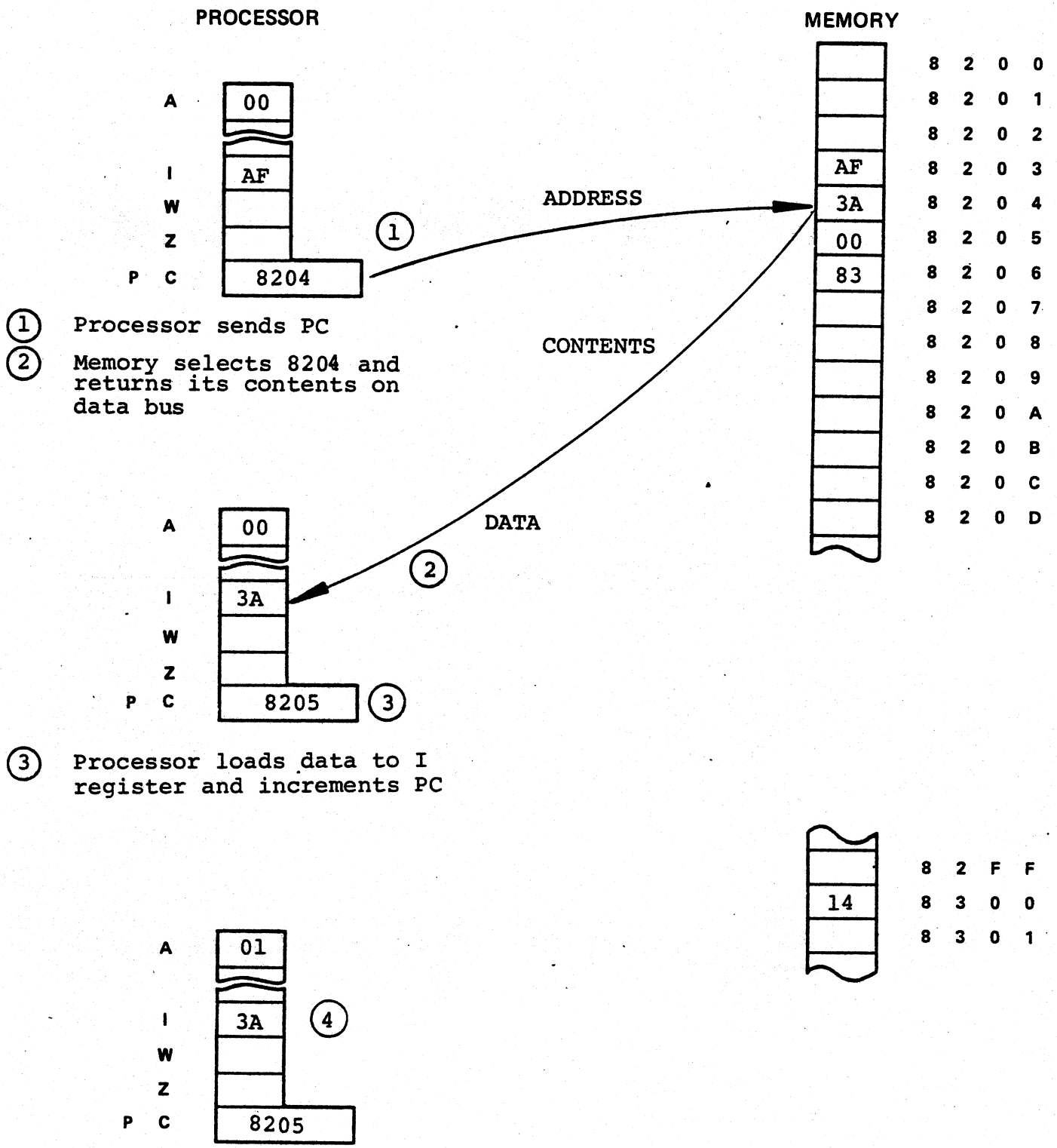
2.3.1 The LDA Instructions

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 addresses.

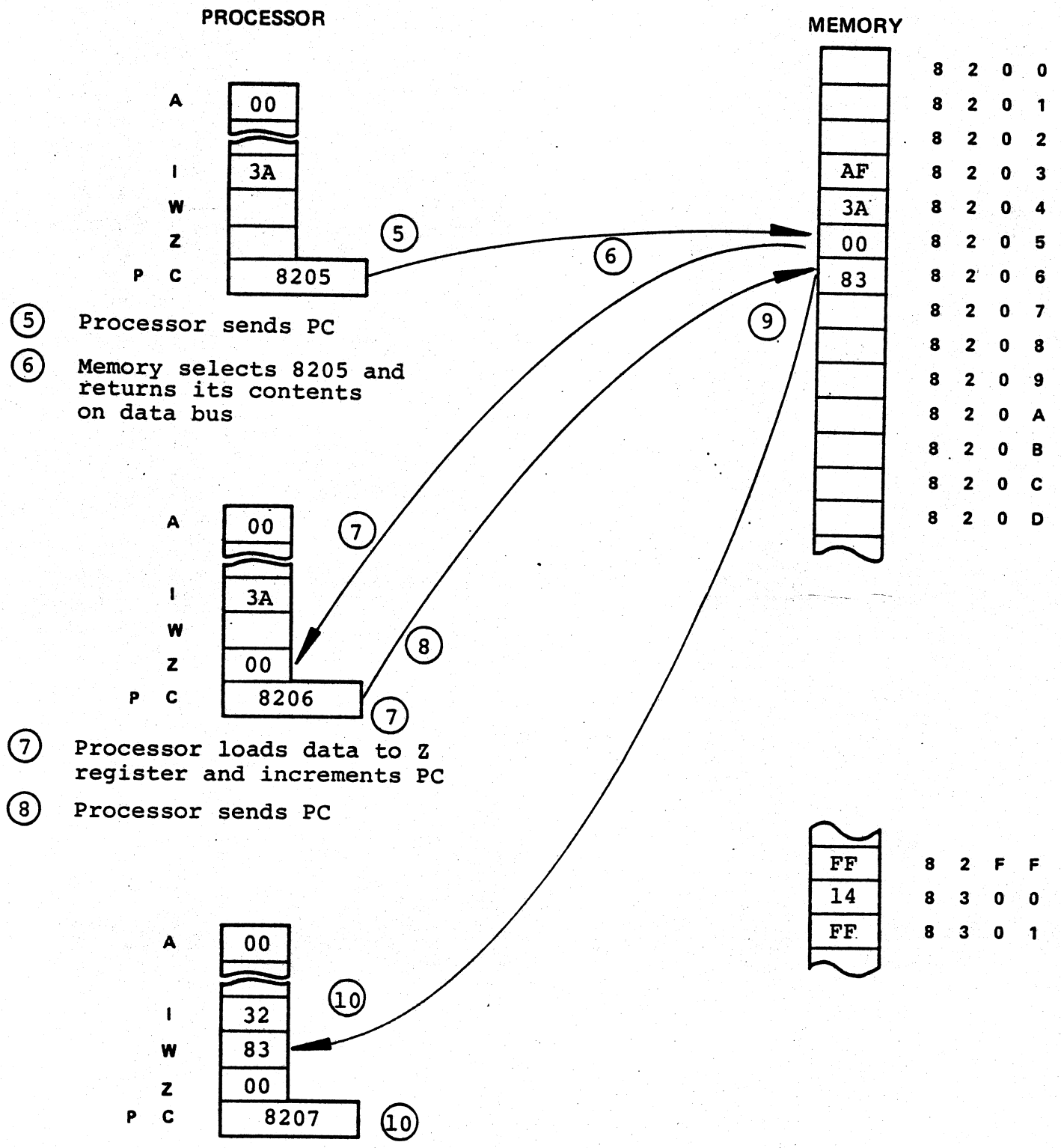
The detailed instruction cycle for LDA is shown in Figures 2-1, 2-2 and 2-3.

LDA INSTRUCTION CYCLE



1. Processor sends PC
2. Memory selects 8204 and returns its contents on data bus
3. Processor loads data to I register and increments PC
4. Processor interprets 3A as a three byte instruction

Figure 2 - 1



(5) Processor sends PC

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

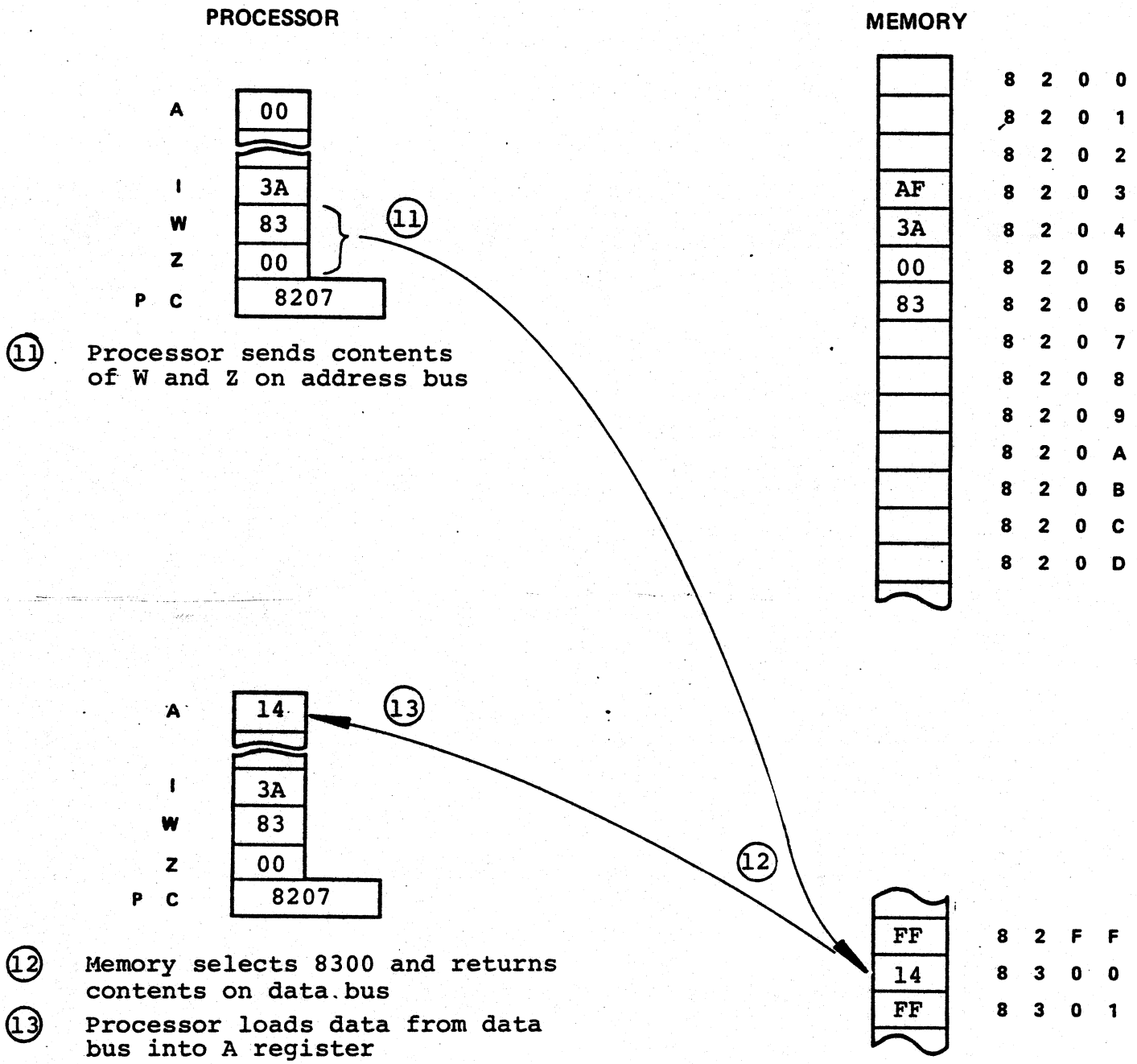
(7) Processor loads data to Z register and increments PC

(8) Processor sends PC

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

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

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 A register

Figure 2 - 3

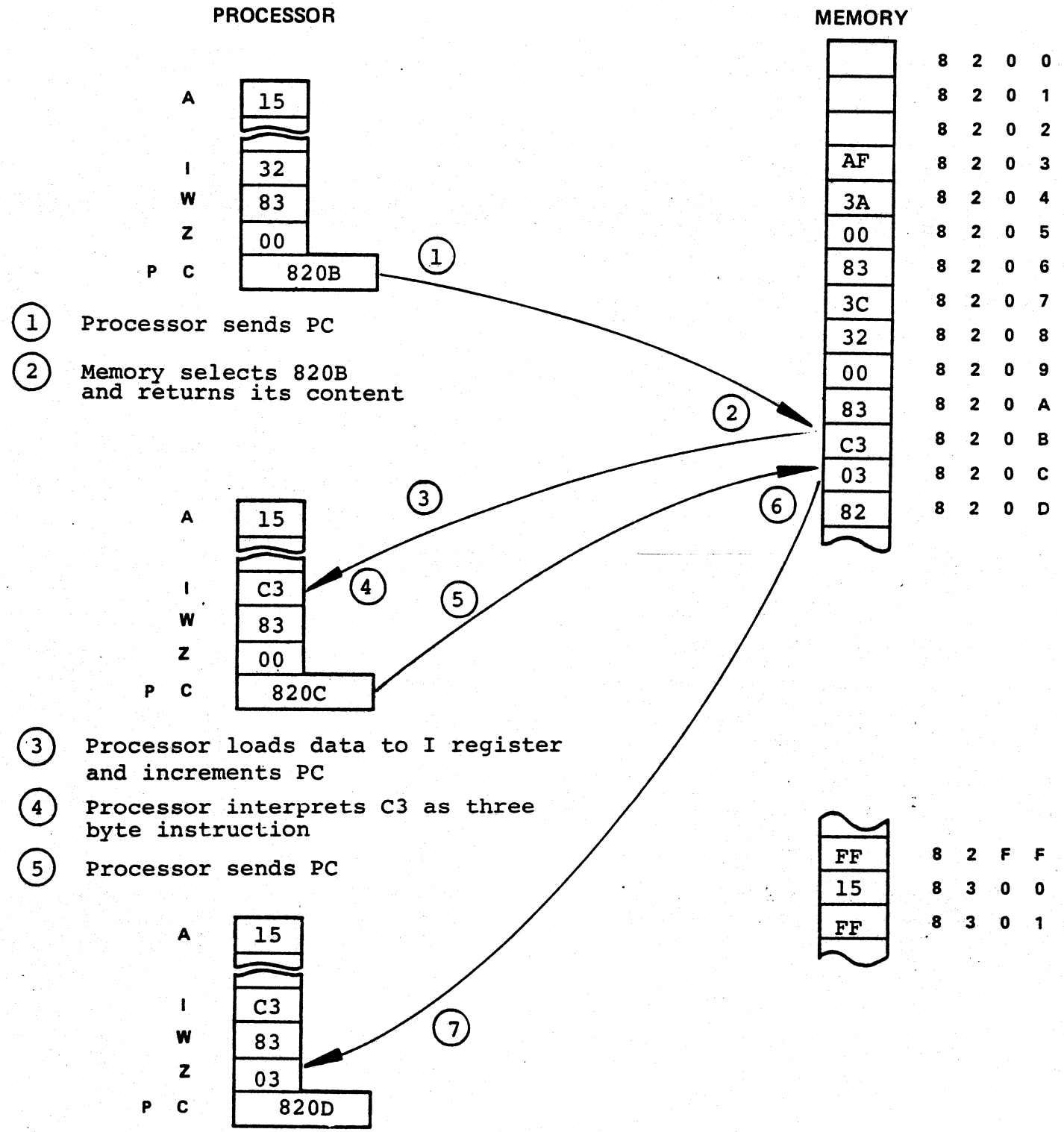
2.3.2 The JMP Instruction

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 words.

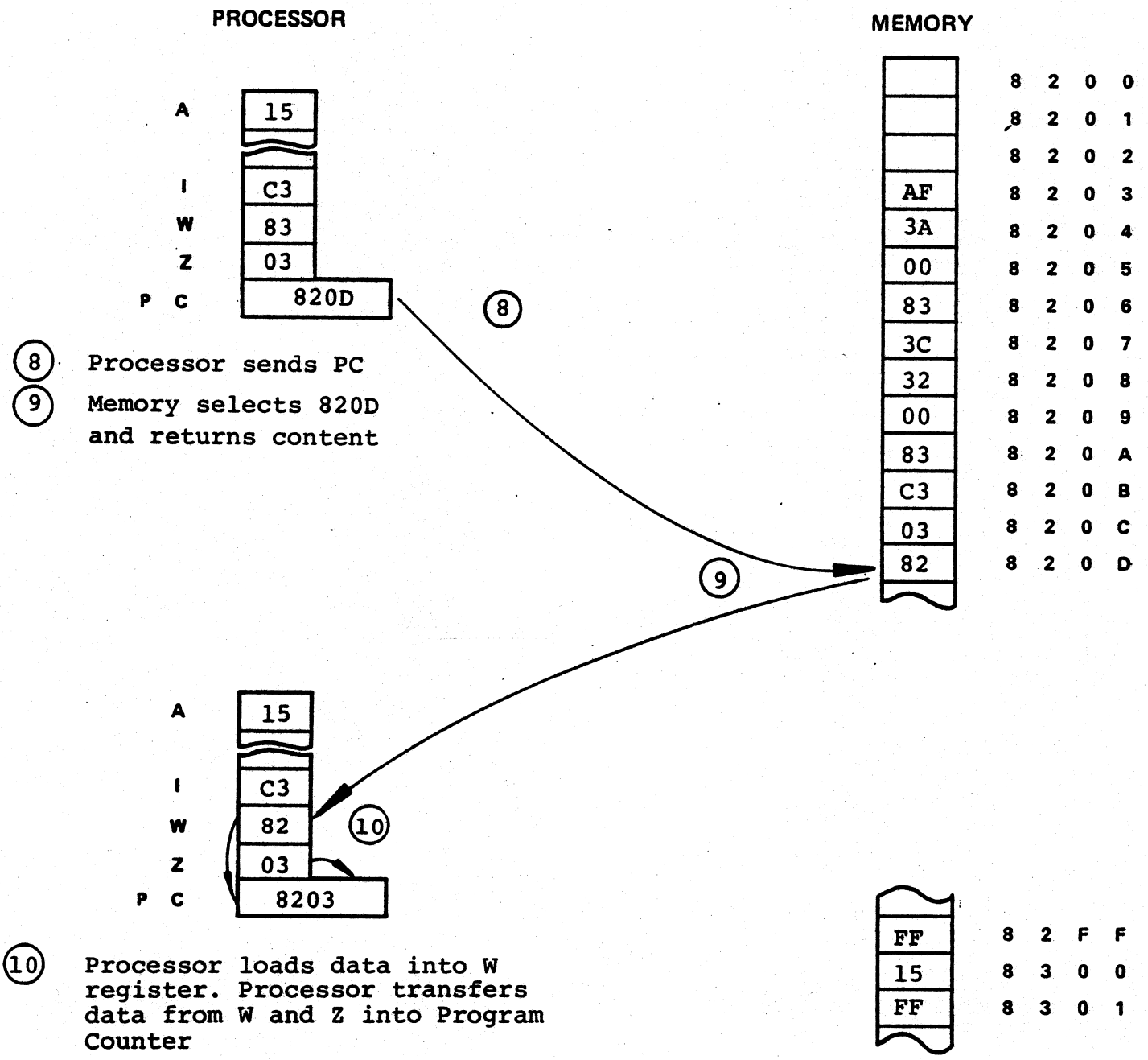
The Execution cycle of the JMP instruction is shown in Figures 2-4 and 2-5.

JMP INSTRUCTION CYCLE



1. Processor sends PC
2. Memory selects 820B and returns its content
3. Processor loads data to I register 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 Z register and increments PC

Figure 2 - 4



(8) Processor sends PC

(9) Memory selects 820D and returns content

(10) Processor loads data into W register. Processor transfers data from W and Z into Program Counter

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 one 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
05	00	8300
06	83	
07	3C	INR A Increment A
08	32	STA 8300 Store A in
09	00	8300
0A	83	
0B	C3	JMP 8203 Jump back to
0C	03	start
0D	82	
8300	14	Arbitrary Data

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

STEP 8201 00

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

STEP 8202 00

STEP 8203 AF

Two more STEP's get us to the Clear A instruction.

STEP 8204 3A

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

STEP 8207 3C

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.

STEP 8208 32

This is STA, another three byte instruction

STEP 820B C3

We have come to the JMP instruction.

STEP

8203 AF

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

REG 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.

STEP

8204 A-00

Now the A register has been cleared.

STEP

8207 A-15

Now the LDA has reloaded from 8300.

ADDR 8207 3C

ADDR displays the instruction

STEP 8208 A-16

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

Let's examine the memory location.

ADDR 8 3 0 0 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.

ADDR 8208 32

Then STEP.

STEP

820B A-16

And look again at 8300:

ADDR 8 3 0 0

8300 16

Now the new value has been stored.

MEM

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.

STEP

8203 AF

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

MEM

8300 16

We have introduced four new instructions and looked at the details of their execution cycles. In Chapter 3 we will begin to develop some fundamental concepts of programming.

2.4 SUMMARY OF INSTRUCTIONS

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

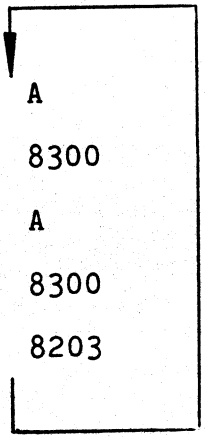
MICROCOMPUTER TRAINING WORKBOOK

CHAPTER 3

PROGRAM LOOPS

3.1 PROGRAM LOOPS AND FLOW CHARTS

The program we used in Chapter 2 was a loop:



```
XRA   A
LDA   8300
INR   A
STA   8300
JMP   8203
```

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 introducted, which describes the program 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

In the program loop shown at 3.1, the content of the A register is repeatedly incremented. Once every 256 times the program loops, the contents become FF and then 00. This change can be detected 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 this instruction by pressing

ADDR	8	2	0	B	820B	C3
MEM	C	2			820B	C2
NEXT					820C	03

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:

ADDR	8	3	0	0	8300	??
MEM	F	C			8300	FC

Now go back to the beginning and step.

ADDR	8	2	0	0	8200	00
STEP					8201	00

Request display of register A,

REG	A	8201	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.

STEP	8207	A-FC

The LDA instruction at 8204 has loaded A with the data from 8300.

STEP	8208	A-FD

(INR A done)

STEP	820B	A-FD

(STA done)

STEP	8203	A-FD

(JNZ done)

Continue stepping until you see:

STEP 8207 A-FF

(LDA done)

STEP 8208 A-00

(INR A done)

Register A has now been incremented from FF to 00.

STEP 820B A-00

(STA done)

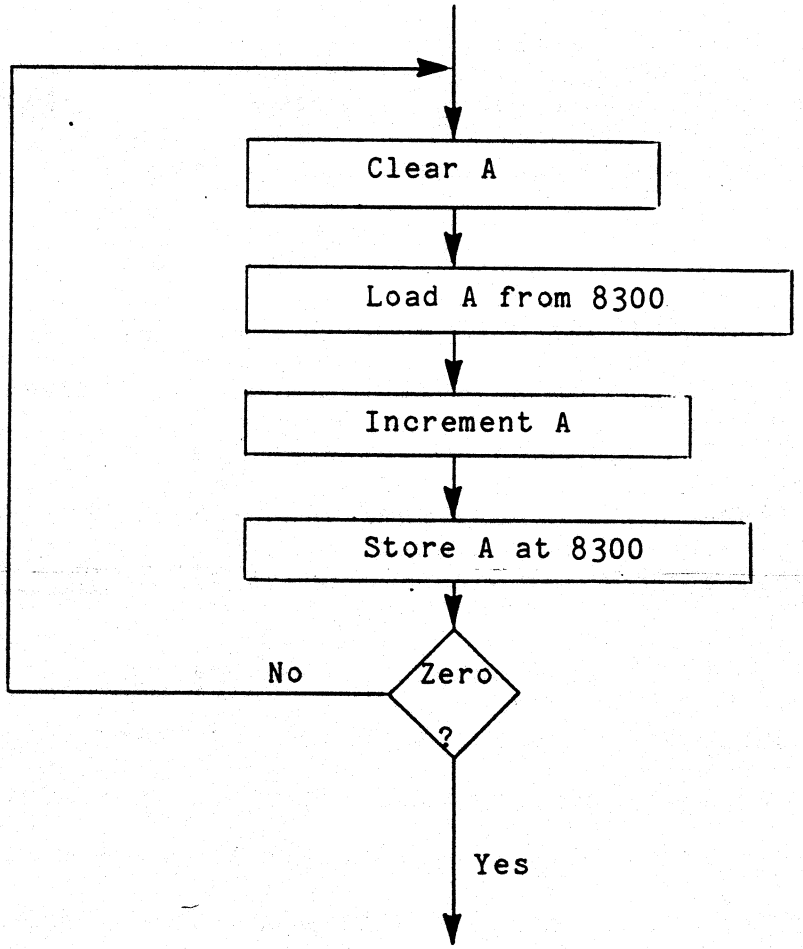
STEP 820E A-00

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 the A register 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. Eight such instructions are available, but the one we shall introduce here 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.

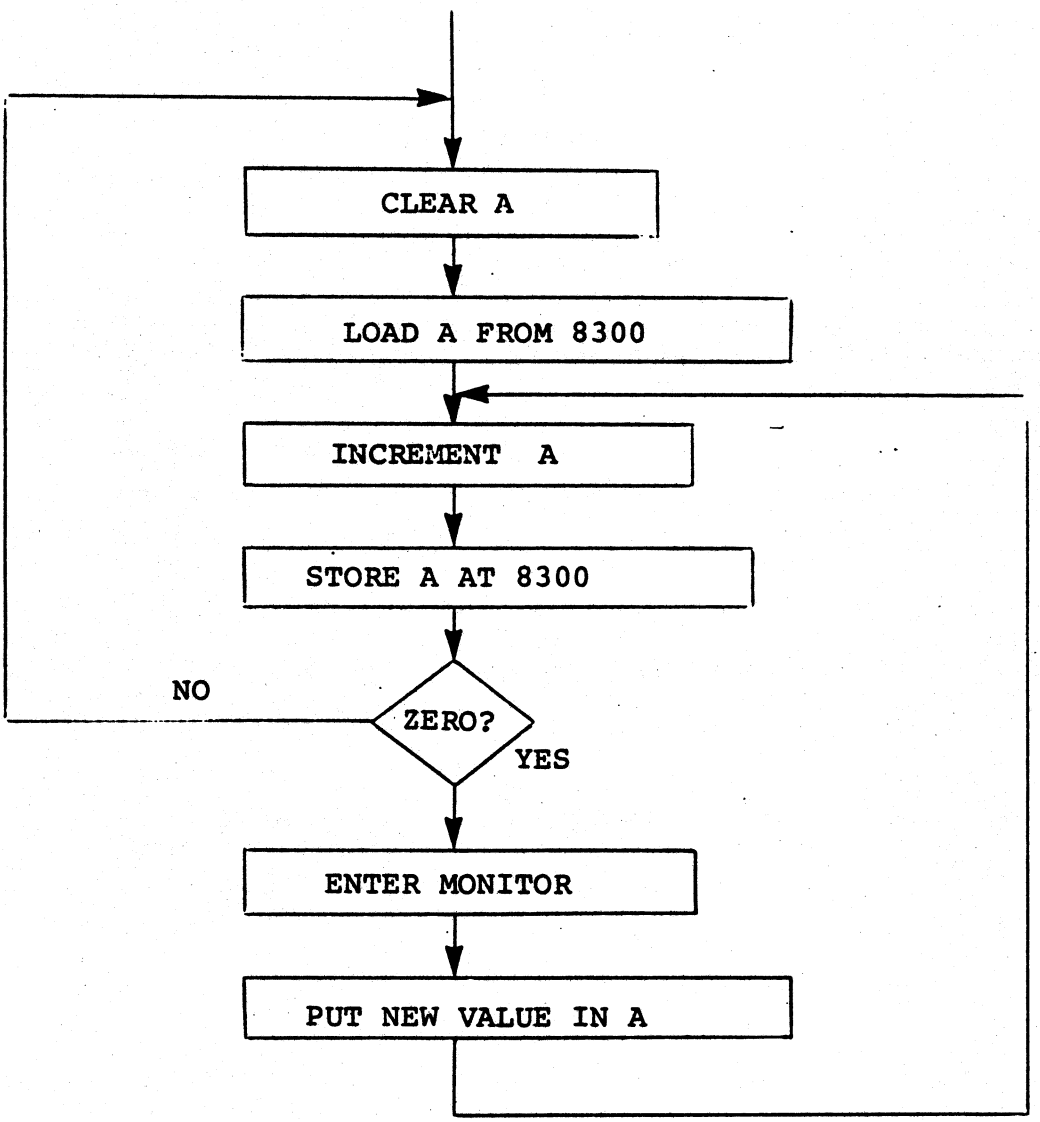

CLEAR A
LOAD A FROM 8300
INCREMENT A
STORE A AT 8300
ZERO?
NO
YES
ENTER MONITOR
PUT NEW VALUE IN A

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 reenter the monitor. Now press

[REG] [A] [820F] [A-00]

(Your jump instruction address)

F	0

820F A-F0

You have entered a large value to the A register.

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 succesively 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 the A register

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.

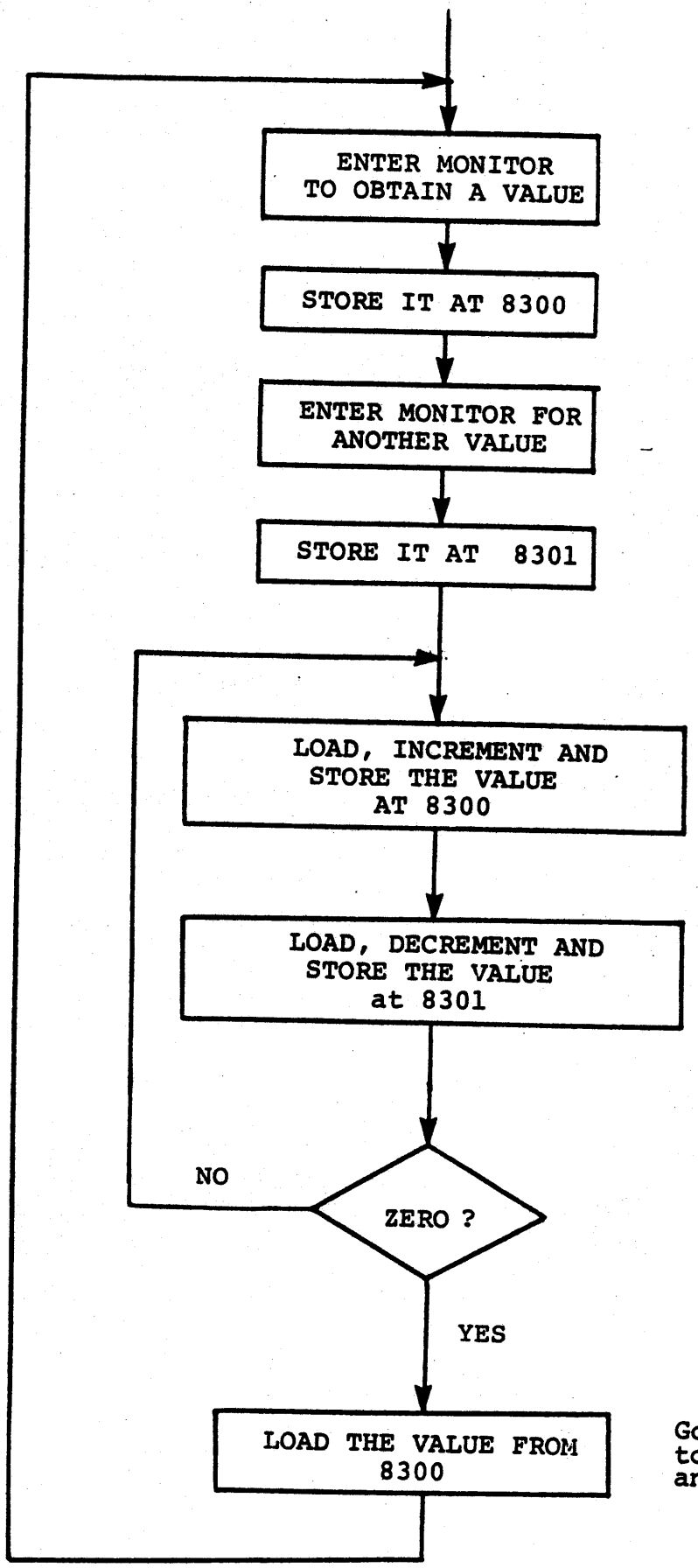

ENTER MONITOR TO OBTAIN A VALUE
STORE IT AT 8300
ENTER MONITOR FOR ANOTHER VALUE
STORE IT AT 8301
LOAD, INCREMENT AND STORE THE VALUE AT 8300
LOAD, DECREMENT AND STORE THE VALUE at 8301
NO
ZERO ?
YES
LOAD THE VALUE FROM 8300
Go back to the monitor to display the result and obtain another value

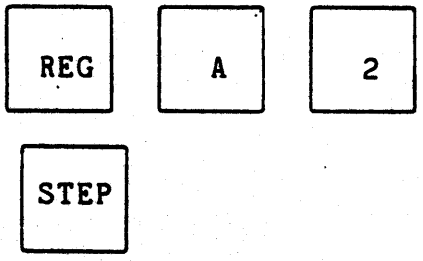
Figure 3 - 2

CODING SHEET MICROCOMPUTER TRAINING SYSTEM INTEGRATED COMPUTER SYSTEMS

ADDR	CODE												
8200	00		NOP										Save three bytes for a
01	00		NOP										future change
02	00		NOP										
03	E7		RST	4									Enter monitor
04	32		STA		8	3	0	0					and save the value
05	00												returned in A at 8300
06	83												
07	E7		RST	4									Enter monitor
08	32		STA		8	3	0	1					and save the value
09	01												returned in A at 8301
0A	83												
0B	3A		LDA		8	3	0	0					Begin loop
0C	00												Load first value
0D	83												
0E	3C		INR		A								Increment and
0F	32		STA		8	3	0	0					store the first
8210	00												value
11	83												
12	3A		LDA		8	3	0	1					Load the second
13	01												value
14	83												
15	3D		DCR		A								Decrement and
16	32		STA		8	3	0	1					store the
17	01												second value
18	83												
19	C2		JNZ		8	2	0	B					Loop until second
1A	0B												value is zero
1B	82												
1C	3A		LDA		8	3	0	0					Exit from loop
1D	00												Load the first
1E	83												value and
1F	C3		JMP		8	2	0	3					go back to
8220	03												monitor to
21	82												display it
22													
23													
24													
25													
26													
27													
28													

Fig. 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
	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. The program is now at JNZ	8219	A-01
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	8216	A-00

Stored. The program is	8219	A-00
again at JNZ but		
the jump does not occur.	821C	A-00
The first value is loaded	821F	A-04
and now the jump		
back to the beginning occurs.	8203	A-04
The monitor again.	0020	A-04
Step again. Back to your	8204	A-04
program with A unchanged.		

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_{10} (FF_{16}) can be added in the two-byte A register.

3.4 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 3.5. 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) Write the code, with comments (do not use locations 83A0-83FF)

d) Key in the code and verify it

e) Step through the program to check it, then run it.

3.5 SUMMARY OF INSTRUCTIONS

00	NOP	Do nothing
AF	XRA A	Clear the A register
3C	INR A	Increment the A register
3D	DCR A	Decrement the A register
3A	LDA	Load the A register
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	the A register 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
XX	high address	the 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

4.1 THE OTHER REGISTERS

In this section 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 the A register. 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. The A register, or accumulator, is very different in that the results of most arithmetic and logical operations are stored in the A register. Similarly, input/output instructions use the A register.

4.1.1 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 contents 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'. 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

4.1.2 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 to A the content of C

Any register content may be added to A:

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

Replace the loop in the addition program of Chapter 3 (addresses 820B to 8221) with the following code, then step through it as before:

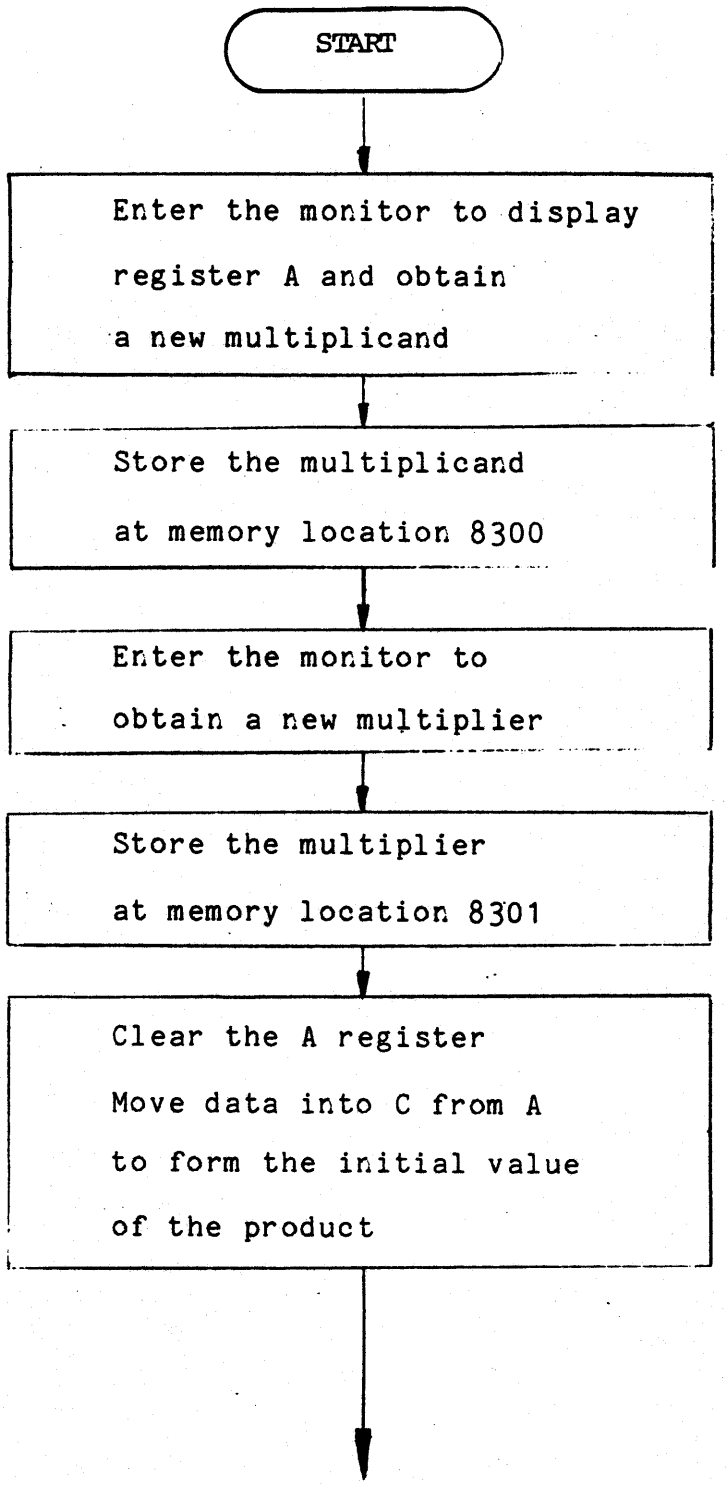
```
820B   3A      LDA  8300
820C   00
820D   83
820E   4F      MOV  C,A
820F   3A      LDA  8301
8210   01
8211   83
8212   81      ADD  C
8213   C3      JMP  8203
8214   03
8215   82
```

4.1.3 Multiplication By Addition

By applying the techniques used for addition in Chapter 3 we can perform a multiplication, since integer multiplication can be viewed as repetitive addition. Once again we will use the monitor functions to obtain input values, but instead of adding one to the other, we will repeatedly add one value (the multiplicand) to a partial product while we decrement the second value (the multiplier) until it reaches zero.

Multiplication can result in a product with as many digits as the sum of the numbers of digits in the multiplier and multiplicand, so this program is very likely to generate carries. The flow chart shown in Figure 4-1 will lose these. We will not solve the problem here: for the

moment use this program for single digit values of multiplicand and multiplier. In this flow chart note the use of circle symbols to label the destination of branching instructions. This permits flow charts to occupy more than one page while still depicting program flow. The program is given in Figure 4-2 .


START
Enter the monitor to display register A and obtain a new multiplicand
Store the multiplicand at memory location 8300
Enter the monitor to obtain a new multiplier
Store the multiplier at memory location 8301
Clear the A register
Move data into C from A
to form the initial value
of the product

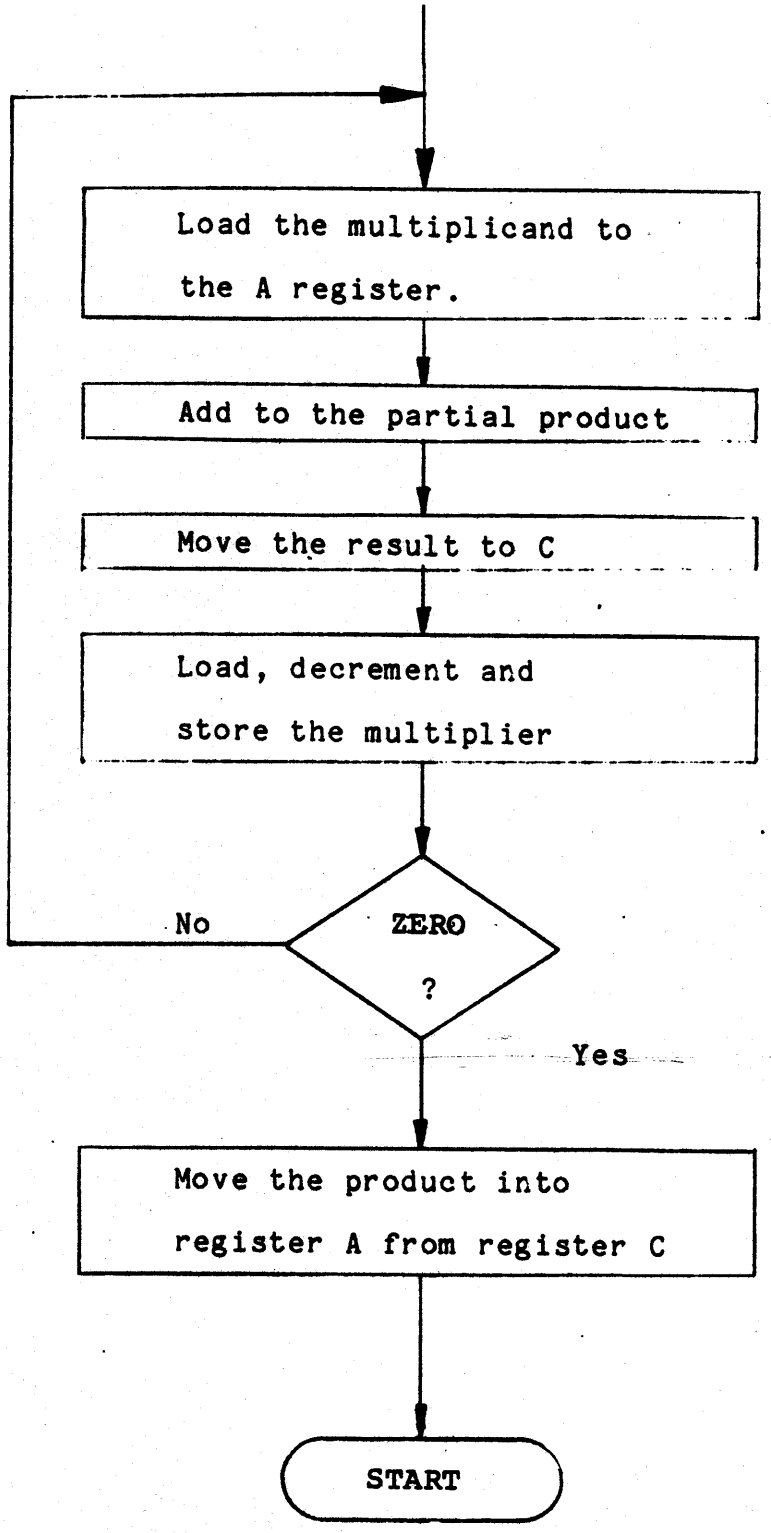

Load the multiplicand to the A register.
Add to the partial product
Move the result to C
Load, decrement and store the multiplier
ZERO ?
No
Yes
Move the product into register A from register C
START

Figure 4-1

INTEGRATED COMPUTER SYSTEMS MICROCOMPUTER TRAINING SYSTEM CODING SHEET

BINARY MULTIPLICATION BY REPETITIVE ADDITION

ADDR	CODE			
8200	00	NOP		Start
01	00			
02	00			
03	E7	RST	4	Enter monitor
04	32	STA	8300	for multiplicand;
05	00			store in memory
06	83			
07	E7	RST	4	Enter monitor
08	32	STA	8301	for multiplier;
09	01			store in memory
0A	83			
0B	AF	XRA	A	Clear A
0C	4F	MOV	C, A	Clear C for product
0D	3A	LDA	8300	Load multiplicand
0E	00			
0F	83			
8210	81	ADD	C	Add to product
11	4F	MOV	C, A	Return product to C
12	3A	LDA	8301	Load multiplier
13	01			
14	83			
15	3D	DCR	A	Decrement multiplier
16	32	STA	8301	Store multiplier
17	01			
18	83			
19	C2	JNZ	820D	If multiplier not
1A	0D			yet zero, loop
1B	82			to repeat addition
1C	79	MOV	A, C	(A) ← Product
1D	C3	JMP	8203	Jump back to
1E	03			enter monitor
1F	82			and display result
8220				
21				
22				
23				
24				
25				
26				
27				
28				

Figure 4-2

Load the program shown in Figure 4-2 and step through it:

RST		8200	00
REG	A	8200	A-??

We will be entering data to A.

STEP	8201	A-??
STEP	8202	A-??
STEP	8203	A-??

The next STEP puts you in the monitor:

STEP	0020	A-EF

Enter a two-digit number:

0	2	0020	A-02
STEP		8204	A-02
STEP		8207	A-02
STEP		0020	A-02

Back in the monitor. Enter two more digits:

0	3	0020	A-03

Continue stepping (from here on we will not show STEP each time - it is implied by a new PC value):

8208	A-03
820B	A-03
820C	A-00
820D	A-00
8210	A-02
8211	A-02
8212	A-02
8215	A-03
8216	A-02
8219	A-02
820D	A-02

A has not reached zero, so the program looped. Continue stepping until PC is 821C:

821C	A-00

Exit from the loop. Now pick up result:

821D	A-06

And return to start:

8203	A-06
0020	A-06

You are back in the monitor, displaying the result and waiting for new input data. Turn the toggle switch to AUTO, press RUN, and try the program for various pairs of digits. (Press RUN after entering each pair of numbers). When STEPping through your program, the monitor displayed its own address (0020) when RST4 was executed. In RUN mode, the calling address is displayed (8204 or 8208).

4.2. THE CARRY AND ZERO FLAGS

In Chapter 3 we defined the instruction JNZ, jump if the result of the last operation was not zero. While it might appear as though the jump was conditioned by the content of A, this is not actually the case. When certain operations leave zero in A, a 'flag' is set in the CPU. The flag may be both set and cleared, and JNZ is one of several instructions which detect the state of the zero flag. Not all instructions affect the flag. For example, data transfer instructions never set any flags: these instructions include LDA, STA, MOV, and others.

4.2.1. Carry

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

```
  75
  94
 ---
 109
  16
```

This carry is generated by the ADD instruction, and sets a condition flag called the carry flag (CY). Like the zero flag which is set when the result of an operation is zero, this flag can be tested to cause a conditional jump to occur.

BINARY CODE:	11010010
HEX CODE:	D2
MNEMONIC:	JNC
SECOND BYTE:	Low-order part of address
THIRD BYTE:	High-order part of address
MEANING:	Jump if the carry flag is not set.

The instruction cycle of this instruction is the same as that for JMP, except that no jump occurs if the carry flag is set.

Single register counting instructions (INR and DCR) affect the zero flag but <u>not</u> the carry flag. If the result of the count is zero, the zero flag is set, otherwise it is cleared.

Arithmetic and logical instructions 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	JNZ	Jump if not zero
CA	JZ	Jump if zero
D2	JNC	Jump if not carry
DA	JC	Jump if carry

4.1.5 Comparison Instructions

In the add and count instructions the flag setting is a result of the operation performed. There is a set of compare instructions whose only function is to set the flags. These instructions permit a program to determine whether the contents of the A register are greater than, equal to, or less than the contents of any specified general purpose register.

For comparing the C register with the A register the instruction is:

BINARY CODE:	10111001
HEX CODE:	B9
MNEMONIC:	CMP C
MEANING:	Compare the contents of A and C and set the flags accordingly.

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

4.3 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 arguments 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. This is referred to as 'immediate data' and the instructions are called 'immediate instructions'. Such an instruction (ADI) was presented in the first chapter.

4.3.1 Move Immediate Instructions (MVI r)

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

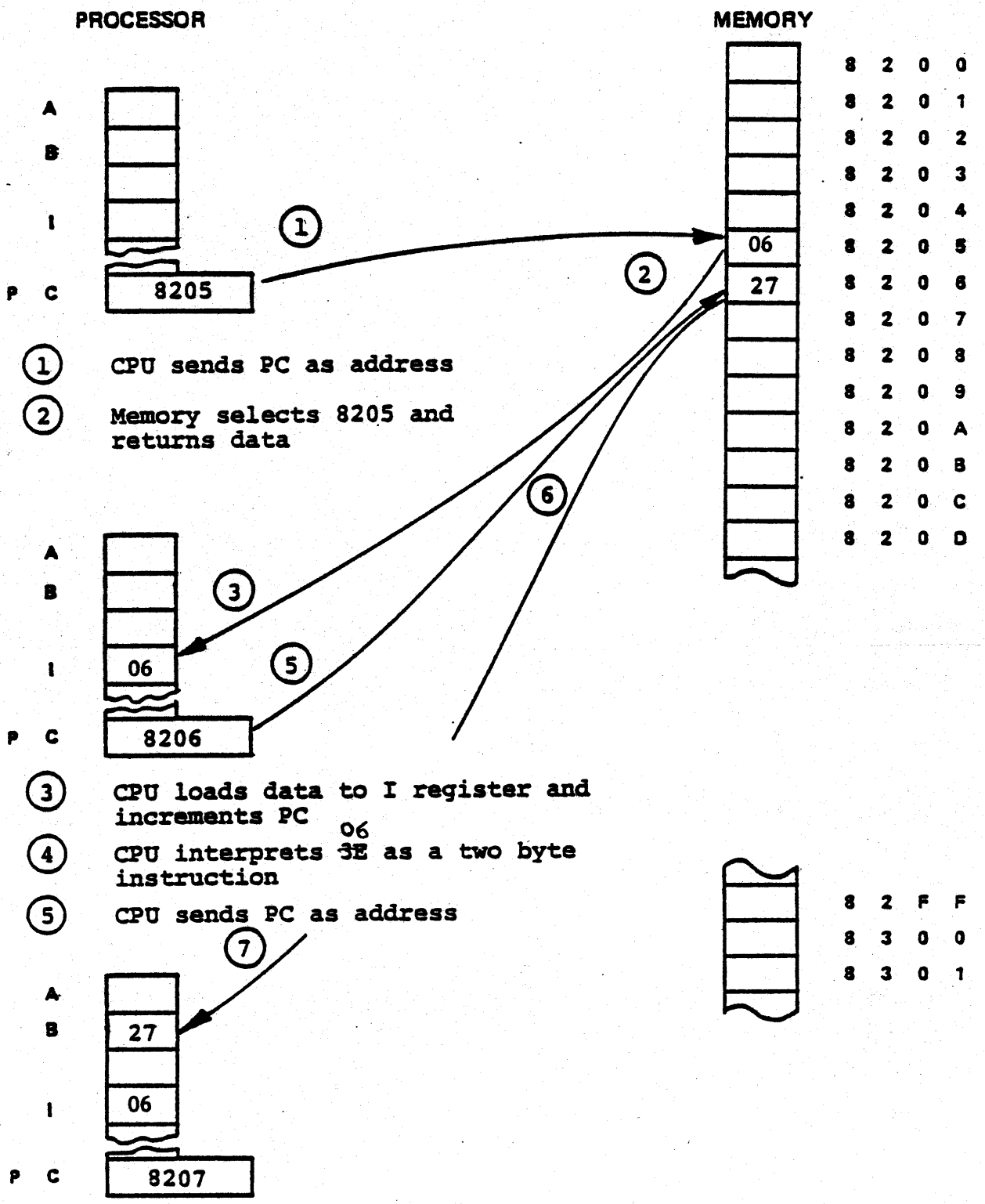
MNEMONIC:	MVI r
SECOND BYTE:	Data
MEANING:	Move the content of the following address into register r.

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. A counter is initialized at 8 and successively decremented (using DCR) to detect completion of the transmission.

The instruction cycle for MVI is shown in Figure 4-3.



(1) CPU sends PC as address

(2) Memory selects 8205 and returns data

(3) CPU loads data to I register and increments PC

(4) CPU interprets 06 3E as a two byte instruction

(5) CPU sends PC as address

(6) Memory selects 8206 and returns data

(7) CPU loads data to B register and increments PC

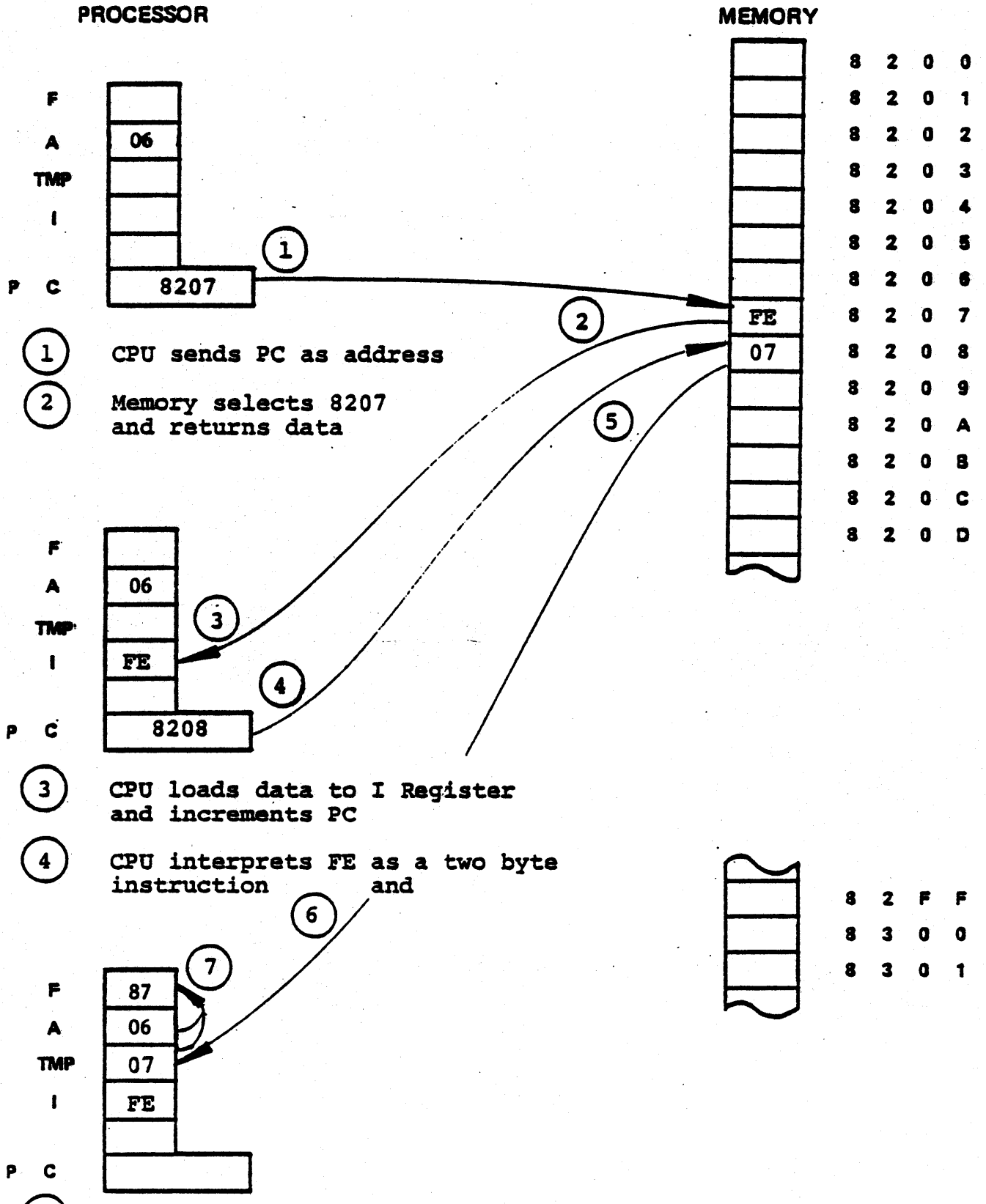
Figure 4-3

4.3.2 Compare Immediate

Immediate instructions also provide data for compare and other arithmetic and logical instructions:

HEX CODE:	FE
SECOND BYTE:	Data
MNEMONIC:	CPI
MEANING:	Subtract the content of the following address from the A register and set all flags to reflect the result. Do not modify the content of A.

From this point on, we will generally omit the practice of showing the binary code for instructions. The purpose of doing so initially was to stress the fact that binary numbers, not hex characters, are what the computer operates on. The instruction cycle for CPI is shown in Figure 4-4.



1. CPU sends PC as address
2. Memory selects 8207 and returns data
3. CPU loads data to I Register and increments PC
4. CPU interprets FE as a two byte instruction and
5. Memory selects 8208 and returns data
6. CPU loads data to TMP register and compares with Accumulator
7. CPU stores results in flags

Figure 4-4

For all of the arithmetic and logical instructions that operate on data in the A register 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.

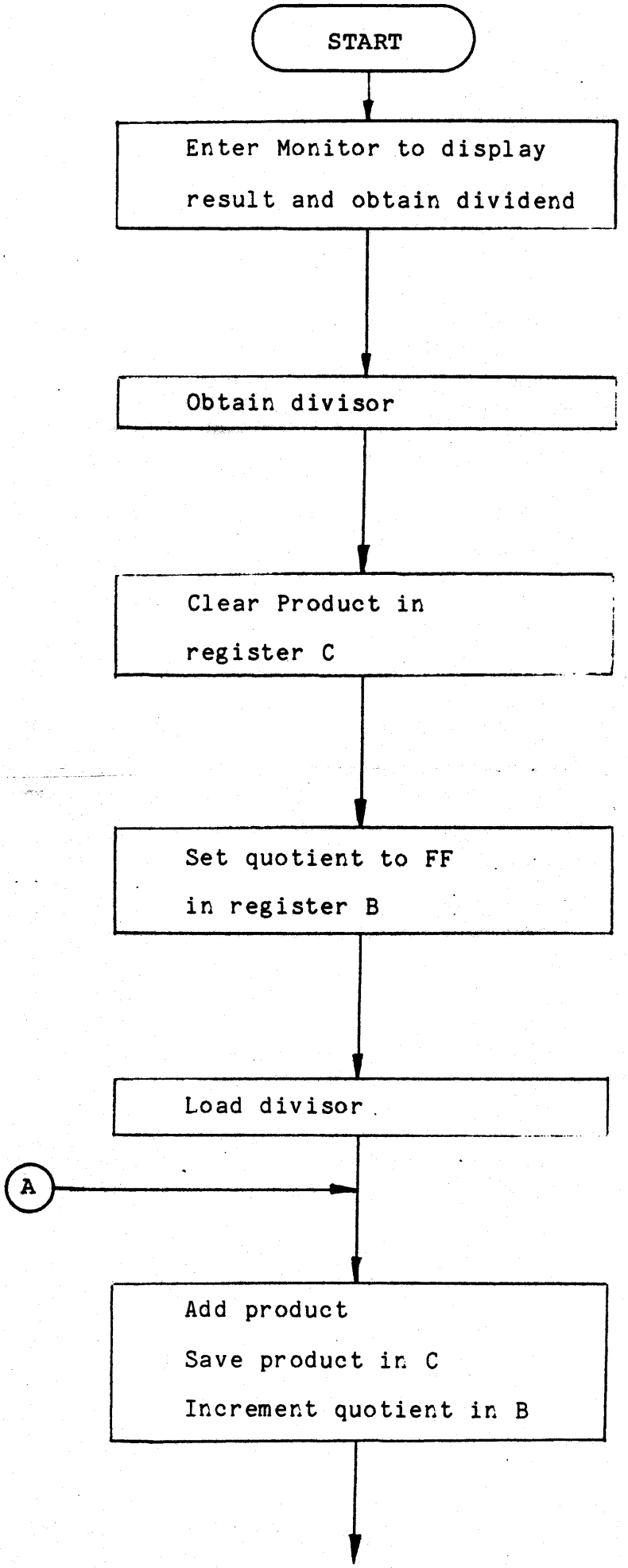
4.3.3 Division by Addition

Integer division, with no fractional result, answers the question "how many times can the divisor be added into a product before the product is greater than the dividend?" If the dividend is 7 and the divisor is 2, the quotient is 3, not 3.5, because this is integer division.

We will modify the binary multiplication program to perform integer division. Instead of counting a multiplier down, we will count a quotient up, and stop when the product is greater than the dividend. Figures 4-5 and 4-6 show the process. The initial steps of obtaining two numbers and storing them, and clearing the product in register C, are retained from the multiplication program.

We initialize the quotient, in register B, to FF rather than zero, because we will increment the quotient at least once, even if the divisor is greater than the dividend. In the loop, we add the divisor into the product, just as in multiplication; increment the quotient, and compare the dividend with the product. Care is needed here to make the correct decision. Since we load the dividend to A and compare it with the product, carry will be set when the product is <u>greater than</u> the dividend, and cleared when the product is equal to or less than the

dividend. Be sure that you get the right answers both when the integer division is exact and when there is a remainder.


START
Enter Monitor to display result and obtain dividend
Obtain divisor
Clear Product in register C
Set quotient to FF in register B
Load divisor
A
Add product
Save product in C
Increment quotient in B

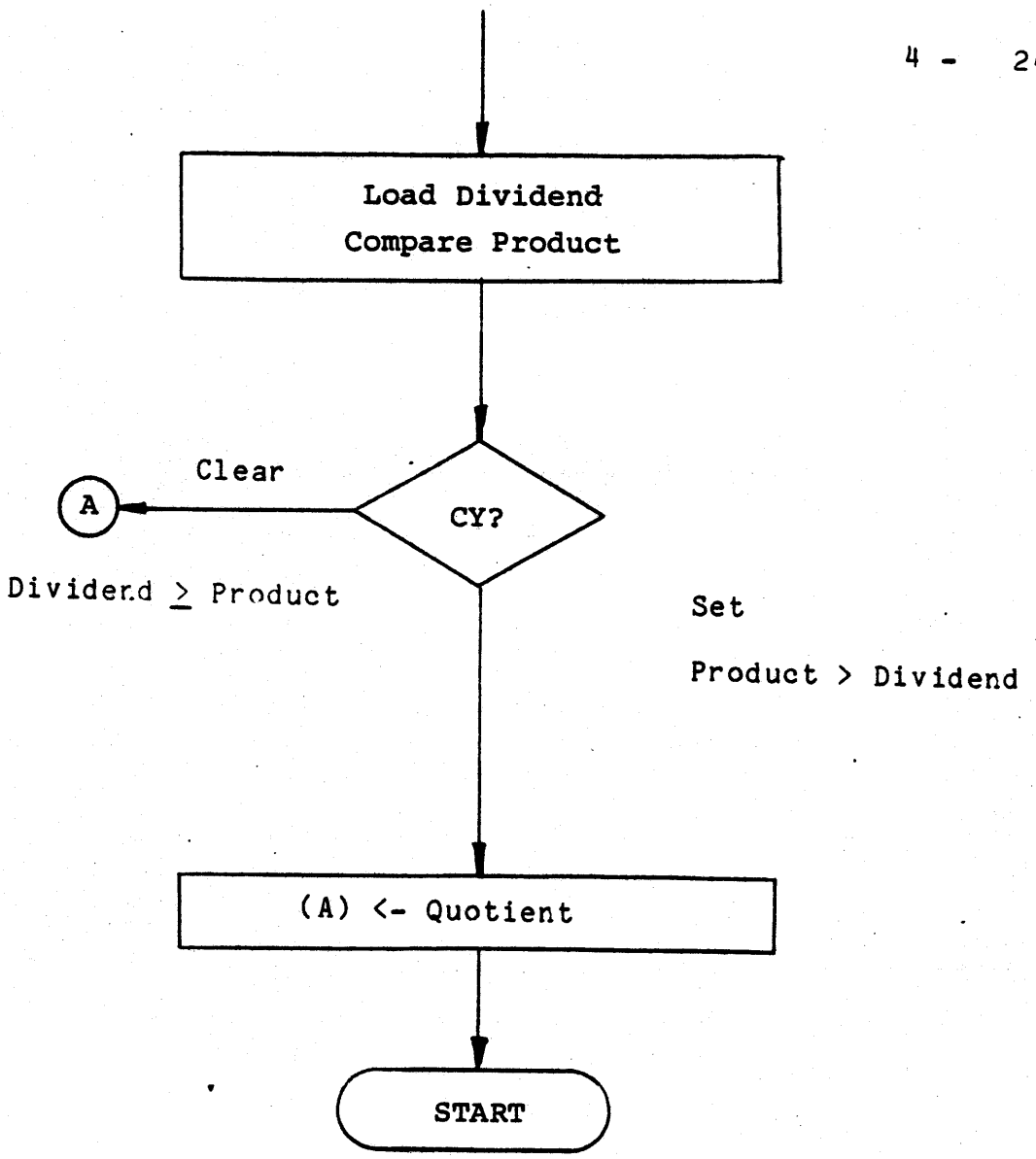

Load Dividend
Compare Product
CY?
Clear
A
Dividend ≥ Product
Set
Product > Dividend
(A) <- Quotient
START

Figure 4-5

Binary Division by Repetitive Addition

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

ADDR	CODE			
8200	00	NOP		Start
01	00	NOP		
02	00	NOP		
03	E7	RST	4	Enter monitor
04	32	STA	8300	for dividend;
05	00			store in memory
06	83			
07	E7	RST	4	Enter monitor
08	32	STA	8301	for divisor;
09	01			store in memory
0A	83			
0B	AF	XRA	A	Clear product
0C	4F	MOV	C,A	(C) ← product
0D	06	MVI	B,FF	Initialize quotient
0E	FF			at -1
0F	3A	LDA	8301	
8210	01			
11	83			
12	81	ADD	C	
13	4F	MOV	C,A	
14	04	INR	B	
15	3A	LDA	8300	
16	00			
17	83			
18	B9	CMP	C	
19	D2	JNC	820F	
1A	0F			
1B	82			
1C	78	MOV	A,B	
1D	C3	JMP	8203	
1E	03			
1F	82			
8220				
21				
22				
23				
24				
25				
26				
27				
28				Figure 4-6

4.4 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.

In this section the term 'transfer notation' is introduced. A capital letter designates a specific 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.

4.4.1 Instruction Effects on Flags

The following register reference instructions and immediate data instructions have been introduced thus far. The list below indicates their effects on the zero (Z) and carry (CY) flags.

INR r	Increment register r (r) <- (r) + 1 If (r) becomes 0 then (Z) <- 1 else (Z) <- 0 The carry flag is not affected.
DCR r	Decrement register r (r) <- (r) - 1 If (r) becomes 0 then (Z) <- 1 else (Z) <- 0 The carry flag is not affected.
MOV d,s	Move data into destination register d from source register s. (d) <- (s) The flags are not affected. The content of s is not affected.
MVI r,data	Move immediate data into register r. Byte 2 of the instruction contains the data. (r) <- (byte 2) The flags are not affected.
ADD r	Add register to accumulator (A) <- (A) + (r) The content of register r is added to the content of register A and the result is placed in the accumulator. The content of register r 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

ADI data

Add immediate data to accumulator
(A) <- (A) + (byte 2)
The content of byte 2 of the instruction is added to the content of register A and the result is placed in the accumulator. Flags are affected as for ADD.

CMP r

Compare accumulator with register
If (A) = (r) then (Z) <- 1
else (Z) <- 0
If (A) < (r) then (CY) <- 1
else (CY) <- 0
The content of A is not affeted.

CPI data

Compare accumulator with immediate data.
If (A) = (byte 2) then (Z) <- 1
else (Z) <- 0
If (A) < (byte 2) then (CY) <- 1
else (CY) <- 0
The content of A is not affected.

XRA A

Clear register A
(A) <- 0
(Z) <- 1
(CY) <- 0

Note XRA r is a logical instruction which operates on the contents of registers r and A and places the result in A. Only when the register specified in the instruction is A (XRA A) does it have the effect of clearing A.

CMP A

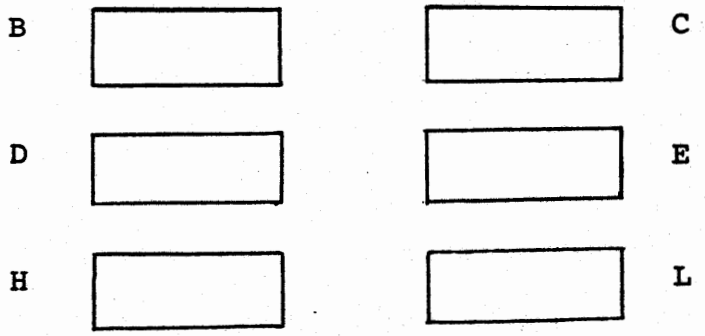
Compare register A with itself. Sets the zero flag and clears the carry flag.
(Z) ← 1
(CY) ← 0

ORA A

Test register A to set condition flags. clear carry.
If (A) = 0 then (Z) ← 1
else (Z) ← 0
always (CY) ← 0

4.5 REGISTER PAIRS

In the foregoing instructions the six general purpose registers (B, C, D, E, H, L) are equivalent to each other. They store data, provide operands for arithmetic and logical instructions, and count. Any one of them will serve as well as another. The general purpose registers are paired:



Their arrangement is like that of the W and Z registers, 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.

4.5.1 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:	0A
MNEMONIC:	LDAX B
MEANING:	Load the A register with the content of the memory location whose address is contained in register pair B,C.

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) <- ((D),(E))
```

The STAX instructions similarly provide for storing data:

02	STAX	B	Store A indirect at B ((B),(C)) <- (A)
12	STAX	D	Store A indirect at D ((D),(E)) <- (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 ((B),(C)) imply the content of the memory location whose address is contained in register pair B,C.

Figure 4-7 illustrates the instruction cycle for STAX D, which typifies this usage of register pairs.

INSTRUCTION CYCLE FOR STAX D INSTRUCTION

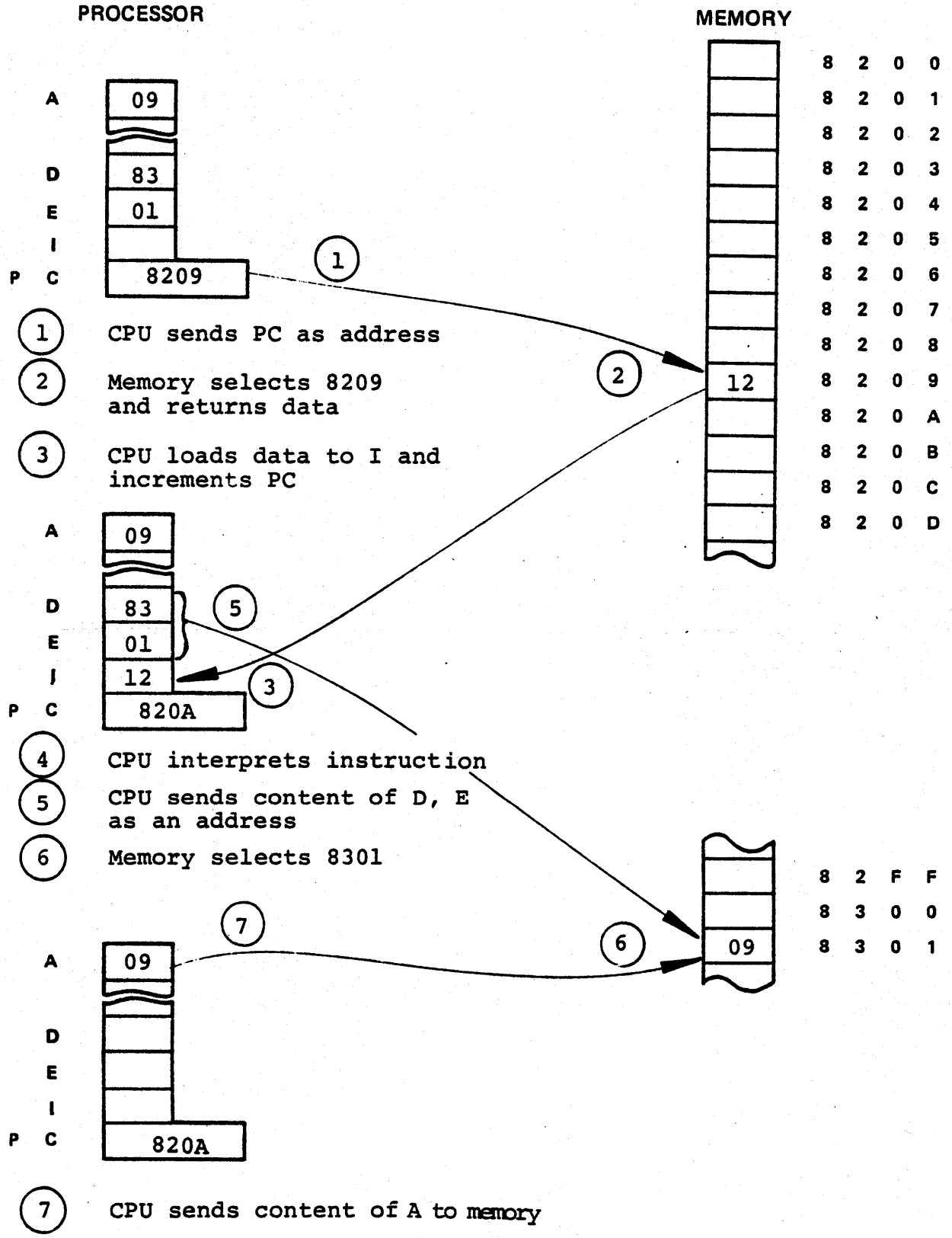


Figure 4-7

4.6 SENSOR CORRECTION EXERCISE, VERSION I

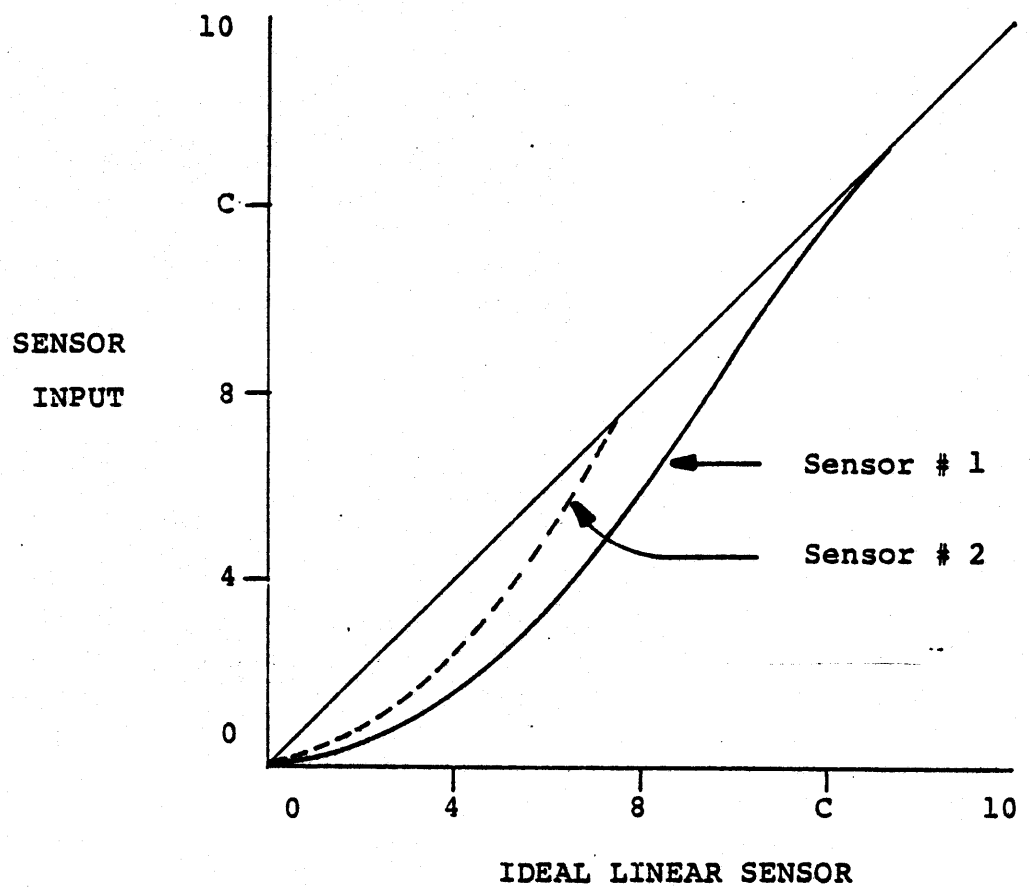
4.6.1 Sensor Characteristics

A sensor is a device for measuring a physical variable such as temperature, pressure, sound, etc. A thermometer, for example, is a device for measuring temperature. Temperature can vary over a tremendous range, of course, and no thermometer can accurately measure all temperatures. Sensors are designed to operate over a limited range of the physical variable they measure.

Even in this range they are not accurate (linear) over the entire scale. A sensor may be calibrated, however, to determine the magnitude of its deviation from linearity for each value that it does measure. This can be shown on a calibration curve, a hypothetical example of which is shown in Figure 4-8.

Notice that each of the calibration curves in Figure 4-8 provides an output lower than the actual value it is meant to measure for low values of the variable, but that both reach a point where they become linear. From these curves we may construct correction tables, which are shown in Table 4-1.

Sensors are often designed to provide readings which differ from their measurement by some factor. An automobile tachometer, for example, measuring the engine's revolutions per minute, gives a reading on a scale of 0 to 8 (generally). This must be multiplied by a scaling factor of 1000 to obtain actual rpms. For our two hypothetical sensors, a scaling factor is also shown in Table 4-1.



SENSOR CALIBRATION CURVES

FIGURE 4- 8

	Sensor #1
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

Scaling Factor 02

	Sensor #2
Sensor Value	Corrected Value
0	0
1	2
2	4
3	4
4	5
5	6
6	7
7	7
> 7	Linear

Scaling Factor 03

Table 4-1

4.6.2 Organizing the Data Structure

We will develop a program to correct a non-linear sensor input value and multiply the result by a scaling factor. In the program the corrected values will be listed in tables. Since the sensors become linear well before full scale, we will store in the table only data for the non-linear area. This gives different table lengths for the two sensors. We will assume that the programmer does not know the table lengths when he designs the program. Since the tables are contiguous, he also does not know the starting address for the second table.

Therefore for each sensor we will store the following information:

a) The starting address for its table

b) The sensor input value at which the sensor has become linear (the linear point)

c) The scaling factor for the sensor

d) The list of corrected values

The starting address for each sensor's table must be accessed knowing only which of the sensors is being read. The remaining data can be included with the correction table. Table 4-2 shows the organization and locations for these data in our data memory.

DATA TABLE FOR SENSORS

ADDR	CODE		
8300	00		RESERVED — Use in version 2
01	08		SENSOR 1 — TABLE ADDRESS
02	16		SENSOR 2 — TABLE ADDRESS
03	00		RESERVED — Table addresses
04	00		for additional
05	00		sensors
06	00		
07	00		
08	02		SENSOR 1 — SCALING FACTOR
09	0B		LINEAR POINT
0A	00		CORRECTED VALUE - INPUT 00
0B	03		01
0C	04		02
0D	05		03
0E	06		04
0F	07		05
8310	08		06
11	09		07
12	09		08
13	0A		09
14	0B		0A
15	0B		0B
16	03		SENSOR 2 — SCALING FACTOR
17	07		LINEAR POINT
18	00		CORRECTED VALUE - INPUT 00
19	02		01
1A	04		02
1B	04		03
1C	05		04
1D	06		05
1E	07		06
1F	07		07
8320			
1			
2			
3			
4			
5			
6			
7			
8			

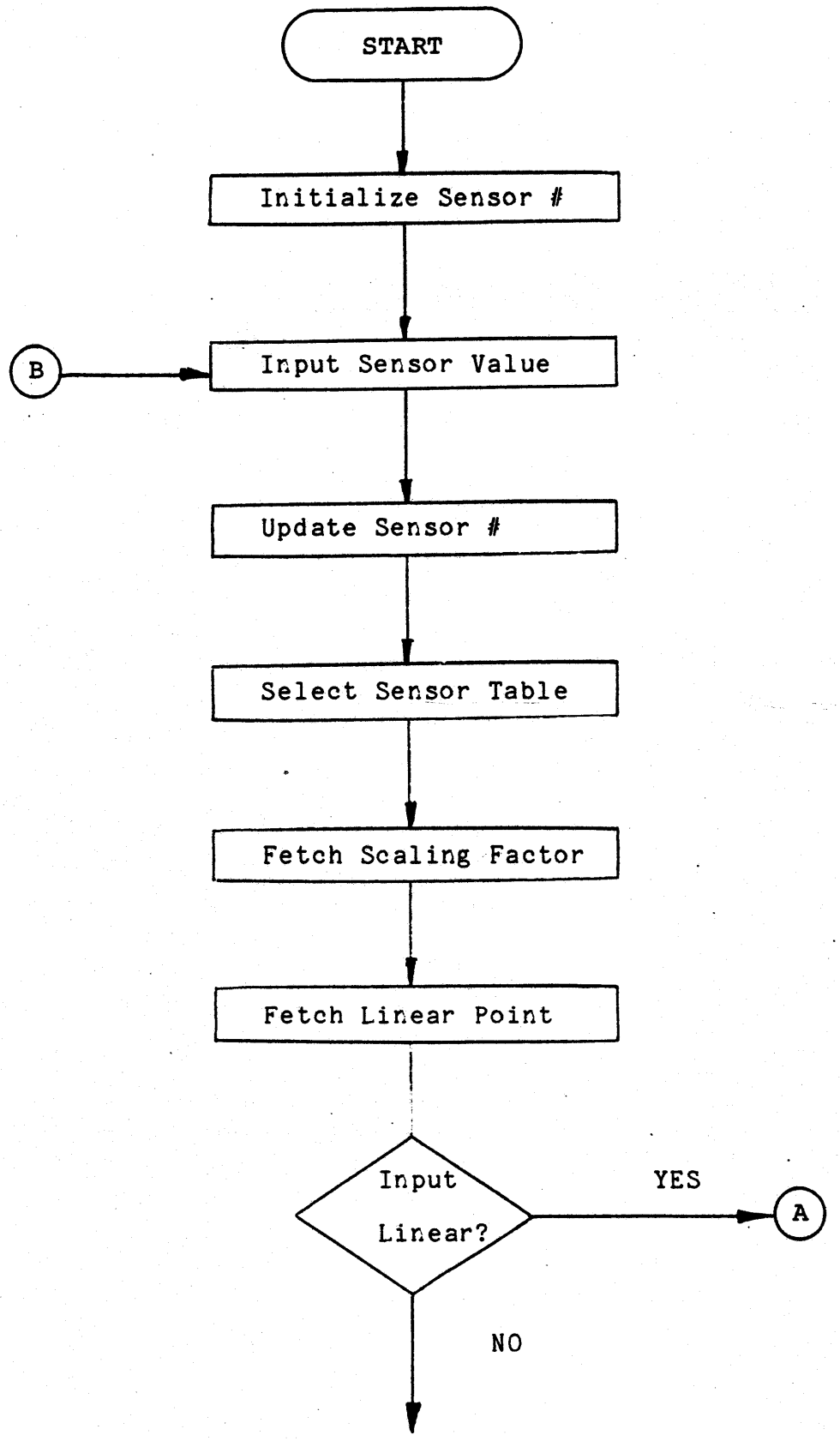
Figure 4-2

4.6.3 Organizing the Program

This exercise will be more complete than previous exercises. The basic program specification is simple: Obtain a sensor value input from one of two sensors, retrieve a corrected value from a table if necessary, multiply the value by a scaling factor, and display the result.

In organizing the program, the assumption is made that all data is stored in tables, and that only the address of the first table is known. A further assumption is that the input data will alternate back and forth between sensors #1 and #2, starting with #1.

We will use the multiplication code developed for the last program, and use the monitor for input and display of results. The procedures for accessing tabular data, however, are new. The design of the data structure in Table 4-2 will dictate the principal organization of the program. Before turning to the flow chart of Figure 4-9, sketch one of your own, then compare it. A program solution is given in Figures 4-10 and 4-11.


START
Initialize Sensor #
B
Input Sensor Value
Update Sensor #
Select Sensor Table
Fetch Scaling Factor
Fetch Linear Point
Input Linear?
YES
A
NO

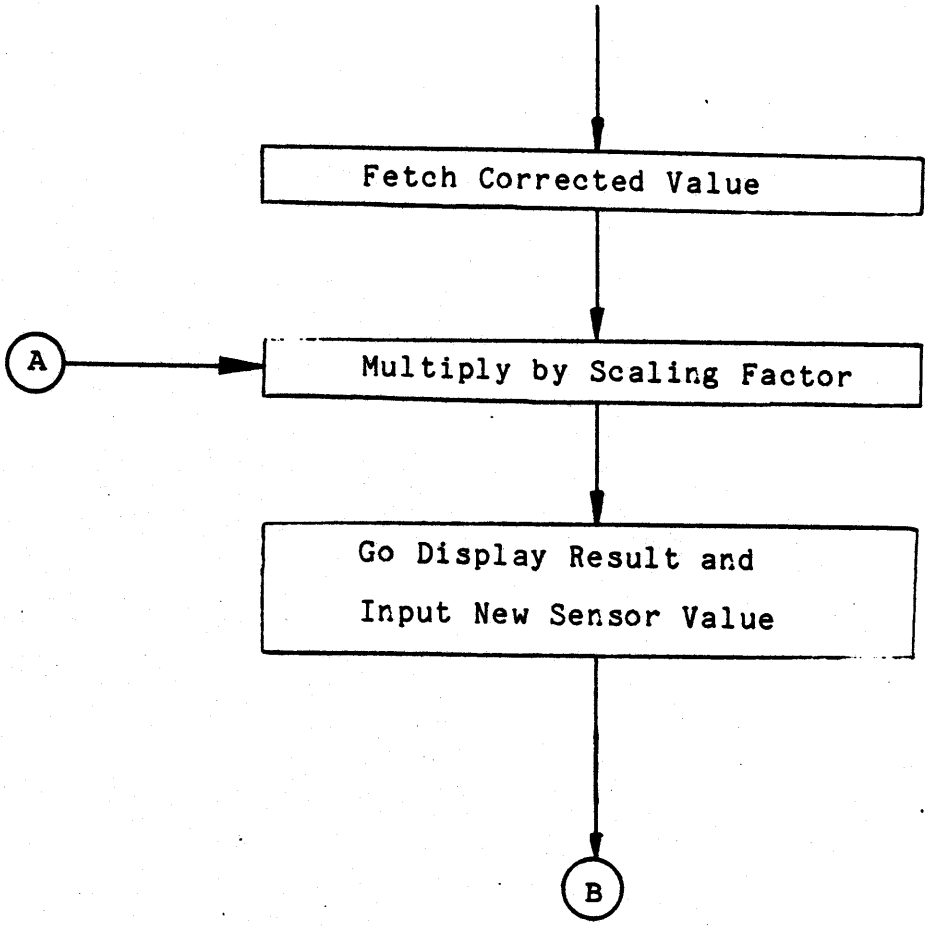

Fetch Corrected Value
A
Multiply by Scaling Factor
Go Display Result and
Input New Sensor Value
B

Figure 4-9

SENSOR CORRECTION PROGRAM

ADDR	CODE														
8200	00		NOP												
01	00														
02	00														
03	3E		MVI		A	,	02								Initialize sensor
04	02														number = 02
05	32		STA		8	3	8	0							Store it
06	80														
07	83														
08	E7		RST		4										Enter monitor for
09	4F		MOV		C	,	A								sensor input
0A	16		MVI		D	,	8	3							Address data
0B	83														memory
0C	1E		MVI		E	,	8	0							Address sensor
0D	80														number
0E	1A		LDAX		D										Load last sensor
0F	3C		INR		A										Next sensor
8210	FE		CPI		03										Is it less than 3?
11	03														
12	DA		JC		8	2	1	7							Jump if less
13	17														than 3 to store
14	82														sensor number
15	3E		MVI		A	,	0	1							Make sensor
16	01														number 1
17	12		STAX		D										Store sensor number
18	5F		MOV		E	,	A								Get sensor data
19	1A		LDAX		D										table address
1A	5F		MOV		E	,	A								Address table
1B	1A		LDAX		D										Load scaling factor
1C	47		MOV		B	,	A								Scaling factor to B
1D	1C		INR		E										Address linear point
1E	1A		LDAX		D										Load linear point
1F	B9		CMP		C										Compare input
8220															
21															
22															
23															
24															
25															
26															
27															
28															

Figure 4-10

INTEGRATED COMPUTER SYSTEMS MICROCOMPUTER TRAINING SYSTEM CODING SHEET

ADDR	CODE			
8 220	DA	JC	8229	Jump if linear
1	29			
2	82			
3	1C	INR	E	Address correction
4	7B	MOV	A, E	table start
5	81	ADD	C	Add sensor input
6	5F	MOV	E, A	Address and load
7	1A	LDAX	D	corrected value
8	4F	MOV	C, A	Corrected value to C
9	AF	XRA	A	Clear A for multiply
A	81	ADD	C	Add linearized input
B	05	DCR	B	Decrement multiplier
C	C2	JNZ	822A	Continue until
D	2A			multiply complete
E	82			
F	C3	JMP	8208	Jump back to
8 230	08			output result
1	82			
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 4-11

Load the program and data tables and verify carefully. Use the solution given first, then try your own solution if it is different. Start at 8203 and press REG A. We will step though the program and describe the operations in some detail. Follow the coding sheet and flow chart as we go:

Move immediate to A	8203	A-??
places 02 in A,	8205	A-02
which is stored in the	8208	A-02
data table as the current sensor number. This is the Initializing procedure.		
From the monitor, we may	0020	A-02
Input a Sensor Value:		
[1]	0020	A-01
The value 1 will be stored	8209	A-01
in register C. Next we will		
Update the Sensor #,		
putting 83 in D	820A	A-01
and 80 in E.	820C	A-01
	820E	A-01
Look at register pair D,E		
[ADDR] [D] [MEM]	8380	DE02

This is a new sequence of keys for inspecting the content of a register pair:

[ADDR] [B,D or H] [MEM]

The content of the register pair appears at the left. The right four locations display the name of the register pair and the content of the memory location addresses by the pair. The display format is not preserved, and must be keyed in each time.

Now we will load (D,E)

820F 3C

[REG] [A]

820F A-02

This part of the code (820a to 8217) updates the current sensor number, which must alternate between 1 and 2 each time.

The sensor number has been incremented from 2 to 3. Now we will test its magnitude with CPI,

8210 A-03

And jump if it is less than 3.

8212 A-03

It is not, so (A) <- 1

8215 A-03

and will be stored in 8380.

8217 A-01

Satisfy yourself that each time we pass through these instructions, the sensor number will alternate between 1 and 2.

By putting the sensor number in E, 8218 A-01

we form address 8301 in D,E; 8219 A-01

ADDR	D	MEM	8301	DE08
REG	A	STEP	821A	A-08

and load its content. The number 08 is an offset (from 8300) which gives us the low-order byte of the address of the first entry of the table for sensor #1 (8308), thus selecting the correct Sensor Table.

821B A-08

ADDR	D	MEM	8308 DE01

Now we Fetch the Scaling Factor for sensor #1,

REG	A	STEP	821C A-02

and store it in B. 821D A-02

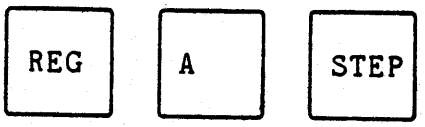
Register pair B,C now contains the scaling factor and input value:

NEXT	821D	B-02
NEXT	821D	C-01
STEP	821E	C-01

Register pair D,E, which holds our table pointer (current address in the table), has been incremented to point to the next entry:

ADDR	D	MEM	8309 DE0B

We will load its content, the Linear Point, and



821F A-0B

compare it with the input value. 8220 A-0B

We are now poised at a decision point. If the sensor value is equal to or greater than the linear point, we do not need to access the correction table.

In this case it is less. 8223 A-0B

To Fetch the Corrected Value,

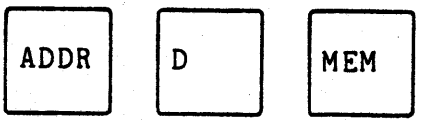
we increment the table pointer, 8224 A-0A

move the low byte to A and 8224 A-0A

add the sensor input. 8226 A-0B

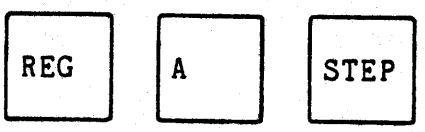
We have computed the value of a table pointer by adding the sensor value to the address of the first correction entry.

Now we return the pointer to E, 8227 A-0B



830B DE03

load the corrected value,



8228 A-03

and substitute it for the input value in (C). 8229 A-03

We multiply by the Scaling Factor, just as we did in section 4.2.

The A register is cleared;	822A	A-00
add corrected input,	822B	A-03
decrement the counter (B),	822C	A-03
loop,	822A	A-03
add input again,	822B	A-06
decrement counter	822C	A-06
and jump out of the loop	822F	A-06
And so back to the beginning,	8208	A-06
to display the results and		
get a value for sensor #2.	0020	A-06

Now RUN the program:

	RST	REG	A	RUN		
					8209	A-??
(Sensor #1)			1	RUN	8209	A-06
(Sensor #2)			2	RUN	8209	A-06
(Sensor #1)			2	RUN	8209	A-08
(Sensor #2)			2	RUN	8209	A-0C

This STEP through of your program is keyed to both the flow chart and the coding sheet. If you are at all confused by it, STEP through it again, following both documents carefully. In addition to illustrating the use of new instructions, this program demonstrates two important

concepts: incrementing an address in a register pair to access successive entries in a table, and the computation of addresses.

4.7 ADDITIONAL INSTRUCTIONS FOR REGISTER PAIRS

4.7.1 Load Immediate, Increment and Decrement

Several additional instructions useful for dealing with register pairs are defined here. They could have been used in the foregoing exercise, although there was no difficulty in programming the problem without them. They are:

```
LXI    rp          (rp refers to a register pair.)
INX    rp
DCX    rp
```

Example:

```
LXI    rp          Load immediate data to register pair:
 xx                (rl) <- (byte 2)
 yy                (rh) <- (byte 3)
```

The content of byte 2 of the instruction is loaded to the low order register (C, E, or L) of the register pair. The content of byte 3 is loaded to the high order register(B, D, or H). The flags are not affected. The LXI instructions are:

```
01   LXI  B
11   LXI  D
21   LXI  H
```

These instructions are most commonly used to load an address pair, but they can equally be used to initialize counters or otherwise enter data into a pair of registers.

Increment and Decrement Instructions are:

	INX rp	Increment Register Pair
03	INX B	(rl) <- (rl) + 1
13	INX D	If (rl) becomes 0 then
23	INX H	(rh) <- (rh) + 1
		Flags are not affected

	DCX rp	Decrement Register Pair
0B	DCX B	(rl) <- (rl) - 1
1B	DCX D	If (rl) becomes FF then
2B	DCX H	(rh) <- (rh) - 1
		Flags are not affected.

These instructions are used almost exclusively to change an address held in a register pair. In the foregoing exercise we could have used INX B instead of INR C, and INX D instead of INR E, with no change in the program's operation. Since all of the table addresses were within 830D, there was no need to alter the high byte of the address, but if the table had started within the 82xx region and ended in the 83xx region, the INX B and INX D instructions would have to be used.

Note that INX and DCX do not affect the flags, whereas INR and DCR affect all flags except carry. This difference is important. In some

applications it is desirable that the flags resulting from a previous operation be retained while a memory address is changed. On the other hand if a loop is to be repeated until a counter reaches zero, the INR or DCR instruction must be used to set or clear the zero flag.

4.7.2 Use of a Memory Location as a Register

Register pair H,L is primarily intended for addressing memory, and the memory location addressed by (H,L) is available to the CPU as though it were another register. All of the register reference instructions (MOV, MVI, INR, DCR, ADD, XRA, ORA, CMP, and others not yet presented) have counterparts that perform the same function using the memory location addressed by (H,L). 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 the instructions involving memory reference, and also several instructions specific to register pair H,L.

4.7.3 Memory Reference Instructions

INR M

Increment Memory
((H)(L)) <- ((H)(L)) + 1
Increment the content of the memory location addressed by the content of register pair H,L.
If ((H)(L)) becomes 0 then (Z) <- 1
else (Z) <- 0
The carry flag is not affected.

DCR M

Decrement Memory
((H)(L)) <- ((H)(L)) - 1
Decrement the content of the memory location addressed by the content of register pair H,L.
If ((H)(L)) becomes 0 then (Z) <- 1
else (Z) <- 0
The carry flag is not affected.

MOV M,s

Move data into memory
((H)(L)) <- (s)
The memory location addressed by the register pair H,L is loaded with the content of source register s.
The flags are not affected.
The content of s is not affected.

MOV d,M

Move data from memory
(d) <- ((H)(L))
Destination register d is loaded with the content of the memory location addressed by register pair H,L
The flags are not affected.
The content of the memory location is not affected.

MVI M,data

Move immediate data into memory
((H)(L)) <- (byte 2)
The memory location addressed by register pair H,L is loaded with the content of byte 2 of the instruction.
The flags are not affected.

ADD M

Add memory to accumulator
(A) <- (A) + ((H)(L))
The content of the memory location addressed by register pair H,L 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

CMP M

Compare accumulator with memory
If (A) = ((H)(L)) then (Z) <- 1
else (Z) <- 0
If (A) < ((H)(L)) then (CY) <- 1
else (CY) <- 0
The contents of A and ((H)(L)) are not affected.

4.7.4 Additional Instructions for H,L

The following instructions specifically involve register pair H,L. Their primary function is for use in addressing memory, although the DAD instruction is also very useful in arithmetic.

DAD rp

Add the content of register pair rp to the content of H,L.
(H),(L) <- (H),(L) + (rh) (rl)
If the result of the addition is greater than FFFF, then (CY) <- 1
else (CY) <- 0

The HEX codes for DAD instructions are:

09 DAD B

19 DAD D

29 DAD H

In the sensor correction exercise this instruction could have been used to add the table address (in pair D,E) to the input value (in register L), but the scaling factor would have to be held elsewhere and register H set to zero. This will be used in the revised version of the sensor correction problem at the end of this chapter.

The DAD instruction performs a 'double precision' add: two bytes or 16 bits are involved. DAD H adds the content of H,L to itself - that is, the content is doubled. Another useful instruction is:

EB	XCHG	Exchange H,L with D,E (E) <-> (L) (D) <-> (H) The contents of registers H and L are exchanged with the contents of D and E. The flags are not affected.

This is one of only two exchange instructions in the 8080. All other data transfer instructions are one-way: the content of the source register or memory location is duplicated while the previous content of the destination is lost. In the XCHG instruction the previous content of all four registers are preserved but in different registers. It is especially useful when two different memory pages are successively accessed.

The content of (H,L) may be loaded and stored using LHLD and SHLD:

2A	LHLD	Load H and L Direct (L) <- ((byte 3)(byte 2)) (H) <- ((byte 3)(byte 2) + 1) The content of the location addressed by byte 3 and byte 2 of the instruction is moved to register L. The content of the memory location at
xx		
yy		

		the succeeding address is moved to register H.
22	SHLD xx yy	Store H and L Direct ((byte 3)(byte 2)) <- (L) ((byte 3)(byte 2) + 1) <- (H) The content of register L is moved to the memory location addressed by byte 3 and byte 2 of the instruction. The content of register H is moved to the memory location at the succeeding address.

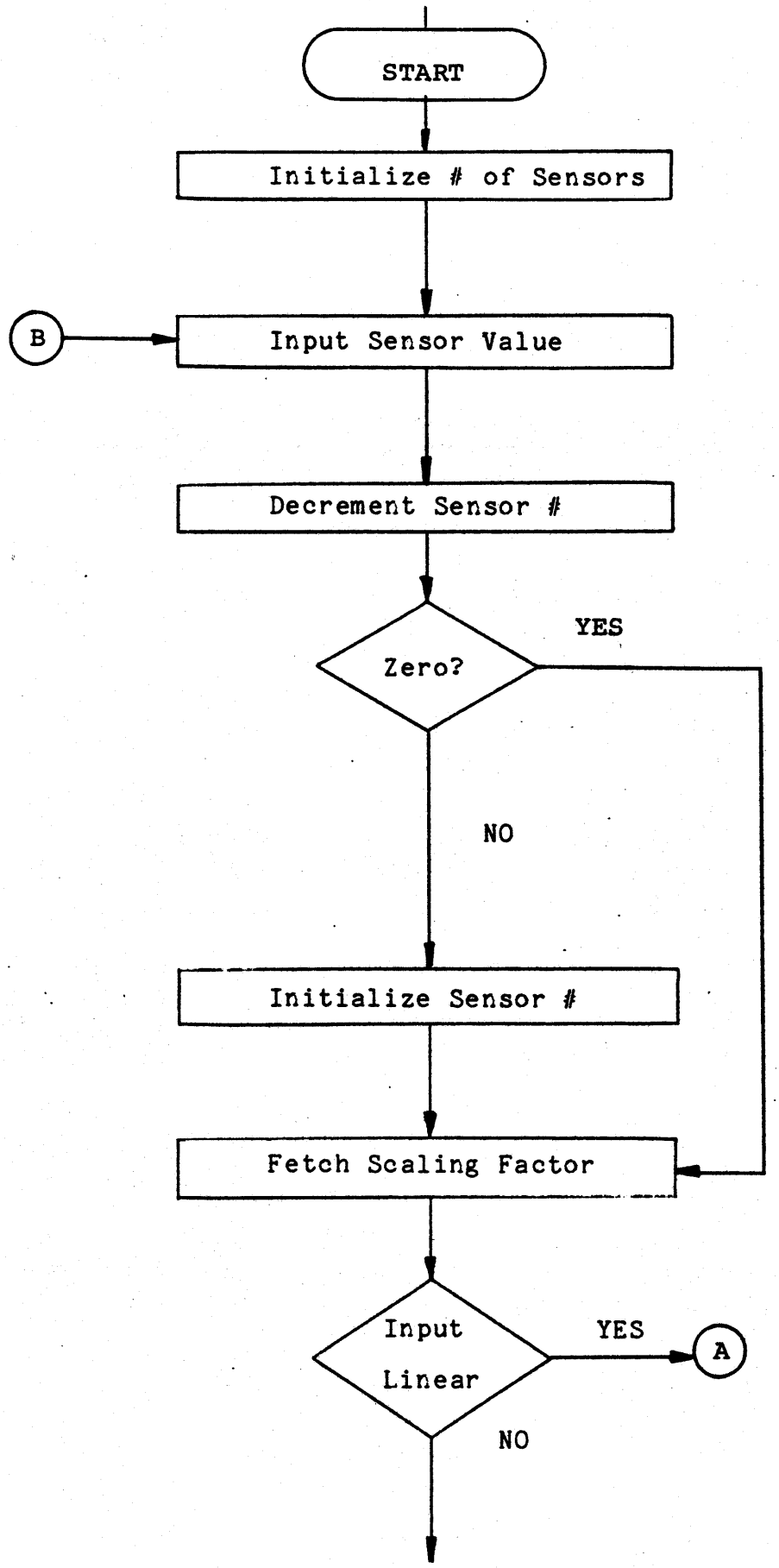
4.8 SENSOR CORRECTION, VERSION 2

In the following exercise we will duplicate the sensor correction program of Table 4-9 with three exceptions. The data table (Table 4-2) will store the number of sensors which will be used, so that it need not be part of the program. We will address the data table with register pair H,L instead of D,E and use memory reference instruction such as MOV A,M, and do a double precision multiply for the scaling.

4.8.1 Double Precision

Double precision means that a number is stored in two bytes, giving a precision of 16 bits (one part in 65,536). It is often the case that one byte (one part in 256) of precision is sufficient, but in multiplication or division we can use double precision in the operation and then discard the less significant part of the result. In our earlier scaling, having only a single precision multiply forced us to restrict the input and scaling factors to single digit values. With a double precision multiply we can use full bytes for both input and scaling factor, multiply to obtain a four byte result, and output the high order byte.

The revised flow chart and coding are presented in Figures 4-12 through 4-14.


START
Initialize # of Sensors
B
Input Sensor Value
Decrement Sensor #
Zero?
YES
NO
Initialize Sensor #
Fetch Scaling Factor
Input
Linear
YES
A
NO

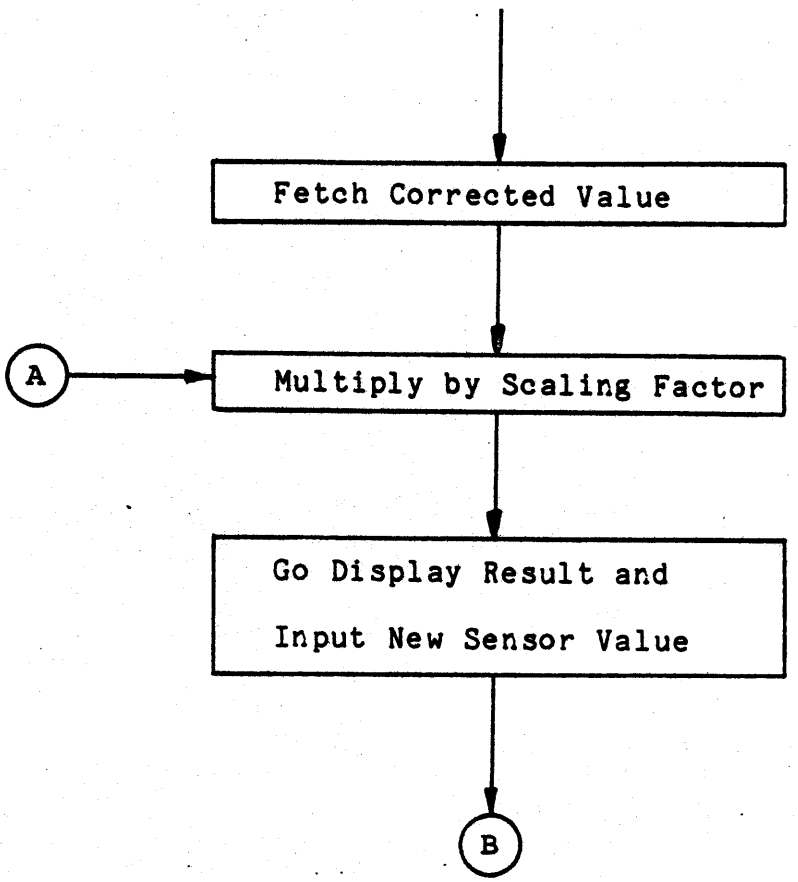

Fetch Corrected Value
A
Multiply by Scaling Factor
Go Display Result and
Input New Sensor Value
B

Figure 4-12

SENSOR CORRECTION PROGRAM

ADDR	CODE						Comments
8200	00		NOP				Save 3 spaces
01	00						
02	00						
03	3A		LDA		8300		Load highest
04	00						sensor number
05	83						
06	32		STA		8380		Store it at 8380
07	80						
08	83						
09	E7		RST		4		Enter monitor
0A	4F		MOV		C, A		Input to C
0B	06		MVI		B, 00		Clear B for
0C	00						later addition
0D	21		LXI		H, 8380		Address previous
0E	80						sensor number
0F	83						
8210	35		DCR		M		Decrement it to
11	C2		JNZ		8219		next sensor number
12	19						and test for zero
13	82						
14	3A		LDA		8300		Load highest
15	00						sensor number
16	83						and store it as
17	77		MOV		M, A		the new sensor
18	79		MOV		A, C		Input to A again
19	6E		MOV		L, M		Address and load
1A	6E		MOV		L, M		data table address
1B	5E		MOV		E, M		Scaling factor to E
1C	23		INX		H		Address linear point
1D	BE		CMP		M		Compare sensor input
1E	D2		JNC		8224		Jump if linear point
1F	24						less than or
8220	82						equal to input
21							
22							
23							
24							
25							
26							
27							
28							

Figure 4-13

ADDR		CODE													
8	0														
8222	1	23			INX			H							Address corrected values
	2	09			DAD			B							Add sensor input
	3	4E			MOV			C , M							Corrected value to C
	4	60			MOV			H , B							Clear H and L
	5	68			MOV			L , B							(reg B was 0)
	6	09			DAD			B							Add linear input
	7	1D			DCR			E							Decrement multiplier
	8	C2			JNZ			8226							Continue until
	9	26													multiply complete
	A	82													
	B	7D			MOV			A , L							Result to A
	C	C3			JMP			8209							Jump to output
	D	09													
	E	82													
	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 4-14

4.8.2 Running the Program

Load the new program. You must also load the data table (Table 4-2) if it is not still in your memory. Enter 02 at location 8300, for the highest sensor number.

Now reset and press REG, A, RUN to arrive at the data input point. From here we will trace the data in the processor.

RST	REG	A	RUN	820A	A-02

Leave 02 as the input value. We are about to move the input value into C, clear B, and load registers H and L with an LXI H instruction.

Step three times and observe the registers:

	820B	A-02
	820D	A-02
	8210	A-02
NEXT	8210	B-00
NEXT	8210	C-02

The content of register pair H,L addresses the memory location where the old sensor number is stored:

ADDR	8/H	MEM	8380	HL02
			8211	C2

ADDR H MEM

8380 HL01

Since the content of 8380 did not reach zero, the JNZ instruction (C2) will cause a jump:

8219 6E

The instruction at 8219 is MOV L,M. The content of memory location 8380 will be moved into L, so the memory address will become 8301, pointing to the table address for sensor number 1:

821A 6E

ADDR H MEM

8301 HL08

Another MOV L,M will put the table address into H,L, and point to the scaling factor:

821B 5E

ADDR H MEM

8308 HL02

At 821B we have MOV E,M to save the scaling factor in register E, and then INX H to address the linear point:

821C 23

821D BE

ADDR H MEM

8309 HL0B

The next instruction (BE at 821D) compares the linear point (0B at 8309) with the content of register A. Before executing it, review the

registers:

REG	A	(sensor input)	821D	A-02
NEXT		(B cleared)	821D	B-00
NEXT		(sensor input)	821D	C-02
NEXT		(not used)	821D	D-??
NEXT		(scaling factor)	821D	E-02
NEXT		(flags - ignore)	821D	F-2A
NEXT		(high address)	821D	H-83
NEXT		(low address)	821D	L-09

The following instructions compare the sensor input with the linear point, and finding the input not greater the jump to 8224 is not taken:

821E L-09

8221 L-09

At 8221 register pair H is incremented to address the first point in the table of corrected values:

8222 L-0A

At 8222 register pair B,C (containing 0002) is added to register pair HL (containing 830A):

8223 L-0C

This addresses the linearized value for a sensor input of 02:

ADDR H MEM 830C HL04

The following three instructions move that value to C and clear H and L. We are finished with H and L for addressing and now need them for the double precision multiply:

8224 60

8225 68

8226 09

Before starting the multiplication review the registers again.

REG A 8226 A-02

NEXT 8226 B-00

NEXT 8226 C-04

NEXT 8226 D-??

NEXT	8226	E-02
NEXT	8226	F-8A
NEXT	8226	H-00
NEXT	8226	L-00

In the multiplication we will add (BC) = 0004 into (HL) = 0000 as we count down in register E from 02 to 00. You can watch this in register L.

8227	L-04
8228	L-04
8226	L-04
8227	L-08
8228	L-08
822B	L-08

Register E has been counted down to zero and the program has exited from the loop. At 822B the single precision result is moved into A from L, and then the jump back to the monitor entry occurs:

		822C	L-08
REG	A	822C	A-08
		8209	A-08
STEP		0020	A-08

By using DAD B in the multiplication loop we have computed a double precision product, but have only looked at the low-order part (L). Change the scaling factors at 8308 and 8316 to C0 (making both the same will produce identical results for each sensor except in their non-linear regions). Run the program for various inputs. Each time the program returns to the monitor, display the contents of H,L to see the complete multiple precision result.

RST	REG	A	RUN	820A	A-02
	1	6		820A	A-16
	RUN			820A	A-80

ADDR	H	MEM		1080	HLCE

For input values of B or greater, both sensors give the same result:

REG	A	1	6	820A	A-16
	RUN			820A	A-80
	ADDR	H	MEM	1080	HLCE

4.9 SUMMARY

In this exercise we have seen register pair H,L used as a store for a memory address, which we modified in four ways:

a) Loading it initially with LXI H

b) Copying data from an addressed memory location into L, with MOV L,M.

c) Incrementing it, with INX H.

d) Adding a variable to the address, with DAD B.

We have also seen the memory location addressed by H,L used as a counter (DCR M) and for a comparison (CMP M), in each case affecting the flags. We have seen it as a source register (MOV L,M; MOV E,M; MOV C,M) and as a destination register (MOV M,A).

Finally we observed register pair H,L used for addition with DAD B, in the multiplication loop as well as in addressing.

4.10 INSTRUCTION CARD

The instruction card shows all of the 8080 instructions. Most of the data transfer and counting instructions have now been introduced, as well as a few of the arithmetic and branch instructions. Study the organization of this chart so that you can readily find an instruction when you need it.

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	

INTEGRATED COMPUTER SYSTEMS, INC.
4445 Overland Avenue / Culver City, California 90230 USA / Tel: (213) 559-9265 / TWX: 910-340-6350
European office: Boulevard Louis Schmidt 84, Bte 6 / 1040 Brussels, Belgium / Tel: (02) 735 6003 / Telex: 62473

MICROCOMPUTER TRAINING WORKBOOK

CHAPTER 5

MEMORY HARDWARE

5. 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:

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

5.1 MEMORY TECHNOLOGY

A memory device includes semiconductor circuits or elements to serve four functions:

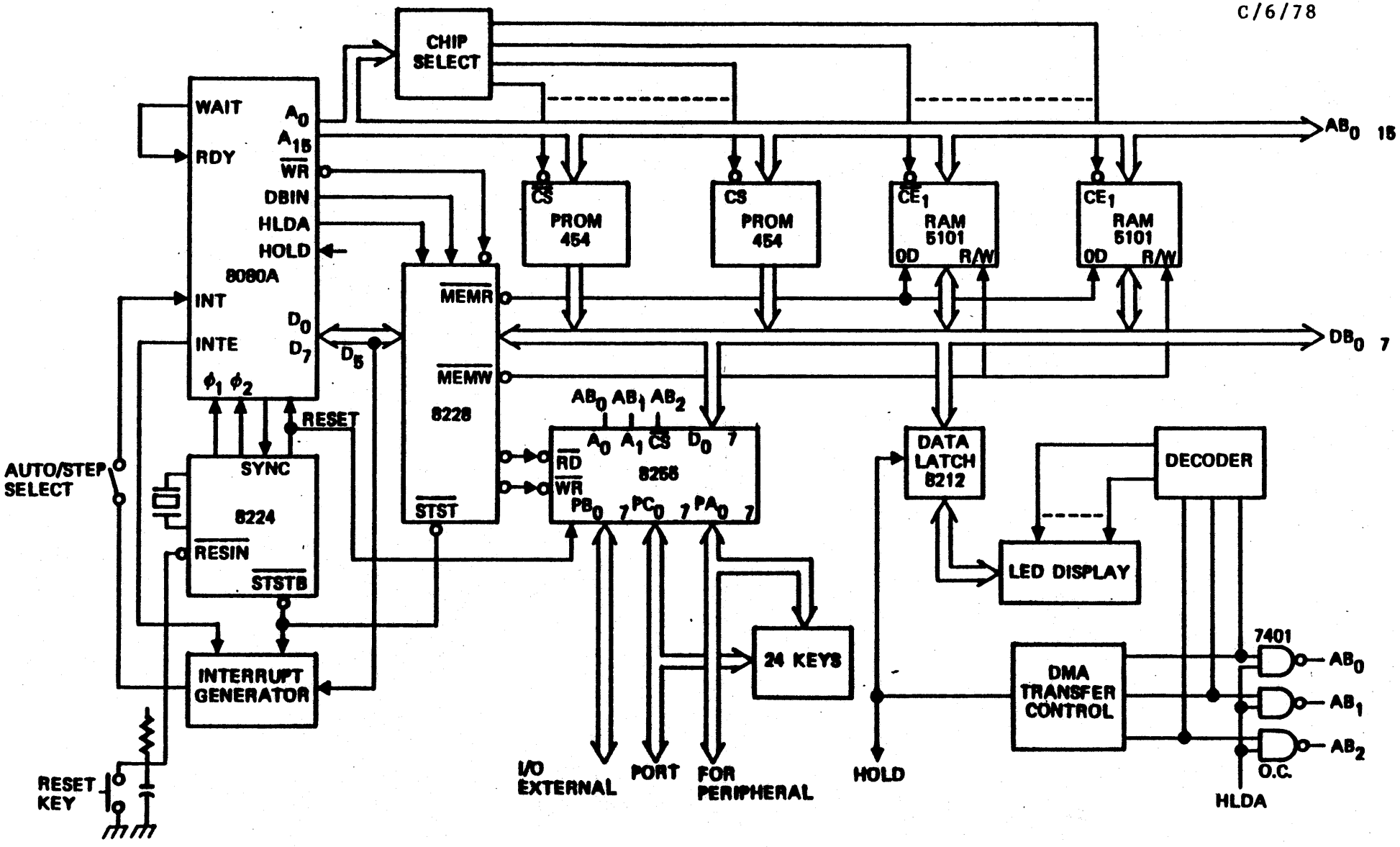
a) Store data in an ordered array

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 256 locations, addressed by the low-order eight bits of the system address bus. The ROM and RAM memories of your MTS system are shown in the schematic diagram, Figure 5-1. 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. This convention is illustrated in Figure 5-2.



MICROCOMPUTER TRAINING SYSTEM CONFIGURATION

FIGURE 5-1

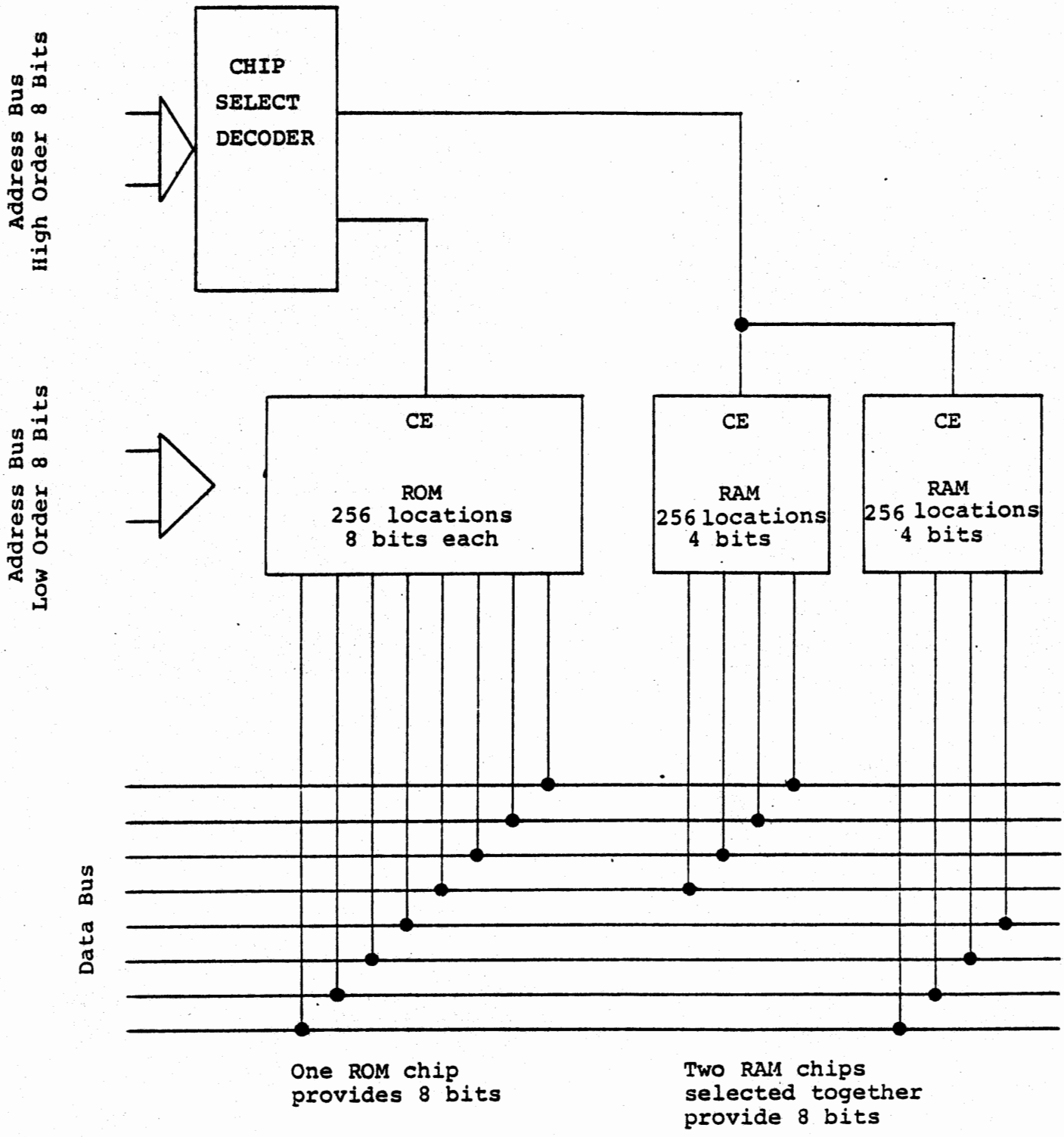

Address Bus
High Order 8 Bits
CHIP
SELECT
DECODER
Address Bus
Low Order 8 Bits
CE
ROM
256 locations
8 bits each
CE
RAM
256 locations
4 bits
CE
RAM
256 locations
4 bits
Data Bus
One ROM chip
provides 8 bits
Two RAM chips
selected together
provide 8 bits

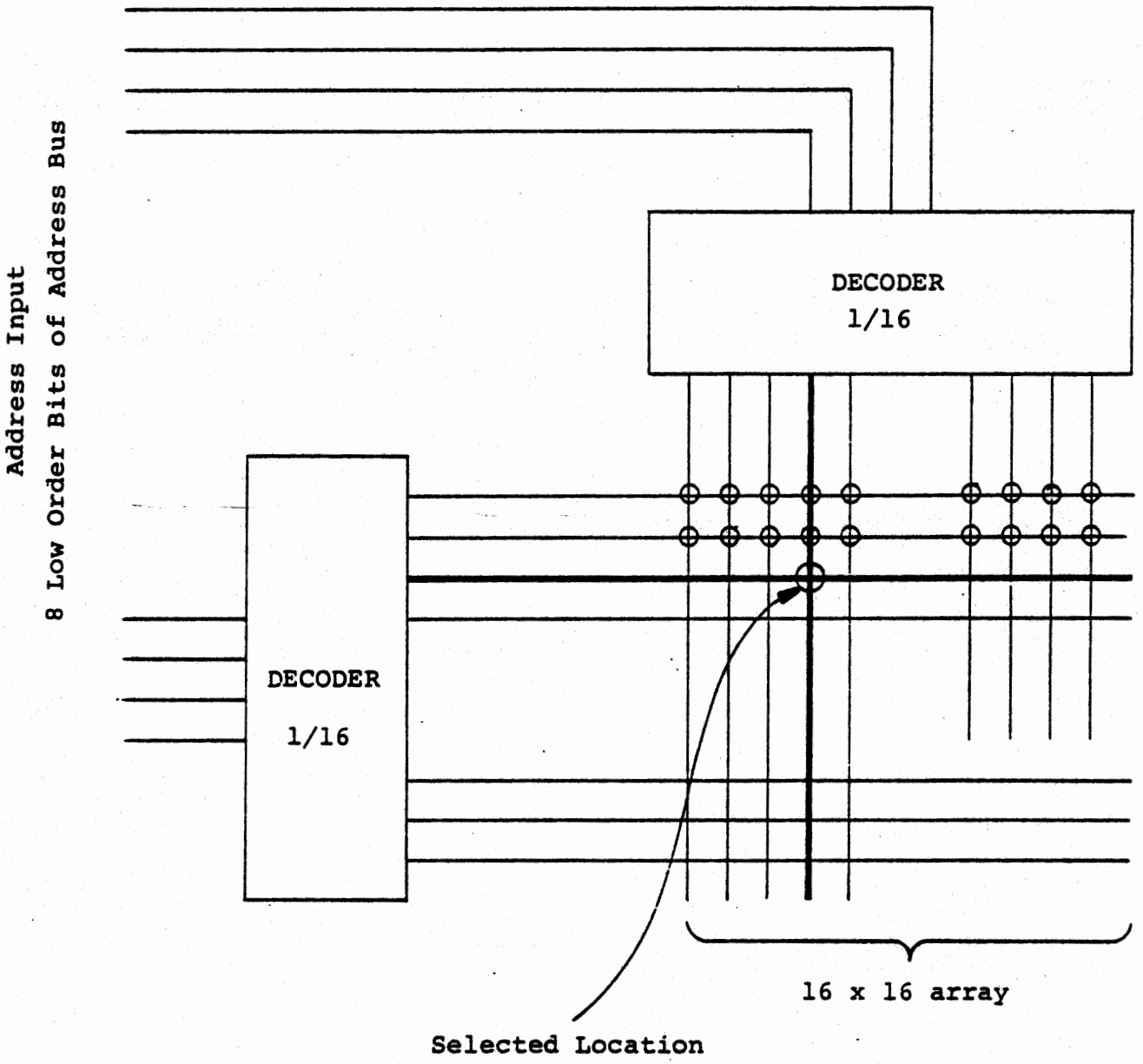
Figure 5 - 2

5.1.1 Storage Techniques

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 electrically eraseable and Programmable Read only Memory (PROM) devices, such as the MTS' 454, 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 microcomputer has no facility for applying such high energy signals, so the PROM cannot be altered while it is in the circuit. Other types of PROMs are erased by exposure to an intense ultraviolet light, and may then be reprogrammed electrically.

In read-write memory the data are stored in the form of current or charge in transistors. Static RAMs, such as the MTS' 5101, 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.

The MTS memory devices have an array of 256 storage locations, each arranged as a square 16 cells high and 16 cells wide. The eight address lines received by the device are divided into two groups of four lines. Each group is decoded to select one of 16 lines, as shown in Figure 5-3. The intersection of the two lines is the selected location. Gates at that location connect the input and output of the storage circuit to the control circuitry within the device. This array is replicated four times at each address to provide the four bits stored by the RAM device, or eight times in the ROM.



INTERNAL ADDRESS DECODING

IN A MEMORY DEVICE

Figure 5 - 3

5.1.2 Chip Select Logic

Every memory device in the system receives the eight low order lines from the address bus, decodes the bit patterns, selects one location and connects it internally. The high order eight bits of the address bus are decoded externally to select one ROM or two RAMs. In an 8080 computer system with 65,536 bytes of memory, the high order address would have to be fully decoded to select among 256 separate memory devices (or pairs of devices).

The MTS is equipped with four ROM chips (1024 bytes) and two pairs of RAM chips (512 bytes), with provision for two additional pairs of RAM chips. It is therefore necessary to decode only eight of the possible 256 high order addresses. This is accomplished by a single 2155 address decoder, which has three address inputs and eight decoded outputs. Each output is connected to one ROM chip or to one pair of RAM chips.

The decoding is thus incomplete: three of the high order address bits enter the 2155 and the other five are ignored. In this configuration the physical memory appears to be replicated 32 times. You can test this with your microcomputer. Press ADDR and enter any of these addresses:

0000, 0400, 0800, 0C00, 1000, 1400, ...7C00

The same data (31) will be seen at each address because the same monitor (ROM) location has been selected in each case.

In your own program memory (RAM) you may also substitute addresses at

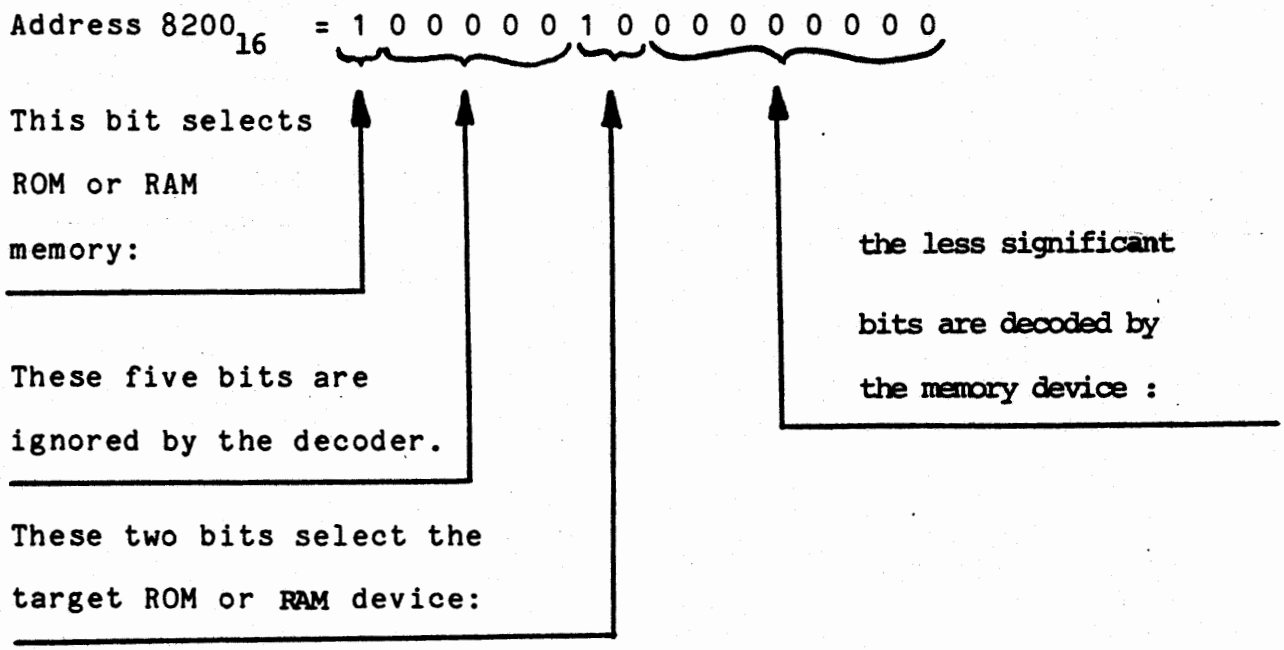
intervals of 400_{16} ; for example:

8600 instead of 8200

9A01 instead of 8201

FE02 instead of 8202

The address bits decoded for chip selection are the highest bit (A15), which distinguishes ROM (high bit = 0) from RAM (high bit = 1), and the two low order bits (A8 and A9). The following diagram will clarify:



Provision is made on the circuit board for an additional input to the 2155 address decoder to disable all of its chip select lines so that external memory can be added using a different decoder, but it is hard to imagine this being appropriate. Programs needing more than 1024 bytes of RAM generally belong in expensive development systems with text editors, assemblers, compilers, and floppy disks.

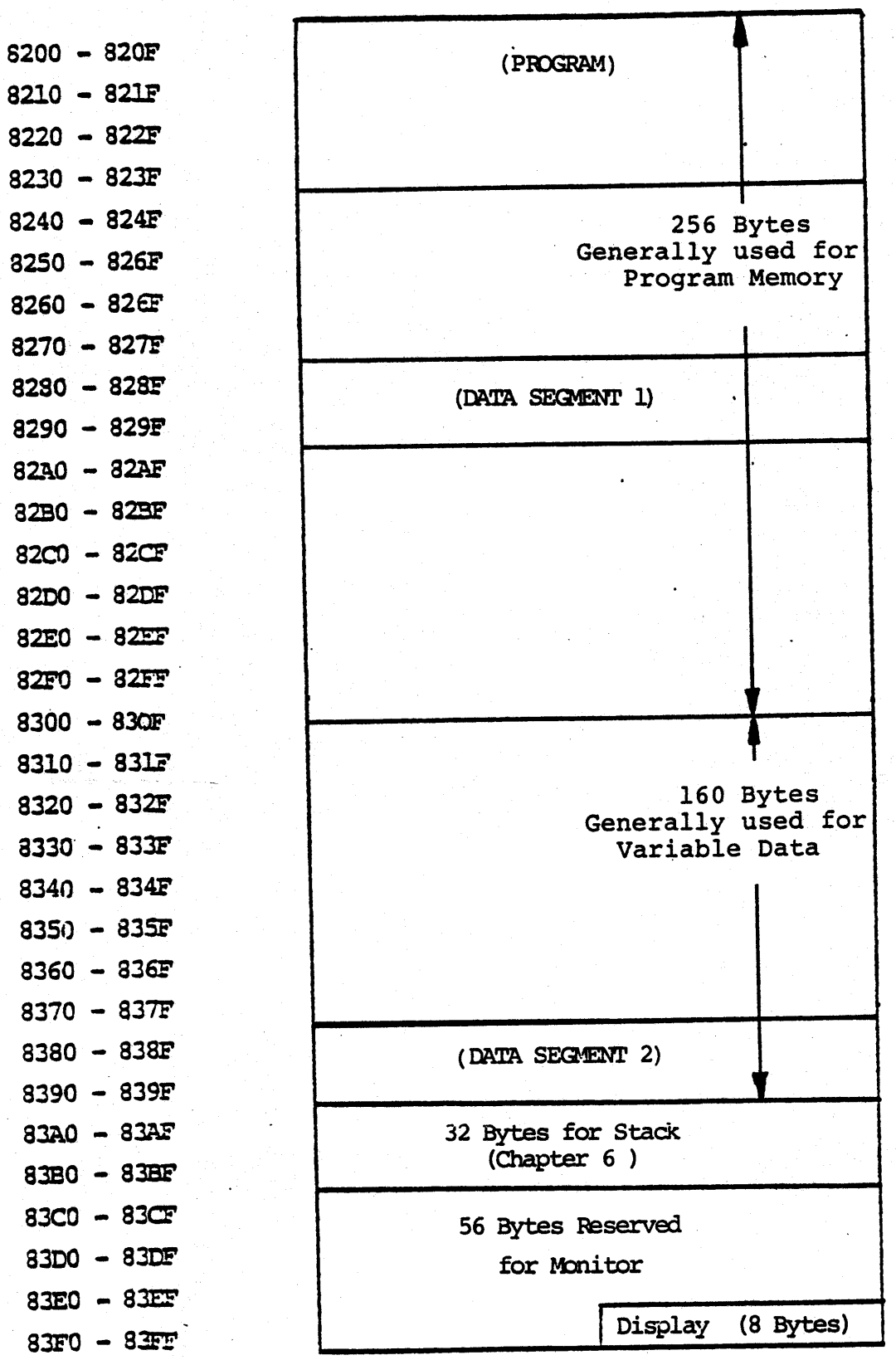
5.2 MEMORY PAGES

All 256 bytes of an MTS memory device have the same high address (e.g. 82) and all possible low addresses (i.e. 00 through FF). This is called a page of memory. With small memory devices it corresponds to a physical separation: a single 454 ROM chip or a pair of 5101 RAM chips is one page. This affects addressing, since only the low-order bytes of addresses change within a given page.

For example, you could clear data memory (8300 through 839F) with this program:

```
8203  LXI  H,83A0        Load address 83A0 in H,L
8204  A0
8205  83
8206  DCR  L             Decrement L
8207  MVI  M,00          Store zero in address (H,L)
8208  00
8209  JNZ  8206          Loop until L = zero
820A  06
820B  82
```

There may be occasion to use addresses with the same low-order byte in two separate pages for data, e.g. for storing argument pairs. This involves a violation of the division suggested above between program and memory. That division is not sacred, however, and memory should be used as efficiently as possibile. It is often useful to make a memory map for any program that is divided into modules, or if large areas of memory are used for variable data. Figure 5-4 illustrates such a map.



MEMORY MAP

READ WRITE MEMORY

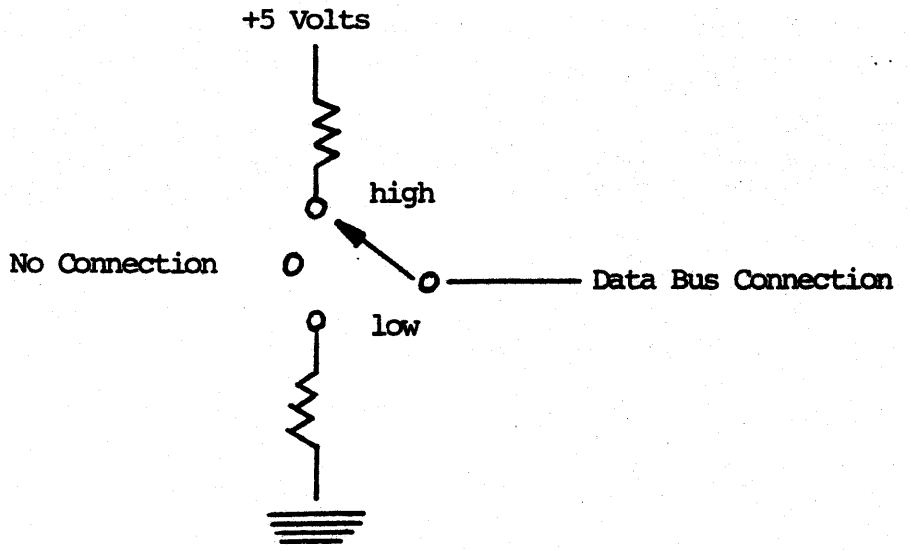
Figure 5 - 4

5.3 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).

5.3.1 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 8228 System Controller, each 454 ROM and 5101 RAM (on the data bus only), and the 8255 Peripheral Interface.

5.3.2 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 8228 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 (via the 8228 and the control bus) 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.

5.3.3 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.

5.4 MEMORY SIGNALS AND TIMING

5.4.1 Machine States and Transitions

Figure 5-5 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 ø 1 and ø 2 clocks, as shown. SYNC is generated by the CPU at the beginning of each machine cycle. The ø 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.

5.4.2 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.

MEMORY ACCESS TIMING

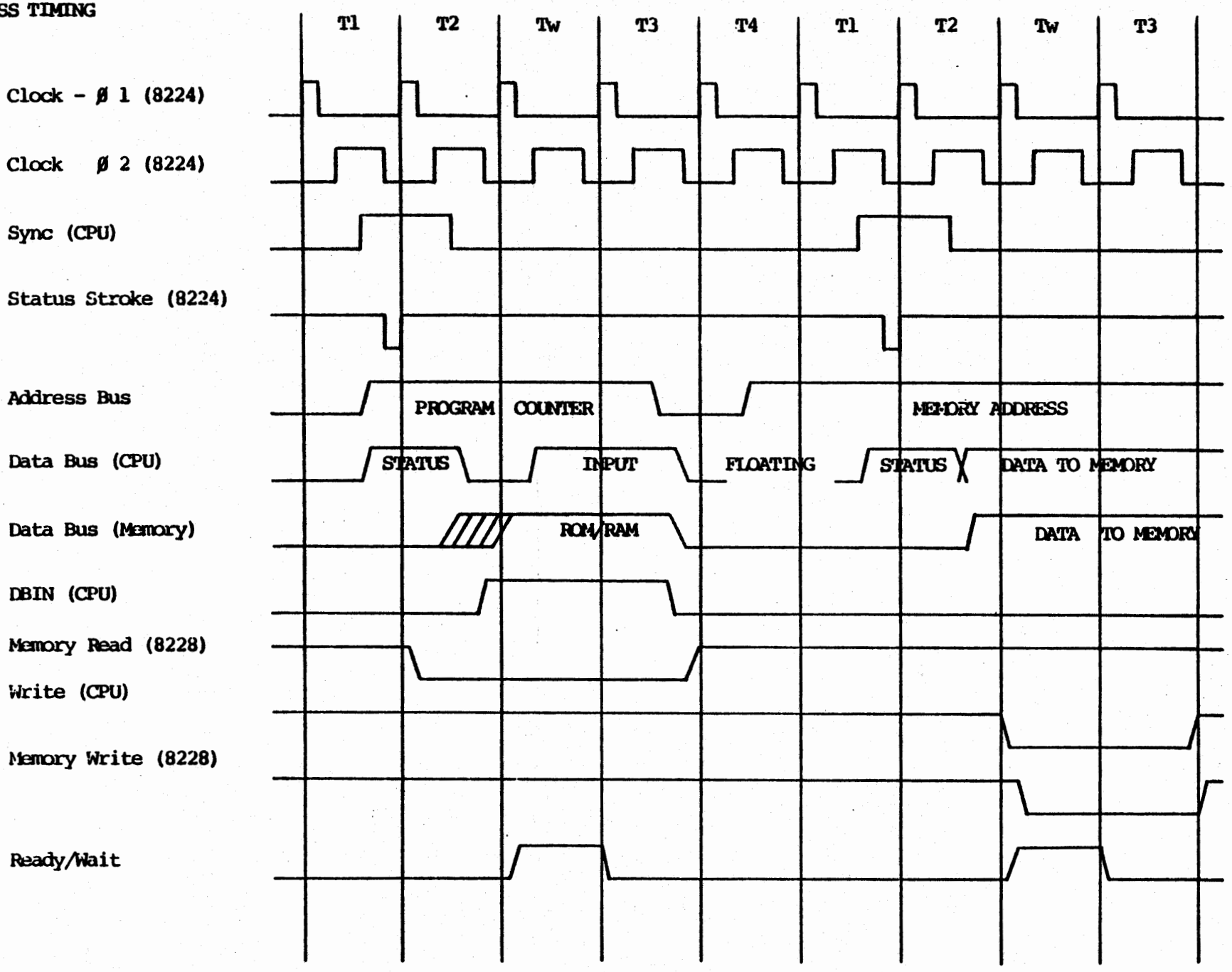


Figure 5 - 5

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 (8 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. Because the controller also isolates the CPU data bus from the system data bus, it is permissible for the memory read to overlap the status output from the CPU.

5.4.3 Second State (T2) and Wait State (TW)

During state T2 a signal (DBIN) is raised to indicate that the processor is ready to receive data. The DBIN signal is terminated during state T3. CMOS RAM is relatively slow: it may not have data ready and on the data bus by the time the CPU is ready for it at the end of T2. To provide for this, if the 8080 READY signal is low at the end of T2 the CPU enters a WAIT state, Tw. If the READY signal is generated externally the WAIT state lasts indefinitely (but always an integral number of clock periods) until the READY signal becomes high. When it enters this state the CPU outputs a WAIT signal.

In the MTS the READY signal is not generated externally. It simply connects the CPU's WAIT output to its READY input. Therefore the CPU always finds READY low (i.e. not ready) at the end of T2, enters the WAIT state Tw, raises the WAIT signal, and at the end of one clock period finds the READY signal high. It then enters T3, drops the WAIT output, and proceeds to read data from the data bus. Even though the ROM is fast enough to need no waiting period this system always provides it, since it does not know the source of the instruction: RAM or ROM.

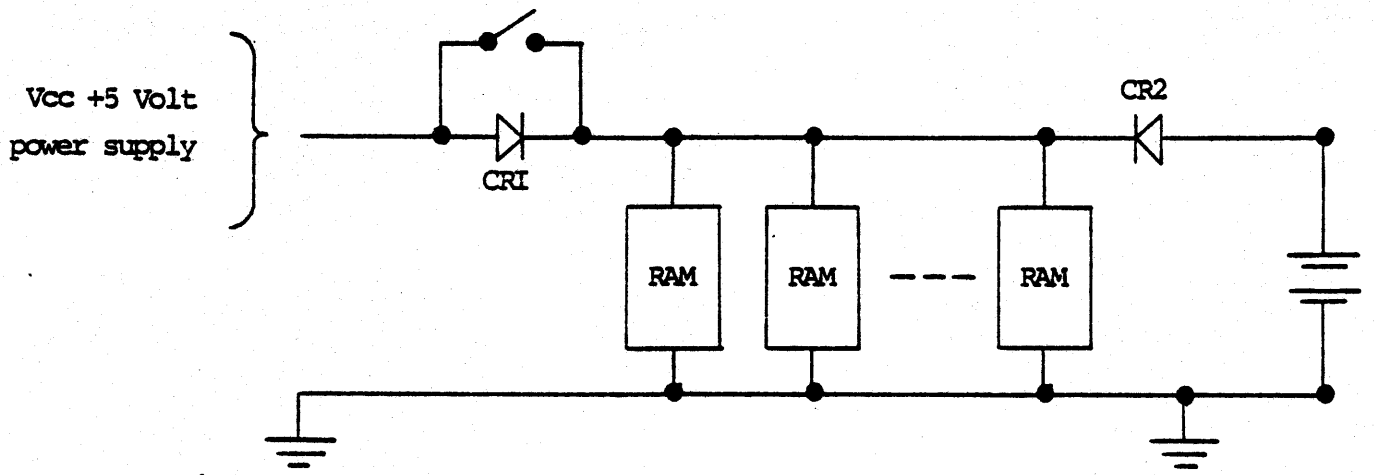
5.4.4 States T3, T4 and T5

During T3 the data bus is read by the CPU, and since this is an instruction FETCH it is loaded to the I register. 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 5-5 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 8228 passes it on to the system data bus. The CPU generates a WRITE command and the 8228 copies it to the memory devices. Once again a WAIT state is entered. After Tw the standard T3 state occurs. With a fast memory the T3 state would provide time enough for writing. The Tw state doubles that time, while reducing the processor's speed by about 25%.

5.5 BATTERY BACKUP

The reason for using CMOS memory is that it can retain its data with a single low voltage power source and extremely low current. This makes it practical to provide battery backup for the memory of the MTS with two small dry cells (AA or AAA cells will do). The MTS includes connection points, two diodes and a toggle switch as shown below. When the toggle switch is open, the diodes isolate one power source or the other. When the external power supply delivers 5 volts, the 3 volt battery is isolated by the back biased diode CR2. When the external supply is off or disconnected the battery delivers about 2.4 volts to the memory alone, the other loads being isolated by CR1. When the memory is to be used by the microprocessor, the switch must be closed to avoid a diode drop from reducing the voltage delivered to the memory. When the power is to be disconnected the switch should be opened to minimize the load on the battery. It is recommended that the RST key be depressed while the switch is being toggled, to protect data in the memory from possible transient voltage pulses. If you choose to rewire the circuit and use a rechargeable Ni-Cad battery, the reprint from Electronics magazine on the following page may be of interest.

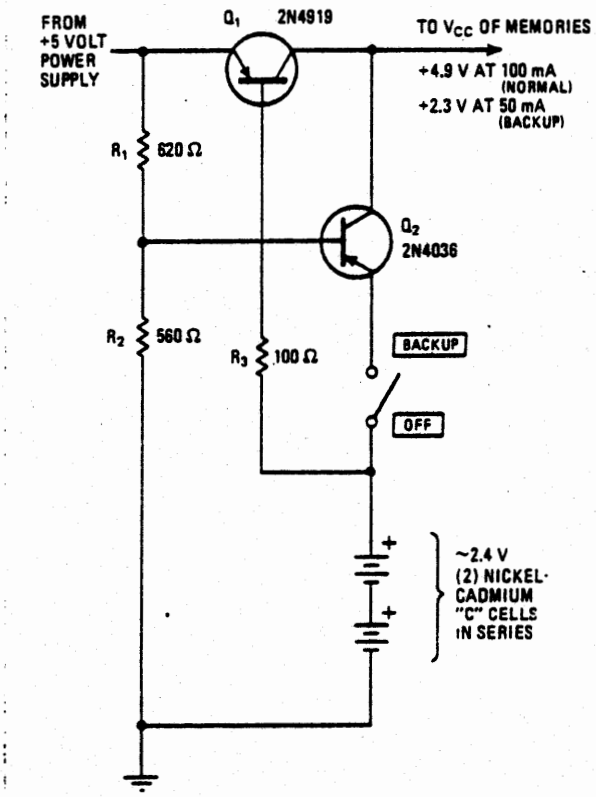


BATTERY BACKUP FOR MEMORY POWER

Figure 5 - 6

2.4-V battery backup protects microprocessor memory

by Raymond N. Bennett
Advanced Technology Laboratories Inc., Bellevue, Wash.



Memory saver. A series pair of nickel-cadmium "C" cells, each nominally rated at 1.25 volts, puts out about 2.4 volts and can deliver 2.3 volts to microprocessor memories to prevent loss of data in the event of supply failure. Transistors saturate to less than 100 mv.

Using diodes to isolate a backup battery from the power supply of microprocessor memories works fine—if the 0.7- to 1.0-volt drop across each diode can be tolerated. A more efficient circuit (see figure) substitutes saturable switching transistors that have a drop of less than 100 millivolts, which minimizes current drain and therefore extends battery life.

Moreover, the voltage of the nickel-cadmium battery supply need only be 2.4 volts, since during a power failure a saturated transistor then delivers all of 2.3 v to the memories. That is more than enough for such metal-oxide-semiconductor devices as the 2102 static random-access memory, which begins to lose data if its supply drops below about 2 v.

The circuit shown in the figure is connected between the +5-v power-supply line and the supply input of the memories. When the 5-v supply is functioning normally, transistor Q_1 is biased heavily into conduction by the difference between the supply voltage and that of the Ni-Cad batteries: 5 v − 2.4 v = 2.6 v. The voltage delivered to the memories is then about 4.9 v, since the drop across Q_1 is at most 100 millivolts. During this time, the R_1-R_2 voltage divider holds transistor Q_2 off, and the batteries receive a charge of about 20 milliamperes through R_3 and the base-emitter junction of Q_1.

When power failure occurs and the 5-v supply drops below about 3.1 v (which is 2.4 v + V_{BE}), Q_1 begins to cut off, isolating the dying 5-v supply from the load. At the same time, Q_2, biased by the R_1-R_2 voltage divider, begins to conduct, connecting the backup batteries to the load. The reverse bias on transistor Q_1 prevents the Ni-Cads from discharging through the supply circuit.

Both Q_1 and Q_2 were chosen for their very low saturation characteristics. Although their current ratings seem far in excess of what is needed, the result is that they exhibit a $V_{CE(SAT)}$ of less than 100 millivolts. But any pnp power transistors of the same general qualifications as those specified, such as the GE Powertab series, should suffice.

The standby switch has been included to permit defeating of the battery backup feature. □

Designer's casebook is a regular feature in *Electronics*. We invite readers to submit original and unpublished circuit ideas and solutions to design problems. Explain briefly but thoroughly the circuit's operating principle and purpose. We'll pay $50 for each item published.

Reprinted with permission of:
Electronics/February 3, 1977

MICROCOMPUTER TRAINING WORKBOOK

CHAPTER 6

MODULES, SUB-ROUTINES AND THE STACK

6.1 PROGRAM MODULES

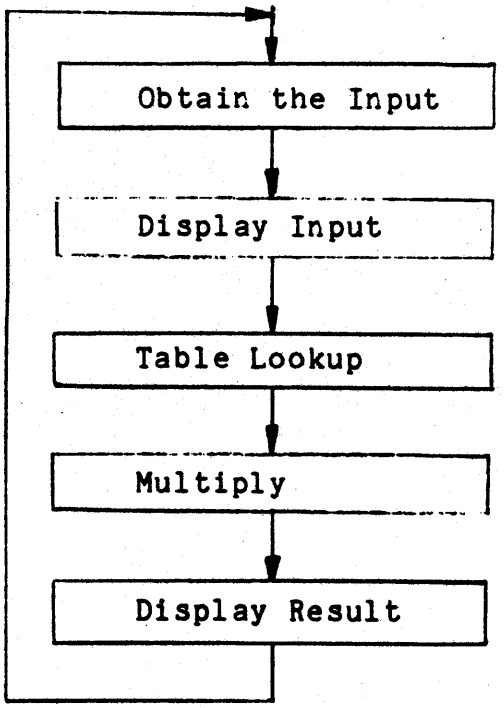
The design and hardware of a complex machine are always divided into modules, each having a limited function and a limited set of inputs and outputs. The purpose is to make each module comprehensible to the designer and to make it fit within a physically realizable structure (such as a circuit board). 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:

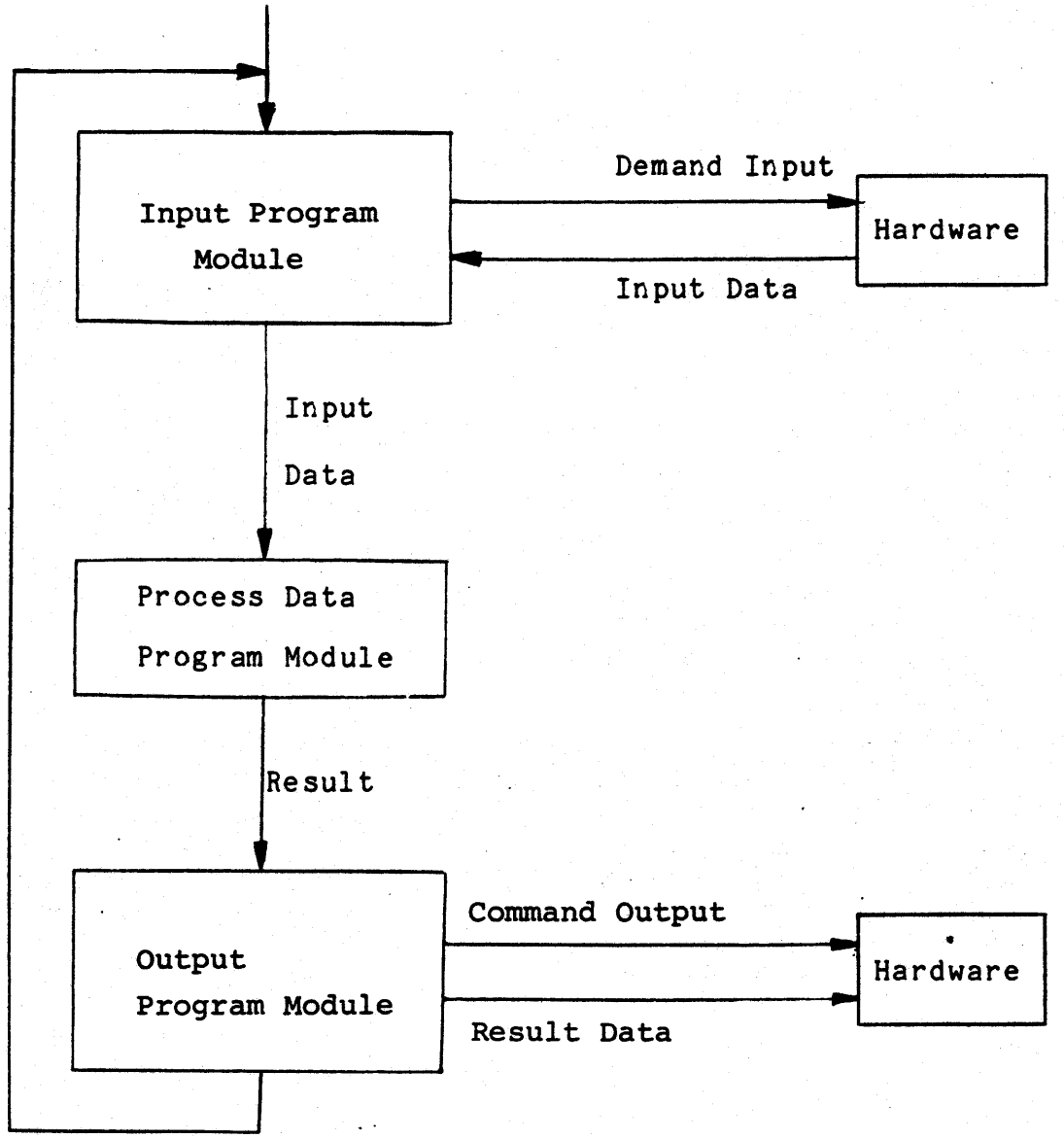
If the input and output functions were part of your program you might program them all 'in-line', with a series of instructions to accept hexadecimal keys and display them (possibly with a loop for input of two or more keys), followed by the instructions for the table lookup for a linearized value, followed 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 separate modules and develop them independently. This could be done with a subsequent integration of the several modules into an in-line program. Alternatively we could put them into separate places in the program memory and write a control program that would jump into each of them. Consider a very simple

linear procedure comprising input, process, and output.



The input may involve several data items (as for instance in the addition and multiplication problems), 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, except that we used only one entry to the monitor both for output of a result and input of new data. By the end of this chapter you will have learned ways to call upon the monitor for input and output as separate functions.

Why would we do this? In Chapter 4 we started with a multiplication that was valid only for single digit inputs. Then we improved it to handle two digit inputs. If the program had been organized as in Figure 6-1 we could have rewritten the multiplication module with no effect on the table lookup module. If we decided to reorganize the data tables the table lookup module could be revised with no effect on the multiplication module. If we decided to take sensor identification as an input, instead of processing the sensors sequentially, we could add another input module and modify the main program.

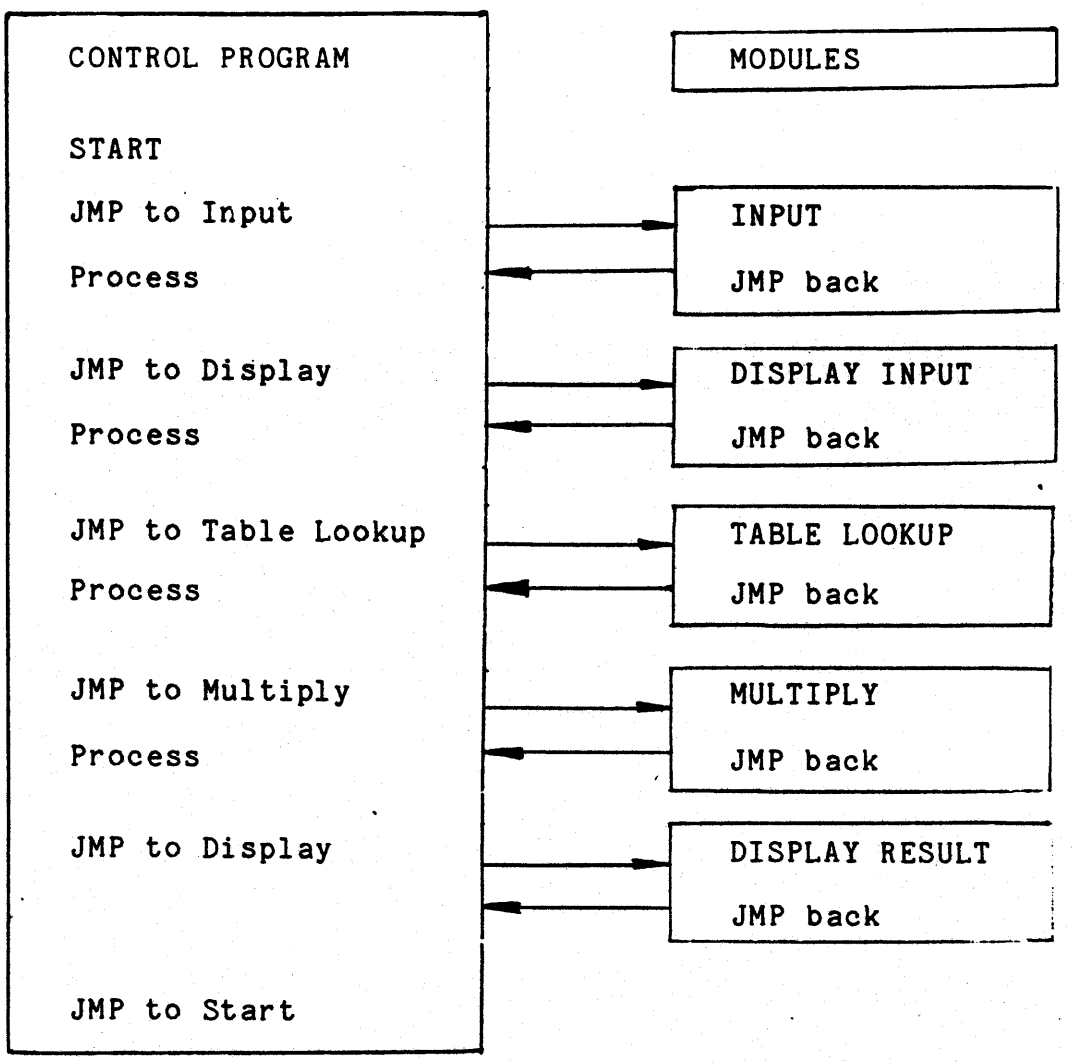

CONTROL PROGRAM
START
JMP to Input
Process
JMP to Display
Process
JMP to Table Lookup
Process
JMP to Multiply
Process
JMP to Display
JMP to Start
MODULES
INPUT
JMP back
DISPLAY INPUT
JMP back
TABLE LOOKUP
JMP back
MULTIPLY
JMP back
DISPLAY RESULT
JMP back

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 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 a specified memory address.

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.

Call:

The address of the module.

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?

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?

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?

N.B. A calling program should not have to worry about protecting the content of its registers when it calls a subroutine. The subroutine specifications should state which registers will be used to return results. All others should be returned without modification.

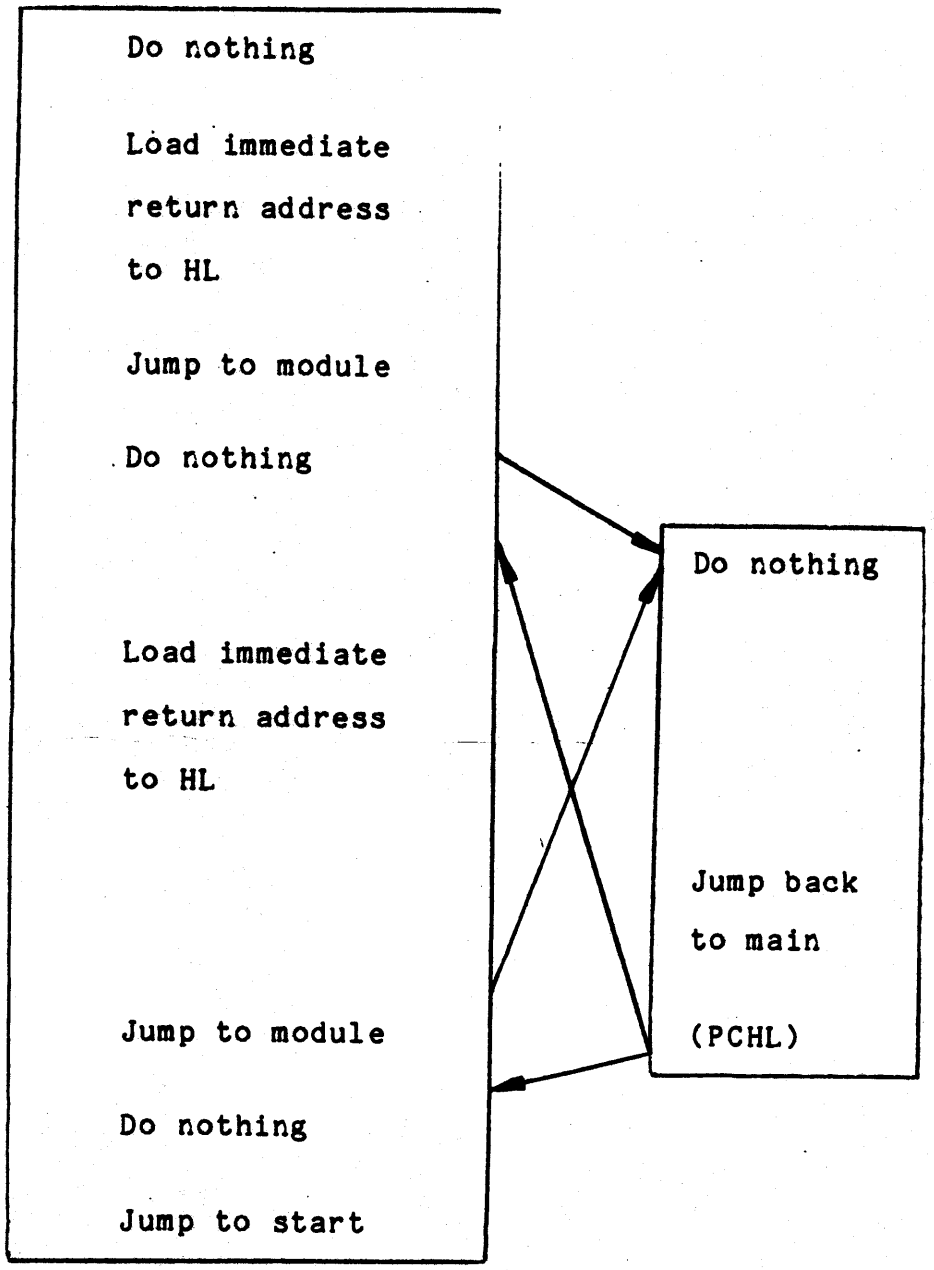
It may appear that the need to specify all of this (and often much more) makes the use of program modules a nuisance. In fact it is one of the best reasons for modular design: it forces a discipline that may otherwise be neglected. When such items are well-defined, many programming errors may be avoided.

Suppose that one module serves a function that is needed several times in the program - displaying data, for instance. In the sensor correction program we want to display the input, and also 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 demands that the specification include two return addresses and a definition of the new control variable.

A much better procedure is for the main 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

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

CODING SHEET

ADDR	CODE											
8200	00		NOP									
01	00											
02	00											
03	21		LXI		H,	8209						Address of next
04	09											NOP Instruction
05	82											
06	C3		JMP		8220							Jump to Module
07	20											
08	82											
09	00		NOP									
0A	21		LXI		H,	8210						Address of next
0B	10											NOP Instruction
0C	82											
0D	C3		JMP		8220							Jump to Module
0E	20											
0F	82											
8210	00		NOP									
11	C3		JMP		8200							Jump to start
12	00											
13	82											
14												
15												
16												
17												
18												
19												
1A												
1B												
1C												
1D												
1E												
1F												
8220	00		NOP									Module
21	E9		PCHL									Jump to address
22												in HL
23												
24												
25												
26												
27												
28												

Figure 6-3

When you have loaded the program, step though it. The program counter should show this sequence:

NOP	8200	00
NOP	8201	00
NOP	8202	00
LXI H	8203	21
JMP	8206	C3
NOP	8220	00
PCHL	8221	E9
NOP	8209	00
LXI H	820A	21
JMP	820D	C3
NOP	8220	00
PCHL	8221	E9
NOP	8210	00
JMP	8211	C3
NOP	8200	00
NOP	8201	00

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 that for us automatically. When these are used the module becomes a subroutine.

6.2 SUBROUTINES

6.2.1 Subroutine Access

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 fundamental to program architecture:

HEX CODE:	CD
MNEMONIC:	CALL
SECOND BYTE:	Low address
THIRD BYTE:	High address
MEANING:	Call 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 the Program

Revise the Do Nothing program (Figure 6-3) by replacing the following op-codes:

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.

Now use the program we have in the MTS to follow this. Step though your program to 8206, the CALL:

[STEP] 8206 CD

The monitor can display the stack pointer as a register pair:

[ADDR] [1/P] [MEM] 83D3 SP??

Now step to execute the CALL instruction:

[STEP] 8220 00

Display the stack pointer again:

ADDR 1/P MEM 83D1 SP09

The next memory location contains the high byte of the return address:

NEXT 83D2 82

Any time that you display a register pair you can see the following sequential memory pairs by pressing NEXT. In debugging programs you will more often be interested in the return address than the value of the stack pointer:

ADDR 2/T MEM 8209 ST00

Now step twice to return to the main program:

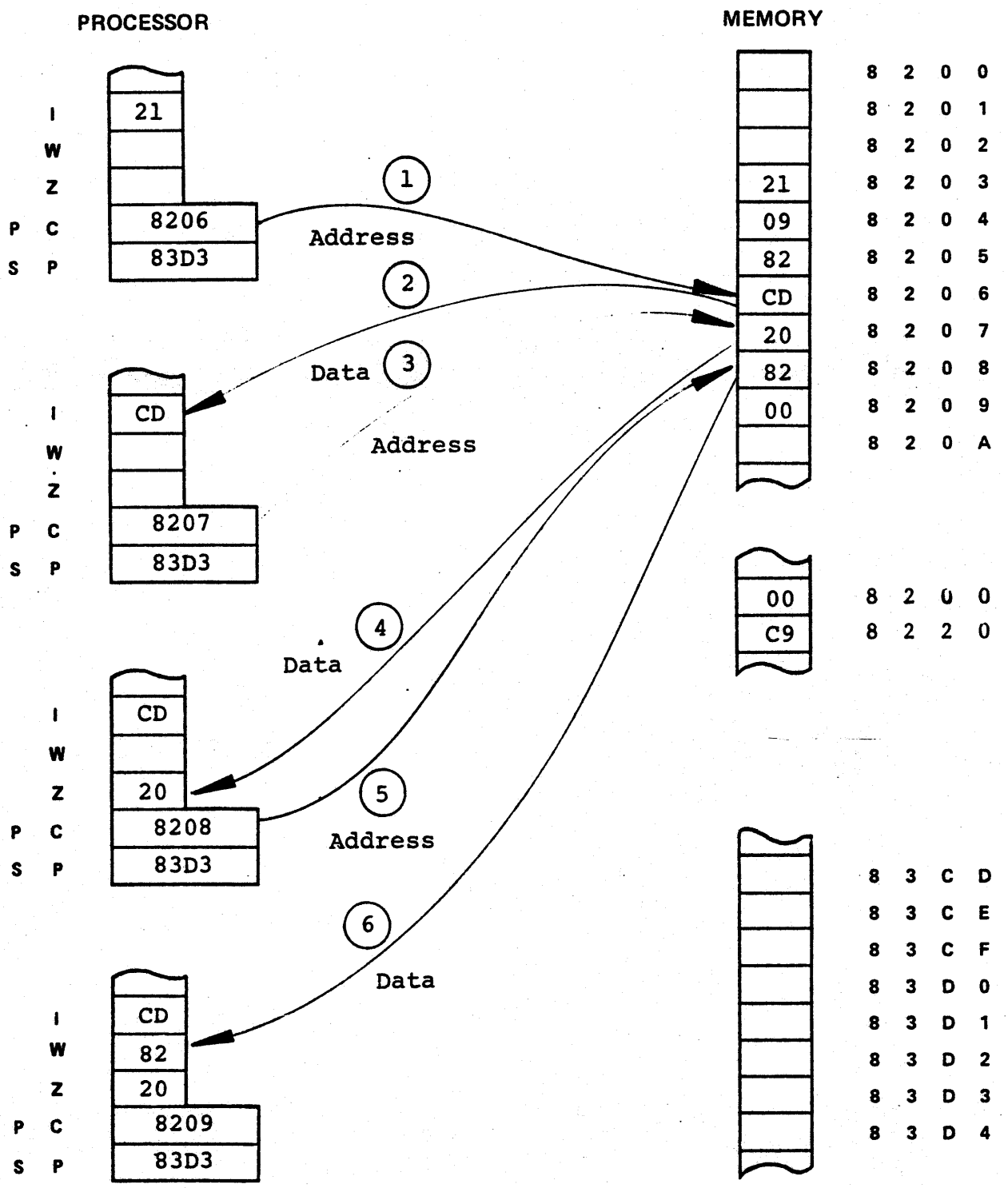
8221 C9

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 twice as the following two bytes are loaded into registers 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 (83D3) near the top of memory: this was loaded by the monitor program when power was turned on.

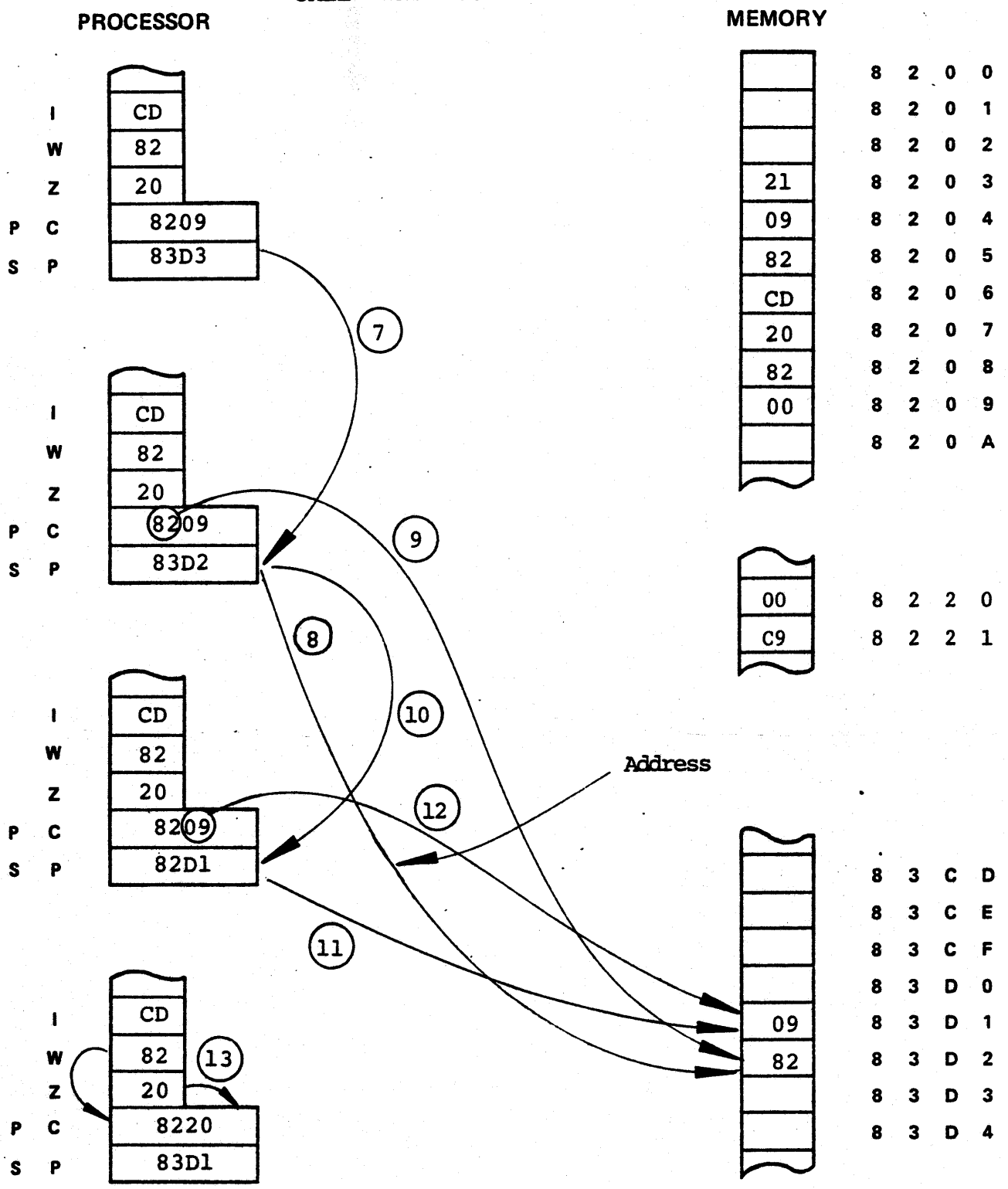


As in a jump instruction, the PC is used to address the instruction code and the two following bytes, which are loaded into I, Z and W respectively

Figure 6-4

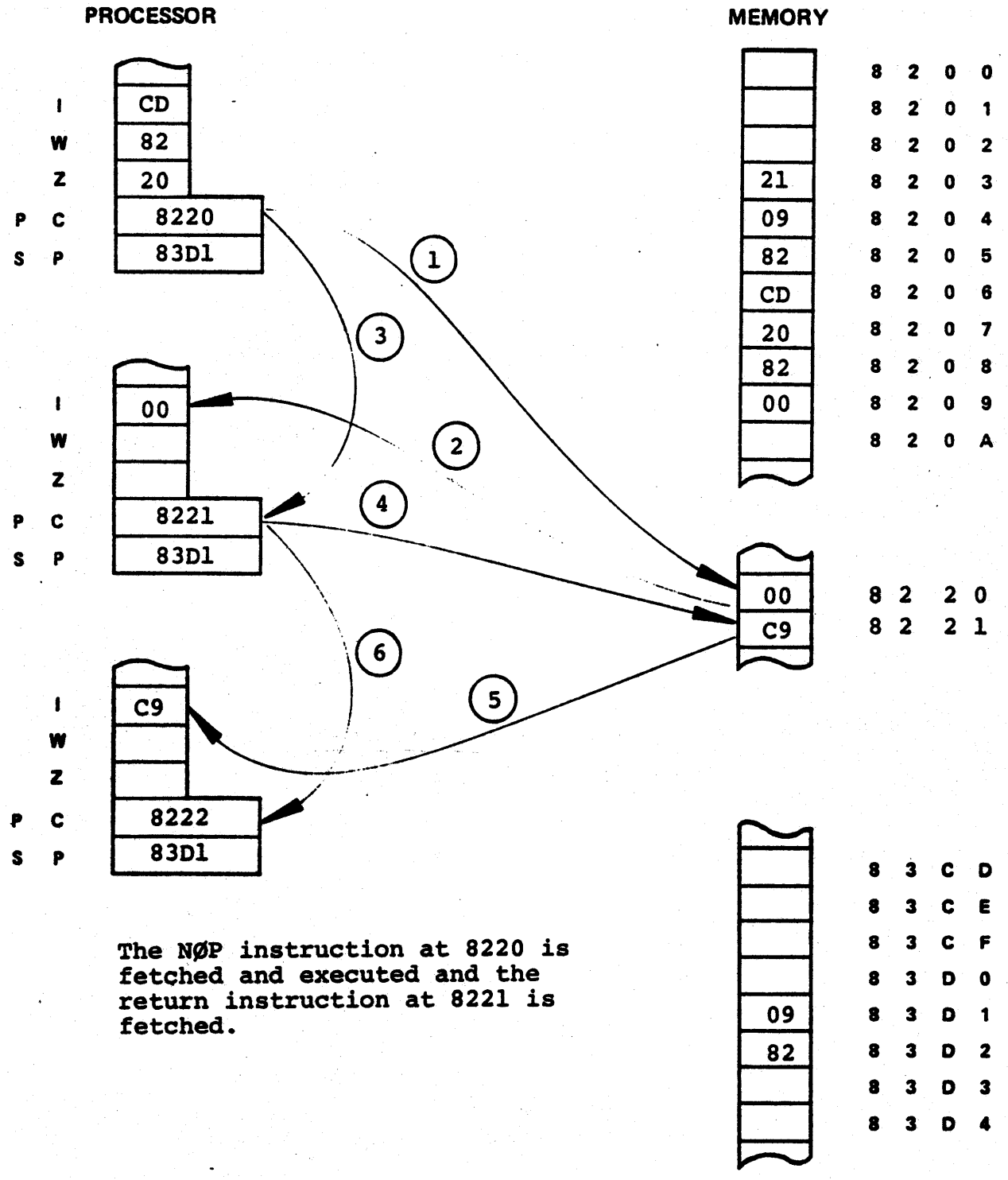
Figure 6-5 shows the stack writing operation in a CALL instruction. The content of the stack pointer is decremented and sent out on the address bus. The high byte of the program counter is sent out on the data bus to be written to the selected location in the stack area of the memory. Now the stack pointer is decremented again and the low byte of the program counter is written to the memory at the next location below the high byte. 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 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 in 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



The NØP instruction at 8220 is fetched and executed and the return instruction at 8221 is fetched.

Figure 6-6

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

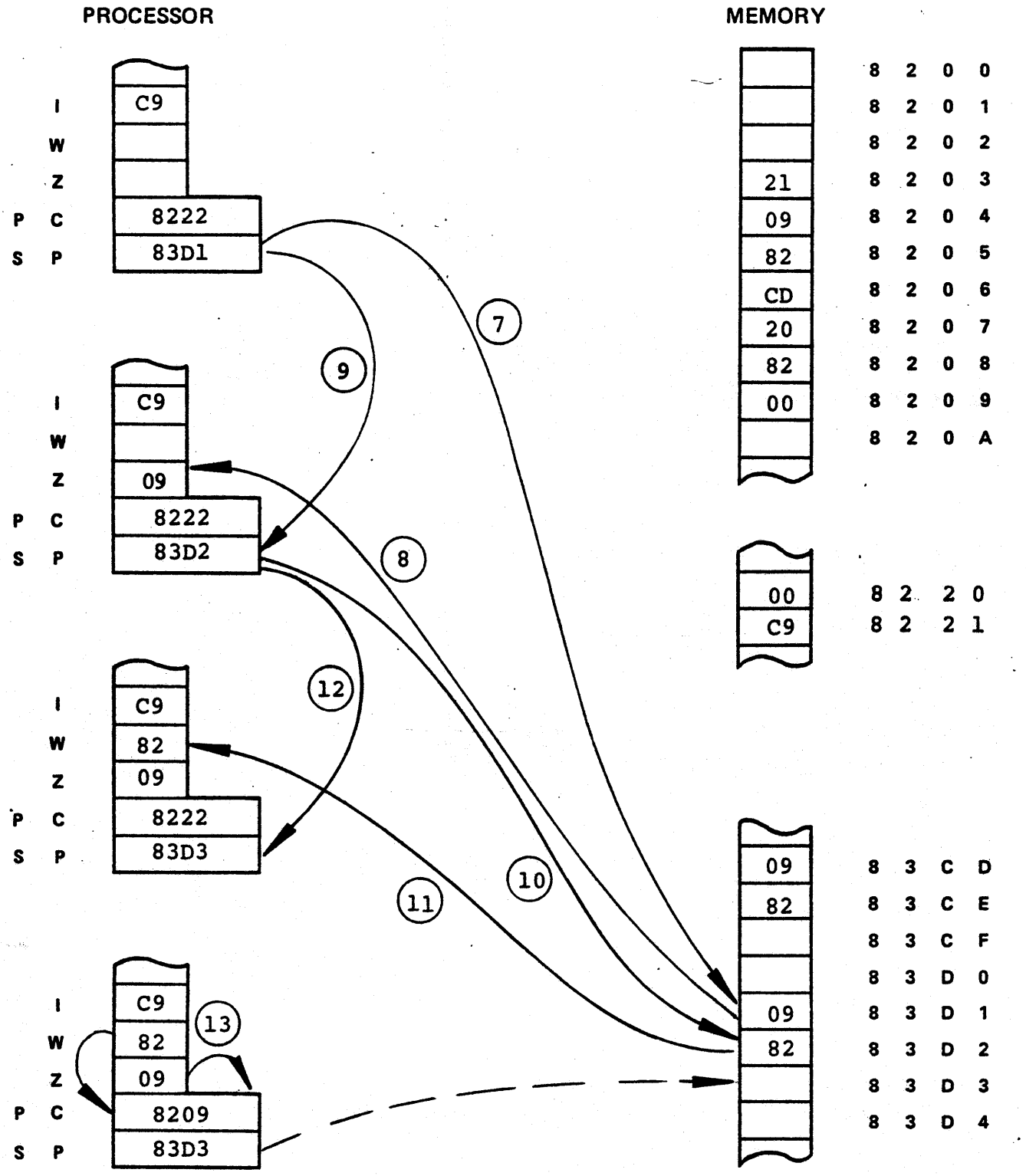
MNEMONIC: RET

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 the next two pages.

6.2.4. Execution of Return

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, and the low byte of the return address is moved into Z. The stack pointer is incremented to address the high byte, which is moved into W. The stack pointer is incremented again and the content of W and Z is moved to the program counter to accomplish the jump. 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.



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. Display the stack pointer and then up through the stack (using NEXT) when the program counter is at 821C.

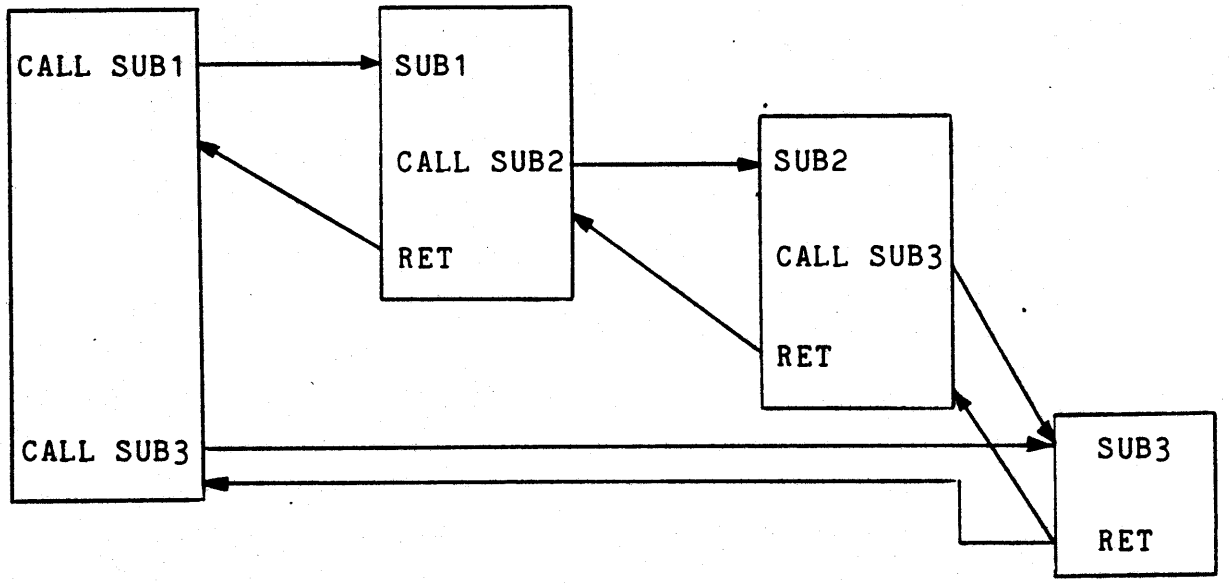

CALL SUB1
CALL SUB3
SUB1
CALL SUB2
RET
SUB2
CALL SUB3
RET
SUB3
RET

Figure 6-8

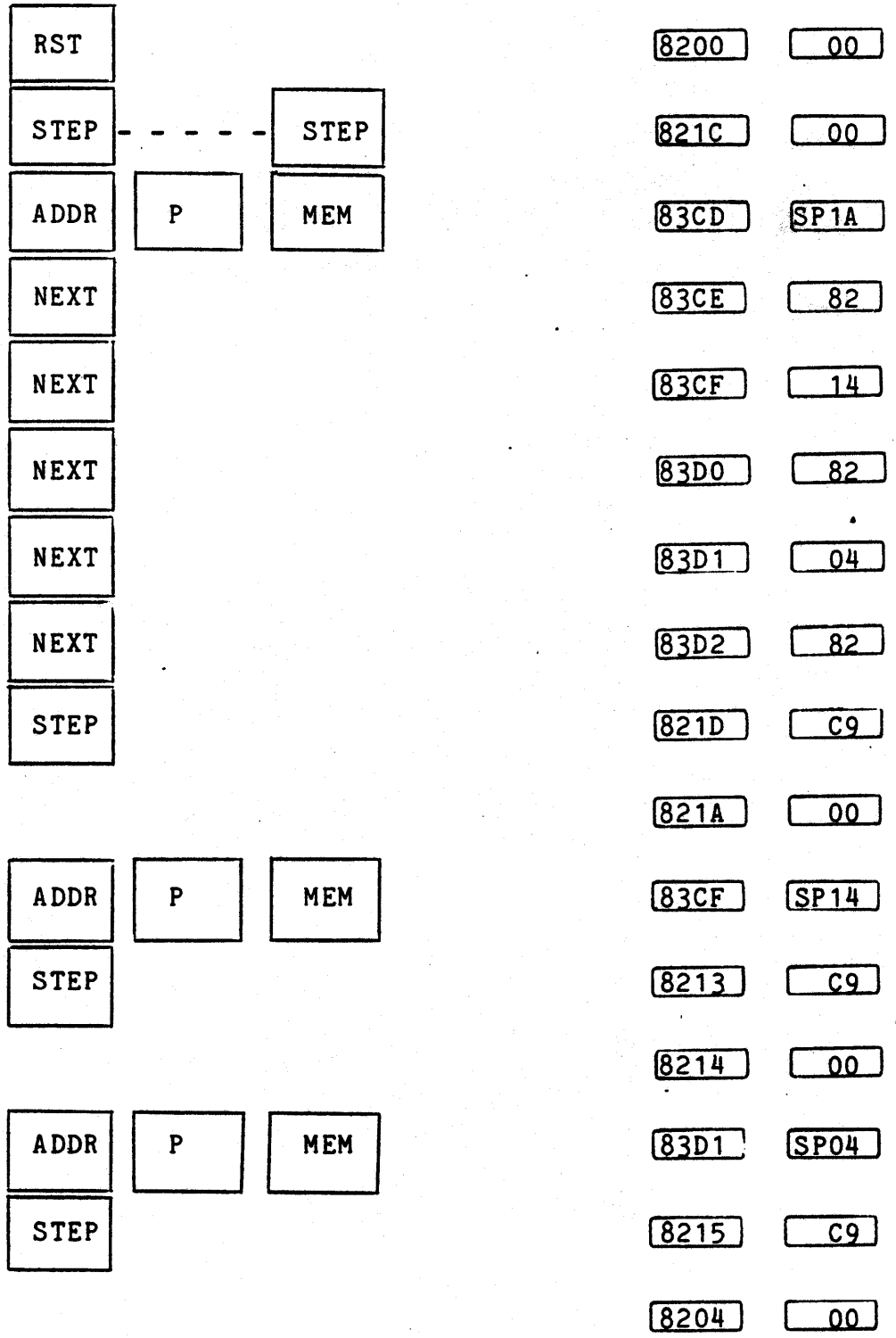
INTEL DO NOTHING PROGRAMS

INTEGRATED COMPUTER SYSTEMS MICROCOMPUTER TRAINING SYSTEM CODING SHEET

ADDR	CODE													
8200	CC		NOP											MAIN
01	CD		CALL					SUB1						
02	10													
03	82													
04	00		NOP											
05	CD		CALL					SUB3						
06	1C													
07	82													
08	00		NOP											
09	C3		JMP				8200							
0A	00													
0B	82													✓
0C														
0D														
0E														
0F														
8210	00		NOP											SUB1
11	CD		CALL					SUB2						
12	16													
13	82													
14	00		NOP											
15	C9		RET											↓
16	00		NOP											SUB2
17	CD		CALL					SUB3						
18	1C													
19	82													
1A	00		NOP											
1B	C9		RET											↓
1C	00		NOP											SUB3
1D	C9		RET											↓
1E														
1F														
8220														
21														
22														
23														
24														
25														
26														
27														
28														

Figure 6-9

Trace the Program flow through the dummy subroutines of Figure 6-9:



8205	CD
821C	00
83D1	SP08
821D	C9
8208	00

ADDR P MEM

STEP

6.3 SUBROUTINE SPECIFICATION

Figure 6-10 shows a flow chart for the sensor correction problem written as a series of subroutines and a main program. We will develop these modules separately and then integrate the complete program.

6.3.1 Subroutine Development

The chief reason for writing modules as subroutines is to permit the same module to be called from various program locations. 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, 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.

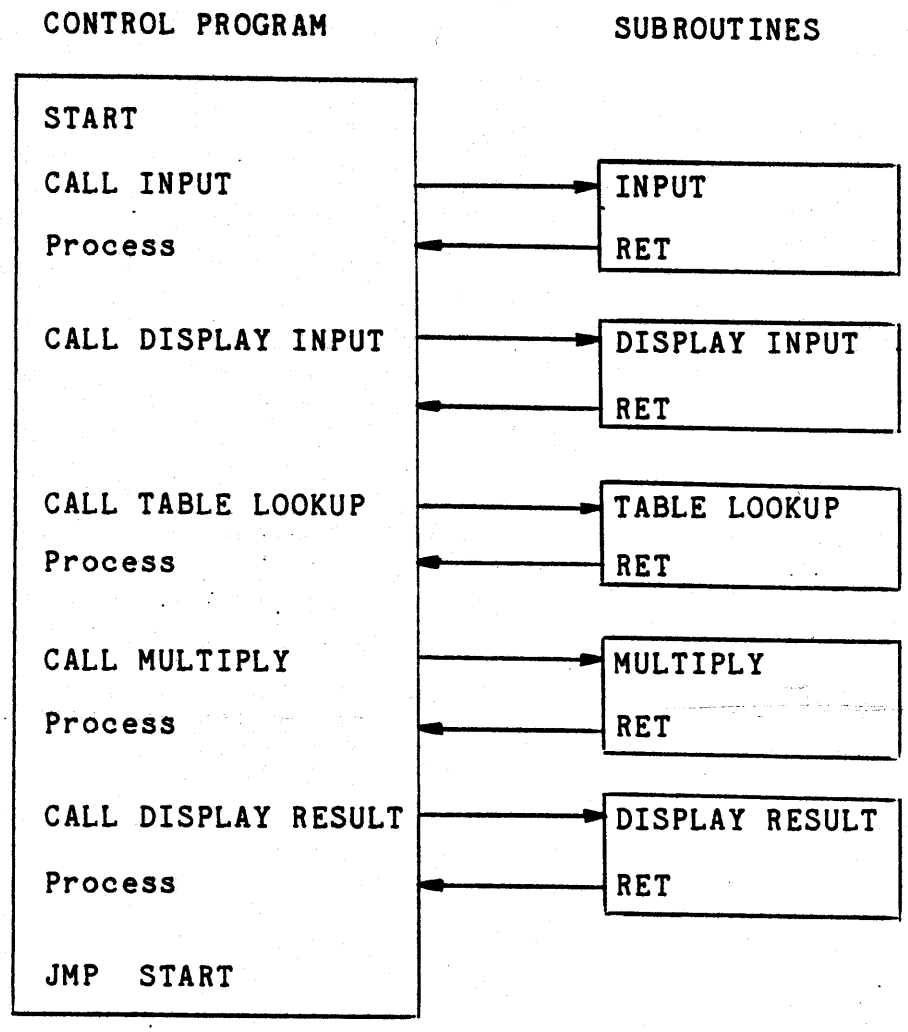

CONTROL PROGRAM
SUBROUTINES
START
CALL INPUT
Process
CALL DISPLAY INPUT
CALL TABLE LOOKUP
Process
CALL MULTIPLY
Process
CALL DISPLAY RESULT
Process
JMP START
INPUT
RET
DISPLAY INPUT
RET
TABLE LOOKUP
RET
MULTIPLY
RET
DISPLAY RESULT
RET

Figure 6-10

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 and test the modules

g) 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.

Input:

Accept two keys as a one byte sensor input.

Display Input:

Display the input in the third and fourth locations of the display.

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 in the four right hand digits of the display.

6.3.2 Two Monitor Subroutines, GETKY and DBY2

Section 6.10 of this chapter presents the specifications for a number of monitor subroutines which are available to the user. We will use two of the subroutines described there: GETKY (6.10.2) and DBY2 (6.10.6). Read the specifications carefully. These routines should be tested, both to be sure that they fit the needs of the sensor correction program and to gain familiarity with them. The test is simple:

```
8200    CD      CALL GETKY - Get a key
  01    3D
  02    02
  03    11      LXI  D,83FB - Address for display
  04    FB      (Why?  Change it and see what happens)
  05    83
  06    CD      CALL  DBY2
  07    98
  08    02
  09    C3      JMP  8200
  0A    00
  0B    82
```

You cannot step through this program because the monitor will not know that a key you press is intended for your programmed call to GETKY. It supposes you are giving commands or data to the monitor program. After loading the program, operate it with RUN, but with the STEP/AUTO toggle switch still in STEP position. Then try it in AUTO (return to STEP position when done).

The output will appear in the third and fourth locations of your display. You can now see the hex value that is assigned to the command keys. Note that RST is not a key that can be detected by GETKY; it

serves a hardware function much like a power on/off switch.

6.4 MONITOR BREAKPOINTS

It is often desirable to trace program flow without the tedious task of stepping through lengthy loops, or through previously tested and proven program modules. Breakpoints permit you to use the RUN key (but only with the toggle switch in STEP position) to cause your program to run without apparent interference until you reach a specific instruction. Breakpoints also permit you to call GETKY and other monitor input subroutines, but still step through you own program instructions.

6.4.1 Using Breakpoints

With the above program loaded, do this:

ADDR	8206	8206	CD
BRK		8206	BP.
ADDR		8200	CD
RUN			

Now your program is running, with control in the GETKY subroutine, waiting for a key. Press a hex key:

3	8206	CD

GETKY has accepted your input and returned to 8203. The LXI D instruction was executed, and the program counter reached 8206. Since

the toggle switch is at STEP, the monitor is constantly monitoring your program execution (hence its name) and it finds that you have entered a breakpoint at 8206. It now behaves as though you had stepped to this point. Display register A and you will see the key you entered:

REG	A	8206	A-03

Now press RUN again and your program calls DBY2 to display the input.

RUN	03	

Press another hex key:

5	8206	A-05

Just as if you had used STEP, the monitor retains your request to display A.

You can enter up to eight breakpoints:

ADDR	8203	BRK	8203	BP.

and you can look at the list of breakpoints:

NEXT	8206	BP00
NEXT	8203	BP00

Now program execution will stop at each of these points.

RUN	05	

Your program has called DBY2 and GETKY, so press another key:

9	8203	A-09

There are two ways of clearing breakpoints.

BRK	8203	BP00
CLR	8206	BP00

The breakpoint at 8203 has been cleared; the one at 8206 remains.

A reset clears all breakpoints:

RST	BRK	0000	BP00

Now enter a break at 8206 again.

ADDR	8206	BRK	8206	BP

The right hand digits are blank; you can enter a number here:

3	8206	BP03

Now program execution will stop after the instruction at 8206 has been executed three times and is about to be executed again:

RUN		
1		01
2		02
3		03
4	8206	CD

BRK	8206	BP00

The count you entered has been decremented to zero; now program execution will stop here every time, before executing the instruction.

RUN		04
5	8206	CD

You can easily restore a count by pressing BRK and the count you want.

BRK	8206	BP00
3	8206	BP03

Each breakpoint has a separate count so you can manipulate them to stop at one location each time it is reached, at another location after 5 repetitions, etc.

Remember that if you are using breakpoints you must avoid using RST to go back to starting address 8200; use ADDR 8200 instead. RST clears all breakpoints.

You will want to use breakpoints if you have trouble with your development of the sensor correction program because of the use of GETKY for input. We now proceed to develop the input subroutine module. Practice the use of breakpoints here even if you have no trouble.

6.5 SENSOR PROGRAM SUBROUTINES

6.5.1 The INPUT Subroutine

Since GETKY only gets one key - one hex digit - we must call it twice. But we want the two keys combined into one byte, not treated as two separate bytes.

Let us review the relationship between two hexadecimal nibbles (a nibble is half a byte, or one four bit hex character) and a byte.

$$\text{Value of Nibble} = B_3 \times 2^3 + B_2 \times 2^2 + B_1 \times 2^1 + B_0 \times 2^0$$

$$\begin{aligned}\text{Value of Byte} &= B_7 \times 2^7 + B_6 \times 2^6 + B_5 \times 2^5 + B_4 \times 2^4 \\ &+ B_3 \times 2^3 + B_2 \times 2^2 + B_1 \times 2^1 + B_0 \times 2^0\end{aligned}$$

$$\begin{aligned}\text{Value of Byte} &= (B_3 \times 2^3 + B_2 \times 2^2 + B_1 \times 2^1 + B_0 \times 2^0) \times 2^4 \\ &+ (B_3 \times 2^3 + B_2 \times 2^2 + B_1 \times 2^1 + B_0 \times 2^0)\end{aligned}$$

Therefore we can say:

$$\text{Value of Byte} = (\text{Nibble 1}) \times 2^4 + \text{Nibble 0}$$

By convention nibble 1 is the first key entered and nibble 0 is the second. What we must do is read two keys; multiply the first by $2^4 = 16_{10} = 10_{16}$ and add the second.

The procedure for input will be this:

Call GETKY for nibble 1

Multiply by 10_{16}

Save result in a register

Call GETKY for nibble 0

Add it to previous result.

Since we will have a multiplication subroutine we can call it for the multiplication. What will that require? If we are able to specify MULT so that no extra moving of data between registers is necessary, the procedure will be:

```
CALL  GETKY
  (C) <- Key (done by GETKY)
MVI   E,10    (multiplier)
CALL  MULT
  (L) <- 10   X  Key (done by MULT)
          16
  (H) <- 0    (since the product cannot exceed F0)
```

The result is in L, which is preserved during subsequent calls to GETKY. We should recognize, though, that multiplication by 10 in a binary computer is just about as easy as manual multiplication by 10 . Remember that the set of ADD instructions includes:

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

or

A) <- 2 x (A)

If we do this repeatedly, we get the following results:

```
ADD  A      2 x (A)
ADD  A      4 x (A)
ADD  A      8 x (A)
ADD  A     16 x (A)
```

Now MOV L,A will place the result in L just as MULT would have done. This procedure takes the same program space (five bytes) as:

```
MVI  E,10   (two bytes)
CALL MULT   (three bytes)
```

Therefore we will use the ADD A procedure instead of a call to MULT.

Now we can specify the INPUT Subroutine:

Function

Accept two keys and form a one byte sensor input value, using the first key as the high order digit.

Call

```
CD          CALL INPUT
F0
82
```

Inputs

From keyboard

Outputs

Sensor input in register A

Extent

82F0 through 82FD
Calls GETKY

Registers Used

A, B, C, D, L
Registers E and H are preserved.

Constraints

GETKY retains control until a key has been pressed and released, and for 20 milliseconds thereafter. The delay is exaggerated in STEP mode.

Try writing an INPUT subroutine on your own, and test it with a CALL to DBY2. Then look at the coding presented in Figure 6-11.

INPUT SUBROUTINE

ADDR	CODE														
8 2F 0	C	D		C	A	L	L		G	E	T	K	Y		Get high nibble
1	3	D													as first key
2	0	2													
3	8	7		A	D	D		A							Multiply by 1016
4	8	7		A	D	D		A							
5	8	7		A	D	D		A							
6	8	7		A	D	D		A							
7	6	F		M	O	V		L	,	A					Save high nibble
8	C	D		C	A	L	L		G	E	T	K	Y		Get low nibble
9	3	D													as second key
A	0	2													
B	8	5		A	D	D		L							Add to high x 1016
C	4	F		M	O	V		C	,	A					(C) ← Byte
D	C	9		R	E	T									
E															
F			TE	S	T		F	O	R		I	N	P	U	T
8 20 0	C	D		C	A	L	L		I	N	P	U	T		
1	F	0													
2	8	2													
3	1	1		L	X	E		D	,	8	3	F	B		Address digit 4
4	F	B													
5	8	3													
6	C	D		C	A	L	L		D	B	Y	2			Display input
7	9	8													
8	0	2													
9	C	3		J	M	P		8	2	0	0				
A	0	0													
B	8	2													
C															
D															
E															
F															
8 0															
1															
2															
3															
4															
5															
6															
7															
8															

Figure 6-11

6.5.2 Display Result Subroutine - DRES

We have programmed a double precision multiplication in Chapter 4. Its result appears in registers (H,L) and since the specification for DBY2 states that those registers are preserved, it seems appropriate to specify that the double precision result to be displayed by DRES will be placed in (H,L) at input.

Function

Display a four digit number in the right hand four digits of the display, using DBY2 as a subroutine.

Call

```
CD          CALL  DRES
E0
82
```

Inputs

Four digit number in H,L

Outputs

Seven segment codes for four digits are stored at 83FC-83FF

Extent

82E0 through 82EB
Calls DBY2

Registers

A, B, D, E are used
Registers H and L are preserved.

Write a DRES subroutine (check the specs again) and test it. Then look at Fig 6-12.

DISPLAY RESULT SUBR DRES

ADDR		CODE															
8240	0	11			LXI			D,	83	FF							Address right
	1	FF															hand digit
	2	83															
	3	7D			MOV			A,	L								Display low byte
	4	CD			CALL			DBYZ									
	5	98															
	6	02															
	7	7C			MOV			A,	H								Display high byte
	8	CD			CALL			DBYZ									
	9	98															
	A	02															
	B	C9			RET												
	C																
	D																
	E																
	F			TEST			USING				INPUT & DRES						
8200	0	CD			CALL			INPUT									Get high byte
	1	FO															
	2	82															
	3	67			MOV			H,	A								
	4	CD			CALL			INPUT									Get low byte
	5	FO															
	6	82															
	7	6F			MOV			L,	A								
	8	CD			CALL			DRES									Display
	9	EO															
	A	82															
	B	C3			JMP			8200									
	C	00															
	D	82															
	E																
	F																
8	0																
	1																
	2																
	3																
	4																
	5																
	6																
	7																
	8																

Figure 6-12

6.5.3 Table Lookup Subroutine (TABLU)

The Table Lookup Subroutine Specification is:

Function

Given a sensor number and a one byte input, obtain a scaling factor and a linear point for the sensor. If the input is in the non-linear region, obtain a linearized value.

Call

```
CD              CALL  TABLU
B0
82
```

Inputs

Register C	Input value
Pair H,L ((H),(L))	Memory Address Sensor Number

Outputs

Register E	Scaling Factor
Register C	Linearized Input

Extent

82B0 through 82C0

Registers Used

A, B, C, E, H, L
Register D is preserved

Constraints

A table of scaling factors, linear points and corrected values must be in memory within locations 8301 to 83BF. The format is to be in accordance with figure 4-17.

Look at the code in the final program of Chapter 4, extract the table look-up portion, and write it in the form of a subroutine, originating at 82B0. A solution is shown in Figure 6-13. We will test this with the calling program after integration.

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

ADDR	CODE		Instruction	Operand	Comment
8 2 B 0	6	E	MOV	L, M	Sensor number
1	2	6	MVI	H, 83	Address list of
2	8	3			addresses
3	6	E	MOV	L, M	Address data table
4	5	E	MOV	E, M	Scaling factor
5	2	3	INX	H	Address linear point
6	7	9	MOV	A, C	(A) ← Sensor Input
7	B	E	CMP	M	Compare linear point
8	D	2	JNC	82C0	Jump if linear
9	C	0			
A	8	2			
B	0	6	MVI	B, 00	Clear B
C	0	0			
D	2	3	INX	H	Address corrected
E	0	9	DAD	B	value in table
F	4	E	MOV	C, M	
8 2 C 0	C	9	RET		
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 6-13

6.5.4 Double Precision Multiply Subroutine (MULT)

Function

Multiply two input bytes and return the double precision product.

Call

```
CD          CALL MULT
D0
82
```

Inputs

```
Register (E)    Multiplier
Register (C)    Multiplicand
```

Outputs

```
Registers (H,L)  Product
```

Extent

```
82D0  through  82DA
```

Registers Used

```
B, C, E, H, L
Registers A and D are preserved
```

You have used this routine often enough to know it by heart. But what happens if one of the inputs is zero? Write a MULT subroutine which checks for this. Then compare it with Figure 6-14.

MULTIPLICATION SUBROUTINE MULT

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

ADDR		CODE														
8 2D	0	21		L	X	I		H	,	0	0	0	0		Clear Product
	1	00													
	2	00													
	3	AF		X	R	A		A							Clear A
	4	47		M	O	V		B	,	A					Clear B
	5	BB		C	M	P		E							Test for zero
	6	CA		J	Z			8	2	D	E				multiplier and
	7	DE													exit if zero
	8	82													
	9	09		D	A	D		B							Add multiplicand
	A	1D		D	C	R		E							Decrement multiplier
	B	C2		J	N	Z		8	2	D	9				Repeat until
	C	D9													multiplier zero
	D	82													
	E	C9		R	E	T									
	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 6-14

6.5.5 The Integrated Program

We now have all of the modules needed to integrate the final sensor correction program. We did not specify a routine for displaying the input, as DBY2 will do that and is fully defined. Draw a flow chart and write the main calling program. Make sure before each call that you are passing the proper arguments to the subroutine, i.e. that all required values are in the proper registers. Then compare your work with Figures 6-15 and 6-16.

SENSOR CORRECTION - MAIN
FLOW DIAGRAM

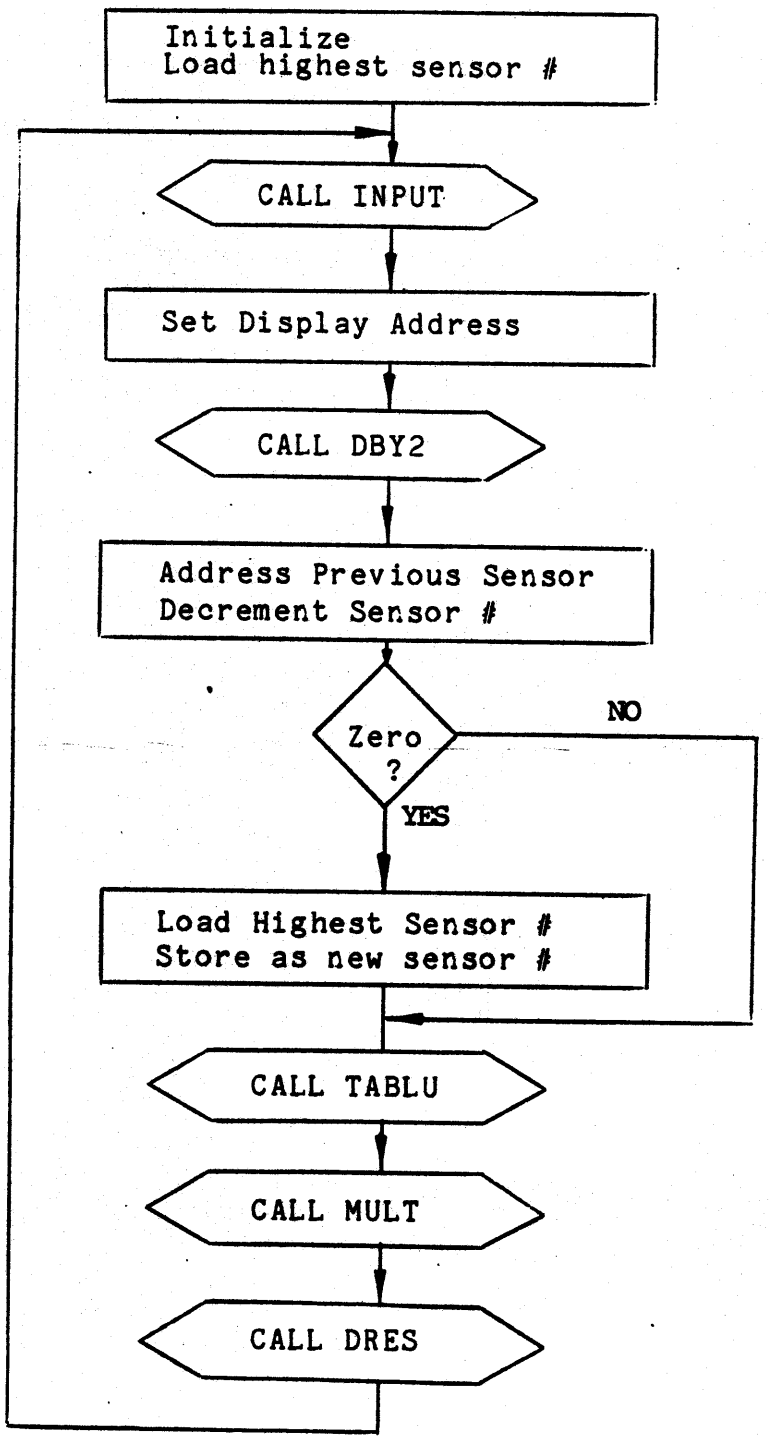


Figure 6-15

ADDR	CODE				Comments
8200	00	NOP			
01	00				
02	00				
03	3A	LDA	8300		Copy highest
04	00				sensor number
05	83				
06	32	STA	8380		to current
07	80				sensor number
08	83				
09	CD	CALL	INPUT		Get input data
0A	F0				
0B	82				
0C	11	LXI	D, 83FB		Address digit 4
0D	FB				
0E	83				
0F	CD	CALL	DBY2		Display input
8210	98				
11	02				
12	21	LXI	H, 8380		Address current
13	80				sensor number
14	83				
15	35	DCR	M		Decrement
16	C2	JNZ	821D		Jump to process
17	1D				data unless
18	82				sensor number 0
19	3A	LDA	8300		Load highest
1A	00				sensor number
1B	83				and store as
1C	77	MOV	M, A		current sensor
1D	CD	CALL	TABLU		Get corrected
1E	B0				input and
1F	82				scaling factor
8220	CD	CALL	MULT		Multiply corrected
21	D0				input by scaling
22	82				factor
23	CD	CALL	DRES		Display result
24	E0				
25	82				
26	C3	JMP	8209		Back to input
27	09				
28	82				

Figure 6-16

Enter the code, verify it, then verify that all your subroutines are still loaded and intact. As we have already STEPed through much of the code on these pages, you understand the dynamics of all of the instructions. If your program fails, debug it using breakpoints, or recheck your memory locations.

When you press RUN, the display will go blank. Enter two numbers. The entered numbers and the results will appear (to the eye) simultaneously, and will remain until the next two numbers are pressed. Remember that you are toggling back and forth between sensor #1 and sensor #2. If you lose track of which is which, restart. Place 0C in the scaling factor for sensor #1 (8208), and 0B for the factor in sensor #2 (8316). Then try some of the following inputs:

Input	Sensor #1	Sensor #2
01	24	16
05	54	42
0B	84	79
12	D8	C6
4B	384	339
D6	A08	932
FF	AF5	BF4

Try various calibration factors. You can select values which will allow you to construct the entire hexadecimal multiplication table. Do this for an exercise. (Hint: change the linear point, and use different calibration factors).

6.5.6 Alternate Subroutine Entries

It is often very useful to design a subroutine to permit several entry points. As an example, consider the multiplication subroutine. Suppose we wish to generate the function:

$$z = ax + by$$

This can be done by the following procedure:

```
(C) <- a
(E) <- x
CALL  82D0
(C) <- b
(E) <- y
CALL  82D3
```

The second call enters the subroutine beyond the instruction that clears the product, so the partial product from the first multiplication is preserved, and the second product is added to it.

We have been using the monitor subroutine DBY2, and loading an address for the digits to be displayed. In fact the subroutine has two preceding entry points, as you saw from the specifications:

```
0294      7E      MOV  A,M      (DMEM)
0295      11      LXI  D, 83FF (DBYTE)
          FF
          83
0298      E5        (save H,L)  (DBY2)

          :
          :
02A9      C9      RET
```

Entering at DMEM (by CALL 0294) will display the content of the memory location addressed by (HL), in the right hand two digits of the display. Entering at DBYTE (by CALL 0295) will display the content of register A in the right hand two digits. Entering at DBY2 (by CALL 0298) will display the content of register A in the digits addressed by (DE).

6.5.7 Conditional Call and Return

We have been using five jump instructions: JMP, JNZ, JZ, JNC, JC. Four more will be introduced later. Since CALL and RET instructions are special jumps, they also have corresponding conditional versions:

CD	CALL	Call (unconditional)
C4	CNZ	Call if not zero
CC	CZ	Call if zero
D4	CNC	Call if not carry
DC	CC	Call if carry set
C9	RET	Return (unconditional)
C0	RNZ	Return if not zero
C8	RZ	Return if zero
D0	RNC	Return not carry
D8	RC	Return if carry set

The conditional calls are infrequently used. Conditional returns more often have some value. Both in TABLU and MULT given as solutions to the preceding exercise, there are conditional jumps to the return instruction which could be replaced by the corresponding conditional returns. A version of the multiplication subroutine using two conditional returns is shown in Figure 6-17. At 82D5 the program tests register E; the conditional return RZ at 82D6 returns if E is zero. Then the double precision ADD is performed at 82D7, and if a carry occurs the conditional return RC at 82D8 terminates the multiplication. The calling program can also test the carry flag at return, either by a conditional jump or a conditional call to an error processing subroutine. An interesting feature is that the RZ instruction serves double duty here; it returns either if the multiplier is zero initially, or when it has been decremented to zero.

MULTIPLICATION SUBROUTINE MULT

ADDR		CODE														
82D	0	21			LXI			H,	0	0	0	0			Clear Product
	1	00													
	2	00													
	3	AF			XRA			A							Clear A
	4	47			MOV			B,	A						Clear B
	5	BB			CMP			E							If multiplier zero
82D	6	C8			RZ										return with result
	7	09			DAD			B							Add multiplicand
	8	D8			RC										Return on overflow
	9	1D			DCR			E							Decrement multiplier
	A	C3			JMP			82D6							Loop to return
	B	D6													when multiplier = 0
	C	82													or continue
	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 6-17

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 down from the top, the spring pops the next one up. The microprocessor has PUSH and POP instructions to place data into the stack, and remove 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,C), (D,E), or (H,L).
E5    PUSH  H

C1    POP   B          Pop data into register pair (B,C), (D,E)
D1    POP   D          or (H,L) from the stack.
E1    POP   H
```

Suppose that a program needs to call MULT, then DRES, but also needs to retain the content of (H,L). Since each of the registers is used in at least one of these subroutines, we must save the address 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 MULT and POP H after return from DRES will save and recover the data. The content of any of the three register pairs can be saved in this manner.

The program listed in Figure 6-18 uses the data table, MULT and DRES from the preceeding exercise. Register pair H,L addresses the table of linearized values for sensor number 1 and B,C addresses the table for sensor number 2. These addresses are saved while MULT, DRES and GETKY are called, then restored after the call (GETKY is used merely to signal that you are ready for the next data pair). Load the program and check for the following results; then we will trace the stack:

Table Entries (hex)		Result
03 x 02	=	0006
04 x 04	=	0010
05 x 04	=	0014
06 x 05	=	001E
07 x 06	=	002A
08 x 07	=	0038
09 x 07	=	003F

MULTIPLICATION OF TWO TABLES

ADDR	CODE													
8200	00		NOP											
01	00													
02	00													
03	21		LXI		H	,	8	3	0	A				Address table
04	0A													for sensor 1
05	83													
06	01		LXI		B	,	8	3	1	8				Address table
07	18													for sensor 2
08	83													
09	23		INX		H									Next table entry
0A	5E		MOV		E	,	M							to register E
0B	03		INX		B									Next table entry
0C	0A		LDAX		B									to register A
0D	B7		ORA		A									Test for end of table
0E	CA		JZ		8200									At end go to
0F	00													start over
8210	82													
11	E5		PUSH		H									Save addresses
12	C5		PUSH		B									
13	4F		MOV		C	,	A							Multiplicand to C
14	CD		CALL		MULT									Multiply
15	D0													
16	82													
17	CD		CALL		DRES									Display Result
18	E0													
19	82													
1A	CD		CALL		GETKY									Wait for key
1B	3D													command to
1C	02													continue
1D	C1		POP		B									Recover addresses
1E	E1		POP		H									
1F	C3		JMP		8209									Go to next table
8220	09													entry
21	82													
22														
23			NOTE:		THE									SHORTER TABLE
24			MUST		BE		TE							RMINATED WITH
25			OO		AT		8320							TO MAKE
26			THIS		PROGRAM									DETECT
27			END		AT		820D							
28														

Figure 6-18

Enter a breakpoint at 8211, just before the first PUSH is executed; and another at 82E0, the start of DRES. Press RUN, and at 8211 observe the stack pointer:

ADDR P MEM 83D3 SP??

The stack is empty. Now execute the PUSH H, and check it again.

8212 05

ADDR P MEM 83D1 SP0B

The stack top contains the address from H,L and points to the data entry, 03:

ADDR T MEM 830B ST03

Now execute PUSH B:

8213 4F

ADDR P MEM 83CF SP19

ADDR T MEM 8319 ST02

Step into subroutine MULT.

8214 CD

82D0 21

ADDR P MEM 83CD SP17

The stack now contains the following, which you can check by pressing NEXT:

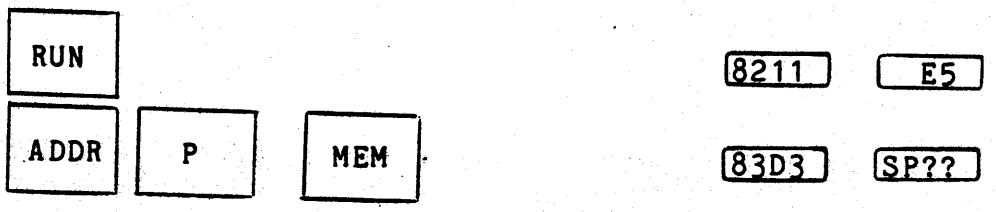
Stack Address	Data	
83CD	SP.17	Return Address for MULT
83CE	82	
83CF	19	Address from B,C
83D0	83	
83D1	0B	Address from H,L
83D2	83	

Now press RUN to reach your breakpoint at 82E0, and review the stack again.

83CD	SP.1A	Return
83CE	82	address
83CF	19	
83D0	83	
83D1	0B	
83D2	83	

The top of the stack has been replaced with the return address for DRES.

Another RUN will display the result, and wait for any key to command continue.



The stack is empty again: that is, the pointer is at the top. If you review the empty part of the stack (starting at 83CF) you will see the present contents of L, H, C and B, but this is not because you placed them there; it happens that the monitor pushes data into them in the same sequence that you used. The monitor shares your stack, so you will find various other data at lower addresses, even though your RET and POP instructions do not themselves alter the stack contents, but only the pointer.

6.7 PROCESSOR STATUS WORD (PSW)

For the PUSH and POP instructions only, register A and the flags are treated as a register pair, with A the high order member. This permits register A and the flags to be saved and recovered despite intervening steps that affect them. Consider this program segment:

```
8200  ADD   D          (A) <- (A) + (D)
  01  PUSH  PSW        Save A,F
  02  INR   E          Count
  03  MOV   A,E        Move counter to A
  04  CPI   06         Test for end
  05  JZ    8209       Jump if end (zero) to POP and exit
  06  POP   PSW        Restore A,F
  07  JNC    8200      Jump if no ADD carry to start of loop
  08  JMP    820B      Else go to carry handling section.
  09  POP   PSW        Restore  A,F
  0A  RET (exit from loop)
  0B  (process carry from ADD)
```

The A register and flags are affected in testing for the end of the loop, and that test is to take precedence over the test for a carry from the ADD. PUSH PSW saves the flags for the test; it also saves register A for the next addition. Note that we have one PUSH and two POP instructions, but only one POP will be executed. The instructions are:

```
F5    PUSH   PSW        Push A and F
                        into the stack.
```

```
F1      POP     PSW         Pop A and F
                            from the stack.
```

6.8 STACK POINTER INSTRUCTIONS

These instructions are defined for completeness. You are urged not to use them when working with MTS until you fully understand 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 restart. Other instructions include:

```
33          INX   SP          Increment stack pointer
3B          DCX   SP          Decrement stack pointer
39          DAD   SP          ((H,),(L)) <- ((H),(L)) + (SP)
F9          SPHL              (SP) <- ((H),(L))
```

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. You can maintain two separate stacks by using SPHL.

6.8.1 Exchange Stack Top with H,L

The 'Stack Top' refers to two bytes: the byte addressed by the stack pointer and the byte at the next higher address. On a RET instruction these provide the return address; a POP instruction brings them to the

designated register pair. Either of those instructions increments the stack pointer twice, so a new stack top is addressed. We have another way of accessing the stack top:

E3 XTHL Exchange stack top with H and L.
(SP) <-> (L)
(SP) + 1 <-> (H)
The stack pointer content is unchanged.
No flags are affected.

This is often used to provide two more bytes of readily available storage when a program requires more than six general purpose registers. For instance if four different memory locations must be accessed we can use BC for one address, DE for a second, and HL for two more by use of XTHL.

6.8.2 Using the Stack

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 preceeding 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. However, it is a poor general practice, and should be used only when trying to save space and squeeze the last instruction of a program, developed in RAM, into a ROM production model.

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.

6.9 SUBROUTINE CLASSIFICATION

We will define four kinds of subroutines, which are not mutually exclusive.

Global Subroutines

Local Subroutines

Reentrant Subroutines

Interrupt Service Routines

6.9.1 Global Subroutines

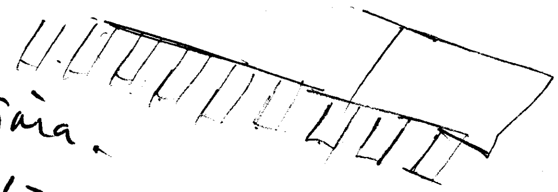
A global subroutine is one which is available to be called from any other program module. Typically it serves a general purpose function such as multiplication, exponentiation, etc. It must be fully specified so that other programmers may use it. A number of restrictions are usually applied, although 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.

d) Global subroutines are almost always transparent to the user, i.e. all registers returned with their content unchanged, except as they are used to return results.



Sigma.

617 843 5000

chips to controll step motors

$1.20

61/64 in

1/8 in thick ¼

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.

Of the subroutines used in the sensor correction problem, clearly INPUT, MULT and DRES could be treated as global subroutines. In fact, you will use them again in a later exercise. TABLU is too specialized: it demands a particular data table organization.

6.9.3 Re-Entrant Subroutines

A reentrant subroutine is one that can be called even though it is already in use. A number of the monitor subroutines exemplify this. Any subroutine that is subject to interrupts and which is called by an interrupt service routine must be reentrant. 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.

6.10 MONITOR SUBROUTINES

The remainder of this chapter describes monitor subroutines that are available to you.

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
Key pressed: Key value in A; CY set

Registers:

A and B

Constraints

Uses output port C and input port A. Interface adaptor must be programmed for these port assignments, which is done by the monitor at power on or Reset.

Leaves output port C loaded with different data depending on which key was pressed.

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 and DELAY

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, C and D are used. The contents of registers 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).

Note: If GETKY is called by a user program while the AUTO/STEP toggle switch is in STEP mode, the delay is exaggerated to about two seconds.

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 register A.

Call

```
CD          CALL  ENTBY
36
03
```

Extent

0336 through 0374, including local subroutines.
Also calls DBYTE and GETKY

Inputs

Keyboard

Outputs

Command key in register A and B. 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.

Registers

A, B, C, D, H, L

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 keys 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 register A.

Call

```
CD          CALL  ENTWD
46
03
```

Extent

0346 through 0374
Including local subroutine. Also calls DWORD, DMEM, and GETKY

Inputs

Keyboard

Outputs

Command key in registers A and B. Last four hexadecimal keys in registers H and L. Number of hexadecimal keys pressed in register D.

Registers

A, B, C, D, H, L

6.10.5 Monitor Display Digit Subroutine (OFFSET)

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  OFFSET
A9
02
```

Extent

02A9 through 02C1

Inputs

a) Hexadecimal value in register A. (Note: a value greater than 0F will result in an erroneous display.)

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 seven segment code is also returned in register A.

C) The address in register D, E is decremented by one.

Registers

a) Register pair H,L is used, in addition to D,E and A.

b) Only the memory location addressed by D,E is affected.

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.

Calls

```
CD          CALL  DMEM
94          Display ((H),(L)) in right hand digits
02
CD          CALL DBYTE
95          Display (A) in right hand digits
02
CD          Call DBY2
98           Display (A) at location ((D),(E))
02
```

Extent

0294 through 02A8
Calls SPLIT and OFFSET

Inputs

DMEM - Memory address in H,L
DBYTE - Byte in A
DBY2 - Byte in A and memory address for display in D,E.

DMEM and DBYTE initialize register pair DE to 83FF to display the byte in the right hand positions.

Outputs

Register C contains 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, B, C, D, E are used

Registers H, L are preserved

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 83F8 the monitor operation may be disrupted.

Output port C is loaded with 80, to turn on the display and energize all keyboard input lines. Register A contains 80 at return.

6.10.7 Monitor Display Word Subroutine - DYPC, DWORD, DWD2

Function

Display two bytes of data as four hexadecimal digits.

Calls

```
CD          CALL DYPC
CE          Displays content of program
02          counter at last RST4 or RST7

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

02CE through 02DC
Calls DBY2

Inputs

Data to be displayed (two bytes):

a) for DYPC: stored at 83DA, 83DB
b) for DWORD and DWD2: in HL
c) for DWD2 only: display address in register pair DE

Outputs

Register C contains more significant byte of display. Register pair DE is decremented by 4 from the initial value provided by DYPC or DWORD or at entry to DWD2.

Registers

All registers are used.
Registers H,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 83F8 the monitor operation may be disrupted.

6.10.8 Monitor Subroutine CLRGT, CLEAR, CLRLP

Function

Clear part or all of the display or memory.

Calls

```
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
02          digits to be cleared in B
```

Extent

0282 through 0293

Inputs

CLEAR, CLRGT - none
CLRLP - number of digits in B
highest address in (H,L)

Output

Contents of display memory area starting at right are set to 0.

Registers

B, H, L

6.10.9 Monitor Subroutine Display Enable (DYEN)

Function

Enable the DMA channel for display. Also causes all keys to be enabled for input test.

Call

```
CD          CALL DYEN
A4
02
```

Extent

02A4 through 02A8

Input

None

Output

Outputs 80 to port C.
Returns 80 in register A

Registers

A is used

Comment

If output port C is used for other purposes, the most significant bit must be set high to enable the display. DYEN accomplishes this. After a call to DYEN the input instruction IN PORTA (DB00) will load FF to the A register if no key is pressed; if any key is pressed at least one bit will be zero.

6.10.10 Monitor Subroutine SPLIT

Function

Separate a byte into two hexadecimal digits, each right justified.

Call

```
CD          CALL  SPLIT
C2
02
```

Extent

02C2 through 02CD

Input

Data byte in register A

Outputs

Data byte in register C. More significant digit in register B. Least significant digit in register A.

Registers

A, B, C

6.10.11 Monitor Subroutine DELAY, DEL1

Function

Wait in a loop for a defined time.

Call

```
CD          CALL DELAY
36          Wait 1.3 milliseconds
02

CD           CALL DEL1
38           Wait for a time
02           set in register A
```

Extent

0236 through 023C

Input

DELAY - None
DEL1 - Enter with a value in register A, proportional to the delay desired.

Output

None

Registers Used

A

MICROCOMPUTER TRAINING WORKBOOK

CHAPTER 7

LOGIC AND BIT MANIPULATION

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.

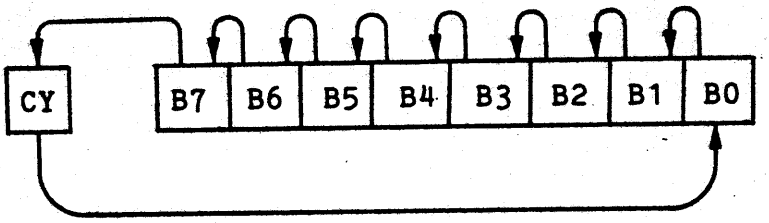
In this chater two sets of instructions will be introduced: rotate commands, which work on the accumulator and carry flag only; and logical functions, which generally involve the accumulator and another register.

7.1 ROTATE COMMANDS

Rotate is a command to move each bit in the accumulator to an adjacent position.

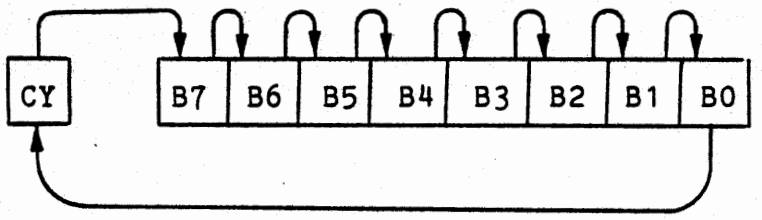
17 RAL Rotate Accumulator Left Through Carry

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.



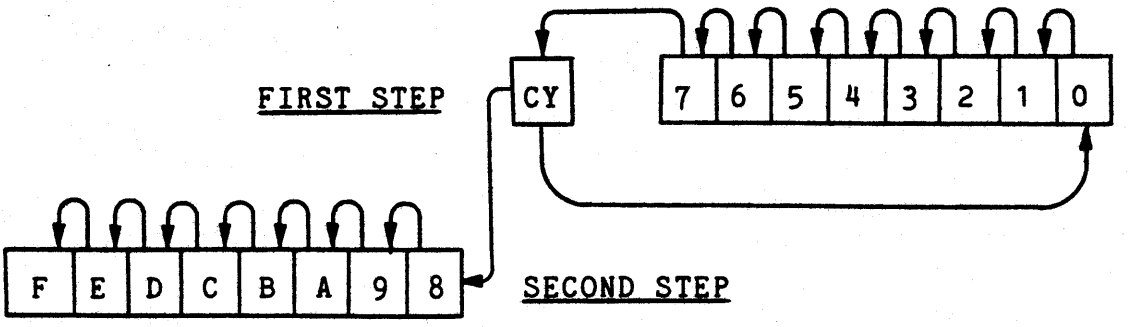
These two rotate commands are sometimes called 'arithmetic shift' because they can be used to double or halve the value of the A register content, and are used in multiplication and division. They can also be used to obtain access to an individual bit. To illustrate the arithmetic properties of rotate, consider the following simple binary numbers:

0111 1110

They are identical, except that the second number has been shifted left one bit, and as a result has been doubled in magnitude.

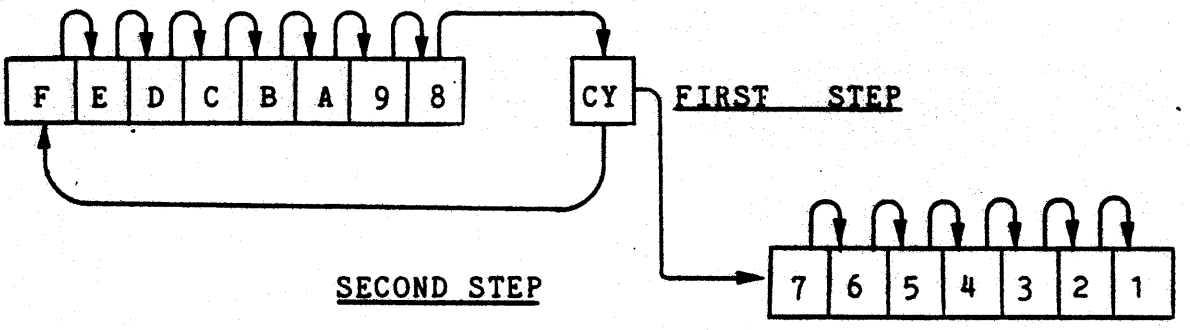
7.1.1 Rotate Exercise

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:



We will 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. If LSB is 1, we will double the value. If LSB is 0, we will halve the value. Place the result in H,L and use DWD2 to display the result at the right side of the display (set (D,E) = 83FF).

The calls to ENTWD and DWD2 are:

```
CD        CALL  ENTWD
46
03

CD        CALL  DWD2
D4
02
```

You can use REG and MEM as the two command keys for double and halve. (When you enter four or more hexadecimal keys, using ENTWD, you will see two digits appear in the right hand position of the display. These show the content of the memory location you have addressed, which is not of interest here but is part of the function of ENTWD). Flow chart and code this exercise yourself, then look at the solution given. If there are differences, try both programs. You will soon realize that a problem solution can be implemented with a variety of programs. A flow chart is shown in Figure 7-1, and a coding sheet in Figure 7-2.

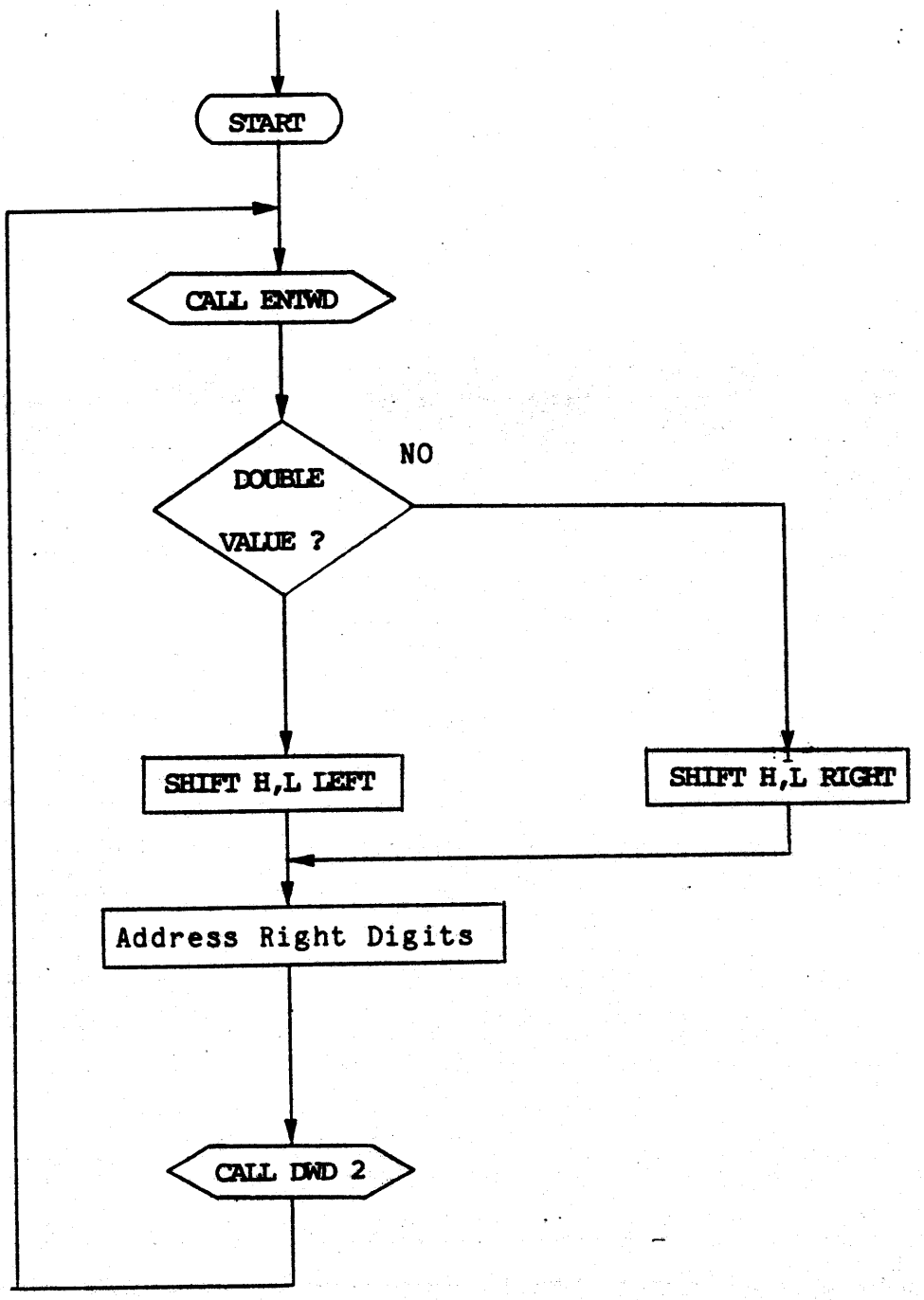

START
CALL ENTWD
DOUBLE
VALUE ?
NO
SHIFT H,L LEFT
SHIFT H,L RIGHT
Address Right Digits
CALL DWD 2

Figure 7-1

ARITHMETIC SHIFT

ADDR	CODE																
8200	00		NOP														
01	00																
02	00																
03	CD		CALL						ENTWD								Get word and command
04	46																
05	03																
06	1F		RAR														Command bit to CY
07	D2		JNC					8214									Jump if bit = 0
08	14																
09	82																Double the value
0A	B7		ORA				A										Clear carry
0B	7D		MOV				A	,	L								Less significant byte
0C	17		RAL														
0D	6F		MOV				L	,	A								
0E	7C		MOV				A	,	H								More significant byte
0F	17		RAL														
8210	67		MOV				H	,	A								
11	C3		JMP				8	2	1	A							Go display result
12	1A																
13	82																Halve the value
14	7C		MOV				A	,	H								More significant byte
15	1F		RAR														
16	67		MOV				H	,	A								
17	7D		MOV				A	,	L								Less significant byte
18	1F		RAR														
19	6F		MOV				L	,	A								
1A	11		LXI				D	,	83FF								Address display right digit
1B	FF																
1C	83																
1D	CD		CALL						DWD2								
1E	D4																
1F	02																
8220	C3		JMP				8	2	0	0							
21	00																
22	82																
23																	
24																	
25																	
26																	
27																	
28																	

FIGURE 7-2

7.1.2 Multiplication and Division by Two

Now we will modify the program to allow repeated multiplication or division by 2. At the end of your program replace the final jump with SHLD, to store the content of (H,L) in two memory locations (SHLD 8300).

Now call ENTWD by placing another call at the next location in your program, 8223. Test the second least significant bit in the command. If it is zero, use the new value of H,L. If it is one, recover the old value, using LHLD.

To test the second least significant bit in the command requires two right shift commands. Now restore the least significant bit to the carry flag by a left shift command, and jump back to decide whether to multiply or divide by 2.

Now we will define command key functions, as follows:

REG (or NEXT) New Data x 2

MEM (or RUN) New Data / 2

CLR (or STEP) Old Data x 2

BRK (or ADDR) Old Data / 2

After entering data once using REG or MEM, repeated depressions of CLR or BRK will successively multiply (or divide) the entry number. Note that this type of division is by truncation, e.g. 5/2 = 2, not 2.5, and 1/2 = 0.

An extension of the flow chart of Figure 7-1, to follow the CALL DWD2 (instead of returning to start) is shown in Figure 7-3, and the code in Figure 7-4.

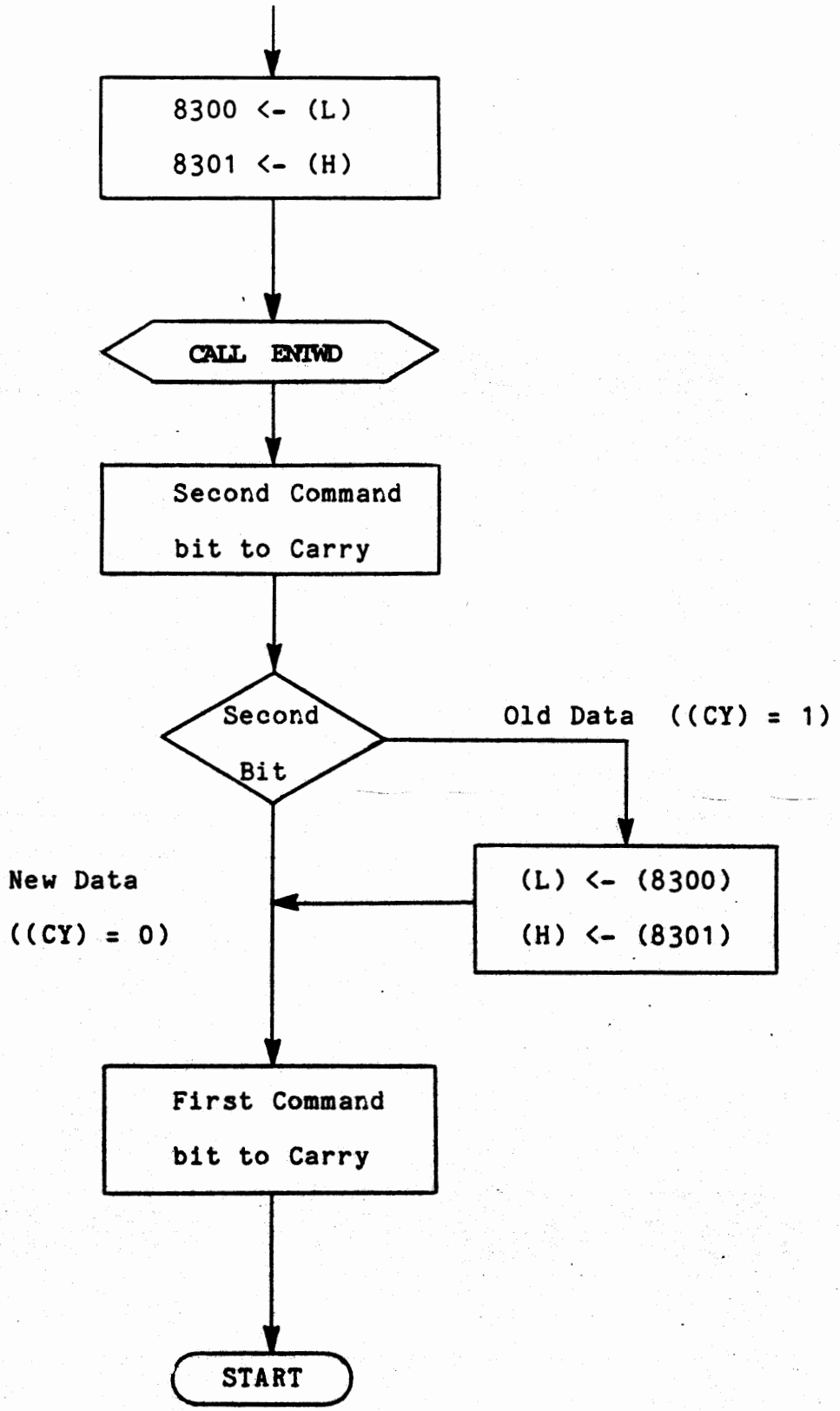

8300 <- (L)
8301 <- (H)
CALL ENIWD
Second Command
bit to Carry
Second
Bit
Old Data ((CY) = 1)
(L) <- (8300)
(H) <- (8301)
New Data
((CY) = 0)
First Command
bit to Carry
START

Figure 7-3

REVISED ARITHMETIC SHIFT

ADDR		CODE														
8220	0	22			SHLD				8300							
	1	00														
	2	83														
	3	CD			CALL				ENTWD							Get word and command
	4	46														
	5	03														
	6	1F			RAR											Second bit to CY
	7	1F			RAR											
	8	D2			JNC				822E							Use new data
	9	2E														if bit = 0
	A	82														
	B	2A			LHLD				8300							Recover old data
	C	00														if bit = 1
	D	83														
	E	17			RAL											First bit to CY
	F	C3			JMP				8207							
8230	0	17														
	1	82														
	2															
	3															
	4															
	5															
	6															
	7															
	8															
	9															
	A															
	B															
	C															
	D															
	E															
	F															
8	0															
	1															
	2															
	3															
	4															
	5															
	6															
	7															
	8															

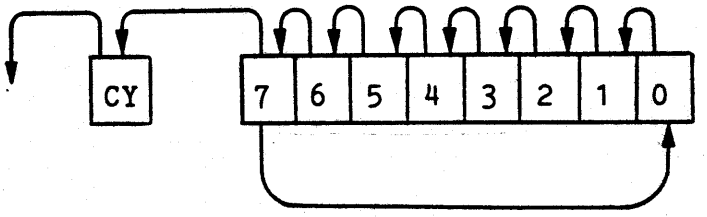
FIGURE 7-4

7.1.3 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.

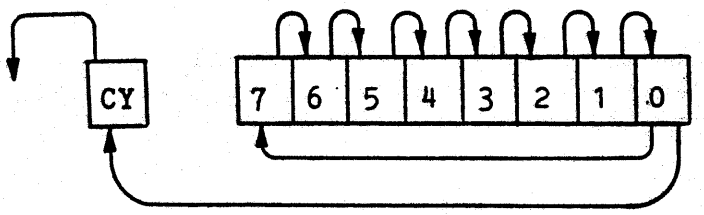
07 RLC Rotate Left

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.



OF RRC Rotate Right

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 Double and Halve parts of the program 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		1234	48D0
CLR		1234	90A1
CLR		1234	2143

7.1.4 Other Shift Functions

A left shift of the accumulator, since it doubles the value of its content, can be duplicated by adding it to itself using the ADD A instruction. This differs from the rotate left command in that it always leaves zero in the least significant bit. It also sets or clears all flags, while the rotate instructions affect only the carry flag. The double precision add instruction DAD H can be used to duplicate shifting left in the H,L register pair.

The addition with carry instructions will be covered in another chapter, but one of them is similar to ADD A and so we introduce it here:

HEX CODE: 8F

MNEMONIC: ADC A

MEANING: Add the content of register A and the content of the carry flag to the content of register A and place the result in register A. All flags are affected.

The result is identical to RAL except that all flags are affected, because the old carry is added in.

7.1.5 Carry Flag Controls

The commands RAR, RAL and ADC A all enter the carry flag into register A. It is often necessary to operate on the carry flag before using one of these. The carry flag can be cleared, set, or complemented, by the following instructions:

B7	ORA A	Clears the carry flag. Sets or clears other flags according to the content of register A. (CY) <- 0 (see Section 7.3.3 for more detail)
37	STC	Set the carry flag. (CY) <- 1

3F	CMC	Complement the carry flag.
		If (CY) = 1, (CY) <- 0
		if (CY) = 0, (CY) <- 1

7.2 PROGRAM EXERCISE I

We will plan a program which will display the content of a register in binary form. Instead of calling a monitor subroutine to display a byte as two hexadecimal digits, we will assign a digit to represent each bit. Then according to whether that bit is 1 or 0 we will store a symbol in the memory location that is accessed by the DMA channel for the corresponding digit.

7.2.1 Display Segments

In order to choose symbols for 0 and 1 you need to know how the individual segments of the display are controlled. Each of the eight display locations on your MTS has seven line segments and a decimal point, a total of eight elements. The DMA and display hardware are designed so that each location is controlled by one byte of memory and each element by one bit.

First we will write a program to find out how the bits are assigned, and which memory location controls which display location. You will need this monitor input subroutine:

```
CD      CALL ENTBY
36
03
```

ENTBY accepts data from the keyboard, displaying the value of the hexadecimal key(s) depressed in the rightmost two locations of the MTS

display. This is the subroutine used by the monitor to enter data using the MEM command, so you are familiar with its operation. ENTBY exits whenever a command key is depressed, with the values entered in register L.

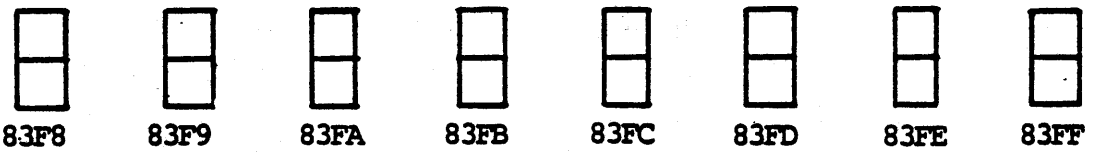
The eight display locations are controlled by the contents of memory addresses 83F8 - 83FF. Loading a byte (two hex keys) in one of these locations will turn on each display element whose controlling bit is set to 1. Write a simple program to test the assignment of display elements of each bit. A simple solution is shown in Figure 7-5. Load the program and experiment, then try displaying in different locations. After you have experimented look at Figure 7-6 and try some of the examples presented.

TO DETERMINE BIT ASSIGNMENTS FOR DISPLAY

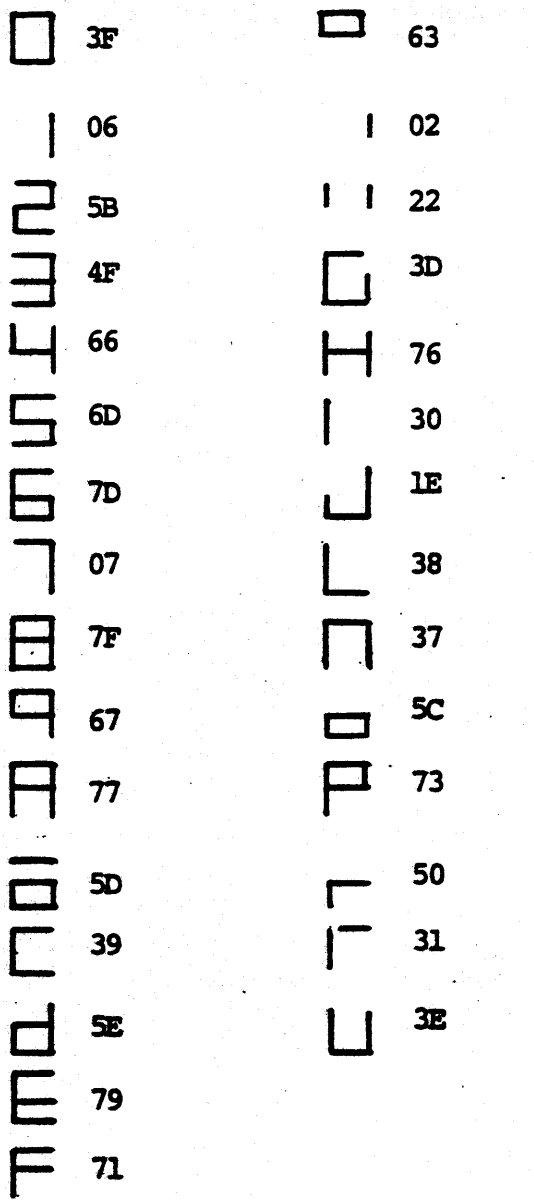
ADDR	CODE													
8200	1	D		C	A	L	L		E	N	T	B	Y	
01	3	6												
02	0	3												
03	7	D		M	O	V		A	,	L				
04	3	2		S	T	A		8	3	F	8			
05	F	8												
06	8	3												
07	C	3		J	M	P		8	2	0	0			
08	0	0												
09	8	2												
0A														
0B														
0C														
0D														
0E														
0F														
8210														
11														
12														
13														
14														
15														
16														
17														
18														
19														
1A														
1B														
1C														
1D														
1E														
1F														
8220														
21														
22														
23														
24														
25														
26														
27														
28														

FIGURE 7-5

DISPLAY ADDRESS ASSIGNMENTS



Bytes to Generate Various Symbols



Byte Value for each Segment

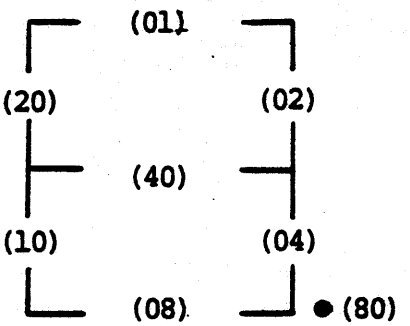


Figure 7 - 6

7.2.2 Binary Display

Now with some suitable symbols we can create the display of a byte in its eight-bit binary form, with a symbol for one or zero shown in each display location. What we need are symbols to represent 0s and 1s, and a program which will display the symbols in the eight display locations. In preparing a flow chart and coding your program, use these hints:

a) Use ENTBY to fetch a byte

b) Initialize addresses and counters

c) Write a loop to store the appropriate symbol in the display location corresponding to each bit in the byte.

It is tricky, but try to devise a solution of your own. Figure 7-7 shows a flow diagram for the program and a coding solution is given in Figure 7-8. The program can run equally well run from most significant bit to least significant or vice versa. This will determine the first display address, whether it is to be incremented or decremented, and whether the shifting is to be left or right. For this program it does not matter which of the six methods of shifting register A we choose, except that there may be some reason to want the original byte restored at the end.

Load and test your program. If you have problems, you may meet a difficulty in using the monitor with a program that operates the display. Each time you step, or reach a breakpoint, the monitor will destroy your display data, since it writes to the same locations. This

becomes a nuisance if a program executes several instructions, stores data in a display position, and repeats a loop with conditional jumps. It is often wiser to place your display data in some different memory area (say 83A8 to 83AF), so that you can inspect those memory locations to see what your display data was. Then change that address to 83F8 when the program is successful.

You may wish to exercise the program using different symbols for 0 and 1. Look again at Figure 7-6, or use your imagination. Save this program - we will use it in the next exercise.

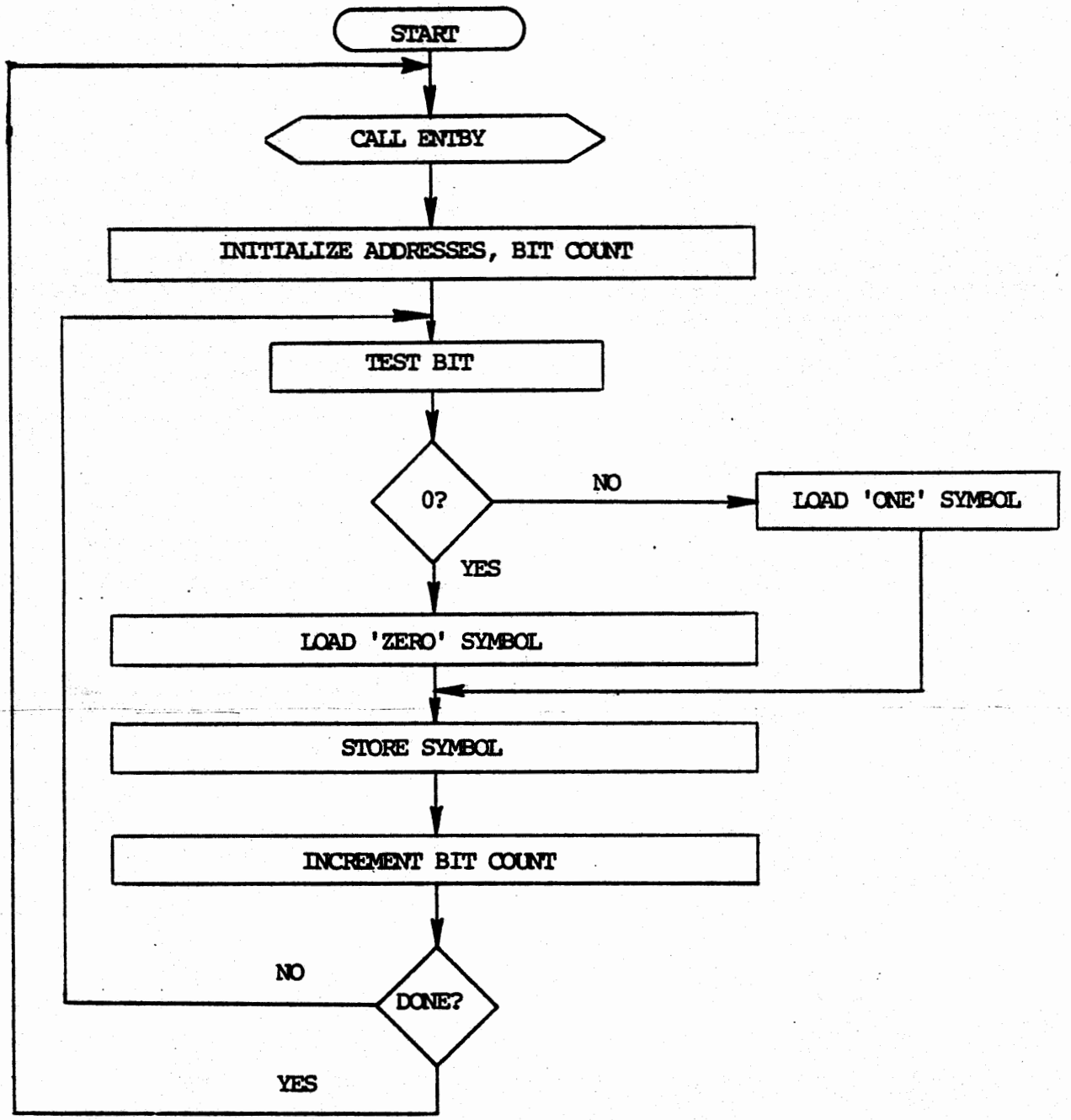

START
CALL ENTBY
INITIALIZE ADDRESSES, BIT COUNT
TEST BIT
0?
NO
LOAD 'ONE' SYMBOL
YES
LOAD 'ZERO' SYMBOL
STORE SYMBOL
INCREMENT BIT COUNT
NO
DONE?
YES

FIGURE 7-7

BINARY DISPLAY

INTEGRATED COMPUTER SYSTEMS MICROCOMPUTER TRAINING SYSTEM CODING SHEET

ADDR	CODE			
8200	00		NOP	
01	00			
02	00			
03	CD		CALL ENTBY	Get a byte
04	36			to display
05	03			
06	01		LXI B, 8221	Change for
07	21			different symbols
08	82			
09	11		LXI D, 83F8	Address display
0A	F8			(use 83A8 for
0B	83			debugging)
0C	26		MVI H, 08	Set bit count
0D	08			
0E	7D		MOV A, L	(A) ← byte
0F	07		RLC	(CY) ← next bit
8210	6F		MOV L, A	(L) ← shifted byte
11	0A		LDAX B	Zero symbol
12	D2		JNC 8218	Jump if bit=0
13	18			
14	82			
15	03		INX B	Address and load
16	0A		LDAX B	one symbol
17	0B		DCX B	Restore address
18	12		STAX D	Display ← symbol
19	13		INX D	Address next digit
1A	25		DCR H	Count bits
1B	C2		JNZ 820E	Continue until
1C	0E			bit count = 0
1D	82			
1E	C3		JMP 8203	
1F	03			
8220	82			
21	3F			Symbol for 0
22	06			Symbol for 1
23	5C			Symbol for 0
24	06			Symbol for 1
25	00			Blank for 0
26	04			Symbol for 1
27				
28				

FIGURE 7-8

7.3 LOGICAL FUNCTIONS

Logical 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 $\cap$.

$X \cap Y$ (X) 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 ⌒ 11101001 = 01101000

or 6E ⌒ E9 = 68

A logical 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,L) is generated, and the result placed in register A, by:

ANA	r	AND (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.

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 is 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	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 by $\oplus$.

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 $$6E \oplus E9 = 85$$

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	XOR (r) with (A); place the result in A. (A) <- (A) $\oplus$ (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.

7.3.5 Immediate Logical Functions

For each of the logical 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:

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 logical 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
```

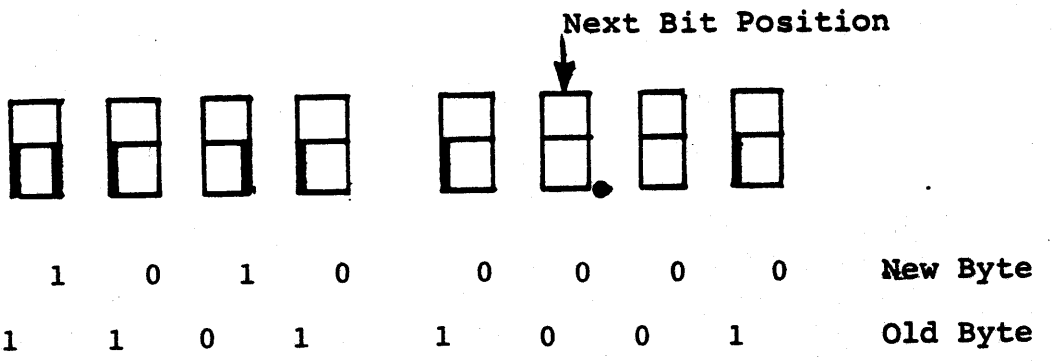
than to shift the data bit to the carry flag and jump if carry.

7.4 PROGRAM EXERCISE II

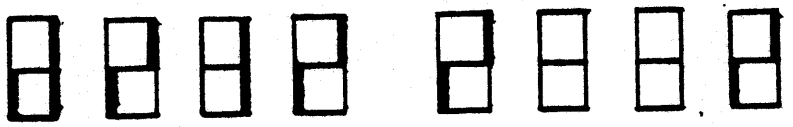
Now we will plan an exercise using bit shifting and display techniques to demonstrate logical functions. We will accept eight bits as a sequence of zeros and ones from the keyboard and display them as they are received, using blank for zero and lower right segment for one. A decimal point will appear in the location where the next bit is to be entered. As the eight bits are entered we will also display a previously entered data byte, using blank for zero and lower left segment for one. The appearance of the display as it will appear is shown in Figure 7-9.

7.4.1 Keyboard Utilization

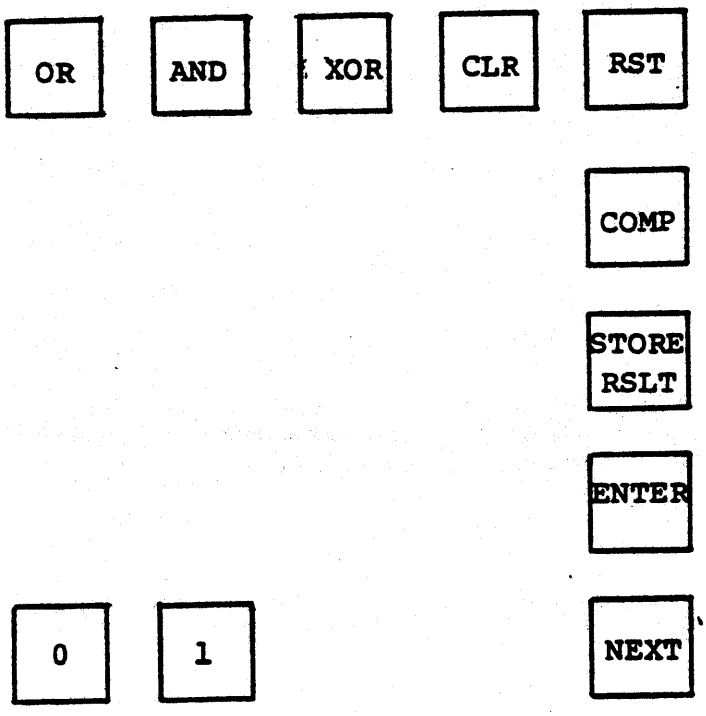
Command keys will be used to generate logical functions of Old and New Bytes, and the Result Byte will be displayed along with the Old and New Bytes. This is also shown in Figure 7-9. Other command keys will be used to control the data entry sequence.



Display During binary data entry



Display showing result of OR



Keyboard Functions

LOGICAL FUNCTIONS - KEYBOARD AND DISPLAY

FIGURE 7 - 9

Control and Function Keys

NAME	KEY	FUNCTION
CLEAR	CLR	Clear all bits of New Byte to 0. Place the bit position marker at the most significant bit.
NEXT	NXT	Move the bit position marker one bit to the right without changing the data
ENTER	STEP	Replace Old Byte with New Byte. Do not change New Byte. Place the bit position marker at the most significant bit.
STORE	RUN	Replace Old Byte with Result Byte, if a result is displayed. Otherwise treat as ENTER.
COMP	ADDR	Complement New Byte.
OR	REG	Form the logical OR of Old and New Bytes and place it in Result Byte.
AND	MEM	Form the logical AND of Old and New Bytes and place it in Result Byte.
XOR	BRK	Form the Exclusive OR of Old and New Bytes and place it in Result Byte.

7.4.2 Outlining the Program

This exercise is sufficiently complicated that we will build it up as a set of subroutines that can be tested individually. First we will prepare broad-brush descriptions of the major modules. A detailed specification will be presented for each module as it is developed:

BININ: Binary input subroutine to accept binary and command keys, and assemble a byte of data.

MBIDY: Multiple Binary Display subroutine to display New Byte, Old Byte, Result Byte, and the bit marker (decimal point) showing the present bit position.

CONTROL: Command Processing module to interpret and execute the commands.

We will be concerned with four data bytes that must be accessed by different modules. This is too many to conveniently keep in registers, so we will assign a fixed memory location for each. Assignment of memory locations will influence program efficiency. New Byte and the bit position marker will be referenced repeatedly. These can be loaded and stored with the LHLD and SHLD commands if they are in adjacent locations.

These considerations lead to the following assignments:

(8301) Result Byte

(8302) Old Byte

(8303) New Byte

(8304) Bit Marker

Program memory assignments will be:

8200 - 866F CONTROL

8270 - 828F BININ

82A0 - 82EF MBIDY

82F0 - 82FF Table of Symbols

During the development of BININ we will use the binary display program you have already developed; later we will replace it with MBIDY.

7.4.3 Binary Display Subroutine (BINDY)

This is a formal description of the display program used in section 7.2, converted into a subroutine:

Function

Display the content of register L in binary format, using a pair of symbols addressed by the content of register pair BC.

Call

```
CD   CALL BINDY
06
82
```

Extent

8206 through 822F
(including symbol table)

Input Data

Register L Data Byte

Output Data

Symbols are stored in DMA locations 83F8 through 83FF according to content of L.

Registers

Registers A, B, C, D, E, H, L used

7.4.4 Binary Input Subroutine (BININ)

In the preceding exercise we used the monitor subroutine ENTBY to get a byte and display it in binary. Now we will create a binary input subroutine BININ, which will call the monitor subroutine GETKY.

Function

Fetch a key using monitor subroutine GETKY. If a command key is received, return with carry clear. If a binary key (0 or 1) is received, enter it into the data byte in the position indicated by the bit marker, and shift the bit marker right. If a hexadecimal key other than 0 or 1 is received, use its least significant bit as a binary input. Data and bit markers are kept in memory.

Call

```
Call BININ
A3
82
```

Extent

82A3 through 82BF

Input Data

Bit Position Marker
Data Byte

Output Data

```
Carry       0 if commands; 1 if binary
Register A  Command Key if any
Register H  Bit Position Marker
Register L  Data Byte
```

Registers

All registers used

Constraints

At the first entry for a new byte:

a) Bit Marker should be 1000 0000

b) New Byte should be 0000 0000

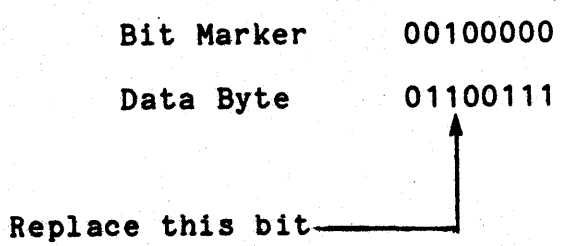
Note that when a command key is pressed, BININ is to behave exactly as GETKY does: return with carry clear. This is readily accomplished with the conditional return.

```
DO        RNC          Return if no carry
```

Since any hexadecimal key is to be treated as binary according to its least significant bit, we can either shift that bit into the carry and ignore the other bits, or we can mask the other bits with the immediate AND instruction, ANI 01.

7.4.5 Modifying Single Bits in a Data Byte

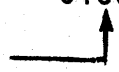
We have defined a bit marker to keep track of which bit is to be entered, and we will use it to modify individual bits. For example:



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:

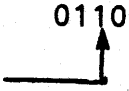
Key is zero:

Complement the bit marker	11011111
Data byte	01100111
AND result	01000111

Bit set to 0 

Key is one:

Bit marker	00100000
Data byte	01100111
OR result	01100111

Bit set to 1 

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):

```
Bit marker          00100000
Data byte           01100111


OR  result          01100111
     Bit set to 1 ----^

Bit marker          00100000

XOR if key 0        01000111
     Bit set to 0 ----^
```

The following technique can be used without any conditional jump instructions whatever. First, mask any unwanted high order bits to ensure that the value is 0 or 1. Then decrement the accumulator so that the key is represented thus:

Key 0	1111 1111
Key 1	0000 0000

Now AND this result with the bit marker:

BIT MARKER	0010 0000
Key 0	0010 0000
Key 1	0000 0000

Now we have the complement of the desired bit. Save this in another register, move the data byte to A and force the marked bit to 1 by an OR with the marker.

Data byte	0110 0111
Bit Marker	0010 0000
OR Result	0110 0111

Now XOR this result with the complemented key. The result will be:

```
Key 0          0100 0111
Key 1          0110 0111
   Bit set ------^
```

XOR with 0 preserves each bit; XOR with 1 complements the bit. After the result is generated and saved in register L, you must shift the bit marker right and store it.

A flow chart for BININ is shown in Figure 7-10, and coding for this routine in Figure 7-11. Figure 7-12 shows a revision of the binary display code developed in section 7.2. Figure 7-13 presents the code for a calling routine which initializes New Byte and the bit marker. The calling program is stored at 8230 to preserve the code you entered previously at 8200, so enter JMP 8230 at address 8200, and convert the binary display program into the subroutine BINDY.

Two sets of symbols are provided. Locations 8221 and 8222 are the symbols for zero and one used by the monitor for hex displays, and the program starts with these values. When the first key is pressed, the first location will show the value of the depressed key, and all others will display zeroes.

The symbols at 8223 and 8224 are a blank for zero and lower right segment for one. Location 8207 determines which symbols are to be used.

Load the programs and try both sets of symbols. Remember that only the least significant bit of the pressed key is tested, so each will have an effect.

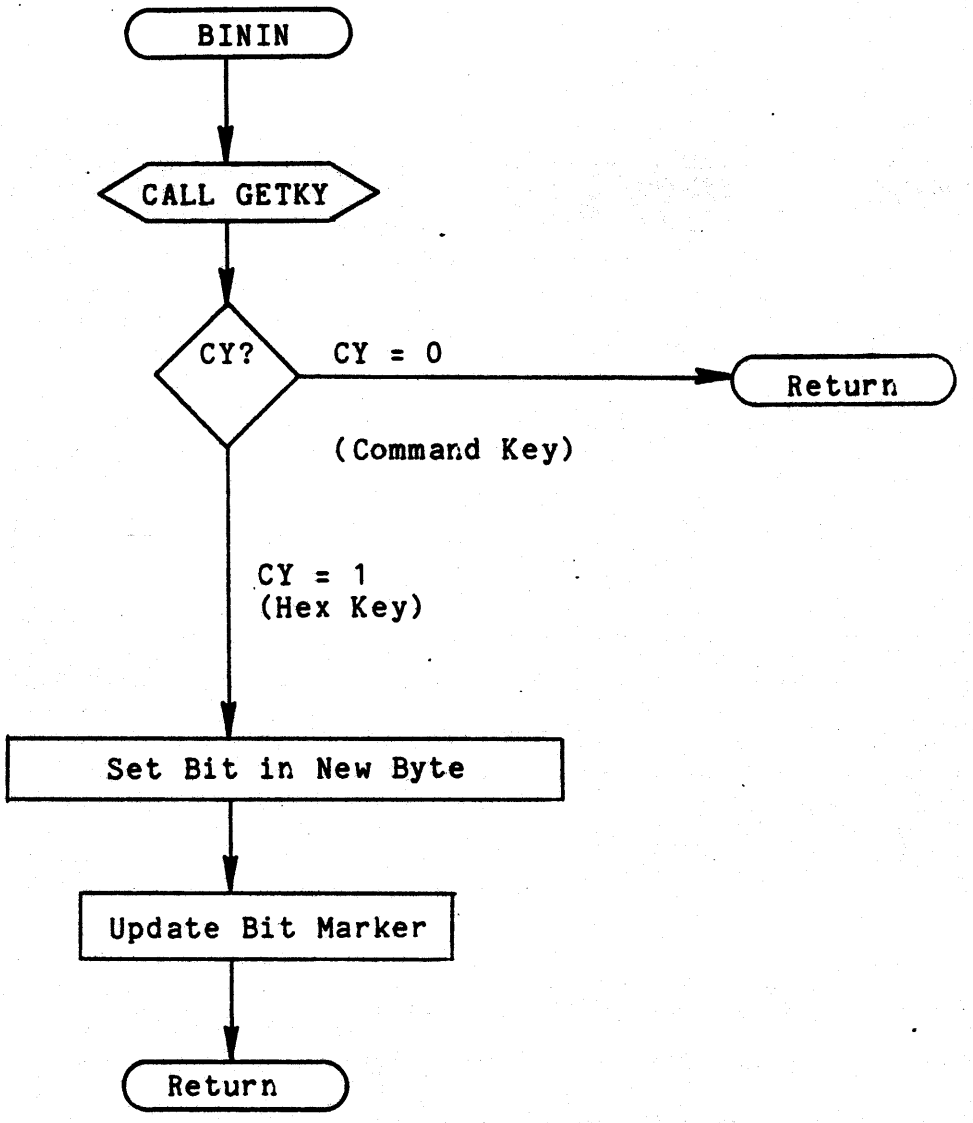

BININ
CALL GETKY
CY?
CY = 0
Return
(Command Key)
CY = 1
(Hex Key)
Set Bit in New Byte
Update Bit Marker
Return

Figure 7-10

BINARY INPUT SUBROUTINE BININ

ADDR	CODE		INSTRUCTION	OPERAND	COMMENT
8 2 7 0	2	A	LHLD	8303	Load
1	0	3			(L) ← data byte
2	8	3			(H) ← bit marker
3	C	D	CALL	GETKY	
4	3	D			
5	0	2			
6	D	0	RNC		Return if command
7	E	6	ANI	01	Mask unwanted
8	0	1			high order bits
9	3	D	DCR	A	11111111 if key 0
A	A	4	ANA	H	Places one in bit
B	4	7	MOV	B, A	position if key=0
C	7	D	MOV	A, L	(A) ← data byte
D	B	4	ORA	H	Force bit to 1
E	A	8	XRA	B	Set bit to 0 or 1
F	6	F	MOV	L, A	(L) ← data byte
8 2 8 0	7	C	MOV	A, H	Shift bit marker
1	0	F	RRC		
2	6	7	MOV	H, A	
3	3	7	STC		Mark binary input
4	2	2	SHLD	8303	
5	0	3			
6	8	3			
7	C	9	RET		
8					
9					
A					
B					
C					
D					
E					
F					
8 0					
1					
2					
3					
4					
5					
6					
7					
8					

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

FIGURE 7-11

BINARY DISPLAY - SUBROUTINE

ADDR	CODE						Comments
8200	C3		JMP	8230			Jump to
01	30						calling program
02	82						
03	00		NOP				
04	00						
05	00						
06	01		LXI	B, 8221			Change for
07	21						different symbols
08	82						
09	11		LXI	D, 83F8			Address display
0A	F8						(use 83A8 for
0B	83						debugging)
0C	26		MVI	H, 08			Set bit count
0D	08						
0E	7D		MOV	A, L			(A) ← byte
0F	07		RLC				(CY) ← next bit
8210	6F		MOV	L, A			(L) ← shifted byte
11	0A		LDAX	B			Zero symbol
12	D2		JNC	8218			Jump if bit=0
13	18						
14	82						
15	03		INX	B			Address and load
16	0A		LDAX	B			one symbol
17	0B		DCX	B			Restore address
18	12		STAX	D			Display ← symbol
19	13		INX	D			Address next digit
1A	25		DCR	H			Count bits
1B	C2		JNZ	820E			Continue until
1C	0E						bit count = 0
1D	82						
1E	C9		RET				Return to make
1F	00		NOP				this a subroutine
8220	00		NOP				
21	3F						Symbol for 0
22	06						Symbol for 1
23	5C						Symbol for 2
24	06						Symbol for 1
25	00						Blank for 0
26	04						Symbol for 1
27							
28							

FIGURE 7-12

CALLING PROGRAM FOR BININ, BINDY

ADDR	CODE											
8230	21		L	X	I		H	,	8	0	0	0	(H) ← Bit marker
1	00												(L) ← 00 for
2	80												new byte
3	22		S	H	L	D			8	3	0	3	(8303) ← new byte
4	03												(8304) ← bit marker
5	83												
6	CD		C	A	L	L			B	I	N	D	Y	Display
7	06												
8	82												
9	CD		C	A	L	L			B	I	N	I	N	Input
A	70												
B	82												
C	C3		J	M	P				8	2	3	6		
D	36												
E	82												
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 7-13

7.4.6 Multiple Binary Display Subroutine (MBIDY)

The next step in program development will be a subroutine to permit the display of Old Byte, New Byte, Result Byte, and bit position marker, all at once. We will call the subroutine once for each of the bytes to be displayed, using a different pair of symbols for each of Old Byte, New Byte and Result Byte. During the first call we will clear the display, and as the subroutine builds the display it will OR the 0 or 1 symbol of the data byte to be displayed with the pre-existing display. Store the display data at another location, 83A8 through 83AF, while debugging. Then if you need to step through your program or use breakpoints the display data will be available for observation at 83A8 - 83AF, even though the monitor has used the content of the display locations.

Function

Display the data byte addressed by (H,L) in binary format, using a pair of symbols (to represent 0 and 1) addressed by the content of register pair (B,C). Pre-existing data in the display is cleared if (A) = 00 at entry. It is preserved if (A) = FF.

Call

```
CD        CALL      MBIDY
A0
82
```

Extent

82A0 through 82FF
(including symbol table, 82F0 - 82FF)

Input Data

```
((H,L))                 Data Byte
(B,C)       Symbol Address:
((B,C))                 Symbol for 1
((B,C) - 1)             Symbol for 0
(A)                     Mask to retain or clear old display
```

Output Data

Symbols are OR'ed into display locations 83F8 through 83FF according to the content of data byte, after old display is masked.

```
(BC)        decremented by 2
(A)         set to FF
```

Registers

All registers are used

In order to selectively clear or retain the display, MBIDY is entered with either 00 or FF in register A. The subroutine will form the AND of this initial mask with the complement of all segments used for the symbols displayed. This mask is AND'ed with the pre-existing bit patterns in each display location. Then the appropriate symbol for a bit of the current data byte is OR'ed into the display to create the new display for that bit position.

The efficient way to handle this is to create the final mask only once, early in the subroutine, and keep it for use as each bit of the data byte is processed. The mask is in register A. The symbols are addressed by B,C and the data byte by H,L. Once we have obtained the data byte, the symbols for 0 and 1, and created the final mask, we can keep all of these in registers and push the original contents of BC and HL onto the stack to get them out of the way. We will still find the registers fully used. Possible register assignments are:

B	Data Byte
C	Mask
D	Symbol for 1
E	Symbol for 0
H	Display
L	Address

With this arrangement we are left with no place for a bit counter, which is a nuisance because we must then use a memory location (or the stack). Moreover, with this set of assignments we must move the data byte into A to shift, at the same time that we need A for masking the old display. This is a good place to use the XTHL instruction, with register assignments like this:

B	Bit Counter		
C	Mask		
D	Symbol for 1		
E	Symbol for 0		
H	Display	or	Data Byte
L	Address	or	Not Used
Stack	Data Byte	or	Display
	Not Used	or	Address

These assignments are used in the solution given for this problem. Recall the storage assignments that have been made for the Bytes and bit marker:

8301	Result Byte
8302	Old Byte
8303	New Byte
8304	Bit Marker

We will store symbols for the displays at successive locations:

82F8	Result	0	symbol
82F9	Result	1	symbol
82FA	Old Byte	0	symbol
82FB	Old Byte	1	symbol
82FC	New Byte	0	symbol
82FD	New Byte	1	symbol
82FE	Bit Marker		0 symbol
82FF	Bit Marker		1 symbol

These sequences make it possible for the calling program initially to load an address for its first call, thereafter decrementing the address in a loop. We could put this loop inside the subroutine, but this would make it very specialized. With an external loop we can make MBIDY very general in function. A flow chart and coding sheets for MBIDY appear in Figures 7-14 through 7-17.

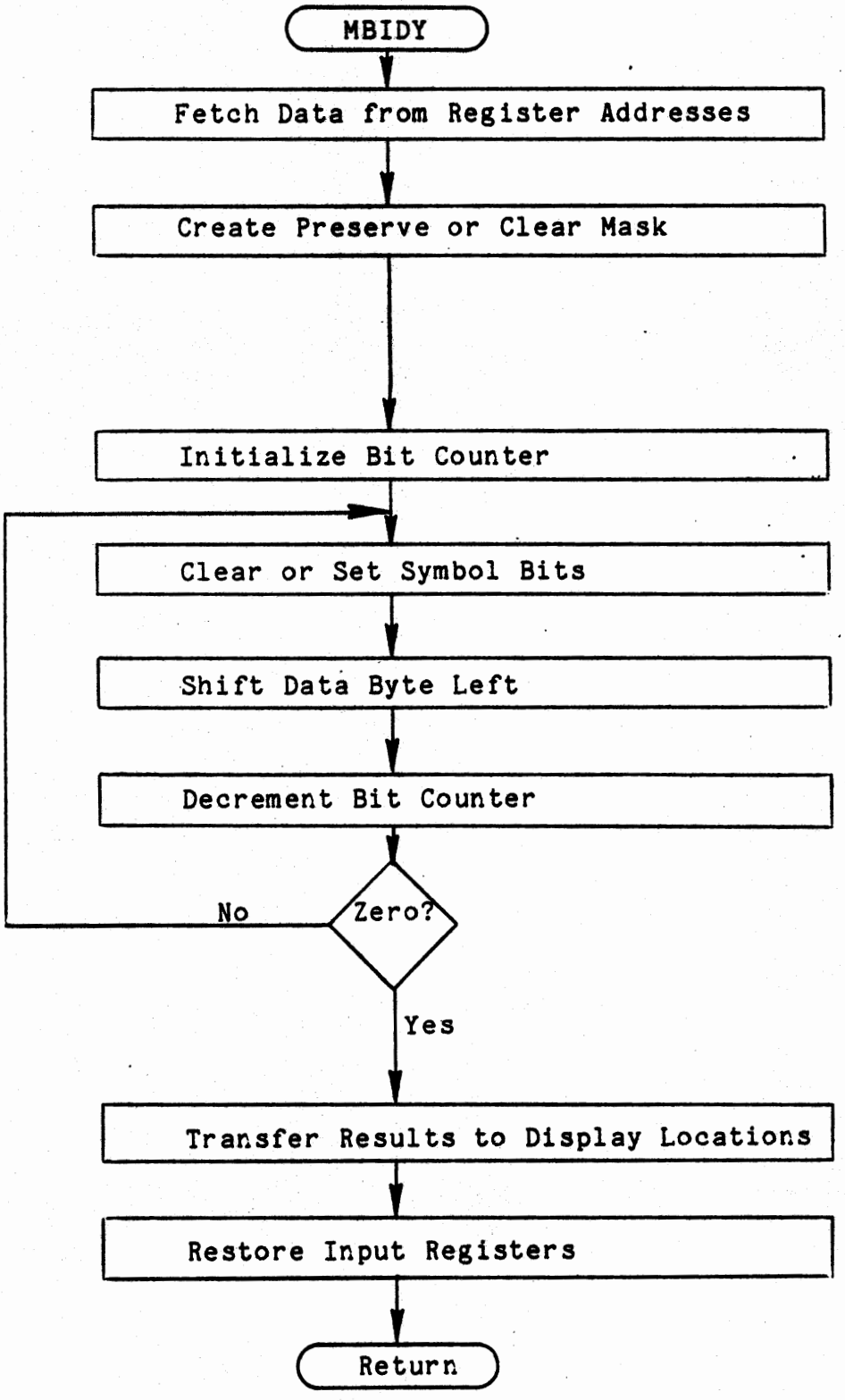

MBIDY
Fetch Data from Register Addresses
Create Preserve or Clear Mask
Initialize Bit Counter
Clear or Set Symbol Bits
Shift Data Byte Left
Decrement Bit Counter
Zero?
No
Yes
Transfer Results to Display Locations
Restore Input Registers
Return

Figure 7-14

Multiple Binary Display with Mask   MBIDY

INTEGRATED COMPUTER SYSTEMS MICROCOMPUTER TRAINING SYSTEM CODING SHEET

ADDR		CODE	Instruction	Comment
8 2A	0	E5	PUSH H	Save data address
	1	66	MOV H,M	(H) ← data byte
	2	6F	MOV L,A	(L) ← mask (temporary)
	3	0A	LDAX B	
	4	57	MOV D,A	(D) ← symbol for 1
	5	0B	DCX B	
	6	0A	LDAX B	
	7	5F	MOV E,A	(E) ← symbol for 0
	8	0B	DCX B	
	9	C5	PUSH B	
	A	B2	ORA D	Create final mask
	B	2F	CMA	with 0 in each
	C	A5	ANA L	position to be
	D	4F	MOV C,A	cleared, 1 to retain
	E	06	MVI B,08	(B) ← Bit Counter
	F	08		
8 2B	0	E5	PUSH H	Stack ← Data Byte
	1	21	LXI H,83A8	(HL) ← Display Addr
	2	A5		
	3	85		
	4	7E	MOV A,M	(A) ← Old Display
	5	A1	ANA C	Mask
	6	B3	ORA E	Enter 0 Symbol
	7	E3	XTHL	(H) ← Data byte
	8	29	DAD H	(CY) ← High bit
	9	E3	XTHL	(HL) ← Display Addr
	A	D2	JNC 82BF	Jump if bit = 0
	B	EF		
	C	82		
	D	A1	ANA C	Clear 0 Symbol
	E	B2	ORA D	Enter 1 Symbol
	F	77	MOV M,A	Display ← Result
8	0			
	1			
	2			
	3			
	4			
	5			
	6			
	7			
	8			FIGURE 7-15

ADDR	CODE													
8 2C 0	2	C		I	N	R	L							Address next digit
1	0	5		D	C	R	B							Count bits
2	C	2		J	N	Z	8	2	B	4				
3	B	4												
4	8	2												
5	E	1		P	O	P	H							Discard data byte
6	2	1		L	X	I	H	,	8	3	A	8		Address memory
7	A	8												
8	8	3												
9	1	1		L	X	I	D	,	8	3	F	8		Address display
A	F	8												
B	8	3												
C	7	E		M	O	V	A	,	M					Copy display data
D	1	2		S	T	A	X		D					into DMA area
E	2	3		I	N	X	H							
F	1	C		I	N	R	E							
8 2D 0	C	2		J	N	Z	8	2	C	C				
1	C	C												
2	8	2												
3	C	1		P	O	P	B							Restore symbol address
4	E	1		P	O	P	H							Restore data address
5	3	E		M	V	I	A	,	F	F				Load A with mask
6	F	F												to retain data
7	C	9		R	E	T								
8														
9														
A														
B														
C														
D														
E														
F														
8 0														
1														
2														
3														
4														
5														
6														
7														
8														

FIGURE 7-16

ADDR	CODE	
82E 0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
A		
B		
C		
D		
E		
F		SYMBOL TABLE
82F 0	00	Result Symbol 0
1	02	Result Symbol 1
2	00	Old Byte Symbol 0
3	10	Old Byte Symbol 1
4	00	New Byte Symbol 0
5	04	New Byte Symbol 1
6	00	Bit marker Off
7	00	with results
8	00	Results Off
9	00	during data entry
A	00	Old byte 0
B	10	Old byte 1
C	00	New byte 0
D	04	New byte 1
E	00	Bit marker 0
F	80	Bit marker 1
8 0		
1		
2		
3		
4		
5		
6		
7		
8		

FIGURE 7-17

7.4.7 Test Program for MBIDY

MBIDY is really more powerful than is needed if we want zeroes to appear as blanks, or if we will always clear the display. However, it demonstrates some important ideas in logical functions, and will be useful in future work. To demonstrate its capability, try it initially with the symbols you have been using for 0 and 1 (blank and lower right segment), and do not clear the display. We will implement the NEXT, CLEAR, and ENTER functions only at this point, so that you can see two bytes (Old and New) and the bit marker. Any other key will simply address the second set of symbols. The flow chart and coding sheets (Figures 7-18 through 7-21) show a program which calls both MBIDY and BININ. Enter the code and run the program using the NEXT, CLEAR and ENTER commands as defined in 7.4.1.

There are a lot of instructions in these programs. Before running, be sure to verify very carefully that you have entered all of them correctly. If the program fails to execute properly, trace the program flow with breakpoints to try to find the cause of the problem.

Keep in mind that when using breakpoints and inspecting memory contents it is very easy to make a simple mistake that can have disastrous consequences. For example, if you are inspecting consecutive data addresses using NEXT and accidentally depress the STEP key, almost anything can happen. After a reasonable amount of time spent in fruitless debugging, always re-verify the contents of memory.

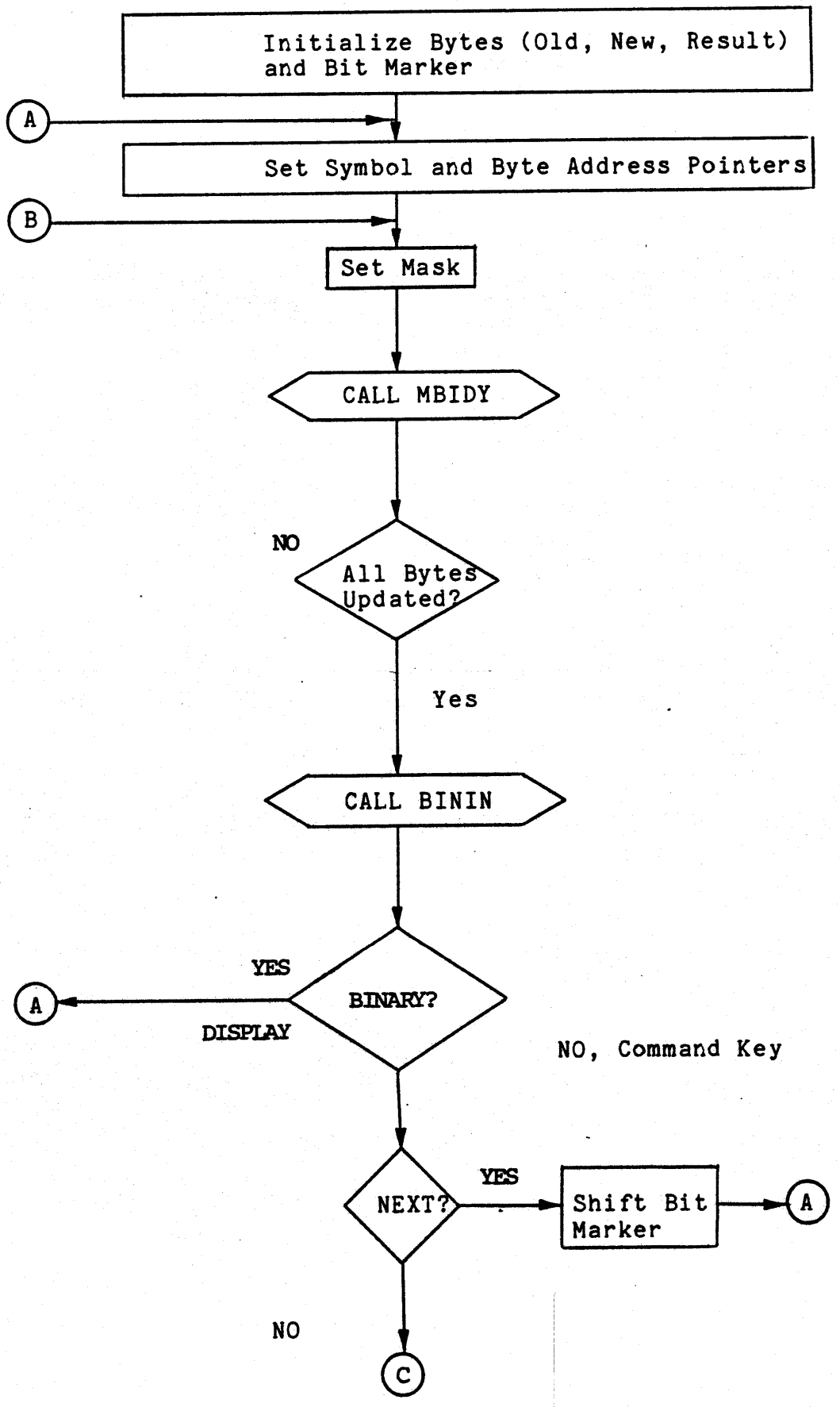

Initialize Bytes (Old, New, Result) and Bit Marker
A
Set Symbol and Byte Address Pointers
B
Set Mask
CALL MBIDY
NO
All Bytes Updated?
Yes
CALL BININ
YES
A
BINARY?
DISPLAY
NO, Command Key
NEXT?
YES
Shift Bit Marker
A
NO
C

FIGURE 7- 17

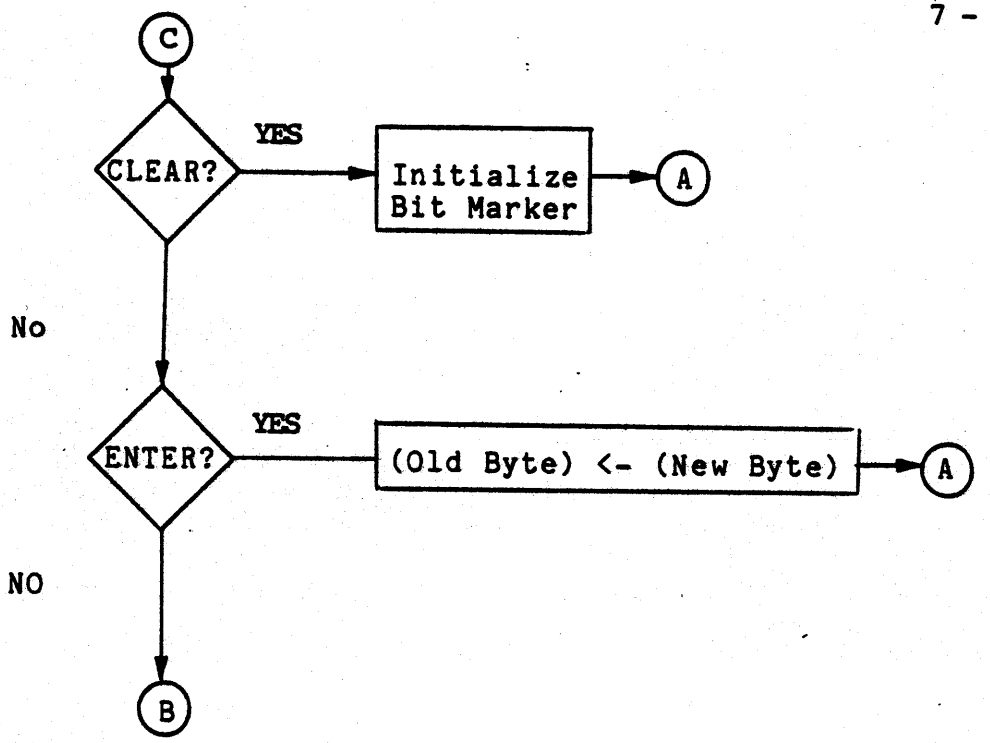


Figure 7-18 (Cont'd)

MAIN - Initialize, Display

ADDR	CODE				
8200	00	NOP			
01	00	NOP			
02	00	NOP			
03	21	LXI	H, 8304	Address bit marker
04	04			
05	83			
06	3E	MVI	A, 80	Set bit marker to left digit
07	80			
08	77	MOV	M, A	Store bit marker
09	AF	XRA	A	Clear A
0A	2D	DCR	L	Address next
0B	C2	JNZ	8208	byte; jumps when
0C	08			8300 is addressed
0D	82			
0E	00	NOP		
0F	00	NOP		
8210	01	LXI	B, 82FF	Display with
11	FF			input symbol
12	82			
13	21	LXI	H, 8304	Start with
14	04			bit marker
15	83			
16	AF	XRA	A	Set mask to clear
17	00	NOP		MVI A, FF for trial
18	CD	CALL	MBIDY	Display
19	A0			
1A	82			
1B	2D	DCR	L	Count down to 8300
1C	C2	JNZ	8218	Loop until all
1D	18			bytes displayed
1E	82			
1F	00	NOP		
8220				
21				
22				
23				
24				
25				
26				
27				
28				

FIGURE 7-19

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

MAIN - Call BININ —; Process Commands

ADDR		CODE														
8220	0	CD			C	A	L	L		B	I	N	I	N	Call for binary
	1	80													input or command
	2	82													
	3	DA			J	C			8	2	1	0			Jump if binary
	4	10													to display input
	5	82													
	6	21			L	X	I		H	,	8	3	0	4	Address bit marker
	7	04													
	8	83													
	9	FE			C	P	I		1	5					Test for NEXT
	A	15													
	B	C2			J	N	Z		8	2	3	4			
	C	34													
	D	82													
	E	7E			M	O	V		A	,	M				NEXT
	F	0F			R	R	C								Shift bit marker
8223	0	77			M	O	V		M	,	A				and store
	1	C3			J	M	P		8	2	1	0			Jump to display
	2	10													
	3	82													
	4	36			M	V	I		M	,	8	0			Set bit marker
	5	80													to left
	6	2D			D	C	R		L						Address new byte
	7	FE			C	P	I		1	7					Test for CLEAR
	8	17													
	9	C2			J	N	Z		8	2	4	1			
	A	41													
	B	82													
	C	36			M	V	I		M	,	0	0			CLEAR
	D	00													(8303) ← 0
	E	C3			J	M	P		8	2	1	0			
	F	10													
8240	0	82													
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

FIGURE 7-20

ADDR	CODE												
8 — 0													
8241	FE		CPI		13							Test for ENTER
2	13											
3	C2		JNZ		824C							
4	4C											
5	82											
6	7E		MOV		A,M							ENTER
7	2D		DCR		L							(8302) ← (8303)
8	77		MOV		M,A							
9	C3		JMP		8210							
A	10											
B	82											
C	01		LXI		B, 82F6							All other commands
D	F6											Address result
E	82											symbols - no bit mask
F	C3		JMP		8216							Jump to display
8250	16											with (HL) = 8303
1	82											
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 7-21

7.4.8 Final Program Implementation

Having tested the modules, we will implement the remaining command keys. You will have observed that determining the value of a command key by repeated use of

```
CPI   xx

JNZ   yyzz
```

is very inefficient. We will improve on that by placing a table of addresses in memory, using the value of the command key to address an entry in that table.

KEY	HEX VALUE	CONTROL FUNCTION
MEM	10	Address for AND procedure
REG	11	Address for OR procedure
ADDR	12	Address for COMPLEMENT procedure
STEP	13	Address for ENTER procedure
RUN	14	Address for STORE procedure
NEXT	15	Address for NEXT procedure
BRK	16	Address for XOR procedure
CLR	17	Address for CLEAR procedure

A complete address requires two bytes, but since the entire program is in memory page 8200 we can use just the low-order byte of the address in the table. Assume that we have the command key value in register A. The process could be:

```
LXI     H,82xx    Address of a table containing addresses of the
                  various procedures. (Dispatch Table)
ADD     L         Add value of command key
MOV     L,A       Put new low-order address byte in L
MOV     L,M       Move content of that address into L.
                  This is the low address of the procedure.
PCHL              Jump to the address for the procedure.
```

Since the smallest value for a command key is 10_{16}, we will start with an address 10_{16} less than the first table entry.

DISPATCH TABLE

ADDRESS	CONTENTS
823A	Address of AND procedure
823B	Address of OR procedure
823C	Address of COMPLEMENT procedure
etc.	

We will load H,L with 822A. If the value of the command key is 10, adding it to (H,L) will give 823A, the address which contains the

address of the AND procedure. If the value is 11, the computed address will be 823B, the pointer to the OR procedure. This type of table is called a dispatch table, as it dispatches the program to the correct processing module.

However, we often need to use H,L to transmit an argument (e.g. a data byte or address) to the function being called or accessed. This conflicts with the use of H,L for a jump, but there is an easy solution. Find the jump address, as indicated above, and push it into the stack with PUSH H. Then do the other preparations and use the return instruction (RET) to jump to the address you have pushed.

Now examine the definitions of the keys given below, and design procedures for each function. If you arrange the sequence of the procedures you will find that they have much in common, and one can simply feed into another. As an example, CLEAR and COMPLEMENT both place data into the memory address for the New Byte. If you preload H,L with the address of the New Byte (8303) the procedure could be:

```
CLEAR    MVI M,FF
COMP     MOV A,M
         CMA
         MOV M,A
```

When the program enters at CLEAR, it will exit with (M) = 0; when it

enters at COMP, it will exit with (M) = $\overline{(M)}$. Flow charts and coding sheets for the revised control program are presented in Figures 7-22 through 7-25. MBIDY and BININ will not require any modification. When you run the program do not forget all of the caveats expressed above!

Exercise all of the function keys thoroughly, to insure that your program is fully debugged. Above all, make certain that you understand the purpose of all of the instructions in each segment of the program.

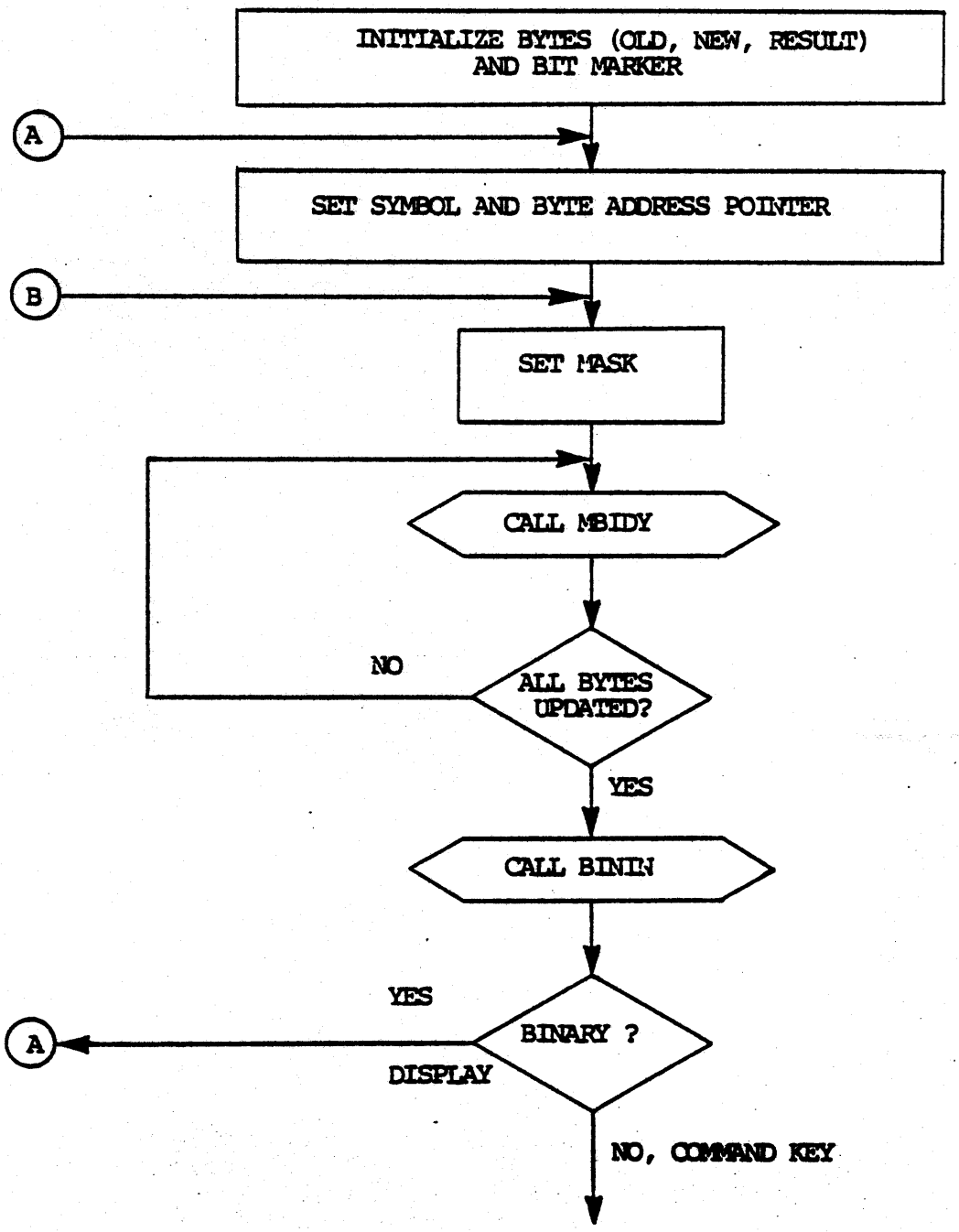

INITIALIZE BYTES (OLD, NEW, RESULT)
AND BIT MARKER
A
SET SYMBOL AND BYTE ADDRESS POINTER
B
SET MASK
CALL MBIDY
NO
ALL BYTES
UPDATED?
YES
CALL BININ
YES
A
BINARY ?
DISPLAY
NO, COMMAND KEY

FIGURE 7-22

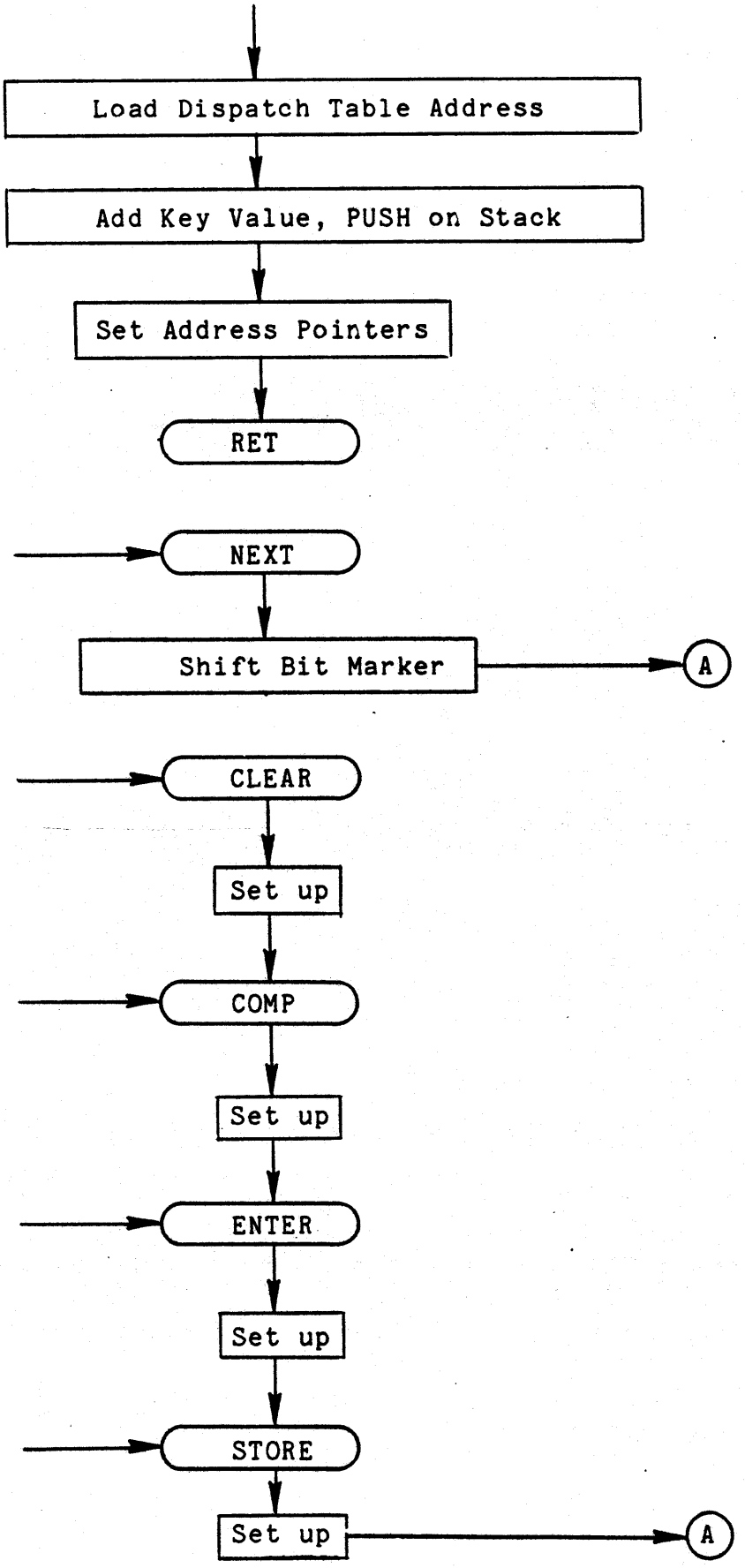


FIGURE 7-22 (Cont'd)

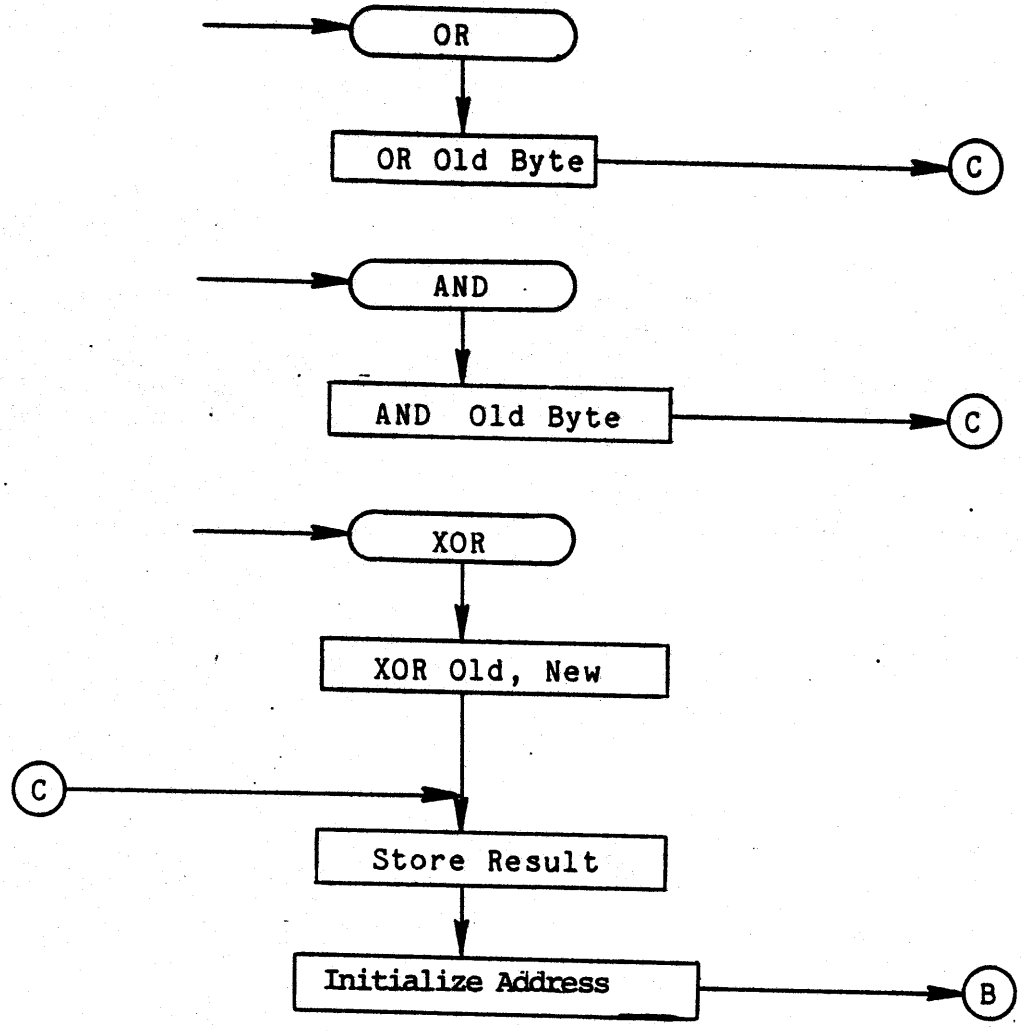


FIGURE 7-22 (Cont'd)

MAIN - Initialize, Display (Unchanged)

CODING SHEET — MICROCOMPUTER TRAINING SYSTEM — INTEGRATED COMPUTER SYSTEMS

ADDR	CODE					Comments
8200	C0		NOP			
01	00		NOP			
02	00		NOP			
03	21		LXI	H,	8304	
04	04					
05	83					
06	3E		MVI	A,	80	
07	80					
08	77		MOV	M,	A	
09	AF		XRA	A		
0A	2D		DCR	L		
0B	C2		JNZ	8205		
0C	05					
0D	82					
0E	00		NOP			
0F	00		NOP			
8210	01		LXI	B,	82FF	Display with input symbol
11	FF					
12	82					
13	21		LXI	H,	8304	Start with bit marker
14	04					
15	83					
16	AF		XRA	A		Set mask to clear
17	00		NOP			MVI A, FF for trial
18	CD		CALL	MBIDY		Display
19	A0					
1A	82					
1B	2D		DCR	L		Count down to 8300
1C	C2		JNZ	8218		
1D	18					
1E	82					
1F	00		NOP			
8220	CD		CALL	BININ		Call for binary input or command
21	70					
22	82					
23	DA		JC	8210		Jump if binary to display input
24	10					
25	82					
26						
27						
28						

FIGURE 7-23

MAIN - Generate Jump Address and Load Data

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

ADDR		CODE														
8222	6	21			L	X	I		H	,	8	2	2	A		Start of table
	7	2A														-1016
	8	82														
	9	85			A	D	D		L							Add key to table
	A	6F			M	O	V		L	,	A					address
	B	6E			M	O	V		L	,	M					(HL) ← Jump Addr
	C	E5			P	U	S	H		H						(ST) ← Jump Addr
	D	21			L	X	I		H	,	8	3	0	4		Address bit marker
	E	04														
	F	83														
8223	0	56			M	O	V		D	,	M					(D) ← Bit marker
	1	36			M	V	I		M	,	80					Set bit marker
	2	80														to left position
	3	2D			D	C	R		L							Address and
	4	7E			M	O	V		A	,	M					Load New Byte
	5	2D			D	C	R		L							
	6	2D			D	C	R		L							Address and load
	7	5E			M	O	V		E	,	M					previous result
	8	2C			I	N	R		L							Address Old Byte
	9	C9			R	E	T									Jump to process
	A	57														Address for AND
	B	53														Address for OR
	C	4C														Address for COMP
	D	4E														Address for ENTER
	E	4F														Address for STORE
	F	42														Address for NEXT
8224	0	5B														Address for XOR
	1	4A														Address for CLR
	2															
	3															
	4															
	5															
	6															
	7															
	8															

FIGURE 7-24

MAIN - Process Commands

ADDR		CODE														
~~8~~	0															
--	1															
824	2	7	A		M	O	V		A	,	D					NEXT
	3	0	F		R	R	C									Shift old bit
	4	2	C		I	N	R		L							marker right
	5	2	C		I	N	R		L							Address bit marker
	6	7	7		M	O	V		M	,	A					(8304) ← bit marker
	7	C	3		J	M	P		8	2	1	0				
	8	1	0													
	9	8	2													
	A	3	E		M	V	I		A	,	F	F				CLEAR - set up
	B	F	F													for COMP to set 0
	C	2	F		C	M	A									COMP (A) ← (Ā)
	D	2	C		I	N	R		L							Address new byte
	E	5	F		M	O	V		E	,	A					ENTER
	F	7	3		M	O	V		M	,	E					STORE
825	0	C	3		J	M	P		8	2	1	0				
	1	1	0													
	2	8	2													
	3	B	6		O	R	A		M							OR Old, New
	4	1	3		J	M	P		8	2	5	C				
	5	5	C													
	6	8	2													
	7	A	6		A	N	A		M							AND Old, New
	8	C	3		J	M	P		8	2	5	C				
	9	5	C													
	A	8	2													
	B	A	E		X	R	A		M							XOR Old, New
	C	2	D		D	C	R		L							Address Result
	D	7	7		M	O	V		M	,	A					Store Result
	E	0	1		L	X	I		B	,	8	2	F	5		Address Result
	F	F	5													Symbols, excluding
826	0	8	2													Bit Marker
	1	2	1		L	X	I		H	,	8	3	0	3		Address new byte
	2	0	3													
	3	8	3													
	4	C	3		J	M	P		8	2	1	6				Go display
	5	1	6													
	6	8	2													
	7															
	8															

FIGURE 7-25

7.5 SUMMARY

The code presented in this text is only one of many possible solutions to the original problem. After studying it to learn more about various ways of manipulating the data, you may wish to program the exercise by yourself from scratch. Some solutions will be more inefficient, some will be more elegant. You can challenge yourself by counting the number of memory locations used in our version, then trying to make yours more compact.

This chapter has introduced shift commands and logical functions. There are many variations of logical functions, of course, since they can use registers as sources. We have not yet encountered one accumulator command, DAA, one carry command, CMC. These will be used in the arithmetic sections of chapter 10.

In addition to using logical functions you have had practice using the stack, with PUSH, POP and XTHL as well as CALL and RET. You have calculated an address (as you did in the sensor correction exercise) and used it to find another address. These are all tools that are used constantly in program design. In programming with higher level languages (Fortran, for instance) all of this is hidden from you.

MICROCOMPUTER TRAINING WORKBOOK

CHAPTER 8

INPUT/OUTPUT TECHNIQUES

8. INPUT/OUTPUT TECHNIQUES

Various techniques and peripheral devices may be used with the 8080 to provide input and output capabilities. This chapter describes the common methods of implementing I/O and provides exercises in the use of those that are readily carried out with the MTS.

The techniques differ from each other in three major respects: how the input or output device is addressed; what event initiates the transfer of information; and what form the data are in. (The latter will be treated in Chapter 9).

Addressing

- Isolated Input/Output
- Memory Mapped Input/Output
- Direct Memory Access

Initiation

- Programmed Input/Output
- Interrupt Driven Input/Output
- Timed Input/Output
- Repetitive Direct Memory Access

The MTS includes facilities for all of these in one form or another, so you can learn each of the processes. For some, however, you must add external hardware.

8.1 ISOLATED INPUT/OUTPUT

The address and data busses are used to address input and output devices and transfer data between them and the CPU. The control bus from the 8228 controller includes I/O Read and I/O Write commands in addition to the Memory Read and Memory Write commands. It is the use of these command signals, and the instructions that generate them, that isolate I/O usage from memory usage of the busses.

8.1.1 I/O Ports

Any device with suitable electrical characteristics can be attached to the busses. In general such devices should have high impedance inputs from the bus and tri-state outputs to drive the bus. Intel,' NEC, and others provide the 8212 Input/Output Port for this purpose. The MTS includes one in the LED display circuit. A functional description is given in Figure 8-1; more detail is provided in the Intel 8080 Microcomputer System User's Manual. The principal features are low leakage currents of the inputs and outputs when the device is not selected, data latches, and control gating.

SCHOTTKY BIPOLAR 8212

Functional Description

Data Latch

The 8 flip-flops that make up the data latch are of a "D" type design. The output (Q) of the flip-flop will follow the data input (D) while the clock input (C) is high. Latching will occur when the clock (C) returns low.

The data latch is cleared by an asynchronous reset input ($\overline{CLR}$). (Note: Clock (C) Overides Reset ($\overline{CLR}$).)

Output Buffer

The outputs of the data latch (Q) are connected to 3-state, non-inverting output buffers. These buffers have a common control line (EN); this control line either enables the buffer to transmit the data from the outputs of the data latch (Q) or disables the buffer, forcing the output into a high impedance state. (3-state)

This high-impedance state allows the designer to connect the 8212 directly onto the microprocessor bi-directional data bus.

Control Logic

The 8212 has control inputs $\overline{DS1}$, DS2, MD and STB. These inputs are used to control device selection, data latching, output buffer state and service request flip-flop.

$\overline{DS1}$, DS2 (Device Select)

These 2 inputs are used for device selection. When $\overline{DS1}$ is low and DS2 is high ($\overline{DS1} \cdot DS2$) the device is selected. In the selected state the output buffer is enabled and the service request flip-flop (SR) is asynchronously set.

MD (Mode)

This input is used to control the state of the output buffer and to determine the source of the clock input (C) to the data latch.

When MD is high (output mode) the output buffers are enabled and the source of clock (C) to the data latch is from the device selection logic ($\overline{DS1} \cdot DS2$).

When MD is low (input mode) the output buffer state is determined by the device selection logic ($\overline{DS1} \cdot DS2$) and the source of clock (C) to the data latch is the STB (Strobe) input.

STB (Strobe)

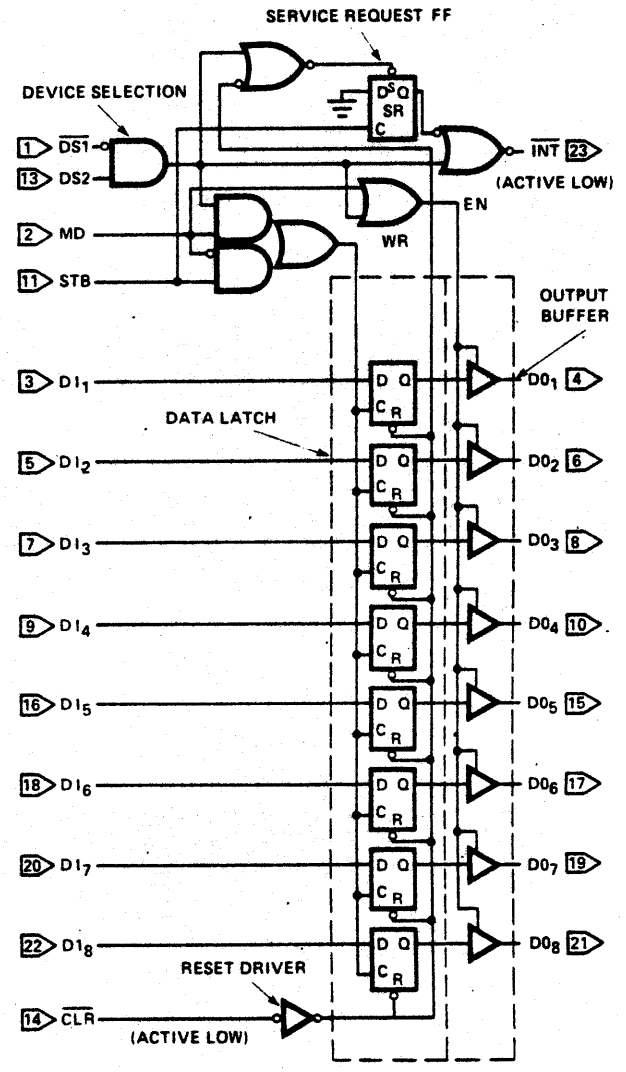
This input is used as the clock (C) to the data latch for the input mode MD = 0) and to synchronously reset the service request flip-flop (SR).

Note that the SR flip-flop is negative edge triggered.

Service Request Flip-Flop

The (SR) flip-flop is used to generate and control interrupts in microcomputer systems. It is asynchronously set by the $\overline{CLR}$ input (active low). When the (SR) flip-flop is set it is in the non-interrupting state.

The output of the (SR) flip-flop (Q) is connected to an inverting input of a "NOR" gate. The other input to the "NOR" gate is non-inverting and is connected to the device selection logic ($\overline{DS1} \cdot DS2$). The output of the "NOR" gate ($\overline{INT}$) is active low (interrupting state) for connection to active low input priority generating circuits.



STB	MD	$(\overline{DS_1} \cdot DS_2)$	DATA OUT EQUALS
0	0	0	3-STATE
1	0	0	3-STATE
0	1	0	DATA LATCH
1	1	0	DATA LATCH
0	0	1	DATA LATCH
1	0	1	DATA IN
0	1	1	DATA IN
1	1	1	DATA IN

$\overline{CLR}$ – RESETS DATA LATCH
SETS SR FLIP-FLOP
(NO EFFECT ON OUTPUT BUFFER)

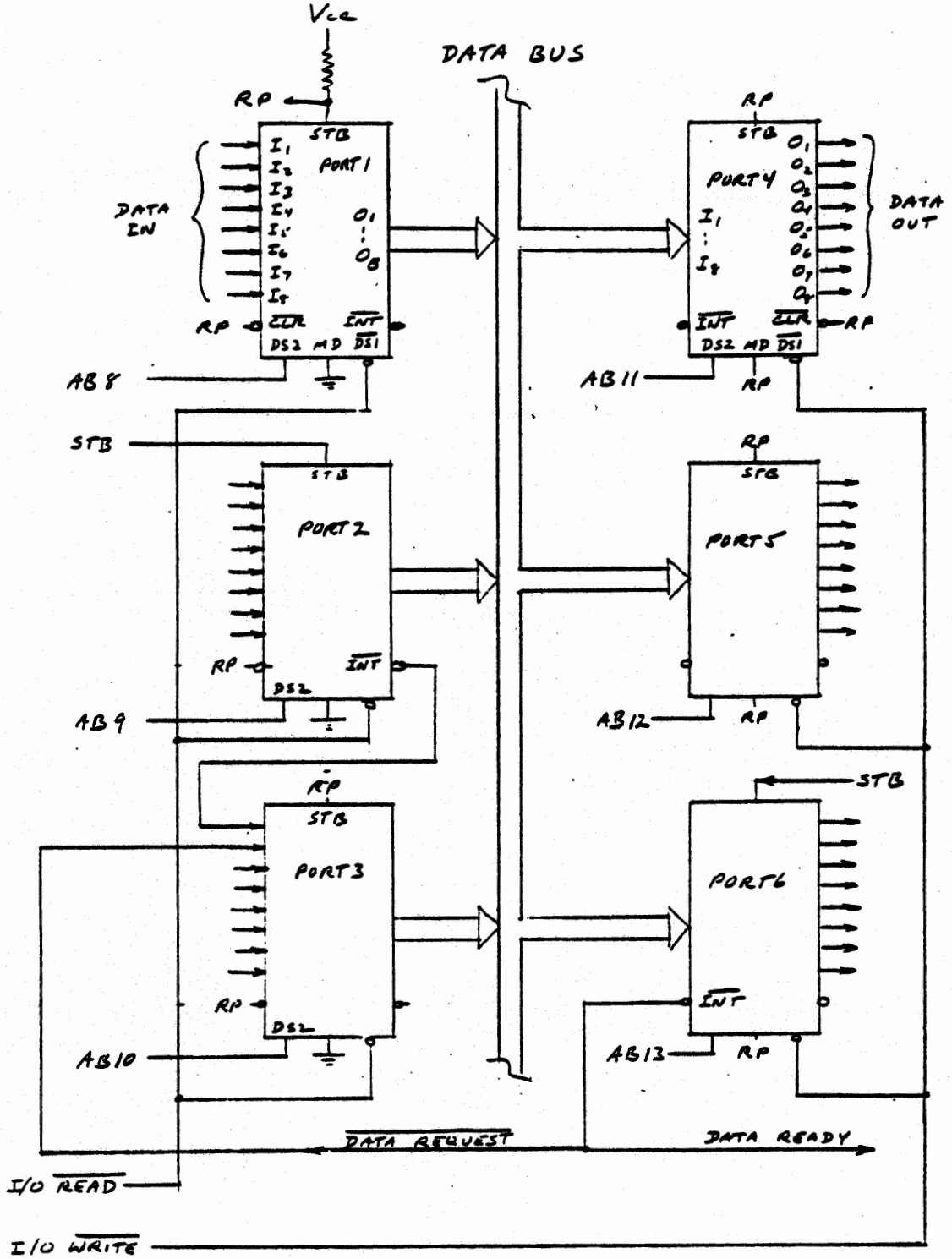
CLR	$(\overline{DS_1} \cdot DS_2)$	STB	*SR	INT
0	0	0	1	1
0	1	0	1	0
1	1	↘	0	0
1	1	0	1	0
1	0	0	1	1
1	1	↘	1	0

*INTERNAL SR FLIP-FLOP

FIGURE 8-1

A suitable arrangement for using several 8212's as input and output ports is shown in Figure 8-2. Each is selected by a single bit of the high address bus to the non-inverting select input DS2, so no additional decoding is necessary. The input ports are enabled by the I/O $\overline{\text{READ}}$ bar command and the outputs by the I/O $\overline{\text{WRITE}}$ command, to the inverting select input $\overline{\text{DS1}}$. Output data from the CPU enters an output port when the device is selected by $\overline{\text{DS1}}$ and DS2, and latched by the 8212 when it is de-selected; the 8212 outputs are always enabled. This behavior is set by the MODE input being pulled high.

The STROBE input is unused for output ports 4 and 5. Output port 6 receives a strobe from some external hardware to indicate a need for new data. With the MODE input high this has no effect on the data outputs, but it sets the INT output low, indicating a need for service. The diagram shows that signal being input to the processor through input port 3. When the CPU loads new data to port 6 $\overline{\text{INT}}$ will be set high again to indicate that the requested data are ready.



ARRAY OF INPUT /OUTPUT PORTS

FIGURE 8-2

Input ports 1 and 3 are direct paths from their inputs onto the data bus when they are selected, because their strobe inputs are pulled high. This makes them suitable for stable data. Input port 2 is designed to receive a fleeting input, which may be gone before the processor can service it. An external strobe is provided to latch the data in the 8212 and set $\overline{\text{INT}}$ low, requesting service from the CPU when it reads port 3.

The CPU accesses these ports with the commands:

DB	IN	Input from port
xx	port address	to register A
		High address <- (byte 2)
		Low address <- (byte 2)
		(A) <- (Data bus)
		No flags are affected
D3	OUT	Output to port
xx	port address	from register A
		High Address <- (byte 2)
		Low Address <- (byte 2)
		(A) <- (Data Bus)
		No flags are affected

These are the only instructions for isolated input and output. They alone create the I/O Read and I/O Write commands to the ports.

Note that the port address is only one byte, not two. In response to one of these instructions the CPU places that byte on the low eight bits of the address bus, and duplicates it on the high eight bits.

This duplication permits the I/O devices to be selected from the high address bus, which is typically less heavily loaded by memory devices than the low address bus.

The addressing shown here, where a single bit on the address bus selects a device, is called linear select. It is economical of hardware but restricts the system size. Port addresses for the devices in figure 8-2 are:

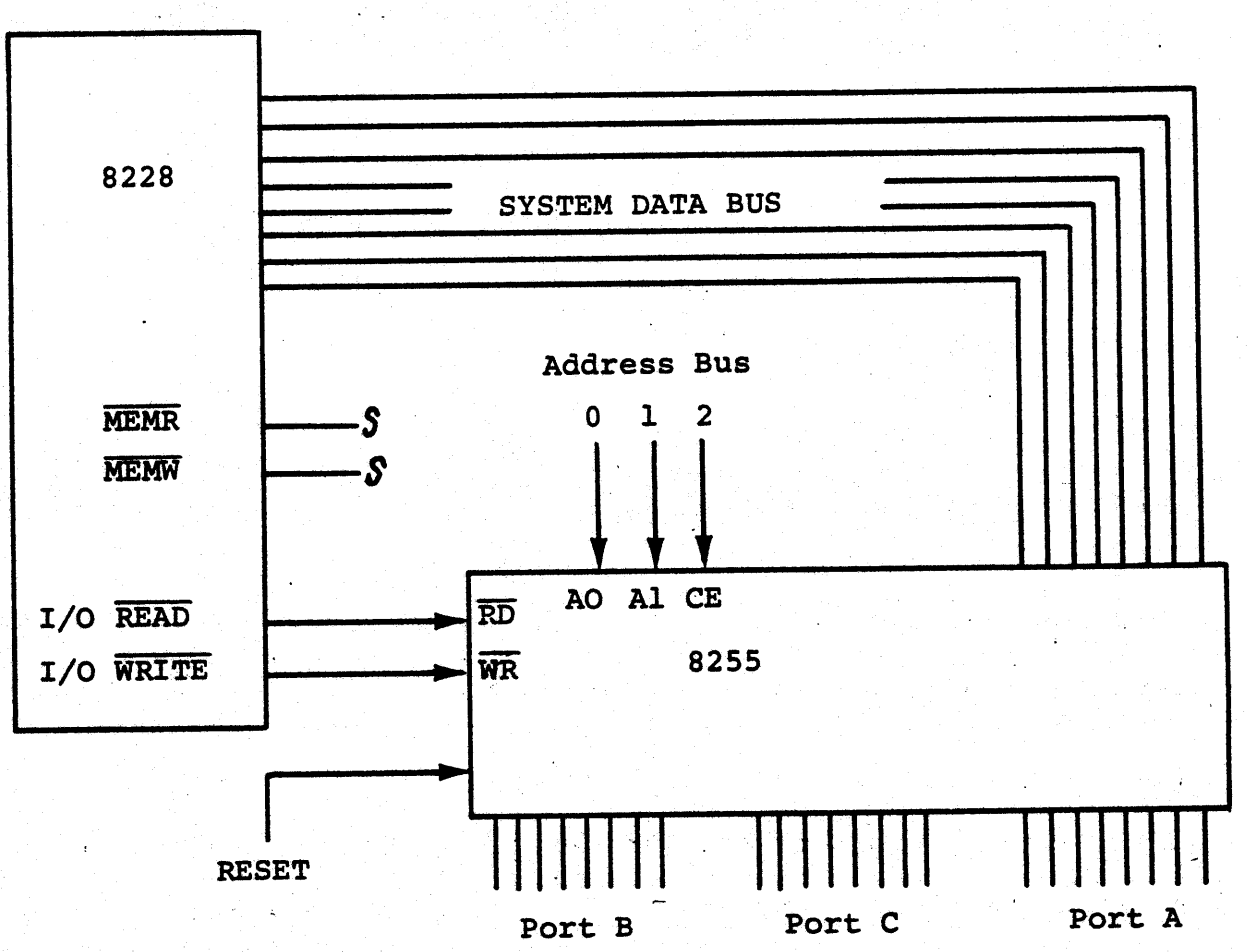
Input port 1	01	00000001
Input port 2	02	00000010
Input port 3	03	00000100
Output port 4	08	00001000
Output port 5	10	00010000
Output port 6	20	00100000

For a larger system some decoding of the address is necessary.

8.1.2 MTS Input/Output

The MTS includes an 8255 Programmable Peripheral Interface Adaptor (Figure 8-3). It has 24 external connections, which can be programmed as inputs or outputs in various combinations. It connects internally to the system data bus and the three low bits of the address bus, and to the I/O Read and I/O Write commands from the 8228. When the 8255 is selected by a low signal on AB2 (i.e. any port address of the form xxxxx0xx), the 8255 will respond to the I/O Read or I/O Write commands. These are generated by the 8228 when the CPU executes one of the instructions:

DB	IN	Input to register A
xx	port address	(A) <- (Port)
D3	OUT	Output from register A
xx	port address	(Port) <- (A)



ISOLATED INPUT/OUTPUT

WITH THE 8255

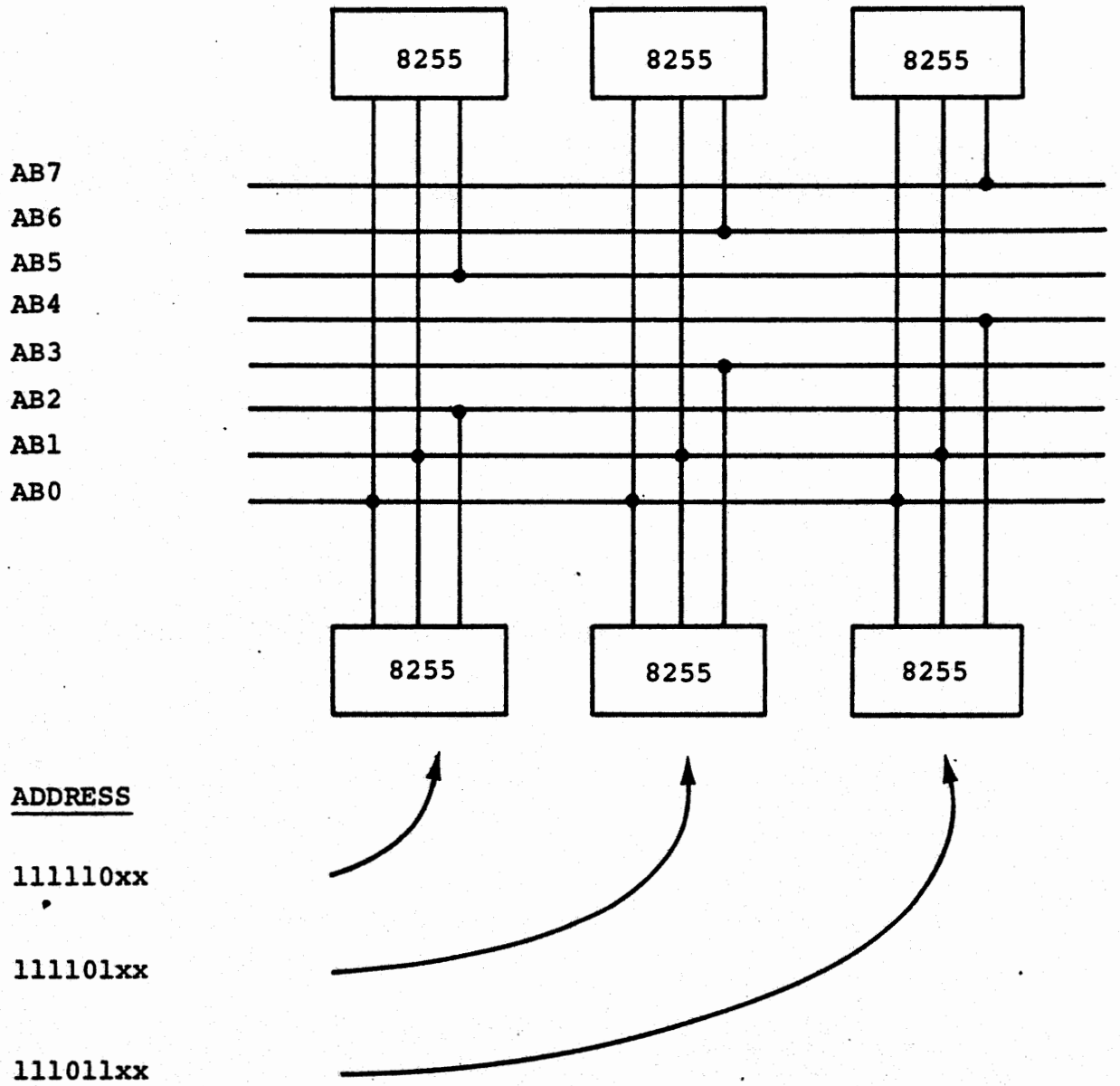
Figure 8 - 3

The 8255 is selected if bit 2 is 0. Bits 0 and 1 select one of the three eight-bit ports. If the OUT instruction is used the 8080 places the content of the A register on the data bus and the 8255 copies it into the selected port, provided that is programmed for output. If the IN instruction is used the 8255 places its present input or the content of its data latch onto the data bus, and the 8080 copies the data into register A.

The port address can theoretically address 256 input or output. devices. Each 8255 occupies four address; in the MTS the address is not fully decoded. The coding of the address is:

00-F8	xxxx x000	8255 Port A
01-F9	xxxx x001	8255 Port B
02-FA	xxxx x010	8255 Port C
03-FB	xxxx x011	8255 Control
	xxxx x1xx	8255 Not Selected

Although 00, 01, 02 and 03 or any other bytes with the same three low bits will select ports, it is often desirable to hold the 'don't care' bits high if any system expansion is planned. Up to six 8255's can then be selected with no additional decoding, as shown in Figure 8-4.



MULTIPLE I/O PORTS ON ADDRESS BUS

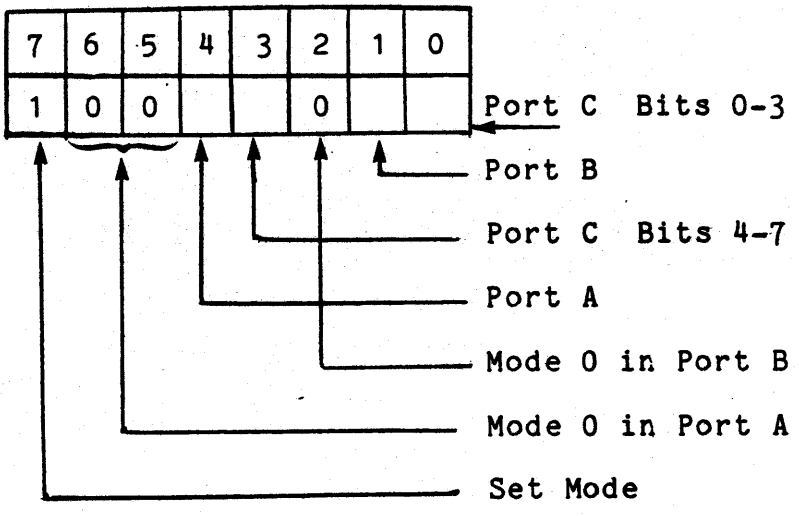
Figure 8-4

In addition to the three external ports, the 8255 has a 'control port' addressed by 11 in the low bits of the address. This is used to program the external ports for input or output, and to select the mode of operation. The monitor programs the 8255 with the instructions:

```
3E   MVI  A,92      Write 10010010
92                  to the control port.
D3   OUT  CNTPT
FB
```

This sets ports A and B for input and port C for output. Ports A and B are each eight bit ports and can be programmed independently of each other. In the basic mode of operation (mode 0) port C is divided into two four-bit ports which can be independently programed for input or output. Thus 16 different combinations of input and output assignments are available in mode 0.

The bits in the control byte are defined as follows:



Notes	Control Byte		Port A	Port C	Port C	Port B
	Hex	Binary		Bits 4-7	Bits 0-3	
(3)	80	1000 0000	Out	Out	Out	Out
(3)	81	1000 0001	Out	Out	In	Out
(3)	82	1000 0010	Out	Out	Out	In
(3)	83	1000 0011	Out	Out	In	In
	88	1000 1000	Out	In	Out	Out
	89	1000 1001	Out	In	In	Out
	8A	1000 1010	Out	In	Out	In
	8B	1000 1011	Out	In	In	In
(1)	90	1001 0000	In	Out	Out	Out
(1)	91	1001 0001	In	Out	In	Out
(1,2)	92	1001 0010	In	Out	Out	In
(1)	93	1001 0011	In	Out	In	In
	98	1001 1000	In	In	Out	Out
	99	1001 1001	In	In	In	Out
	9A	1001 1010	In	In	Out	In
	9B	1001 1011	In	In	In	In

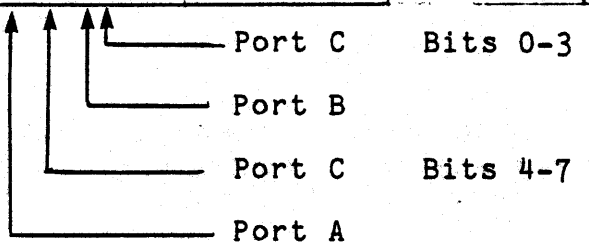


Table 8-1 8255 Mode 0 Combinations

Notes to Table 8-1:

(1) Only the four combinations marked are suitable for use with the MTS if the keyboard is to be used.

(2) This combination is set by the monitor whenever it controls the keyboard and display.

(3) Port A and Port C (bits 4-7) should not both be programmed for output, since the keyboard would then short them together.

The 8255 provides a second mode of operation for port A or port B or both, in which certain bits of port C are used for 'handshaking' with external devices. For input in this mode the external device places its data at the input port and gives a strobe pulse to one bit of port C. This stores the data in an eight bit latch associated with the eight bit input port, and generates other status bits in port C which are accessible both to the CPU (by reading port C) and to the external world at the port C outputs. This allows transient signals to be input and read subsequently by the program at its convenience. For details the student is referred to the Intel 8080 Microcomputer System User's Manual (September 1975 page 5-113).

In the basic input mode which we have been discussing, the data latches follow their inputs whenever the port is addressed. If a port is programmed for input the IN instruction will read the current state of the input. When a port is programmed for output the data latch is loaded by an OUT instruction, and the data remain stable until the next OUT. These data can be read back by the processor; IN will always read the content of the data latch. This does not apply to the control port, for which the IN instruction is not effective.

A third mode of operation is available for port A only, in which it is both an input and an output port suitable for connection to a bi-directional data bus.

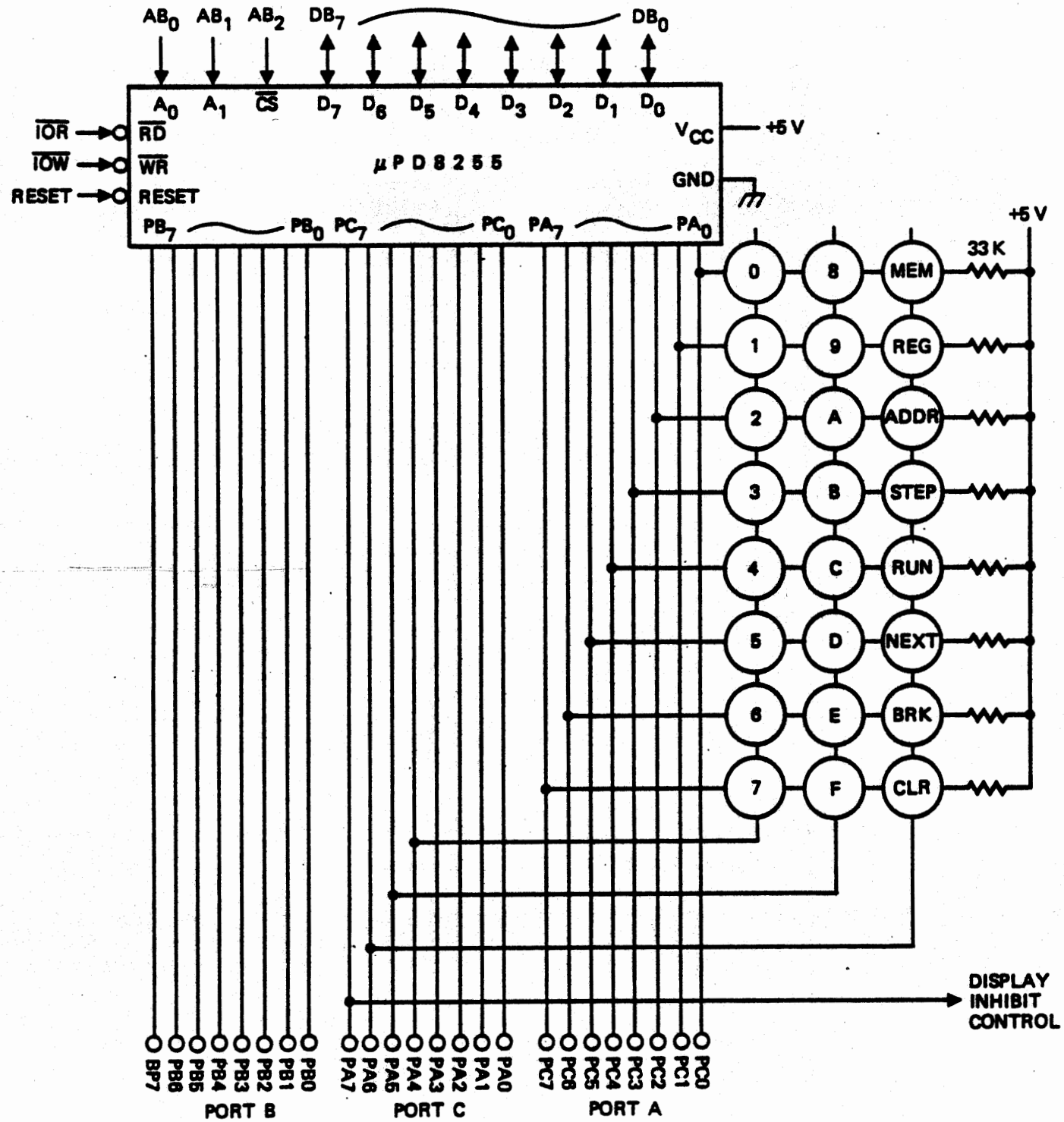
8.1.3 Keyboard Input

To acquire familiarity with the 8255 we will develop a keyboard input program. You have been using the MTS monitor subroutines for this purpose. The subroutines to be developed here will be different in design.

Figure 8-5 shows the connections between the 8255 and the keyboard. The keyboard is a 3 x 8 matrix. Reset is not in the matrix but is directly connected to the reset input. The other keys form three columns: keys 0 through 7; 8 through F; and the command keys. Each row has three keys and a pullup resistor and is connected to an input bit of port A. If no key in the row is pressed that bit of port A will be 1 because of the resistor. If a key is pressed the input bit of port A is connected through the key to one of three output bits of port C. If that output is high the input to port A will still be 1, but if it is low the input will be 0. Thus by setting one bit of port C low and reading port A we can tell which, if any, key is pressed. We can make a quick test to see whether any key in the keyboard is pressed if we set all three outputs (C4, C5 and C6) low and read port A; if the result is 1111 1111 no key is pressed.

There may be a circumstance where we are interested only in a particular key. This can be tested by setting the corresponding column low, reading the input, and masking to exclude all keys except the desired one.

Subroutine KYIN is specified to permit any of these functions.



8255 and Key Input Scanning Circuit

FIGURE 8-5

8.1.4 Subroutine KYIN

Function:

Test the keyboard for any desired key or keys being pressed. Set one or more of output bits C4, C5, C6 low (without affecting any other bits of port C) according to a parameter passed in the call. Read the keyboard and mask with another byte passed as a parameter. Return with the zero flag set if no desired key is pressed; otherwise with zero cleared and the binary input data in register C. Restore the column select bit (C4, C5, or C6) to 1 before returning.

Two alternate entries provide for setting the input parameters to test for any key, and for programming the 8255.

Call

```
CD    CALL KPRG
40    Program the 8255
82    and continue to KTST

CD    CALL KTST
44    Test for any key
82

CD    CALL KYIN
48    Test for specified key
82    or keys in specified
      column or columns
```

Inputs

KPRG: None
KTST: None
KYIN:
a) Key column select in register B
   contains 0 for each desired column.
   Bits 0, 1, 2, 3 and 7 must be 1
b) Key mask in register C
   contains 1 for each desired key

Outputs

Zero flag set if no desired key.
Zero flag clear if desired key is pressed
Keyboard input (00 if no keys) in register C.
Key column select in register B is preserved
(8F for KTST).

Registers

A, B, C, D are used.

Constraints

If KPRG is called, 8255 will be programmed as follows:

C0 - C3	Output
Port B	Output, mode 0
C4 - C7	Output
Port A	Input, mode 0

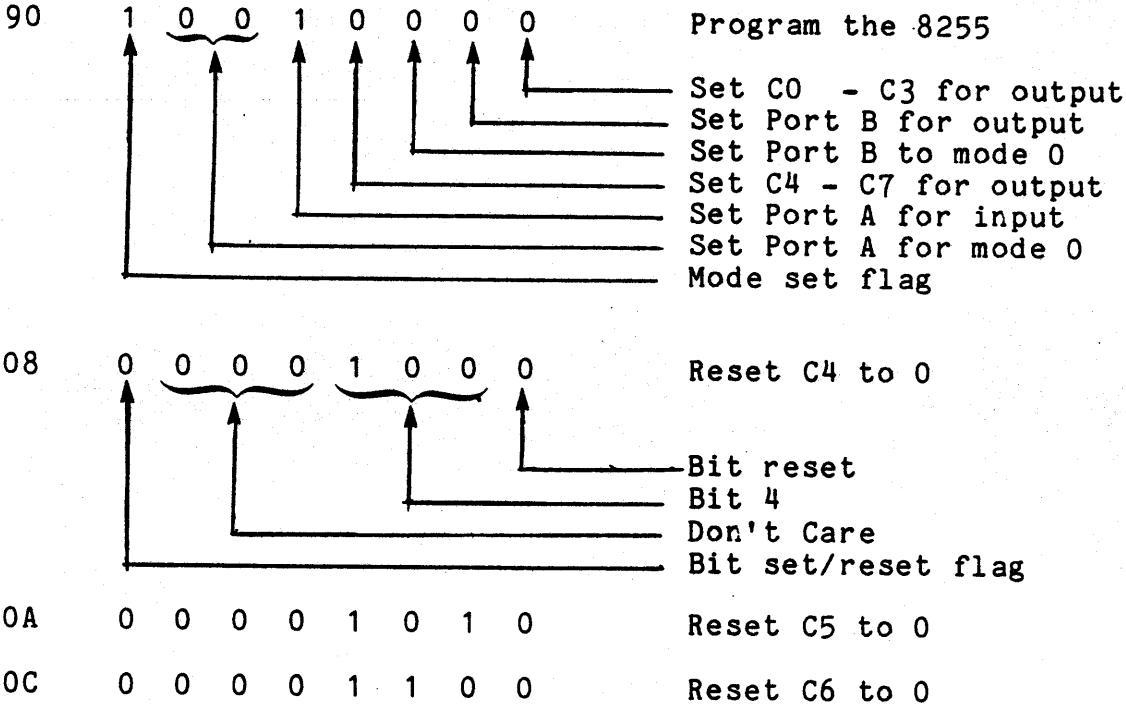
Outputs of all ports are cleared by KPRG.
If KTST or KYIN is called, C4 - C7 and port A must be programmed as shown above.

We have discussed programming the 8255 by writing to the control port. There is another function in the control port: you can set or reset any individual bit of port C. This is done by writing a byte from register A to the control port:

```
3E   MVI  A        (A) <- Selected command
xx
D3   OUT  CNTPT
03
```

This sequence applies to both programming the 8255 and setting bits in port C. The command bytes are distinguished by the high order bit as shown below:

Command Bytes to Control Port



This provides a technique for altering one output bit without changing others. Another technique is to read the content of the output data latch:

```
DB                              IN PORTC
02
```

will read the data latch of the port into register A even though the port is programmed for output. Then you can use "ORA r" or "ORI data" to set desired bits to 1; "ANA r" or "ANI data" to set desired bits to 0. For instance, to set C7, C6 and C5 to 1 and C4 to zero, use this program segment:

```
06    MVI  B,11101111     Set up for C4 low
EF
DB    IN   PORTC          Read old output data
02
F6    ORI  11110000       Set C7, C6, C5, C4 to 1
F0
A0    ANA  B              Set selected bit to 0
D3    OUT  PORTC          Write to port C
02
```

Wherever several bits must be controlled this takes less program space than the individual bit set and reset instructions. Caution: Reading from an output port is not included in the manufacturer's specification for the 8255. That it will work is predictable from the design of the 8212, and proven by experiment with the 8255, but conceivably a future

redesign of the 8255 might not allow it.

Programs that write to the display or to port C, or that program the 8255, are always difficult to debug because whenever the monitor actuates the keyboard and display it destroys whatever your program has done. Suggestion: at each point in the program when an output is written, first store the data in memory. When you read an input, immediately store the data. Being able to recover the data at a subsequent breakpoint makes debugging immensely easier. The STA instructions can be deleted when the program works.

Keyboard reading introduces another problem: at return from the monitor the keys are always released. You can simulate a key input by placing a breakpoint just after the IN instruction. When it is executed you can load some value other than FF in the A register to make sure that the rest of your program functions correctly.

If any peculiar condition arises while you have a key pressed, you can press RST while the other key is held down. Although the contents of your program counter, stack, and display are lost, the registers and memory locations are preserved.

Draw the flow chart and write the program for KYIN. Test it initially with a very simple calling program. To ease debugging, call KYIN, not KTST. The monitor leaves the 8255 programmed with port C for output and port A for mode 0 input.

```
LXI     B,8FFF    Enable all keys
CALL    KYIN      Read keys
JZ                Repeat until
RST 4             key is found
JMP               Then call monitor
```

This will return to the monitor as soon as you press a key. Then you can look in the storage locations where you have saved the inputs and outputs to see if they are what you expect.

When you call the monitor with a key pressed, hold the key down until you see what you have. If you are displaying PC and the instruction, a numeric key will give the Err display as soon as you release it. If you are displaying a register, a numeric key will be entered into the register when you release it. You can retrieve the old value by pressing CLR, however.

Figures 8-6 to 8-9 provide a flow chart, test program, and two versions of KYIN, one with debugging code included.

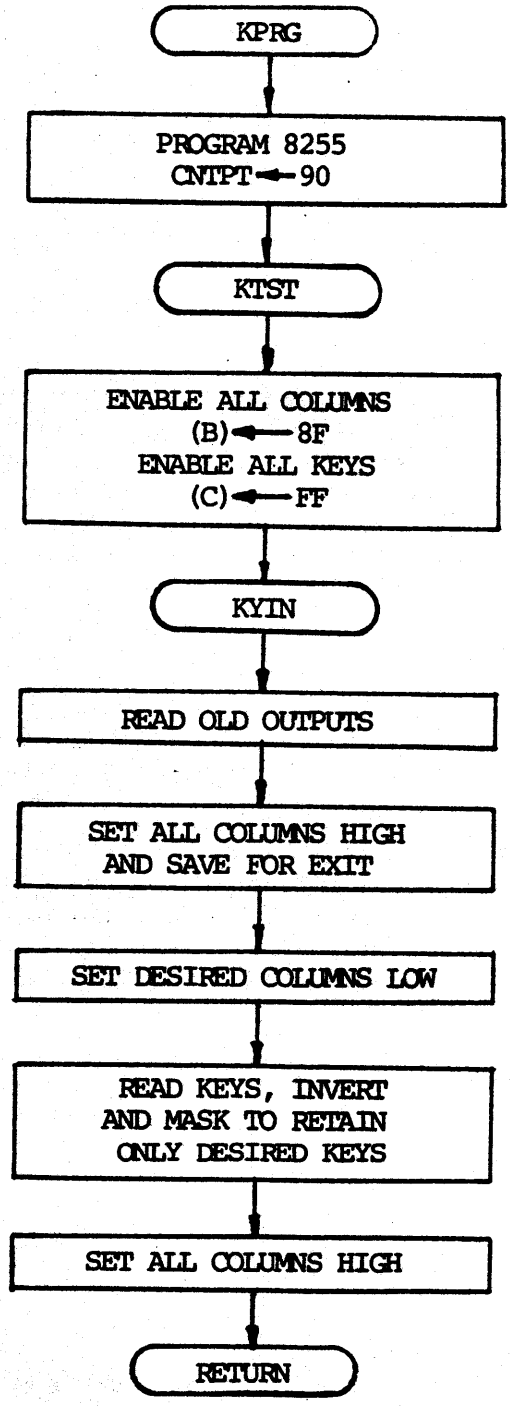

KPRG
PROGRAM 8255
CNTPT←90
KTST
ENABLE ALL COLUMNS
(B)←8F
ENABLE ALL KEYS
(C)←FF
KYIN
READ OLD OUTPUTS
SET ALL COLUMNS HIGH
AND SAVE FOR EXIT
SET DESIRED COLUMNS LOW
READ KEYS, INVERT
AND MASK TO RETAIN
ONLY DESIRED KEYS
SET ALL COLUMNS HIGH
RETURN

FIGURE 8-6

First Test Program

INTEGRATED COMPUTER SYSTEMS MICROCOMPUTER TRAINING SYSTEM CODING SHEET

ADDR	CODE												
8200	01		L	X	I		B	→	8	F	F	F	
01	FF												
02	8F												
03	CD		C	A	L	L		K	Y	I	N		
04	48												
05	82												
06	CA		J	Z			8	2	0	0			
07	00												
08	82												
09	E7		R	S	T		4						
0A	C3		J	M	P		5	2	C	C			
0B	00												
0C	82												
0D													
0E													
0F													
8210													
11													
12													
13													
14													
15													
16													
17													
18													
19													
1A													
1B													
1C					-								
1D													
1E													
1F													
8220													
21													
22													
23													
24													
25													
26													
27													
28													

FIGURE 8-7

KPRG, KTST, KYIN with debugging features

ADDR		CODE																		
8240	0	3E			M	V	I		A	,	9	0			KPRG
	1	90													Program 8255
	2	D3			O	U	T		C	N	T	P	T		Ports B, C output
	3	03													Port A Input
	4	06			M	V	I		B	,	8	F			KTST
	5	8F													(B) ← All columns
	6	0E			M	V	I		C	,	F	F			(C) ← All Keys
	7	FF													
	8	DB			I	N		P	O	R	T	C			KYIN
	9	02													Read old outputs
	A	32			S	T	A		8	3	0	0			Save for debugging
	B	00													
	C	83													
	D	F6			O	R	I		F	O					Set all columns
	E	FO													high and save
	F	57			M	O	V		D	,	A				in register D
8250	0	A0			A	N	A		B						Set selected
	1	32			S	T	A		8	3	0	1			columns low
	2	01													and save for
	3	83													debugging
	4	D3			O	U	T		P	O	R	T	C		
	5	02													
	6	DB			I	N			P	O	R	T	A		Read keys
	7	00													
	8	32			S	T	A		8	3	0	2			Save for debugging
	9	02													I
	A	83													
	B	2F			C	M	A								Invert and mask
	C	A1			A	N	A		C						so desired key
	D	4F			M	O	V		C	,	A				=1 if pressed
	E	7A			M	O	V		A	,	D				Set all columns
	F	D3			O	U	T		P	O	R	T	C		high
8260	0	02													
	1	C9			R	E	T								
	2														
	3														
	4														
	5														
	6														
	7														
	8														

FIGURE 8-8

KPRG, KTST, KYIN with debugging removed

ADDR	CODE		Instruction	Comment
8240	3E		MVI A, 90	KPRG
1	90			Program 8255
2	D3		OUT CNTPT	Ports B,C Out
3	03			Port A In
4	06		MVI B, 8F	KTST
5	8F			(B) ← All Columns
6	0E		MVI C, FF	(C) ← All Keys
7	FF			
8	DB		IN PORTC	KYIN
9	02			Read old outputs
A	F6		ORI F0	Set all columns
B	F0			high and save
C	57		MOV D, A	in reg D
D	A0		ANA B	Set selected
E	D3		OUT PORTC	columns low
F	02			
8250	DB		IN PORTA	Read keys and
1	00			invert so key
2	2F		CMA	pressed = 1
3	A1		ANA C	Mask unwanted
4	4F		MOV C, A	Keys and store
5	7A		MOV A, D	Set all columns
6	D3		OUT PORTC	high
7	02			
8	C9		RET	
9				
A				
B				
C				
D				
E				
F				
8 0				
1				
2				
3				
4				
5				
6				
7				
8				

FIGURE 8-9

8.1.5 Using KYIN

Now we can make more interesting use of KYIN. The following program takes any key from 0 - 7 (which appears as a single bit = 1 in register C) and OR's it into a display location at the corresponding display segment bit. By pressing succesive keys, you may 'paint' a character. It also tests for CLR and NXT, either clearing the presently addressed display location or moving to the next location. If children are accessible they will enjoy writing their names in the display - if their names lack K, M, Q, W and X! This demonstrates one requirement of keyboard input: you must distinguish between a key being held down for a long time versus repetitive depressions of the same key. The numeric keys and CLR don't care in this program, but if you do not test for release of NXT it will step across the display many times before you can let go of the key.

Keyboard input programs normally provide for 'debouncing'. Many electrical switches do not change from closed to open perfectly, but 'bounce' between the two states for some milliseconds.. This can occur in the switch contact itself, or it can be created by a TTL circuit sensing the contact. To avoid seeing a single closure as multiple operations there is usually a time delay circuit or program used to require that the key be open for 10 to 30 milliseconds before it is accepted again. Such a provision is included in the MTS monitor subroutine GETKY, even though the MTS keys seem to be completely free of bounce.

This program design illustrates restriction of the MTS: you must not alternately disable and enable the display in a loop that is fast compared to the DMA timer. This is discussed later in this chapter. To avoid it, do not call KPRG except in response to some human action or after a substantial delay. Try writing the flow chart and program, then check Figures 8-10 through 8-12.

KEYBOARD DISPLAY PROGRAM

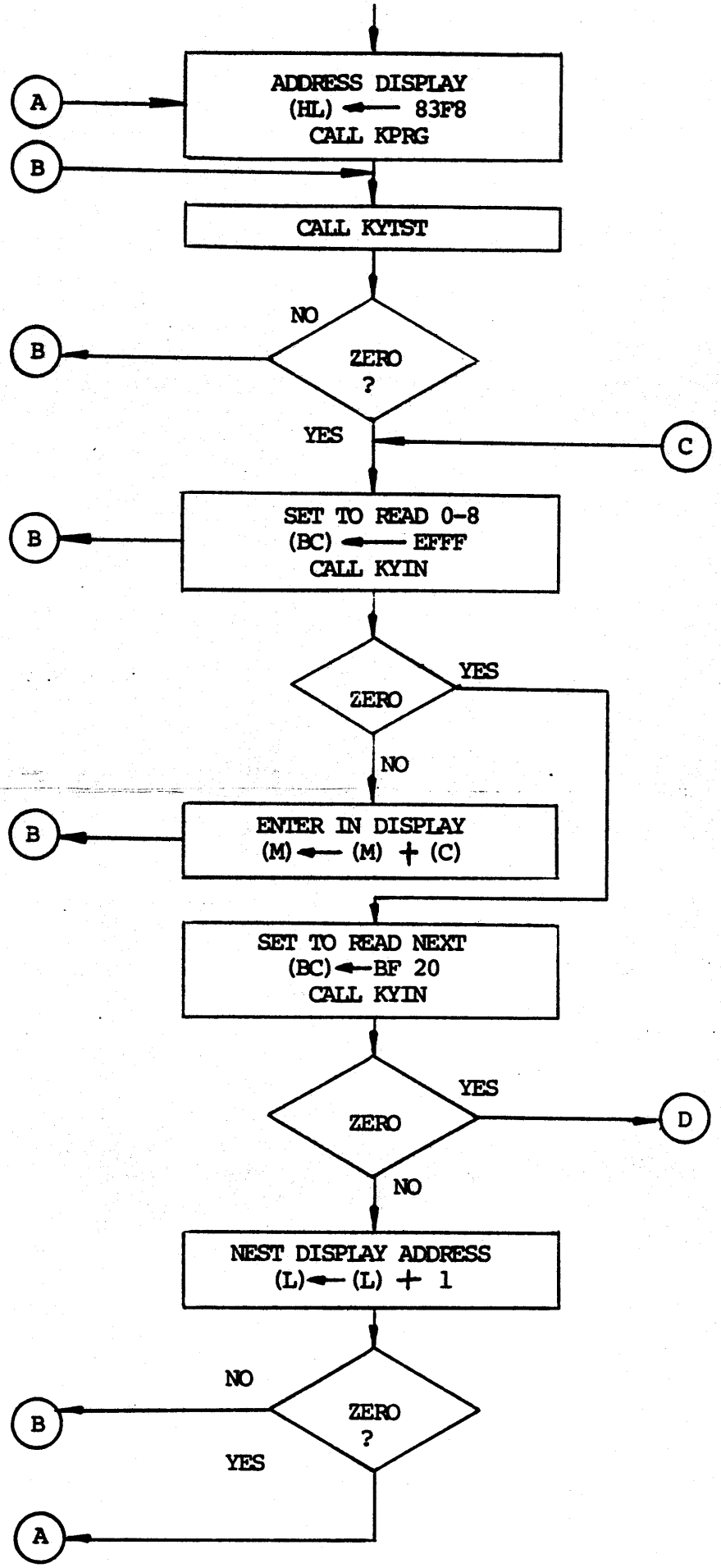


FIGURE 8-10

KEYBOARD DISPLAY PROGRAM (CONT'D)

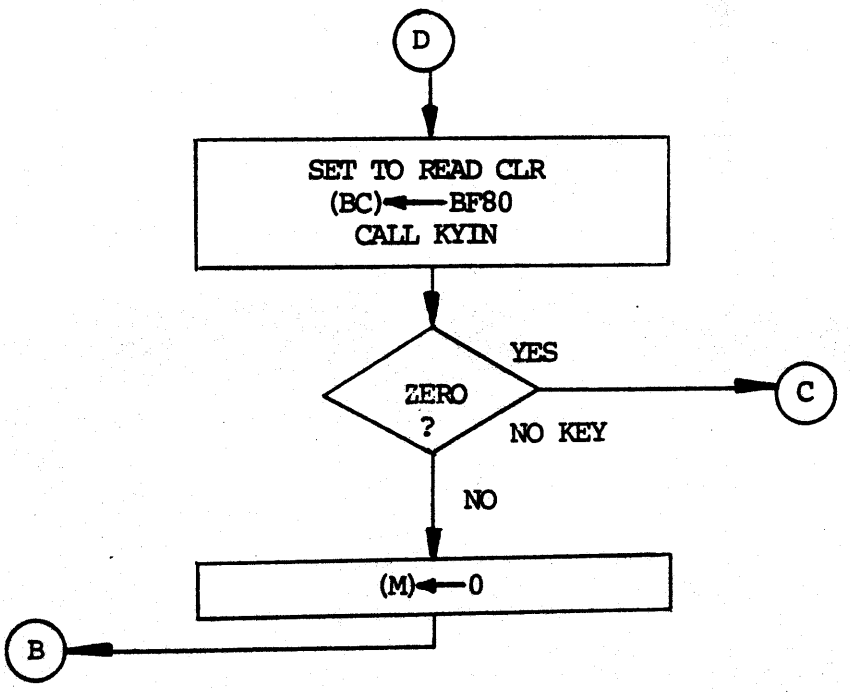


FIGURE 8-10 (Cont'd)

KEYBOARD DISPLAY PROGRAM

ADDR	CODE										
8200	00		NOP								
01	00		NOP								
02	00		NOP								
03	21		LXI		H,	83	F8				Address display
04	F8										
05	83										
06	CD		CALL		KPRG						Program 8255
07	40										only once
08	82										
09	CD		CALL		KTST						Test for any key
0A	44										
0B	82										
0C	C2		JNZ		8209						Wait for keys
0D	09										to be released
0E	82										
0F	00		NOP								For break point
8210	01		LXI		B,	EFFF					Set to read
11	FF										Keys 0-7
12	EF										
13	CD		CALL		KYIN						
14	48										
15	82										
16	CA		JZ		8220						Jump if no key
17	20										from 0-7
18	82										
19	7E		MOV		A,	M					OR key into
1A	B1		ORA		C						display
1B	77		MOV		M,	A					
1C	C3		JMP		8209						Go wait for
1D	09										release
1E	82										
1F	00		NOP								
~~8220~~											
21											
22											
23											
24											
25											
26											
27											
28											

FIGURE 8-11

KEYBOARD DISPLAY PROGRAM

INTEGRATED COMPUTER SYSTEMS — MICROCOMPUTER TRAINING SYSTEM — CODING SHEET

ADDR	CODE		Instruction	Operand	Comment
8220	0	1	LXI	B, BF20	Set to read NEXT
1	2	0			
2	B	F			
3	C	D	CALL	KYIN	
4	4	8			
5	8	2			
6	C	A	JZ	8230	Jump if not NEXT
7	3	0			
8	8	2			
9	2	C	INR	L	Next display address
A	C	2	JNZ	8209	Go wait for release
B	0	9			
C	8	2			
D	C	3	JMP	8200	At end of display
E	0	0			go to start
F	8	2			
8230	0	1	LXI	B, BF80	Set to read CLR
1	8	0			
2	B	F			
3	C	D	CALL	KYIN	
4	4	8			
5	8	2			
6	C	A	JZ	8210	Jump if not CLR
7	1	0			to read more keys
8	8	2			
9	3	6	MVI	M, 00	Clear display digit
A	0	0			
B	0	0	NOP		
C	C	3	JMP	8209	Go wait for release
D	0	9			
E	8	2			
F	0	0	NOP		
8 0					
1					
2					
3					
4					
5					
6					
7					
8					

FIGURE 8-12

8.1.6 Other I/O Interfaces

Isolated input/output is by no means restricted to the 8255; it is defined by the use of the IN and OUT instructions and the I/O Read and I/O Write commands. The necessary interface to the data bus, address bus and the command signals can be built with TTL and Tri-State circuits. Also, Intel, NEC and others offer several other devices made for this interface.

Many computer terminals use the 8251 Programmable Communication Interface for serial data communications. This has an interface to the 8080 system quite similar to that of the 8255, except that it needs the system clock. The student is again referred to the Intel 8080 User's Manual for detailed descriptions of these devices. Figure 8-13 shows how a number of devices can be connected to the system busses.

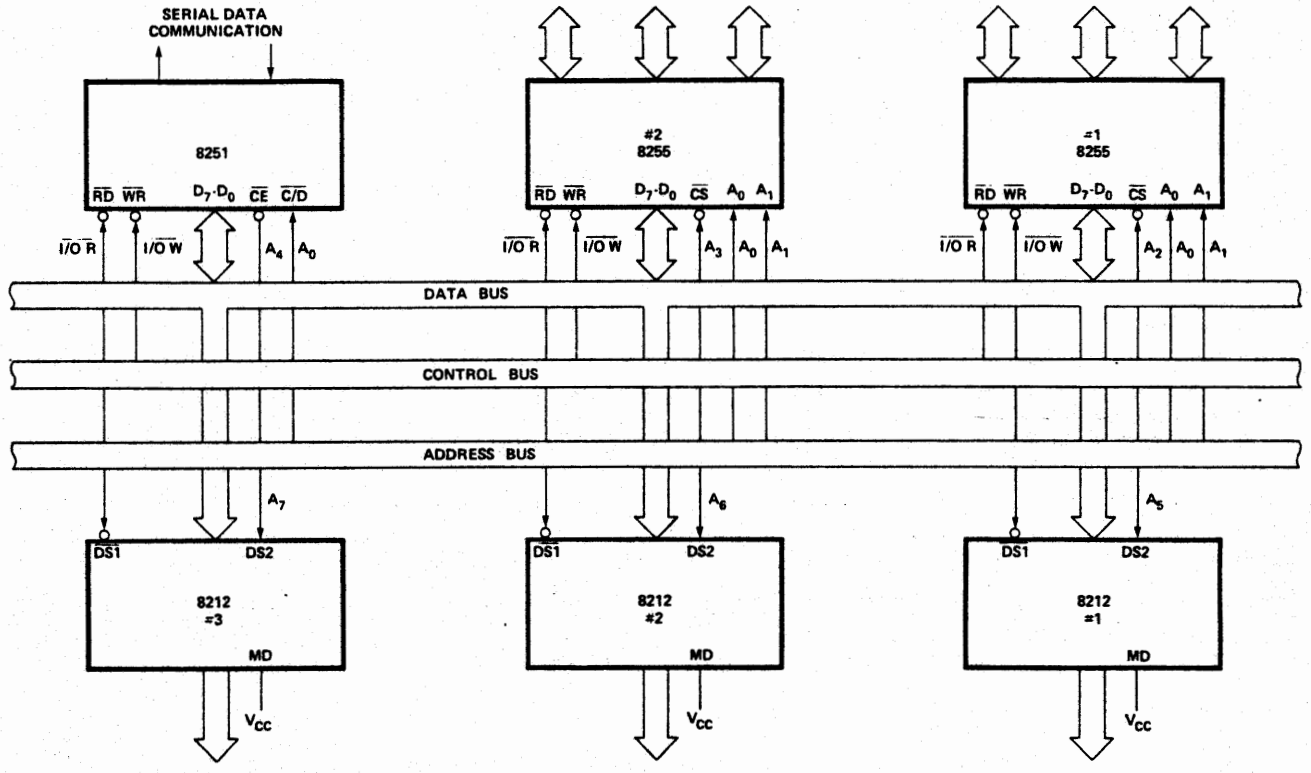


Figure 3-16. Typical I/O Interface.

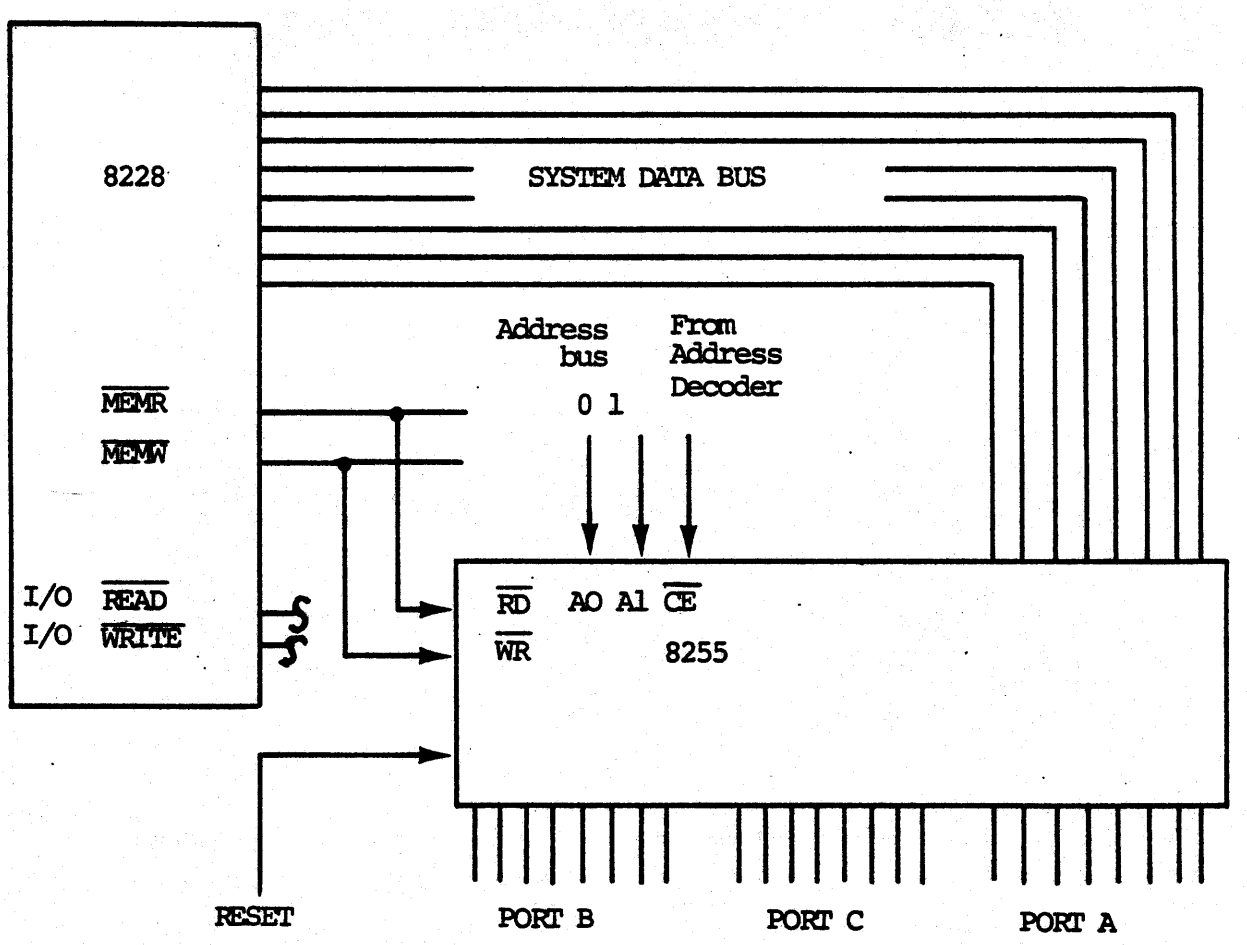
(FROM INTEL 8080 USER'S MANUAL)

FIGURE 8-13

8.2 MEMORY MAPPED INPUT/OUTPUT

An alternative to isolated input/output is 'memory mapped I/O'. The input or output device is connected to the Memory Read and/or Memory Write command signals from the 8228, instead of the I/O Read and I/O write commands. Figure 8-14 shows such a connection. Here the IN and OUT instructions are not used, since the device is not connected to the command signals they generate. Instead any memory read or write command can be used. LDA may be used in place of IN, STA in place of OUT. All the convenience of register addressing and transfer becomes available. If port A and port B are both programmed for input they could be read by:

```
LXI     H,FFF8    Address port A
MOV     E,M       (E) <- (port A)
INX     H         Address port B
MOV     D,M       (D) <- (port B)
```



MEMORY MAPPED INPUT/OUTPUT
WITH THE 8255

Figure 8-14

The arithmetic and logic instructions become available for direct use with the input port. If you want to wait for a change in the input data you could use this:

```
      LXI     H,FFF8    Address port A
      MOV     A,M       (A) <- (port A)
  ->  CMP     M         (A) = (port A)?
  |   JZ                Wait while equal
  -                     Exit at change
```

Or you can test for an input of 1111 1111:

```
      LXI     H,FFF8
      INR     M
  |-  JZ
  v
```

The INR M command is only partially effective. If port A is programmed for input, you cannot effectively write to it. Nevertheless the flags will be set as though you incremented the data.

While memory mapped I/O has some definite advantages, it sacrifices the two byte IN and OUT instructions. LDA and STA are three byte instructions; only by maintaining the I/O address in a register pair do you reduce the program length.

Note in Figure 8-11 that the chip enable of the 8255 now receives a decoded signal from the address bus. Clearly it must have a unique address, or at least an address for which no memory location exists. A typical scheme in small systems is to use all addresses from 8000 to

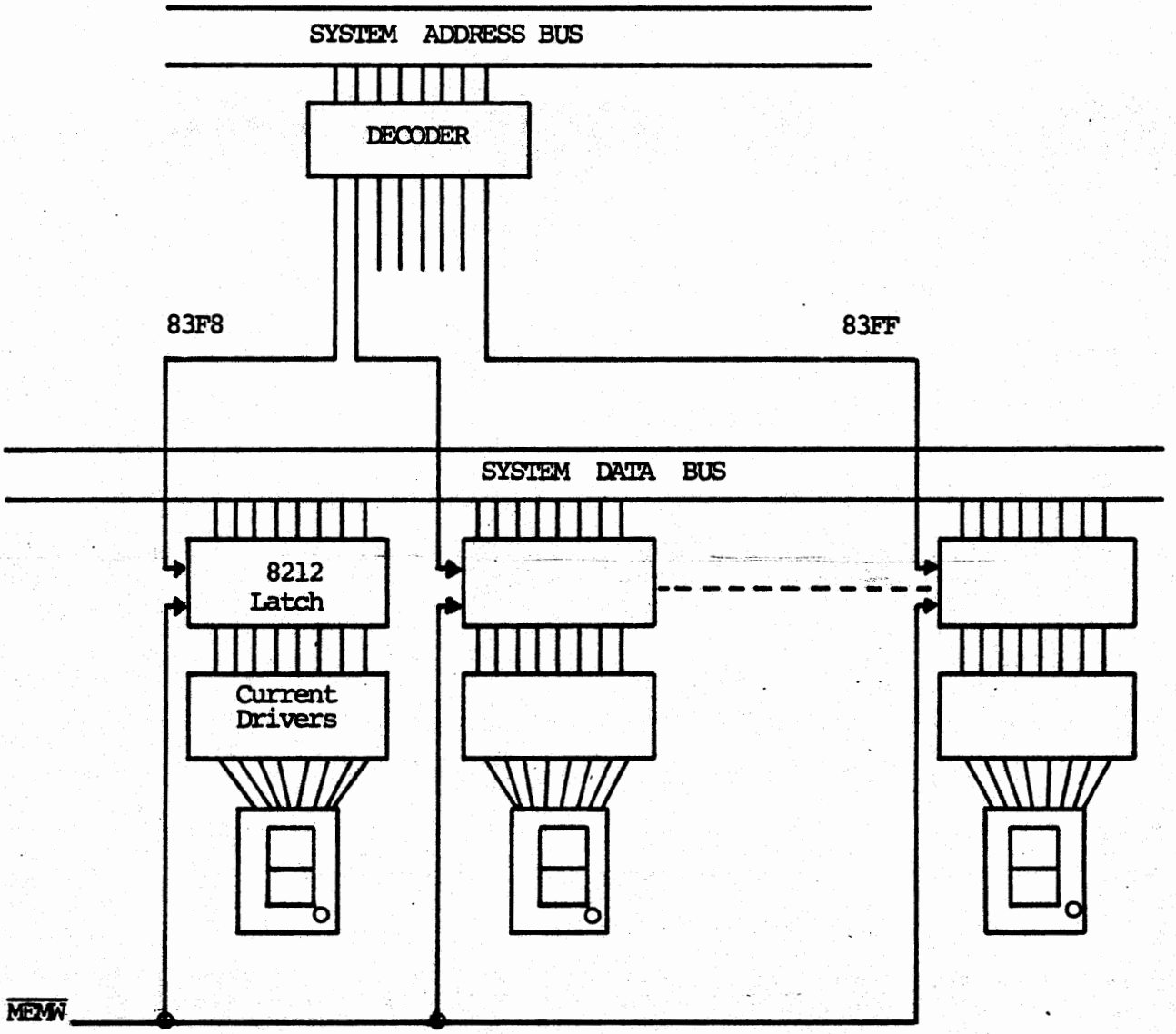
FFFF for input/output and 0000 to 7FFF for memory. The MTS does not do this. In fact the partial decoding of the memory address precludes memory mapped I/O if the empty memory sockets are filled. If they are left empty, then the chip select for one pair could be used for an I/O device.

Memory mapped I/O is probably overused in hardware design. For most applications isolated I/O is more efficient in both hardware and program space - but the difference is very small.

8.3 DIRECT MEMORY ACCESS

The third method of input and output is direct memory access, in which data are written to the processor's memory, or read from it, by external hardware as well as by the CPU. This is very efficient for the program, but typically it demands more external hardware than input and output ports require. We will describe in detail the DMA system used in the MTS for its display.

Let us suppose for a moment that we did not have memory devices at addresses 8300 - 83FF in the MTS, but a set of output latches, as shown in Figure 8-15. Now to display a digit we would use memory mapped I/O, addressing 83F8, 83F9, etc and write to those apparent memory locations. The data would be stored in the 8212 latches and would drive the LED displays. This demands eight latches and eight current drivers. Direct Memory Access provides an alternative which in this case takes less external hardware and appears almost identical to the program.



MEMORY MAPPED DISPLAY

Figure 8-15

8.3.1 Repetitive Direct Memory Access

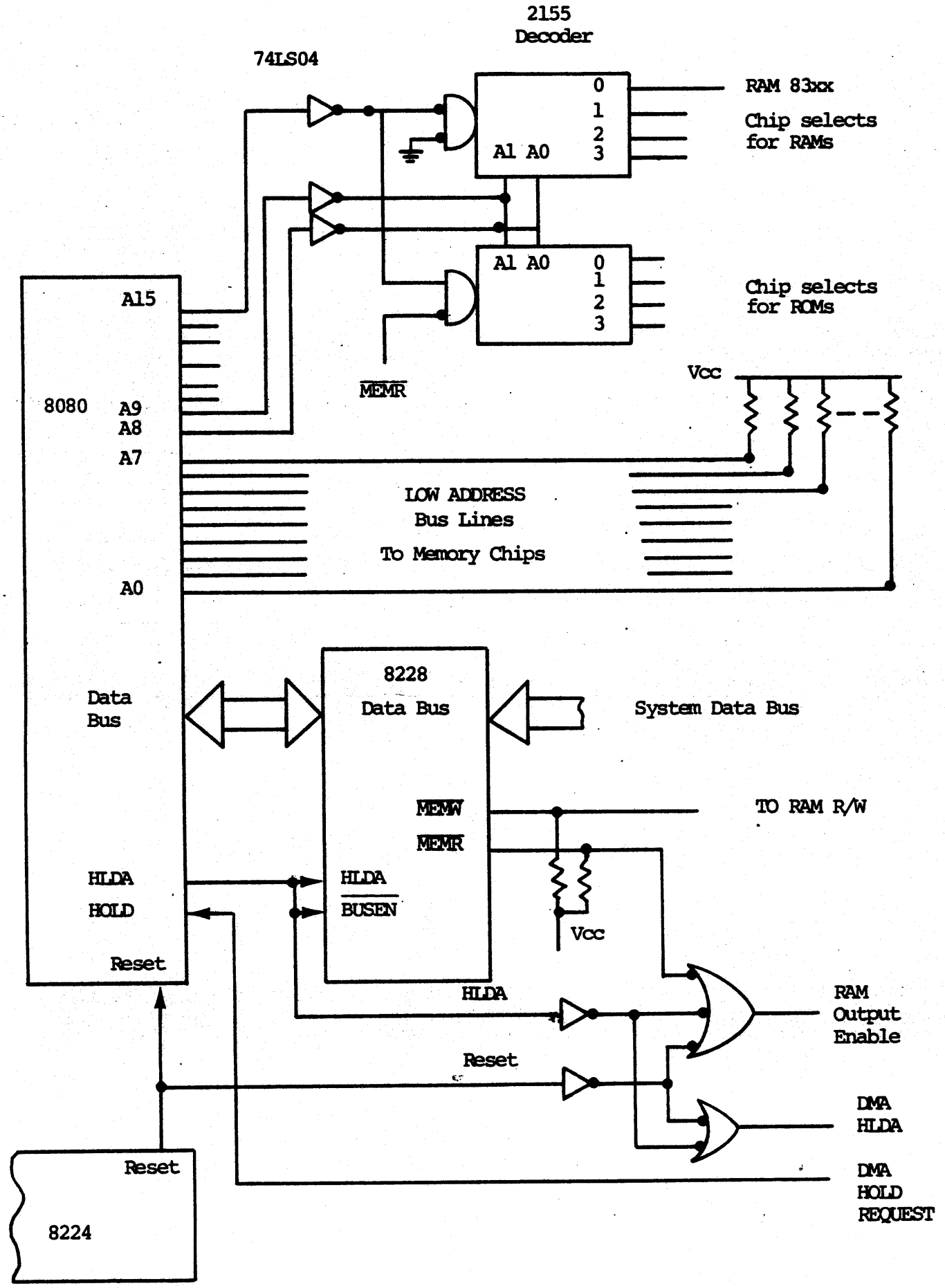
In using the seven segment displays of the MTS you have been operating a repetitive direct memory access system. Data are written into a fixed set of addresses, and the DMA hardware periodically obtains data from these addresses and displays it. This is a very attractive scheme for displays of the kind used here, and also for video displays and some kinds of control systems. In each case the same data need to be accessed repetitively because very little external storage is provided. For the seven segment displays of the MTS only one digit is stored externally, while that digit is illuminated. Then the DMA channel obtains the next digit and displays it.

Figure 8-16 shows the circuit connections to the 8080 that are involved in the DMA operation. The DMA channel periodically issues HOLD Request. The 8080 suspends its use of the address, data, and control busses as soon as possible (i.e. when any memory read or write process is finished). It then issues Hold Acknowledge (HLDA) to the 8228 system controller and to the DMA channel, and floats the address busses. In response to HLDA the 8228 floats the data and control busses. (By float we mean place the device connections to the busses in the high impedance state so that other devices can drive the busses).

HLDA is OR gated with the memory read command $\overline{\text{MEMR}}$ (which is floated by 8228 but pulled up by a resistor) and with RESET, so that the read write memory outputs are enabled in response to any of these signals. Thus during HLDA the selected RAM will drive the data bus. Another OR gate delivers HLDA to the DMA channel to permit it to control the address bus.

RESET is OR'ed into both of these signals so that the DMA circuit will function whenever the RESET key is pressed. This is a valuable trouble shooting tool, because if a failure results in a blank display it can immediately be isolated between the DMA channel and the microprocessor.

The low address lines, which go directly to the memory chips, are floated by the 8080, pulled up by resistors, and the lowest three bits are controlled by the DMA channel. The high address lines are also floated. These do not have external resistors so they are actually in the high impedance state. Only three of these lines are used in the MTS: A15, A9 and A8. The low power Schottky inverters (75LS04) have internal pullup resistors, so their outputs go low as if an address of 1xxxxx11 had been output by the 8080. This address selects the RAM chips for page 8300. This pair of chips therefore is selected, its outputs are enabled, and the $\overline{\text{MEMW}}$ signal is pulled up by a resistor to indicate a read operation.



DMA CONNECTIONS TO 8080

Figure 8-16

Figure 8-17 shows the circuit of the DMA channel. The timing source is a linear integrated circuit Single Shot (555) that generates a narrow pulse at a fairly long interval, provided that the enable signal from the output port is high. (Reset forces all ports to the input mode, which allows the single shot to run). The ENABLE also controls the 2155 Decoder that selects one digit, so that when DMA is disabled no digit will be driven.

The pulse from the 555 provides the HOLD request to the 8080 and increments the 223 Binary Counter. When the HLDA is received from the 8080 the open collector AND gates place the new content of the counter onto address bus bits 0, 1 and 2, thereby addressing the desired digit position in memory (all other bits of the low address bus being held high by resistors). The selected memory location is read onto the data bus and received by the 8212 data latch, and at the trailing edge of the HOLD pulse the data are latched into the 8212 and delivered to the segment drivers. The timing relationship is shown in Figure 8-18.

The digit address from the counter also addresses the 2155 Decoder to select one of the eight drive transistors for the eight digits. This remains stable until the next HOLD request pulse.

It was mentioned earlier in this chapter that you should not alternately enable and disable the display at time intervals short compared to the DMA time interval. Disabling inhibits the 555 Single Shot, so the address will not change until 0.5 milliseconds after it is enabled again. If the enable signal is given soon after the disable, and the two are alternated, the 555 will not produce any pulses. Nevertheless

the display driver is enabled for part of the time period. This results in one digit being repeatedly driven at a higher duty cycle than is intended.

If you operate the MTS in a darkened room and cover the illuminated digits, you will see a faint ghost in a blank digit. This is because for the 2 to 5 microsecond duration of the HOLD request, the decoder has selected a new digit but the 8212 still holds data from a preceding digit. In a critical application the ghost could be eliminated by gating the HOLD pulse with the enable signal to the decoder.

We have described the DMA channel of the MTS in some detail. In the next section we will discuss DMA for purposes other than display, but in more general terms.

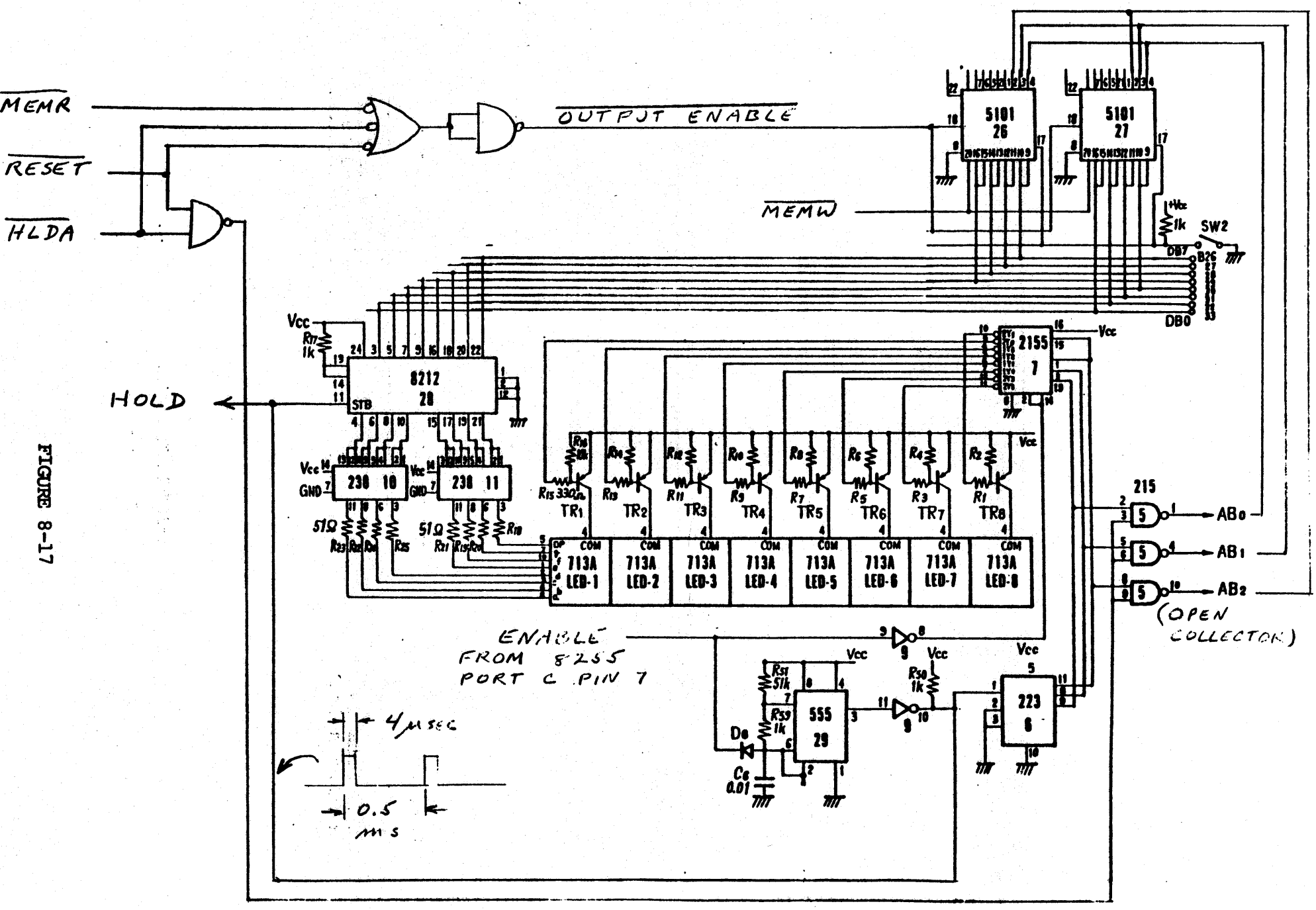
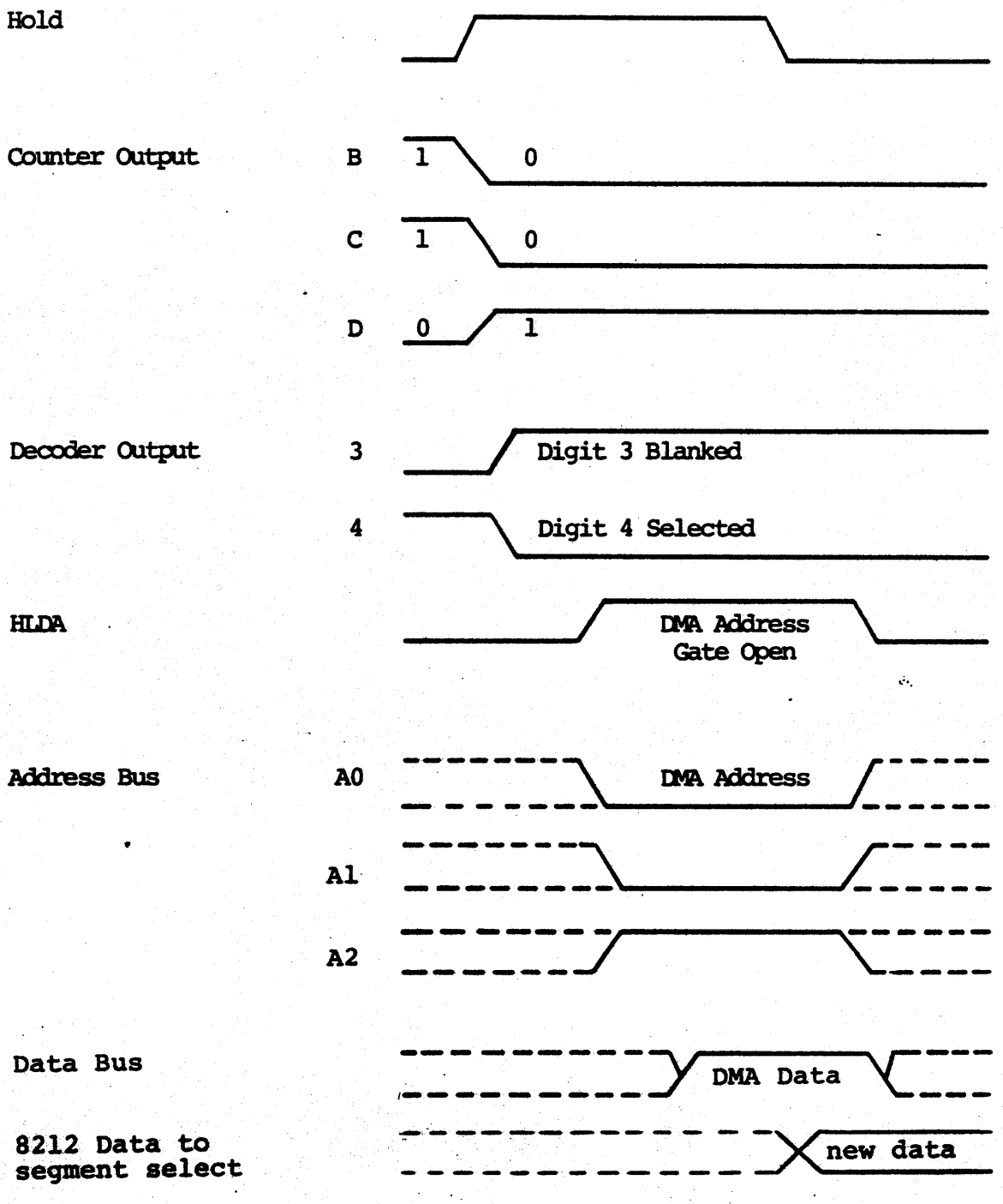

MEMR
RESET
HLDA
OUTPUT ENABLE
MEMW
HOLD
5101 26
5101 27
8212 28
2155 7
238 10
238 11
713A LED-1
713A LED-2
713A LED-3
713A LED-4
713A LED-5
713A LED-6
713A LED-7
713A LED-8
TR1
TR2
TR3
TR4
TR5
TR6
TR7
TR8
215
AB0
AB1
AB2
(OPEN COLLECTOR)
ENABLE FROM 8255 PORT C PIN 7
555 29
223 6
SW2
DB7
DB0
4 μsec
0.5 ms
Vcc
GND
51Ω

FIGURE 8-17



DMA TIMING IN MTS

Figure 8-18

8.3.2 DMA Input and Output

Direct memory access is commonly used in computer systems for both input and output if a high data rate is required. Reading or writing to magnetic disc memory is a typical example; Intel's Microcomputer Development System/Diskette Operating System operates at 250,000 bits per second or about 30 microseconds per byte. The 8080 could not keep up with such a data rate on a programmed or interrupt driven input system. In fact Intel uses their series 3000 Bipolar Microprocessor for the disc controller.

The disadvantage of DMA is the significant amount of external hardware required. It should seldom be used unless high data rates are mandatory, or in specialized situations such as repetitive DMA where the hardware is minimized. The hardware always includes the following:

a) Address counter to store and alter the memory address to be read or written (represented by the 223 Three Bit Counter in the MTS)

b) Address Bus buffer to isolate the DMA address from the system bus (the open collector AND gates)

c) Data-Bus buffer to isolate the DMA data from the system bus (the 8212)

d) Gating circuits to appropriately command memory read or memory write.

e) Timing or signal input to initiate the hold request (the 555).

In any DMA system other than a repetitive DMA there must be some means for the processor to inform the DMA channel that output data are ready, and for the DMA channel to inform the processor that input data have been stored or output data accepted. This can be handled as a separate programmed I/O, with the processor and channel exchanging discrete signals. If DMA input and output are both provided it can be done by writing a control byte into a specified memory location as the last operation in the DMA sequence; then the processor and channel both sample that location periodically. The most common practice, however, is to use a discrete output from the processor to initiate output and enable input, and an interrupt from the channel when data transfer is complete.

Sophisticated DMA systems generally provide for reading and writing to variable areas of memory. For output the processor will send a memory address and a byte count to the channel, which thereafter takes data from the given and succeeding addresses until the designated number of bytes have been read. For input the channel may interrupt to request a memory address where data are to be stored.

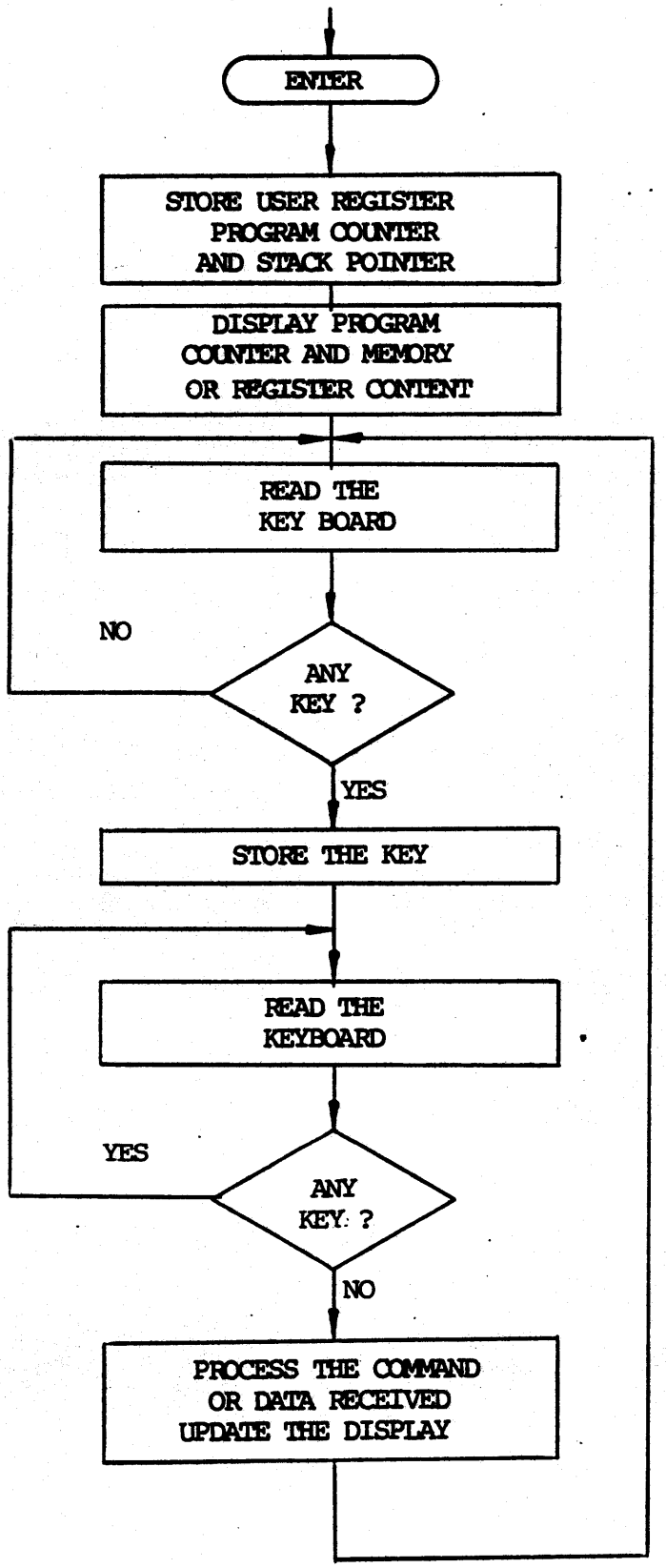
The DMA facility of the MTS is dedicated to its display; it is not practical to modify the system for external DMA.

8.4 I/O INITIATION

8.4.1 Programmed I/O

Because a computer operates in sequential fashion, it is not always ready to receive an input or produce an output. If it is fast in comparison to the input device or the output requirement, which it often is, the computer can sample the input or produce the output at its own convenience. This is called 'Programmed I/O'. It is used in the MTS for the keyboard input. When the computer is slow compared to the input or output requirement, as in a magnetic disc system, we use direct memory access, but typically with either programmed or interrupt I/O to initiate and/or terminate the DMA operation. This section will be mainly concerned with the subject of interrupts.

Consider the MTS keyboard input. When the monitor is in control (running), almost all of its time is spent waiting for keyboard input (See Figure 8-19). The program has nothing better to do with its time. It can process any command you give it and get back to reading the keyboard long before you can press another key.

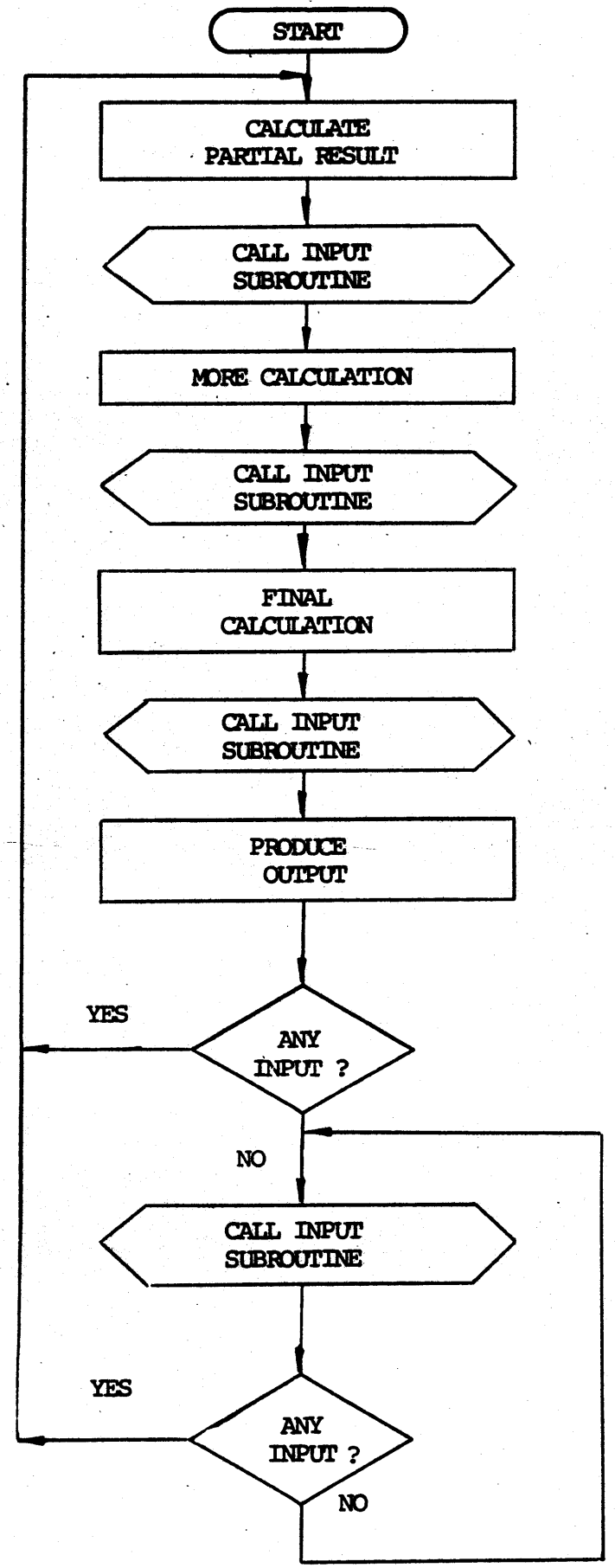


KEYBOARD TESTING IN THE MONITOR

FIGURE 8-19

The processor can tell whether you have pressed a key because a unique state exists (all inputs high) when no key is pressed. It tests for this state after each new key input before processing the key, to avoid processing a single key stroke repetitively, and yet be able to react to multiple operations of the same key. In many input applications there is no special state which has a significance different from all others, and the processor must know by other means whether a particular input has been processed. There are, of course, applications where it does not matter; a digital voltmeter will process the input as fast as it can update its display whether the data has changed or not.

In some systems the processor has lengthy functions to perform, which must be interrupted to handle input or output. This can be done by repeatedly calling an input subroutine during the main processing, as suggested in Figure 8-20. This tends to be time wasting, and it demands that the programmer consider how long his processing will take in comparison to the input requirement.



PROGRAMMED INPUT/OUTPUT

Figure 8-20

Another method is offered by the strobed input feature of the 8255; the input can be fleeting and asynchronous, but will be stored in the data latch of the 8255 until the program is ready to handle it. This is very suitable for infrequent inputs such as may exist in control systems. Sometimes, however, the system may demand a very prompt response to its occasional inputs, or it may give many inputs during the course of the processor's other calculations, each demanding some degree of processing or at least storage before the next input is delivered. It is for this kind of requirement that interrupt driven systems were invented.

8.4.2 Interrupt Driven I/O

When an external event occurs that demands the processor's immediate attention, hardware is used to cause a branch in the program. Instead of repeated calls to an input (or output) subroutine at predetermined intervals, as suggested in Figure 8-20, that call is created when and only when it is needed. The 8080 and most other microprocessors include interrupt handling capability.

We will discuss the internal and external logic required to create an interrupt; the MTS interrupt system; and the design of interrupt service subroutines.

8.4.2.1 Interrupt Logic

The following signals of the 8080/8228 system are involved in the logic handling an interrupt:

INT Interrupt. Request input to the 8080. It is driven high by external hardware to request service.

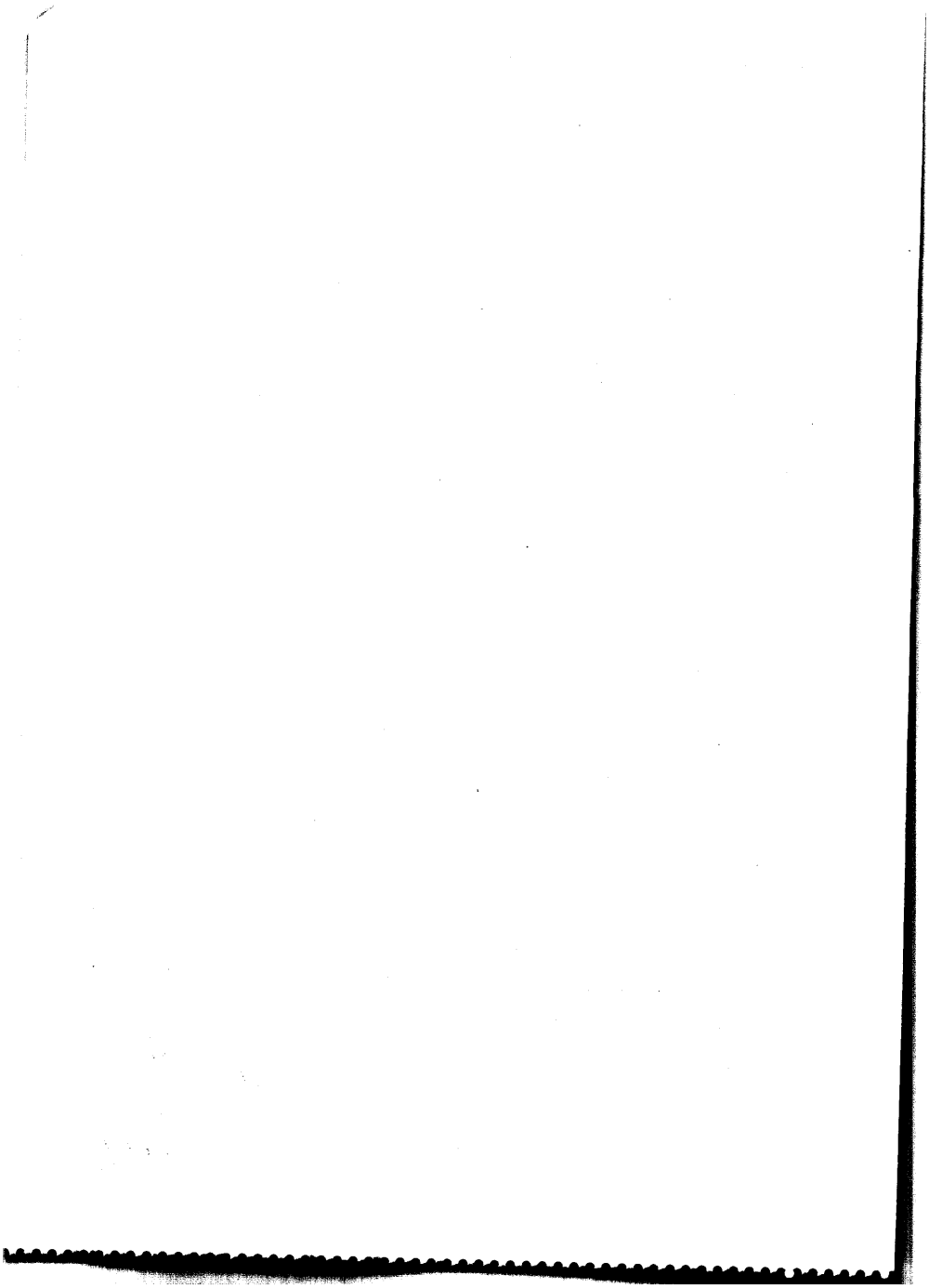
INTE Interrupt Enable. A flip flop in the 8080 and also an external output, signifying that an interrupt will be accepted.

INT F/F Interrupt Accept. A flip flop in the 8080 signifying that an interrupt has been accepted.

INTA Interrupt Acknowledge. A signal passed in the status byte to the 8228, and also an output signal from the 8228 available to external hardware.

To create an interrupt the external logic must (in general) perform two functions: request an interrupt by raising INT, and respond to INTA by giving the 8080 an instruction. The instruction is usually one of the special one-byte restart calls: RST0, RST1, etc. These are essentially identical to the CALL instruction except that the address is implied by the op-code. Thereafter the processor executes an interrupt service subroutine just as it would any other subroutine.

Interrupt



Some systems have a requirement to test INTE to be sure that an interrupt will be accepted. In other systems it can be used as an indication that an interrupt has been accepted. It is not generally necessary to use this signal externally. It is internally gated with the interrupt request, so that interrupts will not be honored unless the interrupt system is enabled.

The interrupt system is enabled by a RESET, or by the instruction:

FB EI Enable Interrupt

This instruction sets the INTE flag high, but it is carefully arranged to be too late for the next instruction to be interrupted. It is guaranteed that one instruction (usually a RETurn from the interrupt service subroutine) will be executed before another interrupt is accepted.

The interrupt system is disabled by execution of an interrupt. This ensures that the interrupt service subroutine can accomplish its functions without itself being interrupted. It can also be disabled by the instruction:

F3 DI Disable Interrupt

This is commonly used when some time dependent task is to be executed and must not be delayed by interrupts, or when a process is being performed that will affect the results of the next interrupt.

Provided that INTE is set, the INT input sets the internal INT Flip Flop at the end of the current instruction, which is completed before any other action occurs.

When the next instruction cycle starts with INT F/F set, some special events occur. The CPU starts its normal cycle, sending out the PC content and status data. The status includes INTA, a bit on the data bus during status strobe time, which commands the 8228 to issue the INTA command instead of the MEMR command. Then an instruction is placed on the data bus, either by external logic or by the 8228 itself, so that this is loaded into the instruction register in place of the next programmed instruction. During this cycle the 8080 does not increment the program counter, so the address of the instruction that has been interrupted is preserved. The 8080 clears the INT F/F and the Interrupt Enable Flag, so that the next instruction will not be interrupted.

8.4.2.2 Restart Instructions

It is usual (but not necessary) that the instruction placed on the data bus in response to INTA is one of the special one-byte call instructions, RST0 to RST7. These are equivalent to normal CALL's except that the call address is implied by the op-code, as shown in Figure 8-21. The diagrams of Figures 8-22 through 8-24 show the process, and Figure 8-25 (From the Intel 8080 User's Manual) shows the timing.

HEX CODE	INSTR	BINARY CODE			CORRESPONDS TO	NEW PROGRAM	COUNTER	
C7	RST 0	11	000	111	CALL 0000	0000000000000	000	000
CF	RST 1	11	001	111	CALL 0008	0000000000000	001	000
D7	RST 2	11	010	111	CALL 0010	0000000000000	010	000
DF	RST 3	11	011	111	CALL 0018	0000000000000	011	000
E7	RST 4	11	100	111	CALL 0020	0000000000000	100	000
EF	RST 5	11	101	111	CALL 0028	0000000000000	101	000
F7	RST 6	11	110	111	CALL 0030	0000000000000	110	000
FF	RST 7	11	111	111	CALL 0038	0000000000000	111	000

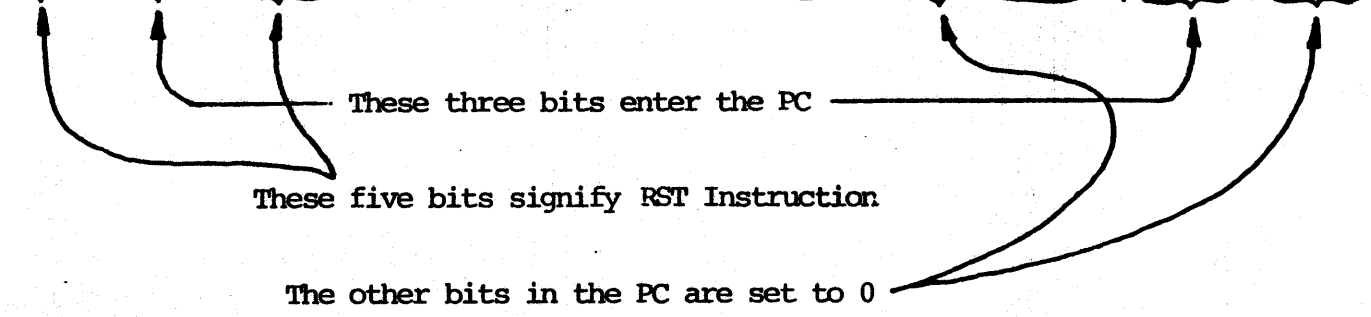
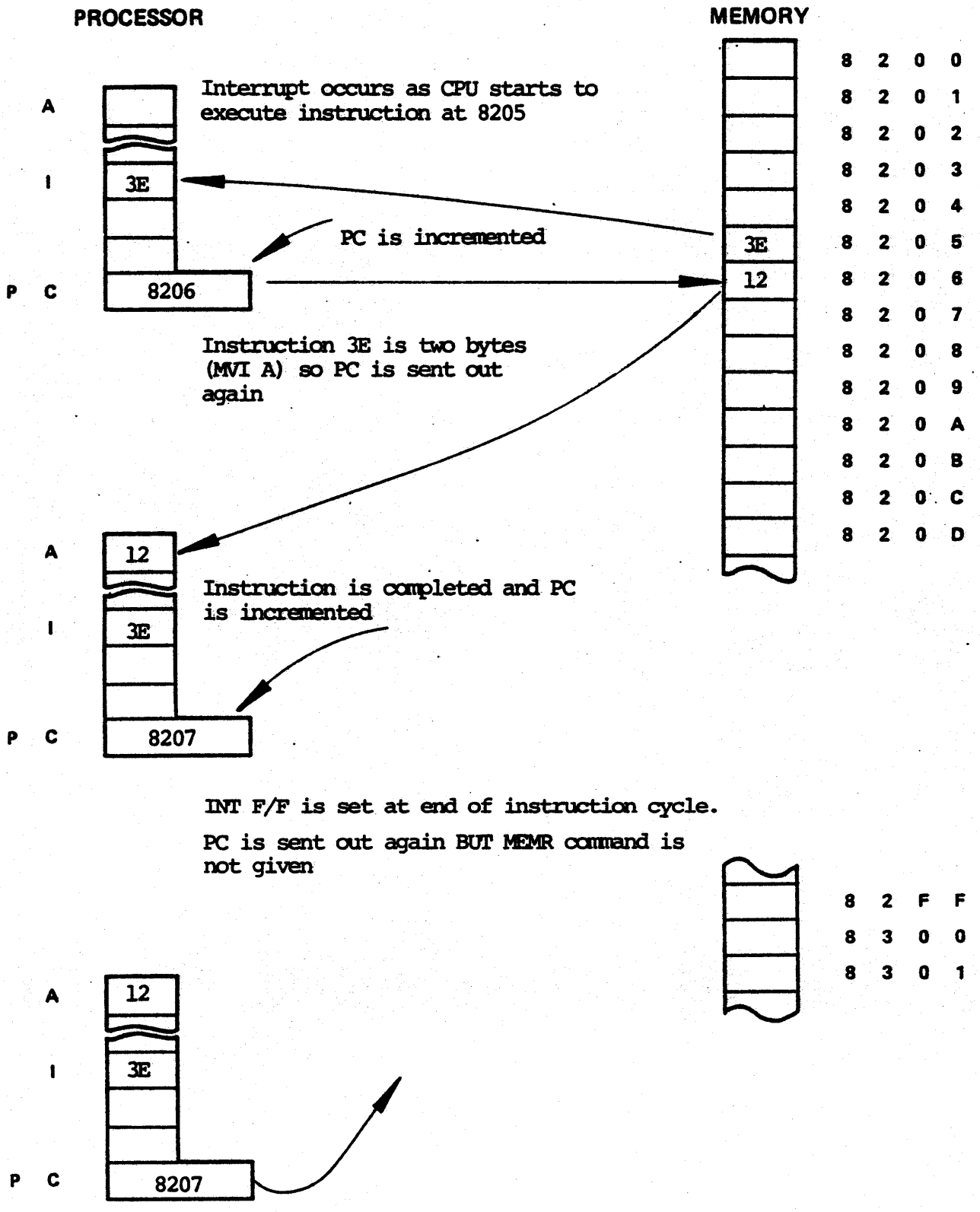


FIGURE 8-21

CODING AND EFFECT OF RST INSTRUCTIONS

FIGURE 8-21



INTERRUPT PROCESSING

Figure 8-22

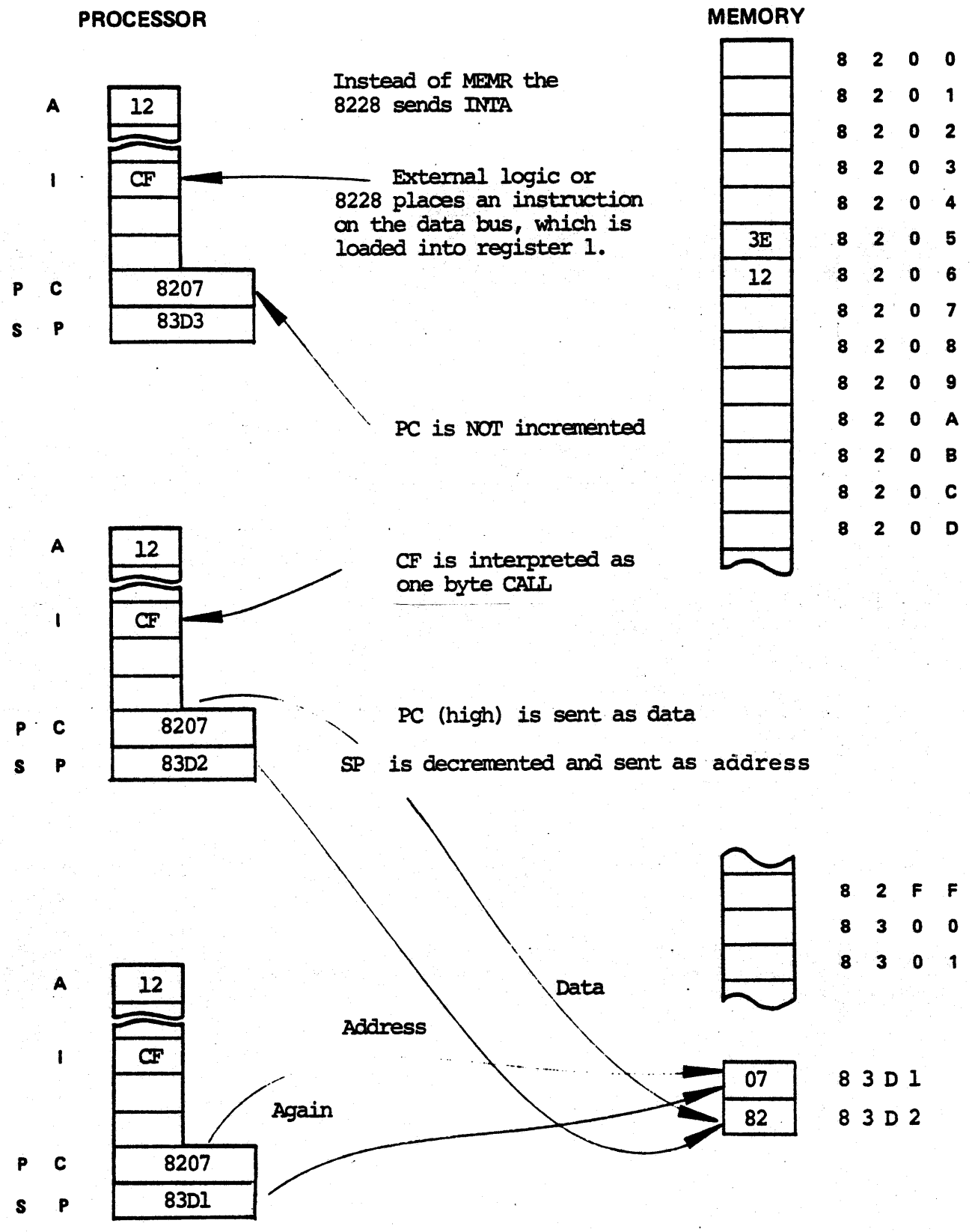

PROCESSOR
MEMORY
A
I
P C
S P
12
CF
8207
83D3
Instead of MEMR the 8228 sends INTA
External logic or 8228 places an instruction on the data bus, which is loaded into register 1.
PC is NOT incremented
3E
12
8 2 0 0
8 2 0 1
8 2 0 2
8 2 0 3
8 2 0 4
8 2 0 5
8 2 0 6
8 2 0 7
8 2 0 8
8 2 0 9
8 2 0 A
8 2 0 B
8 2 0 C
8 2 0 D
CF is interpreted as one byte CALL
8207
83D2
PC (high) is sent as data
SP is decremented and sent as address
8 2 F F
8 3 0 0
8 3 0 1
Data
Address
Again
07
82
8 3 D 1
8 3 D 2
8207
83D1

Figure 8-23

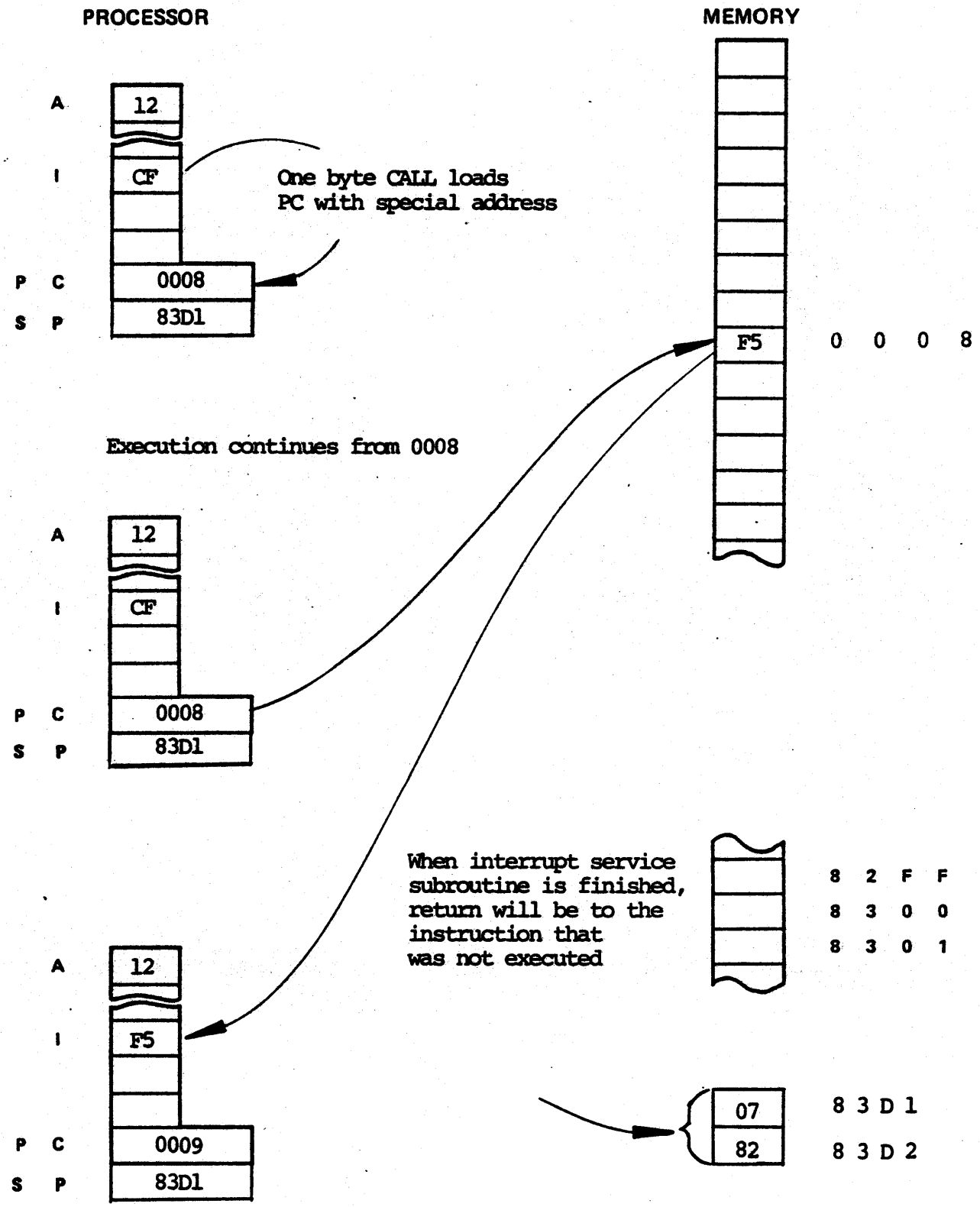

PROCESSOR
MEMORY
A
I
P C
S P
12
CF
0008
83D1
One byte CALL loads
PC with special address
F5
0 0 0 8
Execution continues from 0008
A
I
P C
S P
12
CF
0008
83D1
When interrupt service
subroutine is finished,
return will be to the
instruction that
was not executed
8 2 F F
8 3 0 0
8 3 0 1
A
I
P C
S P
12
F5
0009
83D1
07
82
8 3 D 1
8 3 D 2

Figure 8-24

INTERRUPT SEQUENCES

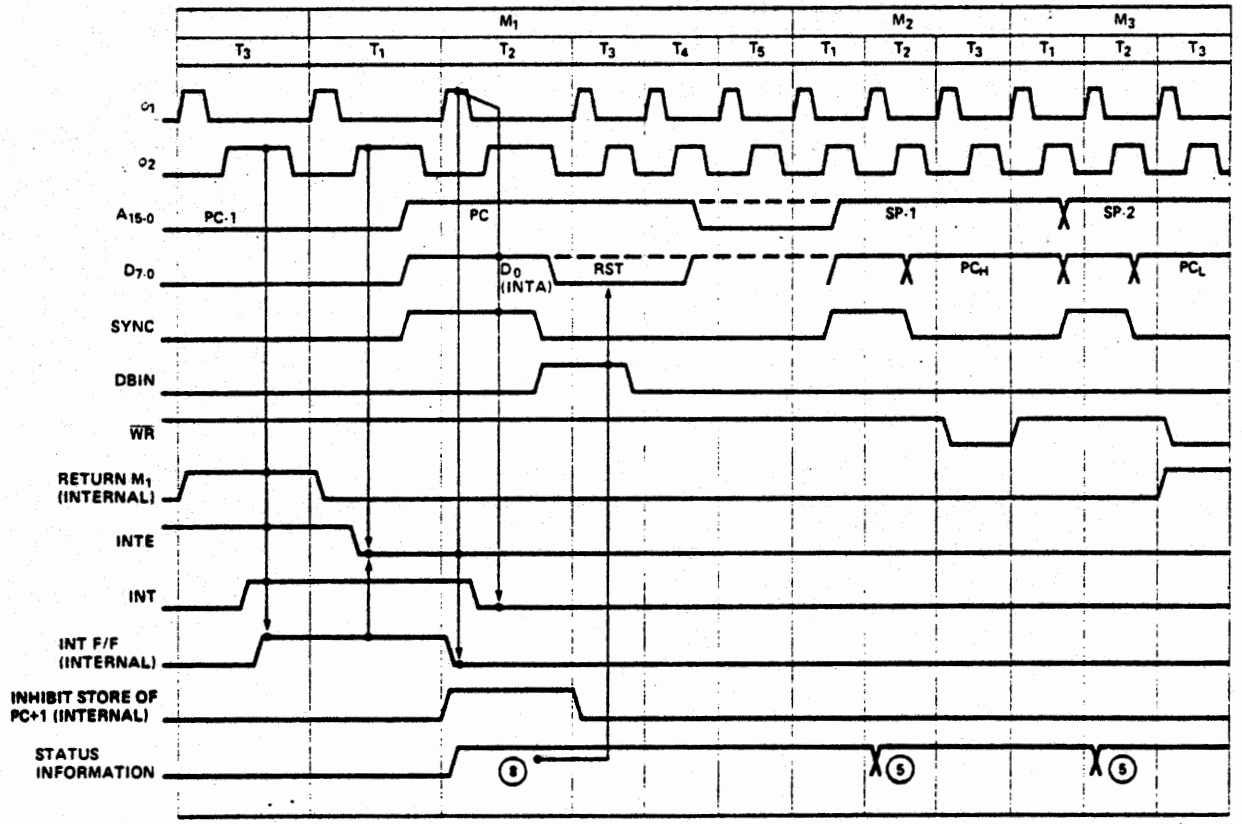
The 8080 has the built-in capacity to handle external interrupt requests. A peripheral device can initiate an interrupt simply by driving the processor's interrupt (INT) line high.

The interrupt (INT) input is asynchronous, and a request may therefore originate at any time during any instruction cycle. Internal logic re-clocks the external request, so that a proper correspondence with the driving clock is established. As Figure 2-8 shows, an interrupt request (INT) arriving during the time that the interrupt enable line (INTE) is high, acts in coincidence with the ϕ_2 clock to set the internal interrupt latch. This event takes place during the last state of the instruction cycle in which the request occurs, thus ensuring that any instruction in progress is completed before the interrupt can be processed.

The INTERRUPT machine cycle which follows the arrival of an enabled interrupt request resembles an ordinary FETCH machine cycle in most respects. The M_1 status bit is transmitted as usual during the SYNC interval. It is accompanied, however, by an INTA status bit (D_0) which acknowledges the external request. The contents of the program counter are latched onto the CPU's address lines during T_1, but the counter itself is not incremented during the INTERRUPT machine cycle, as it otherwise would be. In this way, the pre-interrupt status of the program counter is preserved, so that data in the counter may be restored by the interrupted program after the interrupt request has been processed.

The interrupt cycle is otherwise indistinguishable from an ordinary FETCH machine cycle. The processor itself takes no further special action. It is the responsibility of the peripheral logic to see that an eight-bit interrupt instruction is "jammed" onto the processor's data bus during state T_3. In a typical system, this means that the data-in bus from memory must be temporarily disconnected from the processor's main data bus, so that the interrupting device can command the main bus without interference.

The 8080's instruction set provides a special one-byte call which facilitates the processing of interrupts (the ordinary program Call takes three bytes). This is the RESTART instruction (RST). A variable three-bit field embedded in the eight-bit field of the RST enables the interrupting device to direct a Call to one of eight fixed memory locations. The decimal addresses of these dedicated locations are: 0, 8, 16, 24, 32, 40, 48, and 56. Any of these addresses may be used to store the first instruction(s) of a routine designed to service the requirements of an interrupting device. Since the (RST) is a call, completion of the instruction also stores the old program counter contents on the STACK.



NOTE: (N) Refer to Status Word Chart on Page 2-6.

Figure 2-8. Interrupt Timing

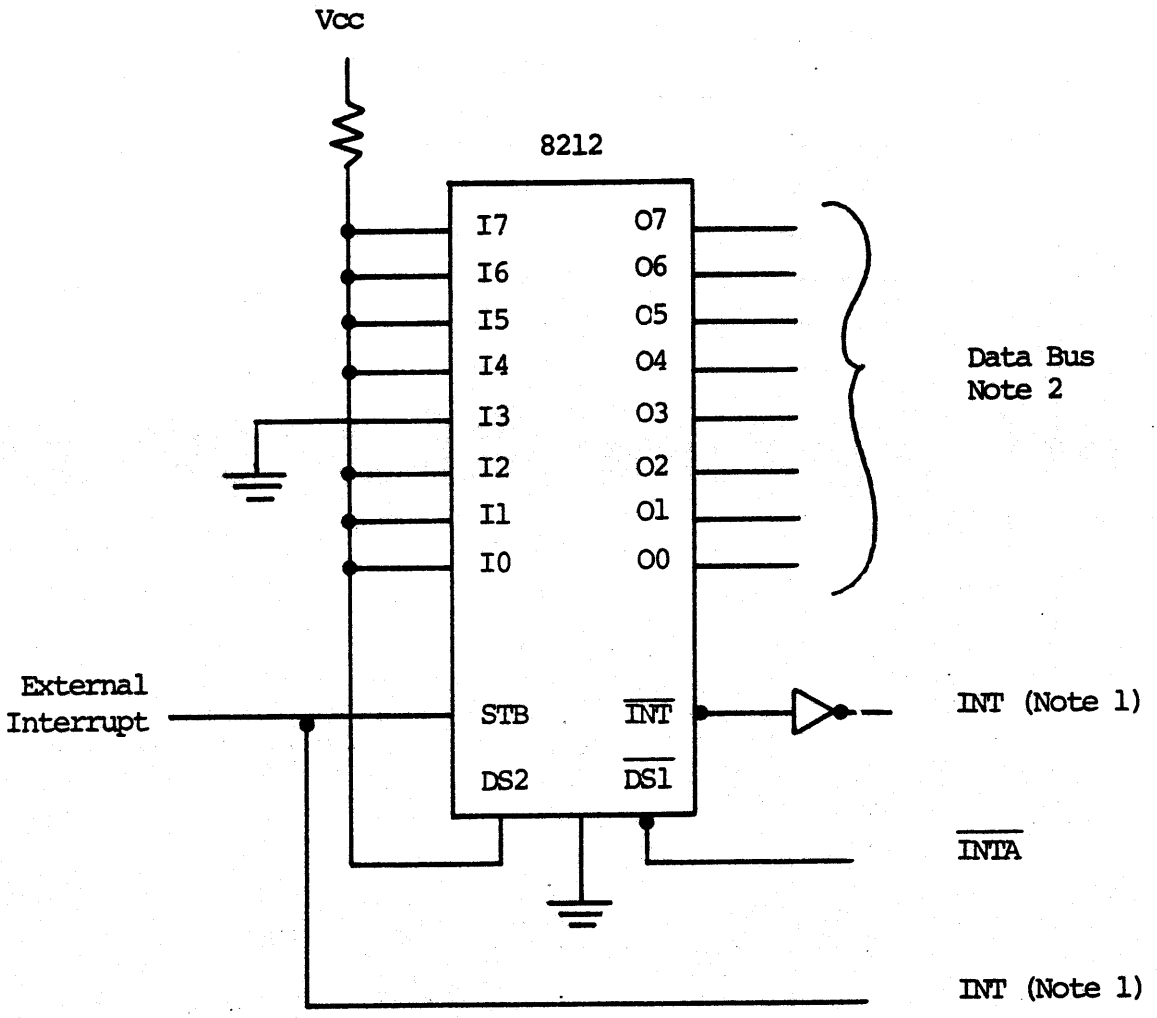
FIGURE 8-25

8.4.2.3 Interfaces for RST Instruction

The restart (or other) instruction that is to be placed on the data bus during INTA must not interfere with the data bus at other times. It is best to buffer the data bus with a tri-state device such as the Intel or NEC 8212, or two 74125 Quad Buffers. Figure 8-26 shows an 8212 generating RST6 in response to an external interrupt.

When more than one device is to interrupt the 8080, it is often useful to use vectored interrupts. Each device creates a different RST instruction, thereby calling a different service routine. Figure 8-27 shows an arrangement with which two independent interrupts can create three different restarts: RST5 for INT1, RST6 for INT2, and RST4 for both at once.

In a small system, the data bus can tolerate some resistive pullup, and tri-state or open collector inverters or gates can be used to pull down specific bits. Figure 8-28 shows such a configuration.

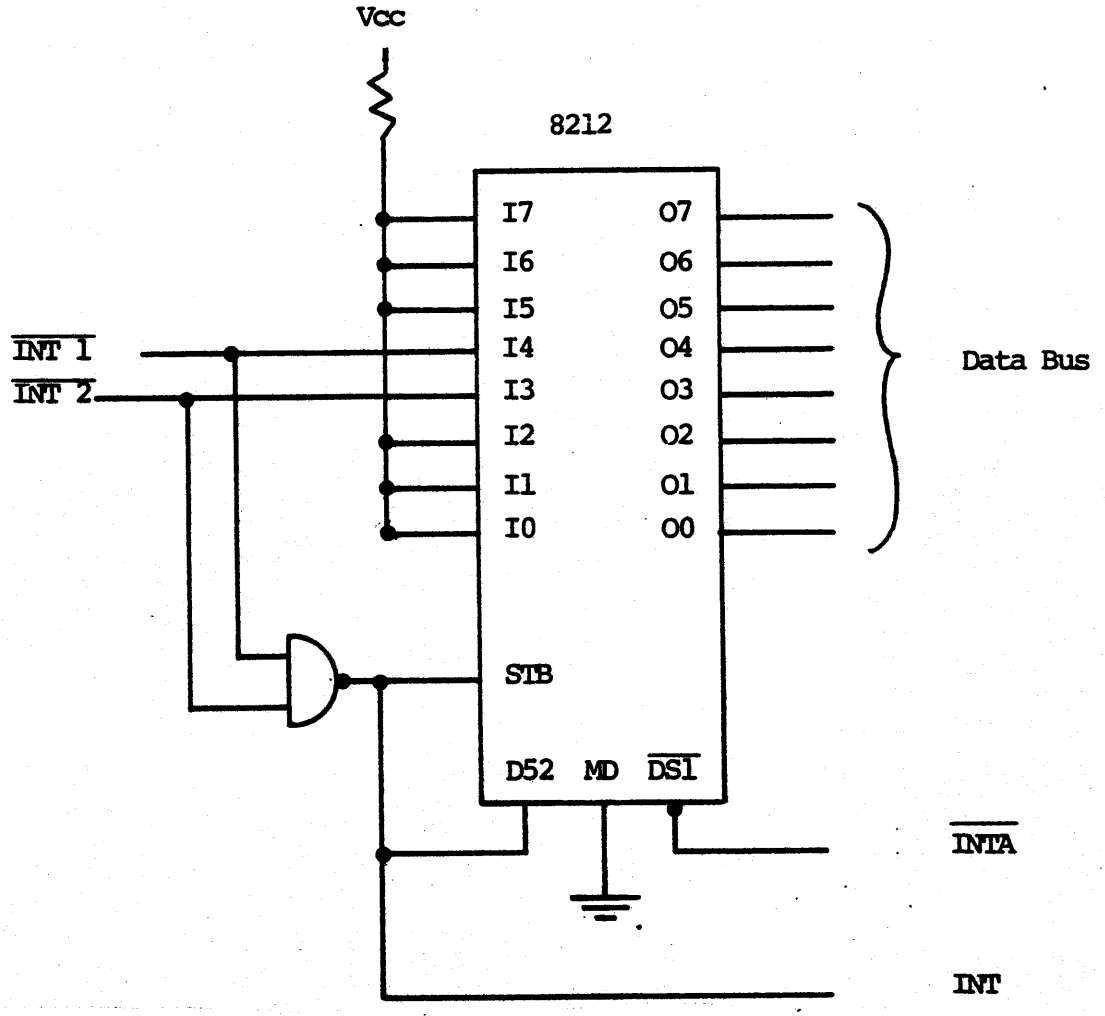


Note 1 : If the external interrupt is a continuous signal it should provide the interrupt to the 8080. If it is a pulse, the 8212 can store it and provide the interrupt request.

Note 2 : The configuration shown places RST 6 (F7) on the data bus during INTA.

RESTART PORT WITH 8212

Figure 8-26



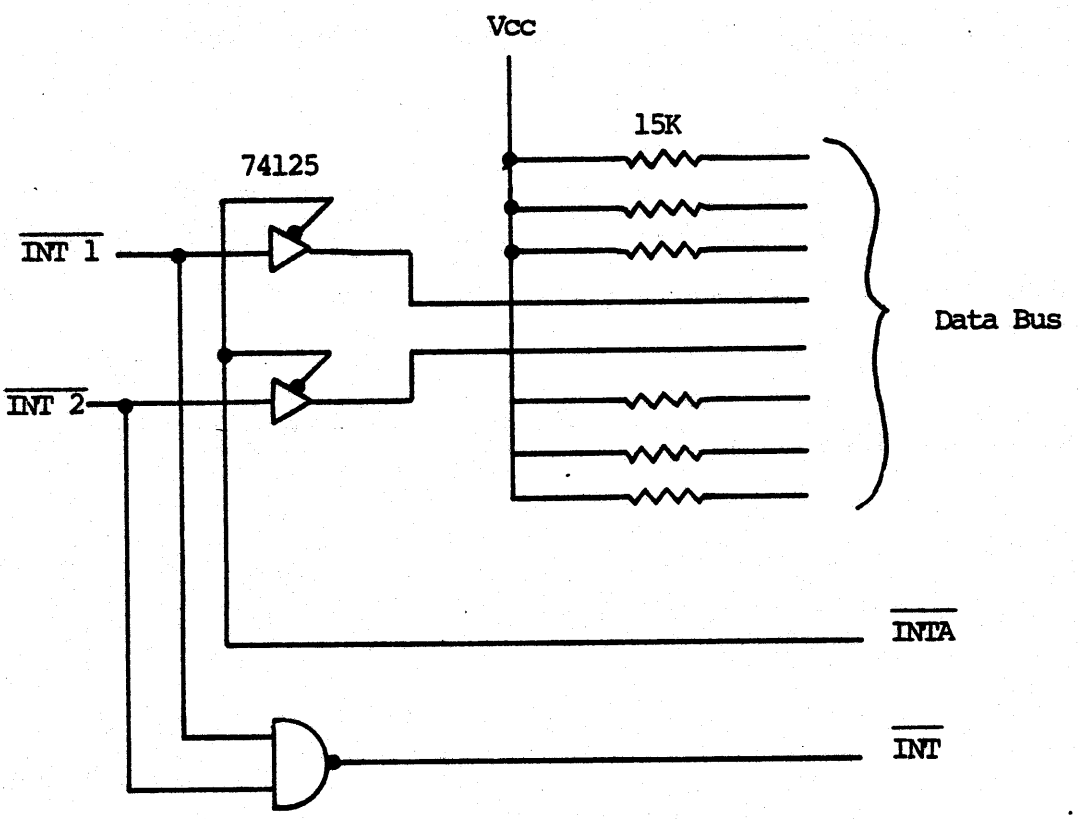
FUNCTION

INPUTS:	NONE	INT 1		INT 2		BOTH		OTHER
$\overline{\text{INT 1}}$	1	0	0	1	1	0	0	1
$\overline{\text{INT 2}}$	1	1	1	0	0	0	0	1
$\overline{\text{INT A}}$	1	1	0	1	0	1	0	0
OUTPUTS:								
INT	0	1	1	1	1	1	1	0
BUS	Z	Z	EF	Z	F7	Z	E7	Z

Z = High Independance State
EF = RST 5
F7 = RST 6
E7 = RST 4

VECTORED RESTART PORT

Figure 8-27



VECTORED INTERRUPT USING RESISTORS

Figure 8-28

8.4.2.4 The 8228 Generated RST7

For systems that need only one kind of interrupt the 8228 can by itself create an RST7 instruction (CALL 0038) without external logic. If its INTA output is pulled up to +12 volts (through a 1K resistor), the 8228 recognizes this as an input (instead of an output) that commands it to place FF on the data bus in response to INTA from the 8080. This avoids any need for external logic to provide that instruction.

8.4.2.5 HALT instruction

Many microprocessor based systems have no function to perform while they are waiting for input. The program can be made to cycle indefinitely in one place with:

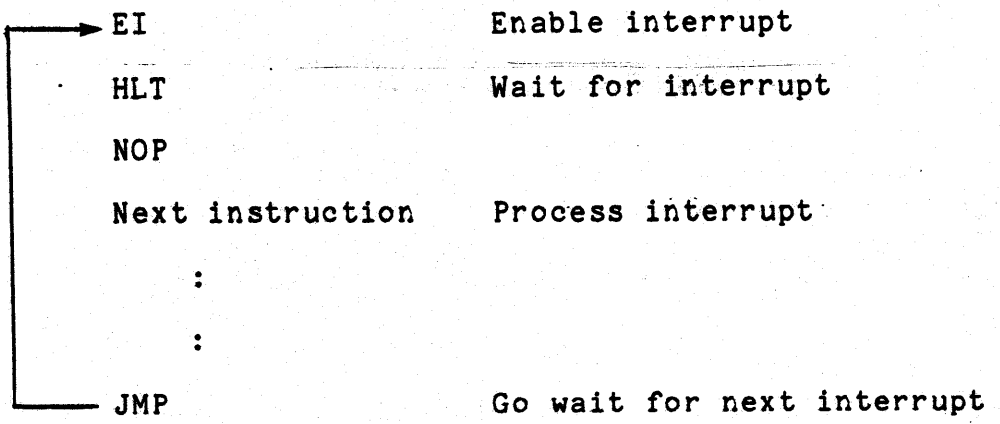
```
8200 C3        JMP 8200
     00
     82
```

Now an interrupt with an RST instruction will call an interrupt service routine which handles all of the processing, and the return will go back to 8200. An alternative is the instruction:

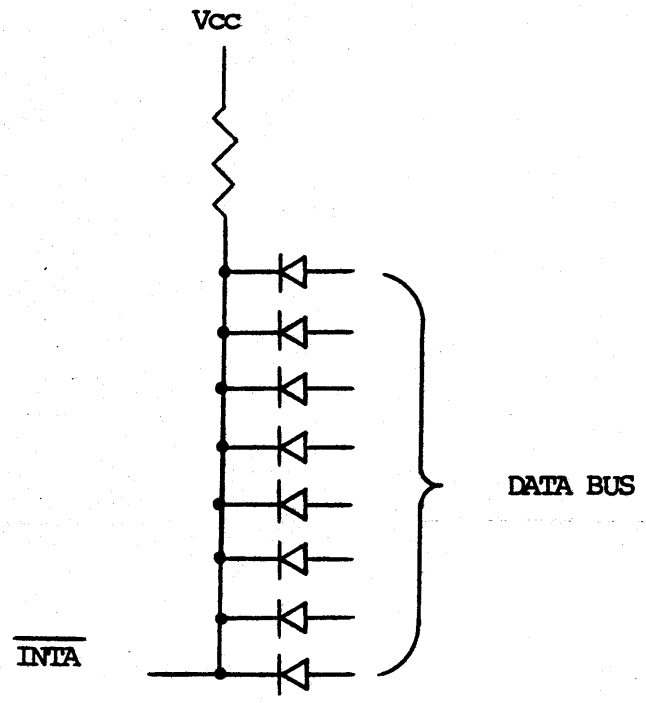
```
76   HLT        Halt at this address until
                an interrupt occurs.
```

When this instruction is executed the processor enters a WAIT state until an interrupt occurs. Now if $\overline{\text{INTA}}$ is OR'ed with $\overline{\text{MEMR}}$, the next instruction in the program will be read and program execution will continue:

Program Flow:



This avoids the need for placing a special instruction on the data bus. Note, however, that the byte following HLT will be read twice, because the program counter is not incremented during $\overline{\text{INTA}}$. Therefore this instruction should be NOP. Alternatively, a NOP instruction may be placed on the data bus through diodes during $\overline{\text{INTA}}$.



This scheme places all the capacitance of the diodes on the data bus at all times, so it must be used with care.

8.4.3 The MTS Interrupt System

When the MTS executes your program in STEP mode (whether it was started with the STEP or RUN key) an interrupt is generated by the MTS hardware as each of your instructions is executed, causing an RST7 that calls the monitor program. The monitor then operates as an interrupt service

subroutine, which we will describe later. The hardware involved will show something of the timing relationship of an interrupt system.

MTS uses the 8228 generated RST7, so there is no external logic to place an instruction on the data bus. R62, located between the 8080 and the 8224, pulls INTA up to +12 volts to cause the RST7 in response to INTA.

Figure 8-29 shows the interrupt circuit and timing. Recall that an 8080 instruction cycle comprises one to five machine cycles. Each machine cycle includes three to five clock periods, or states. The first state of each machine cycle is identified by a status strobe signal from the 8224; this is shown in the timing diagram as STSTB. During the first state of every machine cycle the 8080 sends out signals on the data bus to identify the operations to be carried out. These are latched by the 8228 and provide the information to generate all the control signals - $\overline{\text{MEMR}}$, $\overline{\text{MEMW}}$, I/O $\overline{\text{READ}}$, I/O $\overline{\text{WRITE}}$ and $\overline{\text{INTA}}$. Status strobe identifies the time at which the status data can be latched, both by the 8228 and any other device that needs it - such as the MTS interrupt system.

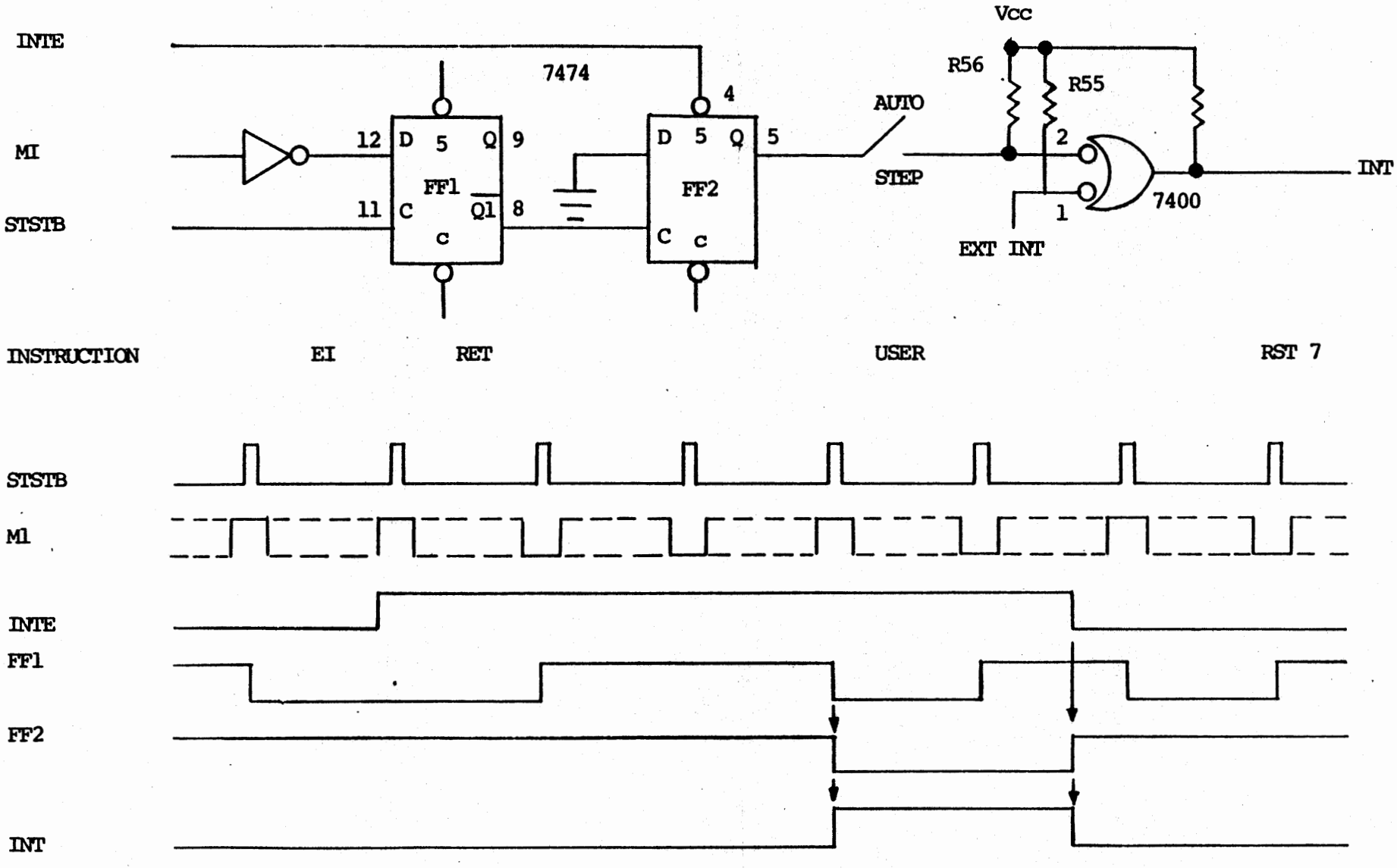


FIGURE 8-29

MTS STEP MODE INTERRUPT CIRCUIT AND TIMING

Figure 8-24

One of the data bits in the status byte is M1. This identifies the first (or only) machine cycle in each instruction cycle. In the timing diagram of Figure 8-29 we show four instructions: EI, RET, an unidentified instruction of the user's program, and RST7. EI is a single cycle instruction; RET and RST7 are three cycle instructions, and we have chosen a two cycle instruction (MVI A, for instance) as the user instruction. M1 is seen as a high signal during status strobe at the start of each instruction cycle.

Our timing diagram starts with the monitor in control; INTE has been set low at entry to the monitor. Just before returning to the user's program, the monitor includes an EI instruction which sets INTE high, just as the next instruction begins.

The MTS hardware includes a dual D flip flop (7474, or NEC 214). This device has four inputs to each flip flop: Set, Clear, Data, and Clock. The Set and Clear inputs force the flip flop to 1 or 0 independent of the clock. They are active low signals and all but one are unused in this circuit. In the absence of Set and Clear, the flip flop stores the data input at the moment when the clock input changes from low to high. It ignores the state of these inputs except at that transition. Thus FF1, high at the start of the timing diagram, copies the inverted M1 signal at the start of the EI instruction. It stays low when M1 occurs again with status strobe at the start of RET, but when the second machine cycle of RET begins M1 stays low, so the inverted M1 is clocked into FF1 and sets it high. Thus FF1 will be low continuously for single instructions but will always become high at the start of a second

machine cycle. It follows this sequence whether the interrupts are enabled or not.

INTE is taken into the set input of FF2. Since this is an active low input that overrides the clock, FF2 is high as long as INTE is low. When INTE is set high by the EI instruction, FF2 comes under control of its data and clock inputs. The D input is tied to ground, so the rising edge of the clock will set it low. FF2 receives its clock from the inverted output of FF1, so when FF1 goes low with INTE high, FF2 goes low. When the MTS is operating in AUTO mode, with interrupts enabled, FF2 will always be low; but in STEP mode FF2 will almost always be high. Just at the end of the return from the monitor, when the first machine cycle of your instruction resets FF1, FF2 goes low to create another interrupt. If you watch this with an oscilloscope you can see why operating in STEP mode slows your program so much: most of the time is spent in the monitor. The AUTO/STEP toggle switch simply disconnects FF2 from the INT input.

The gate permits you to enter a separate interrupt for your own purposes. This is in the 7400 or NEC 201 chip below the 8228, and the right hand side of R55 is connected to it. When the mode switch is at AUTO, either input to the gate can be used as an external input. To experiment with an external interrupt, connect a test clip to R55, set the mode switch to AUTO, and either use a program you have loaded or enter a trivial program such as:

```
8200        C3 JMP 8200
01          00
02          82
```

and start the program with STEP (not with RUN). Now an external interrupt will return to the monitor. Touch your test lead to ground to generate the interrupt.

Also experiment with the DI and EI instructions. Enter this:

```
8200 F3     DI
  01 3C     INR A
  02 C2     JNZ 8201
  03 01
  04 82
  05 FB     EI
  06 00     NOP
  07 C3     JMP 8200
  08 00
  09 82
```

The DI instruction prevents the external interrupt from being effective until the EI at 8205 enables interrupts again. When you operate this, again using STEP to initiate it but in AUTO mode, your external interrupt with the test lead will always return you to the monitor at address 8207. The interrupt cannot affect the instruction immediately following the EI.

You will find that if you try to operate this program in STEP mode, the monitor will not interrupt it. It is a requirement of the MTS interrupt logic (not of the 8080) that the interrupt is not generated until a multi-cycle instruction has been completed and the next instruction has started. In normal operation this allows the monitor's return and one user instruction to be executed before the monitor is called again. With this test program the single-cycle NOP does not create an interrupt. The JMP is executed, the monitor initiates the interrupt, but the instruction being processed at that time is Disable Interrupt, which makes the interrupt ineffective even though it had already been received. If you change the instruction at 8206 from NOP to:

```
8206 77        MOV  M,A
```

or any other instruction requiring two memory cycles, then the interrupt will occur as the JMP is executed and the monitor will be called before DI is executed at 8200.

8.5 INTERRUPT SERVICE ROUTINES

When an interrupt occurs the interrupt instruction generally calls an interrupt service routine. This is a subroutine, but it has two special requirements. It must:

a) Preserve the environment.

b) Find out why it was called.

8.5.1 Preserving the Environment

An interrupt service routine does not use the registers to exchange data with a calling program. On the contrary, it must preserve the contents of all registers and flags, and restore those contents before returning to the instruction that was interrupted. The interrupted program module makes no special provisions for the interrupt, and except for the time taken by the interrupt service its functions must not be interfered with. It may be interrupted but not disrupted, and the service routine must be transparent.

The first several instructions in any interrupt service routine are almost invariable PUSH instructions to save the registers:

```
PUSH    PSW     Save A and flags
PUSH    B       Save B,C
PUSH    D       Save D,E
PUSH    H       Save H,L
```

The routine can now use all of the registers to perform its functions - typically input and/or output. When finished it restores the environment that existed before the interrupt by popping the registers in reverse order:

```
POP     H
POP     D
POP     B
POP     PSW
EI
RET
```

Remember that the interrupt itself disabled the interrupt system, so to restore the environment, allowing for another interrupt, there must be an EI in the service routine. If this is placed immediately before the return, it is guaranteed that the return will be executed. Placing it earlier in the interrupt routine will allow another interrupt to interrupt the interrupt routine! This is sometimes done, but usually with priority interrupt systems (which are discussed below), and requires special consideration. Many interrupt service routines cannot tolerate being interrupted. This is the case with the MTS monitor, for instance. Other program modules may also be intolerant of interrupts. They must be protected by a DI instructions, and at some point must also include EI.

8.5.2 Identifying the Source of the Interrupt

Commonly a system will have only one generalized interrupt service routine to handle a variety of interrupts. For instance an 'intelligent' communications terminal might be interrupted by a transmit next character signal, or by an operator's keystroke. Hardware can be provided to call different interrupt service routines, as we showed earlier. This adds cost and introduces the problem of simultaneous interrupts from different sources. If there is not a severe time constraint it is usually less costly to use programmed I/O rather than providing for vectored priority interrupts. We will define these terms but otherwise in this course will not be concerned with them.

8.5.2.1 Vectored Interrupt Systems

This is a combination of hardware and software such that each different source of interrupt calls a service routine specific to the device that created the interrupt.

The prior discussion of RST instructions showed how vectored interrupts can be created by placing different instructions on the data bus in response to INTA. Other schemes are possible, For instance the program may store the address of a module to process the next interrupt, if a particular sequence is expected.

8.5.3 Priority Interrupt Systems

A combination of hardware and software guaranteeing that an interrupt from one source is given priority over another; the higher priority can interrupt the lower, or, if they arrive simultaneously, will be handled first. This can be extended to many levels if necessary.

Specific hardware devices (LSI chips) are available to perform this function. In combination with software the 8255 can also create a priority interrupt system.

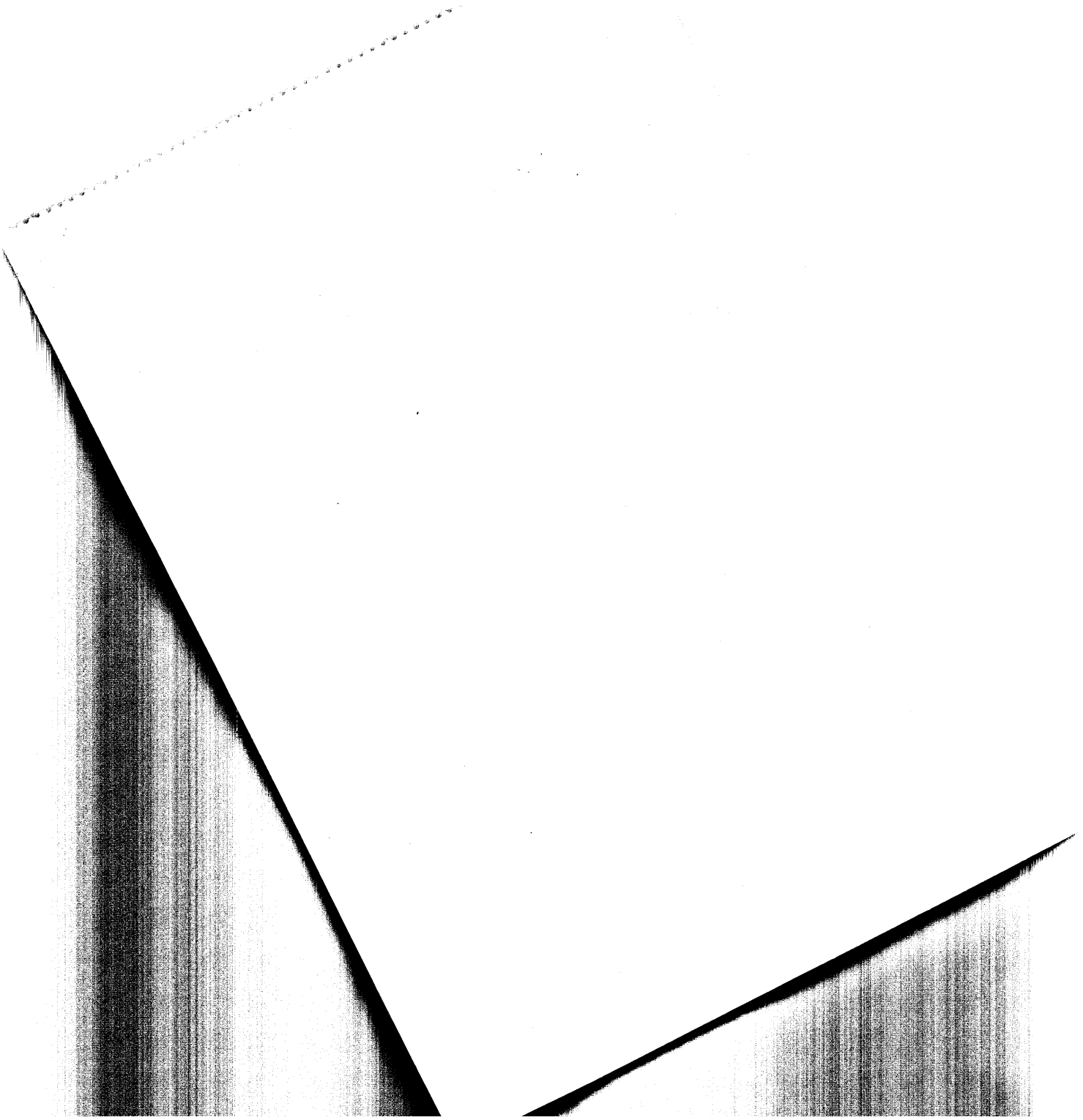
8.5.4 Timed Interrupt Systems

Systems that need to know the time of day often use a hardware counter, operating on the computer's crystal clock, to generate an interrupt once every millisecond (or any other desirable interval). An interrupt service routine increments a 'clock' address in memory. The service routine may also conduct I/O operations at this time, checking each input port to see if any service is needed. This scheme provides frequent service to all I/O ports without requiring each I/O device to create interrupts, and is called 'polling'.

RST 5 & RST 6 Interrupts

Call 0028

0050



8.6 USING INTERRUPTS WITH MTS

The MTS can readily be modified to accept two vectored interrupts, RST5 and RST6. The actual interrupts call 0028 and 0030, but the monitor was programmed to contain jump instructions to your program area:

```
0028    JMP  8228
0030    JMP  8230
```

Thus you can place interrupt service routines at those locations and provide hardware to enter the restart instructions required. To do this you must disable the RST7 insertion feature of the 8228 by removing the 1K pullup resistor from INTA, and inserting the desired RST by one of the schemes described earlier. Note that having done this, you must also provide the RST7 (FF) for MTS generated interrupts.

You can avoid the hardware requirement altogether by entering a jump address in the monitor's memory area. This will cause a jump to your interrupt service routine when an RST7 is received. There are significant restraints on the use of this system. Nevertheless it gives valuable experience with interrupt handling with no hardware except a clip lead. In the following exercise we will develop an interrupt service routine that uses the RST7.

8.6.1 Interrupt Service Routine Exercise

The program to be developed represents a timed interrupt system. The interrupt service routine has two tasks:

a) Increment a multi-byte counter in memory.

b) Read the keyboard. When a key is pressed convert it to hexadecimal and store in memory; when the key is released set an indicator in memory.

The control program will do the following:

a) Display the contents of the counter that is being updated by the interrupt service routine.

b) Test the service routine's indicator for a new input, and when one is found clear the indicator and process the input.

c) In response to a hex key input shift the digit into a four byte store in memory. Display the input data instead of the counter.

d) In response to a command, process the data as follows:

CLR Clear the input

MEM Replace the contents of the counter with the input data and clear the input.

REG Add the input data to the contents of the counter and clear the input.

BRK Call the monitor

NEXT Display the input

RUN Display the counter

After any command key except NEXT has been processed the counter display will be resumed. We will define the data memory area as follows:

Address	Contents
8280 - 8285	Counter
8286	Key indicator
8287	Key value
8288 - 828F	Not assigned
8290 - 8293	Key input data

The solution given to this problem uses the following program memory assignments:

Address	Module
8200 - 821F	Main Control Program
8220 - 822F	Display subroutine
8230 - 824F	Interrupt service
8250 - 825F	Copy and Clear subroutine
8260 - 828A	Keyboard subroutine
828B - 829F	KTST and KYIN
82A0 - 82CF	Key data processing
82D0 - 82EF	Command processing
82F0 - 82FF	Addition subroutine

The key indicator will be set to FF when the key is released; keyboard scanning will be inhibited until the control program clears the indicator to 00. Other states of the indicator may be used by the interrupt routine for its own purposes and must not be altered by the control program. The key value is available to the program only while the indicator is set to FF.

We will initially develop the counter function in the interrupt service routine, reserving space for a call to a keyboard input subroutine. In the control program we will provide for initialization of all the storage (including the counter and keyboard memory), for display of the counter, and for loading the monitor jump address. Again, the keyboard functions will be omitted initially.

When we use RST7, there are two constraints which must be observed to make the new interrupt service routine function with the monitor. First, the control program (or some subroutine) must store the low byte of the jump address at memory location 83D4. (Only the low byte is stored; the jump is always to page 82xx.) This is dictated by the fact that the monitor loads 83D4 with a jump address after normal servicing of an RST7 interrupt. As the monitor lives in ROM memory, it must put all of its computed address and data in RAM, hence the 83D4 usage. We want the entry from the monitor to be 8238, so this program segment is needed:

```
3E    MVI  A,38
38
32    STA  83D4
D4
83
```

This must be executed each time the program is started with the RUN or STEP key. During program debugging it is generally most convenient to have it inside a repetitive loop. Note that storing the jump address disables the STEP and breakpoint functions of the monitor. When you first try the program, instead of the STA instruction use a call to the interrupt service routine:

Final	Debug
32 STA 83D4	CD CALL 8230
D4	30
83	82

This will allow you to step through to check program flow. Write the main program, a subroutine to clear part of memory, and a display routine. Coding solutions are given in Figures 10-30 through 10-32.

INTEGRATED COMPUTER SYSTEMS MICROCOMPUTER TRAINING SYSTEM CODING SHEET

ADDR	CODE											
8200	21		LXI	H,	8	3	8	0				
01	80											
02	83											
03	0E		MVI	C,	2	0						
04	20											
05	CD		CALL		C	L	R	M				
06	50											
07	82											
08	3E		MVI	A,	3	8						
09	38											
0A	CD		CALL		I	N	T	S				
0B	30											
0C	82											
0D	21		LXI	H,	8	3	8	0				
0E	80											
0F	83											
8210	CD		CALL		D	I	S	P				
11	20											
12	82											
13	C3		JMP		8	2	0	8				
14	08											
15	82											
16	00		NOP									
17	00											
18	00											
19	00											
1A	00											
1B	00											
1C	00											
1D	00											
1E	00											
1F	00											
8220												
21												
22												
23												
24												
25												
26												
27												
28												

FIGURE 8-30

ADDR	CODE			
8220	11		LXI D,83FF	(DE) ← Display Add.
1	FF			
2	83			
3	7E		MOV A,M	(A) ← Data
4	23		INX H	Address next byte
5	CD		CALL DBY2	Call monitor
6	95			display subroutine
7	02			
8	7B		MOV A,E	Use low byte of
9	FE		CPI F8	display address
A	F8			as counter
B	D2		JNC 8223	
C	23			
D	82			
E	C9		RET	
F				
8 0			NOTE: ENTER DISP WITH	
1			MEMORY ADDRESS IN H,L.	
2			FOUR BYTES ARE	
3			DISPLAYED AS EIGHT	
4			DIGITS.	
5				
6			ALL REGISTERS ARE USED	
7				
8				
9				
A				
B				
C				
D				
E				
F				
8230	C9		RET	Dummy for INTS
1				
2				
3				
4				
5				
6				
7				
8				

FIGURE 8-31

CLEAR MEMORY SUBROUTINE CLEAR

ADDR	CODE			
8250	36		MVI	M,00
1	00			
2	23		INX	H
3	0D		DCR	C
4	C2		JNZ	8251
5	50			
6	82			
7	C9		RET	
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				

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

FIGURE 8-32

8.6.2 Writing the Interrupt Service Routine

The monitor jumps into your interrupt service routine with the registers and return address already saved. You must provide two entries: one for a CALL during debugging and one for the monitor entry, and both must result in the registers being saved in exactly the same way. Your entry at 8230 should have the same instructions that the monitor has.

Your program counter, as displayed by the monitor, is actually the return address stored by RST7. To make it readily accessible the monitor extracts it from the stack and stores it in a fixed address.

```
0038  F3    DI    Disable Interrupt- to permit entry
                  by CALL or programmed RST7.

0039  E3    XTHL  (ST) <-> (HL)
                  Place (HL) in the stack and return
                  address in HL.

003A  22    SHLD  PCADDR
003B  DA          (83DA) <- low return address
003C  83          (83DB) <- high return address

003D  C5    PUSH  B Save registers
003E  D5    PUSH  D
003F  F5    PUSH  PSW
```

After this sequence the stack contains:

83CB	Flags
CC	(A)
CD	(E)
CE	(D)
CF	(C)
D0	(B)
D1	(L)
D2	(H)

The return address is at 83DA and 83DB, not in the stack. The addresses will be different if you enter the monitor from a subroutine because 83D2 and 83D1 will contain a return address, but the sequence of storage will be the same.

You should duplicate the instructions shown above at 8230 through 8237. This will allow three ways of using the subroutine: by an RST6, which will enter the monitor and jump to 8230; by CALL 8230; or by placing the jump address 38 in memory location 83D4 and allowing monitor interrupts with RST7 to jump there. You cannot yet step through the subroutine because your return address is stored in 83DA and 83DB, and will be destroyed by the monitor. For debugging it is wise to overcome this by recovering the return address and pushing it into the stack:

```
8230   F3      DI
  31   E3      XTHL
  32   22      SHLD PCADDR
  33   DA
  34   83
  35   C5      PUSH  B
  36   D5      PUSH  D
  37   F5      PUSH  PSW
  38   2A      LHLD  PCADDR
  39   DA
  3A   83
  3B   E5      PUSH   H
  3C   FB      EI
```

Now the EI at 823C enables monitor interrupts, and you can step through the subroutine. Insert any multi-byte instruction after EI; leave some space (NOP's) and then create the return segment of the interrupt service routine, starting at 8248. The complete coding is shown in Figure 10-33.

MICROCOMPUTER TRAINING WORKBOOK

CHAPTER 9

DATA FORMAT

9. DATA FORMAT

In Chapter 8 you used only discrete inputs and outputs, each bit being essentially independent of all others. An output at C4, C5 or C6 selects a column of the keyboard; an input at any bit of port A comes from one key. The timing of inputs and outputs, apart from their sequence, has no meaning. We will now consider parallel I/O, where a data byte representing a number is transferred, and serial I/O, where the timing of signals carries information.

9.1 PARALLEL INPUT/OUTPUT

Clearly the 8255 data ports are principally intended for 8-bit, parallel data transfer. Such data might come from a paper tape reader, an analog to digital converter, another computer, a keyboard that includes built-in scanning and decoding, or a communications device that includes serial to parallel conversion. A usual characteristic of such devices is that they generate a strobe signal indicating that an input byte is ready for the computer. When port A or port B of the 8255 is programmed to input mode 1, it uses some bits of port C to handle the strobe and give an interrupt to the 8080, and responds with an acknowledgement to the input device when the computer has accepted the data. Some input devices are designed to demand such an acknowledgement before entering the next byte, or to recognize an error condition if it is not received.

9.1.1 Paper Tape Reader Example

Figure 9-1 shows bit assignments and timing for mode 1 input through an 8255. Consider how this would be used with a high-speed paper tape reader.

SILICON GATE MOS 8255

Input Control Signal Definition

$\overline{STB}$ (Strobe Input)

A "low" on this input loads data into the input latch.

IBF (Input Buffer Full F/F)

A "high" on this output indicates that the data has been loaded into the input latch; in essence, an acknowledgement. IBF is set by the falling edge of the STB input and is reset by the rising edge of the RD input.

INTR (Interrupt Request)

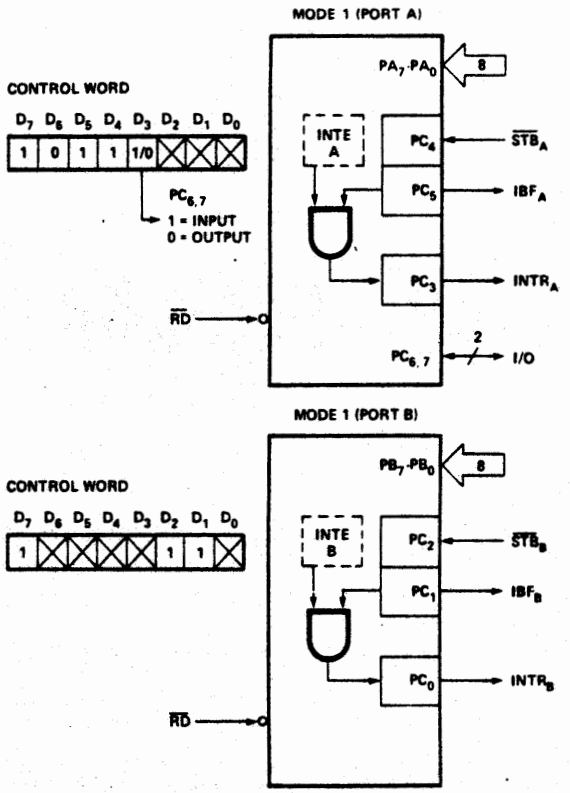
A "high" on this output can be used to interrupt the CPU when an input device is requesting service. INTR is set by the rising edge of $\overline{STB}$ if IBF is a "one" and INTE is a "one". It is reset by the falling edge of $\overline{RD}$. This procedure allows an input device to request service from the CPU by simply strobing its data into the port.

INTE A

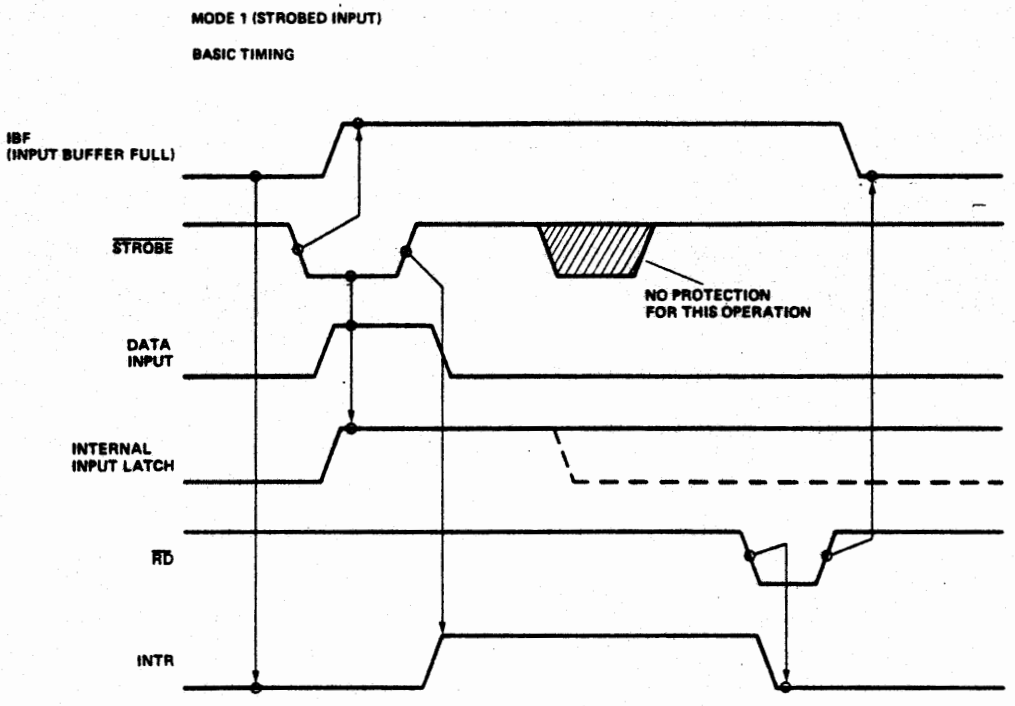
Controlled by bit set/reset of PC_4.

INTE B

Controlled by bit set/reset of PC_2.



Mode 1 Input

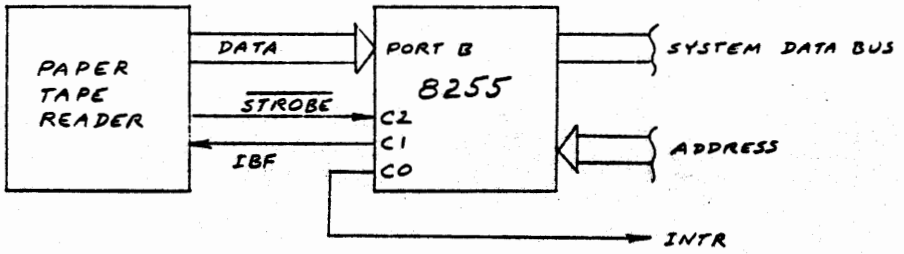


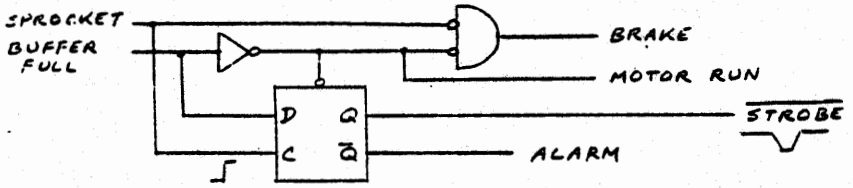
Basic Timing Input

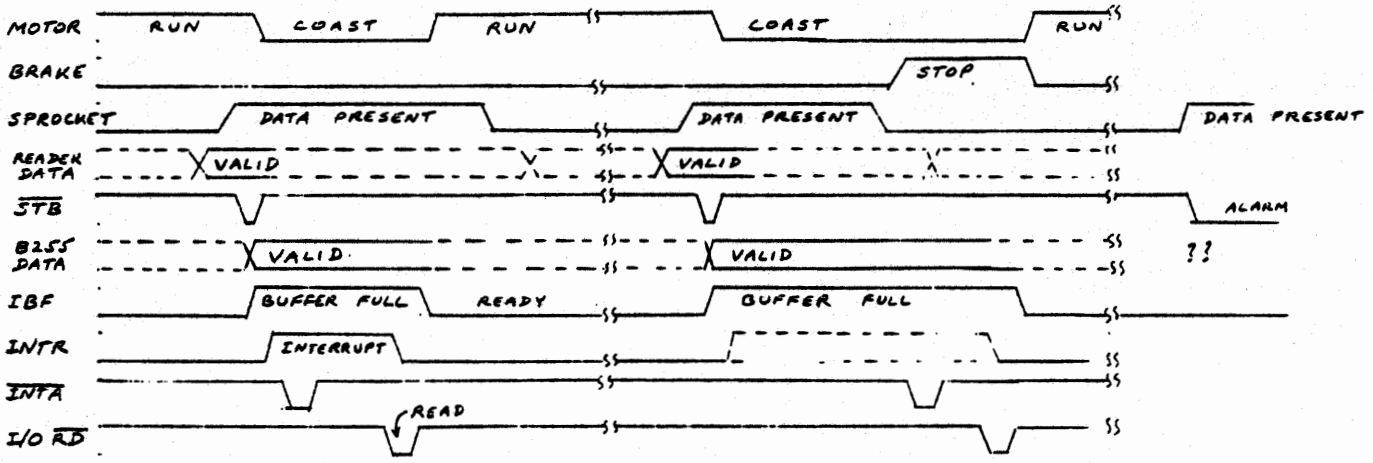
FIGURE 9-1

The photoelectric reader senses holes in the paper tape. The sprocket hole (which is present at every character position even though there may be no other holes) is sensed to indicate that the data holes are in position to be read. The sprocket hole signal provides the strobe to latch data into the 8255. The logic and timing diagram of Figure 9-2 shows the sprocket hole signal clocking a D flip-flop. The IBF signal is taken into the D input. Since it is (presumably) low, indicating that the buffer is ready to take data, the flip-flop is reset. Its output is the strobe signal; this enters the data into the 8255 data latch and sets IBF high. IBF high sets the D flip-flop through the asynchronous set input, ending the strobe pulse and latching the data. The end of strobe sets the 8255's interrupt request output. The 8080 acknowledges the interrupt, calls the interrupt service routine, and reads the data from the 8255.

The act of reading (I/O RD) resets IBF, indicating that the buffer is again available. All of this is normally accomplished while the sprocket hole is still visible to the reader. (At 1000 characters per second it lasts for about 200 microseconds, time enough for a reasonable interrupt service routine). While the IBF signal is high the reader's motor is allowed to coast; when IBF is reset it runs again.







HIGH SPEED PAPER TAPE READER INTERFACE

FIGURE 9-2

In the second segment of the timing diagram the CPU is not available to read the data promptly. Either it has disabled the 8255 interrupt, or its program has disabled all interrupts. The IBF signal stays high beyond the sprocket hole signal. This signals the paper taper reader that, although the 8255 has accepted and latched the present character, it may not be ready in time for the next. The mechanism now applies a brake to stop paper motion before the next character. When the data are finally accepted by the CPU by an I/O Read, the motor can run again.

The final segment of the timing diagram shows a failure: IBF is not set by the strobe (perhaps the 8255 has been reprogrammed). Strobe goes low but fails to rise again. This can generate a visible alarm signal to indicate a loss of data.

9.1.2 Computer to Computer Interface

Some applications overburden a microprocessor, particularly when two or more tasks require fast interrupt service response. One solution, of course, is to use a faster or more powerful computer such as a bipolar bit-slice machine, whose instruction time may be a small fraction of the 8080's. Often it is more economical to divide the task between two microprocessors. They will then need to communicate with each other. This can be handled in three ways:

a) Through input/output ports

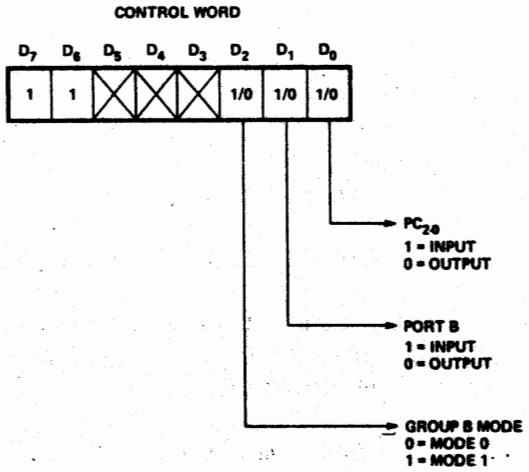
b) Direct memory access

c) Memory sharing

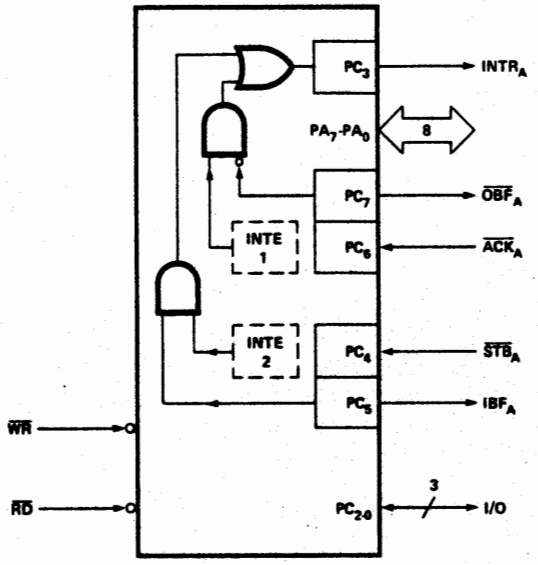
9.1.2.1 I/O Port Interface

One computer can write an output to a data latch (such as the 8212) and create an interrupt to another, which can then read the data through a similar port or through a tri-state buffer. The 8255 has the ability to operate port A as a tri-state, bi-directional bus interface. This avoids the need for a second device between the systems. The 8255 is connected as an I/O port to one 8080 (the master) and port A is connected to the data bus of the other 8080 (the slave). Six bits of port C are used for handshaking between the processors; the slave needs additional gating to enable port A to interact with its bus.

Figure 9-3 defines mode 2 of the 8255, and Figure 9-4 shows the connection between two processors through the 8255. The master writes and reads ports A and B as in any other use of the device. The slave is connected to port A. It can address the 8255 through an I/O Read or Write with a port address that gives it a select signal. I/O Write and select generate an $\overline{\text{STB}}$ input to C4, latching the slave's data bus content into the port A data latch. This much of its behavior is similar to mode 1 input. I/O Read and select generate an $\overline{\text{ACK}}$ input to C6, which places the data latch content onto the port A outputs and so onto the slave's data bus. Otherwise port A is in the high impedance state. $\overline{\text{IBF}}$ (port C5) goes low when the input buffer is empty. $\overline{\text{OBF}}$ (port C7) goes low when the output buffer is full. Either of these will generate an interrupt to the slave CPU to indicate that the 8255 needs service. These two signals may also be taken to other input ports of the slave, so that it can determine which kind of service is needed.



Mode 2 Control Word



Mode 2

Operating Modes

Mode 2 (Strobed Bi-Directional Bus I/O)

This functional configuration provides a means for communicating with a peripheral device or structure on a single 8-bit bus for both transmitting and receiving data (bi-directional bus I/O). "Handshaking" signals are provided to maintain proper bus flow discipline in a similar manner to Mode 1. Interrupt generation and enable/disable functions are also available.

Mode 2 Basic Functional Definitions:

- Used in Group A only.
- One 8-bit, bi-directional bus Port (Port A) and a 5-bit control Port (Port C).
- Both inputs and outputs are latched.
- The 5-bit control port (Port C) is used for control and status for the 8-bit, bi-directional bus port (Port A).

Bi-Directional Bus I/O Control Signal Definition

INTR (Interrupt Request)

A high on this output can be used to interrupt the CPU for both input or output operations.

Output Operations

$\overline{OBF}$ (Output Buffer Full)

The $\overline{OBF}$ output will go "low" to indicate that the CPU has written data out to Port A.

$\overline{ACK}$ (Acknowledge)

A "low" on this input enables the tri-state output buffer of Port A to send out the data. Otherwise, the output buffer will be in the high-impedance state.

INTE 1 (The INTE Flip-Flop associated with $\overline{OBF}$)

Controlled by bit set/reset of PC_6.

Input Operations

$\overline{STB}$ (Strobe Input)

A "low" on this input loads data into the input latch.

IBF (Input Buffer Full F/F)

A "high" on this output indicates that data has been loaded into the input latch.

INTE 2 (The INTE Flip-Flop associated with IBF)

Controlled by bit set/reset of PC_4.

FIGURE 9-3

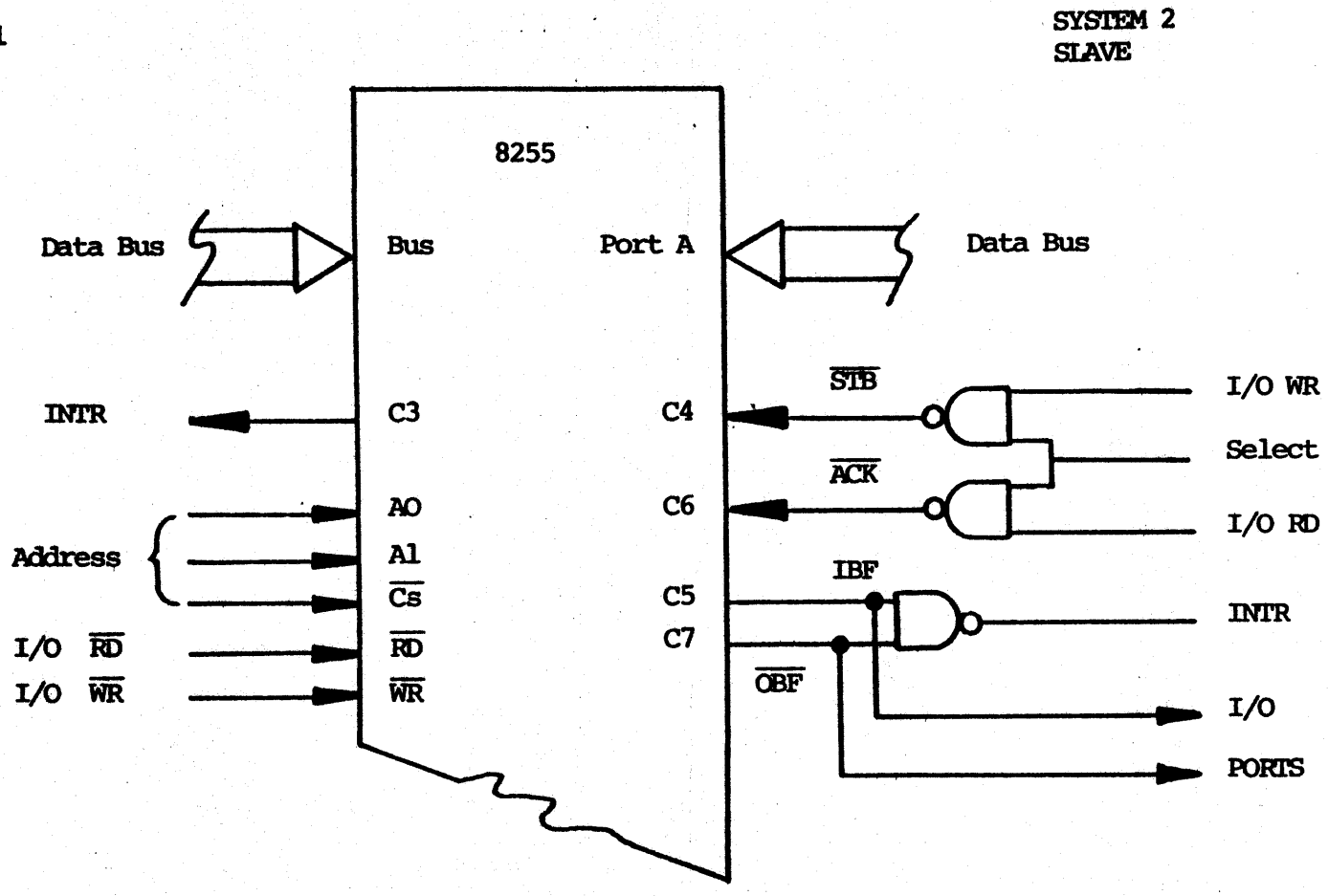


FIGURE 9-4

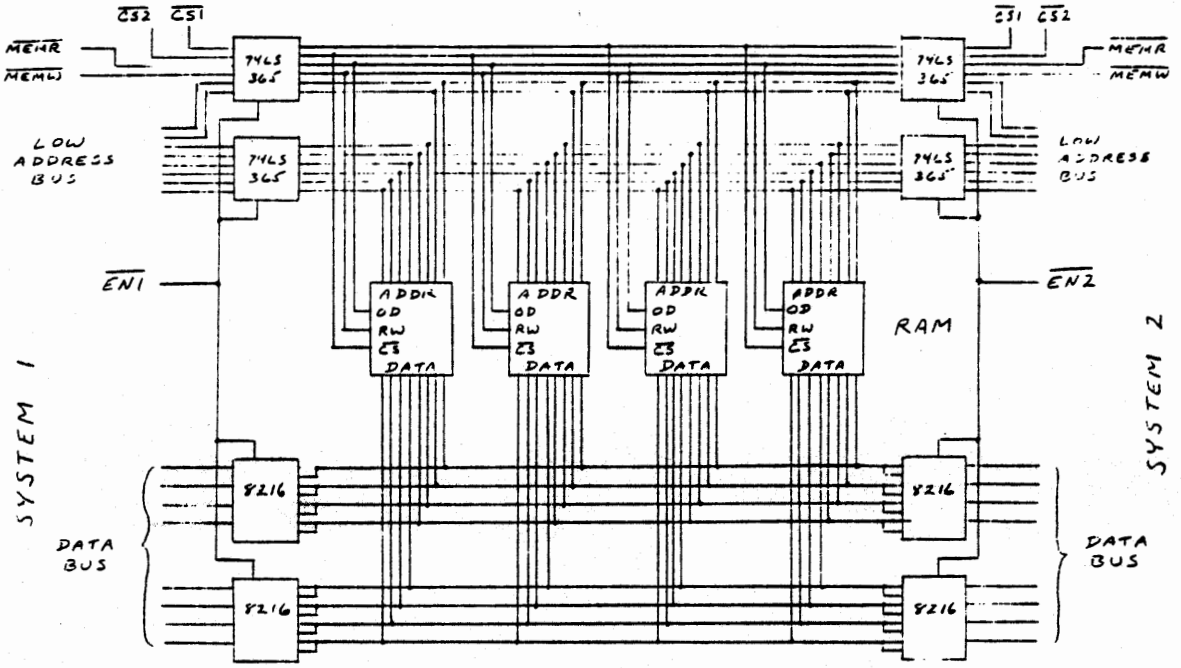
Figure 8-29

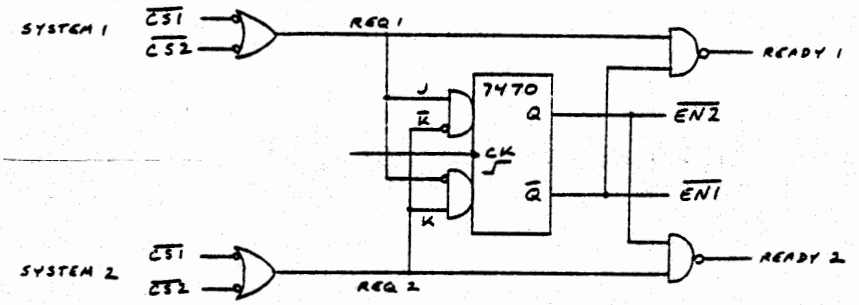
9.1.2.2 Direct Memory Access Interface

Clearly a DMA channel can be established between two processors. It may be handled by I/O ports with one processor given direct memory access to the other processor, or there may be separate hardware to operate DMA to both processors. This subject will not be covered further since DMA has been extensively discussed.

9.1.2.3 Shared Memory

A powerful but somewhat expensive technique for interfacing two processors is shown in Figure 9-5. Some part of memory is fully accessible to both processors, and either can address it at any time. As the figure shows, ten logic chips are needed to share 512 bytes (four chips) of RAM. The interesting point is the ready access each processor has to the data: it is simply addressed like any other part of memory. The timing diagram in Figure 9-5 shows what happens if both processors address the memory at the same time. Whoever gets there first has immediate access, while the other must enter a WAIT state for one clock period. If the first processor uses the memory for two consecutive reads or writes (with an INR M or SHLD instruction, for instance), the other must wait for two clock periods. It is guaranteed access within one full instruction cycle, however, unless the other processor is executing a program stored in the shared memory. (That operation is not unreasonable. A master CPU might pass some lengthy task to a slave by loading a program into the shared memory).


CS2 CS1
MEMR
MEMW
74LS 365
LOW ADDRESS BUS
EN1
ADDR
OD
RW
CS
DATA
RAM
SYSTEM 1
SYSTEM 2
8216
DATA BUS
CS1 CS2
EN2


SYSTEM 1
CS1
CS2
REQ 1
READY 1
7470
J
K
CK
Q
EN2
EN1
SYSTEM 2
REQ 2
READY 2

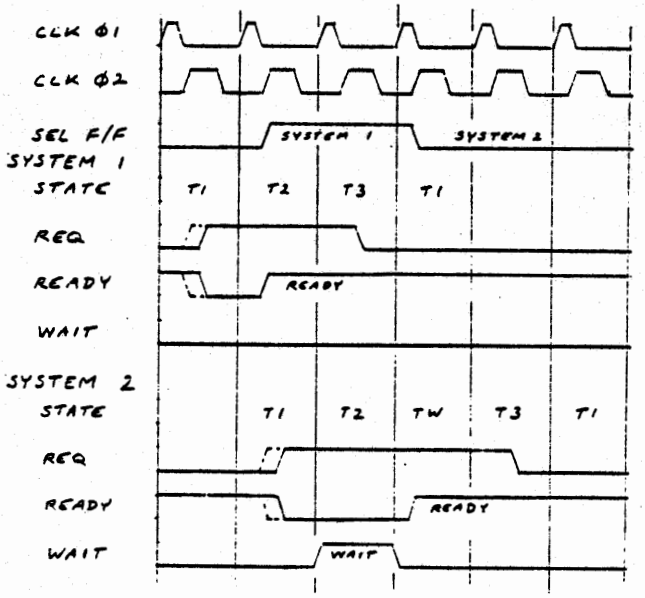
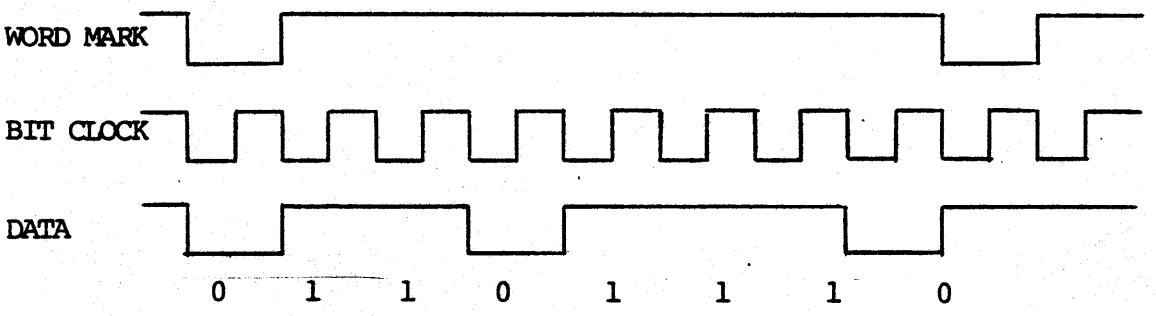

CLK Φ1
CLK Φ2
SEL F/F
SYSTEM 1
SYSTEM 2
SYSTEM 1 STATE
T1
T2
T3
REQ
READY
WAIT
SYSTEM 2 STATE
TW

FIGURE 9-5

9.2 SERIAL INPUT/OUTPUT

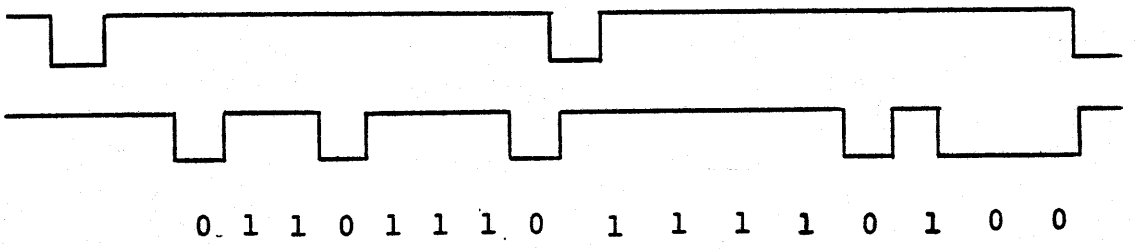
We can attach a meaning to the time of arrival of a data bit, just as we attach a meaning to its position in a binary number. To communicate an eight bit number from one machine to another, the sender outputs a discrete signal on one bit of a data port, thereafter sending successive bits at fixed time intervals. In the early days of computers it was common to send a data signal and two timing signals as discrete outputs.



9.2.1 Signal Coding

These signals are easy to generate and interpret. The sender switches the clock signal at some convenient time interval. Each time it is switched low, a new data bit is sent on a separate line. The receiver observes the clock and reads the data bit when the clock switches high. The first bit of each word is accompanied by a word mark. This delineates characters so that if an occasional bit is lost the entire message will not be garbled.

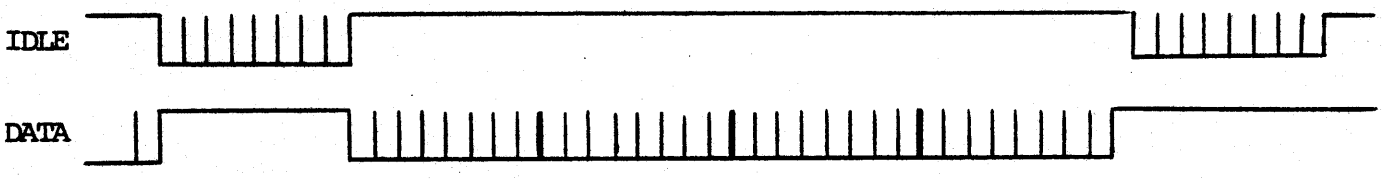
This scheme is simple, but transmitting it over long distances is extravagant as the timing signals carry very little information. If both transmitter and receiver have accurate timing sources, the bit clock is unnecessary. The receiver can recreate it, starting from the edge of the word mark. There are several ways of transmitting the word mark on the same wires with the data, thereby greatly reducing the cost.



We can put time intervals between words on the data line and fit the word marks into the intervals. If they can be distinguished from the data bits (by a narrower or wider pulse, or a different frequency, for instance) they will still serve the same function.

9.2.2 Synchronous Communication

A technique which is in common use is to send word marks only infrequently, maintaining a well synchronized clock over a long message. The word mark is now transmitted not as a single pulse for each word, but as a special, recognizable pattern called an Idle character.



This is merged into the normal data stream as though it were part of the message. It fulfills the role of a word mark in controlling synchronization of the bit clock and in marking the boundary of a character. When the receiver is seeking synchronization, it collects eight bits and compares the pattern with that of the known idle. If the pattern is wrong it discards the oldest bit and shifts in the next. This continues until the idle pattern is recognized, indicating that synchronization has been achieved and communication can begin. It is common in such systems to have at least some degree of reverse communication or feedback from the receiver to the sender, which is used to say 'OK' or 'HELP'. This is called a supervisory channel and is only used to operate the communication system, not to transmit messages.

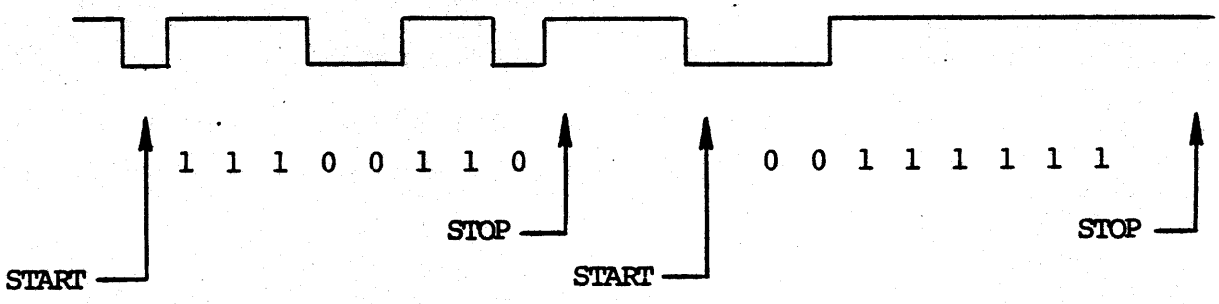
This method is referred to as 'synchronous communication' because of the requirement for continuously synchronized send and receive signals. After the initial period of seeking synchronization, the receiver stays synchronized by observing signal transitions in the data stream. Its crystal clock is able to maintain sync even if long strings of data are all ones or all zeros, of if the signal is temporarily lost. Thus all the signals on the communications line are part of the message being sent. If there is a break in the message, the sender must fill the spaces with idle characters so that the time from the beginning of one word to the beginning of the next is always exactly one word time.

9.2.3 Asynchronous Communication

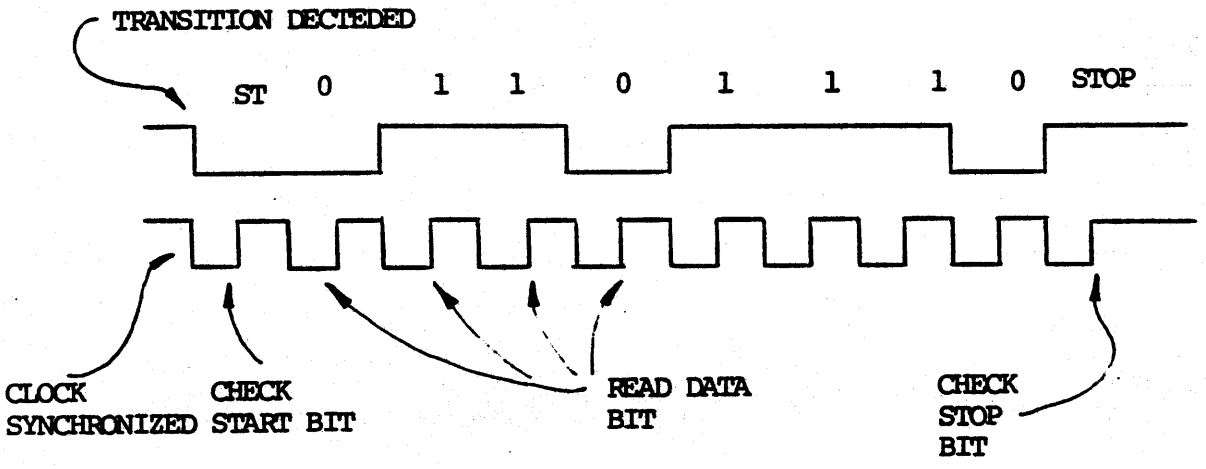
An alternative method is especially suited to devices such as the teletype, whose characters are transmitted and received asynchronously. There may be long pauses between characters, but occasionally one character will quickly follow another.

The transmission rate for a teletype is usually 10 characters per second or approximately 120 words per minute (a very fast typing speed). The same signal format has been adopted for faster electronic communication devices.

In asynchronous communication each character is independent and carries its own word mark. The adopted convention is for each data character to be preceded by a zero, followed by one or more bit-times (intervals) of the 'one' signal.



After some period of time with no data, (i.e. constant 'one' signals) the receiver will see a transition to zero. This signals the start of a character, and the receiver synchronizes its clock.



One half bit-time later the receiver checks the start bit. If it is not zero, an error has been made. Thereafter the receiver accepts eight bits, reading them at one bit-time intervals, then tests the stop bit to

see that it is a 'one'. Now the receiver waits until another transition to zero marks the start of next character.

This data format has been adopted for asynchronous communication by the American National Standards Institute and by CCITT. The data content is also coded in a standardized form. These standards were promulgated by the American Standards Committee on Information Interchange (ASCII).

9.3 TRANSMITTING AND RECEIVING ASCII CHARACTERS

A special purpose communications device, the 8251, is available as a peripheral to the 8080. This is a 'Universal Synchronous - Asynchronous Receiver - Transmitter' (UART). It is a very capable device, and in any busy system its use is well justified. Often, however, the microprocessor has little enough to do that it can readily handle serial communications by 'bit banging' - processing and timing each bit under program control. In this exercise we will program the MTS to send and receive ASCII characters.

The receiving process has been generally described.. We will extend this definition to include 'echoing' and show that the bit banging task is common to both transmitting and receiving.

9.3.1 Echoing

Echoing is a common procedure when a teletype or other keyboard input terminal is used with a computer. The computer receives data from the keyboard and returns the same data to the printer. It appears to the user that he is typing directly to the printer. In fact, the printer mechanism is actuated by the signal returned from the computer, and not by the keyboard. This also provides confirmation of correct receipt of the input by the computer. If it is done over telephone lines it requires 'full duplex' communication - simultaneous transmission in both direction. The computer terminal commonly includes a 'full duplex/half duplex' switch. In full duplex it only prints what it receives from the telephone line, while in half duplex it prints

directly from the keyboard.

Echoing may also be used in communication between computers or between computers and 'intelligent' terminals. It is a simple means of error detection, although extravagant in communication bandwidth.

Figure 9-6 shows two forms of echoing. Bit echoing implies that when each bit is read by the receiver, in the middle of its time period, it is immediately transmitted back to the sender. It is very easy to accomplish with the procedure called bit banging: each time the receiver samples a bit it copies that bit onto its output port. If the transmitter is local, so no significant transmission delay is involved, the transmitter can check the echo just before sending the next bit. The latter technique is of limited value, however, because any telephone connection is likely to introduce intolerable transmission delay.

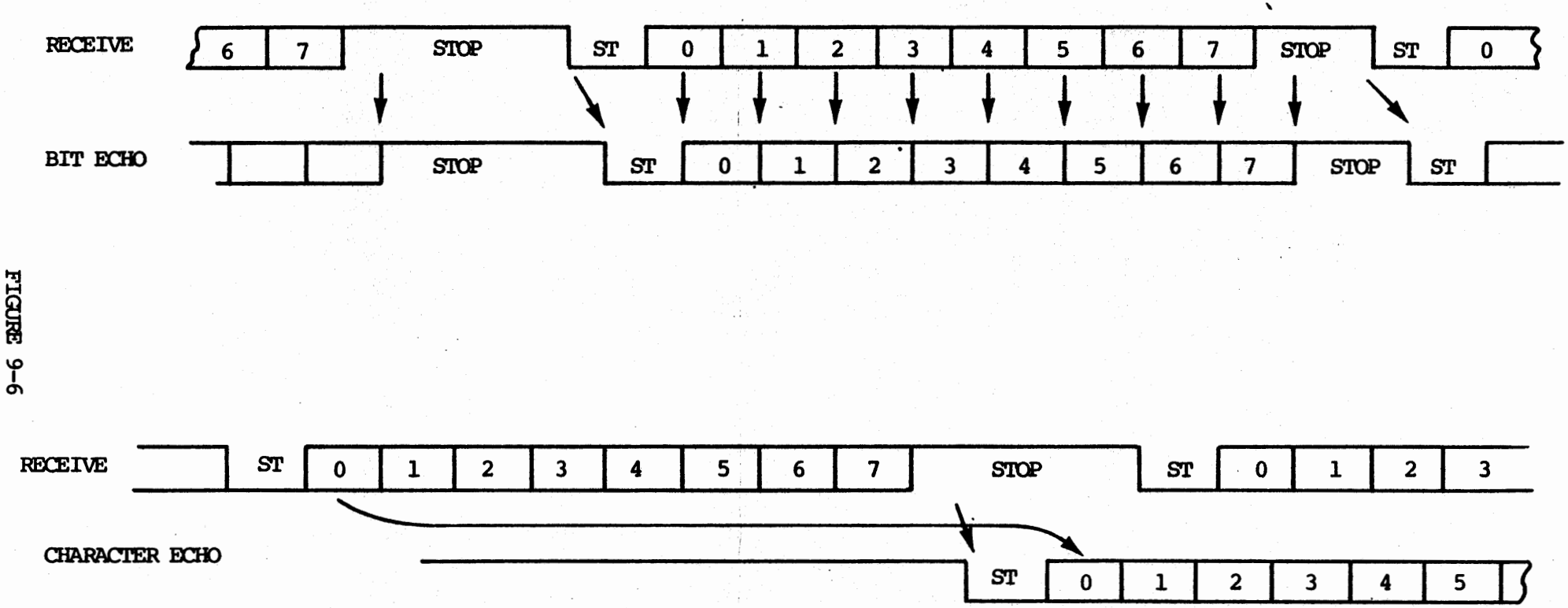


FIGURE 9-6

Figure 8-31

Character echoing is more commonly used over a communication link. The receiver echoes only when it has received the full character. An echo implies not only that the character reached the receiver, but that it entered a program that will process it and has been appropriately stored. Character echoing, moreover, is demanded by some communication facilities such as the current loop, which we will discuss later. Full duplex character echoing demands that the receive and send processes within the terminal be independent of each other. Each must keep track of time independently of the other. This implies all the complexities of full duplex communication, which we shall not treat here. This exercise provides for bit echoing on receive, but only half duplex operation otherwise.

Figure 9-7 shows a bit handler subroutine, used either in transmitting or receiving, with or without bit echoing or bit echo checking. Each time it is called it transmits a bit, waits one bit-time, then receives a bit. Alternate entries allow the calling program to preload a half bit-time to the time counter or to avoid the delay altogether.

Figure 9-8 shows a subroutine for receiving. It repeatedly calls the bit handler for input with no delay until it receives a start bit, then it calls the bit handler for a half bit-time delay and tests to be sure that it still has a start bit. Thereafter it calls the bit handler for full bit-times, each time shifting the received bit into a character and also returning the bit to be echoed.

Figure 9-9 shows a subroutine for sending a character. After sending a half bit-time of stop bit it shifts the character out one bit at a time,

and shifts the received echo bit into the character. It returns with the echo in A, ready to be compared with the original value by the calling program.

In either sending or receiving, the bit handler's timing loop ties up the processor so that no other activities can occur. In some systems this is perfectly acceptable. This program is suitable, for instance, for connecting the MTS to a teletype or other terminal. In many systems, however, sending and receiving and other functions must overlap. Then bit banging can only be done on a timed interrupt basis, and an 8251 or other peripheral becomes much more attractive. You may want to try to develop a program that can send and receive independently at the same time. It is possible at a low data rate.

BIT HANDLER

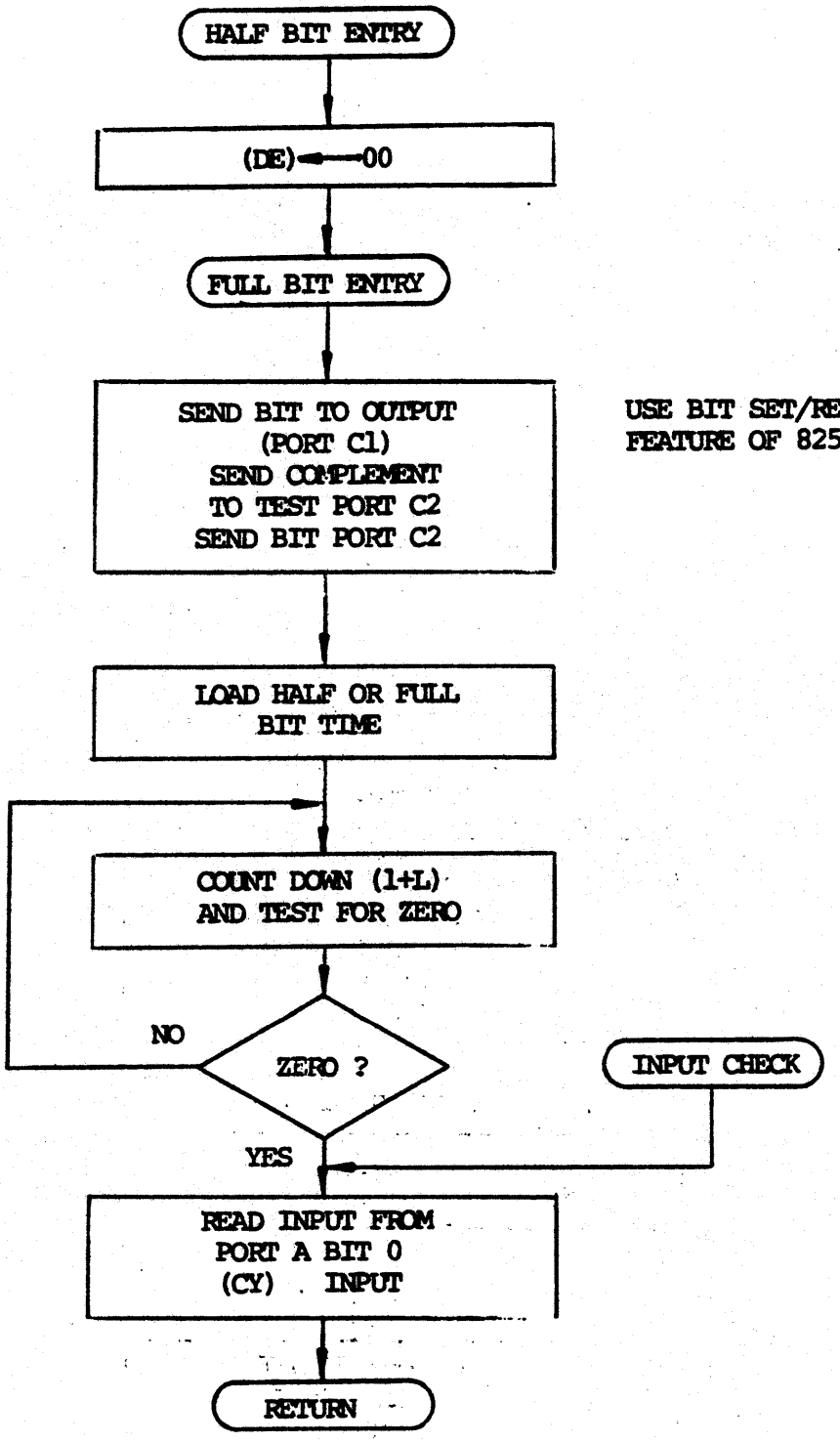


Figure 9-7

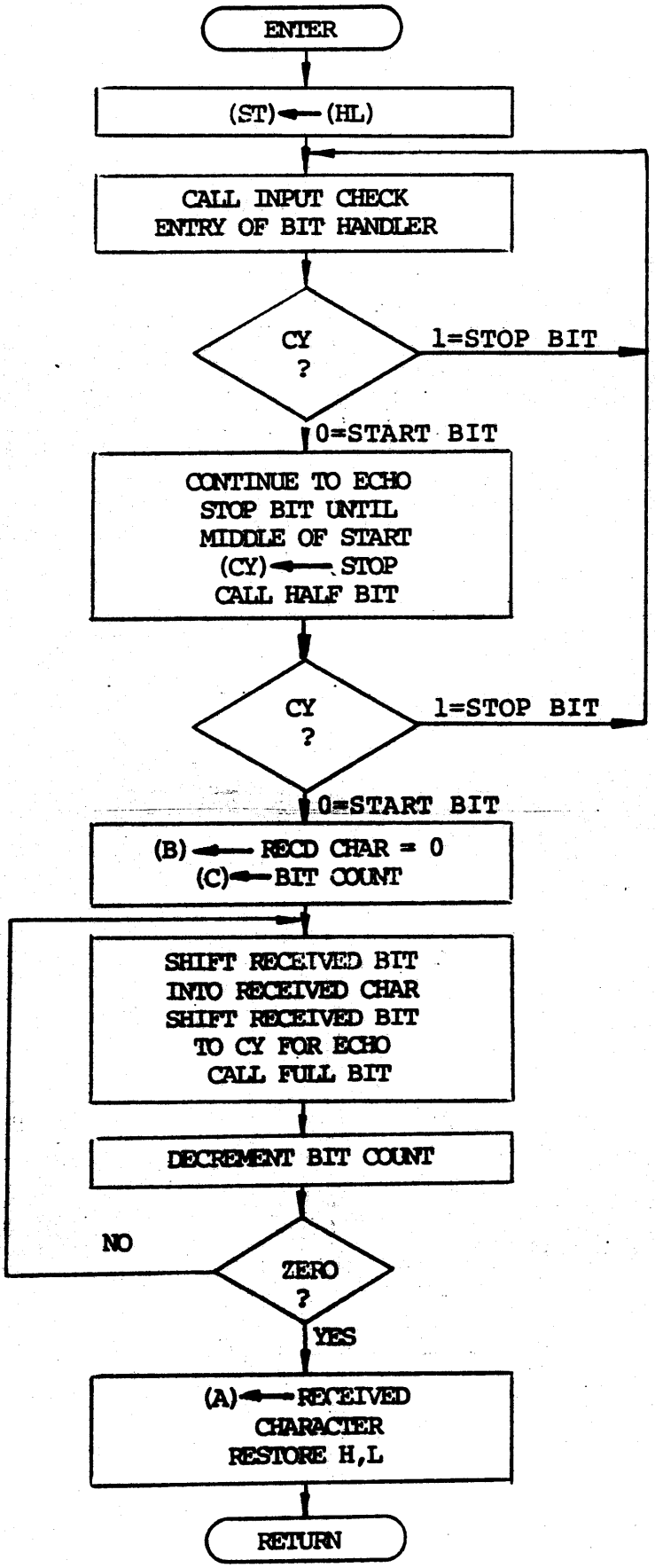

ENTER
(ST)←(HL)
CALL INPUT CHECK
ENTRY OF BIT HANDLER
CY
?
1=STOP BIT
0=START BIT
CONTINUE TO ECHO
STOP BIT UNTIL
MIDDLE OF START
(CY)←STOP
CALL HALF BIT
CY
?
1=STOP BIT
0=START BIT
(B)←RECD CHAR = 0
(C)←BIT COUNT
SHIFT RECEIVED BIT
INTO RECEIVED CHAR
SHIFT RECEIVED BIT
TO CY FOR ECHO
CALL FULL BIT
DECREMENT BIT COUNT
NO
ZERO
?
YES
(A)←RECEIVED
CHARACTER
RESTORE H,L
RETURN

Figure 9-8

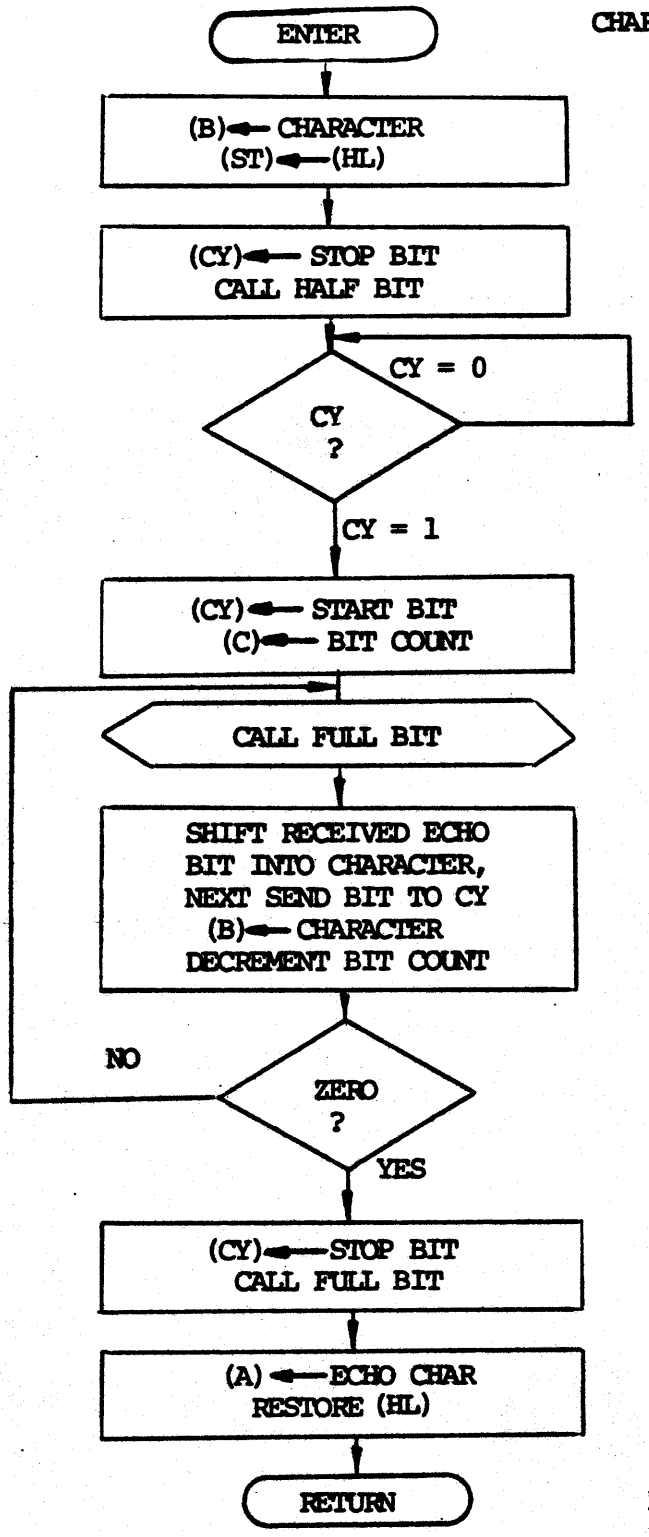

ENTER
CHARACTER IN A
(B)← CHARACTER
(ST)←(HL)
(CY)← STOP BIT
CALL HALF BIT
CY = 0
CY
?
CY = 1
(CY)← START BIT
(C)← BIT COUNT
CALL FULL BIT
SHIFT RECEIVED ECHO
BIT INTO CHARACTER,
NEXT SEND BIT TO CY
(B)← CHARACTER
DECREMENT BIT COUNT
NO
ZERO
?
YES
(CY)←STOP BIT
CALL FULL BIT
(A) ←ECHO CHAR
RESTORE (HL)
RETURN
ECHO IN A
STOP ECHO IN CY

FIGURE 9-9

Figures 9-10 and 9-11 show test programs for the send and receive programs. The sending program transmits a message from memory location 8300 up, until it has sent an FF character. The ASCII code defines this as 'Break', and it is found on all full keyboard terminals. The send test program can operate by itself repeatedly sending data stored in RAM by monitor commands. It provides word marks and a message mark useful for triggering an oscilloscope. If you have a teletype or other terminal available the receive program will take data from the teletype, and when you send Break the send program will transmit back whatever you sent.

TEST PROGRAM FOR SEND

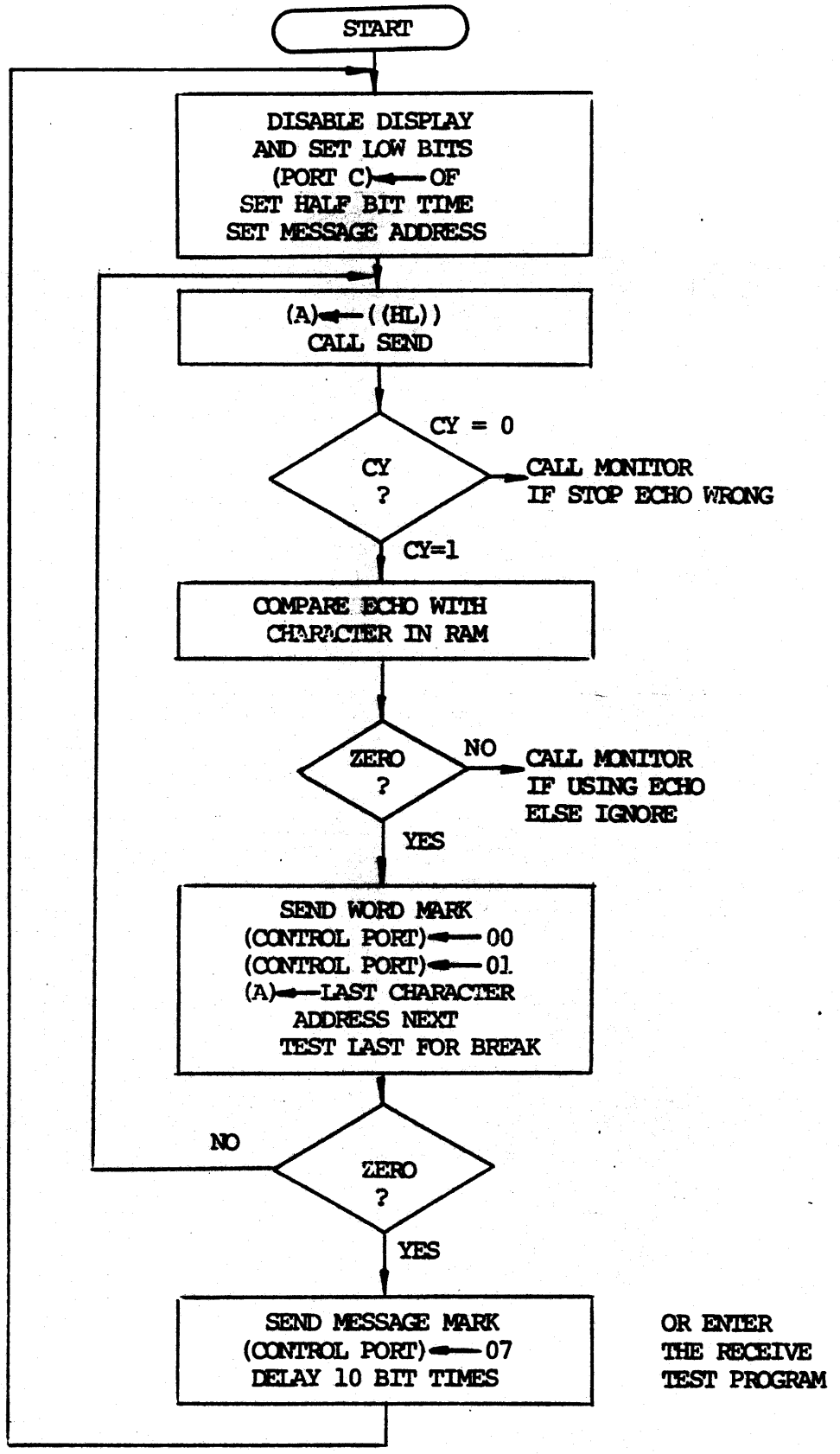


FIGURE 9-10

TEST FOR RECEIVE
(ENTER AFTER BREAK CHARACTER SENT)

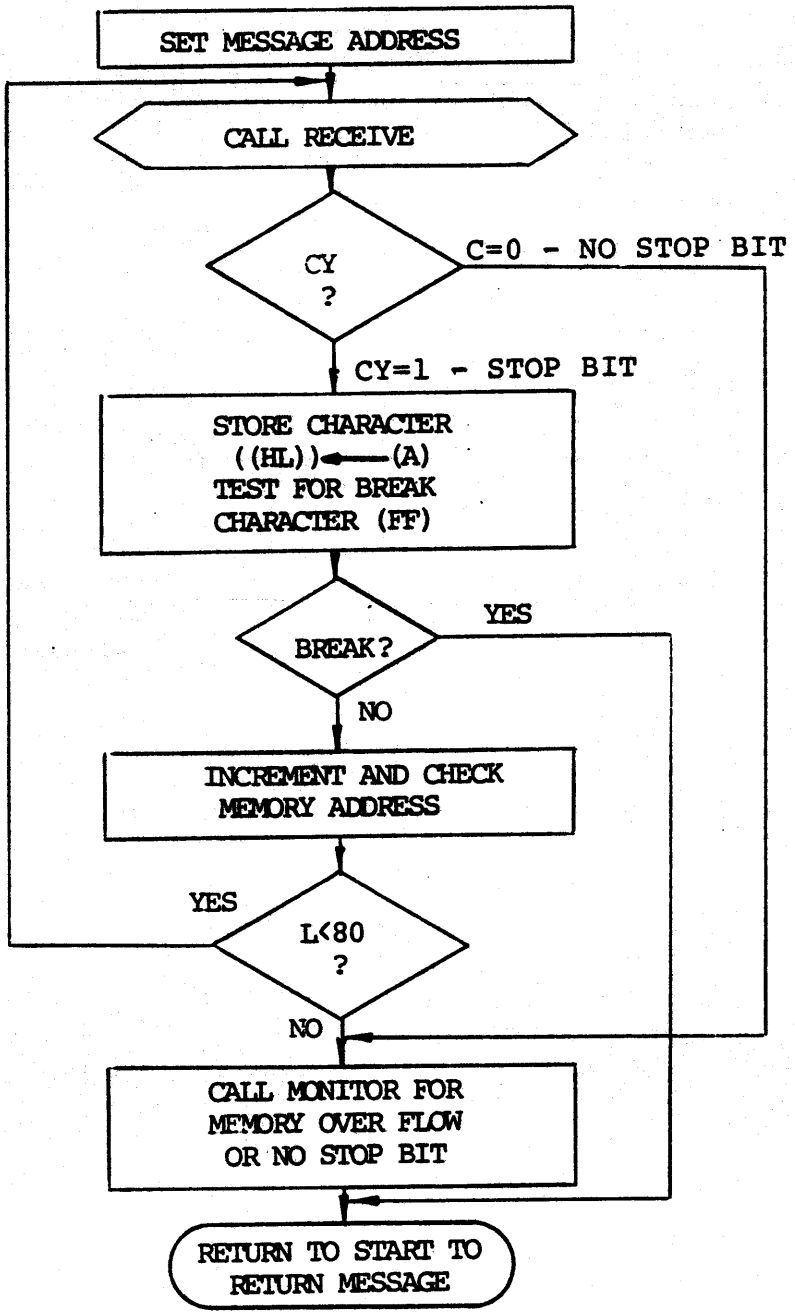


FIGURE 9-11

Calculating the timing loop is rather difficult because of the variable length of the instructions. There are ten different execution times ranging from 2.5 to 11.5 microseconds. Table 9-1 shows the timing for various instructions. Note that the MTS takes one extra clock cycle for each memory access because of the slow CMOS memory; thus the five machine cycles of an XTHL instruction take 18 clock periods with fast memory but 23 in MTS. Table 9-2 shows the timing calculations used to derive the delay times used. The coding solutions for this exercise are presented in Figures 9-12 through 9-17.

INSTRUCTION TIMING	Clock Periods 8080	MTS
MOV r,r	5	6
MOV r,M; MOV M,r	7	9
MVI r	7	9
MVI M	10	13
LXI rp	10	13
LDA; STA	13	17
LDAX; STAX	7	9
LHLD; SHLD	16	21
SPHL; PCHL	5	6
XCHG	4	5
XTHL	18	23
POP	10	13
PUSH	11	14
INR r; DCR r	5	6
INR M; DCR M	10	13
INX rp; DCX rp	5	6
DAD rp	10	11
ADD r; ADC r; SUB r; SBB r ANA r; XRA r; ORA r; CMP r	4	5
ADD M,etc	7	9
ADI etc	7	9
RLC; RRC; RAL; RAR DAA; CMA; STC; CMC	4	5
JMP; JNZ; etc	10	13
CALL	17	22
CNZ etc - executed	17	22
- not executed	11	14
RET	10	13
RNZ etc - executed	11	14
- not executed	5	6
HLT (if interrupted immediately)	7	9
NOP	4	5
IN; OUT	10	13
EI; DI	4	5
RST	11	14

Table 9-1

TIMING
Counting Clock Periods for Program Segments

Bit Handler Timing Loop:

```
82F5 DCX  H       6
  F6 MOV  A,H     6
  F7 ORA  L       5
  F8 JNZ         13
                 30
```

Remainder of Bit handler with full bit-time call:

```
82E3 MVI  A       9
  E5 RAL          5
  E6 OUT         13
  E8 XRI          9
  EA OUT         13
  EC XRI          9
  EE OUT         13
  F0 LHLD        21
  F3 XCHG         5
  F4 DAD         11
     (Timing Loop)
  FB IN          13
  FD RAR          5
  FE RET         13
                139
```

For half bit-time call:

```
82E0 LXI  D      13
                152
```

Send (Bit Loop Only)

```
82AC CALL  22
  AF NOP          5
  B0 MOV  A,B     6
  B1 RAR          5
  B2 MOV  B,A     6
  B3 DCR  C       6
  B4 JNZ         13
                 63
```

Receive (Bit Loop Only)

```
82D1 MOV  A,B     6
  D2 RAR          5
  D3 MOV  B,A     6
  D4 NOP          5
  D5 CALL        22
  D8 DCR  C       6
  D9 JNZ         13
                 ---
                 63
```

Alternate timing loop:

```
     DCR  L       6
     JNZ         13
                 ---
                 19
```

Clocks/Bit = (Send or Receive) + (Bit Handler) + 2N (Timing Loop)

where N is the half bit-time count

= 63 + 139 + 2N(30)
= 202 + 60N

Time/Bit

T = 0.5 (202 + 60N) microseconds
N = (T - 101) / 30

Baud Rate	Time/Bit (microseconds)	N Decimal	N Hex
75	13333.3	441	01B9
110	9090.9	300	012C
150	6666.7	219	00DB
300	3333.3	108	006C
600	1666.7	52	0032
1200	833.3	Not Useable	

N = (T - 101) / 19
(Using shorter timing loop)

300	3333.3	170	AA
600	1666.7	82	52
1200	833.3	38	26

Table 9-2

BIT HANDLER

ADDR	CODE			
82E0	11		LXI D,0000	
1	00			
2	00			
82E3	3E		MVI A,01	Address bit set/reset
4	01			for port C1
5	17		RAL	Bit 0 ← Output Bit
6	D3		OUT CNTPT	Send bit to
7	03			communication port
8	EE		XRI 07	Address port C2
9	07			and invert output
A	D3		OUT CNTPT	Send inverted bit
B	03			to test port C2
C	EE		XRI 01	Set output to
D	01			true value
E	D3		OUT CNTPT	Send true value
F	03			to test port C2
82F0	2A		LHLD 8280	Load half bit time
1	80			
2	83			
3	EB		XCHG	(DE) ← Half bit time
4	19		DAD D	(HL) ← Half or
82F5	2B		DCX H	full bit time
6	7C		MOV A,H	Decrement until
7	B5		ORA L	H and L both 0
8	C2		JNZ 82F5	Loop for time
9	F5			
A	82			
B	DB		IN PORTA	Read received bit
C	00			or echo
D	1F		RAR	(CY) ← Bit
E	C9		RET	
F	00		NOP	
8000				
1			ENTER WITH	BIT TO BE
2			SENT IN CY	
3			RETURN WITH	RECEIVED
4			BIT IN CY	
5			CALL 82E0	FOR HALF BIT
6			CALL 82E3	FOR FULL BIT
7			AFTER PRIOR	CALL
8			TO 82E0	

FIGURE 9-12

ADDR		CODE														
8 2C	0	E	5		P	U	S	H		H						Save HL
	1	C	D		C	A	L	L		8	2	F	B			Call for input test
	2	F	B													
	3	8	2													
	4	D	A		J	C			8	2	C	1				Wait for start bit
	5	C	1													
	6	8	2													
	7	3	7		S	T	C									Echo stop bit for
	8	C	D		C	A	L	L		8	2	E	0			Half bit time
	9	E	0													
	A	8	2													
	B	D	A		J	C			8	2	C	1				If not still start
	C	C	1													bit, go back and
	D	8	2													wait
	E	0	1		L	X	I		B	,	0	0	0	7		(B) = Received char
	F	0	9													(C) = Bit count
8 2D	0	0	0													including stop bit
82D	1	7	8		M	O	V		A	,	B					Shift received
	2	1	F		R	A	R									bit into MSB of
	3	4	7		M	O	V		B	,	A					received character
	4	1	7		R	A	L									STC for no echo
	5	C	D		C	A	L	L		8	2	E	3			Call for full
	6	E	3													bit time
	7	8	2													
	8	0	D		D	C	R		C							Count bits
	9	C	2		J	N	Z		8	2	D	1				Loop until stop
	A	D	1													bit time
	B	8	2													
	C	7	8		M	O	V		A	,	B					
	D	E	1		P	O	P		H							
	E	C	9		R	E	T									
	F	0	0		M	O	P									
8	0															
	1															
	2															
	3															
	4															
	5															
	6															
	7															
	8															

FIGURE 9-13

SEND (CHARACTER IN A)

ADDR		CODE												
8 2A	0	4	7		M	O	V		B	,	A			(B) =
	1	E	5		P	U	S	H						Save H,L for caller
	2	3	7		S	T	C							
	3	C	D		C	A	L	L		8	2	E	0	half bit
	4	E	0											
	5	8	2											
	6	D	2		J	N	C		8	2	A	2		
	7	A	2											
	8	8	2											
	9	0	E		M	V	I		C	,	0	9		
	A	0	9											
	B	3	F		C	M	C							Start bit
	C	C	D		C	A	L	L		8	2	E	3	Send bit for
	D	E	3											full bit
	E	8	2											NOP to make
	F	0	0		N	O	P							SEND = REC in time
8 2B	0	7	8		M	O	V		A	,	B			Shift echo
	1	1	F		R	A	R							
	2	4	7		M	O	V		B	,	A			bit to carry
	3	0	D		D	C	R		C					
	4	C	2		J	N	Z		8	2	A	C		Loop until
	5	A	1											
	6	8	2											
	7	3	7		S	T	C							Send stop bit
	8	C	D		C	A	L	L		8	2	E	3	
	9	E	3											
	A	8	2											
	B	7	8		M	O	V		A	,	B			(A) ← Echo
	C	E	1		P	O	P		H					Restore caller's HL
	D	C	9		R	E	T							Return
	E	0	0		N	O	P							
	F	0	0		N	O	P							
8	0													
	1				E	N	T	E	R		W	I	T	H CHARACTER IN A
	2				R	E	T	U	R	N		W	I	T H ECHO IN A
	3				A	N	D		S	T	U	P		BIT ECHO IN CY
	4													
	5													
	6													
	7													
	8													

FIGURE 9-14

TEST PROGRAM FOR SEND

ADDR	CODE											
8200	00		NOP									
01	00											
02	00											
03	3E		MVI		A	,	0	F				Disable display,
04	0F											enable key input,
05	D3		OUT		P	O	R	T	C			set C0-C3 high
06	02											
07	21		LXI		H	,	0	1	2	C		Set half bit time
08	2C											
09	01											
0A	22		SHLD				8	3	8	0		
0B	80											
0C	83											
0D	21		LXI		H	,	8	3	0	0		Message address
0E	00											
0F	83											
8210	7E		MOV		A	,	M					(A) ← Character
11	CD		CALL				S	E	N	D		
12	A0											
13	82											
14	D4		CNC		0	0	2	0				Call monitor if
15	20											stop echo wrong
16	00											
17	BE		CMP		M							Compare echo
18	C4		CNZ		0	0	2	0				with character
19	20											sent, call monitor
1A	00											if wrong
1B	3E		MVI		A	,	0	0				Set port C0
1C	00											low to generate
1D	D3		OUT		C	N	T	P	T			word mark
1E	03											
1F	3C		INR		A							Set port C0
8220												high to terminate
21												word mark
22												
23												
24												
25												
26												
27												
28												

FIGURE 9-15

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

ADDR	CODE		Instruction	Operand	Comment
8220	D	3	OUT	CNTPT	Terminate word mark
1	0	3			
2	7	E	MOV	A, M	(A) ← Last Char
3	2	3	INX	H	Address next
4	3	C	INR	A	Test last char
5	C	2	JNZ	8210	If not break
6	1	0			go back to send
7	8	2			more
8	3	E	MVI	A, 06	Set port C3 low
9	0	6			to indicate end
A	D	3	OUT	CNTPT	of transmission
B	0	3			
C	C	3	JMP	8240	If a sending
D	4	0			device is available
E	8	2			jump to receive
F	0	E	MVI	C, 0A	Set to wait 10
8230	0	A			bit times
1	C	D	CALL	82F0	Use bit handler
2	F	0			for delay
3	8	2			
4	0	D	DCR	C	Count bits
5	C	2	JNZ	8231	and loop
6	3	1			
7	8	2			
8	C	3	JMP	8200	Go back to start
9	0	0			
A	8	2			
B					
C					
D					
E					
F					
8 0					
1					
2					
3					
4					
5					
6					
7					
8					

FIGURE 9-16

TEST PROGRAM FOR RECEIVE

ADDR		CODE														
8240	0	21		LXI		H,	8300									
	1	00														
	2	83														
	3	CD		CALL		82C0										Receive character
	4	C0														
	5	82														
	6	D2		JNC		8255										Jump if no stop bit
	7	55														
	8	82														
	9	77		MOV		M, A										Store character
	A	3C		INR		A										Test for Break
	B	CA		JZ		8200										At break return
	C	00														to start and
	D	82														return message
	E	23		INX		H										Next address
	F	7D		MOV		A, L										Test for overflow
8250	0	FE		CPI		80										
	1	80														
	2	DA		JC		8243										Continue if L<80
	3	43														
	4	82														
8255	5	E7		RST		4										
	6	C3		JMP		8200										
	7	00														
	8	82														
	9															
	A															
	B															
	C															
	D															
	E															
	F															
8	0															
	1															
	2															
	3															
	4															
	5															
	6															
	7															
	8															

FIGURE 9-17

3.3 Parity and LRC

The most common, and easiest, forms of error detection are 'parity' and 'longitudinal redundancy check', LRC. If a communication character has eight bits available and only seven bits are needed to define all of the characters, as in the ASCII code, the eighth bit can be used for error detection. Count the ones in the character to be sent. If the number is even make the eighth bit zero, but if the number is odd make the eighth bit one. Every character sent will have an even number of ones; this is 'even parity'. The receiver checks to be sure that every character has even parity; any exception is an error.

The 8080 includes an automatic test for parity. The parity flag is set if the result of an arithmetic or logical operation has even parity. Like the zero and carry flags, this can cause a conditional jump, call or return. The instructions are:

```
E2    JPO  Jump if Parity Odd
xx    low address
yy    high address

EA    JPE  Jump if Parity Even
xx    low address
yy    high address

E4    CPO  Call if Parity Odd
xx    low address
yy    high address
```

EC CPE Call if Parity Even

xx low address

yy high address

E0 RPO Return if Parity Odd

E8 RPE Return if Parity Even

To assign even parity to a character you can use a program segment like this:

```
      MOV  A,M      Load the character
      ORA  A        Set/Clear flags
   ┌─ JPE          Jump if parity even
   │  ORI  80      Set parity even
   └─►MOV  M,A     Return character
```

LRC is a similar scheme in which all of the ones in each bit position of many characters are counted and forced to be even (or odd) by adding one extra character.

Bit	7	6	5	4	3	2	1	0
Message	1	1	0	0	1	1	0	0
	0	0	1	1	0	0	1	1
	1	1	1	1	1	1	0	0
	1	0	0	0	0	0	0	1
	0	1	1	0	1	0	0	1
LRC	1	1	1	0	1	0	1	1

The LRC is the exclusive OR of all the message characters. It can be formed, along with parity, by:

```
      MOV  A,M      Load the character
      ORA  A        Set/Clear flags
      JPE           Jump if parity even
      ORI  80       Set parity even
      MOV  M,A      Return character
      XRA  B        Generate LRC
      MOV  B,A      Return LRC
```

The same procedure is used to check the LRC. When it is executed on a message that includes an LRC the final result must be 00 if the message is error-free.

9.4 EQUIPMENT INTERFACING

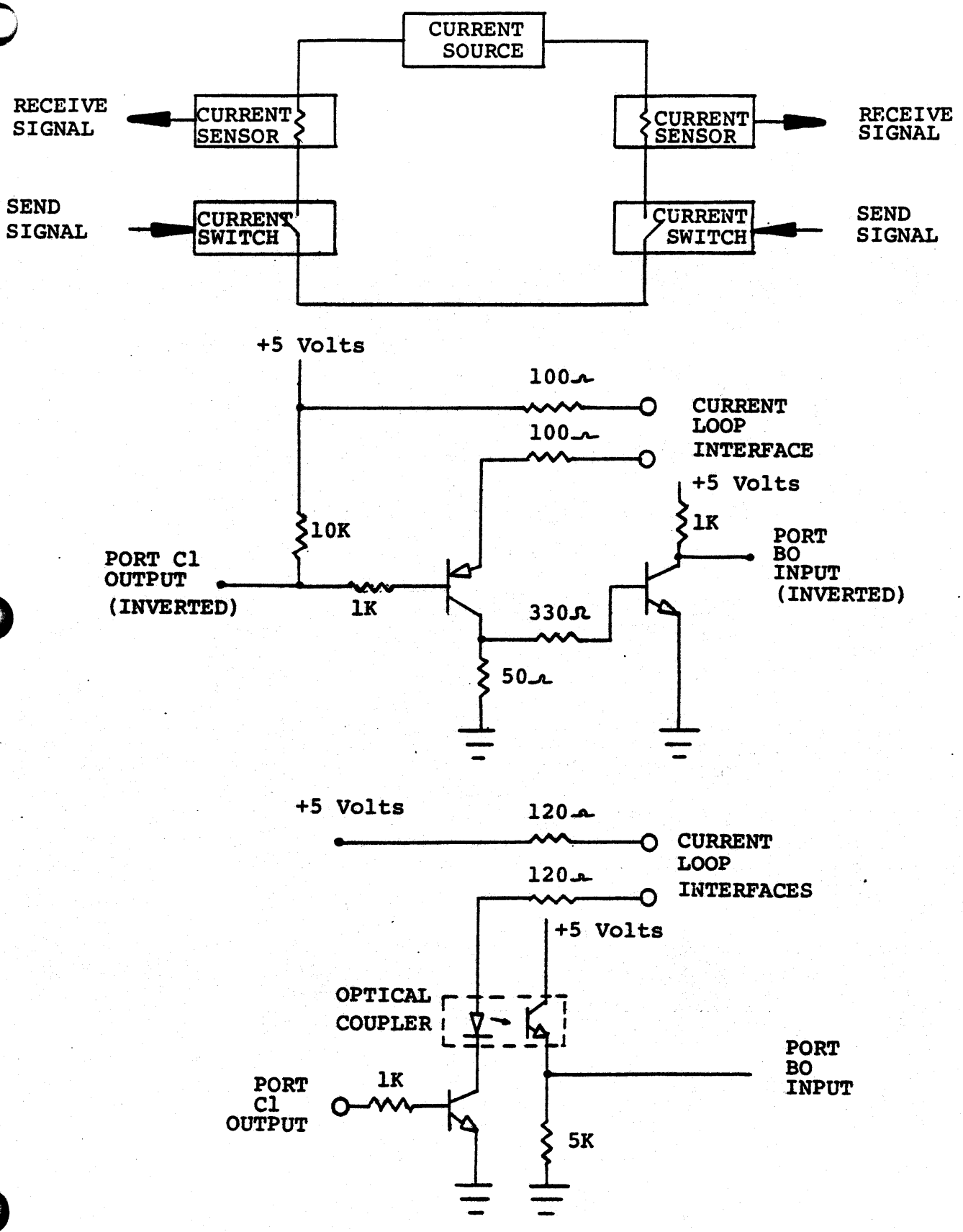
Connecting the MTS to a teletype or other computer terminal generally requires an interface circuit. Many teletypes are set up for a current loop interface, described below. Teletypes and other terminals used with computers or modems typically have RS232C interface that swings positive (+5 to +25 volts) for a one and negative (-5 to -25 volts) for zero. The easiest way to generate an RS232C interface is with a specialized interface circuit such as the Fairchild 9616. A negative voltage is required.

9.4.1 Current Loop Interface

Many teletypes communicate by a 'current loop' interface using a single pair of wires. The loop is powered by a current source that will drive 20 milliamperes (sometimes 62.5 ma). Each device connected to it can sense the presence or absence of current in the loop for receiving, and can open or close the circuit for transmitting. This is a half-duplex system. It can be used for communication in both directions, but not simultaneously; the receiving station must keep the circuit closed. Echoing is not used. Figure 9-18 shows two suitable circuits. Note that the first circuit has inverted input and output, so your program must complement the data.

9.4.2 Magnetic Tape Cassette Modem

Figure 9-19 shows a digital modem (modulator-demodulator) suitable for recording and reading data with a consumer cassette recorder. In conjunction with monitor programs SEROT and SERIN, it will copy your programs onto cassettes and reload them. This circuit is included on the hardware interface circuit board supplied with the Integrated Computer Systems Course 536. Refer to appendix A, page A-3 for instructions on the use of SEROT and SERIN.



CURRENT LOOP INTERFACES

FIGURE 9-18

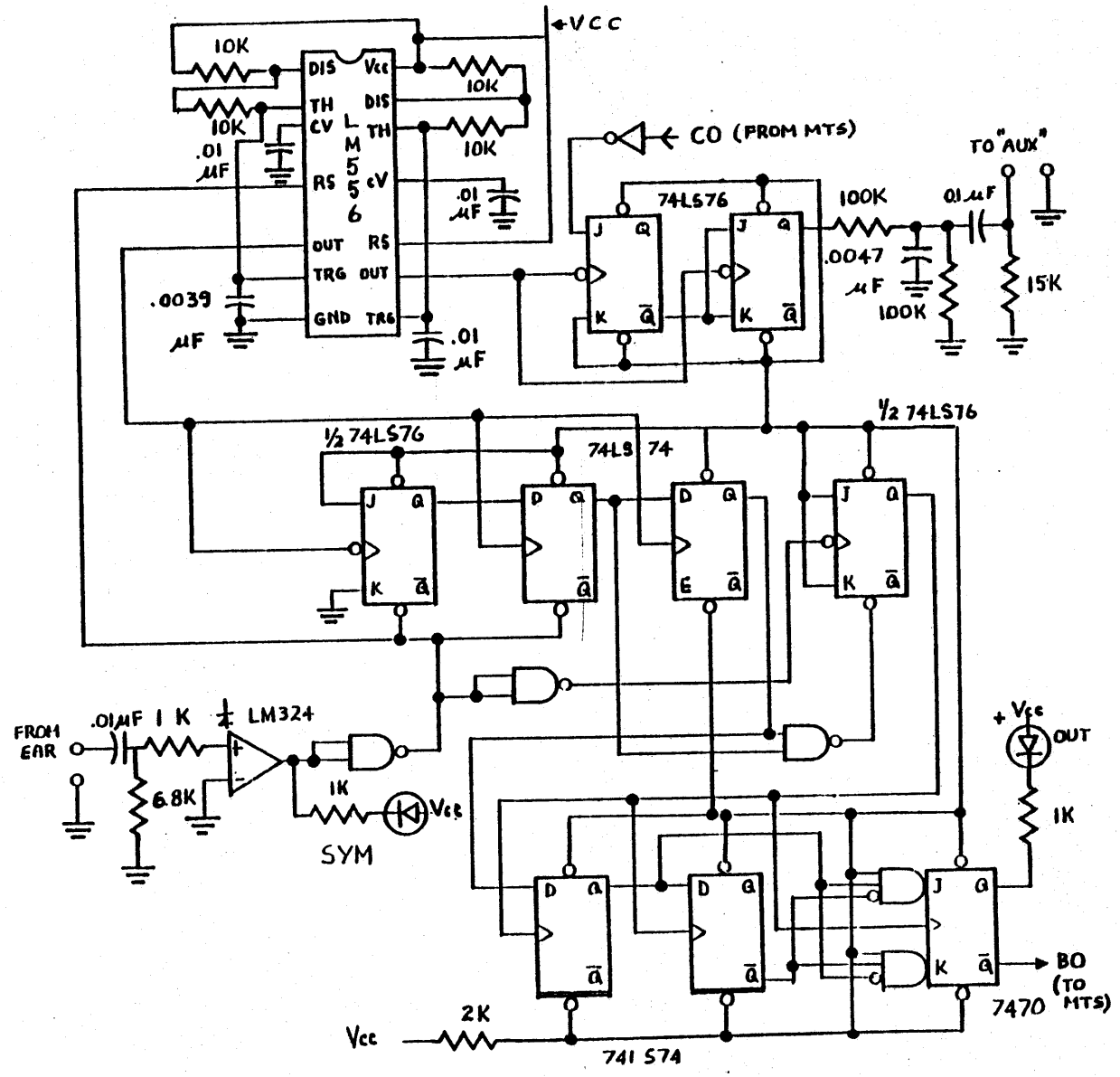


FIGURE 9-19 TAPE CASSETTE MODEM

MICROCOMPUTER TRAINING WORKBOOK

CHAPTER 10

BINARY AND DECIMAL ARITHMETIC

NOTE:

Early versions of the Microcomputer Training System included the NEC 8080A as the central processing unit. This is different from the Intel 8080A in its handling of decimal subtraction; these differences are described in the text of this chapter. More recent Microcomputer Training System's utilize the NEC 8080AF; this processor is logically identical to the Intel device. If the unit delivered includes the NEC 8080AF, the decimal subtraction operations in Section 10.4 will yield erroneous results; and in Section 10.6.2 you must use the subroutine of Figure 10-33 rather than 10-32.

10. BINARY AND DECIMAL ARITHMETIC

A number of the exercises presented in earlier chapters have included some arithmetic functions, including (in Chapter 4) addition, subtraction and multiplication. In this chapter we introduce decimal arithmetic, the subtract instructions, multiple precision addition and multiplication, negative numbers, fractions, and division, and review the basic concepts of binary arithmetic.

10.1 BINARY ADDITION

The rules for binary addition were presented in Chapter 1, section 1.2.4, and a quick review of that material is suggested. The complete addition table for binary arithmetic is:

```
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 10
```

Addition of two bit numbers produces carries into the third position. This extends to full eight bit addition:

```
   1111 1111
+  1111 1111
-------------
= 11111 1110
```

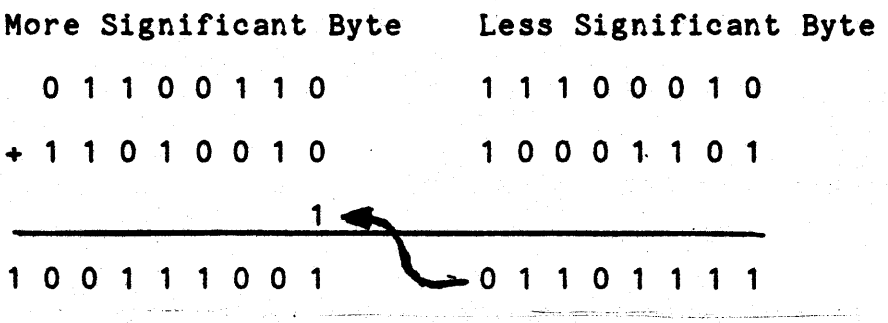
Eight bit addition can generate a carry into the ninth position. The addition of two numbers of any size may produce a carry into the next bit position. When a carry is generated, however, the sum never has ones in all positions. The example above shows the addition of the two largest possible eight bit numbers. A carry is generated but the least significant bit is zero. This is of fundamental importance for multiple precision addition.

10.1.1 Multiple Precision:

The use of more than one word to represent a number is termed multiple precision. If the number is an integer, this permits a greater value

than can be represented in a single word. If the number is a fraction it permits greater precision than can be represented in a single word. The number of words used is often used to describe the operation. Thus double precision refers to arithmetic operations using two words, triple precision to three words, etc.

Consider a double precision addition in which each number is represented by two memory words (or bytes in an eight bit machine):



We add the two less significant bytes, and if a carry is generated, as above, it must be added in with the more significant bytes. Even if every bit in all four bytes was one, only a single carry bit is generated from the complete addition. This permits a multiple precision addition to proceed as follows:

a) Add the two less significant bytes.
b) Add the next two bytes, and if a carry resulted from the preceding addition add it into the sum.
c) Repeat (b) for as many bytes as are required.

The ADC instruction was introduced in Chapter 7 as a means of shifting. Now it appears as an arithmetic instruction to be used for multiple precision arithmetic. As with the other arithmetic and logical

instructions there is a version of ADC using each of the registers as a source:

```
8F   ADC  A        Add the content of the
88   ADC  B        named register and the
89   ADC  C        carry flag to the content
8A   ADC  D        of register A, and place
8B   ADC  E        the result in register A.
8C   ADC  H
8D   ADC  L        All flags are set or reset
8E   ADC  M        according to the result.
```

A double precision add of the content of register pairs B,C and D,E could be done by:

```
MOV   A,C      (A) <- Less significant byte
ADD   E        Ignore previous carry on first addition
MOV   E,A      Store less significant byte
MOV   A,B      (A) <- More significant byte
ADC   D        Add with carry
MOV   D,A      Store more significant byte
```

The 8080 includes a separate double precision add function, however, allowing two register pairs to be added directly. The above could have been performed by:

```
XCHG           Move (D,E) into (H,L)
DAD   B        Add (B,C) to (H,L)
XCHG           Put the result in (D,E)
```

Of course if one addend had been in HL originally and we wanted the result in HL, a single DAD instruction would do the job. Therefore double precision is usually done with DAD rather than ADC.

For convenience in discussing these functions we will refer to the augend (a number to which another will be added to generate a sum) and the addend (a number to be added to an augend to generate a sum).

10.2 FOUR BYTE ADDITION

We will use the following specification for this exercise:

a) To a four byte number in memory locations 8380 - 8383 add the four byte number in 8390 - 8393.

b) Place the result in 8380 - 8383 and clear 8390 - 8393.

c) Display the result.

Write a subroutine for the addition, to be called with addresses and byte count already loaded. Note that you can modify addresses and count bytes without affecting the carry flag, because INR and DCR affect all flags except carry; INX and DCX affect no flags at all.

Figures 10-1 through 10-4 present flow charts and coding sheets for this exercise.

MAIN PROGRAM FOR 4 BYTE ADD AND DISPLAY

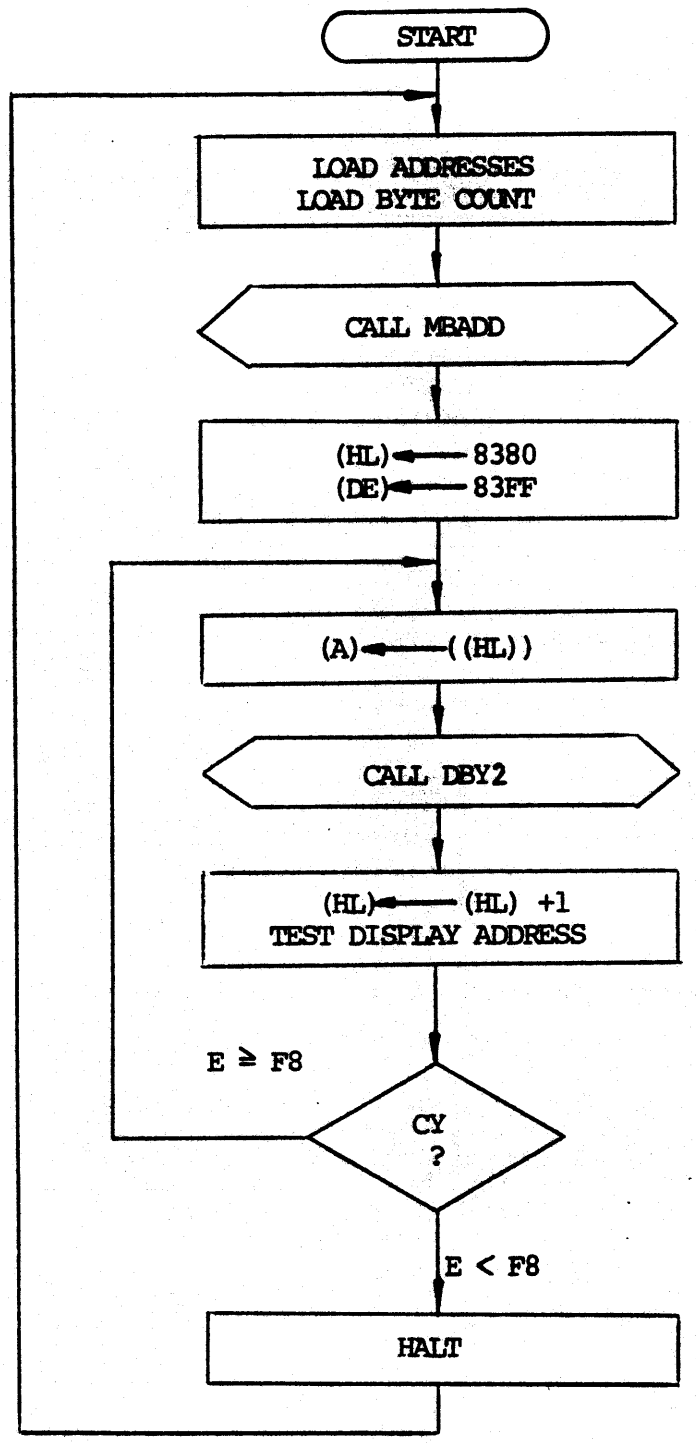
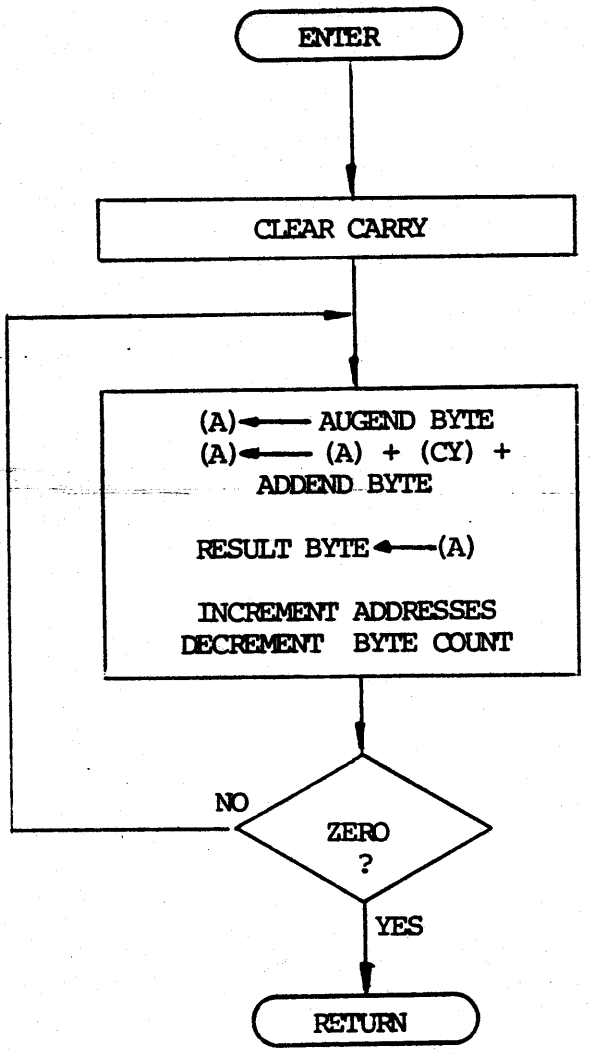


FIGURE 10-1

MULTI BYTE ADD SUBROUTINE



(DE) = AUGEND ADDRESS
(HL) = ADDEND ADDRESS
(C) = BYTE COUNT

FIGURE 10-2

MAIN PROGRAM FOR 4 BYTE ADD AND DISPLAY

ADDR	CODE			
8200	00	NOP		
01	00	NOP		
02	00	NOP		
03	11	LXI	D, 8380	Address for
04	80			Augend and result
05	83			
06	D5	PUSH	D	Save for display
07	21	LXI	H, 8390	Address for
08	90			Addend, to be
09	83			cleared
0A	0E	MVI	C, 04	Byte count
0B	04			for addition
0C	CD	CALL	MBADD	Multi Byte Add
0D	F0			subroutine
0E	82			
0F	E1	POP	H	(HL) ← Data address
8210	11	LXI	D, 83FF	(DE) ← Display address
11	FF			
12	83			
13	7E	MOV	A, M	(A) ← Byte
14	CD	CALL	DBY2	Display byte
15	98			subroutine
16	02			
17	23	INX	H	Address next byte
18	7B	MOV	A, E	Test display address
19	FE	CPI	F8	to see if all have
1A	F8			been displayed
1B	D2	JNC	8213	Continue until
1C	13			left digit displayed
1D	82			
1E	00	NOP		
1F	00	NOP		
8220	76	HLT		
21	C3	JMP	8200	
22	00			
23	82			
24				
25				
26				
27				
28				

FIGURE 10-3

MULTI-BYTE ADDITION SUBROUTINE

CODING SHEET — MICROCOMPUTER TRAINING SYSTEM — INTEGRATED COMPUTER SYSTEMS

ADDR	CODE	INSTRUCTION	OPERAND	COMMENT
8 2F 0	B7	ORA	A	Clear Carry
1	1A	LDAX	D	(A) ← Augend
2	8E	ADC	M	Add Addend
3	12	STAX	D	((DE)) ← Sum
4	36	MVI	M, 00	Clear Addend
5	00			
6	13	INX	D	Next augend address
7	23	INX	H	Next addend address
8	0D	DCR	C	Count bytes
9	C2	JNZ	82F1	Loop until finished
A	F1			
B	82			
C	FB	EI		Enable Interrupt
D	C9	RET		if used with
E				interrupt exercise
F				
8 0		NOTE:	ENTER WITH	
1		(DE) =	ADDRESS FOR AUGEND	
2			AND RESULT	
3		(HL) =	ADDRESS FOR ADDEND	
4			TO BE CLEARED	
5		(C) =	NUMBER OF BYTES	
6				
7				
8				
9				
A				
B				
C				
D				
E				
F				
8 0				
1				
2				
3				
4				
5				
6				
7				
8				

FIGURE 10-4

The calling program uses a feature that is seldom convenient with the monitor - the HLT instruction. After displaying the result your task is finished until you load new data, so it is reasonable to HLT until an interrupt occurs. As long as the STEP/AUTO toggle switch is in the STEP position, however, the monitor interrupts at every instruction, so you cannot really halt. You will be interrupted, go back to the start and do the addition and display again. Since the augend now contains the result and the addend is cleared, the result will be the same and The display will be fixed, as though the halt had been effective. Now if you turn the switch to AUTO, the processor will indeed halt until you press RST or introduce an interrupt some other way. The difference is not visible unless you watch with an oscilloscope. The modification shown in Figure 10-5 uses a trick to make it visible. We turn on the decimal point at the right hand digit just before the halt, and turn it off immediately afterward, so it is only illuminated during the halt. Try it in both STEP and AUTO modes.

MODIFY MAIN TO DISPLAY HALT

CODING SHEET — MICROCOMPUTER TRAINING SYSTEM — INTEGRATED COMPUTER SYSTEMS

ADDR	CODE		INSTRUCTION	OPERAND	COMMENT
8 2 0	21		LXI	H, 83FF	Address right
1	FF				hand display digit
2	83				
3	7E		MOV	A, M	Fetch display
4	EE		XRI	80	Turn decmal
5	80				point on
6	77		MOV	M, A	Display
7	76		HLT		Halt
8	7E		MOV	A, M	Fetch display
9	EE		XRI	80	Turn decimal
A	80				point off
B	77		MOV	M, A	Display
C	C3		JMP	8200	Jump back
D	00				
E	82				
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 10-5

10.3 BINARY SUBTRACTION

The process of subtraction is defined by these equations:

```
If      A = B + C
then    A - B = C
and     A - C = B
```

This can be expressed in terms of 8080 instructions:

```
MOV     A,B
ADD     C           (A) <- (B) + (C)
SUB     B           (A) <- (A) - (B)    result is equal to C
```

Successive ADD and SUB of the same values cancel each other, except that flags may be affected. The subtract instruction is again one of a set which includes one for each register:

```
97   SUB  A        Subtract the content of the named
90   SUB  B        register from the content of the
91   SUB  C        A register. If the content of the
92   SUB  D        named register was less than the
93   SUB  E        A register set the carry flag. Set
94   SUB  H        or clear the other flags according
95   SUB  L        to the results of the subtraction.
96   SUB  M
```

Like ADD, SUB ignores and destroys the previous content of the carry flag. Another set of instructions SBB r, includes the carry flag:

SBB (A)←(A) - (r) - (CY)

(A)←(A) - (B) - (CY)

The result of SUB or SBB sets or clears the carry flag, which is meant to be passed to the next more significant byte. In subtraction, it becomes a borrow flag. It is set if the subtrahend (B, in the example) is greater than the minuend (A), and in multi-byte subtraction the borrow is subtracted from A when the next byte is processed. This is done by the subtract with borrow instruction:

```
9F   SBB  A     Subtract from the content
98   SBB  B     of the A register the
99   SBB  C     content of the CARRY
9A   SBB  D     flag and the content
9B   SBB  E     of the named register.
9C   SBB  H     Place the result in
9D   SBB  L     register A.  Set or clear
9E   SBB  M     all flags according to the result.
```

A double precision subtraction can be done by:

```
MOV  A,C
SUB  L
MOV  E,A        (E) <- (C) - (L)
MOV  A,B
SBB  H
MOV  D,A        (D) <- (B) - (H) - (CY)
```

The result in (DE) is (BC) - (HL). Multiple precision subtraction would use the SBB M instruction:

```
LDAX B
SBB  M
STAX D          ((DE)) <- ((BC)) - ((HL)) - (CY)
INX  B
INX  D          next addresses
INX  H
```

Note that we have used three register pairs for addresses, and register A for the subtraction, leaving no register available to count bytes. We can keep a byte counter in a fixed memory location and use LDA, DCR A, STA to count, or we can use the stack. But be careful: POP PSW to bring a counter into register A will destroy the carry flag, which is needed. This is a place where the XTHL instruction is very useful. Write a subroutine for a general purpose multi-byte subtraction, entering with:

(A) = number of bytes
(B,C) = address for minuend
(D,E) = address for difference
(H,L) = address for subtrahend

We can use the same calling program as for the addition, except that we must load an address to (B,C) and initialize a byte counter in A, and the call will be to the subtract subroutine at 82D0. Place the minuend (from which the subtrahend will be subtracted) at 8370 - 73; the difference at 8380 - 83, and the subtrahend at 8390 - 93. Since they are to be kept separate, do not clear any of these areas during the operation. For convenience in an exercise of the following section, leave a NOP immediately after the SBB M instruction.

MULTI-BYTE SUBTRACT

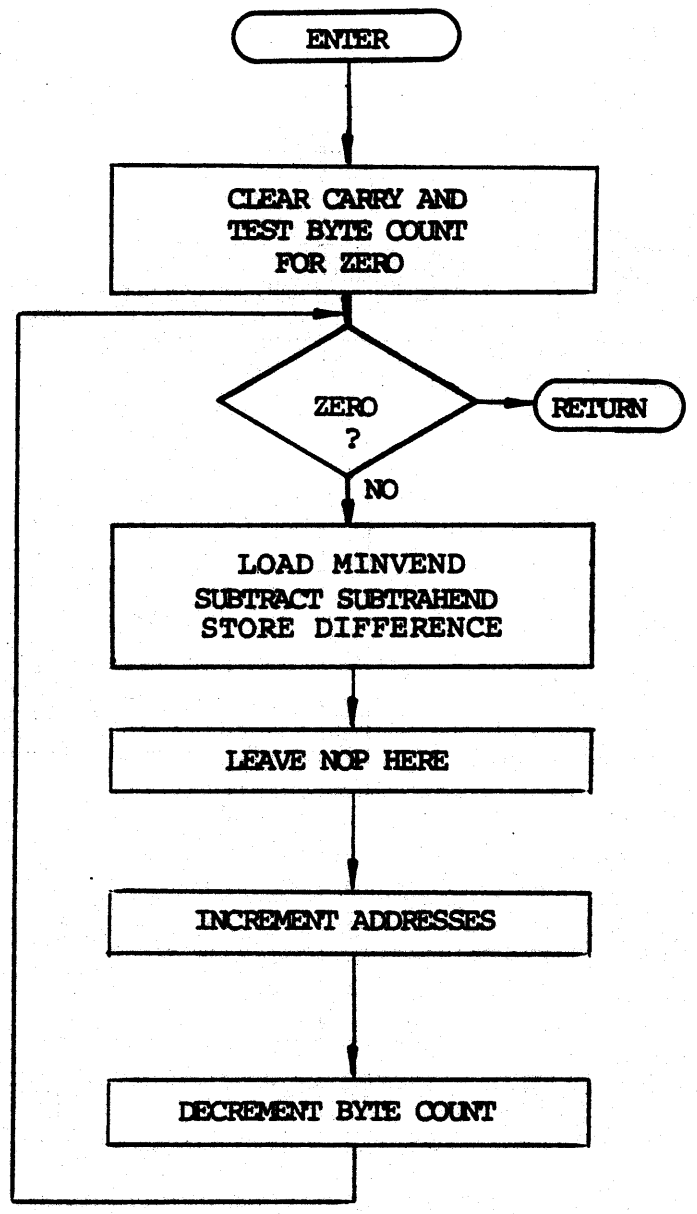


FIGURE 10-6

MAIN PROGRAM FOR 4 BYTE SUBTRACT

ADDR	CODE		COMMENT
8200	01	LXI B, 8370	Address for
01	70		minnend
02	83		
03	11	LXI D, 8380	Address for
04	80		difference
05	83		
06	D5	PUSH D	Save for display
07	21	LXI H, 8390	Address for
08	90		subtrahend
09	83		
0A	3E	MVI A, 04	Byte count in A
0B	04		
0C	CD	CALL MBSUB	
0D	D0		
0E	82		
0F	E1	POP H	(HL) ← Result Addr
8210	11	LXI D, 83FF	(DE) ← Displ Addr
11	FF		
12	83		
13	7E	MOV A, M	(A) ← Byte
14	CD	CALL DBY2	Display Byte
15	78		Subroutine
16	02		
17	23	INX H	Address next byte
18	7B	MOV A, E	Test display address
19	FE	CPI F8	
1A	F8		
1B	D2	JNC 8213	
1C	13		
1D	82		
1E	00	NOP	
1F	00	NOP	
8220			
21			
22			
23			
24			
25			
26			
27			
28			FIGURE 10-7

DISPLAY HALT

ADDR	CODE											
8220	21		LXI		H,	83FF						This segment is not needed
1	FF											
2	83											
3	7E		MOV		A,	M						
4	EE		XRI		80							
5	80											
6	77		MOV		M,	A						
7	76		HLT									
8	7E		MOV		A,	M						
9	EE		XRI		80							
A	80											
B	77		MOV		M,	A						↓
C	C3		JMP		8200							Go back to start
D	00											
E	82											
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 10-8

MULTI-BYTE SUBTRACTION SUBROUTINE

ADDR		CODE															
8 2D	0	B	7		O	R	A		A								Clear CY and test
	1	C	8		R	Z											for zero byte count
	2	F	5		P	U	S	H		P	S	W					Save byte count
	3	0	A		L	D	A	X		B							Load Minuend
	4	9	E		S	B	B		M								Subtract Subtrahend
	5	0	0		N	O	P										To be used later
	6	1	2		S	T	A	X		D							Store difference
	7	0	3		I	N	X		B								Next addresses
	8	1	3		I	N	X		D								
	9	2	3		I	N	X		H								
	A	E	3		X	T	H	L									(ST) ← (HL);
	B	7	C		M	O	V		A	,	H						(A) ← Byte Count
	C	E	1		P	O	P		H								Restore HL and stack
	D	3	D		D	C	R		A								Decrement count
	E	C	3		J	M	P		8	2	D	1					Go to test for
	F	D	1														zero and return
8	0	8	2														when finished
	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 10-9

The subroutine can be changed from subtraction to addition by altering one instruction (at 82D4):

```
9E   SBB  M      to subtract
8E   ADC  M      to add
```

We now introduce a scheme that is not available to programs stored in ROM but can be very convenient for programs in RAM. The program can modify itself by altering the instruction in response to an input. After the display, and before jumping back to the start, take a key input for a command to add or subtract. Use NEXT (=15) for add; STEP (=13) for subtract. For any undefined key enter NOP instead of either ADC or SBB. Use the monitor subroutine GETKY, which waits for a key to be entered. Figures 10-10 through 10-12 show a coding example.

PROGRAM MODIFY MODULE

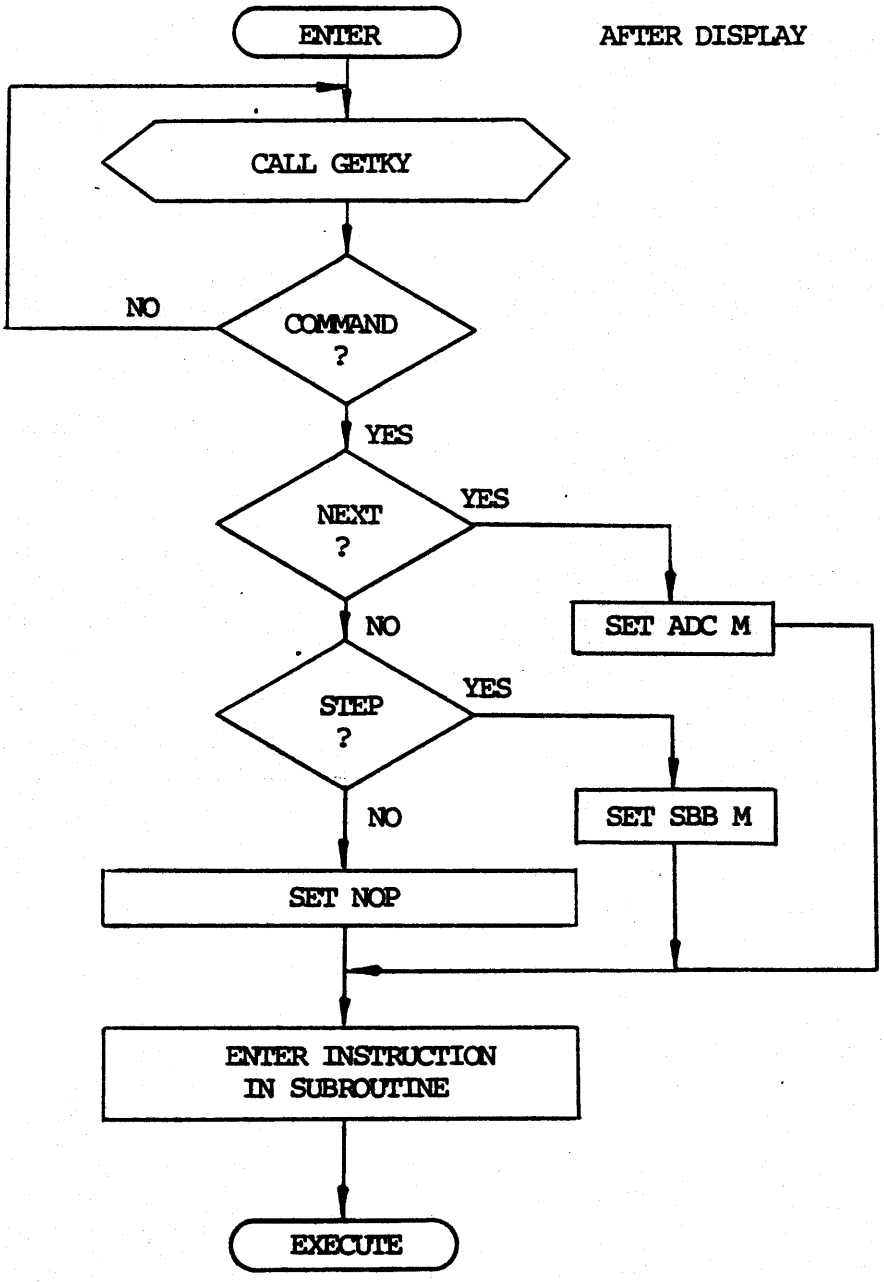


FIGURE 10-10

MICROCOMPUTER TRAINING SYSTEM CODING SHEET

MODIFY SUBR BY KEY INPUT

ADDR	CODE		INSTRUCTION	COMMENT
8 2 2 0	C D		CALL GETKY	Accept Key
1	3 D			
2	0 2			
3	D A		JC 8220	Reject hex keys
4	2 0			
5	8 2			
6	2 1		LXI H, 82D4	Address of add
7	D 4			or subtract
8	8 2			instruction
9	0 6		MVI B, 8E	Enter ADCM
A	8 E			
B	F E		CPI 15	Is key NEXT ?
C	1 5			
D	C A		JZ 823A	If so, go store
E	3 A			ADC M in
F	8 2			subroutine
8 2 3 0	0 6		MVI B, 9E	Enter SBB M
1	9 E			
2	F E		CPI 13	Is key STEP ?
3	1 3			
4	C A		JZ 823A	If so, go store
5	3 A			SBB M in
6	8 2			subroutine
7	0 6		MVI B, 00	Enter NOP
8	0 0			for any other key
9	0 0		NOP	
A	7 0		MOV M, B	Enter instruction
B	C 3		JMP 8200	in subroutine
C	0 0			and execute
D	8 2			
E				
F				
8 0				
1				
2				
3				
4				
5				
6				
7				

INTEGRATED COMPUTER SYSTEMS

FIGURE 10-11

MULTI-BYTE ADD / SUBTRACT SUBROUTINE

ADDR		CODE															
8 2D	0	B	7		O	R	A		A								Clear CY and test
	1	C	8		R	Z											for zero byte count
	2	F	5		P	U	S	H		P	S	W					Save byte count
	3	0	A		L	D	A	X		B							Load augend/minuend
	4	0	0		N	O	P										To be replaced
	5	0	0		N	O	P										
	6	1	2		S	T	A	X		D							Store sum/difference
	7	0	3		I	N	X		B								Next addresses
	8	1	3		I	N	X		D								
	9	2	3		I	N	X		H								
	A	E	3		X	T	H	L									(ST) ← (HL);
	B	7	C		M	O	V		A	,	H						(A) ← Byte Count
	C	E	1		P	O	P		H								Restore HL and stack
	D	3	D		D	C	R		A								Decrement count
	E	C	3		J	M	P		8	2	D	1					Go to test for
	F	D	1														zero and return
8	0	8	2														when finished
	1																
	2																
	3																
	4																
	5																
	6																
	7																
	8																
	9																
	A																
	B																
	C																
	D																
	E																
	F																
8	0																
	1																
	2																
	3																
	4																
	5																
	6																
	7																FIGURE 10-12
	8																

10.4 DECIMAL ADDITION AND SUBTRACTION

Often the microprocessor will have a human interface for its arithmetic results, and decimal input and output will be required. The 8080 provides an instruction to convert a binary result to a decimal result:

27 DAA Decimal Adjust Accumulator

This tests the result of an arithmetic instruction and corrects the content of the accumulator to create a 'packed decimal' result, in the form of two decimal digits. Before exploring the operation in detail we will insert the instruction into the subroutine of the previous exercise. To compare results of decimal versus binary arithmetic, we will provide for inserting or deleting this instruction under keyboard control as we did the ADC and SBB instructions. Use the key RUN to invoke binary and ADDR to invoke decimal results, and interpret them as you did NEXT or STEP . Insert NOP after ADC or SBB for binary, DAA for decimal. As before, any undefined key should place a NOP in place of the ADC or SBB.

If the numbers used generate no carries, the binary and decimal results are alike. Try putting 33 33 33 33 at 8370 - 73 for the augend or minuend and 22 22 22 22 at 8390-93 for the addend or subtrahend. Then addition will produce 55 55 55 55; subtraction, 11 11 11 11. Try your program with those numbers to make sure it works. Coding examples are shown in Figures 10-13 and 10-14.

CODING SHEET — MICROCOMPUTER TRAINING SYSTEM — INTEGRATED COMPUTER SYSTEMS

ADDR	CODE		LABEL	OPCODE	OPERAND	COMMENTS
8220	C	D		CALL	GETKY	Accept key
1	3	D				
2	0	2				
3	D	A		JC	8220	Reject hex keys
4	2	0				
5	8	2				
6	2	1		LXI	H, 82D4	Address of add
7	D	4				or subtract
8	8	2				instruction
9	0	6		MVI	B, 8E	Enter ADC M
A	8	E				
B	F	E		CPI	15	Is key NEXT ?
C	1	5				
D	C	A		JZ	823A	If so, go store
E	3	A				ADC M in
F	8	2				subroutine
8230	0	6		MVI	B, 9E	Enter SBB M
1	9	E				
2	F	E		CPI	13	Is key STEP ?
3	1	3				
4	C	A		JZ	823A	If so, go store
5	3	A				SBB M in
6	8	2				subroutine
7	C	3		JMP	8240	Go process other
8	4	0				Key commands
9	8	2				
A	7	0		MOV	M, B	
B	C	3		JMP	8200	
C	0	0				
D	8	2				
E						
F						
8 0						
1						
2						
3						
4						
5						
6						
7						
8						FIGURE10-13

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

ADDR	CODE	Instruction	Operand	Comments
8240	23	INX	H	Address 82D5
1	06	MVI	B,00	Enter NOP
2	00			
3	FE	CPI	14	Is key RUN ?
4	14			(means HEX)
5	CA	JZ	823A	Go enter NOP
6	3A			
7	82			
8	06	MVI	B,27	Enter DAA
9	27			
A	FE	CPI	12	Is key ADDR ?
B	12			(means decimal)
C	CA	JZ	823A	Go enter DAA
D	3A			
E	82			
F	2B	DCX	H	Address 82D4
8250	06	MVI	B,00	Enter NOP in
1	00			place of ADC [illegible]
2	C3	JMP	823A	
3	3A			
4	82			
5				
6				
7				
8				
9				
A				
B				
C				
D				
E				
F				
8 0				
1				
2				
3				
4				
5				
6				
7				
8				

FIGURE 10-14

Now compare the binary and decimal operations. Enter these data:

8370 43	low byte	Augend or Minuend
71 65		
72 87		
73 09	high byte	
8390 78	low byte	Addend or Subtrahend
91 77		
92 77		
93 07	high byte	

Run your program using the steps shown below:

RUN,NEXT	Augend	0 9 8 7 6 5 4 3
(binary add)	Addend	0 7 7 7 7 7 7 8
	Sum	1 0 F E D C B B

No carries have occured except for 09.+ 07.

ADDR,NEXT	Augend	0 9 8 7 6 5 4 3
(decimal add)	Addend	0 7 7 7 7 7 7 8
	Sum	1 7 6 5 4 3 2 1

Carries have occured from all digits.

RUN, STEP	Minuend	0 9 8 7 6 5 4 3
(binary subtract)	Subtrahend	0 7 7 7 7 7 7 8
	Difference	0 2 0 F E D C B

Borrows have occured from the first and second bytes.

ADDR, STEP	Minuend	0 9 8 7 6 5 4 3
(decimal subtract)	Subtrahend	0 7 7 7 7 7 7 8
	Difference	0 2 0 9 8 7 6 5

Borrows have occured from the first five digits.

The binary to decimal correction process for addition works as follows: the addition is performed, and a flag called Auxiliary Carry is set if a carry occurs from bit 3 to bit 4 - that is, from the first digit to the second. When DAA is executed, the content of the accumulator and both Carry (CY) and Auxiliary Carry (AC) flags are tested. (Auxiliary carry is a flag which is set if a carry or borrow occurs from bit 3 to bit 4 as a result of add or subtract operations). Then the following is done:

If the value of the low four bits exceeds 9, or if AC is set, add 06 to the accumulator. These corrections occur:

ADC	07 + 08 -> 0F	no carry
DAA	0F + 06 -> 15	
ADC	08 + 08 -> 10	AC set
DAA	10 + 06 -> 16	

After this correction to the low digit, if the value of the high four bits exceeds 9 or if CY is set, add 60 to the accumulator. These corrections are made:

```
ADC     70 + 80 -> F0       no carry
DAA     F0 + 60 -> 50

ADC     80 + 80 -> 00       CY set
DAA     00 + 60 -> 60       CY still set
```

Note that when 60 is added it may set the CY but will not clear it. The following examples taken from the experiment with the program show the correction process in operation:

```
ADC     43 + 78 -> BB       no carry
DAA     BB + 06 -> C1
        C1 + 60 -> 21       sets CY

ADC     65 + 77 + CY -> DD  no carry
DAA     DD + 06 -> E3
        E3 + 60 -> 43       sets CY

ADC     87 + 77 + CY -> FF  no carry
DAA     FF + 06 -> 05       sets CY
        05 + 60 -> 65       CY still set

ADC     09 + 07 + CY -> 11  sets AC
DAA     11 + 06 -> 17
```

Caution: The DAA instruction only works correctly while the CY and AC flags are still set or cleared in response to the arithmetic instruction that produced the binary result. Any intervening arithmetic or logical instruction, or INR or DCR, affects its operation. The safe procedure is always to place DAA immediately after the instruction whose result is to be corrected.

The DAA correction is also effective in subtraction in the NEC 8080 but not in other versions of the 8080. The NEC 8080 contains another flag indicating that a subtract instruction has been executed. This modifies the action of the DAA instruction, so that carry and auxiliary carry are recognized as borrows, and DAA subtracts 06 and/or 60 as required by the content of the accumulator and carry flags.

DAA is also effective in counting up (with INR A) but not in counting down (with DCR A). To count down in decimal you must do a subtraction. You may use the subtract immediate command:

D6	SUI	Subtract the content
xx	data	of byte 2 from the accumulator.

If you want to investigate the DAA command further, the program shown in Figure 10-15 will let you try different instructions and view the results.

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

ADDR	CODE																
8200	7	D		M	O	V		A	,	L							Recover previous value
01	3	C		I	N	R		A									
02	0	0		N	O	P											Experiment with other instructions
03	0	0		N	O	P											
04	0	0		N	O	P											
05	2	7		D	A	A											Decimal Adjust
06	6	F		M	O	V		L	,	A							Save result
07	C	D		C	A	L	L		D	B	Y	T	E				Display result
08	4	5															
09	0	2															
0A	C	D		C	A	L	L		G	E	T	K	Y				Wait for a key
0B	3	D															
0C	0	2															
0D	D	2		J	N	C		8	2	0	0						Jump if command
0E	0	0															
0F	8	2															
8210	6	F		M	O	V		L	,	A							Save new key
11	A	F		X	R	A		A									Clear the flags
12	7	D		M	O	V		A	,	L							Recover the key
13	C	3		J	M	P		8	2	0	5						Go to DAA
14	0	5															
15	8	2															
16																	
17																	
18																	
19																	
1A																	
1B																	
1C																	
1D																	
1E																	
1F																	
8220																	
21																	
22																	
23																	
24																	
25																	
26																	
27																	
28																	

FIGURE 10-15

).5 BINARY MULTIPLICATION

Multiplication of integers is a process of repeated addition, or a substitute process that gives the same result.

$$3 + 3 + 3 + 3 = 12$$
$$4 \times 3 = 12$$

We have previously performed multiplication by repetitive addition. This is the easiest way, and the required program can be very short and easy to write, but it is very slow when the multiplier is large. The usual computer multiplication process is similar to what we do by hand.

Multiplicand		362
Multiplier	x	426

1972	=	6	x	362
7240	=	20	x	362
144800	=	400	x	362
Product 154012	=	426	x	362

In our familiar multiplication process we simply multiply the multiplicand by each component of the multiplier and add the individual products. Multiplication becomes trivially easy if the multiplier happens to comprise only ones and zeros:

```
      362
x     101
-------
      362        1 x 362
        0        0 x 362
    36200      100 x 362
-------
    36562
```

With binary numbers, of course, multiplication is that easy. According to whether each bit in the multiplier is zero or one, the multiplicand, appropriately shifted, is added into a partial product. Figure 10-16 shows the process, with an example of two 8-bit numbers. At most the multiplication, including any carry from the last position, will fill a 16-bit number. The flow chart shows one appropriate procedure. Write a program to implement the process. A solution is provided in Figure 10-17.

BINARY MULTIPLICATION

```
MULTIPLICAND          0 1 1 0 0 0 1 0
MULTIPLIER            0 0 1 0 0 1 1 0 ←
                                      0   0
                    0 1 1 0 0 0 1 0       1
                  0 1 1 0 0 0 1 0         1
                            0             0
                          0               0
                0 1 1 0 0 0 1 0           1
                        0                 0
                      0                   0
                0 0 0 1 1 1 0 1 0 0 0 1 1 0 0
```

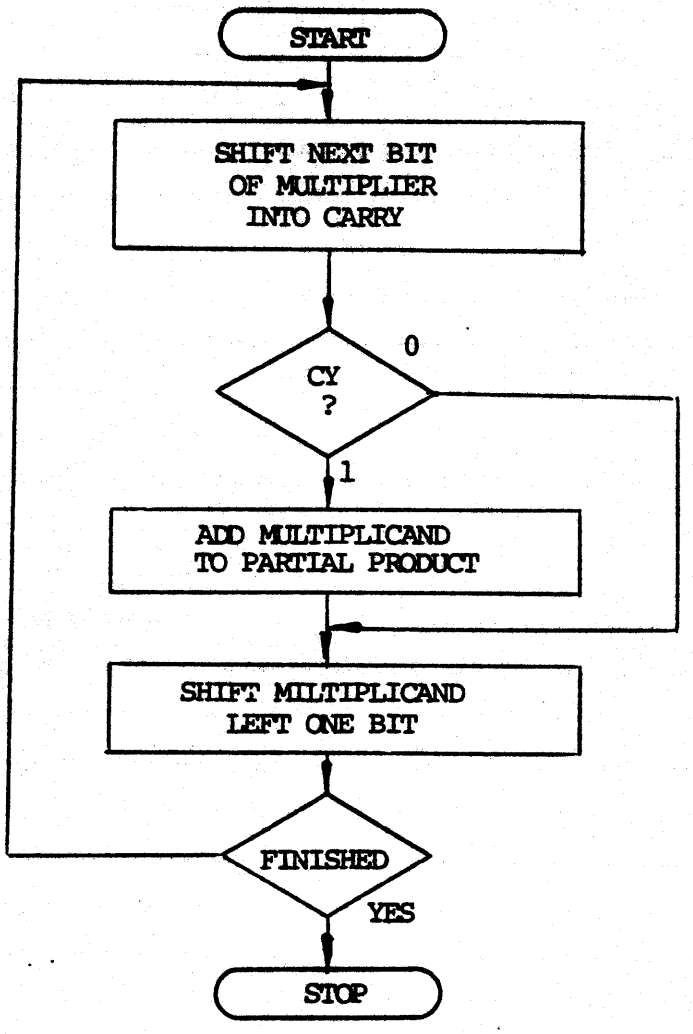


FIGURE 10-16

CODING SHEET

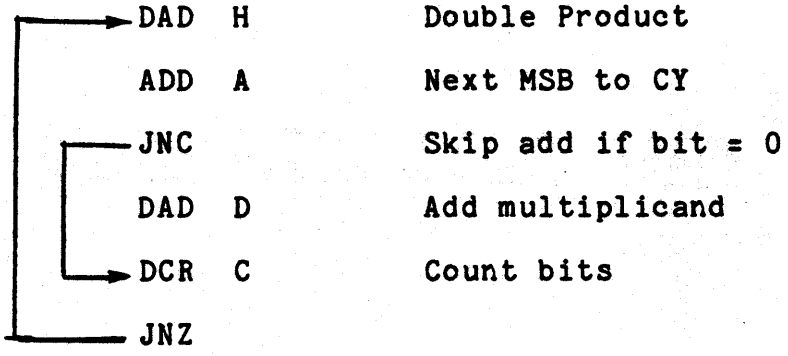
MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

ADDR	CODE															
8200	CD			C	A	L	L		E	N	T	B	Y		Get multiplicand
01	36														
02	03														
03	E5			P	U	S	H		H						Save it
04	CD			C	A	L	L		E	N	T	B	Y		Get multiplier
05	36														
06	03														
07	7D			M	O	V		A	,	L					(A) ← Multiplier
08	D1			P	O	P		D							(E) ← Multiplicand
09	21			L	X	I		H	,	0	0	0	0		Clear product
0A	00														and high byte
0B	00														of multiplicand
0C	54			M	O	V		D	,	H					
0D	1F			R	A	R									Shift multiplier
0E	D2			J	N	C		8	2	1	2				Skip add if bit = 0
0F	12														
8210	54														
11	19			D	A	D		D							Add multiplicand
12	EB			X	C	H	G								to product
13	29			D	A	D		H							Double multiplicand
14	EB			X	C	H	G								
15	B7			O	R	A		A							Test multiplier
16	C2			J	N	Z		8	2	0	D				for zero
17	0D														
18	82														
19	7D			M	O	V		A	,	L					Display product
1A	CD			C	A	L	L		D	B	Y	T	E		low byte
1B	95														
1C	02														
1D	7C			M	O	V		A	,	H					Display product
1E	CD			C	A	L	L		D	B	Y	2			high byte
1F	98														
8220	02														
21	C3			J	M	P		8	2	0	0				Go back for
22	00														input
23	82														
24															
25															
26															
27															
28															

FIGURE 10-17

There is an alternate scheme, sometimes more convenient, in which the multiplication is done backwards:



The product is developed from most significant position toward least significant, and instead of shifting the multiplicand we shift the product. The result is identical. This requires a bit counter, since the product must be shifted eight times, whereas the previous program can stop as soon as the multiplier reaches a value of zero. Figure 10-17 shows the process.

ADDR	CODE										
8200	CD		CALL		ENTBY						Get multiplicand
01	36										
02	03										
03	E5		PUSH		H						Save it
04	CD		CALL		ENTBY						Get multiplier
05	36										
06	03										
07	7D		MOV		A,L						(A) ← Multiplier
08	D1		POP		D						(E) ← Multiplicand
09	21		LXI		H,0000						Clear product
0A	00										and high byte
0B	00										of multiplicand
0C	54		MOV		D,H						
0D	0E		MVI		C,08						(C) ← Bit count
0E	08										
0F	29		DAD		H						Shift product
8210	87		ADD		A						Shift multiplier
11	D2		JNC		8215						Skip add if
12	15										multiplier bit = 0
13	82										
14	19		DAD		D						Add multiplicand
15	0D		DCR		C						to product
16	C2		JNZ		820F						Loop until 8 bits
17	0F										completed
18	82										
19	7D		MOV		A,L						Display product
1A	CD		CALL		DBYTE						low byte
1B	95										
1C	02										
1D	7C		MOV		A,H						Display product
1E	CD		CALL		DBY2						high byte
1F	98										
8220	02										
21	C3		JMP		8200						Go back for
22	00										input
23	82										
24											
25											
26											
27											
28											

FIGURE 10-17a

0.6 DECIMAL MULTIPLICATION

Basically the same procedure is used for decimal multiplication, but it must be done digit by digit instead of a byte at a time, and since decimal adjustment is necessary the additions must take place in the accumulator. It is common, but not necessary, to use unpacked decimal arithmetic (one decimal digit per byte) if multiplication and division are to be done, because it is more efficient. The decimal multiplication subroutine developed here is for packed decimal, with two digit multiplier and multiplicand and four digit result. This is the largest value that can be handled without storing data in the memory.

Figure 10-18 shows a flowchart of the subroutine, and Figures 10-19 through 10-21 the code. Like the first binary multiplication method, this shifts the multiplier right and doubles the multiplicand for each bit, stopping when the multiplier reches zero. It also requires a bit counter, initialized to four bits, because after the first digit of the multiplier has been handled the original multiplicand must be recovered and multiplied by ten for the second digit.

The program used for the binary multiplication provides the input and display functions, calling this subroutine instead of doing the arithmetic itself.

(HL) = 0000
(D) = 00
(E) = Multiplicand
(A) = Multiplier

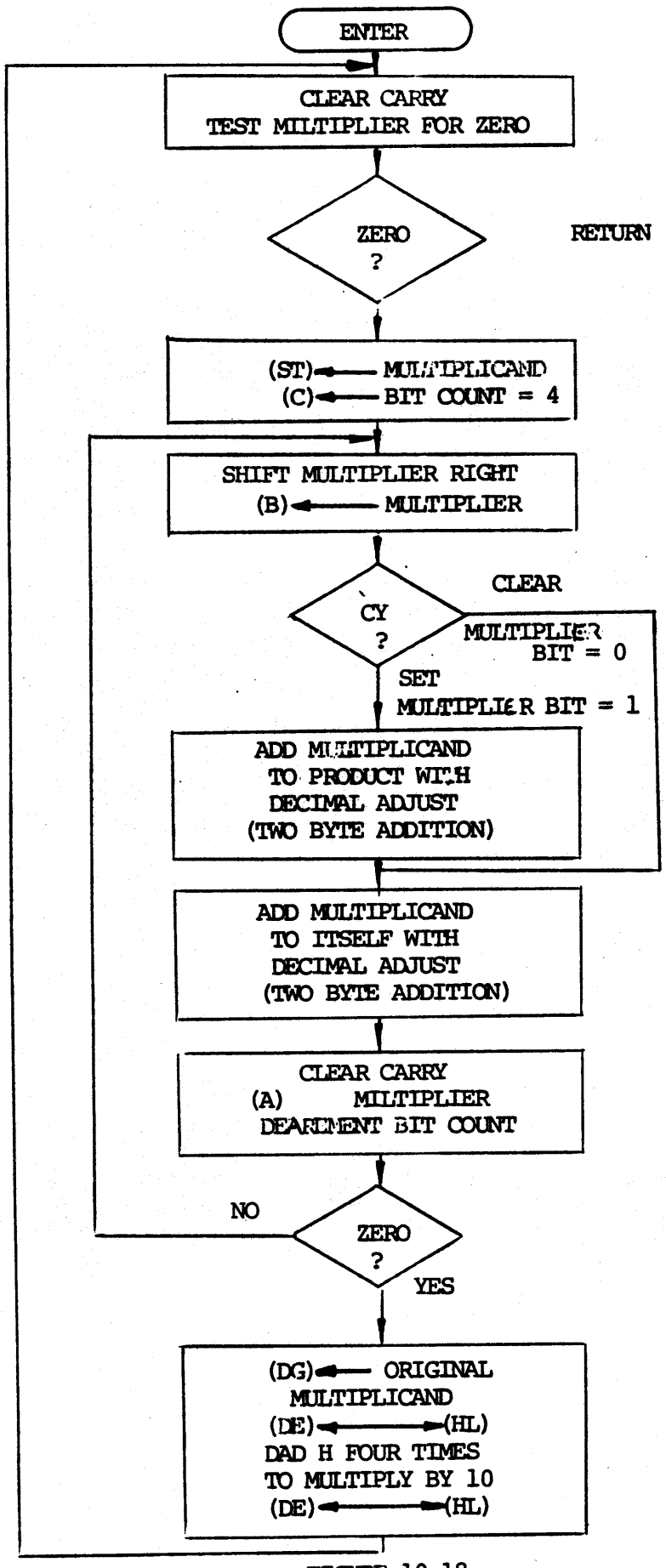


FIGURE 10-18

DATA ENTRY AND DISPLAY FOR DECIMAL MULTIPLY

ADDR	CODE		
8200	CD	CALL ENTBY	Get multiplicand
01	36		
02	03		
03	E5	PUSH H	Save it
04	CD	CALL ENTBY	Get multiplier
05	36		
06	03		
07	7D	MOV A,L	(A) ← Multiplier
08	D1	POP D	(E) ← Multiplicand
09	21	LXI H,0000	Clear product
0A	00		and high byte
0B	00		of multiplicand
0C	54	MOV D,H	
0D	CD	CALL DECMU	Decimal multiply
0E	40		subroutine
0F	82		
8210	C3	JMP 8219	
11	19		
12	82		
13			
14			
15			
16			
17			
18			
19	7D	MOV A,L	Display product
1A	CD	CALL DBYTE	low byte
1B	95		
1C	02		
1D	7C	MOV A,H	Display product
1E	CD	CALL DBYT2	high byte
1F	98		
8220	02		
21	C3	JMP 8200	Go back for
22	00		input
23	82		
24			
25			
26			
27			
28			

FIGURE 10-19

PACKED DECIMAL MULTIPLY SUBROUTINE

ADDR	CODE									Comments
8240	B7		ORA		A					Start Digit Loop
1	C8		RZ							Return if multiplier 0
2	D5		PUSH		D					(ST) ← multiplicand
3	0E		MVI		C, 04					(C) ← digit bit count
4	04									
5	1F		RAR							Start bit loop
6	47		MOV		B, A					(B) ← Multiplier
7	D2		JNC		8252					Skip add if
8	52									multiplier bit = 0
9	82									
A	7D		MOV		A, L					Add multiplicand
B	83		ADD		E					to product
C	27		DAA							
D	6F		MOV		L, A					
E	7C		MOV		A, H					
F	8A		ADC		D					
8250	27		DAA							
1	67		MOV		H, A					
8252	7B		MOV		A, E					Double multiplicand
3	83		ADD		E					
4	27		DAA							
5	5F		MOV		E, A					
6	7A		MOV		A, D					
7	8A		ADC		D					
8	27		DAA							
9	57		MOV		D, A					
A	AF		XRA		A					Clear carry
B	78		MOV		A, B					(A) ← Multiplier
C	0D		DCR		C					Decr bit count
D	C2		JNZ		8245					End of bit loop
E	45									
F	82									
8 0			CONTINUED							NEXT PAGE
1										
2										
3										
4										
5										
6										
7										
8										

FIGURE 10-20

PACKED DECIMAL MULTIPLY SUBR - CONT'D

ADDR	CODE											
8260	D1		POP		D							Restore multiplicand
1	EB		XCHG									
2	29		DAD		H							Multiply by 10
3	29		DAD		H							for high digit
4	29		DAD		H							
5	29		DAD		H							
6	EB		XCHG									
7	C3		JMP		8240							
8	40											
9	82											
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												

FIGURE 10-21

10.7 OTHER REPRESENTATIONS OF NUMBERS

There are many ways of storing numeric values in a computer, and we have used only two: binary unsigned integer and packed decimal unsigned integer. There are numerous others, including:

Binary Number Representations

Unsigned integer
Twos complement (signed binary)
Fractional, fixed binary point
Floating point

Decimal Number Representation

Packed, unsigned integer
Unpacked, unsigned integer
Hundreds Complement (signed decimal)
Tens complement (signed, unpacked)
Fractional, fixed decimal point
Floating Point

We will discuss the representation of signed numbers using twos or hundreds complement, and both fixed and floating point fractions.

10.6.1 Negative Binary Numbers

Positive integers (1, 2, 3...) are called natural numbers. They are abstractions, created for the purpose of counting objects, but they may be used to represent a physical reality. Negative numbers are a higher

level of abstraction. There are no negative quantities in the physical world.

Negative numbers are represented by convention, and are usually distinguished from positive numbers by a sign. In ordinary decimal arithmetic the minus sign is used: -10, -133 etc. To represent a negative binary number in machine form, where only two symbols (0 and 1) are available, requires a convention.

The convention that has been adopted is that of a sign bit. In arithmetic operations involving negative binary numbers, the sign bit is the most significant bit of the byte or word or set of words used to represent a number. If the bit is zero, the number is positive; if the bit is one, the number is negative:

```
0 x x x x x x x     A positive binary number
1 x x x x x x x     A negative binary number
```

If we wish to deal only with positive numbers, we do not need a sign bit.

In decimal arithmetic we may have the sequence .... -2, -1, 0, +1, +2,... The negative representation of a number is simply the number with a sign in front of it. Conversion from positive to negative simply involves changing the sign. We can do this simply with binary numbers by complementing:

+2	0 0 0 0 0 0 1 0
+1	0 0 0 0 0 0 0 1
0	0 0 0 0 0 0 0 0
-0	1 1 1 1 1 1 1 1
-1	1 1 1 1 1 1 1 0
-2	1 1 1 1 1 1 0 1

This is called a ones complement notation but unfortunately it does not meet the requirements of the basic laws of arithmetic. For example, adding +2 and -1:

```
                0 0 0 0 0 0 1 0
                1 1 1 1 1 1 1 0
              -------------------
(discard  ---> 1  0 0 0 0 0 0 0 0
carry)
```

produces a false result (zero!). The problem lies in the quantity called minus zero, which is undefined. We must represent negative binary numbers such that the basic laws of arithmetic (addition and multiplication) are valid.

This is accomplished by using a twos complement representation. The twos complement of a number is formed by complementing and adding one:

```
 1   0 0 0 0 0 0 0 1
---
 1   1 1 1 1 1 1 1 0
+1   0 0 0 0 0 0 0 1
=-1  1 1 1 1 1 1 1 1

 2   0 0 0 0 0 0 1 0
---
 2   1 1 1 1 1 1 0 1
+1   0 0 0 0 0 0 0 1
=-2  1 1 1 1 1 1 1 0
```

Now we can add +2 and -1 and obtain the correct result:

```
+2   0 0 0 0 0 0 1 0
-1   1 1 1 1 1 1 1 1
-------------------
1    0 0 0 0 0 0 0 1
```

(discard carry) →

Since in twos complement notation the high bit of the binary number indicates its sign, positive numbers range from 00 to 7F (0 to +127) and negative numbers from FF to 80 (-1 to -128). Consider these examples:

	5A	0 1 0 1 1 0 1 0
twos complement	-5A	1 0 1 0 0 1 1 0
	24	0 0 1 0 0 1 0 0
twos complement	-24	1 1 0 1 1 1 0 0

Now consider addition and subtraction:

	5A	0 1 0 1 1 0 1 0
Subtract	24	0 0 1 0 0 1 0 0
	36	0 0 1 1 0 1 1 0
	5A	0 1 0 1 1 0 1 0
Add	(-24)	1 1 0 1 1 1 0 0
	36	0 0 1 1 0 1 1 0
	24	0 0 1 0 0 1 0 0
Subtract	5A	0 1 0 1 1 0 1 0
	-36	1 1 0 0 1 0 1 0

Using twos complement representation, negative and positive numbers can be added and subtracted to obtain a signed result in twos complement notation. The sign of the result is also available in the sign flag. This is set if the high bit of the result of an arithmetic, logical or counting operation is 1, reset if the result is zero. Like the zero flag and the carry flag, it will control the action of several conditional instructions.

F2	JP	Jump if Plus
xx	low address	(if high bit is 0)
yy	high address	
FA	JM	Jump if Minus
xx	low address	(if high bit is 1)
yy	high address	
F4	CP	Call if Plus
xx	low address	
yy	high address	
FC	CM	Call if Minus
xx	low address	
yy	high address	
F0	RP	Return if Plus
F8	RM	Return if Minus

Like the other conditional instructions, these respond to a flag set by one of the arithmetic or logical instructions (also DAA, INR and DCR),

not to the present content of the accumulator.

Two's complement representation permits addition, subtraction, multiplication and division of signed numbers, giving correct results in two's complement form, correctly signed, provided that the magnitude of the result does not exceed the allowed range for the number of bits used (-128 to +127 for one byte). In many applications the programmer can be certain that the limits will not be exceeded. If results reach the limits, however, an 'arithmetic overflow' will occur.

```
  40      0 1 0 0   0 0 0 0
+ 40      0 1 0 0   0 0 0 0
-128      1 0 0 0   0 0 0 0
          └── negative
```

There are two ways of treating this problem. One is simply to provide additional capacity. If two byte numbers are used, only the highest bit of the high byte represents the sign, and values from 0 to + 32767 and - 1 to - 32768 can be represented.

```
  40      0 0 0 0   0 0 0 0   0 1 0 0   0 0 0 0
+ 40      0 0 0 0   0 0 0 0   0 1 0 0   0 0 0 0
  80      0 0 0 0   0 0 0 0   1 0 0 0   0 0 0 0
          └── still positive
```

With multiple precision arithmetic this can be carried to as many bytes as are necessary.

Another way of handling arithmetic overflow is to test for it. If two positive numbers are added and the result is negative, an overflow occurs. If two negative numbers are added producing a positive result, an overflow occurs. Subtraction of numbers with the same sign or addition of numbers with different signs cannot produce overflow. In most cases where only addition and subtraction are required, it is easier to provide additional storage capacity so that overflow cannot occur, but for multiplication and division the test for overflow is likely to be necessary.

10.6.2 Exercise

Write a program that will accept a binary number of two bytes, and on command do one of the following:

NEXT key: Store the number as entered.

STEP key: Change the sign of the number and store it.

RUN key: Subtract the number from the previously stored value.

ADDR key: Add the number to the previously stored value.

CLR key: Clear the stored value.

After each entry display the result. If the result is negative, display its twos complement with a minus sign. A flow chart and coding sheets are presented in Figure 10-22 through 10-26. Avoid destroying the decimal multiplication subroutine - it will be used again.

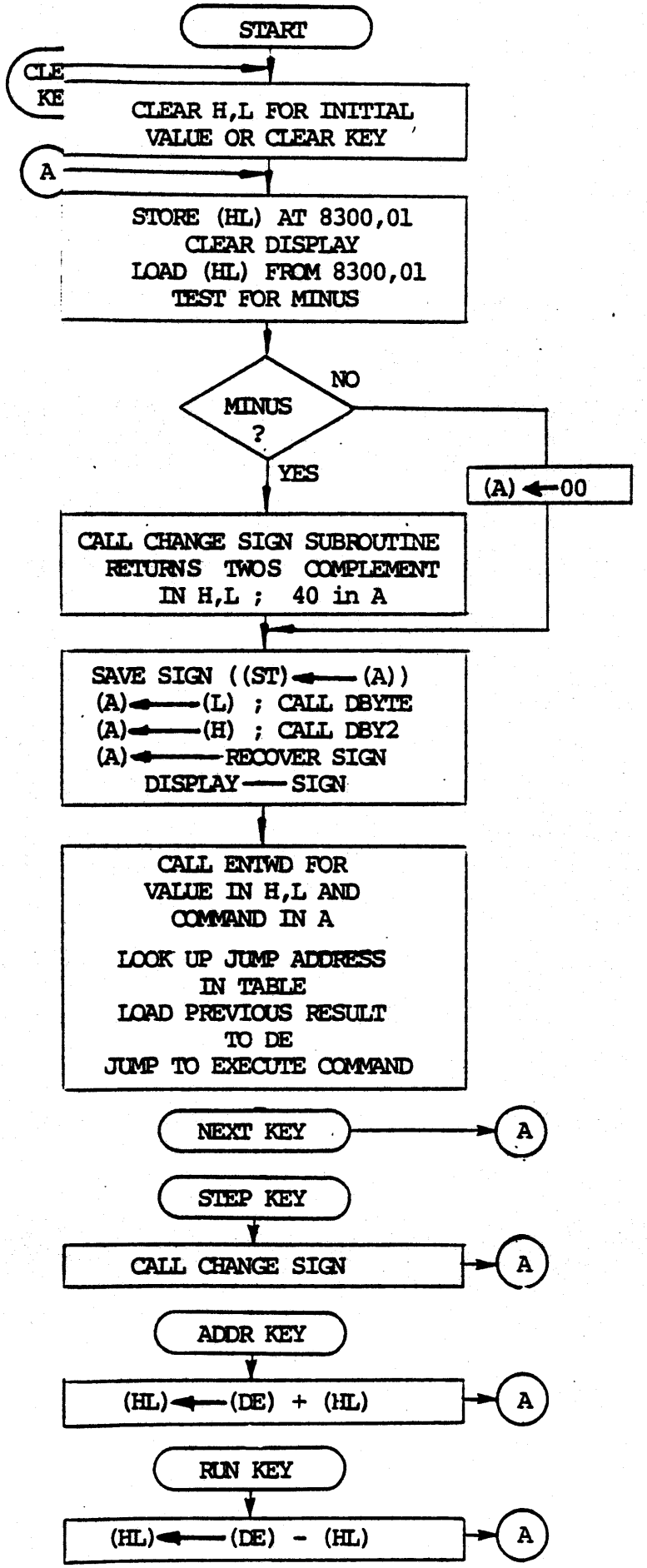

START
CLE
KE
CLEAR H,L FOR INITIAL
VALUE OR CLEAR KEY
A
STORE (HL) AT 8300,01
CLEAR DISPLAY
LOAD (HL) FROM 8300,01
TEST FOR MINUS
MINUS
?
NO
YES
(A) ←00
CALL CHANGE SIGN SUBROUTINE
RETURNS TWOS COMPLEMENT
IN H,L ; 40 in A
SAVE SIGN ((ST)←(A))
(A)←(L) ; CALL DBYTE
(A)←(H) ; CALL DBY2
(A)←RECOVER SIGN
DISPLAY←SIGN
CALL ENTWD FOR
VALUE IN H,L AND
COMMAND IN A
LOOK UP JUMP ADDRESS
IN TABLE
LOAD PREVIOUS RESULT
TO DE
JUMP TO EXECUTE COMMAND
NEXT KEY
A
STEP KEY
CALL CHANGE SIGN
A
ADDR KEY
(HL)←(DE) + (HL)
A
RUN KEY
(HL)←(DE) - (HL)
A

FIGURE 10-22

CHANGE SIGN, ADD, SUBTRACT EXERCISE

INTEGRATED COMPUTER SYSTEMS MICROCOMPUTER TRAINING SYSTEM CODING SHEET

ADDR	CODE	Mnemonic	Operand	Comment
8200	21	LXI	H, 0000	To clear initial value
01	00			
02	00			
03	22	SHLD	8300	Store result
04	00			
05	83			
06	CD	CALL	CLEAR	Clear Display
07	87			
08	02			
09	2A	LHLD	8300	Load Result
0A	00			
0B	83			
0C	AF	XRA	A	Test for minus
0D	84	ADD	H	
0E	3E	MVI	A, 00	Blank if positive
0F	00			
8210	00	NOP		
11	FC	CM	CHSIGN	Returns two's complement of HL and (A) = 40
12	90			
13	82			
14	F5	PUSH	PSW	Save symbol for sign
15	7D	MOV	A, L	(A) ← low byte
16	CD	CALL	DBYTE	Display low byte
17	95			
18	02			
19	7C	MOV	A, H	(A) ← high byte
1A	CD	CALL	DBY2	Display high byte
1B	98			
1C	02			
1D	F1	POP	PSW	Recover sign
1E	12	STAX	D	Display sign
1F	00	NOP		
8220				
21				
22				
23				
24				
25				
26				FIGURR 10-23
27				
28				

SHARE 8/31 EXERCISE — DATA ENTRY AND COMMAND INTERPRETATION

ADDR		CODE														
8220	0	CD			C	A	L	L		E	N	T	W	D		
	1	46														
	2	03														
	3	EB			X	C	H	G								(DE) ← Data
	4	21			L	X	I		H	,	8	2	2	0		Jump table address
	5	20														−10
	6	82														
	7	85			A	D	D		L							Add key value
	8	6F			M	O	V		L	,	A					(10−17)
	9	6E			M	O	V		L	,	M					Get jump address
	A	E5			P	U	S	H		H						(ST) ← jump addr
	B	2A			L	H	L	D		8	3	0	0			(HL) ← old data
	C	00			-											
	D	83														(DE) ← old data
	E	EB			X	C	H	G								(HL) ← new data
	F	C9			R	E	T									Jump to execute
8223	0	06			M	E	M									Ignore entry
	1	06			R	E	G									Ignore entry
	2	76			A	D	D	R	(	A	D	D	)			
	3	70			S	T	E	P	(	C	H	A	N	G	E	SIGN)
	4	82			R	U	N		(	S	U	B	T	R	ACT)	
	5	03			N	E	X	T	(	S	T	O	R	E	)	
	6	06			B	R	K									Ignore entry
	7	00			C	L	R		(	C	L	E	A	R	)	
	8															
	9															
	A															
	B															
	C															
	D															
	E															
	F															
8	0															
	1															
	2															
	3															
	4															
	5															
	6															
	7															
	8															

FIGURE 10-24

ADDR		CODE															
8 2 7	0	C	D		C	A	L	L		C	H	S	I	G	N	STEP KEY	
	1	9	0														
	2	8	2														
	3	C	3		J	M	P		8	2	0	3				Go store result	
	4	0	3														
	5	8	2														
	6	7	B		M	O	V		A	,	E					ADD	
	7	8	5		A	D	D		L							NOTE: DADD	
	8	0	0		N	O	P									COULD DO ALL	
	9	6	F		M	O	V		L	,	A					THIS INSTEAD	
	A	7	A		M	O	V		A	,	D						
	B	8	C		A	D	C		H								
	C	0	0		N	O	P										
	D	6	7		M	O	V		H	,	A						
	E	C	3		J	M	P		8	2	0	3					
	F	0	3														
8 2 8	0	8	2														
	1	0	0		N	O	P										
8 2 8	2	7	B		M	O	V		A	,	E					SUBTRACT	
	3	9	5		S	U	B		L								
	4	0	0		N	O	P										
	5	6	F		M	O	V		L	,	A						
	6	7	A		M	O	V		A	,	D						
	7	9	C		S	B	B		H								
	8	0	0		N	O	P										
	9	6	7		M	O	V		H	,	A						
	A	C	3		J	M	P		8	2	0	3					
	B	0	3														
	C	8	2														
	D	0	0		N	O	P										
	E	0	0		N	O	P										
	F	0	0		N	O	P										
8	0																
	1																
	2																
	3																
	4																
	5																
	6																
	7																
	8																

FIGURE 10-25

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

ADDR	CODE		INSTRUCTION	OPERAND	COMMENT
8290	7	D	MOV	A, L	CHSIGN
1	2	F	CMA		Complement
2	C	6	ADI	01	Add 1 for less
3	0	1			significant byte
4	6	F	MOV	L, A	(L) ← LSB
5	7	C	MOV	A, H	(A) ← MSB
6	2	F	CMA		
7	C	E	ACI	00	Add CY
8	0	0			
9	6	7	MOV	H, A	(H) ← MSB
A	3	E	MVI	A, 40	(A) ← minus sign
B	4	0			
C	C	9	RET		
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					FIGURE 10-26
8					

0.6.2 Signed Decimal Numbers

Packed decimal numbers can be represented with signs in hundreds complement form. In this notation -1 is represented by 99, -2 by 98 etc. To change the sign of a decimal number, the least significant byte is subtracted from 100 and succeeding bytes are subtracted from 99. In packed decimal form the range of a single byte becomes -19 to +79

+ 79	0 1 1 1 1 0 0 1
- 19	1 0 0 0 0 0 0 1

Thus signed decimal numbers must generally occupy more than one byte in order to be useful.

Hundreds complement notation applies equally well to packed or unpacked decimal representation.

Packed

+ 7999	0 1 1 1 1 0 0 1 1 0 0 1 1 0 0 1
- 1999	10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

Unpacked

+ 99	0 0 0 0 1 0 0 1 0 0 0 0 1 0 0 1
- 99	1 0 0 1 0 0 0 0 1 0 0 1 0 0 0 1

In unpacked form the high byte is 0 if the number is positive, 9 if the number is negative. Decimal arithmetic with signed numbers in hundreds complement form works correctly for all operations.

For unpacked decimal numbers the tens complement can also be used:

```
   24      = 0 0 0 0 0 0 1 0  0 0 0 0 0 1 0 0
 - 24      = 0 0 0 0 0 1 1 1  0 0 0 0 0 1 1 0
```

With unpacked decimal arithmetic the DAA instruction is not sufficient to adjust arithmetic results. If you add the two numbers above you get:

```
                 0 0 0 0 1 0 0 1  0 0 0 0 1 0 1 0
DAA gives:       0 0 0 0 1 0 0 1  0 0 0 1 0 0 0 0
```

It is necessary to test separately for a carry into the high digit of each byte and change it into a carry to the next byte. This would have corrected the result above. For this reason packed decimal arithmetic is more convenient when only addition and subtraction are involved. Modify the calculator program that was done for binary twos complement numbers to perform the same functions for packed decimal numbers, using hundreds complement.

We can also do decimal multiplication using the packed decimal multiply subroutine developed earlier. Use the MEM key to command multiply. Copy the low multiplier byte to A, clear the product (H,L), and call DECMU. This program will only give valid results if the sum or product lies within - 2000 to + 8000, and only for a two digit multiplier, but you will see that it works for negative numbers, giving a hundreds complement result. There are limitations on this, however - negative numbers must be represented as hundreds complement to as many bytes as the result is to be taken. Here we have a four digit multiplicand and a two digit multiplier. The correct result appears in the <u>low</u> four digits

of the product. If we generated higher digits they would be wrong for a negative multiplicand. A better procedure for multiplication is to convert negative numbers to sign and magnitude before multiplying, and handle the sign separately. Figures 10-27 through 10-33 illustrate the coding.

ADDR	CODE												
8200	21		LXI		H,	0000							To clear initial
01	00												value
02	00												
03	22		SHLD			8300							Store result
04	00												
05	83												
06	CD		CALL			CLEAR							Clear Display
07	87												
08	02												
09	2A		LHLD			8300							Load Result
0A	00												
0B	83												
0C	AF		XRA		A								Test for minus
0D	84		ADD		H								
0E	3E		MVI		A,	00							Blank if positive
0F	00												
8210	00		NOP										
11	FC		CM		CHSIGN								Returns 100's
12	90												complement of HL
13	82												and (A) = 40
14	F5		PUSH		PSW								Save symbol for sign
15	7D		MOV		A,	L							(A) ← low byte
16	CD		CALL		DBYTE								Display low byte
17	95												
18	02												
19	7C		MOV		A,	H							(A) ← high byte
1A	CD		CALL		DBY2								Display high byte
1B	98												
1C	02												
1D	F1		POP		PSW								Recover sign
1E	12		STAX		D								Display sign
1F	00		NOP										
8220													
21													
22													
23													
24													
25													
26													
27													
28													

FIGURE 10-27

ADDR		CODE														
8220	0	CD		CALL					ENTWD							
	1	46														
	2	03														
	3	EB		XCHG												(DE) ← Data
	4	21		LXI		H	,	8220								Jump table address
	5	20														−10
	6	82														
	7	85		ADD		L										Add key value
	8	6F		MOV		L	,	A								(10-17)
	9	6E		MOV		L	,	M								Get jump address
	A	E5		PUSH		H										(ST) ← jump addr
	B	2A		LHLD		8300										(HL) ← old data
	C	00														
	D	83														(DE) ← old data
	E	EB		XCHG												(HL) ← new data
	F	C9		RET												Jump to execute
8230	0	A6		MEM		(MULTIPLY)										
	1	06		REG												
	2	76		ADDR		(ADD)										
	3	70		STEP		(CHANGE SIGN)										
	4	82		RUN		(SUBTRACT)										
	5	03		NEXT		(STORE)										
	6	06		BRK												
	7	00		CLR		(CLEAR)										
	8															
	9															
	A															
	B															
	C															
	D															
	E															
	F															
8	0															
	1															
	2															
	3															
	4															
	5															
	6															
	7															
	8															

FOGURE 10-28

PACKED DECIMAL MULTIPLY SUBROUTINE

ADDR		CODE															
8 24	0	B	7		O	R	A		A								Start Digit Loop
	1	C	8		R	Z											Return if multiplier 0
	2	D	5		P	U	S	H		D							(ST) ← multiplicand
	3	0	E		M	V	I		C	,	0	4					(C) ← digit bit count
	4	0	4														
	5	1	F		R	A	R										Start bit loop
	6	4	7		M	O	V		B	,	A						(B) ← Multiplier
	7	D	2		J	N	C		8	2	5	2					Skip add if
	8	5	2														multiplier bit = 0
	9	8	2														
	A	7	D		M	O	V		A	,	L						Add multiplicand
	B	8	3		A	D	D		E								to product
	C	2	7		D	A	A										
	D	6	F		M	O	V		L	,	A						
	E	7	C		M	O	V		A	,	H						
	F	8	A		A	D	C		D								
8 25	0	2	7		D	A	A										
	1	6	7		M	O	V		H	,	A						
825	2	7	B		M	O	V		A	,	E						Double multiplicand
	3	8	3		A	D	D		E								
	4	2	7		D	A	A										
	5	5	F		M	O	V		E	,	A						
	6	7	A		M	O	V		A	,	D						
	7	8	A		A	D	C		D								
	8	2	7		D	A	A										
	9	5	7		M	O	V		D	,	A						
	A	A	F		X	R	A		A								Clear carry
	B	7	8		M	O	V		A	,	B						(A) ← Multiplier
	C	0	D		D	C	R		C								Decr bit count
	D	C	2		J	N	Z		8	2	4	5					End of bit loop
	E	4	5														
	F	8	2														
8	0				C	O	N	T	I	N	U	E	D				NEXT PAGE
	1																
	2																
	3																
	4																
	5																
	6																
	7																FIGURE10-29
	8																

PACKED DECIMAL MULTIPLY SUBR - CONTD

ADDR		CODE														
8	26	0	D1		POP		D									Restore multiplicand
		1	EB		XCHG											
		2	29		DAD		H									Multiply by 10
		3	29		DAD		H									for high digit
		4	29		DAD		H									
		5	29		DAD		H									
		6	EB		XCHG											
		7	C3		JMP		8240									
		8	40													
		9	82													
		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 10-30

COMMAND EXECUTION — PACKED DECIMAL

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

ADDR	CODE														STEP KCY
8270	C	D		C	A	L	L		C	H	S	I	G	N	STEP KCY
1	9	0													
2	8	2													
3	C	3		J	M	P		8	2	0	3				Go store result
4	0	3													
5	8	2													
6	7	B		M	O	V		A	,	E					ADD
7	8	5		A	D	D		L							
8	2	7		D	A	A									DAA for Decimal
9	6	F		M	O	V		L	,	A					
A	7	A		M	O	V		A	,	D					
B	8	C		A	D	C		H							
C	2	7		D	A	A									DAA for Decimal
D	6	7		M	O	V		H	,	A					
E	C	3		J	M	P		8	2	0	3				
F	0	3													
8 0	8	2													
1	0	0		N	O	P									
2	7	B		M	O	V		A	,	E					SUBTRACT
3	9	5		S	U	B		L							
4	2	7		D	A	A									DAA for Decimal
5	6	F		M	O	V		L	,	A					
6	7	A		M	O	V		A	,	D					
7	9	C		S	B	B		H							
8	2	7		D	A	A									DAA for Decimal
9	6	7		M	O	V		H	,	A					
A	C	3		J	M	P		8	2	0	3				
B	0	3													
C	8	2													
D	0	0		N	O	P									
E	0	0		N	O	P									
F	0	0		N	O	P									
8 0															
1															
2															
3															
4															
5															
6															
7															
8															

FIGURE 10-31

HUNDRED'S COMPLEMENT - WITH NEC 8080

10-65

ADDR	CODE													
8 29 0	A	F		X	R	A		A						Subtract low
1	9	5		S	U	B		L						byte from 00
2	2	7		D	A	A								Decimal adjust
3	6	F		M	O	V		L	,	A				
4	3	E		M	V	I		A	,	0	0			Subtract high byte
5	0	0												and borrow
6	9	C		S	B	B		H						from 00
7	2	7		D	A	A								Decimal adjust
8	6	7		M	O	V		H	,	A				
9	3	E		M	V	I		A	,	4	0			(A) ← Minus sign
A	4	0												
B	C	9		R	E	T								
C				V	A	L	I	D		F	O	R		NEC 8080
D					N	O	T		F	O	R		I	NTEL
E														
F														
8 2A 0														
1														
2														
3														
4														
5				E	X	E	C	U	T	E		M	U	LTIPLY
82A 6	7	D		M	O	V		A	,	L				(A) ← Multiplier
7	2	1		L	X	I		H	,	0	0	0	0	Clear product
8	0	0												
9	0	0												
A	C	D		C	A	L	L		D	E	C	M	U	Multiply
B	4	0												
C	8	2												
D	C	3		J	M	P		8	2	0	3			Go to store
E	0	3												and display
F	8	2												
8 0														
1														
2														
3														
4														
5														
6														
7														
8														

FIGURE 10-32

HUNDRED'S COMPLEMENT - VALID FOR INTEL

ADDR		CODE														
8	29	0	3	E		M	V	I		A	,	9	A			Represents 100
		1	9	A												
		2	9	5		S	U	B		L						Subtract LSD
		3	C	6		A	D	I		0	0					Add 0 to make
		4	0	0												DAA valid
		5	2	7		D	A	A								Decimal Adjust
		6	6	F		M	O	V		L	,	A				Save LSD
		7	3	E		M	V	I		A	,	9	9			Subtract higher
		8	9	9												digits from 99
		9	C	E		A	C	I		0	0					but add in carry
		A	0	0												from lower digits
		B	9	4		S	U	B		H						Subtract higher digit
		C	C	6		A	D	I		0	0					Add 0 to make
		D	0	0												DAA valid
		E	2	7		D	A	A								Decimal Adjust
		F	6	7		M	O	V		H	,	A				Save higher digit
8	2A	0	3	E		M	V	I		A	,	4	0			(A) ← minus sign
		1	4	0												
		2	C	9		R	E	T								
		3	0	0												
		4	0	0												
		5	0	0		E	X	E	C	U	T	E		M	U	LTIPLY
		6	7	D		M	O	V		A	,	L				
		7	2	1		L	X	I		H	,	0	0	0	0	
		8	0	0												
		9	0	0												
		A	C	D		C	A	L	L		D	E	C	M	U	
		B	4	0												
		C	8	2												
		D	C	3		J	M	P		8	2	0	3			
		E	0	3												
		F	8	2												
8		0														
		1														
		2														
		3														
		4														
		5														
		6														
		7														
		8														

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

FIGURE 10 -33

0.6.3 Fractional Numbers

A fractional value in the decimal number system is expressed by digits to the right of a decimal point.

0.1 = 1/10

0.01 = 1/100

0.11 = 1/10 + 1/100 = 11/100

In the binary number system fractional values are also expressed by digits to the right of a binary point.

$0.1 = 1/2 = 1/10_2$

$0.01 = 1/4 = 1/100_2$

$0.11 = 1/2 + 1/4 = 3/4 = 11_2/100_2$

The beauty of this representation is that all the arithmetic operations of integer numbers apply equally to fractional numbers and mixed numbers.

3 10/16	0 0 1 1 . 1 0 1 0
+ 4 7/16	+ 0 1 0 0 . 0 1 1 1
= 8 1/16	= 1 0 0 0 . 0 0 0 1

Twos complement, tens complement, and hundreds complement still work with fractional values.

- 3 10/16	1 1 0 0 . 0 1 1 0
+ 4 7/16	0 1 0 0 . 0 1 1 1
= 0 13/16	0 0 0 0 . 1 1 0 1

Computers use two binary point systems: fixed point and floating point. The examples above are fixed point. Each number has its binary point in the same place. Generally multi-byte precision is needed in real problems, and the binary point lies between two of the bytes. A four byte number can represent any value from - 32768.0 to + 32767.9999847 with a precision of .0000152 (one part in 65536).

For many purposes floating point numbers are much more satisfactory. This is equivalent to scientific notation with the number represented as a fraction times the number system base raised to a power.

$$0.9876 \times 10^{4} = 9876$$

To avoid the difficulties of showing exponents in print this is often shown as:

0.9876E04

where E represents '10 with exponent'.

Scientific notation is very convenient for multiplication and division. The two fractions are multiplied (or divided) and the exponents are added (or subtracted).

$$\begin{array}{rl} & 0.9000\ E04 \\ \times & 0.2000\ E02 \\ \hline = & 0.1800\ E06 \end{array}$$

For addition and subtraction, however, the numbers must be converted to fixed point format.

```
    0.9000 E04 = 9000.0000
+   0.2000 E02 = 0020.0000
=   0.9020 E04 = 9020.0000
```

In a computer as on paper, the fraction (or mantissa) must be stored separately from the exponent. Each can be positive or negative, and expressed in twos, tens or hundreds complement form. Generally a computing system that is doing floating point arithmetic will operate in binary form, converting from decimal at input and to decimal at output.

Decimal/binary/decimal conversions are treated in Appendix D.

MICROCOMPUTER TRAINING WORKBOOK

CHAPTER 11

REVIEW OF INSTRUCTIONS

NOTE:

Your Microcomputer Training System may include the NEC 8080AF, which is logically identical to the Intel 8080A. Comments in this chapter regarding the Intel device should be understood to refer also to the NEC 8080AF. Such comments occur on page 11-6, 11-8, 11-10 and 11-11. The Intel 8080A and the NEC 8080AF do not have a subtract flag.

1. REVIEW OF INSTRUCTIONS

You have now met all of the instructions of the 8080, and actually used most of them. We will review the instruction set and look at the code structure and flags. The instructions can be divided into several categories:

a) Data Transfer Instructions

b) Counting Instructions

c) Accumulator/Carry Instructions

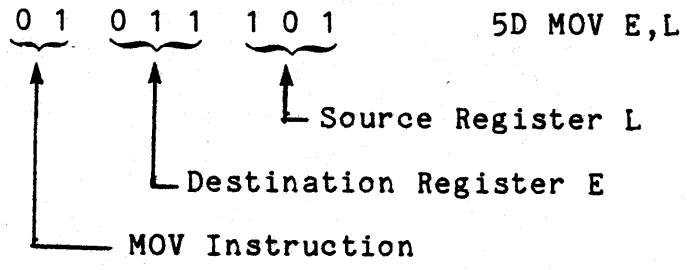
d) Arithmetic and Logical Instructions

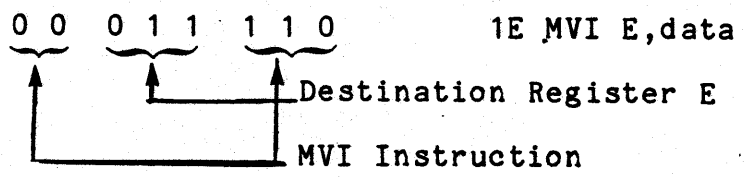
e) Branch Instructions

f) Input/Output Instructions

11.1 DATA TRANSFER

Data transfer instructions include MOV, MVI, STA, etc. All register reference instructions in the 8080 comform to a pattern in which three bits identify a source, or else a different three bits identify a destination, or both.





Other data transfer instructions are the eight instructions that load and store the accumulator and register pair H,L:

3A	LDA	yyxx	32	STA	yyxx
0A	LDAX	B	02	STAX	B
1A	LDAX	D	12	STAX	D
2A	LHLD	yyxx	22	STHL	yyxx

The four LXI instructions:

01	LXI	B
11	LXI	D
21	LXI	H
31	LXI	SP

The stack instructions:

C5	PUSH	B	C1	POP	B
D5	PUSH	D	D1	POP	D
E5	PUSH	H	E1	POP	H
F5	PUSH	PSW	F1	POP	PSW

The register pair transfer instructions:

EB	XCHG	(DE) <-> (HL)
E3	XTHL	(ST) <-> (HL)
F9	SPHL	(SP) <- (HL)
E9	PCHL	(PC) <- (HL)

The 8080 has an abundance of data transfer instructions, yet is lacking three needed functions that therefore require multiple instructions:

a) Exchange BC with HL

```
PUSH    B              (BC) <-> (HL)
PUSH    H
POP     B
POP     H
```

b) Initialize the stack to a new location and push the old stack pointer into the new stack.

```
LXI     H,0000
DAD     SP
LXI     SP,new location
PUSH    H
```

It is easier to restore the old value:

```
POP     H
SPHL
```

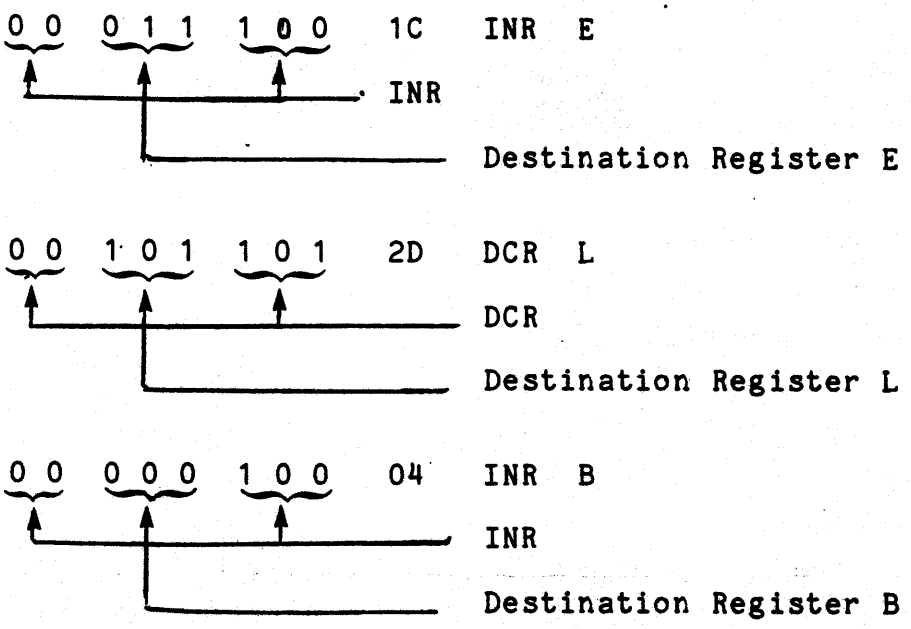
c) Save all registers and flags.

Some microprocessors have a single command that pushes all registers into the stack; others, such as the Intel 4040, have a duplicate set of registers. In the 8080 four instructions are needed.

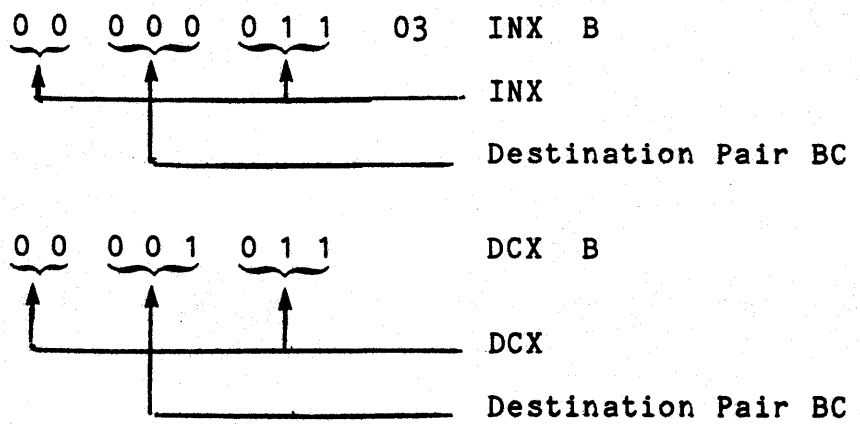
Data transfer instructions do not affect any flags (Except POP PSW, which restores the flags to the state when PUSH PSW was executed).

.2 COUNTING INSTRUCTIONS

The INR and DCR instructions use the same register identification that appears in MOV.



The structure is modified for register pair instruction



The counting instructions affect flags as follows:

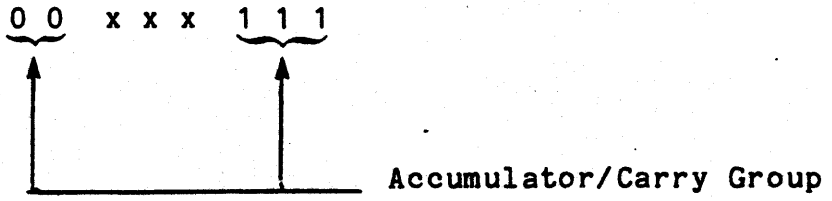
INX:	No flags
DCX:	No flags
INR:	Set or clear zero, sign, parity
	Does not affect carry
	* Does not affect auxilliary carry
	* Clears subtract
DCR:	Set or clear zero, sign, parity
	Does not affect carry
	* Does not affect auxilliary carry
	* Sets subtract

* These statements apply to the NEC 8080, not to the Intel 8080.

Zero, sign and parity flags may be used to cause a conditional branch as a result of INR or DCR. INR or DCR may be used in a loop with ADC or SBB instructions, since carry is preserved.

1.3 ACCUMULATOR/CARRY INSTRUCTIONS

These instructions affect only the accumulator and flags. The instruction format is:



0 0	0 0 0	1 1 1	07	RLC
0 0	0 0 1	1 1 1	0F	RRC
0 0	0 1 0	1 1 1	17	RAL
0 0	0 1 1	1 1 1	1F	RAR
0 0	1 0 0	1 1 1	27	DAA
0 0	1 0 1	1 1 1	2F	CMA
0 0	1 1 0	1 1 1	37	STC
0 0	1 1 1	1 1 1	3F	CMC

The rotate instructions shift the accumulator left or right.

RLC Copies bit 7 to bit 0 and CY and shifts other bits left.

RRC Copies bit 0 to bit 7 and CY and shifts other bits right. Previous carry is lost.

RAL Copies bit 7 to CY, CY to bit 0 and shifts other bits left

RAR Copies bit 0 to CY, CY to bit 7 and shifts other bits right

STC Sets carry

CMC Complements carry

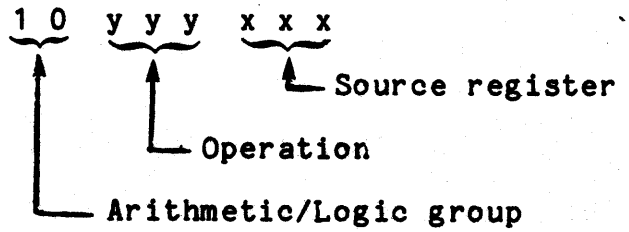
These instructions do not affect any flags except carry, even though execution may result in the accumulator containing zero or having a different sign or parity condition. To set or clear the flags to correspond to the content of the accumulator you must execute a logical or arithmetic instruction.

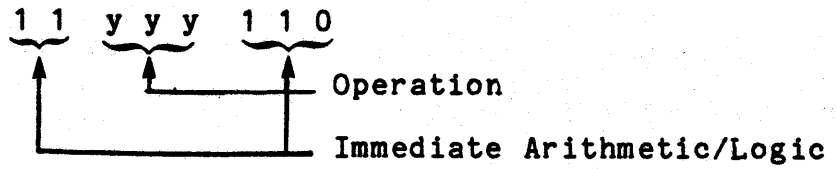
CMA complements the accumulator but affects no flags.

DAA corrects the result of an add or subtract (NEC 8080 only) to decimal. It affects sign, zero, parity and carry flags. It does not affect subtract or auxiliary carry flags, in the NEC 8080.

1.4 ARITHMETIC AND LOGICAL INSTRUCTIONS

There are eight types of instructions and each has nine possible sources: the seven registers, the memory location addressed by (HL), and the program memory (the immediate instructions). As in the MOV instructions the three low bits designate the source, the next three bits specify which of the instructions is intended:





The operations designated by bits 5, 4, 3, are:

1 0	0 0 0	x x x	ADD	(A) <- (A) + (r)
1 0	0 0 1	x x x	ADC	(A) <- (A) + (r) + (CY)
1 0	0 1 0	x x x	SUB	(A) <- (A) - (r)
1 0	0 1 1	x x x	SBB	(A) <- (A) - (r) - (CY)
1 0	1 0 0	x x x	ANA	(A) <- (A) AND (r)
1 0	1 0 1	x x x	XRA	(A) <- (A) XOR (r)
1 0	1 1 0	x x x	ORA	(8A) <- (A) OR (r)
1 0	1 1 1	x x x	CMP	(see below)

The same coding for the operation applies to the immediate instructions.

CMP r (or CMP M) performs a subtract operation and sets or clears the flags appropriately, but discards the result instead of storing it in the accumulator.

The four DAD instructions are also included in the arithmetic group. They are:

```
09   DAD  B        (HL) <- (HL) + (BC)
19   DAD  D        (HL) <- (HL) + (DE)
29   DAD  H        (HL) <- (HL) + (HL)
39   DAD  SP       (HL) <- (HL) + (SP)
```

These instructions affect only the carry flag (and in the NEC 8080 they clear the subtract flag). They can be used both for double precision arithmetic and to index a memory address. The latter is especially useful when operations are to be performed on bytes that are spaced from each other by some fixed or variable distance.

11.4.1 The Flags

The flag register (Program Status Word, PSW) contains 6 bits in the NEC 8080 (5 bits in the Intel 8080). These are arranged as indicated below.

Bit	7	6	5	4	3	2	1	0
NEC	Sign	Zero	Sub	AC	1	Par	1	CY
Intel	Sign	Zero	0	AC	0	Par	1	CY

The following list summarizes how these are affected by the various instructions:

Sign: Set if the high bit of the result is 1, cleared if 0, by the following instructions:

INR, DCR, DAA

Any arithmetic or logical instruction.

Zero: Set if the result is zero, cleared if not, by:

INR, DCR, DAA

Any arithmetic or logical instruction

Parity: Set if parity of the reslt is even, cleared if odd, by:

INR, DCR, DAA

Any arithmetic or logical instruction

Subtract: (NEC 8080 only):

Set by SUB, SBB, SUI, SBI, CMP, DCR

Cleared by ADD, ADC, ADI, ACI, DAD, INR

Auxiliary Carry: Set if a carry or borrow occurs from bit 3 to bit 4 as a result of: ADD, ADC, ADI, ACI, SUB, SBB, SUI, SBI, CMP. The same instructions clear it if the digit carry does not occur. It is not affected by shifts or logical or count instructions.

Carry Set or cleared by any shift or arithmetic operation, including CMP, DAD and DAA. Cleared by any of the logical instructions ANA, ORA, XRA. Set by STC; complemented by CMC. Not affected by count instructions.

11.5 BRANCH INSTRUCTIONS

Jump, Call Return, Restart and PCHL are the branch instructions.

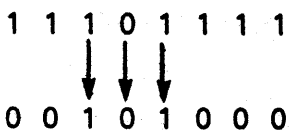
1 1 0 0 0 0 1 1	C3	JMP
1 1 0 0 1 0 0 1	C9	RET
1 1 0 0 1 1 0 1	CD	CALL
1 1 1 0 1 0 0 1	E9	PCHL

All of the branch instructions include 11 as the two high bits (bits 7 and 6) of the instruction. The three low bits distinguish among the branch and conditional branch and various non-branching instructions. The conditional branches use bits 5 and 4 to determine which flag is to be tested and bit 3 to indicate whether the jump is to be executed when the flag is set or when it is clear.

1 1 x x x 0 1 0	Conditional Jump
1 1 x x x 1 0 0	Conditional Call
1 1 x x x 0 0 0	Conditional Return
0 0 0	If not Zero
0 0 1	If Zero
0 1 0	If not Carry
0 1 1	If Carry
1 0 0	If Parity Odd
1 0 1	If Parity Even
1 1 0	If Plus
1 1 1	If Minus

The Restart instructions use the three bits 5, 4 and 3 as part of the address for the single byte CALL. They are copied into the corresponding three bits of the program counter while the remaining bits are all set to zero. For instance, RST 5 jumps to 0028:

EF RST 5



11.6 INPUT/OUTPUT

DB IN
xx port address

D3 OUT
xx port address

The port address is copied to both the high eight bits and the low eight bits of the address bus. I/O Read or I/O Write is activated. The CPU copies the data bus to (A) on input; copies (A) to the data bus on output.

FB Enable Interrupt
F3 Disable Interrupt

Set or clear the internal interrupt enabled flip-flop. EI is not effective until one instruction following EI has been executed.

MICROCOMPUTER TRAINING WORKBOOK

APPENDIX A

THE ICS MONITOR

ICS MICROCOMPUTER TRAINING SYSTEM DESCRIPTION

1.1 The ICS Microcomputer Training System uses the NEC 8080A microprocessor.

1.2 There are 1024 bytes of Eraseable ROM (NEC 454) located at addresses 0000 to 03FF and 512 bytes of CMOS RAM at addresses 8200 to 83FF. RAM is expandable by another 512 bytes at 8000 to 81FF.

1.3 An 8255 Programmable Peripheral Interface chip is provided for Input/Output.

1.4 Ram is switched to battery power when the PROTECT/ENABLE switch is set to protect.

1.5 A Keyboard is provided with 25 keys. The top right key gives a reset signal to the 8080A. The other switches provide input to the 8255.

1.6 A display is provided with eight digit positions. This is driven by DMA using the contents of addresses 83F8 through 83FF for digit positions 1 through 8.

1.7 The complete instruction set for the 8080A is given in the 8080 Microcomputer Systems User's Manual, together with detailed specifications of the machine's internal state during instruction execution and a description of all registers.

1.8 The MTS board layout is shown in Figure A-1. A block diagram is presented in Figure A-2. The complete circuit diagram appears in Appendix C.

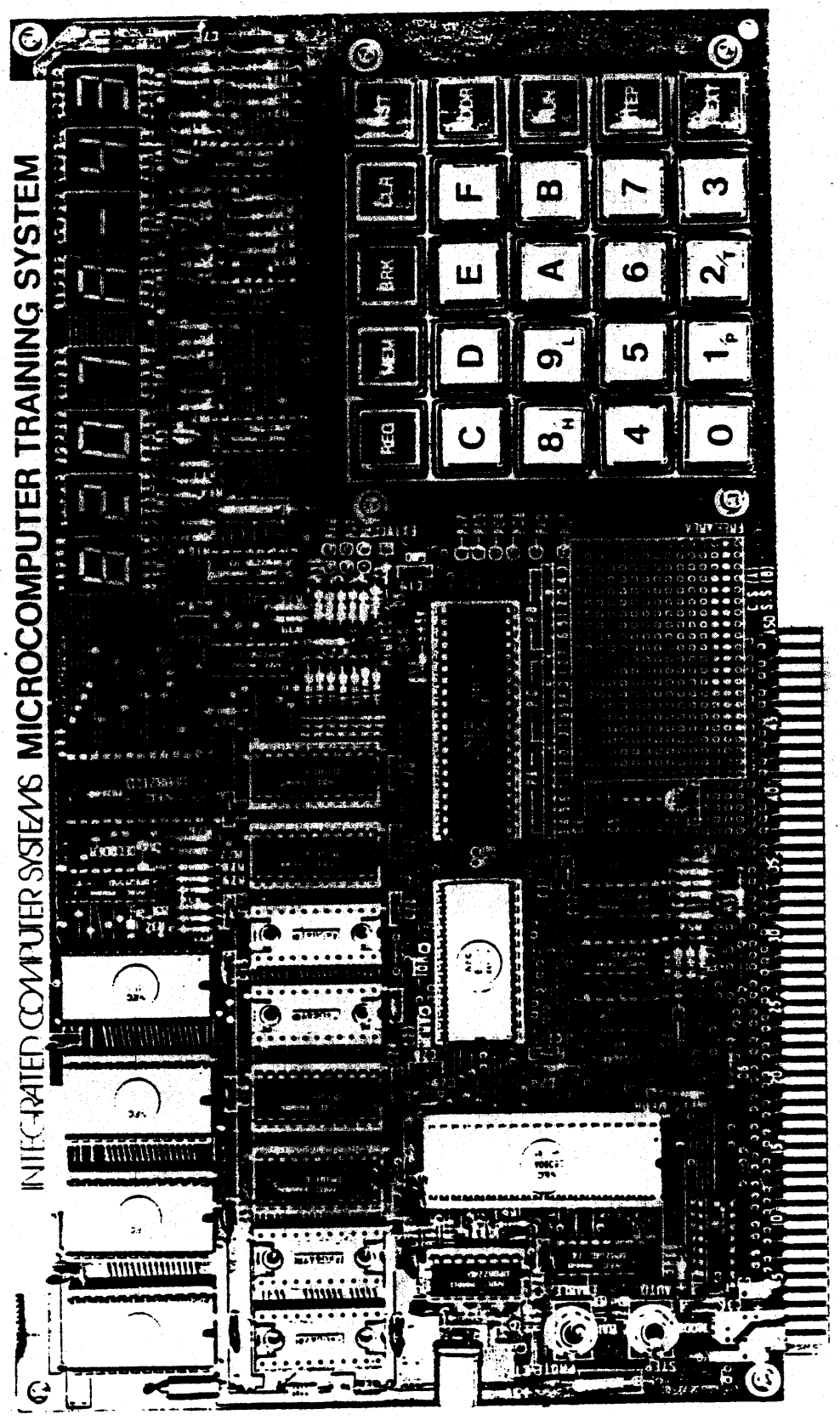
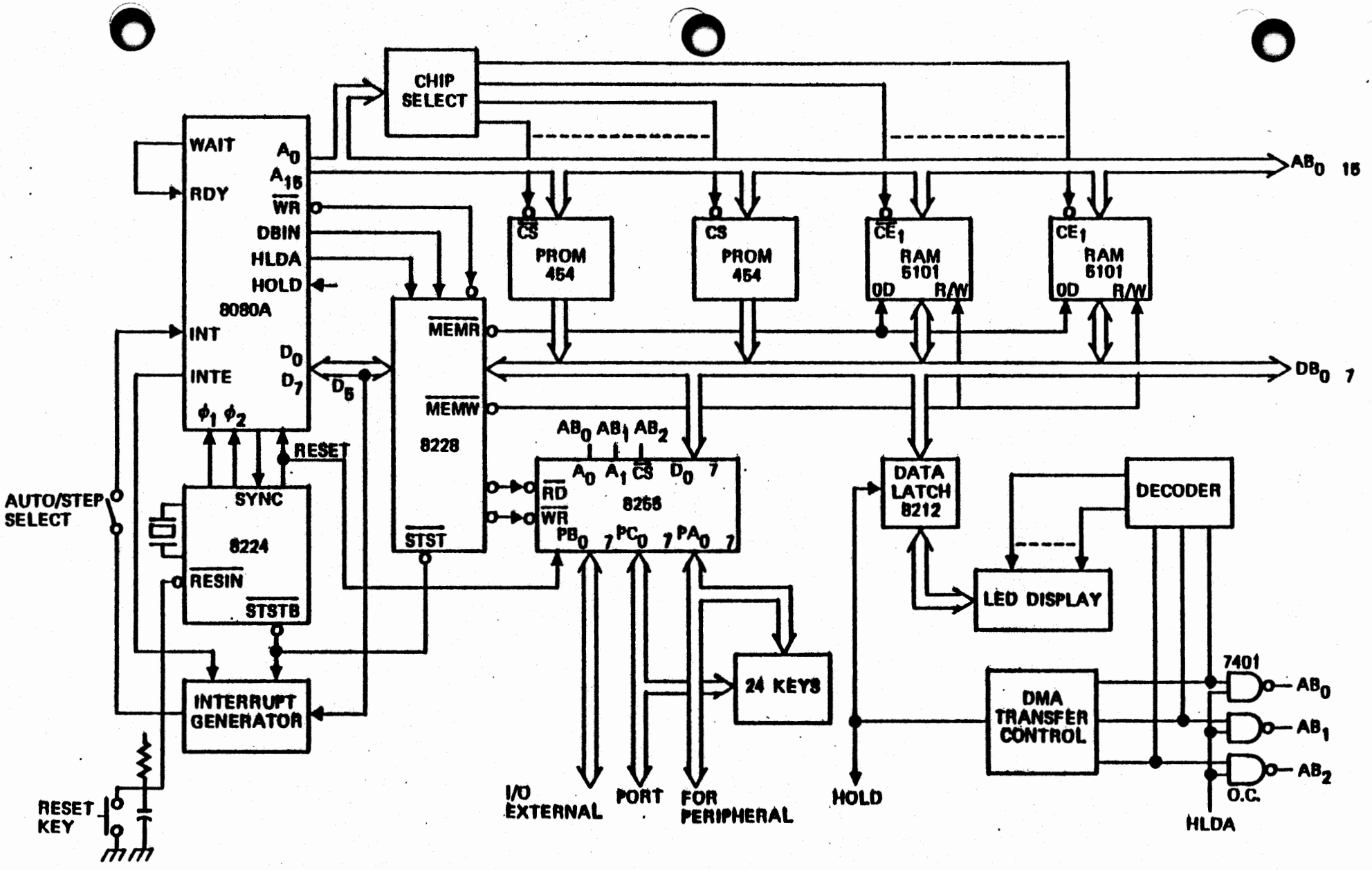

INTEGRATED COMPUTER SYSTEMS MICROCOMPUTER TRAINING SYSTEM
RESET
MEM
BRK
GO
C
D
E
F
8H
9L
A
B
4
5
6
7
0
1P
2/1
3

FIGURE A1



MICROCOMPUTER TRAINING SYSTEM CONFIGURATION

FIGURE A2

2. GENERAL MONITOR FUNCTIONS

The monitor provides five general functions:

Load memory from keyboard

Store program on tape

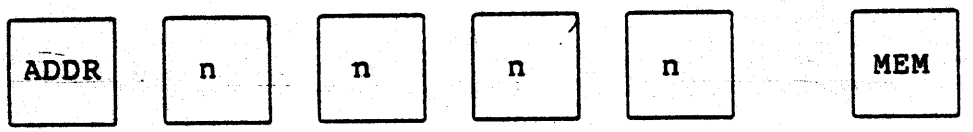
Load program from tape

Operator program in debug mode

Run user program

2.1 Load Memory from keyboard.

2.1.1 To select a memory address, press



(where nnnn is the address: eg ADDR 8200 MEM)
The address will appear in the left four digits, and its present contents will appear in the right two digits.

2.1.2 To enter data to memory, (after pressing MEM) key in two digits. They will replace the two digits at the right.

2.1.3 To confirm those data and proceed to the next memory address, press NEXT

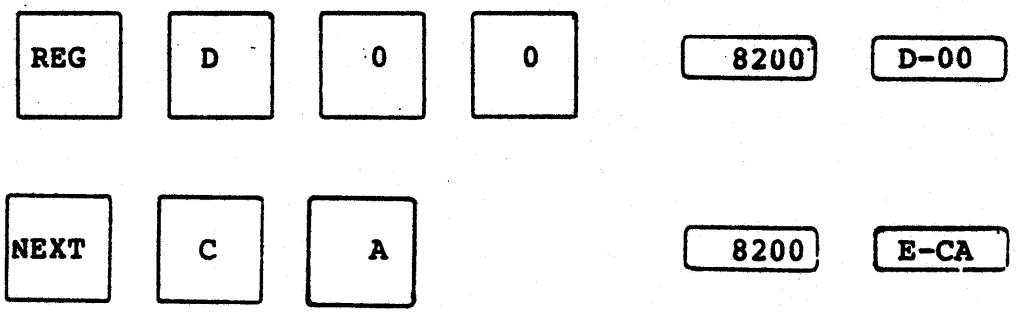
2.1.3 If an error is made and detected before pressing any command key, press [CLR]. This will restore the old data.

2.1.4 Pressing any other command key will confirm the new data. The command will then be processed.

2.2 Store program on tape.

The program SER0T copies binary data from memory to a serial recording medium. An external oscillator and modulator are required for recording on an audio tape cassette recorder. Data are output with 12 bits per memory byte: start bit (0); least significant data bit; successive data bits (8 data bits total); and three bit periods of stop signal (1).

The procedure is to use monitor commands to load the starting address in register pair H,L and the number of bytes to be transmitted in register pair D,E. Then the program starting address is entered by use of the address setting and run procedure. For example, to record 8300 - 83C9:



NEXT	8	3		8200	H-83
NEXT	0	0		8200	L-00
ADDR	0	3	7	5	0375 F3

Turn recorder on

RUN (The display will go off)

At end of transmission the display will show:

039E F3

Turn recorder off.

2.3 Load Program from Tape

The program SERIN loads binary data from a serial recording medium into memory. It is complementary to SER0T: it receives data in the format described above. An external demodulator is required for reading from an audio tape cassette.

The procedure is to load a starting address into register pair H,L; enter the program starting address; start tape; RUN

REG	8/H	8	2	8200	H-82
NEXT	0	0		8200	L-00
ADDR	0	3	B	1	03B1	F3

Turn Recorder on, and wait for about 10 seconds.

RUN (Display goes blank.)

When the display reappears, turn tape off.

A reliable cassette modem using frequency shift keying is shown on page 9-44.

2.4 Operating in Debug Mode

The monitor provides for tracing the flow and results of a user's program. The STEP/AUTO toggle switch must be set to STEP; after each user instruction is executed a hardware interrupt is generated. This causes an entry to the monitor.

Operation of the user's program is initiated by the STEP command or the RUN command. A flag byte (SFLAG) is stored by the monitor when the STEP or RUN key is pressed. This flag determines the procedure to be followed at the next entry to the monitor. With either command the user's program is interrupted at each instruction, but in RUN the return to user is automatic unless a breakpoint is encountered.

If the initiating command was STEP, the monitor activates the keyboard after each user instruction is executed.

If the initiating command was RUN, the monitor tests whether the user's program counter is equal to any of up to eight breakpoints entered by the the user. If not it returns control to the user's program immediately. When a breakpoint is encountered the monitor tests a counter associated with the breakpoint: if non zero it decrements the counter and returns, but if the counter is zero the keyboard and display are activated.

When the display is active under monitor control, it shows an address in display positions 1-4 (the left four digits) and a data byte in positions 7 and 8 (the right two digits). At entry to the monitor the address displayed is the program counter, and the data are either the next instruction or the contents of a register. The latter is identified in digits 5 and 6.

The user may request many other displays, such as another register, another address in memory, a register pair and the contents of the addressed location, the stack pointer, or the user's subroutine return address. These are described in detail in section 3.

3. MONITOR COMMANDS

The major sections of the monitor operate as an interrupt service routine entered by a hardware interrupt automatically generated as each user instruction is executed, provided that the AUTO/STEP switch is in the STEP position.

The user may program entry to the monitor by including the RST4 instruction (E7) in his program. He may alter addresses and flags used by the monitor through his own program, thereby affecting monitor functions. Various monitor subroutines are accessible to the user by normal subroutine calls.

3.1 Monitor Entry

When the monitor is entered by interrupt (RST7) or by programmed call (RST4) the user's registers, program counter, and stack pointer are saved in memory and may be accessed by monitor commands.

The RESET key causes a hardware reset to the 8080. The user's program counter and stack are lost, but his registers are saved. The stack pointer is initialized to 83D3; the memory address and user's program counter are initialized to 8200.

3.2 Monitor Data Storage

At entry to the monitor the user's program counter is popped from the stack and stored at PCADDR. The 8080 registers are pushed onto the stack. If the conditions are met for activating the

keyboard and display the user's stack pointer is stored at SPADDR.

In addition, the monitor stores two addresses and two indicator bytes, as follows:

Memory Address (MADDR): the last memory location accessed via the MEM command (or NEXT, following MEM).

Break Point Address (BKADDR): the location in the breakpoint table of the last breakpoint encountered during user program execution or the last breakpoint displayed by monitor command BRK (or NEXT, following BRK).

Register Name (RGADDR): the name of the last register displayed by REG command, or zero if MEM command has been used since the last REG command.

Step Flag (SFLAG): a control byte that determines the monitor's actions at entry.

When the monitor is awaiting a command or data, register pair H,L generally contains a display address, which points to either the memory address, the user's program counter, a breakpoint, or an address just keyed in by the user following the ADDR command.

Operation of the monitor commands can be described in large part by reference to these addresses:

PCADDR

MADDR

BKADDR

RGADDR

SFLAG

and the display address in H,L.

3.3 Monitor Commands

Monitor commands are issued by pressing one of the eight command keys: ADDR, MEM, NEXT, CLR, REG, STEP, BRK, RUN.
These are discussed below in the order listed.

3.3.1 ADDR

Recalls the user's program counter and makes it the display address. The PC is displayed in the left hand four digits, and the content of memory at that address in the right hand two digits.

If ADDR is followed by hexadecimal keys, the display address is cleared and the hex characters are entered as the display address. In general four characters must be entered, but this depends on the command which follows ADDR. A count of the number of keys is complemented and stored in register D for use by the monitor in executing the next command.

Contents of D:

00	ADDR not used
FF	ADDR used, no hex keys
FE	one hex key
FD	two hex keys

FC	three hex keys
FB	four hex keys

The address, either the user's program counter or the keyed address, is passed to another command section when a command key is pressed. See the sections describing the commands MEM, BRK, STEP and RUN for details of the effects.

3.3.2 MEM

Calls for display of a memory address and its contents. If the preceeding command was not ADDR, the previously stored memory address is used. If ADDR was used, the address in H,L becomes the memory address. This may be the user's program counter or a newly keyed address. If exactly one hex key followed ADDR, that is taken as the name of a register pair, the stack pointer, or the stack top, and the two bytes referred to thereby become the memory address.

Key	Register Pair
1/SP	Stack pointer
2/ST	Stack Top
8/H	H,L
B	B,C
D	D,E

Other single key entries are errors.

With a memory address determined it is displayed in the four left hand digits and the contents of that location are displayed in the

right hand two digits. If the address was derived from a register pair, a label identifying that pair is displayed.

After the MEM command has been issued, the contents of the displayed location can be altered by keying in one or two (or more) hex digits.

The NEXT command increments the memory address and displays the new address and contents. Again, the contents can be altered.

Note that ADDR causes display of a memory address, but the contents cannot be altered until the MEM command has been given. Examples:

MEM	8300 AF
4 4	8300 44

Recalls and displays previous memory address and contents. Contents can be altered by hex keys.

ADDR	8200 01

Recalls and displays user's PC and instruction. Contents cannot be altered.

ADDR	MEM	8200	01

Now contents can be altered. 8200 is now the stored memory address.

NEXT	8201	80

Displays the next byte in memory. 8201 is the stored memory address. Contents can be altered.

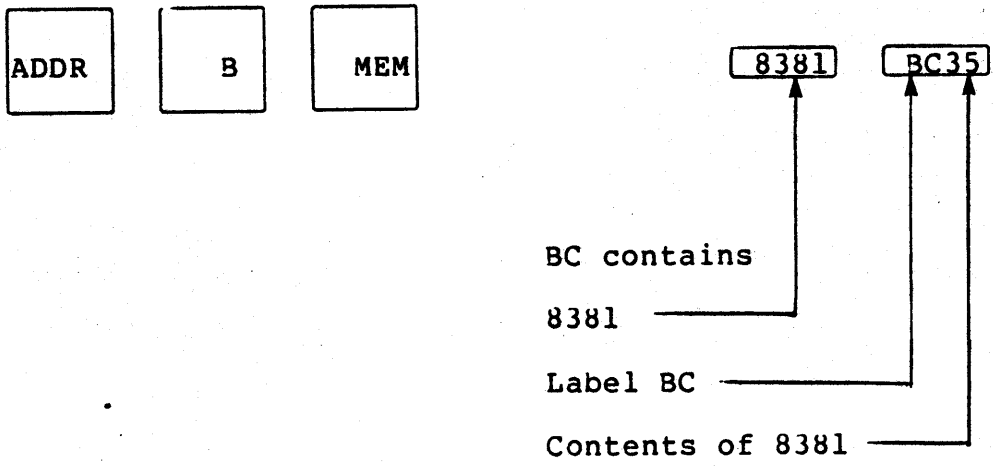
ADDR	8	3	8	0	8380	44

Displays 8380 again, but contents are protected until:

MEM	8380	44

Now 8380 is the stored memory address and its contents can be altered.

Register pair display



3.3.3 NEXT

This increments the memory address if a memory location is being displayed.

When a register is displayed NEXT selects the next register in sequence: A, B, C, D, E, F, H, L.

When a breakpoint is displayed NEXT calls for display of the next breakpoint in the list. If there is only one breakpoint in the table, NEXT has no effect.

3.3.4 CLR

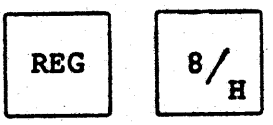
CLR removes hexadecimal data keyed in after the last command key. If an address is being entered, the program counter again becomes the displayed address. If data are being entered to a register or memory address, the previous contents are restored.

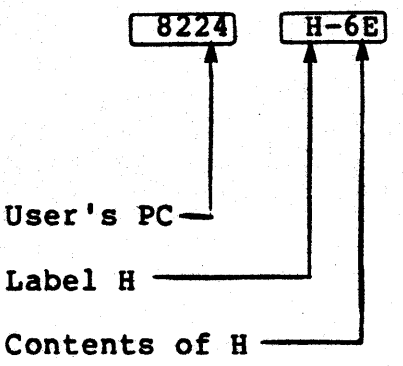
In the breakpoint system, CLR deletes the displayed breakpoint from the list.

3.3.5 REG

REG is followed by a hex key naming the register desired.

[REG] [n] Displays the current contents of the user's program counter and the contents of register n, with a label.





If followed by any hexadecimal key or keys the contents of the displayed register are altered.

REG 8/H 3 2 8224 H-32

If followed by NEXT, the next register (alphabetically) is displayed.

NEXT 8224 L-13

The name of the register selected for display is retained, and at subsequent entry to the monitor the selected register will be displayed. When the MEM key is used, the register name is cleared. Further entries to the monitor will display the contents of the current address. A register name is stored (as one byte at RGADDR) when a register is selected by REG n or by NEXT while a register is being displayed.

If REG follows an ADDR command the effect of the ADDR command is lost. REG always shows the program counter in the left hand four digits.

3.3.6 STEP

STEP sets SFLAG = 1 to indicate that the monitor keyboard and display functions are to be activated at the next entry to the monitor. All user registers are restored, the interrupt system is enabled, and control is returned to the user's program at the location stored in PCADDR. The user's program is interrupted upon execution of the next instruction and the monitor is reactivated.

If the STEP (or RUN) command immediately follows an ADDR command with four (or more) hexadecimal keys, then the address entered becomes the user's program counter, and control is passed to that location.

3.3.7 RUN

RUN sets SFLAG = 0 to indicate that the RUN command was issued and then returns to the user's program exactly as in STEP. The user's program is interrupted at each instruction to test for breakpoints, but the keyboard and display are not activated unless a breakpoint is encountered and its count reaches zero. When this occurs the monitor behaves as though a STEP had been used.

3.3.8 Breakpoints

BRK Displays the address of the current breakpoint, which is the last breakpoint encountered. In the usual case it is equal to the program counter, unless the step key was used or a programmed entry to the monitor was made. If the program has not yet encountered any breakpoint, then it will be the last breakpoint displayed. Along with the address, a label (BP) and the count for that breakpoint are displayed.

If no breakpoint has been entered the display will show:

0000	BP.00

A breakpoint is entered by:

ADDR	8	2	1	0	8210	10
BRK					8210	BP00
4	(optional count)				8210	BP04

When RUN is pressed, this address will be encountered and executed four times, stopping on the fifth. Then the display shows the program counter and instruction:

8210	10

BRK 8210 BP00

reak shows the breakpoint, now counted down to zero.

. new count may now be keyed in:

2 4 8210 BP24

r the breakpoint may be removed:

CLR 0000 BP00

The display of address 0000 shows that no breakpoints exist. If other breakpoints are still stored, the most recently used or displayed would now be displayed.

If more than one breakpoint is stored (and eight are permitted) NEXT will display each in turn. Whenever a breakpoint is displayed it may have a new count entered or it may be cleared. RST clears all break points.

4. MONITOR SUBROUTINES AND DISPLAY

4.1 Display

Data stored in locations 83F8 - 83FF are displayed by the DMA channel. This is normally enabled by the monitor; it can be controlled as follows:

```
3E      MVI   A,80        Set high bit = 1
8D                        to enable display.
D3      OUT PORTC
FA
```

or

```
AF      XRA   A           Set high bit = 0
D3      OUT PORTC         to kill display
FA
```

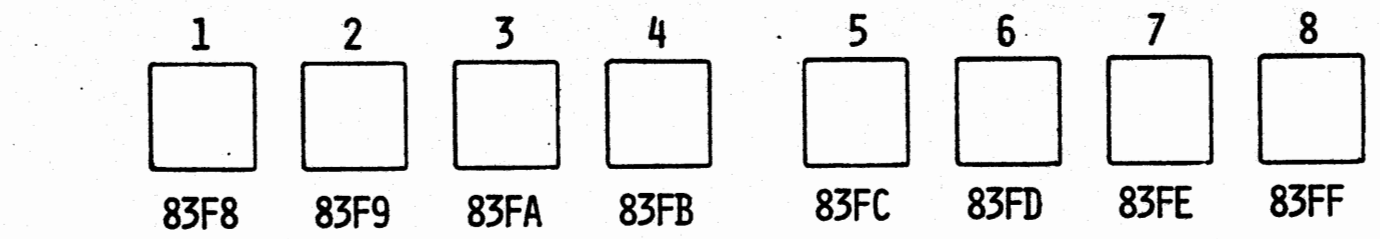
The display is refreshed at approximately one millisecond intervals by DMA. The contents of 83F8 drive the leftmost digit, 83FF the rightmost digit. Each bit controls a segment; the high bit is the decimal point (see Figure A-3). The RAM Memory Map is shown in Figure A-4.

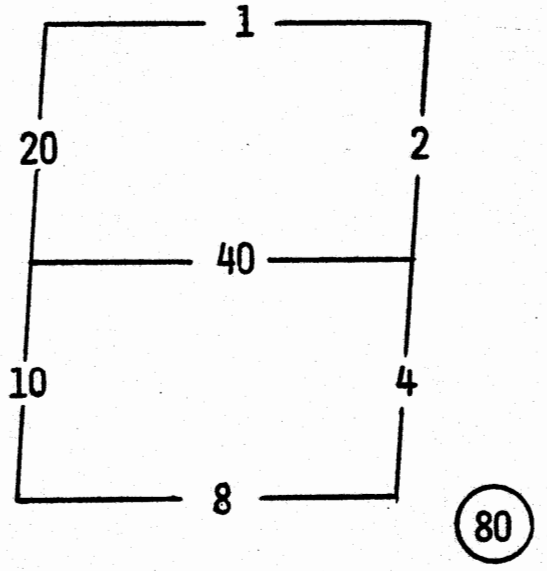
4.2 Display Subroutines

The following subroutines are available to the user.

SPLIT (ADDRESS 02C2)

Enter with a byte in register A. Return with the original value in C; the high order digit in register B (shifted to the right); and the low order digit in register A. This is used by DBYTE.

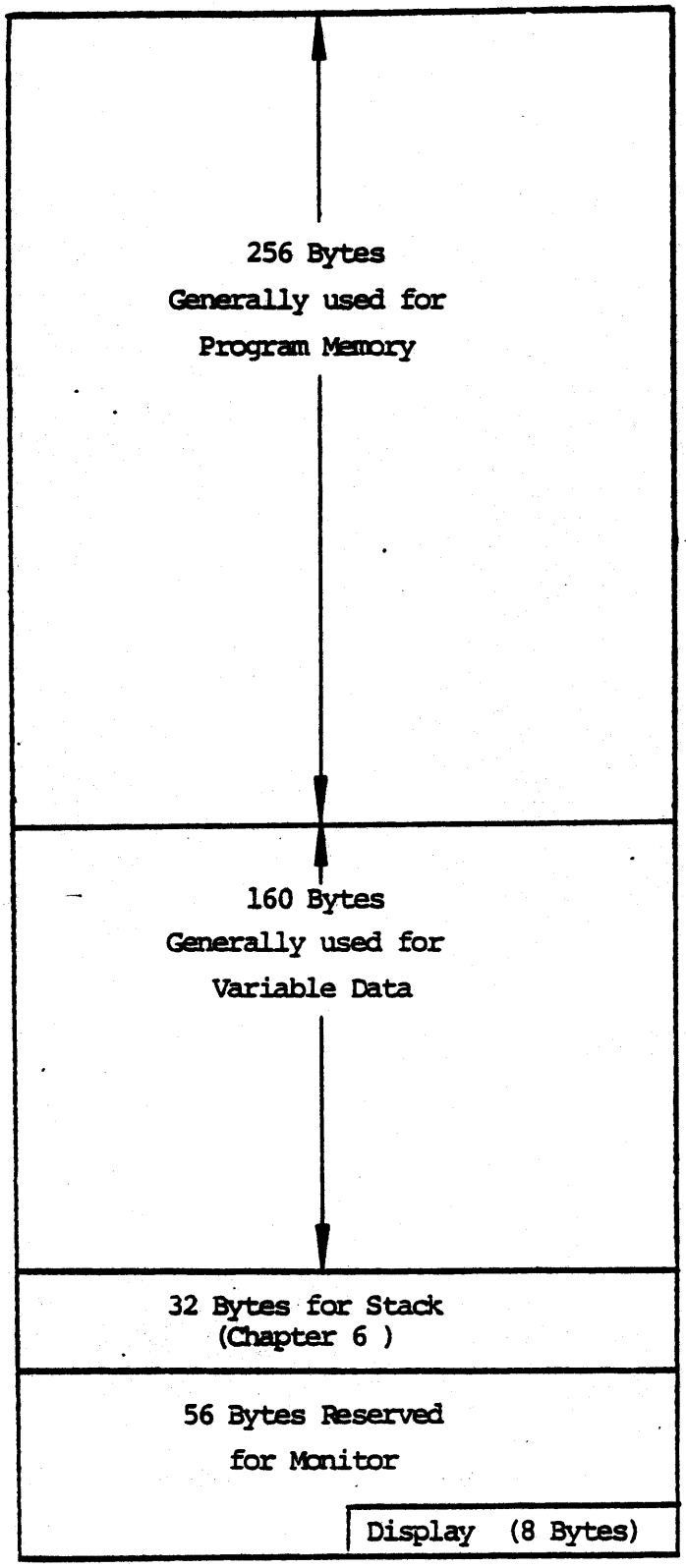




HEXADECIMAL CODES FOR LED SEGMENTS

FIGURE A3

8200 - 820F
8210 - 821F
8220 - 822F
8230 - 823F
8240 - 824F
8250 - 826F
8260 - 826F
8270 - 827F
8280 - 828F
8290 - 829F
82A0 - 82AF
82B0 - 82BF
82C0 - 82CF
82D0 - 82DF
82E0 - 82EF
82F0 - 82FF
8300 - 830F
8310 - 831F
8320 - 832F
8330 - 833F
8340 - 834F
8350 - 835F
8360 - 836F
8370 - 837F
8380 - 838F
8390 - 839F
83A0 - 83AF
83B0 - 83BF
83C0 - 83CF
83D0 - 83DF
83E0 - 83EF
83F0 - 83FF



MEMORY MAP
READ WRITE MEMORY

FIGURE A4

OFFSET (Address 02A9)

Enter with a digit in A and a display location in D,E. Generates the seven segment equivalent and stores in the display location. Decrements D,E to point to the next higher digit.

DMEM (Address 0294)

DBYTE (Address 0295)

DBY2 (Address 0298)

These are three alternate entries to the same subroutine, which calls SPLIT once and OFFSET twice, to display a byte as two digits.

Enter DMEM with a memory address in H,L; its contents will be displayed at the right.

Enter DBYTE with a byte in A; it will be displayed at the right.

Enter DBY2 with a byte in A and one of the digit display addresses in D,E. The byte in A will be displayed at the digit addressed and the next leftward digit.

DWORD (Address 02D1)

DWD2 (Address 02D4)

These two entries call DBY2 twice to display the contents of H and L as four digits. DWORD displays at the left; DWD2 permits entry

with a location in D,E.

4.3 Monitor Input Subroutines

The monitor has four useful subroutines for keyboard data entry, which your program can call. They are:

SCAN (Address 0257)

This scans the keyboard once. If no key is pressed it returns with carry cleared. If a key is pressed it returns with carry set and the value of the key in the A register. It uses register B; all other registers are preserved. This subroutine is also called by GETKY.

GETKY (Address 023D)

This calls SCAN repeatedly until a key is pressed, and then waits until the key has been released for long enough to ensure that contact bounce will not make the key appear to have been pressed twice. It returns with the key value duplicated in registers A and C; and the carry set for hex keys, cleared for command keys.

SCAN and GETKY return the hex value for hex keys and the following for command keys:

MEM	10
REG	11
ADDR	12
STEP	13
RUN	14

NEXT 15

BRK 16

CLR 17

The following routines also display the data entered:

ENTBY (Address 0336)

ENTWD (Address 0346)

Each of these calls GETKY to obtain one or more keys. As they are entered, hex keys are shifted into registers H and L and counted in register D. They return to the calling program when a command key is pressed: that key is duplicated in registers A, B and C; register D contains a count of the number of hex keys entered; H and L contain the last four digits entered.

The two subroutines are identical in accepting key input, but they also display the input, and here they differ. ENTBY displays only the last two digits, in the two right hand digit positions. ENTWD displays the last four digits in the leftmost positions.

N.B. These subroutines (except for SCAN) involve delays because of the debouncing requirement, and run very slowly in debug mode.

4.4 Subroutine Specifications and Listings

The above subroutines are fully specified in Chapter 6.10, and listings appear in Appendix B.

MICROCOMPUTER TRAINING WORKBOOK

APPENDIX B

THE ICS MONITOR PROGRAM LISTING

```
                        TITLE  MTS  MONITOR PROGRAM  WRITTEN 1/77 BY EDWARD LAPPI
                ;
0000                    ORG 0
                ;
                ;       RESET: RESET ENTRY TO MONITOR
                ;
0000 31D383     RESET:  LXI SP,TOS      ;LOAD NEW SP
0003 E5                 PUSH H          ;SAVE HL
0004 C5                 PUSH B          ;SAVE OTHER REGISTERS
0005 D5                 PUSH D
0006 F5                 PUSH PSW
0007 21DE83             LXI H,BKTBL     ;LOAD BKPT TABLE POINTER
000A 22DC83             SHLD BKADDR
000D 061A               MVI B,26        ;CLEAR OUT OLD BREAKPOINTS
000F AF                 XRA A
0010 32D583             STA RGADDR      ;MAKE DISPLAY MODE INSTRUCTION
0013 77         RLP:    MOV M,A         ;CLEAR  24 PLACES FOR BKPTS
0014 05                 DCR B
0015 23                 INX H
0016 C21300             JNZ RLP
0019 210082             LXI H,8200H     ;LOAD PC AND M POINTER
001C C36B00             JMP BYPASS      ;GO AROUND RST ENTRY
                ;
0020                    ORG 20H         ;RST 4 LOCATION
0020 F3         RST4:   DI              ;ENTRY INTO MONITOR VIA SOFTWARE
0021 E3                 XTHL            ;SAVE PC, REGS
0022 C5                 PUSH B
0023 D5                 PUSH D
0024 F5                 PUSH PSW
0025 C36E00             JMP SAVEP
                ;
0028                    ORG 28H         ;RST 5 LOCATION
0028 C32882     RST5:   JMP 8228H
                ;
0030                    ORG 30H         ;RST 6 LOCATION
0030 C33082     RST6:   JMP 8230H
                ;
0038                    ORG 38H         ;RST 7 START LOCATION
0038 F3         RST7:   DI              ;DISABLE INTERRUPT
0039 E3                 XTHL            ;SWITCH OLD PC WITH HL
003A 22DA83             SHLD PCADDR     ;SAVE PC
003D C5                 PUSH B          ;SAVE REGS
003E D5                 PUSH D
003F F5                 PUSH PSW
0040 3AD483             LDA SFLAG       ;SEE IF SINGLE STEP MODE
0043 A7                 ANA A
0044 CA5300             JZ  BKTST       ;IF 0 LAST COMMAND WAS RUN
0047 FE02               CPI 2           ;  SO TEST FOR BREAK POINT
0049 DA7100             JC  NORUN       ;IF 1 LAST COMMAND WAS STEP
004C CA6200             JZ  KYTST       ;IF 2 USER HAS REQUESTED INT ON KEY
```

```
004F 6F                 MOV L,A         ;IF GREATER JUMP INTO USER PROGRAM
0050 2682               MVI H,82H       ;AT 82XX WITH SFLAG FOR XX
0052 E9                 PCHL
                ;
0053 CDDD02     BKTST:  CALL BKLOC      ;SEE IF BKPT SET
0056 D25D01             JNC RERUN       ;IF NONE FOUND, RUN
0059 B7                 ORA A
005A CA7100             JZ NORUN        ;IF FOUND, SEE IF COUNT=0
005D 3D                 DCR A           ;IF NOT, DCR COUNT AND SAVE
005E 12                 STAX D
005F C35D01             JMP RERUN       ;RUN AGAIN
                ;
0062 CD5702     KYTST:  CALL SCAN       ;TEST FOR ANY KEY PRESSED
0065 D25D01             JNC RERUN       ;IF SFLAG SET=2 BY USER,
0068 C37100             JMP NORUN       ;NO RUN IF ANY KEY DEPRESSED
                ;
006B 22D883     BYPASS: SHLD MADDR      ;ENTRY FROM RESET
006E 22DA83     SAVEP:  SHLD PCADDR     ;SAVE PC
0071 3E92       NORUN:  MVI A,92H       ;INITIALIZE PIA
0073 D3FB               OUT CNTPT
0075 210000             LXI H,0         ;GET SP LOCATION
0078 39                 DAD SP
0079 22D683             SHLD SPADDR
007C CD8202             CALL CLRGT      ;CLEAR RIGHT SIDE OF DISP
007F CDCE02             CALL DYPC       ;DISPLAY CURRENT PC
0082 3AD583             LDA RGADDR      ;IS A REG TO BE DISPLAYED ?
0085 B7                 ORA A
0086 C29301             JNZ DREG2
0089 CD9402             CALL DMEM       ;IF NOT DISPLAY INSTRUCTION AND
008C C39B00             JMP CMD         ;WAIT FOR COMMAND
                ;
                ;ERR: ERROR OUTPUT SECTION
                ;     PUTS ERR ON DISPLAY
                ;
008F CD8702     ERR:    CALL CLEAR      ;CLEAR DISPLAY
0092 23                 INX H           ;ADDRESS HIGH DIGIT
0093 3679               MVI M,ECODE     ;LOAD CODE FOR E
0095 23                 INX H           ;GO TO NEXT DIG
0096 3650               MVI M,RCODE     ;LOAD CODE FOR R
0098 23                 INX H
0099 3650               MVI M,RCODE
                ;
                ;CMD: COMMAND ENTRY.  DECODES COMMAND KEY AND ROUTES
                ;     ACCORDINGLY
                ;
009B CD3D02     CMD:    CALL GETKY      ;GET A KEY
009E DA8F00             JC ERR          ;IF NOT COMMAND, ERROR
00A1 2ADA83             LHLD PCADDR     ;RESTORE PC TO HL
00A4 1600       CMD2:   MVI D,0         ;CLEAR FLAG FOR ADDR DATA
00A6 87         CMD3:   ADD A           ;DOUBLE VAL OF COMMAND KEY
```

```
00A7 E5             PUSH H          ;SAVE HL
00A8 CD8202         CALL CLRGT      ;CLEAR RIGHT SIDE OF DISPLAY
00AB 019B00         LXI B,PTR-22    ;LOAD START OF TABLE
00AE 81             ADD C
00AF 4F             MOV C,A         ;ADD OFFSET
00B0 0A             LDAX B
00B1 6F             MOV L,A         ;GET ADDRESS FROM TABLE
00B2 03             INX B
00B3 0A             LDAX B
00B4 67             MOV H,A
00B5 01D583         LXI B,PGADDR    ;READY FOR SECTION
00B8 AF             XRA A
00B9 E3             XTHL            ;SWITCH WITH HL
00BA C9             RET             ;GO TO SECTION
                    ;
00BB D000   PTR:    DW MEMCH        ;IMP- MUST NOT CROSS PAGE BOUNDS
00BD 8201           DW REGC         ;FROM START OF CMD TO END OF PTR
00BF 6E01           DW GOTOC
00C1 4401           DW STEPP
00C3 4501           DW RUNP
00C5 C290           DW NEXTC
00C7 B701           DW BRKPT
00C9 9B00           DW CM           ;CLEAR GOES TO ZERO
                    ;
00CB 02     NEXTC:  STAX B          ;CLEAR REGISTER
00CC 23             INX H           ;GO TO NEXT MEM LOCATION
00CD C31001         JMP SAVEHL      ;JUMP INTO MEMCH
                    ;
                    ;MEMCH: ALLOWS CHANGING OF MEMORY
                    ;
00D0 02     MEMCH:  STAX B          ;SET DISPLAY MODE FOR INSTRUCTION
00D1 82             ADD D           ;GET DIG COUNT
00D2 CA1301         JZ OLDAD        ;IF NOT ADRS, USE OLD MEM REF
00D5 FEFE           CPI 0FEH        ;IF NO KEYS OR MORE THAN ONE
00D7 C21001         JNZ SAVEHL      ; HL CONTAINS DESIRED ADDRESS
00DA 7D             MOV A,L         ;ELSE GET DOUBLE REGISTER
00DB 210501         LXI H,DRTBL     ;GET TABLE START
00DE 0E05           MVI C,5         ;# OF DR
00E0 BE     GLP:    CMP M           ;SEE IF RIGHT DR
00E1 23             INX H
00E2 CAEE00         JZ FDR          ;IF RIGHT, LEAVE LOOP
00E5 23             INX H           ;ELSE SKIP SYMBOL
00E6 23             INX H
00E7 0D             DCR C           ;SUBTRACT LOOP COUNT
00E8 C2E000         JNZ GLP         ;IF NOT ZERO, GO BACK
00EB C38F00         JMP ERR         ;ELSE ERROR MADE
00EE 7E     FDR:    MOV A,M         ;GET SYMBOL
00EF 23             INX H           ;AND DISPLAY
00F0 66             MOV H,M
00F1 6F             MOV L,A
```

```
00F2 22FC83             SHLD LOWDGT-3
00F5 2AD683             LHLD SPADDR     ;GET SP
00F8 79                 MOV A,C         ;GET COUNT VALUE
00F9 87                 ADD A           ;DOUBLE
00FA FE09               CPI 9           ;SEE IF TOO HIGH
00FC DA0101             JC VOK          ;SKIP IF OK
00FF 3E08               MVI A,8         ;ELSE MAKE =8
0101 23         VOK:    INX H           ;ADD TO SP
0102 3D                 DCR A           ;VALUE OF OFFSET BY LOOPING UNTIL
0103 C20101             JNZ VOK         ;A=0
0106 79                 MOV A,C         ;SEE IF SP
0107 FE05               CPI 5
0109 CA1001             JZ SAVEHL       ;IF SO, DO NOT MEM FETCH
010C 5E                 MOV E,M         ;GET DE FROM MEMORY
010D 23                 INX H
010E 56                 MOV D,M
010F EB         SAVEMA: XCHG            ;PUT MEM ADDRESS IN HL
0110 22D883     SAVEHL: SHLD MADDR      ;MAKE NEW MEM ADDRESS
0113 2AD883     OLDAD:  LHLD MADDR      ;GET ADDRESS
0116 CDD102             CALL DWORD      ;DISPLAY
0119 CDD302             CALL DREG       ;DISPLAY DATA
011C CD3503             CALL ENTBY      ;ENTER NEW DATA BYTE
011F FE17               CPI CLE         ;SEE IF CLEAR USED
0121 CA1301             JZ OLDAD        ;IF USED, CLEAR AND DO AGAIN
0124 15                 DCR D           ;SEE IF COUNT =0
0125 7D                 MOV A,L         ;GET DATA
0126 2AD883             LHLD MADDR      ;LOAD HL WITH ADDRESS TO BE CHNG
0129 FA3101             JM NOUPD        ;DO NOT UPDATE IF ZERO
012C 77                 MOV M,A         ;SAVE
012D BE                 CMP M           ;SEE IF EQUAL
012E C28F00             JNZ ERR         ;IF NOT, ERROR
0131 78         NOUPD   MOV A,B         ;GET COMMAND
0132 C3A400             JMP CMDJ
                ;
0135 01         DRTBL   DB 01H          ;TABLE FOR DR DISPLAY AND FETCH
0136 6DF3               DB 6DH,0F3H
0138 026DB1             DB 02H,6DH,0B1H
013B 08766B             DB 08H,76H,6BH
013E 085D89             DB 08H,5DH,89H
0141 085EF9             DB 08H,5EH,0F9H
                ;
                ;STEP & RUN ROUTINES
                ;
0144 3C         STEPP:  INR A           ;SET STEP FLAG TO 1
0145 32D483     RUNP    STA SFLAG       ;SAVE 0 OR 1 AS SFLAG
0148 7A                 MOV A,D         ;GET ENTRY
0149 B7                 ORA A           ;SEE IF ZERO
014A CA5A01             JZ CLFST        ;IF SO, USE OLD PC
014D 2F                 CMA             ;REVERSE COUNT
014E B7                 ORA A           ;IF ZERO, OK
```

```
014F CA5701             JZ ADROK
0152 FE04               CPI 4               ;SEE IF LESS THAN 4
0154 DA8F00             JC ERR              ;IF SO, ERROR
0157 22DA83   ADROK:    SHLD PCADDR         ;SAVE PC
015A CD8702   CFST:     CALL CLEAR          ;CLEAR DISPLAY FOR USER
015D F1       RERUN:    POP PSW             ;RESTORE STACK
015E D1                 POP D
015F C1                 POP B
0160 2ADA83             LHLD PCADDR
0163 E3                 XTHL                ;PUT PC ON STACK- GET HL
0164 FB                 EI                  ;ALLOW INTERRUPTS
0165 C9                 RET                 ;START USERS PROGRAM
                ;
                ;GOTOC: ADDRESS SETTING COMMAND
                ;
0166 CDCE02   GOTOC:    CALL DYPC           ;GET PC AND DISPLAY
0169 CD9402             CALL DMEM           ;DISPLAY INSTRUCTION
016C CD4603             CALL ENTWD          ;ENTER ADDRESS
016F FE17               CPI CLE             ;SEE IF CLEAR
0171 CA6601             JZ GOTOC            ;IF CLEAR, GO AGAIN
0174 7A                 MOV A,D             ;SEE IF COUNT=0
0175 B7                 ORA A
0176 C27C01             JNZ NOTPC           ;IF ENTRY MADE SKIP
0179 2ADA83             LHLD PCADDR         ;LOAD PC
017C 2F       NOTPC:    CMA                 ;COMPLIMENT COUNT
017D 57                 MOV D,A             ;SAVE AND PASS TO CMD
017E 78                 MOV A,B             ;PUT COMMAND KEY IN A
017F C3A600             JMP CMD3            ;GO TO COMMAND ROUTER
                ;
                ;REGC: REGISTER MODIFY SEGMENT
                ;
0182 CDCE02   REGC:     CALL DYPC           ;GET AND DISPLAY PC
0185 CD3D02             CALL GETKY          ;GET A KEY
0188 D2A400             JNC CMD2            ;IF COMMAND, PROCESS
018B FE08               CPI 8               ;SEE IF LESS THAN 8
018D DA8F00             JC ERR              ;IF LESS THAN 8, NOT LEGAL
0190 32D583   DRSAV:    STA RGADDR          ;SAVE NEW DISP MODE
0193 CD0303   DREG2:    CALL DREG           ;DISPLAY REGISTER
0196 E5                 PUSH H              ;SAVE ADDRESS
0197 CD3603             CALL ENTBY          ;GET NEW DATA
019A 4D                 MOV C,L             ;GET DATA SAVED
019B E1                 POP H               ;RESTORE STACK
019C FE17               CPI CLE             ;SEE IF CLEAR KEY
019E CA9301             JZ DREG2            ;IF SO, REDISPLAY
01A1 7A                 MOV A,D             ;GET COUNT
01A2 B7                 ORA A
01A3 CAA701             JZ RNE              ;NO CHANGE IF COUNT =0
01A6 71                 MOV M,C
01A7 78       RNE:      MOV A,B             ;GET COMMAND
01A8 FE15               CPI NEXT            ;SEE IF NEXT
```

```
01AA C2A400             JNZ CMD2
01AD 3AD583             LDA RGADDR
01B0 C6F9               ADI 0F9H        ;INCREMENT RGADDR
01B2 F608               ORI 08H         ;CHANGE 16 TO 8
01B4 C39001             JMP DRSAV
                ;
                ;BRKPT: BREAKPOINT MANAGEMENT SEGMENT
                ;
01B7 E5         BRKPT:  PUSH H          ;DISPLAY BP TO INDICATE BKPT
01B8 215DF3             LXI H,BPCODE
01BB 22FC83             SHLD LOWDGT-3
01BE E1                 POP H
01BF 7A                 MOV A,D         ;SEE IF ADDR ENTERED
01C0 B7                 ORA A
01C1 CA0802             JZ DBKPT        ;IF NOT, DISPLAY CURRENT BP
01C4 2F                 CMA
01C5 B7                 ORA A           ;SEE IF GOTO ENTRY MADE
01C6 CACE01             JZ OKVAL        ;IF 0 OR MORE THAN 4, OK
01C9 FE04               CPI 4
01CB DA8F00             JC ERR          ;ELSE ERROR
01CE CDDD02     OKVAL:  CALL BKLOC      ;LOCATE BREAKPOINT
01D1 DA1602             JC OLDBP        ;IF FOUND, USE OLD
01D4 05                 DCR B           ;SEE IF ROOM FOR NEW
01D5 FA8F00             JM ERR          ;IF NOT, ERROR
01D8 1B                 DCX D           ;LOAD IF ROOM
01D9 1B                 DCX D
01DA EB                 XCHG            ;LOAD NEW BKADDR
01DB 22DC83             SHLD BKADDR
01DE 73                 MOV M,E         ;WRITE NEW ADDRESS INTO TABLE
01DF 23                 INX H
01E0 72                 MOV M,D
01E1 CD3603     BKLOP:  CALL ENTBY      ;GET BYTE
01E4 FE17               CPI CLE         ;SEE IF CLEAR
01E6 CA1C02             JZ CLRBK        ;IF CLEAR, CLEAR BKPT
01E9 7D                 MOV A,L         ;GET ENTERED DATA
01EA 2ADC83             LHLD BKADDR     ;GET HL POSITION
01ED 23                 INX H           ;INC TO COUNT
01EE 23                 INX H
01EF 15                 DCR D           ;SEE IF COUNT=0
01F0 FAF401             JM NOBKC        ;IF ZERO, DO NOT CHANGE
01F3 77                 MOV M,A         ;SAVE NEW COUNT
01F4 23         NOBKC:  INX H           ;GO TO NEXT BKPT
01F5 7E                 MOV A,M         ;GET BKPT
01F6 23                 INX H           ;GOTO NEXT ONE UP
01F7 B6                 ORA M           ;SEE IF ZERO TOO
01F8 2B                 DCX H           ;CORRECT POINTER
01F9 C2FF01             JNZ BOK         ;IF NOT, OK
01FC 21DE83             LXI H,BKTBL     ;LOAD START OF TABLE AGAIN
01FF 78         BOK:    MOV A,B         ;GET COMMAND
0200 FE15               CPI NEXT        ;SEE IF NEXT
```

```
0202 C2A400            JNZ CMD2          ;IF NOT, PROCESS COMMAND
0205 22DC83            SHLD BKADDR       ;SAVE NEW BKPT ADDR
0208 2ADC83   DBKPT:   LHLD BKADDR       ;GET BKPT ADDR
020B 5E                MOV E,M           ;GET ENTRY OUT OF TABLE
020C 23                INX H
020D 56                MOV D,M
020E E5                PUSH H            ;SAVE POSITION
020F EB                XCHG
0210 CDD102            CALL DWORD        ;DISPLAY ADDR
0213 E1                POP H             ;GET POSITION BACK
0214 23                INX H             ;GET COUNT
0215 7E                MOV A,M
0216 CD9502   OLDBP:   CALL DBYTE        ;DISPLAY
0219 C3E101            JMP BKLOP         ;GO BACK
021C 2ADC83   CLRBK:   LHLD BKADDR       ;GET ADDR
021F 54                MOV D,H           ;GET ADDR+3
0220 5D                MOV E,L
0221 13                INX D
0222 13                INX D
0223 13                INX D
0224 1A       CLP:     LDAX D            ;LOOP UNTIL END OF TABLE REACHED
0225 77                MOV M,A           ;TRANSFER DATA
0226 23                INX H
0227 13                INX D             ;CHANGE POINTERS
0228 7B                MOV A,E           ;SEE IF AT END
0229 FEF8              CPI (BKTBL+26) AND 0FFH
022B C22402            JNZ CLP           ;IF NOT, CONTINUE
022E 2ADC83            LHLD BKADDR       ;GET BKPT ADDR
0231 0615              MVI B,NEXT        ;MAKE KEY LOOK LIKE NEXT
0233 C3F501            JMP NOBKC+1       ;GO BACK TO SEE IF AT END OF TABLE
                ;
                ;
                ;       DELAY: SUBROUTINE FOR 1 MS DELAY IF NO
                ;                HALTS OR HOLDS
                ;USES REG A AND FLAGS
                ;
0236 3E83       DELAY: MVI A,131
0238 3D         DEL1:  DCR A
0239 C23802            JNZ DEL1
023C C9                RET
                ;       GETKEY: GETS AND DEBOUNCES A KEY FROM THE
                ;                KEYBOARD SCANNER
                ;USES REGISTERS B AND D
                ;REG A AND C CONTAIN THE KEY VALUE
                ;CALLS DELAY AND SCAN
                ;RETURNS WITH CY=0 FOR COMMAND; CY=1 FOR HEX
023D CD5702    GETKY:  CALL SCAN        ;GET A KEY
0240 D23D02            JNC GETKY         ;IF NO KEY FOUND, TRY AGAIN
0243 4F                MOV C,A
0244 1614      RSTDY:  MVI D,20         ;DEBOUNCE CYCLES
```

```
0246 CD3602   DLOOP:  CALL DELAY      ;DELAY
0249 CD5702           CALL SCAN       ;GET H KEY
024C DA4402           JC RSTDY        ;IF ONE, START OVER
024F 15               DCR D
0250 C24602           JNZ DLOOP       ;IF CYCLE DONE, END
0253 79               MOV A,C         ;PUT KEY IN A
0254 FE10             CPI 16          ;COMPARE FOR COMMAND KEY
0256 C9               RET             ; CARRY SET IF HEX
                ;
                ;       SCAN: KEYBOARD SCANNER
                ;             GETS A KEY BY SCANNING THE KEYBOARD
                ;USES REG B
                ;KEY VALUE RETURNED IN REG A
                ;USES THE STACK FOR 1 LEVEL
                ;RETURNS A SET CARRY IF KEY PRESSED
                ;
0257 E5       SCAN:   PUSH H          ;SAVE HL
0258 2EEE             MVI L,11101110B ;SCAN MASK-0 MEANS THAT ROW
025A 0600             MVI B,0         ;COUNT=0
025C 7D       INPUT:  MOV A,L
025D F601             ORI 1           ;MAKE SURE TRANSMISSION BIT SET
025F D3FA             OUT PORTC       ;OUTPUT SCAN DATA
0261 DBF8             IN PORTA        ;READ IN COLUMNS
0263 2F               CMA             ;INVERT
0264 A7               ANA A
0265 C27302           JNZ KP          ;IF KEY PRESSED, NOT ZERO
0268 04               INR B           ;INC COUNTER
0269 78               MOV A,B
026A FE03             CPI 3           ;SEE IF DONE
026C D28002           JNC SCRET       ;IF DONE, RETURN
026F 29       KSCAN:  DAD H           ;SHIFT MASK ONE OVER
0270 C35C02           JMP INPUT
0273 2EFF     KP:     MVI L,0FFH      ;LOAD L WITH -1
0275 2C       FK:     INR L           ;INC L UNTIL 1 FOUND IN INPUT
0276 0F               RRC             ;BYTE
0277 D27502           JNC FK
027A 78               MOV A,B         ;TAKE ROW COUNT AND MULT BY 8
027B 87               ADD A
027C 87               ADD A
027D 87               ADD A
027E 85               ADD L           ;ADD COLUMN COUNT IN L
027F 37               STC             ;SET CARRY
0280 E1       SCRET:  POP H           ;RESTORE STACK
0281 C9               RET
                ;
                ;       CLRGT: CLEARS RIGHT SIDE OF DISPLAY
                ;       CLEAR: CLEARS ALL OF DISPLAY
                ;       CLRLP: CLEARS B DIGITS STARTING AT LOW DGT IN
                ;              HL REGISTERS
                ;USES B,H,L
```

```
                        ;
0282 0604       CLRGT:  MVI B,4         ;4 DIGIT BLANK
0284 C38902             JMP CL
0287 0608       CLEAR:  MVI B,8         ;8 DIGIT BLANK
0289 21FF83     CL:     LXI H,LOWDGT    ;LOW DIGIT OF DISP
028C 3600       CLRLP:  MVI M,0         ;CLEAR
028E 2B                 DCX H           ;CHANGE POINTER
028F 05                 DCR B           ;LOOP COUNT DCR
0290 C28C02             JNZ CLRLP
0293 C9                 RET
                        ;
                        ;       DBYTE: DISPLAY BYTE OF DATA IN A ON LOW DGTS
                        ;       DBY2: DISPLAY BYTE ON ANY DIGITS SPEC BY DE
                        ;       DMEM: DISPLAY BYTE REFERENCED BY HL
                        ;USES REG A,B,D,E
                        ;REG C = OLD REG A
                        ;CALLS SPLIT,OFFSET
                        ;USES 1 LEVEL OF STACK
                        ;
0294 7E         DMEM:   MOV A,M         ;GET DISPLAY DATA FROM HL IN MEM
0295 11FF83     DBYTE:  LXI D,LOWDGT    ;LOW 2 DIGITS
0298 E5         DBY2:   PUSH H          ;SAVE HL
0299 CDC202             CALL SPLIT      ;SPLIT BYTE
029C CDA902             CALL OFFSET     ;GET AND DISPLAY DIGIT LEGEND
029F 78                 MOV A,B         ;GET OTHER HALF
02A0 CDA902             CALL OFFSET
02A3 E1                 POP H           ;RESTORE HL
02A4 3E80       DYEN:   MVI A,80H       ;MAKE SURE DISPLAY IS ON
02A6 D3FA               OUT PORTC
02A8 C9                 RET
                        ;
                        ;       OFFSET: GETS A 4 BIT VALUE AND FINDS THE SYMBOL
                        ;               FOR THE DIGIT AND DISPLAYS IT
                        ;USES REG A,H,L
                        ;REG PAIR D,E ARE DECREMENTED BY 1
                        ;
02A9 21B202     OFFSET: LXI H,TABLE     ;TABLE OF DIGITS 0 TO F
02AC 85                 ADD L           ;OFFSET POINTER BY DIGIT IN A
02AD 6F                 MOV L,A
02AE 7E                 MOV A,M         ;GET CODE FOR DIGIT
02AF 12                 STAX D          ;SAVE IN DISP
02B0 1B                 DCX D           ;MOVE TO NEXT DIGIT
02B1 C9                 RET
                        ;
02B2 3F         TABLE:  DB 3FH          ;MUST NOT CROSS PAGE BOUNDS!!!
02B3 065B4F             DB 6,5BH,4FH    ;CODES FOR THE NUMBERS 0 TO F
02B6 666D7D             DB 66H,6DH,7DH
02B9 077F6F77           DB 7,7FH,6FH,77H
02BD 5D395E             DB 5DH,39H,5EH
02C0 7971               DB 79H,71H
```

```
                        ;
                        ;       SPLIT: SEPARATES A BYTE INTO 2 4 BIT VALUES
                        ;REG A=LOW HALF
                        ;REG B=HIGH HALF
                        ;REG C=ORIG VALUE OF A
                        ;
02C2 4F         SPLIT:  MOV C,A         ;SAVE BYTE
02C3 E6F0               ANI 0F0H        ;GET UPPER HALF
02C5 1F                 RAR             ;MOVE TO LEFT SIDE
02C6 1F                 RAR
02C7 1F                 RAR
02C8 1F                 RAR
02C9 47                 MOV B,A         ;SAVE
02CA 79                 MOV A,C         ;GET OTHER HALF
02CB E60F               ANI 0FH
02CD C9                 RET
                        ;
                        ;       DWORD: DISPLAY HL IN LEFT SIDE OF DISPLAY
                        ;       DWD2: DISPLAY HL IN POSITION GIVEN BY DE
                        ;       DYPC: DISPLAY AND GET PC
                        ;USES A,B,C
                        ;REG D,E DCR BY 4 FOR DWD2, NOT USEFUL FOR DWORD
                        ;CALLS DBY2
                        ;
02CE 2ADA83     DYPC:   LHLD PCADDR     ;GET PC FROM MEMORY
02D1 11FB83     DWORD:  LXI D,LOWDGT-4  ;SETUP FOR TOP PART OF DISP
02D4 7D         DWD2:   MOV A,L         ;DISPLAY LOW BYTE
02D5 CD9802             CALL DBY2
02D8 7C                 MOV A,H         ;DISPLAY HIGH BYTE
02D9 CD9802             CALL DBY2
02DC C9                 RET
                        ;
                        ;       BKLOC: LOCATES A BREAKPOINT IN THE TABLE
                        ;               ADDR TO BE FOUND IN HL
                        ;               RETURNS CY=1 IF FOUND--A=COUNT OF BKPT
                        ;               B=0 IF NO ROOM IN TABLE AND NOT FOUND
                        ;               DE=COUNT ADDR OF POS TO BE USED
                        ;
02DD 11DE83     BKLOC:  LXI D,BKTBL     ;LOAD BKPT TABLE POINTER
02E0 0608               MVI B,8         ;MAX # BKPTS
02E2 1A         BL:     LDAX D          ;GET LOW ADDR
02E3 4F                 MOV C,A         ;SAVE
02E4 BD                 CMP L           ;SEE IF EQUAL TO L GIVEN
02E5 13                 INX D           ;GO TO HIGH ADDR
02E6 1A                 LDAX D
02E7 13                 INX D           ;POINT TO COUNT
02E8 C2EF02             JNZ NOMAT       ;SEE IF EQUAL
02EB BC                 CMP H           ;IF =, COMPARE H VALUES
02EC CAF702             JZ MATCH        ;IF =, MATCH
02EF B1         NOMAT:  ORA C           ;SEE IF END OF TABLE
```

```
02F0 C8              RZ              ;END IF HIGH=LOW=0
02F1 05              DCR B           ;SEE IF PHYSICAL END OF TABLE
02F2 C8              RZ
02F3 13              INX D           ;GO TO NEXT BKPT ENTRY
02F4 C3E202          JMP BL
02F7 1B      MATCH:  DCX D           ;IF MATCH, SAVE POINTER
02F8 1B              DCX D
02F9 EB              XCHG
02FA 22DC83          SHLD BKADDR
02FD EB              XCHG
02FE 13              INX D           ;GET COUNT
02FF 13              INX D
0300 1A              LDAX D          ;PUT IN A
0301 37              STC             ;SET CARRY
0302 C9              RET

                     ;
                     ;       DREG. DISPLAYS REGISTER WITH A LEGEND
                     ;               FINDS LOC IN MEM AND PUTS IN HL
                     ;               HL=PC FOR ZERO REG
                     ;               REG VALUE FOUND IN RGADDR
                     ;USES A,B,D,E
                     ;REG C=OLD REG A
                     ;CALLS CLRGT, DBYTE
                     ;
0303 3AD583  DREG:   LDA RGADDR      ;GET REGISTER VALUE
0306 87              ADD A           ;DOUBLE AND TEST FOR ZERO
0307 CA9402          JZ DMEM         ;IF ZERO, DISPLAY INST AT <PC>
030A 211603          LXI H,RGTBL-16  ;LOAD POINTER OF SYMBOLS AND
030D 85              ADD L           ;OFFSET
030E 6F              MOV L,A
030F 7E              MOV A,M         ;GET SYMBOL
0310 32F083          STA LOWDGT-3    ;DISPLAY
0313 3E40            MVI A,DASH      ;GET A DASH SYMBOL
0315 32F083          STA LOWDGT-2    ;DISPLAY
0318 23              INX H           ;GET OFFSET TO SP
0319 7E              MOV A,M
031A 2AD683          LHLD SPADDR     ;GET SP
031D 85              ADD L           ;OFFSET
031E 6F              MOV L,A
031F 7C              MOV A,H
0320 CE00            ACI 0
0322 67              MOV H,A
0323 C39402          JMP DMEM        ;GO DISPLAY DATA
                                     ;RETURN TO CALLER FROM DMEM
                     ;
0326 76      RGTBL:  DB 76H          ;MUST NOT CROSS PAGE BOUNDARIES!!!
0327 07              DB 7
0328 3806            DB 38H,6        ;SYMBOL, OFFSET FROM SPADDR
032A 7701            DB 77H,1        ;ORDER IS H,L,A,B,C,D,E,F
032C 5D05            DB 5DH,5
```

```
032E 3904               DB 39H,4
0330 5E03               DB 5EH,3
0332 7902               DB 79H,2
0334 7100               DB 71H,0
                ;
                ;       ENTBY: ENTER BYTE.  DISPLAYS ENTRY IN FAR RIGHT
                ;              RIGHT POSITION
                ;USES REG B,C,H
                ;REG A AND B BOTH CONTAIN CMD KEY TERMINATING ENTRY
                ;REG D=# OF DIGITS ENTERED
                ;REG L=VALUE ENTERED
                ;CALLS ENT,ENT2,DBYTE
                ;
0336 CD5B03     ENTBY:  CALL ENT        ;INITIALIZE
0339 CD6403     BYLP:   CALL ENT2       ;GET DIGIT AND SHIFT
033C 7D                 MOV A,L         ;DISPLAY BYTE
033D CD9502             CALL DBYTE
0340 D1                 POP D           ;INC D AND RESTORE TO STACK
0341 14                 INR D
0342 D5                 PUSH D
0343 C33903             JMP BYLP
                ;
                ;       ENTWD: ENTER AND DISPLAY WORD OF DATA IN LEFT
                ;              SIDE OF DISPLAY
                ;USES REG B,C
                ;REG A=CMD KEY THAT TERMINATED ENTRY
                ;REG D=# DIGITS ENTERED
                ;REG HL=WORD ENTERED
                ;CALL CLEAR,ENT,ENT2
                ;
0346 CD5B03     ENTWD:  CALL ENT        ;INITIALIZE
0349 CD6403     WDLP:   CALL ENT2       ;GET A DIGIT AND SHIFT IN
034C CDD102             CALL DWORD      ;DISPLAY
034F D1                 POP D           ;GET AND INC D
0350 14                 INR D
0351 D5                 PUSH D
0352 7A                 MOV A,D         ;SEE IF AT LEAST 4 DIG ENTERED
0353 FE04               CPI 4
0355 D49402             CNC DMEM        ;DISPLAY BYTE ACCESSED BY HL IF >3
0358 C34903             JMP WDLP
                ;
                ;       ENT: INITIALIZES FOR ENT2
                ;       ENT2: GETS AND SHIFTS IN A KEY ENTRY
                ;              DRIVEN BY ENTBY, ENTWD ONLY
                ;
035B E1         ENT:    POP H           ;PUSH D AHEAD OF CALL
035C 1600               MVI D,0
035E D5                 PUSH D
035F E5                 PUSH H
0360 210000             LXI H,0         ;INIT HL TO 0
```

```
0363 C9              RET
0364 CD3D02  ENT2:   CALL GETKY      ;GET A KEY
0367 D27103          JNC CMDKY       ;IF COMMAND, TERMINATE
036A 29              DAD H           ;ELSE MULT OLD BY 16 AND
036B 29              DAD H           ;ADD NEW
036C 29              DAD H
036D 29              DAD H
036E 85              ADD L
036F 6F              MOV L,A
0370 C9              RET             ;GO TO DRIVING ROUTINE
0371 47      CMDKY:  MOV B,A         ;SAVE COMMAND KEY
0372 D1              POP D           ;RESTORE STACK AND D
0373 D1              POP D
0374 C9              RET             ;RETURN TO CALLER OF CALLER OF ENT
                     ;
                     ;
                     ;       SEROT: SERIAL OUTPUT ROUTINE
                     ;ALLOWS DATA TO BE TRANSFERED TO TAPE (CASSETTE OR SIMILAR)
                     ;TRANSMISSION IS AT 110 BAUD.
                     ;STARTING ADDRESS IS PUT INTO HL
                     ;LENGTH OF BLOCK TO BE TRANSFERED IS PUT INTO DE
                     ;PROGRAM IS STARTED FROM THE KEYBOARD BY A RUN
                     ;
0375 F3      SEROT:  DI              ;DISABLE INTERRUPTS
0376 D5              PUSH D          ;SAVE DE
0377 AF              XRA A           ;CLEAR INTERRUPT ENABLE FOR DMA
0378 D3FA            OUT PORTC
037A 0601    OUTBY:  MVI B,1         ;SETUP FOR OUTPUT OF START BIT
037C 0E18            MVI C,DLY3      ;OUTPUT 3 DELAYS FOR SEPARATOR
037E CDA003          CALL DEL
0381 CD9E03          CALL D1         ;OUTPUT START BIT
0384 46              MOV B,M         ;GET DATA TO BE OUTPUTTED
0385 1608            MVI D,8         ;COUNT:  8 BITS AND 1 STOP BIT
0387 CD9E03  OUTBT:  CALL D1         ;OUTPUT BIT AND SHIFT
038A 15              DCR D           ;SEE IF DONE
038B C28703          JNZ OUTBT
038E 23              INX H           ;GO TO NEXT MEM LOCATION
038F D1              POP D           ;GET COUNT
0390 1B              DCX D           ;DECREMENT COUNT
0391 D5              PUSH D
0392 78              MOV A,E         ;SEE IF COUNT=0
0393 B2              ORA D
0394 C27A03          JNZ OUTBY
0397 0EFF            MVI C,0FFH      ;DELAY MAX.  END OF TRANSMISSION
0399 CDA003          CALL DEL
039C D1              POP D           ;RESET STACK
039D E7              RST 4           ;GO BACK
                     ;
                     ;       D1  DELAY AND OUTPUT ROUTINE FOR SEROT AND
```

```
                        ;SERIN.
                        ;
03AE 0E08       D1:     MVI C,DLY1      ;DELAY 1 BIT TIME
03A0 78         DEL:    MOV A,B         ;GET OUTPUT BYTE
03A1 E601               ANI 1           ;GET LS BIT
03A3 D3FA               OUT PORTC       ;OUTPUT
03A5 78                 MOV A,B
03A6 37                 STC             ;ROTATE AND LOAD CARRY (SHIFT
03A7 1F                 RAR             ;1'S FROM LEFT)
03A8 47                 MOV D,A
03A9 CD3602     DLP:    CALL DELAY      ;ACTUAL DELAY
03AC 0D                 DCR C           ;SEE IF DONE
03AD C2A903             JNZ DLP
03B0 C9                 RET
                        ;
                        ;
                        ;       SERIN: SERIAL INPUT ROUTINE THAT COMPLEMENTS
                        ;SEROT.  REGISTERS HL CONTAIN STARTING ADDRESS
                        ;
03B1 F3         SERIN:  DI              ;STOP INTERRUPTS
03B2 AF                 XRA A           ;CLEAR DMA BIT
03B3 D3FA               OUT PORTC
03B5 DBF9       SWT:    IN PORTB        ;GET DATA FROM PORT B
03B7 1F                 RAR             ;SHIFT INPUT INTO CARRY
03B8 DAB503             JC SWT          ;IF THERE, START NOT HIT YET
03BB 0E04       STFND:  MVI C,DELHF     ;DELAY  5 BIT TIMES
03BD CDA003             CALL DEL
03C0 DBF9               IN PORTB        ;MAKE SURE NOT A GLITCH
03C2 1F                 RAR
03C3 DAB503             JC SWT          ;IF ON AFTER 1/2 BIT, GO BACK
03C6 1E80               MVI E,80H       ;INITIALIZE COUNT
03C8 CD9E03     SILOP:  CALL D1         ;DELAY 1 BIT TIME
03CB DBF9               IN PORTB        ;GET INPUT
03CD 1F                 RAR             ;PUT INTO CARRY
03CE 7B                 MOV A,E         ;GET PREVIOUS BITS
03CF 1F                 RAR             ;SHIFT IN CARRY AND MOVE OLD BITS
03D0 5F                 MOV E,A         ;OVER AND STORE
03D1 D2C803             JNC SILOP
03D4 77                 MOV M,A         ;SAVE NEW BYTE
03D5 CD9E03             CALL D1         ;DELAY 1 BIT TIME
03D8 DBF9               IN PORTB        ;GET DATA
03DA 1F                 RAR
03DB D28F00             JNC ERR         ;IF NOT THERE, NO STOP BIT
03DE 23                 INX H           ;IF THERE, GOTO NEXT MEM LOC
03DF 1E64               MVI E,100       ;LOOK FOR LONG STOP TO
03E1 CD3602     FNDST:  CALL DELAY      ;SEE IF END OF TRANSMISSION
03E4 DBF9               IN PORTB        ;SEE IF STILL STOP
03E6 1F                 RAR
03E7 D2BB03             JNC STFND       ;IF NOT, FOUND START
03EA 1D                 DCR E           ;SEE IF LONG ENOUGH
```

```
03EB C2E103             JNZ FNDST
03EE E7                 RST 4               ;IF LONG ENOUGH, END
                ;
                ;
                ;MEMORY MAP AND EQUATE LISTS
                ;
F35D            BPCODE  EQU 0F35DH          ;CODE FOR BP IN BREAKPOINT
0079            ECODE   EQU 79H             ;CODE FOR E IN ERR
0050            RCODE   EQU 50H             ;CODE FOR R IN ERR
0015            NEXT    EQU 15H             ;CODE FOR KEYS--MUST ALSO CHANGE
0011            REG     EQU 11H             ;COMMAND LIST IF ASSIGNMENTS
0013            STEP    EQU 13H             ;ARE CHANGED
0014            RUN     EQU 14H
0010            MEM     EQU 10H
0012            GOTO    EQU 12H
0016            BRK     EQU 16H
0017            CLE     EQU 17H
00FB            CNTPT   EQU 0FBH            ;CONTROL PORT OF PIA
00F8            PORTA   EQU 0F8H            ;PORT A OF PIA
00FA            PORTC   EQU 0FAH            ;PORT C OF PIA
00F9            PORTB   EQU 0F9H            ;PIA PORT B
0008            DLY1    EQU 8               ;DELAY FOR 1 BIT
0018            DLY3    EQU 24              ;DELAY FOR SEPARATOR
0004            DELHF   EQU 4               ;HALF BIT DELAY
0040            DASH    EQU 40H             ;CODE FOR DASH
                ;
                ;
                ;MEMORY ASSIGNMENTS IN RAM
                ;
83CB                    ORG 83CBH
83CB            BOS:    DS 1                ;BOTTOM OF STACK  PSW
83CC                    DS 1                ;A
83CD                    DS 1                ;E
83CE                    DS 1                ;D
83CF                    DS 1                ;C
83D0                    DS 1                ;B
83D1                    DS 1                ;L
83D2                    DS 1                ;H
83D3            TOS:    DS 1                ;TOP OF STACK
83D4            SFLAG:  DS 1                ;STEP FLAG 1=STEP, 0=RUN
83D5            RGADDR: DS 1                ;REG DISPLAY CONTROL
83D6            SPADDR: DS 2                ;ADDR OF SP
83D8            MADDR:  DS 2                ;LAST EXAMINED MEMORY POS
83DA            PCADDR: DS 2                ;USERS PC
83DC            BKADDR: DS 2                ;POINTER TO BKPT TABLE
83DE            BKTBL:  DS 26               ;BREAKPOINT TABLE
83F8                    DS 7
83FF            LOWDGT: DS 1                ;LOW DIGIT OF DISPLAY
0000                    END RESET
```

ADROK	0157	BKADD	83DC	BKLOC	02DD	BKLOP	01E1
BKTBL	83DE	BKTST	0053	BL	02E2	BOK	01FF
BOS	83CB	BPCOD	F35D	BRK	0016	BRKPT	01B7
BYLP	0339	BYPAS	006B	CL	0289	CLE	0017
CLEAR	0287	CLFST	015A	CLP	0224	CLRBK	021C
CLRGT	0282	CLRLP	028C	CMD	0098	CMD2	00A4
CMD3	00A6	CMDKY	0371	CNTPT	00FB	D1	039E
DASH	0040	DBKPT	0208	DBY2	0298	DBYTE	0295
DEL	03A0	DEL1	0238	DELAY	0236	DELHF	0004
DLOOP	0246	DLP	03A9	DLY1	0008	DLY3	0018
DMEM	0294	DREG	0303	DREG2	0193	DRSAV	0190
DRTBL	0135	DWD2	02D4	DWORD	02D1	DYEN	02A4
DYPC	02CE	ECODE	0079	ENT	035B	ENT2	0364
ENTBY	0336	ENTWD	0346	ERR	008F	FDR	00EE
FK	0275	FNDST	03E1	GETKY	023D	GLP	00E0
GOTO	0012	GOTOC	0166	INPUT	025C	KP	0273
KSCAN	026F	KYTST	0062	LOWDG	83FF	MADDR	83D8
MATCH	02F7	MEM	0010	MEMCH	00D0	NEXT	0015
NEXTC	00CB	NOBKC	01F4	NOMAT	02EF	NORUN	0071
NOTPC	017C	NOUPD	0131	OFFSE	02A9	OKVAL	01CE
OLDAD	0113	OLDBP	0216	OUTBT	0387	OUTBY	037A
PCADD	83DA	PORTA	00F8	PORTB	00F9	PORTC	00FA
PTR	00BB	RCODE	0050	REG	0011	REGC	0182
RERUN	015D	RESET	0000	RGADD	83D5	RGTBL	0326
RLP	0013	RNE	01A7	RST4	0020	RST5	0028
RST6	0030	RST7	0038	RSTDY	0244	RUN	0014
RUNP	0145	SAVEH	0110	SAVEM	010F	SAVEP	00EE
SCAN	0257	SCRET	0280	SERIN	03B1	SEROT	0375
SFLAG	83D4	SILOP	03C8	SPADD	83D6	SPLIT	02C2
STEP	0013	STEPP	0144	STFND	03BB	SWT	03B5
TABLE	02B2	TOS	83D3	VOK	0101	WDLP	0349

MICROCOMPUTER TRAINING WORKBOOK

APPENDIX C

HARDWARE LAYOUT AND TEST PROCEDURE

ICS MICROCOMPUTER TRAINING SYSTEM POWER REQUIREMENTS

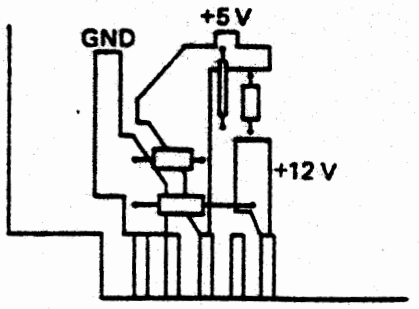
The Microcomputer Training System is delivered as a ready-to-use unit requiring only connection of power supplies for operation. The MTS is designed to operate with only two DC power supplies, +12V and +5V. Among the parts used in the MTS, only the 8080A requires another supply voltage (-5V), which is generated internally.

Both DC power supplies should have sufficient current margin over the following power dissipation specifications :

MTS Power Dissipation	+ 5V ± 5%	1.0 A Max.
	+12V ± 5%	150 mA Max.

External power supply lines should be connected to the board edge finger pins marked +5V, +12V and GND.

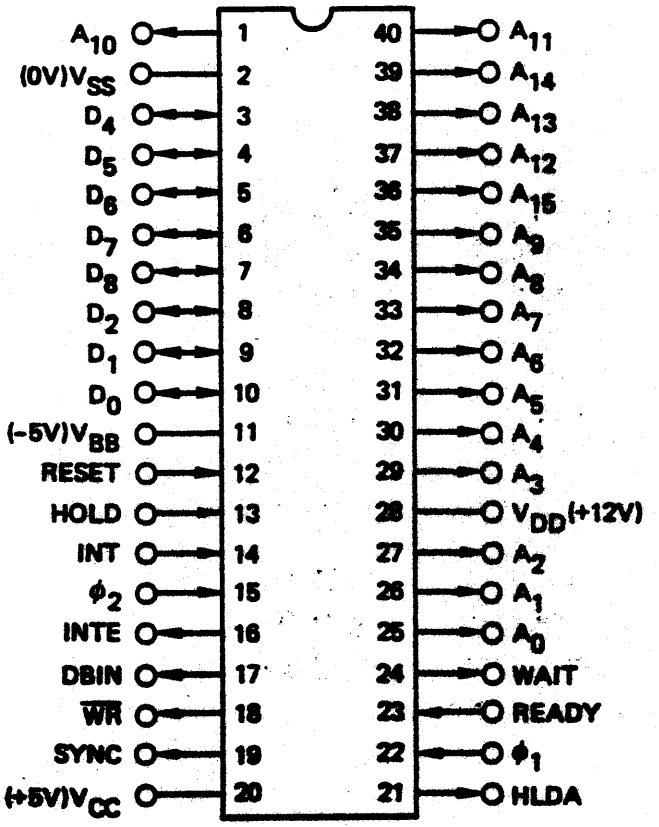
Board Edge Finger Pins.



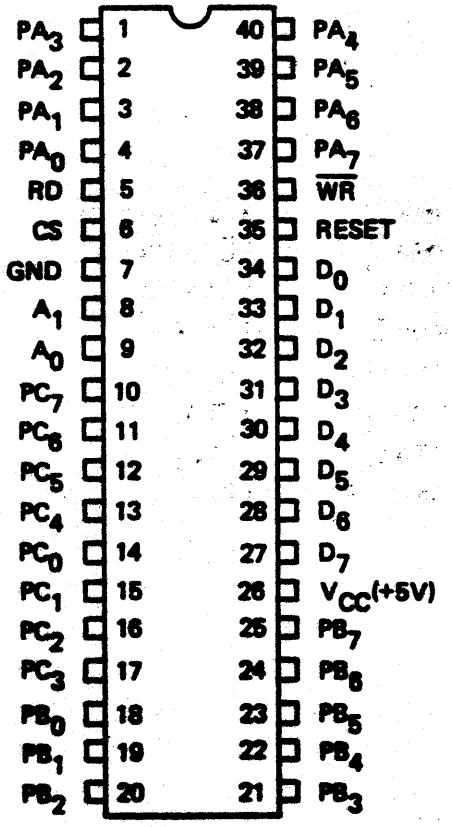
ASSIGNMENT OF BOARD EDGE FINGER PINS

Pin #	Pin Name	Pin #	Pin Name
A 1	GND	B 1	GND
2	GND	2	GND
3	+5V	3	+5V
4		4	
5	+12V	5	+12V
6		6	
7		7	
8		8	
9		9	
10	AB15	10	AB 7
11	AB14	11	AB 6
12	AB13	12	AB 5
13	AB12	13	AB 4
14	AB11	14	AB 3
15	AB10	15	AB 2
16	AB 9	16	AB 1
17	AB 8	17	AB 0
18		18	
19		19	
20		20	
21		21	
22		22	
23		23	
24		24	
25		25	
26		26	DB 7
27		27	DB 6
28		28	DB 5
29		29	DB 4
30		30	DB 3
31		31	DB 2
32		32	DB 1
33		33	DB 0
34		34	
35		35	
36		36	
37		37	
38		38	
39		39	
40		40	
41		41	
42		42	
43		43	
44		44	
45		45	
46		46	
47		47	
48		48	
49		49	
50	GND	50	GND

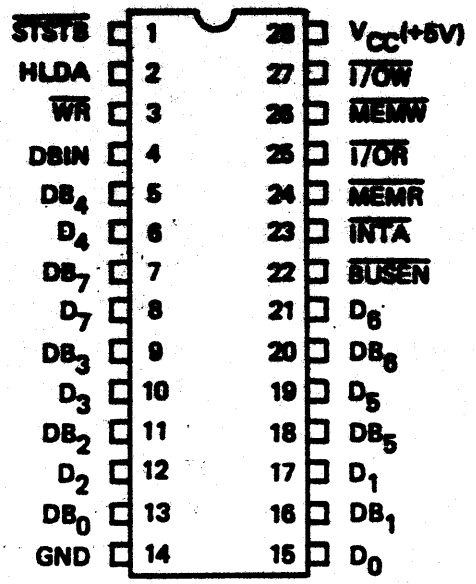
8080A
CENTRAL PROCESSOR UNIT



8255
PROGRAMMABLE PERIPHERAL INTERFACE



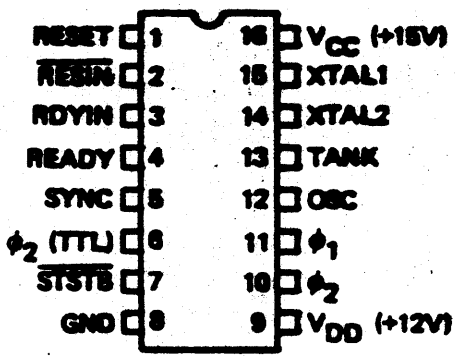
8228
SYSTEM CONTROLLER AND BUS DRIVER



PIN CONFIGURATION OF LSI'S AND IC'S USED IN THE MTS

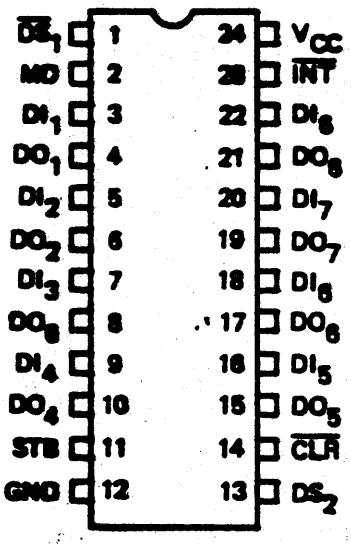
8224

Clock Generator and Driver



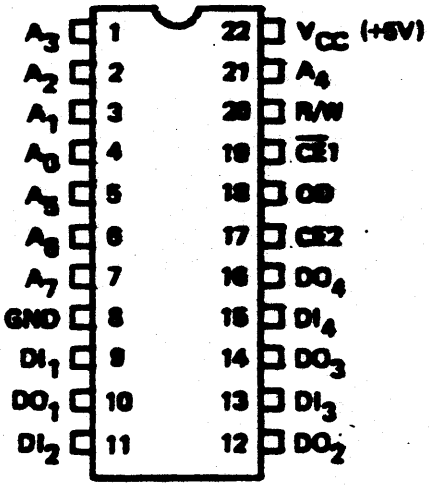
8212

8-Bit I/O Port



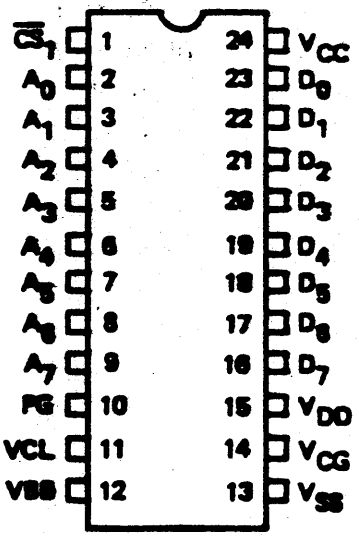
5101

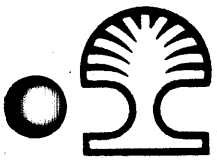
1,024 Bit Static RAM



454

2,048 Bit EEPROM





ICS MICROCOMPUTER TRAINING SYSTEM TEST PROCEDURE

STEP 1 Connect power supply and turn on +5V and +12V. Switch settings are 'ENABLE' and 'AUTO'. LED should display

8200 [ ]

STEP 2 Enter test program as follows by passing indicated keys :

Press Keys	Display Should be :
MEM 2 1	8200 21
NEXT 0 0	8201 00
NEXT 8 4	8202 84

Continue entering remainder of program from attached coding sheets from address 8203 to 8248. Once program has been entered, check to be sure that all instructions have been entered correctly by pressing the keys :

RST NEXT NEXT ..... etc.

STEP 3 Test the RAM as follows :

Press Keys	Display Should be :
ADDR	8200 21
RUN	81FF* [ ]

*(IF YOU HAVE 1024 BYTES OF MEMORY THIS DISPLAY SHOULD BE 7FFF)

STEP 4 Test the ROM and keyboard as follows :

Press Keys	Display Should be :	
0	145B	5D00
1	8E4C	2601
2	50DB	F302
3	60BD	9903
4		04
5		05
6		06
7		07
8		08
9		09

STEP 4 (cont'd) Press Keys	Display Should be :
A	0A
B	0B
C	0C
D	0D
E	0E
F	0F

Step 5 Test the command keys (excluding RST) as follows :

Press Keys	Display Should be :
REG	11
MEM	10
BRK	16
CLR	17

Step 5 (cont'd)

Press Keys	Display Should be :
ADDR	12
RUN	14
STEP	13
NEXT	15

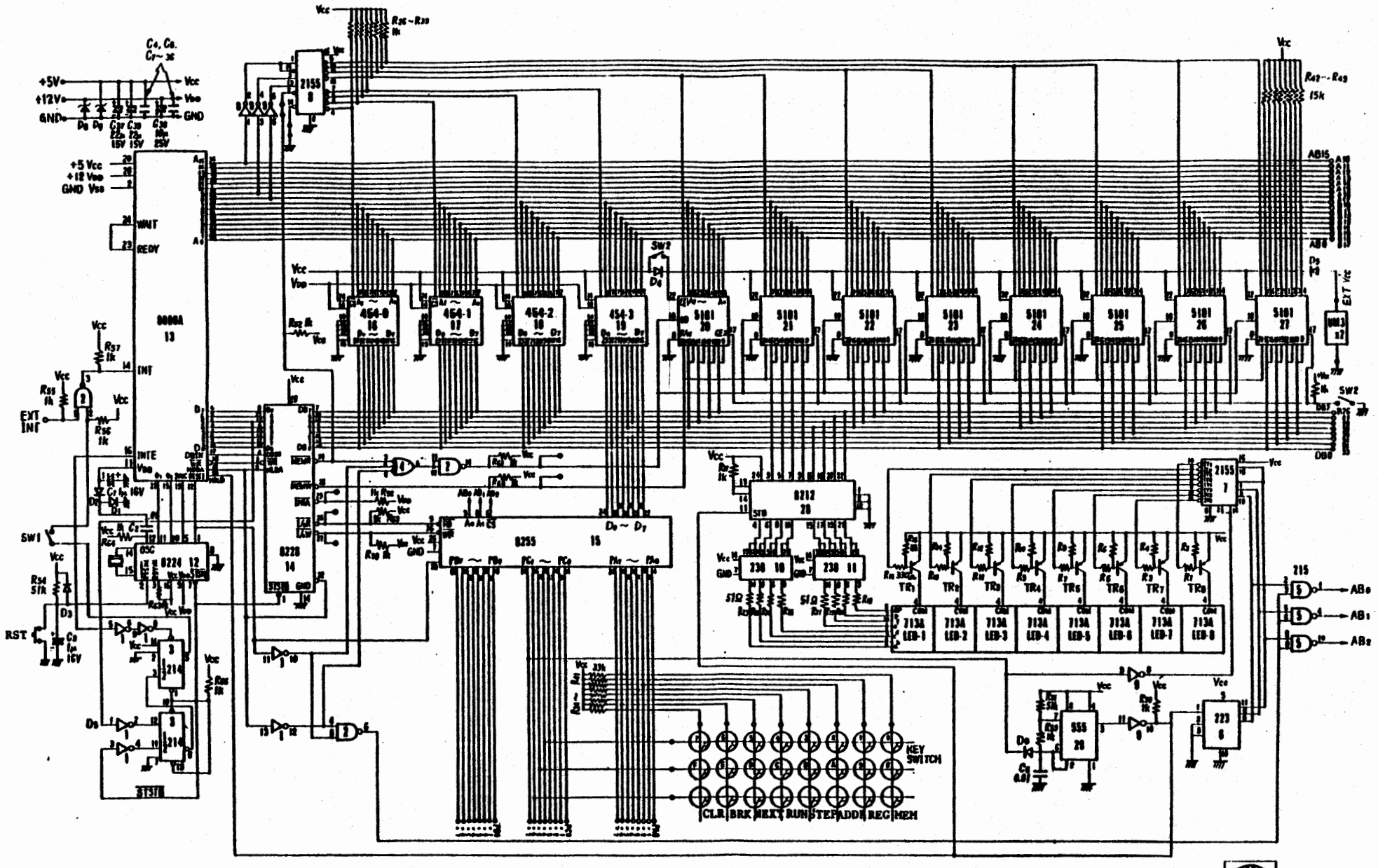
Repeat keys 0,1,2,3, and check previous list of displays for these keys.

The basic functions of the MTS are operational if all displays specified above have occurred during test sequence.

Note : If the RAM test fails, try the ROM test by pressing the following sequence of keys :

Press Keys

RST	ADDR	8	2	2	0	RUN



SCHEMATIC – MICROCOMPUTER TRAINING SYSTEM
INTEGRATED COMPUTER SYSTEMS, INC.

c/10/77

RAM TEST

ADDR	CODE											
8200	21		LXI	H,	8400							Address above highest
1	00											memory location
2	84											
3	11		LXI	D,	8220							Address above test
4	20											program
5	82											
8206	1D		DCR	E								Decrement test
7	C2		JNZ	820C								program address
8	0C											
9	82											
A	1E		MVI	E,	20							At 8200 set 8220
B	20											
820C	2B		DCX	H								Decrement test address
D	4E		MOV	C,	M							Save memory data
E	1A		LDAX	D								Copy test program
F	77		MOV	M,	A							byte to memory
8210	BE		CMP	M								Test for success
1	C2		JNZ	821A								Exit on error
2	1A											
3	82											
4	71		MOV	M,	C							Restore memory data
5	7E		MOV	A,	M							
6	B9		CMP	C								Test for success
7	CA		JZ	8206								Loop if OK
8	06											
9	82											
821A	CD		CALL	DWORD								Exit - display
B	D1											location that could
C	02											not be written
D	CD		CALL	CLRGT								
E	82											
F	02											
8000												
1												
2												
3												
4												
5												
6												
7												
8												

ROM AND KEYBOARD TEST

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

ADDR		CODE														
8220	0	C	D		C	A	L	L		G	E	T	K	Y		
	1	3	D													
	2	0	2													
	3	C	D		C	A	L	L		C	L	E	A	R		
	4	8	7													
	5	0	2													
	6	C	D		C	A	L	L		D	B	Y	T	E		
	7	9	5													
	8	0	2													
	9	7	9		M	Ø	V			A	,	C				
	A	F	E		C	P	I			0	4					
	B	0	4													
	C	D	2		J	N	C			8	2	2	0			
	D	2	0													
	E	8	2													
	F	D	5		P	U	S	H		D						DISPLAY ADDRESS
8223	0	5	1		M	Ø	V			D	,	C				HIGH ADDRESS = KEY
	1	A	F		X	R	A			A						CLEAR LRC
	2	5	F		M	Ø	V			E	,	A				CLEAR LOW ADDRESS
	3	6	7		M	Ø	V			H	,	A				CLEAR ARITHMETIC
	4	6	F		M	Ø	V			L	,	A				SUM
	5	4	7		M	Ø	V			B	,	A				CLEAR FOR ADD
	6	E	B		X	C	H	G								ADDRESS RAM
	7	4	E		M	Ø	V			C	,	M				GET BYTE
	8	E	B		X	C	H	G								SUM TO HL
	9	0	9		D	A	D			B						ADD BYTE
	A	A	9		X	R	A			C						FORM LRC
	B	1	C		I	N	R			E						INCR ADDRESS
	C	C	2		J	N	Z			8	2	3	5			
	D	3	5													
	E	8	2													
	F	D	1		P	Ø	P			D						DISPLAY ADDRESS
8240	0	C	D		C	A	L	L		D	B	Y	Z			
	1	9	8													
	2	0	2													
	3	C	D		C	A	L	L		D	W	D	2			
	4	D	4													
	5	0	2													
	6	C	3		J	M	P			8	2	2	0			
	7	2	0													
	8	8	2													

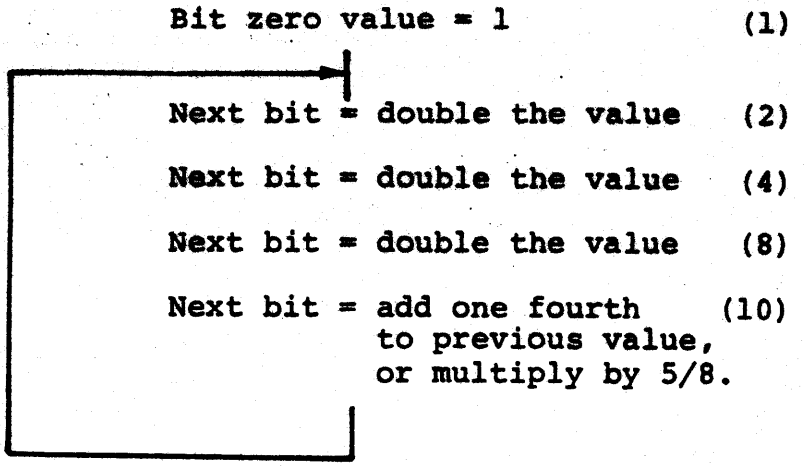
Appendix D

BINARY/DECIMAL CONVERSIONS

Several programs are presented for conversion of decimal data to binary data. All of these are written as subroutines; generally the data to be converted (or a memory address for the data) are entered in register pair HL and the result is returned in the same, with all other registers preserved.

D-1 DECIMAL TO BINARY INTEGER

The conversion from decimal data to binary can be done by calculating and summing the values of the successive bits. Figure D-1 lists the values of the bits. These can be calculated by this procedure.



Bit	Decimal Value	Binary Value
0	1	0001
1	2	0002
2	4	0004
3	8	0008
4	10	000A
5	20	0014
6	40	0028
7	80	0050
8	100	0064
9	200	00C8
10	400	0190
11	800	0320
12	1000	03E8
13	2000	07D0
14	4000	0FA0
15	8000	1F40
16	10,000	2710
20	100,000	186A0
24	1,000,000	F4240
28	10,000,000	989680

Values of Bits in a Decimal Number
Figure D-1

The bit value can be calculated and added into the sum representing the binary value as each bit of the decimal value is processed, or they can be pre-calculated and stored. It is faster and simpler to store a table of the bit values, but this requires memory for the storage, as shown in the program of Figure D-2. The procedure of Figure D-3 calculates the values and pushes them into the stack; then recovers each bit value as the decimal value is shifted. Thus no memory is permanently allocated to the bit value. The stack is used for 38 bytes - six to save registers and 32 for bit values. Either subroutine meets the same specification, except for length.

DECBN Convert four digit packed decimal value to two byte binary.

Enter with decimal value in (HL)

Return with binary value in (HL)

All other registers are preserved.

The program of Figure D-4 can be used to test either of these programs.

DECIMAL TO BINARY WITH TABLE

ADDR		CODE													
8222	0	F5		PUSH		PSW									
	1	D5		PUSH		D									
	2	C5		PUSH		B									
	3	11		LXI		D,	0	0	0	0					
	4	00													
	5	00													
	6	01		LXI		B,	8	2	4	0					Table address for
	7	40													low byte of most
	8	82													significant bit
8229	9	0A		LDAX		B									(A) ← low byte value
	A	03		INX		B									for MSB bit
	B	29		DAD		H									Shift decimal value
	C	D2		JNC		8	2	3	4						Skip add if
	D	34													decimal bit = 0
	E	82													
	F	83		ADD		E									Add table value
8230	0	5F		MOV		E,	A								to decimal value
	1	0A		LDAX		B									in (DE)
	2	8A		ADC		D									
	3	57		MOV		D,	A								
8234	4	03		INX		B									Address value of next
	5	7D		MOV		A,	L								lower bit. Test
	6	B4		ORA		H									binary value for 0
	7	C2		JNZ		8	2	2	9						Loop until all
	8	29													non-zero bits
	9	82													have been converted
	A	EB		XCHG											(HL) ← result
	B	C1		POP		B									Restore registers
	C	D1		POP		D									
	D	F1		POP		PSW									
	E	C9		RET											
	F														
8	0			ENTER		WITH									
	1				(HL)	=	PACKED DECIMAL								
	2						4		DIGITS						
	3			RETURN											
	4				(HL)	=	BINARY EQUIVALENT								
	5			ALL		OTHER			REGISTERS						
	6				PRESERVED										
	7														
	8														

FIGURE D-2a

TABLE OF BIT VALUES

ADDR	CODE	
8240	40	Value of bit 15
1	1F	
2	A0	Bit 14
3	0F	
4	D0	Bit 13
5	07	
6	E8	Bit 12
7	03	
8	20	Bit 11
9	03	
A	90	Bit 10
B	01	
C	C8	Bit 9
D	00	
E	64	Bit 8
F	00	
8250	50	Bit 7
1	00	
2	28	Bit 6
3	00	
4	14	Bit 5
5	00	
6	0A	Bit 4
7	00	
8	08	Bit 3
9	00	
A	04	Bit 2
B	00	
C	02	Bit 1
D	00	
E	01	Bit 0
F	00	
8 0		
1		
2		
3		
4		
5		
6		
7		
8		

FIGURE D-2.4

DECIMAL TO BINARY WITH STACK

ADDR		CODE														
8220	0	F5		PUSH		PSW										
	1	D5		PUSH		D										
	2	C5		PUSH		B										
	3	EB		XCHG												(DE) ← decimal value
	4	3E		MVI		A,	04									To count 4 digits
	5	04														
	6	21		LXI		H,	0001									
	7	01														
	8	00														Store values
8229	9	E5		PUSH		H										1 10 100 1000
	A	29		DAD		H										
	B	E5		PUSH		H										2 20 200 2000
	C	4D		MOV		C,	L									
	D	44		MOV		B,	H									
	E	29		DAD		H										
	F	E5		PUSH		H										4 40 400 4000
8230	0	29		DAD		H										
	1	E5		PUSH		H										8 80 800 8000
	2	09		DAD		B										
	3	3D		DCR		A										
	4	C2		JNZ		8229										
	5	29														
	6	82														
	7	EB		XCHG												(HL) ← decimal value
	8	11		LXI		D,	0000									Clear binary value
	9	00														
	A	00														
	B	3E		MVI		A,	10									To count 16 bits
	C	10														
823D		C1		POP		B										(BC) ← bit value
	E	29		DAD		H										(CY) ← high bit
	F	D2		JNC		8245										Skip add if
8240		45														decimal bit = 0
8241		82														
	2															
	3			ENTER		WITH										
	4			(HL)	=	PACKED DECIMAL										
	5					4		DIGITS								
	6			RETURN												
	7			(HL)	=	BINARY EQUIVALENT										
	8															

DECIMAL TO BINARY WITH STACK

ADDR		CODE													
8	0														
	1														
824	2	EB			XCHG										(HL) ← binary value
	3	09			DAD			B							Add bit values
	4	EB			XCHG										(HL) ← decimal value
824	5	3D			DCR			A							Count bits
	6	C2			JNZ			823D							Loop until stack
	7	3D													cleared of all
	8	82													bit values
	9	EB			XCHG										
	A	C1			POP			B							Restore BC
	B	D1			POP			D							Restore DE
	C	F1			POP			PSW							Restore AF
	D	C9			RET										
	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 D-3.6

TEST FOR DECIMAL TO BINARY

ADDR	CODE			
8200	CD		CALL	ENTWD
1	46			
2	03			
3	CD		CALL	DECBN
4	20			
5	82			
6	11		LXI	D, 83FF
7	FF			
8	83			
9	CD		CALL	DWDZ
A	D4			
B	02			
C	C3		JMP	8200
D	00			
E	82			
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 D-4

A single byte conversion can use either of the foregoing procedures, but a simpler method results from separating the two decimal digits. The low digit, with a value from 0 to 9, is already in binary as well as binary coded decimal form. The high digit, 00 to 90, can be converted by a binary multiplication by 5/8, which only takes five steps.

```
RAR         (A) ← X/2
MOV E,A     (C) ← X/2
RAR         (A) ← X/4
RAR         (A) ← X/8
ADD E       (A) ← X/2 + X/8
```

Figure D-5 shows the complete subroutine, which accepts the two digit decimal number in (L) and returns the binary equivalent in (L).

DECBI Decimal to Binary Integer - one byte

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

ADDR	CODE			
8220	F5		PUSH PSW	
1	7D		MOV A,L	
2	E6		ANI 0F	(A) ← low digit
3	0F			
4	67		MOV H,A	(H) ← low digit
5	AD		XRA L	(A) ← high digit
6	1F		RAR	
7	6F		MOV L,A	(L) ← 1/2 high digit
8	1F		RAR	
9	1F		RAR	(A) ← 1/8 high digit
A	85		ADD L	(A) ← 5/8 high digit
B	84		ADD H	(A) ← binary value
C	6F		MOV L,A	
D	F1		POP PSW	
E	C9		RET	
F				
8200				
1				
2				
3				
4			ENTER WITH	
5			(L) = PACKED DECIMAL	
6			2 DIGITS	
7			RETURN	
8			(L) = BINARY EQUIVALENT	
9			USES REGISTER H	
A			ALL OTHERS PRESERVED	
B				
C				
D				
E				
F				
8300				
1				
2				
3				
4				
5				
6				
7				
8				FIGURE D-5

The procedure of Figure D-5 can also be used with multi-byte values. Almost any realistic program that requires decimal to binary conversion will also have a binary multiplication subroutine, which can be used to multiply the value of the two digit number by an appropriate power of 10 expressed in binary. These values can be stored in a table, or they can also be calculated by binary multiplication. This scheme is by far the best when more than four digits are involved.

D-2 DECIMAL FRACTION TO BINARY FRACTION

Surprisingly, the conversion of a decimal fraction to a binary fraction is significantly simpler than the conversion of integers. The decimal fraction is repeatedly doubled: if a carry out of the fraction results, a one is shifted into the binary value; if no carry occurs, a zero is shifted in. Figure D-6 shows a 16 bit conversion program. For larger numbers of bits, the data would be kept in memory, and the procedure can then be extended to any desired precision.

DCFBF Decimal Fraction to Binary Fraction

ADDR		CODE														
8260	0	F5			PUSH		PSW									Save registers
	1	D5			PUSH		D									
	2	11			LXI		D,	0001								Clear binary fraction
	3	01														with marker in
	4	00														low bit
8265	5	7D		→	MOV		A,	L								Double decimal
	6	87			ADD		A									fraction in (HL)
	7	27			DAA											
	8	6F			MOV		L,	A								
	9	7C			MOV		A,	H								
	A	8F			ADC		A									
	B	27			DAA											
	C	67			MOV		H,	A								
	D	7B			MOV		A,	E								Shift carry from
	E	17			RAL											decimal fraction
	F	5F			MOV		E,	A								into binary fraction
8270	0	7A			MOV		A,	D								
	1	17			RAL											
	2	57			MOV		D,	A								
	3	D2		-	JNC		8265									Loop until marker
	4	65														bit shifts out
	5	82														
	6	EB			XCHG											(HL) ← binary fraction
	7	D1			POP		D									Restore other
	8	F1			POP		PSW									registers
8279	9	C9			RET											
	A															
	B															
	C															
	D															
	E															
	F															
8	0															
	1															
	2															
	3															
	4															
	5															
	6															
	7															
	8															FIGURE D-6

D-3 BINARY TO DECIMAL CONVERSION

Since each bit in a binary number, either integer or fraction, has twice the value of the preceding bit, this conversion starts with a decimal value for the least significant bit and repeatedly doubles that value for succeeding bits. The successive bits of the binary value are tested, and each time a one is encountered, the bit value is summed into the decimal value.

The program of Figure D-7 operates in memory rather than in registers, and allows conversion of any number of bytes. It demonstrates passing parameters to a subroutine through memory with a command and address table. Five areas in memory are required:

Binary Data

Decimal Result

Temporary Bit Value

Value of Least Significant Bit

Command and Address Table

The conversion subroutine is entered with (HL) = address of the command and address table, which contains (in this order):

Number of binary bytes to be converted

Number of decimal bytes

Binary data address

Result address

Temporary bit value address

LSB value address

} address for least significant byte

The conversion program alters only the result and the temporary bit value. None of the other data are changed, so the binary value remains available for further processing and the other data could be stored in ROM.

A subroutine, RECAD, recovers these addresses and places them in registers for use in initialization and in the repetitive conversion loop. In the initialization, the least significant bit value is copied from its permanent location to the temporary bit value area, and the result area is cleared.

In the loop, RECAD is called with a byte count in register C (initially set to 00), and RECAD adds this value to the binary data address from the table, returning the address of the binary data byte now being processed. The data byte addressed is masked by the content of register B (initially set to 01 and subsequently shifted left), giving the value of the current bit.

If the current bit is one, another subroutine, DCADM, is called to add the decimal value of the bit (addressed by BC) to the decimal result (addressed by HL). Then the bit value address is duplicated in HL and another call to DCADM adds the bit value to itself, giving the value of the next higher bit.

At the end of the loop, the bit mask and byte count are recovered, and the bit mask in register B is rotated left before repeating the loop. When it shifts from bit 7 back to bit 0, the byte count is incremented and compared with the number of bytes to be converted.

The command table shown is suitable for conversion of a four byte binary value with 16 integer bits and 16 fractional bits. The coding given is for locations 8280 to 82F4, with the command table, LSB value and scratch pad in 8300-831F; binary data anc decimal result in 8320-832E.

BINARY TO DECIMAL CONVERSION - INITIALIZE

ADDR		CODE													
8 2 8	0	F	5		P	U	S	H		P	S	W			Save registers
	1	C	5		P	U	S	H		B					
	2	D	5		P	U	S	H		D					
	3	E	5		P	U	S	H		H					Save command addr
	4	2	3		I	N	X		H						
	5	4	6		M	O	V		B	,	M				(B) ← decimal bytes
	6	0	E		M	V	I		C	,	0	0			Set 0 offset
	7	0	0												for RECAD
	8	2	3		I	N	X		H						Skip address for
	9	2	3		I	N	X		H						binary data
	A	2	3		I	N	X		H						Address result address
	B	C	D		C	A	L	L		R	E	C	A	D	(HL) ← result address
	C	C	C												(DE) ← bit value addr
	D	8	2												(BC) ← LSB value addr
8 2 8	E	7	7	→	M	O	V		M	,	A				(M) ← decimal bytes
	F	0	A		L	D	A	X		B					Copy LSB value
8 2 9	0	1	2		S	T	A	X		D					into bit value
	1	7	E		M	O	V		A	,	M				(A) ← decimal bytes
	2	3	6		M	V	I		M	,	0	0			Clear decimal
	3	0	0												result
	4	2	3		I	N	X		H						
	5	1	3		I	N	X		D						
	6	0	3		I	N	X		B						
	7	3	D		D	C	R		A						
	8	C	2	⊢	J	N	Z		8	2	8	E			
	9	8	E												
	A	8	2												
	B	E	1		P	O	P		H						Restore command addr
	C	0	1		L	X	I		B	,	0	1	0	0	
	D	0	0												(C) ← byte count = 00
	E	0	1												(B) ← bit mask = 01
	F	0	0		N	O	P								
8	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

FIGURE D-7a

BINARY TO DECIMAL CONVERSION — LOOP

ADDR	CODE		Mnemonic	Operand	Comment
8 2A 0	E	5	PUSH	H	Save command address
1	C	5	PUSH	B	Save mask, count
2	2	3	INX	H	Address decimal bytes
3	E	5	PUSH	H	
4	2	3	INX	H	
5	C	D	CALL	RECAD	(HL)←binary data addr
6	C	C			(DE)←result address
7	8	2			(BC)←bit value address
8	A	6	ANA	M	Mask for current bit
9	E	1	POP	H	Address decimal bytes
A	7	E	MOV	A, M	(A)←decimal bytes
B	E	B	XCHG		(HL)←result addr
C	C	4	CNZ	DCADM	If bit from binary
D	E	0			data = 1 add bit
E	8	2			value to result
F	6	9	MOV	L, C	Duplicate bit value
8 2B 0	6	0	MOV	H, B	address in (HL)
1	C	D	CALL	DCADM	Add bit value to
2	E	0			itself
3	8	2			
4	C	1	POP	B	Recover mask, count
5	E	1	POP	H	Recover command addr
6	7	8	MOV	A, B	
7	0	7	RLC		Rotate bit mask
8	4	7	MOV	B, A	left and restore
9	D	2	JNC	82A0	Loop unless
A	A	0			mask restored
B	8	2			to right bit
C	0	C	INR	C	Count bytes
D	7	E	MOV	A, M	Compare with
E	B	9	CMP	C	final byte count
F	D	2	JNC		Loop unless
8 2C 0	A	0			final byte finished
1	8	2			
2	D	1	POP	D	EXIT - restore
3	C	1	POP	B	registers saved
4	F	1	POP	PSW	at Initialize
82C 5	C	9	RET		
6					
7					
8					[illegible]

INTEGRATED COMPUTER SYSTEMS MICROCOMPUTER TRAINING SYSTEM CODING SHEET

ECAD Recover Addresses from Memory

CODING SHEET

ADDR		CODE																
8	0					E	N	T	E	R		A	T		8	34C	WITH	
	1						(	H	L	)	=	A	D	D	R	ESS	OF	ADDRESSES
	2						(	C	)		=	O	F	F	S	ET	FROM	FIRST
	3																	
	4					R	E	T	U	R	N							
	5						(	H	L	)	=	F	I	R	S	T	ADDRESS	
	6											P	L	U	S	OFFSET	(C)	
	7						(	D	E	)	=	S	E	C	O	ND	ADDRESS	
	8						(	B	C	)	=	T	H	I	R	D	ADDRESS	
	9						(	A	)		=	E	N	T	R	Y	CONTENT	
	A											O	F		R	EGISTER	B	
	B																	
82C	C	7	E			M	O	V		A	,	M						
	D	8	1			A	D	D		C								Add offset
	E	5	F			M	O	V		E	,	A						in (C) to
	F	2	3			I	N	X		H								first address
82D	0	7	E			M	O	V		A	,	M						
	1	C	E			A	C	I		0	0							
	2	0	0															
	3	5	7			M	O	V		D	,	A						
	4	D	5			P	U	S	H		D							Save in stack
	5	7	8			M	O	V		A	,	B						Return (B) in (A)
	6	2	3			I	N	X		H								
	7	5	E			M	O	V		E	,	M						(DE) ← second
	8	2	3			I	N	X		H								address
	9	5	6			M	O	V		D	,	M						
	A	2	3			I	N	X		H								
	B	4	E			M	O	V		C	,	M						(BC) ← third
	C	2	3			I	N	X		H								address
	D	4	6			M	O	V		B	,	M						
	E	E	1			P	O	P		H								(HL) ← first
	F	C	9			R	E	T										address + offset
8	0																	
	1																	
	2																	
	3																	
	4																	
	5																	
	6																	
	7																	
	8																	

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

FIGURE D-7e

DCADM Decimal Add in Memory D - 19 (I/12/77)

ADDR		CODE													
8 2 E	0	E	5		P	U	S	H		H					
	1	D	5		P	U	S	H		D					
	2	C	5		P	U	S	H		B					
	3	5	F		M	O	V		E	→	A				(E) ← byte count
	4	5	7		M	O	V		D	→	A				(D) ← byte count
	5	A	F		X	R	A		A						
8 2 E	6	0	A		L	D	A	X		B					
	7	8	E		A	D	C		M						
	8	2	7		D	A	A								
	9	7	7		M	O	V		M	→	A				
	A	0	3		I	N	X		B						
	B	2	3		I	N	X		H						
	C	1	D		D	C	R		E						
	D	C	2		J	N	Z		8	2	E	6			
	E	E	6												
	F	8	2												
8 2 F	0	7	A		M	O	V		A	→	D				(A) ← byte count
	1	C	1		P	O	P		B						
	2	D	1		P	O	P		D						
	3	E	1		P	O	P		H						
	4	C	9		R	E	T								
	5														
	6				E	N	T	E	R		A	N	D		RETURN WITH
	7					(	H	L	)	=	A	U	G	E	ND ADDRESS
	8					(	B	C	)	=	A	D	D	E	ND ADDRESS
	9					(	A	)		=	B	Y	T	E	COUNT
	A														
	B				A	D	D	E	N	D		P	R	E	SERVED
	C				A	U	G	E	N	D		R	E	P	LACED BY SUM
	D														
	E				A	L	L		R	E	G	I	S	T	ERS PRESERVED
	F					E	X	C	E	P	T		F	L	AGS
8	0														
	1				C	Y		S	E	T		I	F		CARRY FROM
	2					H	I	G	H		B	Y	T	E	ADDITION
	3														
	4														
	5														
	6														
	7														
	8														

COMMAND TABLE AND LSB VALUE

ADDR		CODE	
830	0	0	4
	1	0	B
	2	2	0
	3	8	3
	4	2	4
	5	8	3
	6	1	5
	7	8	3
	8	0	A
	9	8	3
	A	2	5
	B	0	6
	C	8	9
	D	8	7
	E	2	5
	F	1	5
831	0	0	0
	1	0	0
	2	0	0
	3	0	0
	4	0	0
	5	0	0
	6	0	0
	7	0	0
	8	0	0
	9	0	0
	A	0	0
	B	0	0
	C	0	0
	D	0	0
	E	0	0
	F	0	0
8	0		
	1		
	2		
	3		
	4		
	5		
	6		
	7		
	8		

4 BINARY BYTES
11 DECIMAL BYTES
BINARY DATA ADDRESS
DECIMAL RESULT ADDRESS
SCRATCH PAD ADDRESS
ADDRESS FOR LSB VALUE
VALUE OF LSB (11 BYTES)

SCRATCH PAD (11 BYTES)

FIGURE

D-4 BINARY FRACTION TO DECIMAL FRACTION

The program of Figure D-8 is a shortened version of the binary to decimal conversion, taking a two byte binary fraction in (HL) and returning the two byte decimal equivalent in (HL). For economy of program space it does not save the other registers, and returns only the two high bytes of the result in (HL). The other bytes of the conversion are stored in memory, with the least significant at 8308 and most significant at 830F. It requires that its scratch pad and result area occupy the lowest 16 bytes of the page immediately following the least significant bit value, which is stored at 82F8-82FF. The program would work for integers or mixed integer/fraction values if a different LSB value were stored in that location.

BFDCF - Binary Fraction to Decimal Fraction

MICROCOMPUTER TRAINING SYSTEM
INTEGRATED COMPUTER SYSTEMS
CODING SHEET

ADDR		CODE													
8 2 C	0	E	B		X	C	H	G							(DE) ← binary fraction
	1	2	1		L	X	I		H	,	8	3	1	0	Address beyond
	2	1	0												scratch pad and
	3	8	3												result area
	4	A	F		X	R	A		A						Clear scratchpad
	5	2	D		D	C	R		L						and result
	6	7	7		M	O	V		M	,	A				(8300 - 830F)
	7	C	2		J	N	Z		8	2	C	5			End with
	8	C	5												(HL) = 8300
	9	8	2												
	A	0	1		L	X	I		B	,	8	2	F	8	Address least
	B	F	8												significant byte
	C	8	2												of LSB value
8 2 C	D	C	D		C	A	L	L			8	2	E	5	((HL)) ← ((HL)) + ((BC))
	E	E	5												for 8 bytes
	F	8	2												Return (BC) = 8300
8 2 D	0	7	A		M	O	V		A	,	D				(HL) = 8308
	1	1	F		R	A	R								
	2	5	7		M	O	V		D	,	A				Shift binary
	3	7	B		M	O	V		A	,	E				value right
	4	1	F		R	A	R								one bit
	5	5	F		M	O	V		E	,	A				
	6	D	C		C	C			8	2	E	5			If LSB = 1 add
	7	E	5												bit value (8300-8307)
	8	8	2												to result (8308-830F)
	9	6	9		M	O	V		L	,	C				(HL) ← 8300
	A	7	A		M	O	V		A	,	D				Test binary value
	B	B	3		O	R	A		E						for zero
	C	C	2		J	N	Z		8	2	C	D			Loop until zero
	D	C	D												
	E	8	2												
	F	2	A		L	H	L	D			8	3	0	E	(HL) ← result
8 2 E	0	0	E												(two high bytes)
	1	8	3												Complete result
	2	C	9		R	E	T								is in (8308-830F)
	3	0	0		N	O	P								
	4	0	0		N	O	P								
	5				E	N	T	E	R		(	H	L	)	= Binary Fraction
	6				R	E	T	U	R	N	(	H	L	)	= Decimal Fraction
	7				A	L	L		R	E	G	I	S	T	ERS USED
	8														FIGURE D-8a

Decimal Add in Memory for BF2F

INTEGRATED COMPUTER SYSTEMS MICROCOMPUTER TRAINING SYSTEM CODING SHEET

ADDR		CODE														
8 2 E	0															
	1															
	2															
	3															
	4															
8 2 E	5	D	5		P	U	S	H		D						Save binary value
	6	1	E		M	V	I		E	,	0	8				Count 8 bytes
	7	0	8													
	8	A	F		X	R	A		A							Clear CY for first byte
8 2 E	9	0	A		L	D	A	X		B						(A) ← value of bit
	A	8	E		A	D	C		M							Add in decimal
	B	2	7		D	A	A									to bit value or
	C	7	7		M	O	V		M	,	A					result
	D	2	3		I	N	X		H							}
	E	0	3		I	N	X		B							}
	F	1	D		D	C	R		E							} Repeat for all bytes
8 2 F	0	C	2		J	N	Z		8	2	E	9				
	1	E	9													
	2	8	2													
	3	4	4		M	O	V		B	,	H					} Return (BC) = 8300
	4	4	B		M	O	V		C	,	E					} (HL) incr. by 8
	5	D	1		P	O	P		D							Restore binary value
	6	C	9		R	E	T									
	7	0	0		N	O	P									
	8	2	5													Value of LSB
	9	0	6													
	A	8	9													
	B	8	7													
	C	2	5													
	D	1	5													
	E	0	0													
	F	0	0													
8 3 0	0															Scratch pad and
	1															result area
	2															Must be
	3															8300 - 830F
	4															
	5															
	6															
	7															
	8															

FIGURE D-8b

D-5 SUMMARY

The foregoing subroutines occupy one full page (256 bytes) of memory.

8220-825F	Decimal to Binery Integer
8260-827F	Decimal to Binary Fraction
8280-82FF	Binary to Decimal
8300-831F	Command Table, etc. (any 32 bytes)

To perform any useful function with them, you will need the full 1024 bytes of memory in your MTS. If it is equipped with only 512 bytes and you want to pursue development of more complex programs, you should add the additional memory.

Appendix E

CALCULATING TRIGONOMETRIC FUNCTIONS

The sine of an angle (in radians) is calculated from:

(a) $$\sin x = -\frac{x^3}{3!} + \frac{x^5}{4!} - \frac{x^7}{7!} + \text{---}$$

The cosine is generated by a similar series:

(b) $$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \text{---}$$

The exponential function e^x is:

(c) $$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \text{---}$$

All three functions can be generated simultaneously by a procedure that calculates each sucessive term in the series for e^x; adds the terms into a sum for e^x; and adds or subtracts each term to a sine or cosine sum. Each term is calculated from the preceding term, the term number, and the value of x.

$$t_i = xt_{i-1}/i$$

Starting with $t_0 = 1$, this gives:

Term	Value	Disposition
0	1	Enter to cosine
1	x/1	Enter to sine
2	$x^2/2$	Subtract from cosine
3	$x^3/3.2$	Subtract from sine
4	$x^4/4.3.2$	Add to cosine
5	$x^5/5.4.3.2$	Add to sine
6	$x^6/6.5.4.3.2$	Subtract from cosine

The value of x must be expressed in radians, and for reasonably rapid convergence of the series large values of x should be avoided. Since

the sine of an angle is equal to the cosine of its complement:

$$\sin x = \cos\left(\frac{\pi}{2} - x\right)$$

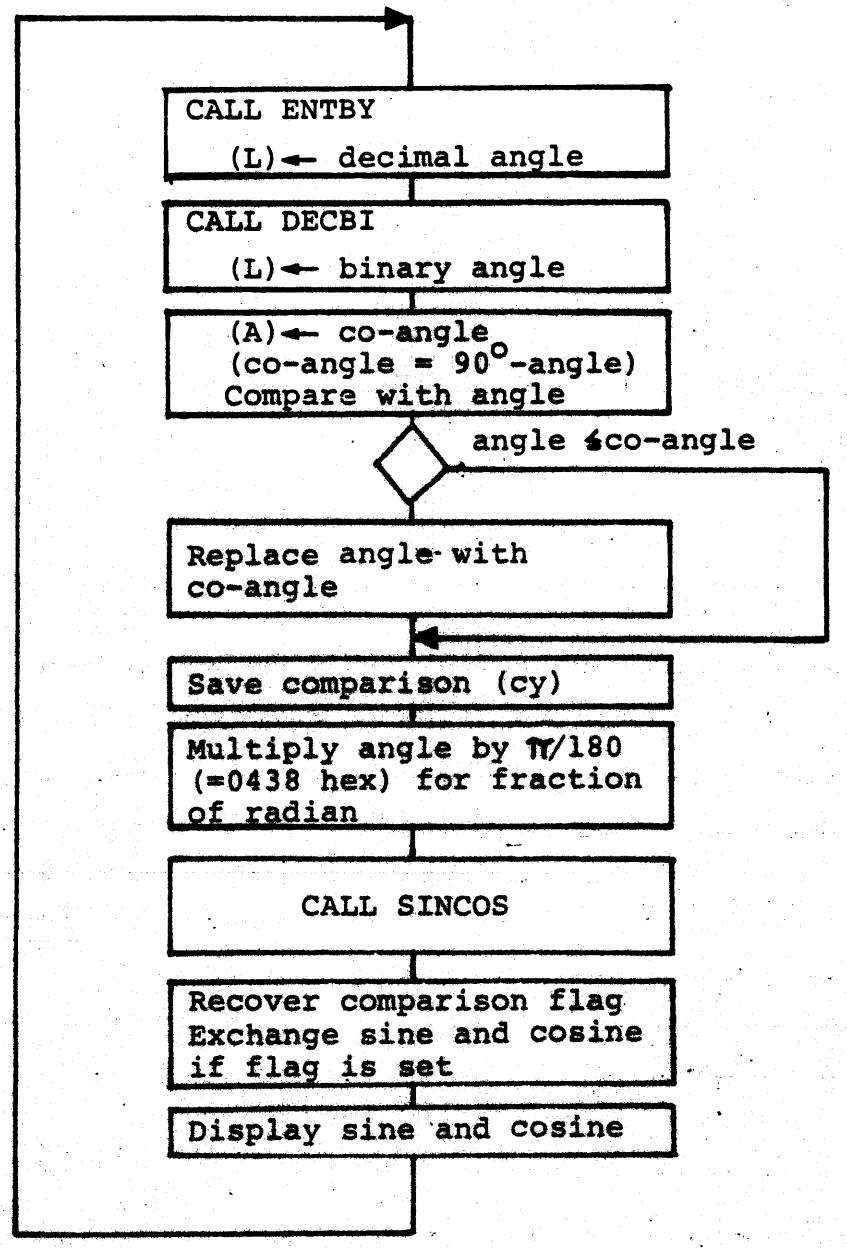
it is easy to restrict the angle to less than 45°, or 0.785 radians. With this limit terms beyond 6 are not needed for 16 bit precision.

In this appendix, we present a subroutine to calculate the sine and cosine, given x as a value between 0 and 0.785 radians. A main program (Figure E-1) will accept an angle in decimal degrees and convert it to binary radians, call SINCOS, and display the results in decimal.

The program also uses a binary multiplication subroutine and a twos complement subroutine, presented in the following pages; the single byte decimal to binary integer conversion of Figure D-5 and the two byte binary fraction to decimal fraction conversion of Figure D-8, in Appendix D. These are also duplicated here.

Memory assignments for the program are:

MAIN	8200-823F
SINCOS	8250-827F
TERM	8280-82AF
DECBI	82B0-82BF
BFDCF	82C0-82FF
Variable Data	8300-830F
BMULT	8310-8330
TWOSC	8336-833F



MAIN PROGRAM

Figure E-1

MAIN - Accept Decimal Angle, Display Sin, Cos

ADDR		CODE															
8 20	0	CD			C	A	L	L		E	N	T	B	Y		(L) ← angle in
	1	36														decimal degrees
	2	03														
	3	CD			C	A	L	L		D	E	C	B	I		(L) ← angle in
	4	B0														binary degrees
	5	82														
	6	3E			M	V	I		A	90						(A) ← 90° (binary)
	7	90														
	8	97			S	U	B		L							(A) ← co-angle
	9	BD			C	M	P		L							Set CY if angle >
	A	F5			P	U	S	H		P	S	W				Save CY
	B	D2			J	N	C		820F							Jump if
	C	0F														angle ≤ co-angle
	D	82														Else replace
	E	6F			M	O	V		L	A						angle with coangle
820	F	26			M	V	I		H	00						Clear high byte
821	0	00														
	1	01			L	X	I		B	0478						Multiply by π/180
	2	78														for angle in radians
	3	04														
	4	CD			C	A	L	L		B	M	U	L	T		(DE) ← angle as
	5	10														binary fraction
	6	83														of one radian
	7	EB			X	C	H	G								
	8	CD			C	A	L	L		S	I	N	C	O	S	(HL) ← sine
	9	50														(DE) ← cosine
	A	82														Recover co-angle
	B	F1			P	O	P		PSW							flag. If set
	C	D2			J	N	C		8220							exchange sine
	D	20														and cosine of
	E	82														co-angle
821	F	EB			X	C	H	G								(HL) ← cosine
8	0															
	1															
	2															
	3															
	4															
	5															
	6															
	7															
	8															

FIGURE E-2a

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

A D D	R	CODE			
8 2 2	0	D 5	PUSH D	Save sine
	1	C D	CALL BFDCF	Convert cosine
	2	C 0		to decimal
	3	8 2		
	4	C D	CALL DWORD	Display at left
	5	D 1		
	6	0 2		
	7	E 1	POP H	
	8	C D	CALL BFDCF	Convert sine
	9	C 0		to decimal
	A	8 2		
	B	1 1	LXI D, 83FF	Display at right
	C	F F		
	D	8 3		
	E	C D	CALL DWORD	
	F	D 4		
8	0	0 2		
	1	C 3	JMP 8200	
	2	0 0		
	3	8 2		
	4			
	5			
	6			
	7			
	8			
	9			
	A			
	B			
	C			
	D			
	E			
	F			
8	0			
	1			
	2			
	3			
	4			
	5			
	6			
	7			
	8			FIGURE E-26

Subroutines SINCOS and TERM are defined in the text below and depicted in Figures E-3 and E-4. SINCOS adds or subtracts successive terms, as discussed early in this appendix. TERM generates the terms, addressing a table of coefficients according to the term number. These coefficients are nominally 1/2, 1/3, 1/4, 1/5, etc. Adjustments to the coefficients for terms 5 and 6 are made as shown in the table of Figure E-5 to correct for rounding errors and absent higher order terms.

The table of Figure E-5 shows the results returned by this program. Note that the adjusted coefficients affect only the least significant digit, for angles between 40 and 50 degrees. The adjustment may be important in some instances, to make sin 45^{o} = cos 45^{o}.

SINCOS Find the sine and cosine of X

Enter with (HL) = X

Return with (BC) = X
(DE) = sin X
(HL) = cos X

Constraints: X must be a fractional value (i.e. less than 1).

The cosine of zero is returned as FFFF.

TERM Find successive terms of e^{x}

Enter with (A) = term number 1 to 8

(BC) = X

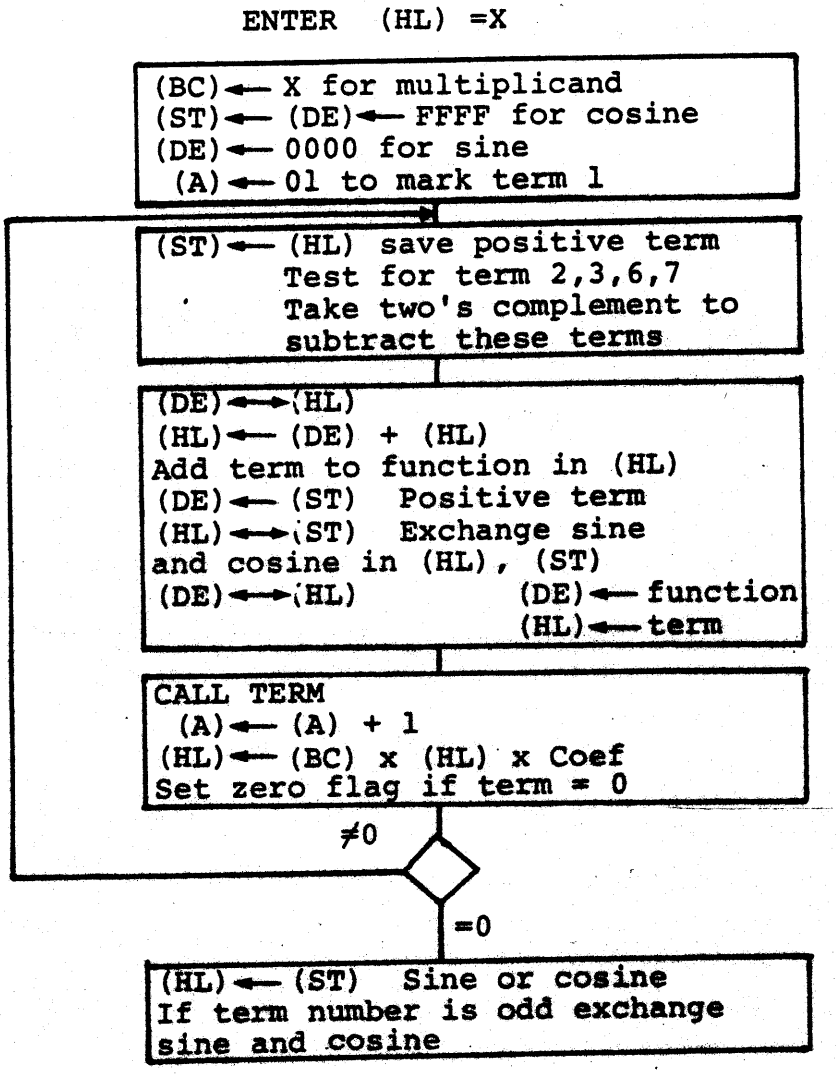
(HL) = previous term

Return with (A) = next term number

(BC) = X

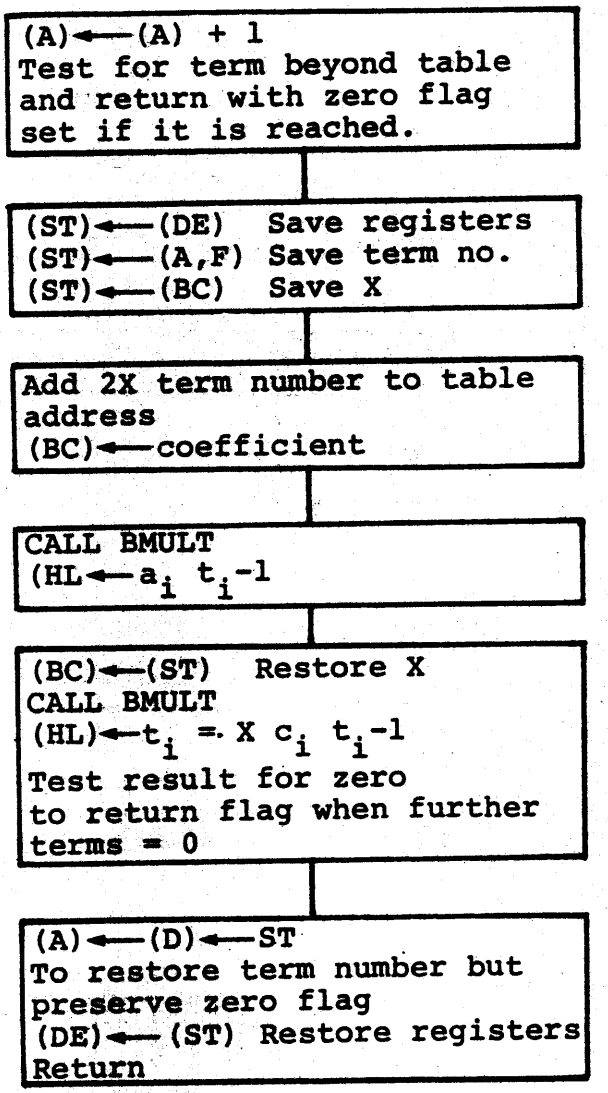
(HL) = new term

Requires a table of values of 1/(A). Term is always positive.



SUBROUTINE SINCOS

Figure E-3



SUBROUTINE TERM

Figure E-4

Coefficients for Successive Terms

Term	Nominal Value		Adjusted	
	Decimal	Hex	Decimal	Hex
a_1	1.0000	-	1.0000	-
a_2	0.5000	8000	0.5000	8000
a_3	0.3333	5555	0.3333	5555
a_4	0.2500	4000	0.2500	4000
a_5	0.2000	3344	0.1953	3200
a_6	0.1667	2AAD	0.0937	1800
a_7	0.1429	2498	0	0
a_8	0.1250	2000	0	0
a_9	0.1111	1C72	0	0

Results of Sine/Cosine Calculation

Angle	Cosine		Sine	
0	0.9999		0.0000	
1	.9998		.0174	
2	.9993		.0349	
3	.9986		.0523	
4	.9975		.0697	
5	.9961		.0871	
10	.9847		.1736	
15	.9658		.2588	
20	.9396	With	.3420	With
25	.9062	adjusted	.4266	adjusted
30	.8659	coefficients	.5000	coefficients
35	.8191		.5736	
40	.7661	*.7660	.6428	
44	.7195	*.7193	.6949	*.6947
45	.7073	*.7071	.7072	*.7071
46	.6949	*.6947	.7195	*.7193
50	.6428		.7661	*.7660
60	.5000		.8659	
75	.2588		.9658	
90	0.0000		0.9999	

*Values with error least significant digit

RESULTS OF SINE/COSINE CALCULATION

Figure E-5

SINCOS — Sine and Cosine of x ($x^E < 1$ radian) (I/12/72)

ADDR	CODE													
8250	4	D		MOV		C	,	L						(BC) ← x
1	4	4		MOV		B	,	H						
2	1	1		LXI		D	,	F	F	F	F			maximum value
3	F	F												for cosine
4	F	F												
5	D	5		PUSH		D								(ST) ← cosine
6	1	3		INX		D								(DE) ← sine = 00
7	3	E		MVI		A	,	0	1					Mark term 1
8	0	1												
8259	E	5		PUSH		H								Save positive term
A	0	F		RRC										Test bit 1 of term
B	0	F		RRC										number.
C	D	C		CC		TWOSC								If term is 2,3,6,7
D	3	6												take twos complement
E	8	3												of term to subtract
F	0	7		RLC										Restore term
8260	0	7		RLC										number
1	E	B		XCHG										(DE) ← term or comp
2	1	9		DAD		D								(HL) ← function
3	D	1		POP		D								(DE) ← positive term
4	E	3		XTHL										Exchange sine/cosine
5	E	B		XCHG										(HL) ← term
6	C	D		CALL		TERM								(HL) ← next term
7	8	0												(A) ← next number
8	8	2												
9	C	2		JNZ		8259								Term sets zero flag
A	5	9												after last term
B	8	2												or if term = 0
C	E	1		POP		H								(HL) ← sine or cosine
D	0	F		RRC										If term number
E	D	8		RC										even, return
F	E	B		XCHG										Else exchange
8270	C	9		RET										sine and cosine
1				ENTER		WITH								
2					(HL) =			x						
3				RETURN										
4					(BC) =			x						
5					(DE) =			SIN						x
6					(HL) =			COS						x
7														
8														FIGURE E-6

INTEGRATED COMPUTER SYSTEMS — MICROCOMPUTER TRAINING SYSTEM — CODING SHEET

TERM - Generate successive terms of e^x E - 11 (1/12/77)

ADDR	CODE	INSTRUCTION	COMMENT
8280	3C	INR A	Next term number
1	FE	CPI 09	Test for past
2	09		last term
3	C8	RZ	Return zero set
4	D5	PUSH D	Save function
5	F5	PUSH PSW	Save term number
6	C5	PUSH B	Save value of x
7	87	ADD A	Double term number
8	11	LXI D, 829C	Address -4 for
9	9C		table of coefficients
A	82		a_0 and a_1 not
B	83	ADD E	stored in table
C	5F	MOV E, A	Address ai
D	1A	LDAX D	
E	4F	MOV C, A	
F	13	INX D	(BC) ← ai
8290	1A	LDAX D	
1	47	MOV B, A	
2	CD	CALL BMULT	(HL) ← ai ti-1
3	10		
4	83		
5	C1	POP B	(BC) ← x
6	CD	CALL BMULT	(HL) ← ti
7	10		= ai ti-1 x
8	83		
9	7C	MOV A, H	Test for
A	B5	ORA L	zero result
B	D1	POP D	(D) ← term number
C	7A	MOV A, D	
D	D1	POP D	(DE) ← function
E	C9	RET	
F	00	NOP	
8 0			
1			
2			
3			
4			
5			
6			
7			
8			FIGURE E-7

COEFFICIENTS FOR TERM

ADDR		CODE				
8 2A	0	0	0			$a_2 = 1/2$
	1	8	0			
	2	5	5			$a_3 = 1/3$
	3	5	5			
	4	0	0			$a_4 = 1/4$
	5	4	0			
	6	4	4	00		$a_5 = 1/5$
	7	3	3	32		
	8	A	D	00		$a_6 = 1/6$
	9	2	A	18		
	A	9	8	00		$a_7 = 1/7$
	B	2	4	00		
	C	0	0	00		$a_8 = 1/8$
	D	2	0	00		
	E	7	2	00		$a_9 = 1/9$
	F	1	C	00		
8	0			↑		
	1			└	ADJUSTED	COEFFICIENTS
	2			↑		
	3			└	EXACT	COEFFICIENTS
	4					
	5					
	6					
	7					
	8					
	9					
	A					
	B					
	C					
	D					
	E					
	F					
8	0					
	1					
	2					
	3					
	4					
	5					
	6					
	7					
	8					

FIGURE E-8

DECBI — Decimal to Binary Integer - one byte

ADDR	CODE			
8 2B 0	F5		PUSH PSW	
1	7D		MOV A,L	
2	E6		ANI 0F	(A) ← low digit
3	0F			
4	67		MOV H,A	(H) ← low digit
5	AD		XRA L	(A) ← high digit
6	1F		RAR	
7	6F		MOV L,A	(L) ← 1/2 high dig.
8	1F		RAR	
9	1F		RAR	(A) ← 1/8 high digit
A	85		ADD L	(A) ← 5/8 high digit
B	84		ADD H	(A) ← binary value
C	6F		MOV L,A	
D	F1		POP PSW	
E	C9		RET	
F	00			
8 0				
1				
2				
3			ENTER WITH	
4			(L) = PACKED DECIMAL BYTE	
5			RETURN WITH	
6			(L) = BINARY INTEGER	
7			REGISTER H USED	
8			OTHER REGISTERS PRESERVED	
9				
A				
B				
C				
D				
E				
F				
8 0				
1				
2				
3				
4				
5				
6				
7				DUPLICATE

BFDCF - Binary Fraction to Decimal Fraction

ADDR		CODE													
8 2C	0	EB		XCHG											
	1	21		LXI		H, 8310									
	2	10													
	3	83													
	4	AF		XRA		A									
	5	2D		DCR		L									
	6	77		MOV		M, A									
	7	C2		JNZ		82C5									
	8	C5													
	9	82													
	A	01		LXI		B, 82F8									
	B	F8													
	C	82													
82C	D	CD		CALL		82E5									
	E	E5													
	F	82													
82D	0	7A		MOV		A, D									
	1	1F		RAR											
	2	57		MOV		D, A									
	3	7B		MOV		A, E									
	4	1F		RAR											
	5	5F		MOV		E, A									
	6	DC		CC		82E5									
	7	E5													
	8	82													
	9	69		MOV		L, C									(HL) ← 8300
	A	7A		MOV		A, D									
	B	B3		ORA		E									
	C	C2		JNZ		82CD									
	D	CD													
	E	82													
	F	2A		LHLD		8300									
82E	0	0E													
	1	83													
	2	C9		RET											
	3	00		NOP											
	4	00		NOP											
	5														
	6														
	7														DUPLICATE
	8														FIGURE D-8a

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

ADDR		CODE														
82E	0															
	1															
	2															
	3															
	4															
82E	5	D	5		P	U	S	H		D						
	6	1	E		M	V	I		E	,	0	8				
	7	0	8													
	8	A	F		X	R	A		A							
82E	9	0	A		L	D	A	X		B						
	A	8	E		A	D	C		M							
	B	2	7		D	A	A									
	C	7	7		M	O	V		M	,	A					
	D	2	3		I	N	X		H							
	E	0	3		I	N	X		B							
	F	1	D		D	C	R		E							
82F	0	C	2		J	N	Z		8	2	E	9				
	1	E	9													
	2	8	2													
	3	4	4		M	O	V		B	,	H					
	4	4	B		M	O	V		C	,	E					
	5	D	1		P	O	P		D							
	6	C	9		R	E	T									
	7	0	0		N	O	P									
	8	2	5													
	9	0	6													
	A	8	9													
	B	8	7													
	C	2	5													
	D	1	5													
	E	0	0													
	F	0	0													
8	0															
	1															
	2															
	3															
	4															
	5															
	6															
	7															DUPLICATE

BMULT (HL) x (BC) → (H, L, D, E) E - 16 (I/12/77)

INTEGRATED COMPUTER SYSTEMS MICROCOMPUTER TRAINING SYSTEM CODING SHEET

ADDR	CODE		Instruction		Comment
8310	E	B	XCHG		(DE) ← multiplier
1	A	F	XRA	A	Clear carry
2	6	F	MOV	L, A	and product
3	6	7	MOV	H, A	
4	E	5	PUSH	H	
5	2	E	MVI	L, 11	(L) ← count
6	1	1			
8317	E	3	XTHL		(HL) ← product
8	D	2	JNC	831C	Jump if multiplier
9	1	C			bit = 0
A	8	3			
B	0	9	DAD	B	Add multiplicand
831C	C	D	CALL	8329	Shift product
D	2	9			Return in (DE)
E	8	3			
F	C	D	CALL	8329	Shift multiplier
8320	2	9			Return in (DE)
1	8	3			
2	E	3	XTHL		
3	2	D	DCR	L	
4	C	2	JNZ	8317	
5	1	7			
6	8	3			
7	E	1	POP	H	(HL) ← high product
8	C	9	RET		
8329	7	C	MOV	A, H	
A	1	F	RAR		
B	6	7	MOV	H, A	
C	7	D	MOV	A, L	
D	1	F	RAR		
E	6	F	MOV	L, A	
F	E	B	XCHG		
8330	C	9	RET		
1					
2			ENTER	WITH	
3			(BC)	= MULT	IPLICAND
4			(HL)	= MULT	IPLIER
5			RETURN		
6			(HL)	= HIGH	PRODUCT
7			(DE)	= LOW	PRODUCT
8			(BC)	PRES	ERVED

TWOSC — Twos complement of (HL) — E - 17 (I/12/77

ADDR	CODE															
8 0																
1																
2																
3																
4																
5																
8336	F	5			P	U	S	H		P	S	W				
7	7	C			M	O	V		A	,	H					
8	2	F			C	M	A									
9	6	7			M	O	V		H	,	A					
A	7	D			M	O	V		A	,	L					
B	2	F			C	M	A									
C	6	F			M	O	V		L	,	A					
D	2	3			I	N	X		H							
E	F	1			P	O	P		P	S	W					
833F	C	9			R	E	T									
8 0																
1																
2					E	N	T	E	R		W	I	T	H		
3						(	H	L	)	=	B	I	N	A	R	Y NUMBER
4					R	E	T	U	R	N						
5						(	H	L	)	=	T	W	O	S		COMPLEMENT
6						O	T	H	E	R	R	E	G	I	S	TERS PRESERVED
7																
8																
9																
A																
B																
C																
D																
E																
F																
8 0																
1																
2																
3																
4																
5																
6																
7																
8																