II

Programming the 8085

Part II of this book is an introduction to assembly language programming for the 8085. It explains commonly used instructions, elementary programming techniques, and their applications.

The content is presented in a format as if for teaching a foreign language. One approach to learning a foreign language is to begin with a few words that can form simple, meaningful, and interactive sentences. After learning a few sentences, the student begins to write a paragraph that can convey an idea in a coherent fashion; then, by sequencing a few paragraphs, begins to compose a letter. Chapters 6 to 11 are arranged in similar fashion—from simple instructions to applications.

Chapter 2 provided an overview of the 8085 instruction set. Chapters 6 and 7 are concerned primarily with the instructions that occur most frequently. The instructions are not introduced according to the five groups as classified in Chapter 2; instead, a few instructions that can perform a simple task are selected from each group. Chapter 6 includes the discussion of instructions from each group—from data copy to branch instructions. Chapter 7 introduces elementary programming techniques such as looping and indexing. Chapter 8 uses the instructions and techniques presented in these chapters to design software delays and counters.

Chapter 9 introduces the concepts of subroutine and stack, which provide flexibility and variety for program design. Chapter 10 includes applications of the concepts presented in Chapter 9, presenting techniques for writing programs concerned with code conversions, and arithmetic routines. Chapter 11 deals with the uses of assemblers and software development systems.

PREREQUISITES

The reader is expected to know the following topics:

- ☐ The 8085 architecture, especially the programming registers.
- ☐ The concepts related to memory and I/Os.
- ☐ Logic operations and binary and hexadecimal arithmetic.

6

Introduction to 8085 Instructions

A microcomputer performs a task by reading and executing the set of instructions written in its memory. This set of instructions, written in a sequence, is called a **program.** Each instruction in the program is a command, in binary, to the microprocessor to perform an operation. This chapter introduces 8085 basic instructions, their operations, and their applications.

Chapters 3 and 4 described the architecture of the 8085 microprocessor and Chapter 2 provided an overview of the instruction set and the tasks the 8085 can perform. This chapter is concerned with using instructions within the constraints and capabilities of its registers and the bus system. A few instructions are introduced from each of the five groups (Data Transfer, Arithmetic, Logical, Branch, and Machine Control) and are used to write simple programs to perform specific tasks.

The simple illustrative programs given in this chapter can be entered and executed on the single-board microcomputers used commonly in college laboratories.

OBJECTIVES

- ☐ Explain the functions of data transfer (copy) instructions and how the contents of the source register and the destination register are affected.
- ☐ Explain the Input/Output instructions and port addresses.
- ☐ Explain the functions of the machine control instructions HLT and NOP.
- ☐ Recognize the addressing modes of the instructions.
- ☐ Draw a flowchart of a simple program.
- ☐ Write a program in 8085 mnemonics to illustrate an application of data copy instructions, and translate those mnemonics manually into their Hex codes.

- ☐ Write a program in the proper format showing memory addresses, Hex machine codes, mnemonics, and comments.
- ☐ Explain the arithmetic instructions, and recognize the flags that are set or reset for given data conditions.
- ☐ Write a set of instructions to perform an addition and a subtraction (in 2's complement).
- ☐ Explain the logic instructions, and recognize the flags that are set or reset for given data conditions.
- ☐ Write a set of instructions to illustrate logic operations.
- ☐ Explain the use of logic instructions in masking, setting, and resetting individual bits.
- ☐ Explain the unconditional and conditional Jump instructions and how flags are used by the conditional Jump instructions to change the sequence of a program.
- ☐ Write a program to illustrate an application of Jump instructions.
- ☐ List the important steps in writing and troubleshooting a simple program.

6.1 DATA TRANSFER (COPY) OPERATIONS

One of the primary functions of the microprocessor is copying data, from a register (or I/O or memory) called the source, to another register (or I/O or memory) called the destination. In technical literature, the copying function is frequently labeled as the **data transfer function,** which is somewhat misleading. In fact, the contents of the source are not transferred, but are copied into the destination register without modifying the contents of the source.

Several instructions are used to copy data (as listed in Chapter 2). This section is concerned with the following operations.

MOV : Move	Copy a data byte.
MVI : Move Immediate	Load a data byte directly.
OUT : Output to Port	Send a data byte to an output device.
IN : Input from Port	Read a data byte from an input device.

The term *copy* is equally valid for input/output functions because the contents of the source are not altered. However, the term *data transfer* is used so commonly to indicate the data copy function that, in this book, these terms are used interchangeably when the meaning is not ambiguous.

In addition to data copy instructions, it is necessary to introduce two machine-control operations to execute programs.

HLT: Halt	Stop processing and wait.
NOP: No Operation	Do not perform any operation.

These operations (opcodes) are explained and illustrated below with examples.

Instructions The data transfer instructions copy data from a source into a destination without modifying the contents of the source. The previous contents of the destination are replaced by the contents of the source.

Important Note: In the 8085 processor, data transfer instructions do not affect the flags.

Opcode	Operand	Description
MOV	Rd,Rs*	Move ☐ This is a 1-byte instruction ☐ Copies data from source register Rs to destination register Rd
MVI	R,8-bit*	Move Immediate ☐ This is a 2-byte instruction ☐ Loads the 8 bits of the second byte into the register specified
OUT	8-bit port address	Output to Port ☐ This is a 2-byte instruction ☐ Sends (copies) the contents of the accumulator (A) to the output port specified in the second byte
IN	8-bit port address	Input from Port ☐ This is a 2-byte instruction ☐ Accepts (reads) data from the input port specified in the second byte, and loads into the accumulator
HLT		Halt ☐ This is a 1-byte instruction ☐ The processor stops executing and enters wait state ☐ The address bus and data bus are placed in high impedance state. No register contents are affected
NOP		No Operation ☐ This is a 1-byte instruction ☐ No operation is performed ☐ Generally used to increase processing time or substitute in place of an instruction. When an error occurs in a program and an instruction needs to be eliminated, it is more convenient to substitute NOP than to reassemble the whole program

*The symbols Rd, Rs, and R are generic terms; they represent any of the 8085 8-bit registers: A, B, C, D, E, H, and L.

Example 6.1 Load the accumulator A with the data byte 82H (the letter H indicates hexadecimal number), and save the data in register B.

Instructions

```
MVI A, 82H,
MOV B,A
```

The first instruction is a 2-byte instruction that loads the accumulator with the data byte 82H, and the second instruction MOV B,A copies the contents of the accumulator in register B without changing the contents of the accumulator.

Example 6.2 Write instructions to read eight ON/OFF switches connected to the input port with the address 00H, and turn on the devices connected to the output port with the address 01H, as shown in Figure 6.1. (I/O port addresses are given in hexadecimal.)

Solution The input has eight switches that are connected to the data bus through the tri-state buffer. Any one of the switches can be connected to +5 V (logic 1) or to ground (logic 0), and each switch controls the corresponding device at the output port. The microprocessor needs to read the bit pattern on the switches and send the same bit pattern to the output port to turn on the corresponding devices.

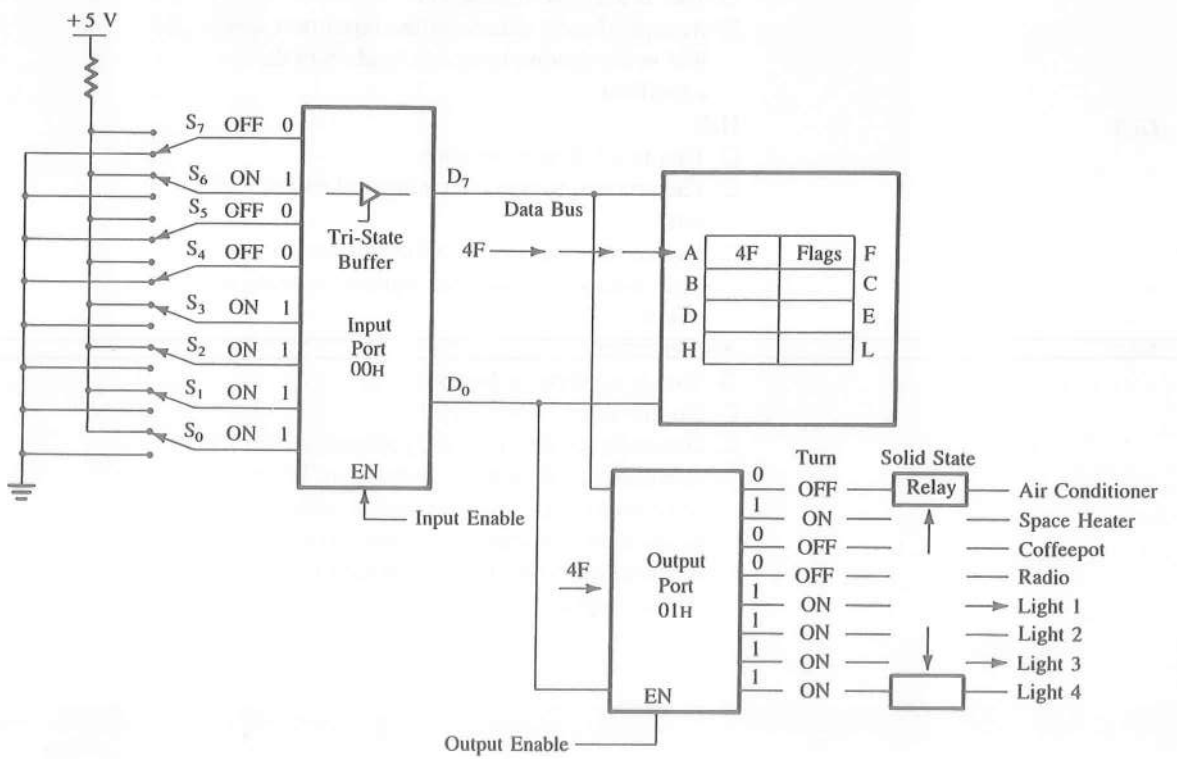


FIGURE 6.1
Reading Data at Input Port and Sending Data to Output Port

Instructions
```
IN    00H
OUT   01H
HLT
```

When the microprocessor executes the instruction IN 00H, it enables the tri-state buffer. The bit pattern 4FH formed by the switch positions is placed on the data bus and transferred to the accumulator. This is called reading an input port.

When the microprocessor executes the next instruction, OUT 01H, it places the contents of the accumulator on the data bus and enables the output port 01H. (This is also called writing data to an output port.) The output port latches the bit pattern and turns ON/OFF the devices connected to the port according to the bit pattern. In Figure 6.1, the bit pattern 4FH will turn on the devices connected to the output port data lines D_6, D_3, D_2, D_1, and D_0: the space heater and four light bulbs. To turn off some of the devices and turn on other devices, the bit pattern can be modified by changing the switch positions. For example, to turn on the radio and the coffeepot and turn off all other devices, the switches S_4 and S_5 should be on and the others should be off. The microprocessor will read the bit pattern 0011 0000, and this bit pattern will turn on the radio and the coffeepot and turn off other devices.

The preceding explanation raises two questions:

1. What are the second bytes in the instructions IN and OUT?
2. How are they determined?

In answer to the first question, the second bytes are I/O port addresses. Each I/O port is identified with a number or an address similar to the postal address of a house. The second byte has eight bits, meaning 256 (2^8) combinations; thus 256 input ports and 256 output ports with addresses from 00H to FFH can be connected to the system.

The answer to the second question depends on the logic circuit (called interfacing) used to connect and identify a port by the system designer (see Chapter 5).

6.1.1 Addressing Modes

The above instructions are commands to the microprocessor to copy 8-bit data from a source into a destination. In these instructions, the source can be a register, an input port, or an 8-bit number (00H to FFH). Similarly, a destination can be a register or an output port. The sources and destination are, in fact, operands. The various formats of specifying the operands are called the addressing modes. The 8085 instruction set has the following addressing modes. (Each mode is followed by an example and by the corresponding piece of restaurant conversation from the analogy discussed in Chapter 2.)

1. Immediate Addressing—MVI R,Data (Pass the butter)
2. Register Addressing—MOV Rd,Rs (Pass the bowl)
3. Direct Addressing—IN/OUT Port# (Combination number 17 on the menu)
4. Indirect Addressing—Illustrated in the next chapter (I will have what Susie has)

The classification of the addressing modes is unimportant, except that it provides some clues in understanding mnemonics. For example, in the case of the MVI opcode, the letter I suggests that the second byte is data and not a register. What is important is to become familiar with the instructions. After you study the examples given in this chapter, you will see a pattern begin to emerge.

6.1.2 Illustrative Program: Data Transfer—From Register to Output Port

PROBLEM STATEMENT

Load the hexadecimal number 37H in register B, and display the number at the output port labeled PORT1.

PROBLEM ANALYSIS

This problem is similar to the illustrative program discussed in Section 2.4.1. Even though this is a very simple problem it is necessary to break the problem into small steps and to outline the thinking process in terms of the tasks described in Section 6.1.

STEPS

Step 1: Load register B with a number.
Step 2: Send the number to the output port.

QUESTIONS TO BE ASKED

- ☐ Is there an instruction to load the register B? YES—MVI B.
- ☐ Is there an instruction to send the data from register B to the output port? NO. Review the instruction OUT. This instruction sends data from the accumulator to an output port.
- ☐ The solution appears to be as follows: Copy the number from register B into accumulator A.
- ☐ Is there an instruction to copy data from one register to another register? YES—MOV Rd,Rs.

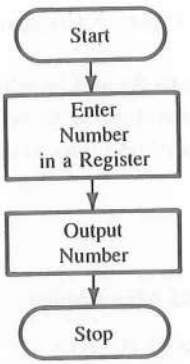
FLOWCHART

The thinking process described here and the steps necessary to write the program can be represented in a pictorial format, called a **flowchart.** Figure 6.2 describes the preceding steps in a flowchart.

Flowcharting is an art. The flowchart in Figure 6.2 does not include all the steps described earlier. Although the number of steps that should be represented in a flowchart is ambiguous, not all of them should be included. That would defeat the purpose of the flowchart. It should represent a logical approach and sequence of steps in solving the problem. A flowchart is similar to the block diagram of a hardware system or to the outline of a chapter. Information in each block of the flowchart should be similar to the heading of a paragraph. Generally, a flowchart is used for two purposes: to assist and clarify the thinking process and to communicate the programmer's thoughts or logic to others.

Symbols commonly used in flowcharting are shown in Figure 6.3. Two types of symbols—rectangles and ovals—are already illustrated in Figure 6.2. The diamond is

FIGURE 6.2
Flowchart



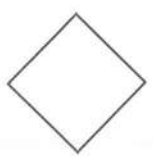
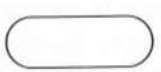
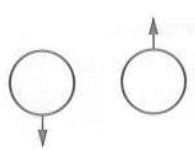
	Meaning
	Arrow: Indicates the direction of the program execution
	Rectangle: Represents a process or an operation
	Diamond: Represents a decision-making block
	Oval: Indicates the beginning or end of a program
	Double-sided rectangle: Represents a predefined process such as a subroutine.
	Circle with an arrow: Represents continuation (an entry or exit) to a different page

FIGURE 6.3
Flowcharting Symbols

used with Jump instructions for decision making (see Figure 6.10), and the double-sided rectangle is used for subroutines (see Chapter 9).

The flowchart in Figure 6.2 includes what steps to do and in what sequence. As a rule, a general flowchart does not include how to perform these steps or what registers are being used. The steps described in the flowchart are translated into an assembly language program in the next section.

ASSEMBLY LANGUAGE PROGRAM

Tasks	8085 Mnemonics
1. Load register B with 37H.	MVI B,37H*
2. Copy the number from B to A.	MOV A,B
3. Send the number to the output—port 01H.	OUT PORT1
4. End of the program.	HLT

TRANSLATION FROM ASSEMBLY LANGUAGE TO MACHINE LANGUAGE

Now, to translate the assembly language program into machine language, look up the hexadecimal machine codes for each instruction in the 8085 instruction set and write each machine code in the sequence, as follows:

8085 Mnemonics	Hex Machine Code
1. MVI B,37H	06
	37
2. MOV A,B	78
3. OUT PORT1	D3
	01
4. HLT	76

This program has six machine codes and will require six bytes of memory to enter the program into your system. If your single-board microcomputer has R/W memory starting at the address 2000H, this program can be entered in the memory locations 2000H to 2005H. The format generally used to write an assembly language program is shown below.

PROGRAM FORMAT

Memory Address (Hex)	Machine Code (Hex)	Instruction		
		Opcode	Operand	Comments
XX00†	06	MVI	B,37H	;Load register B with data 37H
XX01	37			

*A number followed by the letter H represents a hexadecimal number.

†Enter high-order address (page number) of your R/W memory in place of XX.

```
XX02    78          MOV     A,B       ;Copy (B) into (A)
XX03    D3          OUT     PORT1     ;Display accumulator contents
XX04    PORT1*                        ;  (37H) at Port1
XX05    76          HLT               ;End of the program
```

This program has five columns: Memory Address, Machine Code, Opcode, Operand, and Comments. Each is described in the context of a single-board microcomputer.

Memory Address These are 16-bit addresses of the user (R/W) memory in the system, where the machine code of the program is stored. The beginning address is shown as XX00; the symbol XX represents the page number of the available R/W memory in the microcomputer, and 00 represents the line number. For example, if the microcomputer has the user memory at 2000H, the symbol XX represents page number 20H; if the user memory begins at 0300H, the symbol XX represents page 03H. Substitute the appropriate page when entering the machine code of a program.

Machine Code These are the hexadecimal numbers (instruction codes) that are entered (or stored) in the respective memory addresses through the hexadecimal keyboard of the microcomputer. The monitor program, which is stored in Read-Only memory (ROM) of the microcomputer, translates the Hex numbers into binary digits and stores the binary digits in the R/W memory.

If the system has R/W memory with the starting address at 2000H and the output port address 01H, the program will be stored as follows:

Memory Address	Memory Contents		Hex Code
2000	0 0 0 0 0 1 1 0	→	06
2001	0 0 1 1 0 1 1 1	→	37
2002	0 1 1 1 1 0 0 0	→	78
2003	1 1 0 1 0 0 1 1	→	D3
2004	0 0 0 0 0 0 0 1	→	01
2005	0 1 1 1 0 1 1 0	→	76

Opcode (Operation Code) An instruction is divided into two parts: Opcode and Operand. Opcodes are the abbreviated symbols specified by the manufacturer (Intel) to indicate the type of operation or function that will be performed by the machine code.

Operand The operand part of an instruction specifies the item to be processed; it can be 8- or 16-bit data, a register, or a memory address.

---

*Enter the output port address of your system. If an output port is not available on your system, see "How to Execute a Program without an Output Port" later in this section.

An instruction, called a mnemonic or mnemonic instruction, is formed by combining an opcode and an operand. The mnemonics are used to write programs in the 8085 assembly language; and then the mnemonics in these programs are translated manually into the binary machine code by looking them up in the instruction set.

Comments The comments are written as a part of the proper documentation of a program to explain or elaborate the purpose of the instructions used. These are separated by a semicolon (;) from the instruction on the same line. They play a critical role in the user's understanding of the logic behind a program. Because the illustrative programs in the early part of this chapter are simple, most of the comments are either redundant or trivial. The purpose of the comments in these programs is to reinforce the meaning of the instructions. In actual usage, the comments should not just describe the operation of an instruction.

HOW TO ENTER AND EXECUTE THE PROGRAM

This program assumes that one output port is available on your microcomputer system. The program cannot be executed without modification if your microcomputer has no independent output ports other than the system display of memory address and data or if it has programmable I/O ports. (See Chapter 14.) To enter the program:*

1. Push the Reset key.
2. Enter the 16-bit memory address of the first machine code of your program. (Substitute the page number of your R/W memory for the letters XX and the output port address for the label PORT1.)
3. Enter and store all the machine codes sequentially, using the hexadecimal keyboard on your system.
4. Reset the system.
5. Enter the memory address where the program begins and push the Execute key.

If the program is properly entered and executed, the data byte 37H will be displayed at the output port.

HOW TO EXECUTE A PROGRAM WITHOUT AN OUTPUT PORT

If your system does not have an output port, either eliminate the instruction OUT PORT1, or substitute NOP (No Operation) in place of the OUT instruction. Assuming your system has R/W memory starting at 2000H, you can enter the program as follows:

Memory Address	Machine Code	Mnemonic Instruction
2000	06	MVI B,37H
2001	37	
2002	78	MOV A,B
2003	00	NOP

*Refer to the user's manual of your microcomputer for details.

2004	00	NOP
2005	76	HLT

After you have executed this program, you can find the answer in the accumulator by pushing the Examine Register key (see your user's manual).

The program also can be executed by entering the machine code 76 in location 2003H, thus eliminating the OUT instruction.

6.1.3 Illustrative Program: Data Transfer to Control Output Devices

PROBLEM STATEMENT

A microcomputer is designed to control various appliances and lights in your house. The system has an output port with the address 01H, and various units are connected to the bits D_7 to D_0 as shown in Figure 6.4. On a cool morning you want to turn on the radio, the coffeepot, and the space heater. Write appropriate instructions for the microcomputer. Assume the R/W memory in your system begins at 2000H.

PROBLEM ANALYSIS

The output port in Figure 6.4 is a latch (D flip-flop). When data bits are sent to the output port they are latched by the D flip-flop. A data bit at logic 1 supplies approximately 5 V as output and can turn on solid-state relays.

To turn on the radio, the coffeepot, and the space heater, set D_6, D_5, and D_4 at logic 1, and the other bits at logic 0:

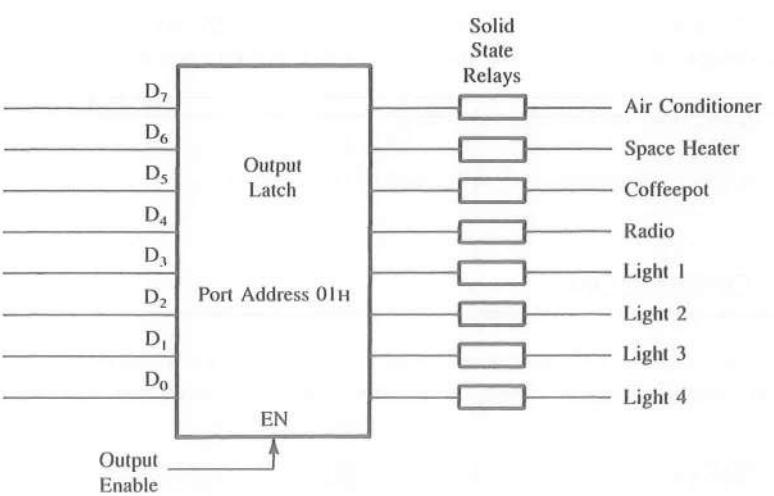


FIGURE 6.4
Output Port to Control Devices

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	
0	1	1	1	0	0	0	0	= 70H

The output port requires 70H, and it can be sent to the port by loading the accumulator with 70H.

PROGRAM

Memory Address	Machine Code	Mnemonic Instruction	Comments
HI-LO*			
2000	3E	MVI A,70H	;Load the accumulator with the bit pattern
2001	70		; necessary to turn on the devices
2002	D3	OUT 01H	;Send the bit pattern to the port 01H, and
2003	01†		; turn on the devices
2004	76	HLT	;End of the program

PROGRAM OUTPUT

This program simulates controlling of the devices connected to the output port by displaying 70H on a seven-segment LED display. If your system has individual LEDs, the binary pattern—0111 0000—will be displayed.

6.1.4 Review of Important Concepts

1. Registers are used to load data directly or to save data bytes.
2. In data transfer (copying), the destination register is modified but the source register retains its data.
3. The 8085 transfers data from an input port to the accumulator (IN) and from the accumulator to an output port (OUT). The instruction OUT cannot send data from any other register.
4. The data copy instructions do not affect the flags.

See Questions and Assignments 1–7 at the end of this chapter.

6.2 ARITHMETIC OPERATIONS

The 8085 microprocessor performs various arithmetic operations, such as addition, subtraction, increment, and decrement. These arithmetic operations have the following mnemonics.

*Change the high-order memory address 20 to the appropriate address for your system.
†Substitute the appropriate port address.

ADD : Add	Add the contents of a register.*
ADI : Add Immediate	Add 8-bit data.
SUB : Subtract	Subtract the contents of a register.
SUI : Subtract Immediate	Subtract 8-bit data.
INR : Increment	Increase the contents of a register by 1.
DCR : Decrement	Decrease the contents of a register by 1.

The arithmetic operations Add and Subtract are performed in relation to the contents of the accumulator. However, the Increment or the Decrement operations can be performed in any register. The instructions for these operations are explained below.

INSTRUCTIONS

These arithmetic instructions (except INR and DCR)

1. assume implicitly that the accumulator is one of the operands.
2. modify all the flags according to the data conditions of the result.
3. place the result in the accumulator.
4. do not affect the contents of the operand register.

The instructions INR and DCR

1. affect the contents of the specified register.
2. affect all flags except the CY flag.

The descriptions of the instructions (including INR and DCR) are as follows:

Opcode	Operand	Description
ADD	R†	Add ☐ This is a 1-byte instruction ☐ Adds the contents of register R to the contents of the accumulator
ADI	8-bit	Add Immediate ☐ This is a 2-byte instruction ☐ Adds the second byte to the contents of the accumulator
SUB	R†	Subtract ☐ This is a 1-byte instruction ☐ Subtracts the contents of register R from the contents of the accumulator
SUI	8-bit	Subtract Immediate ☐ This is a 2-byte instruction

*Memory-related arithmetic operations are excluded here; they are discussed in Chapter 7.
†R represents any of registers A, B, C, D, E, H, and L.

- ☐ Subtracts the second byte from the contents of the accumulator

INR	R*	Increment ☐ This is a 1-byte instruction ☐ Increases the contents of register R by 1 *Caution:* All flags except the CY are affected
DCR	R*	Decrement ☐ This is a 1-byte instruction ☐ Decreases the contents of register R by 1 *Caution:* All flags except the CY are affected

6.2.1 Addition

The 8085 performs addition with 8-bit binary numbers and stores the sum in the accumulator. If the sum is larger than eight bits (FFH), it sets the Carry flag. Addition can be performed either by adding the contents of a source register (B, C, D, E, H, L, or memory) to the contents of the accumulator (ADD) or by adding the second byte directly to the contents of the accumulator (ADI).

Example 6.3

The contents of the accumulator are 93H and the contents of register C are B7H. Add both contents.

Instruction ADD C

		CY	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	
(A) : 93H =			1	0	0	1	0	0	1	1	
+											
(C) : B7H =			1	0	1	1	0	1	1	1	
		1		1	1		1	1	1		Carry
SUM (A) :	[1] 4AH = [2]		0	1	0	0	1	0	1	0	
	CY										

Flag Status:† S = 0, Z = 0, CY = 1

When the 8085 adds 93H and B7H, the sum is 14AH; it is larger than eight bits. Therefore, the accumulator will have 4AH in binary, and the CY flag will be set. The result in the accumulator (4AH) is not 0, and bit D_7 is not 1; therefore, the Zero and the Sign flags will be reset.

---

*R represents any of registers A, B, C, D, E, H, and L.

†The P and AC flags are not shown here. In this chapter, the focus will be on the Sign, Zero, and Carry flags.

Example 6.4

Add the number 35H directly to the sum in the previous example when the CY flag is set.

Instruction ADI 35H

```
                    CY
(A)    :  4AH = [1]  0 1 0 0   1 0 1 0
          +
(Data) :  35H =      0 0 1 1   0 1 0 1
(A)    :  7FH = [0]  0 1 1 1   1 1 1 1
```

Flag Status: S = 0, Z = 0, CY = 0

The addition of 4AH and 35H does not generate a carry and will reset the previous Carry flag. Therefore, in adding numbers, it is necessary to count how many times the CY flag is set by using some other programming techniques (see Section 7.3.2).

Example 6.5

Assume the accumulator holds the data byte FFH. Illustrate the differences in the flags set by adding 01H and by incrementing the accumulator contents.

Instruction ADI 01H

```
                    CY
(A)    :   FFH =     1 1 1 1   1 1 1 1
          +
(Data)  :  01H =     0 0 0 0   0 0 0 1
                     1 1 1 1   1 1 1 1   Carry
                    ~~~~~~~~~~~~~~~~~~~~
(A)    : [1] 00H=  [1] 1 0 0 0   0 0 0 0
          CY
```

Flag Status: S = 0, Z = 1, CY = 1

After adding 01H to FFH, the sum in the accumulator is 0 with a carry. Therefore, the CY and Z flags are set. The Sign flag is reset because D_7 is 0.

Instruction INR A

The accumulator contents will be 00H, the same as before. However, the instruction INR will not affect the Carry flag; it will remain in its previous status.

Flag Status: S = 0, Z = 1, CY = NA

FLAG CONCEPTS AND CAUTIONS

As described in the previous chapter, the flags are flip-flops that are set or reset after the execution of arithmetic and logic operations, with some exceptions. In many ways, the flags are like signs on an interstate highway that help drivers find their destinations.

Drivers may see one or more signs at a time. They may take the exit when they find the sign they are looking for, or they may continue along the interstate and ignore the signs.

Similarly, flags are signs of data conditions. After an operation, one or more flags may be set, and they can be used to change the direction of the program sequence by using Jump instructions, which will be described later. However, the programmer should be alert for them to make a decision. If the flags are not appropriate for the tasks, the programmer can ignore them.

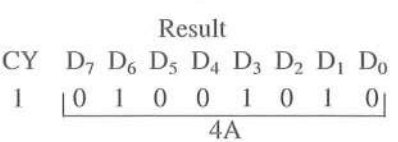
Caution #1 In Example 6.3, the CY flag is set, and in Example 6.4, the CY flag is reset. The critical concept here is that if the programmer ignores the flag, it can be lost after the subsequent instructions. However, the flag can be ignored when the programmer is not interested in using it.

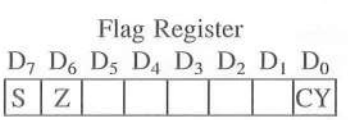
Caution #2 In Example 6.5, two flags are set. The programmer may use one or more flags to make decisions or may ignore them if they are irrelevant.

Caution #3 The CY flag has a dual function; it is used as a carry in addition and as a borrow in subtraction.

The importance of flags cannot be emphasized enough, and a thorough understanding of them is critical in writing assembly language programs.

1. Flags are flip-flops in the ALU (arithmetic/logic unit). They are affected (set or reset) by the operations in the ALU; therefore, operations, such as copy, that take place outside the ALU do not affect the flags.
2. The status of the flags is determined by the result of an operation. In most instances, the result is in the accumulator. However, in some operations, such as Increment (INR), results can be in registers other than the accumulator.
3. There is no relationship between a result and the bit positions of the flag register. In Example 6.3, the answer of the addition is 4AH with a carry. The binary answer is as follows:





Carry Flag Set to 1 because the answer is larger than eight bits; there is a carry generated out of the last bit D_7. During the addition, bits D_0 through D_6 may generate carries, but these carries do not affect the CY flag.

Misconception #1 Bit D_0 in the result (4AH) corresponds to the bit position of the Carry flag D_0 in the flag register; therefore, the Carry flag is reset.

Misconception #2 In the addition process, bits D_0 of 93H and B7H generate a carry (or other bit additions generate carries); therefore the Carry flag is set.

Zero Flag Reset to 0 because the answer is not zero. The Zero flag is set only when all eight bits in the result are 0.

Misconception #3 Bit D_6 in the result (4AH) is 1, and it corresponds to bit D_6 (Zero flag position) in the flag register. Therefore, the Z flag is set.

Sign Flag Reset to 0 because D_7 in the result is 0. The position of the sign flag in the flag register is also D_7. But it is just a coincidence. The microprocessor designer could have chosen bit D_6 for the Sign flag and bit D_7 for the Zero flag in the flag register. The Sign flag is relevant only when we are using signed numbers.

Misconception #4 If the Sign flag is set, the result must be negative.

See Questions and Programming Assignments 9 through 19 at the end of this chapter.

6.2.2 Illustrative Program: Arithmetic Operations—Addition and Increment

PROBLEM STATEMENT

Write a program to perform the following functions, and verify the output.

1. Load the number 8BH in register D.
2. Load the number 6FH in register C.
3. Increment the contents of register C by one.
4. Add the contents of registers C and D and display the sum at the output PORT1.

PROGRAM

The illustrative program for arithmetic operations using addition and increment is presented as Figure 6.5 to show the register contents during some of the steps.

PROGRAM DESCRIPTION

1. The first four machine codes load 8BH in register D and 6FH in register C (see Figure 6.5). These are Data Copy instructions. Therefore, no flags will be affected; the flags will remain in their previous status. The status of the flags is shown as X to indicate no change in their status.
2. Instruction INR C adds 1 to 6FH and changes the contents of C to 70H. The result is nonzero and bit D_7 is zero; therefore, the S and Z flags are reset. However, the CY flag is not affected by the INR instruction.
3. To add (C) to (D), the contents of one of the registers must be transferred to the accumulator because the 8085 cannot add two registers directly. Review the ADD instruction. The instruction MOV A,C copies 70H from register C into the accumulator without affecting (C). See the register contents.

Memory Address (H)	Machine Code	Instruction Opcode	Operand	Comments and Register Contents
				The first four machine codes load the registers as
HI-LO XX00	16	MVI	D,8BH	
01	8B			
02	0E	MVI	C,6FH	
03	6F			

		S Z CY	
A		X X X	F
B		6F	C
D	8B		E
H			L

04	0C	INR	C

Add 01 to (C): 6F + 01 = 70H

		S Z CY	
A	70	0 0 X	F
B		70	C
D	8B		E

05	79	MOV	A,C

06	82	ADD	D
07	D3	OUT	PORT1
08	PORT #	PORT1	
09	76	HLT	

		S Z CY	
A	FB	1 0 0	F
B		70	C
D	8B		E

End of the program

FIGURE 6.5
Illustrative Program for Arithmetic Operations—Using Addition and Increment

4. Instruction ADD D adds (D) to (A), stores the sum in A, and sets the Sign flag as shown below:

```
(A) : 70H =      0 1 1 1  0 0 0 0
      +
(D) : 8BH =      1 0 0 0  1 0 1 1
      ------------------------------
(A) : FBH = [0]  1 1 1 1  1 0 1 1   (see Figure 6.5)
            CY
```

Flag Status: S = 1, Z = 0, CY = 0

5. The sum FBH is displayed by the OUT instruction.

PROGRAM OUTPUT

This program will display FBH at the output port. If an output port is not available, the program can be executed by entering NOP instructions in place of the OUT instruction and the answer FBH can be verified by examining the accumulator A. (Most systems have the Examine-Register operation.) Similarly, the contents of registers C and D and the flags can be verified.

By examining the contents of the registers, the following points can be confirmed:

1. The sum is stored in the accumulator.
2. The contents of the source registers are not changed.
3. The Sign (S) flag is set.

Even though the Sign (S) flag is set, this is not a negative sum. The microprocessor sets the Sign flag whenever an operation results in $D_7 = 1$. The microprocessor cannot recognize whether FBH is a sum, a negative number, or a bit pattern. It is your responsibility to interpret and use the flags. (See "Flag Concepts and Cautions" in Section 6.2.1.) In this example, the addition is not concerned with the signed numbers. With the signed numbers, bit D_7 is reserved for a sign by the programmer (not by the microprocessor), and no number larger than $+127_{10}$ can be entered.

6.2.3 Subtraction

The 8085 performs subtraction by using the method of 2's complement. (If you are not familiar with the method of 2's complement, review Appendix A2.)

Subtraction can be performed by using either the instruction SUB to subtract the contents of a source register or the instruction SUI to subtract an 8-bit number from the contents of the accumulator. In either case, the accumulator contents are regarded as the minuend (the number from which to subtract).

The 8085 performs the following steps internally to execute the instruction SUB (or SUI).

Step 1: Converts subtrahend (the number to be subtracted) into its 1's complement.
Step 2: Adds 1 to 1's complement to obtain 2's complement of the subtrahend.
Step 3: Add 2's complement to the minuend (the contents of the accumulator).
Step 4: Complements the Carry flag.

These steps are illustrated in the following example.

Example 6.6

Register B has 65H and the accumulator has 97H. Subtract the contents of register B from the contents of the accumulator.

Instruction SUB B

	Subtrahend (B): 65H =	0 1 1 0 0 1 0 1
Step 1:	1's complement of 65H =	1 0 0 1 1 0 1 0
	(Substitute 0 for 1 and 1 for 0)	
		+
Step 2:	Add 01 to obtain	0 0 0 0 0 0 0 1
	2's complement of 65H =	1 0 0 1 1 0 1 1
		+

To subtract: 97H – 65H,
Add 97H to 2's complement of 65H = 1 0 0 1 0 1 1 1

1 1 1 1 1 Carry

Step 3: CY [1] 0 0 1 1 0 0 1 0
Step 4: Complement Carry [0] 0 0 1 1 0 0 1 0
Result (A): 32H

Flag Status: S = 0, Z = 0, CY = 0

If the answer is negative, it will be shown in the 2's complement of the actual magnitude. For example, if the above subtraction is performed as 65H – 97H, the answer will be the 2's complement of 32H with the Carry (Borrow) flag set.

6.2.4 Illustrative Program: Subtraction of Two Unsigned Numbers

PROBLEM STATEMENT

Write a program to do the following:

1. Load the number 30H in register B and 39H in register C.
2. Subtract 39H from 30H.
3. Display the answer at PORT1.

PROGRAM

The illustrative program for subtraction of two unsigned numbers is presented as Figure 6.6 to show the register contents during some of the steps.

PROGRAM DESCRIPTION

1. Registers B and C are loaded with 30H and 39H, respectively. The instruction MOV A,B copies 30H into the accumulator (shown as register contents). This is an essential step because the contents of a register can be subtracted only from the contents of the accumulator and not from any other register.
2. To execute the instruction SUB C the microprocessor performs the following steps internally:

Step 1:
39H = 0 0 1 1 1 0 0 1
1's complement of 39H = 1 1 0 0 0 1 1 0
+
Step 2:
Add 01 = 0 0 0 0 0 0 0 1
2's complement of 39H = 1 1 0 0 0 1 1 1
+
Step 3:
Add 30H to 2's complement of 39H = 0 0 1 1 0 0 0 0
CY [0] 1 1 1 1 0 1 1 1

Step 4: Complement carry [1] 1 1 1 1 0 1 1 1 = F7H

Flag Status: S = 1, Z = 0, CY = 1

3. The number F7H is a 2's complement of the magnitude (39H – 30H) = 09H.
4. The instruction OUT displays F7H at PORT1.

PROGRAM OUTPUT

This program will display F7H as the output. In this program, the unsigned numbers were used to perform the subtraction. Now, the question is: How do you recognize that the answer F7H is really a 2's complement of 09H and not a straight binary F7H?

The answer lies with the Carry flag. If the Carry flag (also known as the Borrow flag in subtraction) is set, the answer is in 2's complement. The Carry flag raises a second question: Why isn't it a positive sum with a carry? The answer is implied by the instruction SUB (it is a subtraction).

There is no way to differentiate between a straight binary number and 2's complement by examining the answer at the output port. The flags are internal and not easily displayed. However, a programmer can test the Carry flag by using the instruction Jump On Carry (JC) and can find a way to indicate that the answer is in 2's complement. (This is discussed in Branch instructions.)

Memory Address (H)	Machine Code	Instruction Opcode	Instruction Operand	Comments and Register Contents
HI-LO				
XX00	06	MVI	B,30H	Load the minuend in register B
01	30			Load the subtrahend in register C
02	0E	MVI	C,39H	The register contents:
03	39			
04	78	MOV	A,B	
05	91	SUB	C	
06	D3	OUT	PORT1	
07	PORT#			
08	76	HLT		

A	30		F
B	30	39	C

A	F7	S Z CY 1 0 1	F
B	30	39	C

FIGURE 6.6
Illustrative Program for Subtraction of Two Unsigned Numbers

6.2.5 Review of Important Concepts

1. The arithmetic operations implicitly assume that the contents of the accumulator are one of the operands.

2. The results of the arithmetic operations are stored in the accumulator; thus, the previous contents of the accumulator are altered.
3. The flags are modified to reflect the data conditions of an operation.
4. The contents of the source register are not changed as a result of an arithmetic operation.
5. In the Add operation, if the sum is larger than 8-bit, CY is set.
6. The Subtract operation is performed by using the 2's complement method.
7. If a subtraction results in a negative number, the answer is in 2's complement and CY (the Borrow flag) is set.
8. In unsigned arithmetic operations, the Sign flag (S) should be ignored.
9. The instructions INR (Increment) and DCR (Decrement) are special cases of the arithmetic operations. These instructions can be used for any one of the registers, and they do not affect CY, even if the result is larger than 8-bit. All other flags are affected by the result in the register used (not by the contents of the accumulator).

6.3 LOGIC OPERATIONS

A microprocessor is basically a programmable logic chip. It can perform all the logic functions of the hard-wired logic through its instruction set. The 8085 instruction set includes such logic functions as AND, OR, Ex OR, and NOT (complement). The opcodes of these operations are as follows:*

ANA:	AND	Logically AND the contents of a register.
ANI :	AND Immediate	Logically AND 8-bit data.
ORA:	OR	Logically OR the contents of a register.
ORI :	OR Immediate	Logically OR 8-bit data.
XRA:	X-OR	Exclusive-OR the contents of a register.
XRI :	X-OR Immediate	Exclusive-OR 8-bit data.

All logic operations are performed in relation to the contents of the accumulator. The instructions of these logic operations are described below.

INSTRUCTIONS

The logic instructions

1. implicitly assume that the accumulator is one of the operands.
2. reset (clear) the CY flag. The instruction CMA is an exception; it does not affect any flags.
3. modify the Z, P, and S flags according to the data conditions of the result.
4. place the result in the accumulator.
5. do not affect the contents of the operand register.

*Memory-related logic operations are excluded here; they will be discussed in the next chapter.

Opcode	Operand	Description
ANA	R	Logical AND with Accumulator ☐ This is a 1-byte instruction ☐ Logically ANDs the contents of the register R with the contents of the accumulator ☐ 8085: CY is reset and AC is set
ANI	8-bit	AND Immediate with Accumulator ☐ This is a 2-byte instruction ☐ Logically ANDs the second byte with the contents of the accumulator ☐ 8085: CY is reset and AC is set
ORA	R	Logically OR with Accumulator ☐ This is a 1-byte instruction ☐ Logically ORs the contents of the register R with the contents of the accumulator
ORI	8-bit	OR Immediate with Accumulator ☐ This is a 2-byte instruction ☐ Logically ORs the second byte with the contents of the accumulator
XRA	R	Logically Exclusive-OR with Accumulator ☐ This is a 1-byte instruction ☐ Exclusive-ORs the contents of register R with the contents of the accumulator
XRI	8-bit	Exclusive-OR Immediate with Accumulator ☐ This is a 2-byte instruction ☐ Exclusive-ORs the second byte with the contents of the accumulator
CMA		Complement Accumulator ☐ This is a 1-byte instruction that complements the contents of the accumulator ☐ No flags are affected

6.3.1 Logic AND

The process of performing logic operations through the software instructions is slightly different from the hardwired logic. The AND gate shown in Figure 6.7(a) has two inputs and one output. On the other hand, the instruction ANA simulates eight AND gates, as shown in Figure 6.7(b). For example, assume that register B holds 77H and the accumulator A holds 81H. The result of the instruction ANA B is 01H and is placed in the accumulator replacing the previous contents, as shown in Figure 6.7(b).

Figure 6.7(b) shows that each bit of register B is independently ANDed with each bit of the accumulator, thus simulating eight 2-input AND gates.

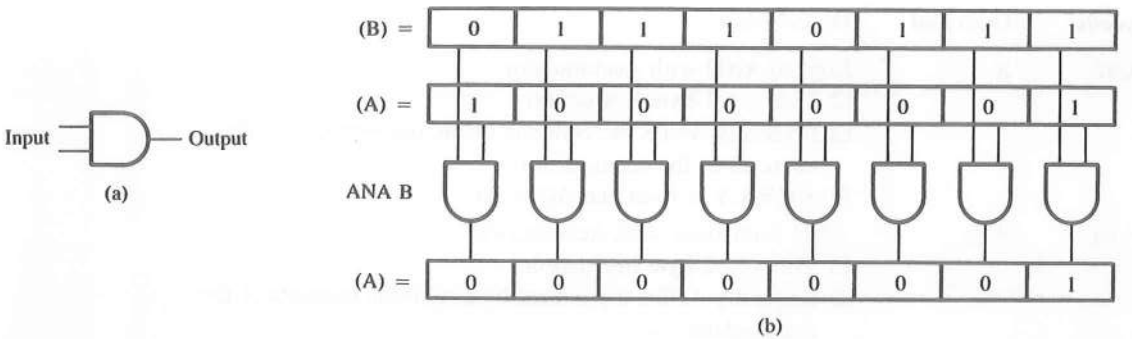


FIGURE 6.7
AND Gate (a) and a Simulated ANA Instruction (b)

6.3.2 Illustrative Program: Data Masking with Logic AND

PROBLEM STATEMENT

To conserve energy and to avoid an electrical overload on a hot afternoon, implement the following procedures to control the appliances throughout the house (Figure 6.8). Assume that the control switches are located in the kitchen, and they are available to anyone in the house. Write a set of instructions to

1. turn on the air conditioner if switch S_7 of the input port 00H is on.
2. ignore all other switches of the input port even if someone attempts to turn on other appliances.

(To perform this experiment on your single-board microcomputer, simulate the reading of the input port 00H with the instruction MVI A, 8-bit data.)

PROBLEM ANALYSIS

In this problem you are interested in only one switch position, S_7, which is connected to data line D_7. Assume that various persons in the family have turned on the switches of the air conditioner (S_7), the radio (S_4), and the lights (S_3, S_2, S_1, S_0).

If the microprocessor reads the input port (IN 00H), the accumulator will have data byte 9FH. This can be simulated by using the instruction MVI A,9FH. However, if you are interested in knowing only whether switch S_7 is on, you can mask bits D_6 through D_0 by ANDing the input data with a byte that has 0 in bit positions D_6 through D_0 and 1 in bit position D_7.

$$\begin{array}{cccc|cccc} D_7 & D_6 & D_5 & D_4 & D_3 & D_2 & D_1 & D_0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} = 80H$$

After bits D_6 through D_0 have been masked, the remaining byte can be sent to the output port to simulate turning on the air conditioner.

PROGRAM

Memory Address	Machine Code	Instruction Opcode	Instruction Operand	Comments
HI-LO				
XX00	3E	MVI	A,Data	;This instruction simulates the
01	9F			; instruction IN 00H
02	E6	ANI	80H	;Mask all the bits except D_7
03	80			
04	D3	OUT	01H	;Turn on the air conditioner if
05	01			; S_7 is on
06	76	HLT		;End of the program

PROGRAM OUTPUT

The instruction ANI 80H ANDs the accumulator data as follows:

$$\begin{array}{rl} & (A) = 1\ 0\ 0\ 1\ \ 1\ 1\ 1\ 1 \quad (9FH) \\ \text{AND} & \\ & (\text{Masking Byte} = 1\ 0\ 0\ 0\ \ 0\ 0\ 0\ 0 \quad (80H) \\ \hline & (A) = 1\ 0\ 0\ 0\ \ 0\ 0\ 0\ 0 \quad (80H) \end{array}$$

Flag Status: S = 1, Z = 0, CY = 0

The ANDing operation always resets the CY flag. The result (80H) will be placed in the accumulator and then sent to the ouptput port, and logic 1 of data bit D_7 turns on the air conditioner. In this example, the output (80H) is the same as the masking data byte (80H) because switch S_7 (or data bit D_7) is on. If S_7 is off, the output will be zero.

The masking is a commonly used technique to eliminate unwanted bits in a byte. The masking byte to be logically ANDed is determined by placing 0s in bit positions that are to be masked and by placing 1s in the remaining bit positions.

6.3.3 OR, Exclusive-OR, and NOT

The instruction ORA (and ORI) simulates logic ORing with eight 2-input OR gates; this process is similar to that of ANDing, explained in the previous section. The instruction XRA (and XRI) performs Exclusive-ORing of eight bits, and the instruction CMA inverts the bits of the accumulator.

Assume register B holds 93H and the accumulator holds 15H. Illustrate the results of the instructions ORA B, XRA B, and CMA. **Example 6.7**

---

1. The instruction ORA B will perform the following operation:

$$
\begin{array}{rl}
 & (B) = 1\ 0\ 0\ 1\ \ 0\ 0\ 1\ 1 \quad (93H) \\
OR & \\
 & (A) = 0\ 0\ 0\ 1\ \ 0\ 1\ 0\ 1 \quad (15H) \\
\hline
 & (A) = 1\ 0\ 0\ 1\ \ 0\ 1\ 1\ 1 \quad (97H)
\end{array}
$$

Flag Status: S = 1, Z = 0, CY = 0

The result 97H will be placed in the accumulator, the CY flag will be reset, and the other flags will be modified to reflect the data conditions in the accumulator.

2. The instruction XRA B will perform the following operation.

$$
\begin{array}{rl}
 & (B) = 1\ 0\ 0\ 1\ \ 0\ 0\ 1\ 1 \quad (93H) \\
X\text{-}OR & \\
 & (A) = 0\ 0\ 0\ 1\ \ 0\ 1\ 0\ 1 \quad (15H) \\
\hline
 & (A) = 1\ 0\ 0\ 0\ \ 0\ 1\ 1\ 0 \quad (86H)
\end{array}
$$

Flag Status: S = 1, Z = 0, CY = 0

The result 86H will be placed in the accumulator, and the flags will be modified as shown.

3. The instruction CMA will result in

$$
\begin{array}{rl}
 & (A) = 0\ 0\ 0\ 1\ \ 0\ 1\ 0\ 1 \quad (15H) \\
CMA & \\
 & (A) = 1\ 1\ 1\ 0\ \ 1\ 0\ 1\ 0 \quad (EAH)
\end{array}
$$

The result EAH will be placed in the accumulator and no flags will be modified.

6.3.4 Setting and Resetting Specific Bits

At various times, we may want to set or reset a specific bit without affecting the other bits. OR logic can be used to set the bit, and AND logic can be used to reset the bit.

Example 6.8

In Figure 6.8, keep the radio on (D_4) continuously without affecting the functions of other appliances, even if someone turns off the switch S_4.

Solution

To keep the radio on without affecting the other appliances, the bit D_4 should be set by ORing the reading of the input port with the data byte 10H as follows:

$$
\begin{array}{rl}
IN\ 00H: & (A) = D_7\ D_6\ D_5\ D_4\ \ D_3\ D_2\ D_1\ D_0 \\
ORI\ 10H: & \quad = 0\ \ 0\ \ 0\ \ 1\ \ \ 0\ \ 0\ \ 0\ \ 0 \\
\hline
 & (A) = D_7\ D_6\ D_5\ \ 1\ \ D_3\ D_2\ D_1\ D_0
\end{array}
$$

Flag Status: CY = 0; others will depend on data.

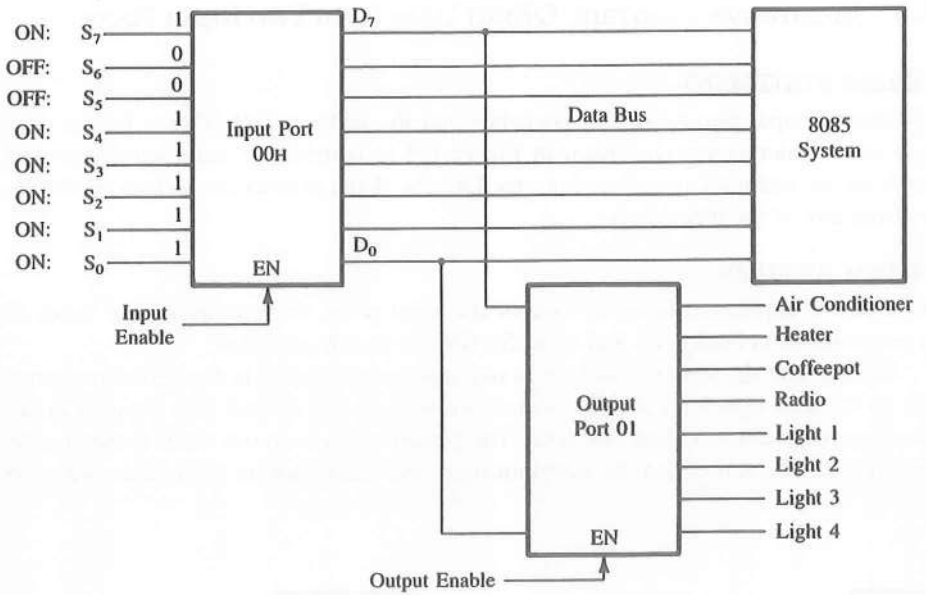


FIGURE 6.8
Input Port to Control Appliances

The instruction IN reads the switch positions shown as D_7–D_0 and the instruction ORI sets the bit D_4 without affecting any other bits.

Example 6.9

In Figure 6.8, assume it is winter, and turn off the air conditioner without affecting the other appliances.

Solution

To turn off the air conditioner, reset bit D_7 by ANDing the reading of the input port with the data byte 7FH as follows:

IN 00H:	(A) =	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
ANI 7FH:	=	0	1	1	1	1	1	1	1
		0	D_6	D_5	D_4	D_3	D_2	D_1	D_0

Flag Status: CY = 0; others will depend on the data bits.

The ANI instruction resets bit D_7 without affecting the other bits.

6.3.5 Illustrative Program: ORing Data from Two Input Ports

PROBLEM STATEMENT

An additional input port with eight switches and the address 01H (Figure 6.9) is connected to the microcomputer shown in Figure 6.8 to control the same appliances and lights from the bedroom as well as from the kitchen. Write instructions to turn on the devices from any of the input ports.

PROBLEM ANALYSIS

To turn on the appliances from any one of the input ports, the microprocessor needs to read the switches at both ports and logically OR the switch positions.

Assume that the switch positions in one input port (located in the bedroom) correspond to the data byte 91H and the switch positions in the second port (located in the kitchen) correspond to the data byte A8H. The person in the bedroom wants to turn on the air conditioner, the radio, and the bedroom light; and the person in the kitchen wants to

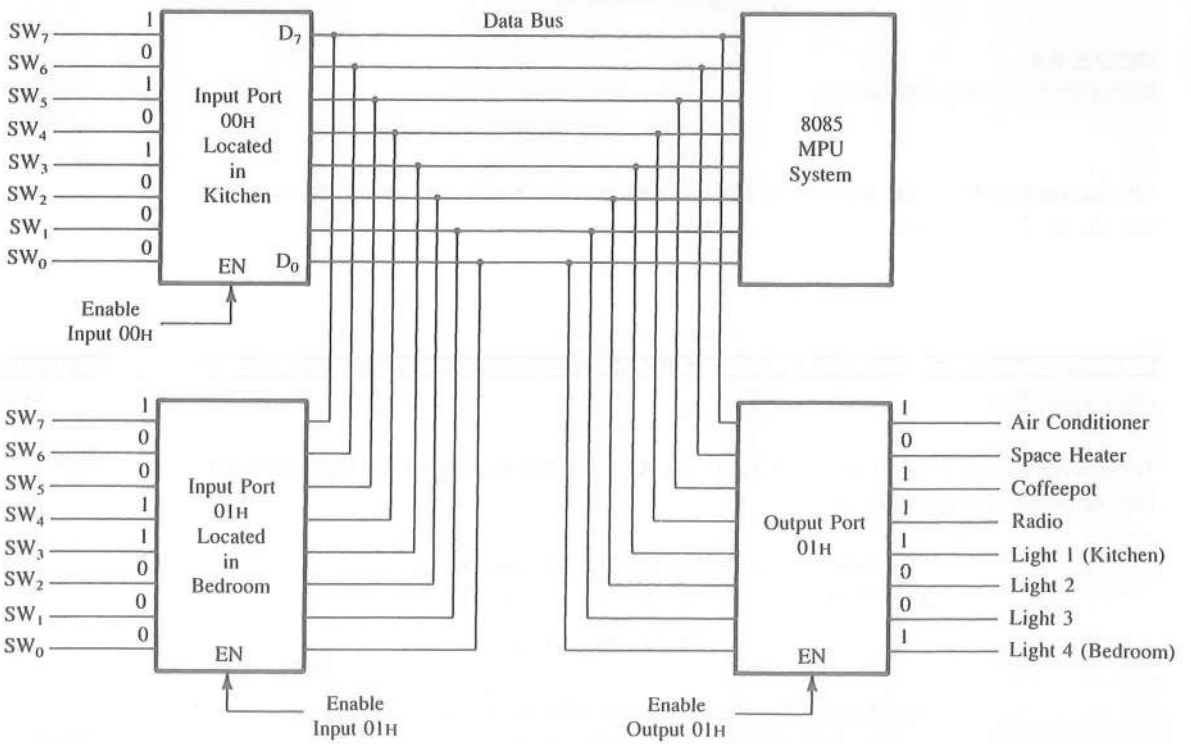


FIGURE 6.9
Two Input Ports to Control Output Devices

turn on the air conditioner, the coffeepot, and the kitchen light. By ORing these two data bytes the microprocessor can turn on the necessary appliances.

To test this program, we must simulate the readings of the input port by loading the data into registers—for example, into B and C.

PROGRAM

Memory Address	Machine Code	Instructions Opcode	Operand	Comments
HI-LO				
XX00	06	MVI	B,91H	;This instruction simulates read-
01	91			; ing input port 01H
02	0E	MVI	C,A8H	;This instruction simulates read-
03	A8			; ing input port 00H
04	78	MOV	A,B	;It is necessary to transfer data
				; byte from B to A to OR with
				; C. B and C cannot be ORed
				; directly
05	B1	ORA	C	;Combine the switch positions
				; from registers B and C in the
				; accumulator
06	D3	OUT	PORT1	;Turn on appliances and lights
07	PORT1			
08	76	HLT		;End of the program

PROGRAM OUTPUT

By logically ORing the data bytes in registers B and C

$$
\begin{aligned}
(B) \rightarrow (A) &= 1\ 0\ 0\ 1 \quad 0\ 0\ 0\ 1 \quad (91H) \\
(C) &= 1\ 0\ 1\ 0 \quad 1\ 0\ 0\ 0 \quad (A8H) \\
\hline
(A) &= 1\ 0\ 1\ 1 \quad 1\ 0\ 0\ 1 \quad (B9H)
\end{aligned}
$$

Flag Status: S = 1, Z = 0, CY = 0

Data byte B9H is placed in the accumulator that turns on the air conditioner, radio, coffeepot, and bedroom and kitchen lights.

6.3.6 Review of Important Concepts

1. Logic operations are performed in relation to the contents of the accumulator.
2. Logic operations simulate eight 2-input gates (or inverters).
3. The Sign, Zero (and Parity) flags are modified to reflect the status of the operation. The Carry flag is reset. However, the NOT operation does not affect any flags.
4. After a logic operation has been performed, the answer is placed in the accumulator replacing the original contents of the accumulator.

5. The logic operations cannot be performed directly with the contents of two registers.
6. The individual bits in the accumulator can be set or reset using logic instructions.

See Questions and Programming Assignments 20–29 at the end of this chapter.

6.4 BRANCH OPERATIONS

The **branch instructions** are the most powerful instructions because they allow the microprocessor to change the sequence of a program, either unconditionally or under certain test conditions. These instructions are the key to the flexibility and versatility of a computer.

The microprocessor is a sequential machine; it executes machine codes from one memory location to the next. Branch instructions instruct the microprocessor to go to a different memory location, and the microprocessor continues executing machine codes from that new location. The address of the new memory location is either specified explicitly or supplied by the microprocessor or by extra hardware. The branch instructions are classified in three categories:

1. Jump instructions
2. Call and Return instructions
3. Restart instructions

This section is concerned with applications of Jump instructions. The Call and Return instructions are associated with the subroutine technique and will be discussed in Chapter 9; Restart instructions are associated with the interrupt technique and will be discussed in Chapter 12.

The Jump instructions specify the memory location explicitly. They are 3-byte instructions: one byte for the operation code, followed by a 16-bit memory address. Jump instructions are classified into two categories: Unconditional Jump and Conditional Jump.

6.4.1 Unconditional Jump

The 8085 instruction set includes one unconditional Jump instruction. The unconditional Jump instruction enables the programmer to set up continuous loops.

INSTRUCTION

Opcode	Operand	Description
JMP	16-bit	Jump ☐ This is a 3-byte instruction ☐ The second and third bytes specify the 16-bit memory address. However, the second byte specifies the low-order and the third byte specifies the high-order memory address

For example, to instruct the microprocessor to go to the memory location 2000H, the mnemonics and the machine code entered will be as follows:

Machine Code	Mnemonics
C3	JMP 2000H
00	
20	

Note the sequence of the machine code. The 16-bit memory address of the jump location is entered in the reverse order, the low-order byte (00H) first, followed by the high-order byte (20H). The 8085 is designed for such a reverse sequence. The jump location can also be specified using a label. While writing a program, you may not know the exact memory location to which a program sequence should be directed. In that case, the memory address can be specified with a label (or a name). This is particularly useful and necessary for an assembler. However, you should not specify both a label and its 16-bit address in a Jump instruction. Furthermore, you cannot use the same label for different memory locations. The next illustrative program shows the use of the Jump instruction.

6.4.2 Illustrative Program: Unconditional Jump to Set Up a Continuous Loop

PROBLEM STATEMENT

Modify the program in Example 6.2 to read the switch positions continuously and turn on the appliances accordingly.

PROBLEM ANALYSIS

One of the major drawbacks of the program in Example 6.2 is that the program reads switch positions once and then stops. Therefore, if you want to turn on/off different appliances, you have to reset the system and start all over again. This is impractical in real-life situations. However, the unconditional Jump instruction, in place of the HLT instruction, will allow the microcomputer to monitor the switch positions continuously.

PROGRAM

Memory Address	Machine Code	Label	Mnemonics	Comments
2000	DB	START:	IN 00H	;Read input switches
2001	00			
2002	D3		OUT 01H	;Turn on devices according to
2003	01			; switch positions
2004	C3		JMP START	;Go back to beginning and
2005	00			; read the switches again
2006	20			

PROGRAM FORMAT

The program includes one more column called *label.* The memory location 2000H is defined with the label START; therefore, the operand of the Jump instruction can be specified by the label START. The program sets up the endless loop, and the microprocessor monitors the input port continuously. The output will reflect any change in the switch positions.

6.4.3 Conditional Jumps

Conditional Jump instructions allow the microprocessor to make decisions based on certain conditions indicated by the flags. After logic and arithmetic operations, flip-flops (flags) are set or reset to reflect data conditions. The conditional Jump instructions check the flag conditions and make decisions to change or not to change the sequence of a program.

FLAGS

The 8085 flag register has five flags, one of which (Auxiliary Carry) is used internally. The other four flags used by the Jump instructions are

1. Carry flag
2. Zero flag
3. Sign flag
4. Parity flag

Two Jump instructions are associated with each flag. The sequence of a program can be changed either because the condition is present or because the condition is absent. For example, while adding the numbers you can change the program sequence either because the carry is present (JC = Jump On Carry) or because the carry is absent (JNC = Jump On No Carry).

INSTRUCTIONS

All conditional Jump instructions in the 8085 are 3-byte instructions; the second byte specifies the low-order (line number) memory address, and the third byte specifies the high-order (page number) memory address. The following instructions transfer the program sequence to the memory location specified under the given conditions:

Opcode	Operand	Description
JC	16-bit	Jump On Carry (if result generates carry and CY = 1)
JNC	16-bit	Jump On No Carry (CY = 0)
JZ	16-bit	Jump On Zero (if result is zero and Z = 1)
JNZ	16-bit	Jump On No Zero (Z = 0)
JP	16-bit	Jump On Plus (if $D_7 = 0$, and S = 0)
JM	16-bit	Jump On Minus (if $D_7 = 1$, and S = 1)
JPE	16-bit	Jump On Even Parity (P = 1)
JPO	16-bit	Jump On Odd Parity (P = 0)

All the Jump instructions are listed here for an overview. The Zero and Carry flags and related Jump instructions are used frequently. They are illustrated in the following examples.

6.4.4 Illustrative Program: Testing of the Carry Flag

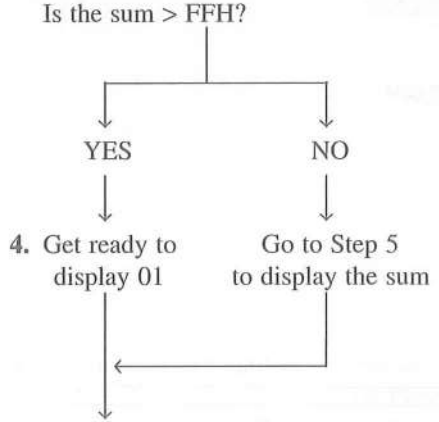
PROBLEM STATEMENT

Load the hexadecimal numbers 9BH and A7H in registers D and E, respectively, and add the numbers. If the sum is greater than FFH, display 01H at output PORT0; otherwise, display the sum.

PROBLEM ANALYSIS AND FLOWCHART

The problem can be divided into the following steps:

1. Load the numbers in the registers.
2. Add the numbers.
3. Check the sum.
   Is the sum > FFH?



5. Display.
6. End.

FLOWCHART AND ASSEMBLY LANGUAGE PROGRAM

The six steps listed above can be converted into a flowchart and assembly language program as shown in Figure 6.10.

Step 3 is a decision-making block. In a flowchart, the decision-making process is represented by a diamond shape. It is important to understand how this block is translated into the assembly language program. By examining the block carefully you will notice the following:

1. The question is: Is there a Carry?
2. If the answer is no, change the sequence of the program. In the assembly language this is equivalent to Jump On No Carry—JNC.

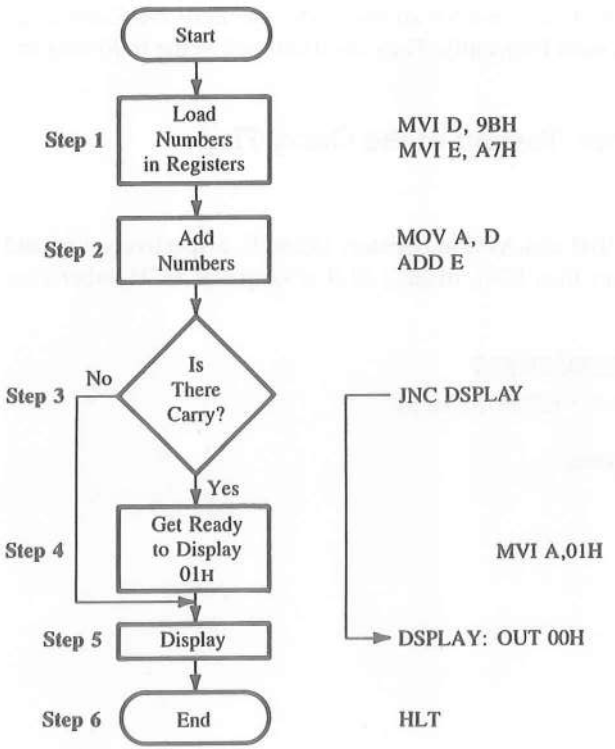


FIGURE 6.10
Flowchart and Assembly Language Program to Test Carry Flag

3. Now the next question is where to change the sequence—to Step 5. At this point the exact location is not known, but it is labeled DSPLAY.
4. The next step in the sequence is 4. Get ready to display byte 01H.
5. After completing the straight line sequence, translate Step 5 and Step 6: Display at the port and halt.

MACHINE CODE WITH MEMORY ADDRESSES

Assuming your R/W memory begins at 2000H, the preceding assembly language program can be translated as follows:

Memory Address	Machine Code	Label	Mnemonics
2000	16	START:	MVI D,9BH
2001	9B		
2002	1E		MVI E,A7H
2003	A7		

```
2004    7A              MOV A,D
2005    83              ADD E
2006    D2              JNC DSPLAY
2007    X
2008    X
2009    3E              MVI A,01H
200A    01
200B    D3      DSPLAY: OUT 00H
200C    00
200D    76              HLT
```

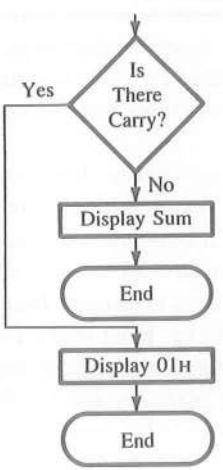
While translating into the machine code, we leave memory locations 2007H and 2008H blank because the exact location of the transfer is not known. What is known is that two bytes should be reserved for the 16-bit address. After completing the straight line sequence, we know the memory address of the label DSPLAY; i.e., 200BH. This address must be placed in the reversed order as shown:

```
2007    0B      Low-order: Line Number
2008    20      High-order: Page Number
```

USING THE INSTRUCTION JUMP ON CARRY (JC)

Now the question remains: Can the same problem be solved by using the instruction Jump On Carry (JC)? To use instruction JC, exchange the places of the answers YES and NO to the question: Is there a Carry? The flowchart will be as in Figure 6.11, and it shows that the program sequence is changed if there is a Carry. This flowchart has two end points; thus it will require a few more instructions than that of Figure 6.10. In this particular example, it is unimportant whether to use instruction JC or JNC, but in most cases the choice is made by the logic of a problem.

FIGURE 6.11
Flowchart for the Instruction Jump On Carry



6.4.5 Review of Important Concepts

1. The Jump instructions change program execution from its sequential order to a different memory location.
2. The Jump instructions can transfer program execution ahead of the sequence (Jump Forward) or behind the sequence (Jump Backward).
3. The unconditional Jump is, generally, used to set up continuous loops.
4. The conditional Jumps are used for the decision-making process based on the data conditions of the result, reflected by the flags.
5. Arithmetic and logic instructions modify the flags according to the data of the result, and the conditional branch instructions use them to make decisions. However, the branch instructions do not affect the flags.

CAUTION

The conditional Jump instructions will not function properly unless the preceding instruction sets the necessary flag. Data Copy instructions do not affect the flags; furthermore, some arithmetic and logic instructions either do not affect the flags or affect only certain flags.

See Questions and Programming Assignments 30–40 at the end of this chapter.

6.5 WRITING ASSEMBLY LANGUAGE PROGRAMS

Communicating with a microcomputer—giving it commands to perform a task and watching it perform them—is exciting. However, one can be uneasy communicating in strange mnemonics and hexadecimal machine codes. This feeling is like the uneasiness one has when beginning to speak a foreign language. How do we learn to communicate with a microcomputer in its assembly language? By using a few mnemonics at a time such as the mnemonics for Read the switches and Display the data. This chapter has introduced a group of basic instructions that can command the 8085 microprocessor to perform simple tasks.

After we know a few instructions, how do we begin to write a program? Any program, no matter how large, begins with mnemonics. And, just as several persons contribute to the construction of a hundred-story building, so the writing of a large program is usually the work of a team. In addition, the 8085 instruction set contains only 74 different instructions, some of them used quite frequently.

In a hundred-story building, most of the rooms are similar. If one knows the basic fundamentals of constructing a room, one can learn how to tie these rooms together in a coherent structure. However, planning and forethought are critical. Before beginning to build a structure, an architectural plan must be drawn. Similarly, to write a program, one

needs to draw up a plan of logical thoughts. A given task should be broken down into small units that can be built independently. This is called the **modular design approach.**

6.5.1 Getting Started

Writing a program is equivalent to giving specific *commands* to the microprocessor in a *sequence* to *perform a task.* The italicized words provide clues to writing a program. Let us examine these terms.

- ☐ *Perform a Task.* What is the task you are asking it to do?
- ☐ *Sequence.* What is the sequence you want it to follow?
- ☐ *Commands.* What are the commands (instruction set) it can understand?

These terms can be translated into steps as follows:

Step 1: Read the problem carefully.
Step 2: Break it down into small steps.
Step 3: Represent these small steps in a possible sequence with a flowchart—a plan of attack.
Step 4: Translate each block of the flowchart into appropriate mnemonic instructions.
Step 5: Translate mnemonics into the machine code.
Step 6: Enter the machine code in memory and execute. Only on rare occasions is a program successfully executed on the first attempt.
Step 7: Start troubleshooting (see Section 6.6, "Debugging a Program").

These steps are illustrated in the next section.

6.5.2 Illustrative Program: Microprocessor-Controlled Manufacturing Process

PROBLEM STATEMENT

A microcomputer is designed to monitor various processes (conveyer belts) on the floor of a manufacturing plant, presented schematically in Figure 6.12. The microcomputer has two input ports with the addresses F1H and F2H and an output port with the address F3H. Input port F1H has six switches, five of which (corresponding to data lines D_4–D_0) control the conveyer belts through the output port F3H. Switch S_7, corresponding to the data line D_7, is reserved to indicate an emergency on the floor. As a precautionary measure, input port F2H is controlled by the foreman, and its switch, S_7', is also used to indicate an emergency. Output line D_6 of port F3H is connected to the emergency alarm.

Write a program to

1. turn on the five conveyer belts according to the ON/OFF positions of switches S_4–S_0 at port F1H.
2. turn off the conveyer belts and turn on the emergency alarm only when both switches—S_7 from port F1H and S_7' from port F2H—are triggered.
3. monitor the switches continuously.

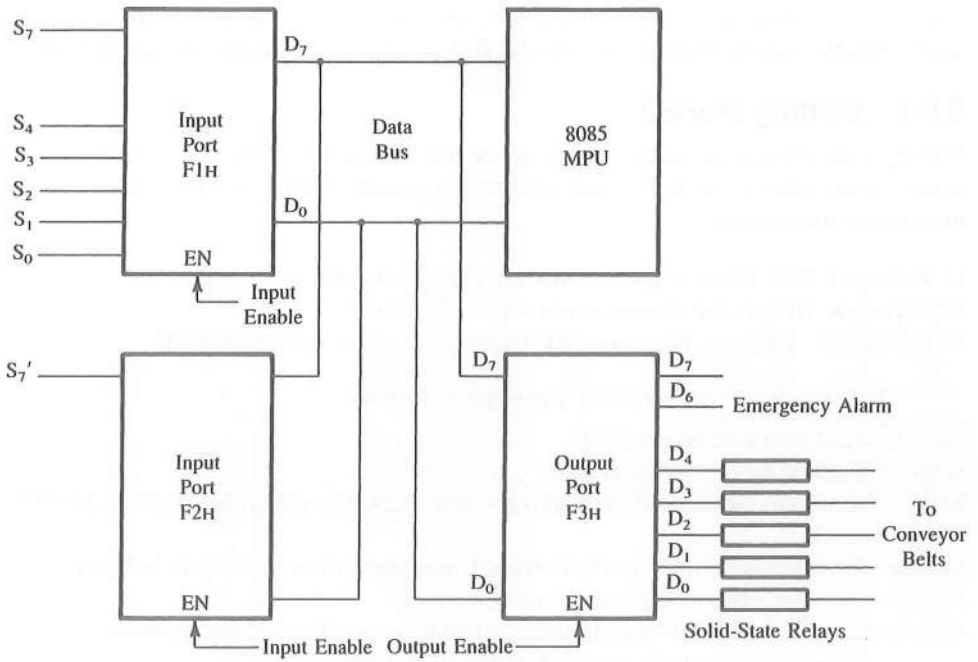


FIGURE 6.12
Input/Output Ports to Control Manufacturing Processes

PROBLEM ANALYSIS

To perform the tasks specified in the problem, the microprocessor needs to

1. read the switch positions.
2. check whether switches S_7 and S_7' from the ports F1H and F2H are on.
3. turn on the emergency signal if both switches are on, and turn off all the conveyer belts.
4. turn on the conveyer belts according to the switch positions S_0 through S_4 at input port F1H if both switches, S_7 and S_7', are not on simultaneously.
5. continue checking the switch positions.

FLOWCHART AND PROGRAM

The five steps listed above can be translated into a flowchart and an assembly language program as shown in Figure 6.13.

6.5.3 Documentation

A program is similar to a circuit diagram. Its purpose is to communicate to others what the program does and how it does it. Appropriate comments are critical for conveying the logic behind a program. The program as a whole should be self-documented.

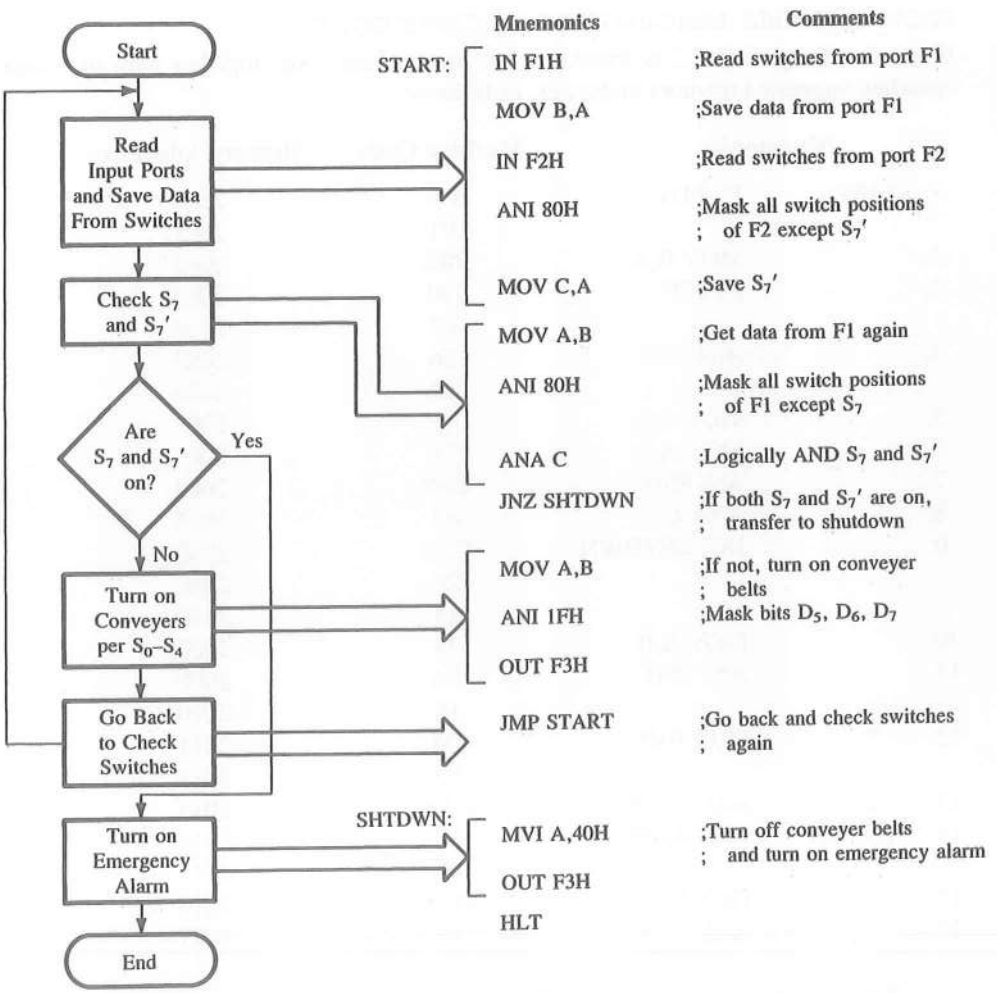


FIGURE 6.13
Flowchart and Program for Controlling Manufacturing Processes

The comments should explain what is intended; they should not explain the mnemonics. For example, the first comment in Figure 6.13 indicates that the switch positions are being read at the input port and that the reading is saved. There is no point in writing: MOV B,A means transfer the contents of the accumulator to register B. Similarly, in a schematic, one does not write the word *resistor* once a resistor is represented by a symbol. A comment can be omitted if it does not say anything more than repeat a mnemonic.

The labels START and SHTDWN indicate what actions are being taken. These are the landmarks in a program. Avoid "cute" labels. Using cute labels in a program is similar to representing a power supply in a schematic by a picture of the sun or a ground with the digit zero.

FROM ASSEMBLY LANGUAGE TO MACHINE CODE

Illustrative Program 6.5.2 is translated into its machine code, together with the corresponding suggested memory addresses, as follows:

	Mnemonics	Machine Code	Memory Addresses
1. START:	IN F1H	DB	2000
		F1	2001
2.	MOV B,A	78①	2002
3.	IN F2H	DB	2003
		F2	2004
4.	ANI 80H	E6	2005
		80	2006
5.	MOV C,A	4F	2007
6.	MOV A,B	78	2008
7.	ANI 80H	E6②	2009
8.	ANA C	A1	200A
9.	JNZ SHTDWN	C2③	200B
		20	200C
		14	200D
10.	MOV A,B	78	200E
11.	ANI 1FH	E6	200F
		1F	2010
12.	OUT F3H	D3	2011
		F3	2012
13.	JMP START	C3④	2013
14. SHTDWN:	MVI A,40H	3E	2014
		40	2015
15.	OUT F3H	D3⑤	2016
16.	HLT	76	2017

This program includes several errors, indicated by the circled numbers beside the codes. (See Assignment 41 for the debugging of this program.)

PROGRAM EXECUTION

The above machine codes can be loaded in R/W memory, starting with memory address 2000H. The execution of the program can be done two ways. The first is to execute the entire code by pressing the Execute key, and the second is to use the Single-Step key on a single-board computer. The Single-Step key executes one instruction at a time, and by using the Examine Register key, you can observe the contents of the registers and the flags as each instruction is being executed. The Single-Step and Examine Register techniques are discussed in Chapter 7 under the topic "Dynamic Debugging."

6.6 DEBUGGING A PROGRAM

Debugging a program is similar to troubleshooting hardware, but it is much more difficult and cumbersome. It is easy to poke and pinch at the components in a circuit, but, in a program, the result is generally binary: either it works or it does not work. When it does not work, very few clues alert you to what exactly went wrong. Therefore, it is essential to search carefully for the errors in the program logic, machine codes, and execution.

The debugging process can be divided into two parts: static debugging and dynamic debugging.

Static debugging is similar to visual inspection of a circuit board; it is done by a paper-and-pencil check of a flowchart and machine code. **Dynamic debugging** involves observing the output, or register contents, following the execution of each instruction (the single-step technique) or of a group of instructions (the breakpoint technique). Dynamic debugging will be discussed in the next chapter.

6.6.1 Debugging Machine Code

Translating the assembly language to the machine code is similar to building a circuit from a schematic diagram; the machine code will have errors just as would the circuit board. The following errors are common:

1. Selecting a wrong code.
2. Forgetting the second or third byte of an instruction.
3. Specifying the wrong jump location.
4. Not reversing the order of high and low bytes in a Jump instruction.
5. Writing memory addresses in decimal, thus specifying wrong jump locations.

The program for controlling manufacturing processes listed in Section 6.5.3 has several of these errors. These errors must be corrected before entering the machine code in the R/W memory of your system.

See Questions and Programming Assignments 41–43 at the end of this chapter.

6.7 SOME PUZZLING QUESTIONS AND THEIR ANSWERS

After one learns something about the microprocessor architecture, memory, I/O, the instruction set, and simple programming, a few questions still remain unanswered. These questions do not fit into any particular discussion. They just lurk in the corners of one's mind to reappear once in a while when one is in a contemplative mood. This section attempts to answer some of these unasked questions.

1. *What happens in a single-board microcomputer when the power is turned on and the Reset key is pushed?*

When the power is turned on, the monitor program stored either in EPROM or ROM comes alive. The Reset key clears the program counter, and the program counter holds the memory address 0000H. Some systems are automatically reset when the power is turned on (called power-on reset).

2. *How does the microprocessor know how and when to start?*

As soon as the Reset key is pushed, the program counter places the memory address 0000H on the address bus, the instruction at that location is fetched, and the execution of the Key Monitor program begins. Therefore, the Key Monitor program is stored on page 00H.

3. *What is a monitor program?*

In a single-board microcomputer with a Hex keyboard, the instructions are entered in R/W memory through the keyboard. The Key Monitor program is a set of instructions that continuously checks whether a key is pressed and stores the binary equivalent of a pressed key in a memory location.

4. *What is an assembler?*

An assembler is a program that translates the mnemonics into their machine code. It is generally not available on a single-board microcomputer.

A program can be entered in mnemonics in a microcomputer equipped with an ASCII keyboard. The assembler will translate mnemonics into the 8085 machine code and assign memory locations to each machine code, thus avoiding the manual assembly and the errors associated with it. Additional instructions can be inserted anywhere in the program, and the assembler will reassign all the new memory locations and jump locations.

5. *How does the microprocessor know what operation to perform first (Read/Write memory or Read/Write I/O)?*

The first operation is always a Fetch instruction.

6. *How does the microprocessor differentiate among a positive number, a negative number, and a bit pattern?*

It does not know the difference. The microprocessor views any data byte as eight binary digits. The programmer is responsible for providing the interpretation.

For example, after an arithmetic or logic operation, if the bits in the accumulator are

$$1\ 1\ 1\ 1\ \ 0\ 0\ 1\ 0 = \text{F2H}$$

the Sign flag is set because $D_7 = 1$. This does not mean it is a negative number, even if the Sign flag is set. The Sign flag indicates only that $D_7 = 1$. The eight bits in the ac-

cumulator could be a bit pattern, or a positive number larger than 127_{10}, or the 2's complement of a number.

7. *If flags are individual flip-flops, can they be observed on an oscilloscope?*

No, they cannot be observed on an oscilloscope. The flag register is internal to the microprocessor. However, they can be tested through conditional branch instructions, and they can be examined by storing them on the stack memory (see Chapter 9).

8. *If the program counter is always one count ahead of the memory location from which the machine code is being fetched, how does the microprocessor change the sequence of program execution with a Jump instruction?*

When a Jump instruction is fetched, its second and third bytes (a new memory location) are placed in the W and Z registers of the microprocessor. After the execution of the Jump instruction, the contents of the W and Z registers are placed on the address bus to fetch the instruction from a new memory location, and the program counter is loaded by updating the contents of the W and Z registers.

SUMMARY

The instructions from the 8085 instruction set introduced in this chapter are summarized below to provide an overview. After careful examination of these instructions, you will begin to see a pattern emerge from the mnemonics, the number of bytes required for the various instructions, and the tasks the 8085 can perform. Read the notations (Rs) as the contents of the source register, (Rd) as the contents of the destination register, (A) as the contents of the accumulator, and (R) as the contents of the register R.*

Instructions	Tasks	Addressing Mode
Data transfer (Copy) Instructions		
1. MOV Rd,Rs	Copy (Rs) into (Rd).	Register
2. MVI R,8-bit	Load register R with the 8-bit data.	Immediate
3. IN 8-bit port address	Read data from the input port.	Direct
4. OUT 8-bit port address	Write data in the output port.	Direct
Arithmetic Instructions		
1. ADD R	Add (R) to (A).	Register
2. ADI 8-bit	Add 8-bit data to (A).	Immediate
3. SUB R	Subtract (R) from (A).	Register
4. SUI 8-bit	Subtract 8-bit data from (A).	Immediate

*R, Rs, and Rd represent any one of the 8-bit registers—A, B, C, D, E, H, and L.

5. INR R	Increment (R).	Register
6. DCR R	Decrement (R).	Register
Logic Instructions		
1. ANA R	Logically AND (R) with (A).	Register
2. ANI 8-bit	Logically AND 8-bit data with (A).	Immediate
3. ORA R	Logically OR (R) with (A).	Register
4. ORI 8-bit	Logically OR 8-bit data with (A).	Immediate
5. XRA R	Logically Exclusive-OR (R) with (A).	Register
6. XRI 8-bit	Logically Exclusive-OR 8-bit data with (A).	Immediate
7. CMA	Complement (A).	
Branch Instructions		
1. JMP 16-bit	Jump to 16-bit address unconditionally.	Immediate
2. JC 16-bit	Jump to 16-bit address if the CY flag is set.	Immediate
3. JNC 16-bit	Jump to 16-bit address if the CY flag is reset.	Immediate
4. JZ 16-bit	Jump to 16-bit address if the Zero flag is set.	Immediate
5. JNZ 16-bit	Jump to 16-bit address if the Zero flag is reset.	Immediate
6. JP 16-bit	Jump to 16-bit address if the Sign flag is reset.	Immediate
7. JM 16-bit	Jump to 16-bit address if the Sign flag is set.	Immediate
8. JPE 16-bit	Jump to 16-bit address if the Parity flag is set.	Immediate
9. JPO 16-bit	Jump to 16-bit address if the Parity flag is reset.	Immediate
Machine Control Instructions		
1. NOP	No operation.	
2. HLT	Stop processing and wait.	

The set of instructions listed here is used frequently in writing assembly language programs. The important points to be remembered about these instructions are as follows:

1. The data transfer (copy) instructions copy the contents of the source into the destination without affecting the source contents.
2. The results of the arithmetic and logic operations are usually placed in the accumulator.
3. The conditional Jump instructions are executed according to the flags set after an operation. Not all instructions set the flags; in particular, the data transfer instructions do not set the flags.

See Section 2.5 for an overview of the instruction set and see inside the cover page for a complete instruction set according to the functions.

QUESTIONS AND PROGRAMMING ASSIGNMENTS

Note: To execute the instructions in the following assignments and use the Examine Register key, the HLT instruction must be replaced by an appropriate instruction for your system. In the Intel SDK-85 system, the HLT can be replaced by the RST1 (code CFH) instruction. If the HLT instruction is used, the system must be Reset to exit from the Halt instruction; that clears the registers.

Section 6.1: Data Transfer (Copy) Operations

1. Specify the contents of the registers and the flag status as the following instructions are executed.

	A	B	C	D	S	Z	CY
MVI A,00H							
MVI B,F8H							
MOV C,A							
MOV D,B							
HLT							

2. Assemble the instructions in Assignment 1, enter the code in R/W memory of a single-board microcomputer, and execute the instructions using the single-step key. Before you begin to execute the instructions, record the initial conditions of the registers and the flags using the Examine Register key. Observe the register contents and the flags as you execute each instruction.
3. Write instructions to load the hexadecimal number 65H in register C, and 92H in the accumulator A. Display the number 65H at PORT0 and 92H at PORT1.
4. Write instructions to read the data at input PORT 07H and at PORT 08H. Display the input data from PORT 07H at output PORT 00H, and store the input data from PORT 08H in register B.
5. Specify the output at PORT1 if the following program is executed.

```
MVI B,82H
MOV A,B
MOV C,A
MVI D,37H
OUT PORT1
HLT
```

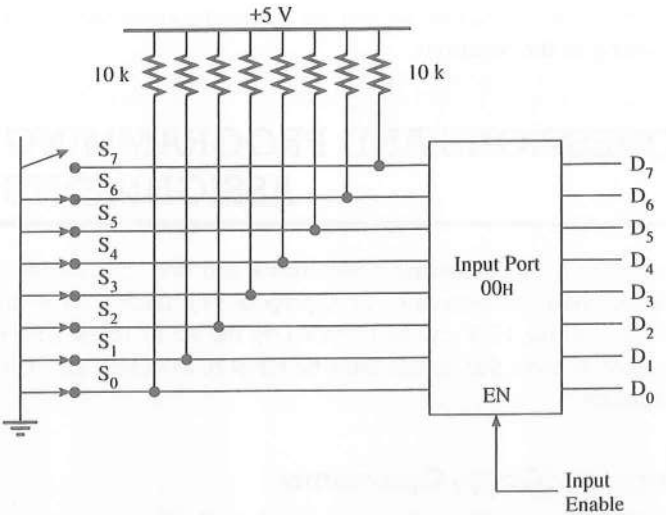


FIGURE 6.14
Input Port with Switches

6. If the switch S_7 of the input PORT0 (see Figure 6.14) connected to the data line D_7 is at logic 1 and other switches are at logic 0, specify the contents of the accumulator when the instruction IN PORT0 is executed.

```
MVI A,9FH
IN PORT0
MOV B,A
OUT PORT1
HLT
```

7. Specify the output at PORT1 and the contents of register B after executing the instructions in Assignment 6.

Section 6.2: Arithmetic Operations

8. Specify the register contents and the flag status as the following instructions are executed. Specify also the output at PORT0.

A	B	S	Z	CY	
00	FF	0	1	0	Initial Contents

```
MVI A,F2H
MVI B,7AH
ADD B
OUT PORT0
HLT
```

9. What operation can be performed by using the instruction ADD A?
10. What operation can be performed by using the instruction SUB A? Specify the status of Z and CY.
11. Specify the register contents and the flag status as the following instructions are executed.

A	C	S	Z	CY	
XX	XX	0	0	0	Initial Contents

```
MVI A,5EH
ADI A2H
MOV C,A
HLT
```

12. Assemble the program in Assignment 11, and enter the code in R/W memory of your system. Reset the system, and examine the contents of the flag register using the Examine Register key. Clear the flags by inserting 00H in the flag register, and execute each instruction using the single-step key. Verify the register contents and the flags as each instruction is being executed.
13. Write a program using the ADI instruction to add the two hexadecimal numbers 3AH and 48H and to display the answer at an output port.
14. Write instructions to
    a. load 00H in the accumulator.
    b. decrement the accumulator.
    c. display the answer.
    Specify the answer you would expect at the output port.
15. The following instructions subtract two unsigned numbers. Specify the contents of register A and the status of the S and CY flags. Explain the significance of the sign flag if it is set.

```
MVI A,F8H
SUI 69H
```

16. Specify the register contents and the flag status as the following instructions are executed.

A	B	S	Z	CY
XX	XX	X	X	X

```
SUB A
MOV B,A
DCR B
INR B
SUI 01H
HLT
```

17. Assemble the program in Assignment 16, and execute each instruction using the Single-Step key. Verify the register contents and the flags as each instruction is being executed.
18. Write a program to
    a. clear the accumulator.
    b. add 47H (use ADI instruction).
    c. subtract 92H.
    d. add 64H.
    e. display the results after subtracting 92H and after adding 64H.

    Specify the answers you would expect at the output ports.
19. Specify the reason for clearing the accumulator before adding the number 47H directly to the accumulator in Assignment 18.

Section 6.3: Logic Operations

20. What operation can be performed by using the instruction XRA A (Exclusive-OR the contents of the accumulator with itself)? Specify the status of Z and CY.
21. Specify the register contents and the flag status (S, Z, CY) after the instruction ORA A is executed.

```
MVI A,A9H
MVI B,57H
ADD B
ORA A
```

22. Assemble the program in Assignment 21 by adding an End instruction (such as RST1 in Intel's SDK-85 system), and execute the program. Verify the register contents and the flags by using the Examine Register key.
23. When the microprocessor reads an input port, the instruction IN does not set any flag. If the input reading is zero, what logic instruction can be used to set the Zero flag without affecting the contents of the accumulator?
24. Specify the register contents and the flag status as the following instructions are executed.

A	B	S	Z	CY
XX	XX	X	X	X

```
XRA A
MVI B,4AH
SUI 4FH
ANA B
HLT
```

25. Assemble the instructions in Assignment 24. Execute each instruction using the Single-Step key, and verify the register contents and flags.

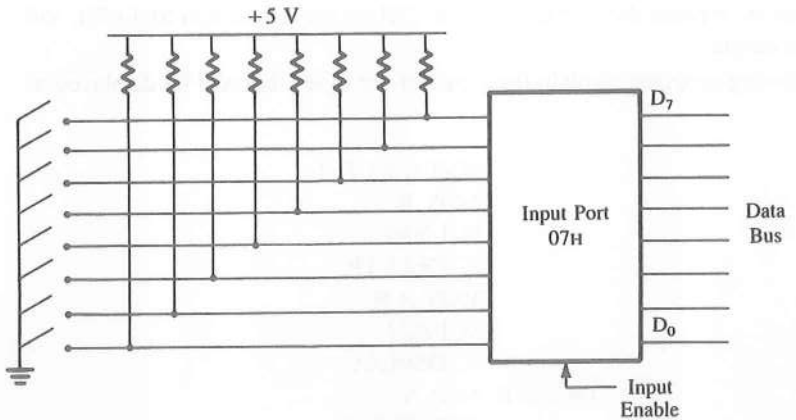


FIGURE 6.15
DIP Switch Input Port with Address 07H

26. Load the data byte A8H in register C. Mask the high-order bits (D_7–D_4), and display the low-order bits (D_3–D_0) at an output port.

27. Load the data byte 8EH in register D and F7H in register E. Mask the high-order bits (D_7–D_4) from both the data bytes, Exclusive-OR the low-order bits (D_3–D_0), and display the answer.

28. Load the bit pattern 91H in register B and 87H in register C. Mask all the bits except D_0 from registers B and C. If D_0 is at logic 1 in both registers, turn on the light connected to the D_0 position of output port 01H; otherwise, turn off the light.

29. Figure 6.15 shows an input port with an 8-key DIP switch. When all switches are off, the microprocessor reads the data FFH. When a switch is turned on (closed), it goes to logic 0 (for all switches ON, the data will be 00H). Write instructions to read the input port and, if all switches are open, set the Zero flag. (Use the instruction CMA to complement the input reading and ORA A to set the Zero flag.)

Section 6.4: Branch Operations

30. What is the output at PORT1 when the following instructions are executed?

```
        MVI A,8FH
        ADI 72H
        JC DSPLAY
        OUT PORT1
        HLT
DSPLAY: XRA A
        OUT PORT1
        HLT
```

31. In Question 30, replace the instruction ADI 72H by the instruction SUI 67H, and specify the output.

32. In the following program, explain the range of the bytes that will be displayed at PORT2.

```
        MVI A,BYTE1
        MOV B,A
        SUI 50H
        JC DELETE
        MOV A,B
        SUI 80H
        JC DSPLAY
DELETE: XRA A
        OUT PORT1
        HLT
DSPLAY: MOV A,B
        OUT PORT2
        HLT
```

33. Specify the address of the output port, and explain the type of numbers that can be displayed at the output port.

```
        MVI A,BYTE1     ;Get a data byte
        ORA A           ;Set flags
        JP OUTPRT       ;Jump if the byte is positive
        XRA A
OUTPRT: OUT F2H
        HLT
```

34. In Question 33, if BYTE1 = 92H, what is the output at port F2H?

35. Explain the function of the following program.

```
        MVI A,BYTE1     ;Get a data byte
        ORA A           ;Set flags
        JM OUTPRT
        OUT 01H
        HLT
OUTPRT: CMA             ;Find 2's complement
        ADI 01H
        OUT 01H
        HLT
```

36. In Question 35, if BYTE1 = A7H, what will be displayed at port 01H?

37. Rewrite the Section 6.4.4 illustrative program for testing the Carry flag using the instruction JC (Jump On Carry).

38. Write instructions to clear the CY flag, to load number FFH in register B, and increment (B). If the CY flag is set, display 01 at the output port; otherwise, display the contents of register B. Explain your results.

39. Write instructions to clear the CY flag, to load number FFH in register C, and to add 01 to (C). If the CY flag is set, display 01 at an output port; otherwise, display the contents of register C. Explain your results. Are they the same as in Question 38?

40. Write instructions to load two unsigned numbers in register B and register C, respectively. Subtract (C) from (B). If the result is in 2's complement, convert the result in absolute magnitude and display it at PORT1; otherwise, display the positive result. Execute the program with the following sets of data.

Set 1: (B) = 42H, (C) = 69H
Set 2: (B) = 69H, (C) = 42H
Set 3: (B) = F8H, (C) = 23H

Section 6.6: Debugging a Program

41. In Section 6.5.3, a program for controlling manufacturing processes is examined, all the errors are marked as 1 through 5. Rewrite the program with the errors corrected.

42. To test this program, substitute the instructions IN F1H and IN F2H by loading the two data bytes 97H and 85H in registers D and E. Rewrite the program to include other appropriate changes. Enter and execute the program on your system.

43. In the program presented in Section 6.5.3, assume that an LED indicator is connected to the output line D_7 of the port F3. Modify the program to turn on the LED when switch S_7 from port F1 is turned on, even if switch S_7' is off.

7

Programming Techniques with Additional Instructions

A computer is at its best, surpassing human capability, when it is asked to repeat such simple tasks as adding thousands of numbers. It does this accurately with electronic speed and without showing any signs of boredom. The programming techniques—such as looping, counting, and indexing—required for repetitious tasks are introduced in this chapter.

Data needed for repetitious tasks generally are stored in the system's R/W memory. The data must be transferred (copied) from memory to the microprocessor for manipulation (processing). The instructions related to data manipulations and data transfer (copy) between memory and the microprocessor are introduced in this chapter, as well as instructions related to 16-bit data and additional logic operations. Applications of these instructions are shown in five illustrative programs. The chapter concludes with a discussion of dynamic debugging techniques.

OBJECTIVES

- ☐ Draw a flowchart of a conditional loop illustrating indexing and counting.
- ☐ List the seven blocks of a generalized flowchart illustrating data acquisitions and data processing.
- ☐ Explain the functions of the 16-bit data transfer instructions LXI and of the arithmetic instructions INX and DCX.
- ☐ Explain the functions of memory-related data transfer instructions, and illustrate how a memory location is specified using the indirect and the direct addressing modes.
- ☐ Write a program to illustrate an application of instructions related to memory data transfer and 16-bit data.
- ☐ Explain the functions of arithmetic instructions related to data in memory: ADD/SUB M. Write a program to perform arithmetic operations that generate carry.

- ☐ Explain the functions and the differences between the four instructions: RLC, RAL, RRC, and RAR. Write a program to illustrate uses of these instructions.
- ☐ Explain the functions of the Compare instructions: CMP and CPI and the flags set under various conditions. Write a program to illustrate uses of the Compare instructions.
- ☐ Explain the term *dynamic debugging* and the debugging techniques: Single Step and Breakpoint.

7.1 PROGRAMMING TECHNIQUES: LOOPING, COUNTING, AND INDEXING

The programming examples illustrated in previous chapters are simple and can be solved manually. However, the computer surpasses manual efficiency when tasks must be repeated, such as adding a hundred numbers or transferring a thousand bytes of data. It is fast and accurate.

The programming technique used to instruct the microprocessor to repeat tasks is called **looping.** A loop is set up by instructing the microprocessor to change the sequence of execution and perform the task again. This process is accomplished by using Jump instructions. In addition, techniques such as counting and indexing (described below) are used in setting up a loop.

Loops can be classified into two groups:

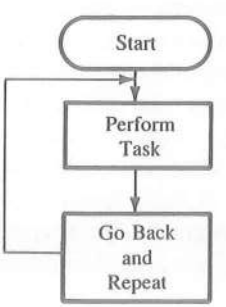
- ☐ Continuous loop—repeats a task continuously
- ☐ Conditional loop—repeats a task until certain data conditions are met

They are described in the next two sections.

7.1.1 Continuous Loop

A continuous loop is set up by using the unconditional Jump instruction shown in the flowchart (Figure 7.1).

FIGURE 7.1
Flowchart of a Continuous Loop



A program with a continuous loop does not stop repeating the tasks until the system is reset. Typical examples of such a program include a continuous counter (see Chapter 8, Section 8.2) or a continuous monitor system.

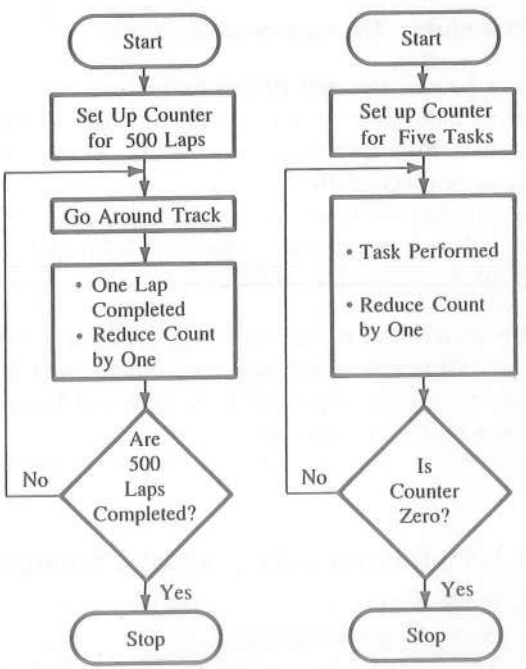
7.1.2 Conditional Loop

A conditional loop is set up by the conditional Jump instructions. These instructions check flags (Zero, Carry, etc.) and repeat the specified tasks if the conditions are satisfied. These loops usually include counting and indexing.

CONDITIONAL LOOP AND COUNTER

A counter is a typical application of the conditional loop. For example, how does the microprocessor repeat a task five times? The process is similar to that of a car racer in the Indy 500 going around the track 500 times. How does the racer know when 500 laps have been completed? The racing team manager sets up a counting and flagging method for the racer. This can be symbolically represented as in Figure 7.2(a). A similar approach is needed for the microprocessor to repeat the task five times. The microprocessor needs a counter, and when the counting is completed, it needs a flag. This can be accomplished with the conditional loop, as illustrated in the flowchart in Figure 7.2(b).

FIGURE 7.2
Flowcharts to Indicate Number of Repetitions Completed



(a) Laps in a Car Race (b) Task Repetitions

The computer flowchart of Figure 7.2(b) is translated into a program as follows:

1. Counter is set up by loading an appropriate count in a register.
2. Counting is performed by either incrementing or decrementing the counter.
3. Loop is set up by a conditional Jump instruction.
4. End of counting is indicated by a flag.

It is easier to count down to zero than to count up because the Zero flag is set when the register becomes zero. (Counting up requires the Compare instruction, which is introduced later.)

Conditional Loop, Counter, and Indexing Another type of loop includes indexing along with a counter. (*Indexing* means pointing or referencing objects with sequential numbers. In a library, books are arranged according to numbers, and they are referred to or sorted by numbers. This is called indexing.) Similarly, data bytes are stored in memory locations, and those data bytes are referred to by their memory locations.

Example 7.1

Illustrate the steps necessary to add ten bytes of data stored in memory locations starting at a given location, and display the sum. Draw a flowchart.

Procedure The microprocessor needs

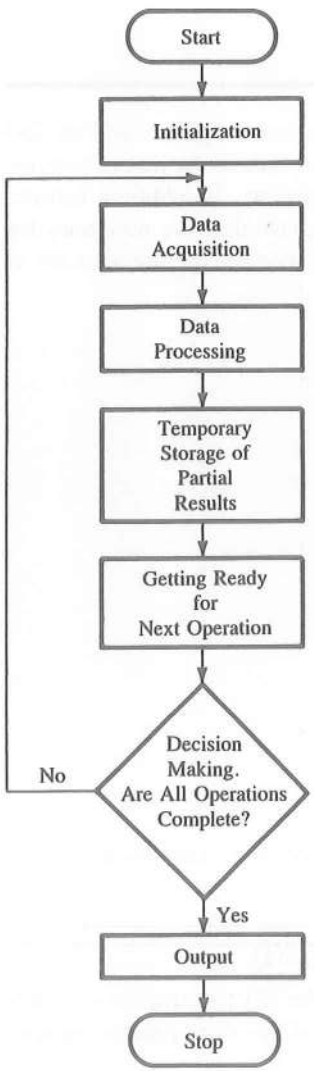
a. a counter to count 10 data bytes
b. an index or a memory pointer to locate where data bytes are stored
c. to transfer data from a memory location to the microprocessor (ALU)
d. to perform addition
e. registers for temporary storage of partial answers
f. a flag to indicate the completion of the task
g. to store or output the result

These steps can be represented in the form of a flowchart as in Figure 7.3.

This generalized flowchart can be used in solving many problems. Some blocks may have to be expanded with additional loops, or some blocks may need to be interchanged in their positions.

7.1.3 Review of Important Concepts

1. Programming is a logical approach to instruct the microprocessor to perform operations in a given sequence.
2. The computer is at its best in repeating tasks. It is fast and accurate.
3. Loops are set up by using the looping technique along with counting and indexing.
4. The computer is a versatile and powerful computing tool because of its capability to set up loops and to make decisions based on data conditions.



Block 1 **1.** This is a planning stage where the initial setting up is done. This is the most difficult step because the initial setup depends on the requirements of the following steps (steps a and b in Example 7.1: setting up counter and indexing).

Block 2 **2.** Data are generally stored in memory or read from an input port. This step is concerned with bringing data into the microprocessor (step c: transfer of data from a memory location to the microprocessor).

Block 3 **3.** This step involves all data manipulations (step d: addition).

Block 4 **4.** This step involves storing data of partial results (step e).

Block 5 **5.** At this point, the microprocessor does not know whether it has completed all the operations. Before it can repeat the task, it needs to get ready for the next operation; for example, point to next memory location or update count.

Block 6 **6.** This is a decision-making step; it decides whether to change the sequence of execution and repeat or to go to the next instruction (step f: a flag check).

Block 7 **7.** This step involves either sending the result to an output port or storing in memory (step g).

FIGURE 7.3
Generalized Programming Flowchart

LOOKING AHEAD

The programming techniques and the flowcharting introduced in this section will be illustrated with various applications throughout the chapter. Additional instructions necessary for these applications will be introduced first. The primary focus here is to analyze a given programming problem in terms of the basic building blocks of the flowchart shown in Figure 7.3.

See Questions and Programming Assignments 1–4 at the end of the chapter.

7.2 ADDITIONAL DATA TRANSFER AND 16-BIT ARITHMETIC INSTRUCTIONS

The instructions related to the data transfer among microprocessor registers and the I/O instructions were introduced in the last chapter; this section introduces the instructions related to the data transfer between the microprocessor and memory. In addition, instructions for some 16-bit arithmetic operations are included because they are necessary for using the programming techniques introduced earlier in this chapter. The opcodes are as follows:

1. Loading 16-bit data in register pairs
   LXI Rp: Load Register Pair Immediate
2. Data transfer (copy) from memory to the microprocessor
   MOV R,M: Move (from memory to register)
   LDAX B/D: Load Accumulator Indirect
   LDA 16-bit: Load Accumulator Direct
3. Data transfer (copy) from the microprocessor to memory
   MOV M,R: Move (from register to memory)
   STAX B/D: Store Accumulator Indirect
   STA 16-bit: Store Accumulator Direct
4. Loading 8-bit data directly in memory register (location)
   MVI M,8-bit: Load 8-bit data in memory
5. Incrementing/Decrementing Register Pair
   INX Rp: Increment Register Pair
   DCX Rp: Decrement Register Pair

The instructions related to these operations are illustrated with examples in the following sections.

7.2.1 16-Bit Data Transfer to Register Pairs (LXI)

The LXI instructions perform functions similar to those of the MVI instructions, except that the LXI instructions load 16-bit data in register pairs and the stack pointer register. These instructions do not affect the flags.

INSTRUCTIONS

Opcode	Operand	
LXI	Rp, 16-bit	Load Register Pair
LXI	B, 16-Bit	☐ This is a 3-byte instruction
LXI	D,16-bit	☐ The second byte is loaded in the low-order register of the register pair (e.g., register C)
LXI	H,16-bit	☐ The third byte is loaded in the high-order register pair (e.g., register B)

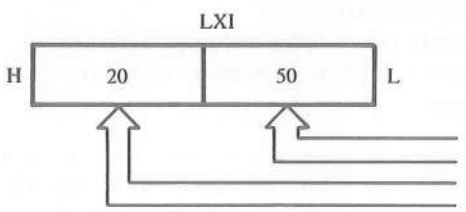
LXI	SP,16-bit	☐ There are four such instructions in the set as shown. The operands B, D, and H represent BC, DE, and HL registers, and SP represents the stack pointer register

Example 7.2

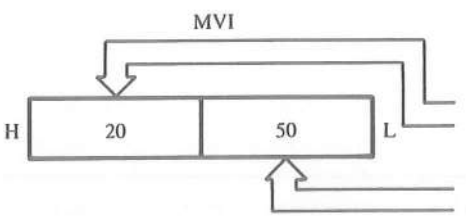
Write instructions to load the 16-bit number 2050H in the register pair HL using LXI and MVI opcodes, and explain the difference between the two instructions.

Instructions Figure 7.4 shows the register contents and the instructions required for Example 7.2.

The LXI instruction is functionally similar to two MVI instructions. The LXI instruction takes three bytes of memory and requires ten clock periods (T-states). On the other hand, two MVI instructions take four bytes of memory and require 14 clock periods (T-states).



Machine Code	Mnemonics	Comments
21	LXI H,2050H	;Load HL registers
50*		;50H in L register and
20		;20H in H register



26	MVI H,20H	;Load 20H in register H
20		
2E	MVI L,50H	;Load 50H in register L
50		

*NOTE: The order of the LXI machine code is reversed in relation to the mnemonics; low-order byte first followed by the high-order byte. This is similar to Jump instructions.

FIGURE 7.4
Instructions and Register Contents for Example 7.2

7.2.2 Data Transfer (Copy) from Memory to the Microprocessor

The 8085 instruction set includes three types of memory transfer instructions; two use the indirect addressing mode and one uses the direct addressing mode. These instructions do not affect the flags.

1. MOV R,M: Move (from Memory to Register)
   ☐ This is a 1-byte instruction

- ☐ It copies the data byte from the memory location into a register
- ☐ R represents microprocessor registers A, B, C, D, E, H, and L
- ☐ The memory location is specified by the contents of the HL register
- ☐ This specification of the memory location is indirect; it is called the indirect addressing mode

2. LDAX B/D: Load Accumulator Indirect
   - ☐ This is a 1-byte instruction
   - LDAX B ☐ It copies the data byte from the memory location into the accumulator
   - LDAX D ☐ The instruction set includes two instructions as shown
   - ☐ The memory location is specified by the contents of the registers BC or DE
   - ☐ The addressing mode is indirect
3. LDA 16-bit: Load Accumulator Direct
   - ☐ This is a 3-byte instruction
   - ☐ It copies the data byte from the memory location specified by the 16-bit address in the second and third byte
   - ☐ The second byte is a line number (low-order memory address)
   - ☐ The third byte is a page number (high-order memory address)
   - ☐ The addressing mode is direct

Example 7.3 The memory location 2050H holds the data byte F7H. Write instructions to transfer the data byte to the accumulator using three different opcodes: MOV, LDAX, and LDA.

Solution Figure 7.5 shows the register contents and the instructions required for Example 7.3. All of these three instructions copy the data byte F7H from the memory location 2050H to the accumulator.

In Figure 7.5(a), register HL is first loaded with the 16-bit number 2050H. The instruction MOV A,M uses the contents of the HL register as a memory pointer to location 2050H; this is the indirect addressing mode. The HL register is used frequently as a memory pointer because any instruction that uses M as an operand can copy from and into any one of the registers.

In Figure 7.5(b), the contents of register BC are used as a memory pointer to location 2050H by the instruction LDAX B. Registers BC and DE can be used as restricted memory pointers to copy the contents of only the accumulator into memory and vice versa; however, they cannot be used to copy the contents of other registers.

Figure 7.5(c) illustrates the direct addressing mode; the instruction LDA specifies the memory address 2050H directly as a part of its operand.

After examining all three methods, you may notice that the indirect addressing mode takes four bytes and the direct addressing mode takes three bytes. The question is: Why not just use the direct addressing mode?

If only one byte is to be transferred, the LDA instruction is more efficient. But for a block of memory transfer, the instruction LDA (three bytes) will have to be repeated for

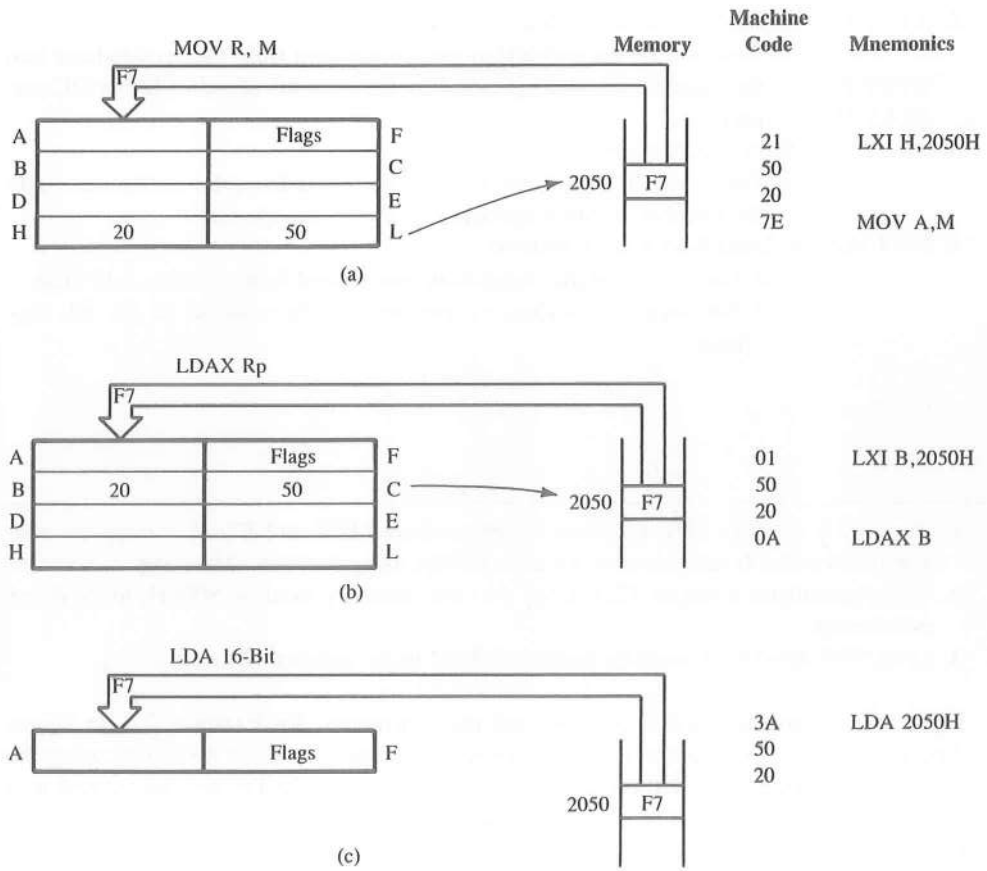


FIGURE 7.5
Instructions and Register Contents for Example 7.3

each memory. On the other hand, a loop can be set up with two other instructions, and the contents of a register pair can be incremented or decremented. This is further illustrated in Section 7.2.6.

7.2.3 Data Transfer (Copy) from the Microprocessor to Memory or Directly into Memory

The instructions for copying data from the microprocessor to a memory location are similar to those described in the previous section. These instructions are as follows:

1. MOV M,R: Move (from Register to Memory).
   - ☐ This is a 1-byte instruction that copies data from a register, R, into the memory location specified by the contents of HL registers

2. STAX B/D: Store Accumulator Indirect
   - ☐ This is a 1-byte instruction that copies data from the accumulator into the memory location specified by the contents of either BC or DE registers

   STAX B
   STAX D
3. STA 16-bit: Store Accumulator Direct
   - ☐ This is a 3-byte instruction that copies data from the accumulator into the memory location specified by the 16-bit operand.
4. MVI M,8-bit: Load 8-bit data in memory
   - ☐ This is a two-byte instruction; the second byte specifies 8-bit data
   - ☐ The memory location is specified by the contents of the HL register

---

Example 7.4

1. Register B contains 32H. Illustrate the instructions MOV and STAX to copy the contents of register B into memory location 8000H using indirect addressing.
2. The accumulator contains F2H. Copy (A) into memory location 8000H, using direct addressing.
3. Load F2H directly in memory location 8000H using indirect addressing.

Solution

Figure 7.6 shows the register contents and the instructions for Example 7.4. In Figure 7.6(a), the byte 32H is copied from register B into memory location 8000H by using the HL as a memory pointer. However, in Figure 7.6(b), where the DE register is used as a memory pointer, the byte 32H must be copied from B into the accumulator first because the instruction STAX copies only from the accumulator.

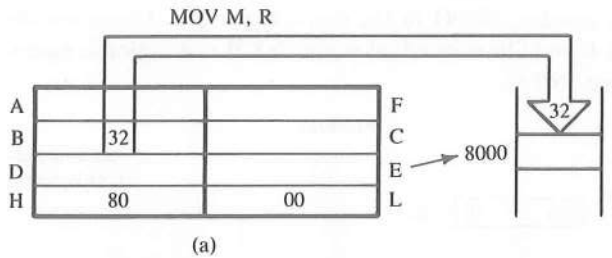
In Figure 7.6(c), the instruction STA copies 32H from the accumulator into the memory location 8000H. The memory address is specified as the operand; this is an illustration of the direct addressing mode. On the other hand, Figure 7.6(d) illustrates how to load a byte directly in memory location by using the HL as a memory pointer.

7.2.4 Arithmetic Operations Related to 16 Bits or Register Pairs

The instructions related to incrementing/decrementing 16-bit contents in a register pair are introduced below. These instructions do not affect flags.

1. INX Rp: Increment Register Pair
   - ☐ This is a 1-byte instruction
   - ☐ It treats the contents of two registers as one 16-bit number and increases the contents by 1
   - ☐ The instruction set includes four instructions, as shown

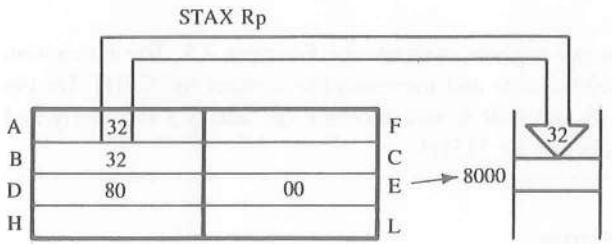
   INX B
   INX D
   INX H
   INX SP



(a)

Machine Code	Mnemonics
21	LXI H,8000H
00	
80	
70	MOV M,B

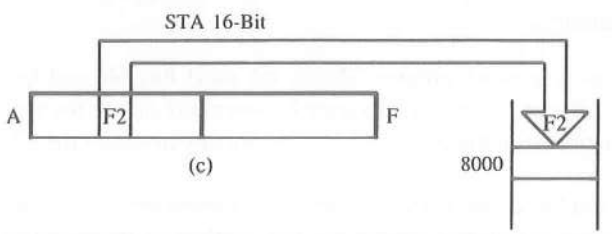
This instruction copies the contents of the accumulator into memory. Therefore, it is necessary first to copy (B) into A.



(b)

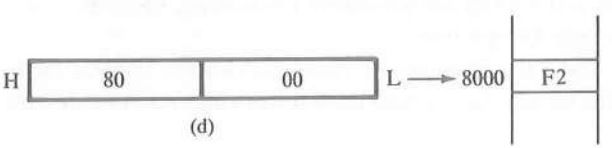
11	LXI D,8000H
00	
80	
78	MOV A,B
12	STAX D

This also requires the transfer of (B) to A.



(c)

32	STA 8000H
00	
80	



(d)

21	LXI H,8000H
00	
80	
36	MVI M,F2H
F2	

FIGURE 7.6
Instructions and Register Contents for Example 7.4

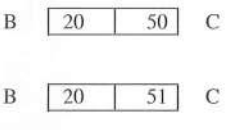
2. DCX Rp: Decrement Register Pair
   - ☐ This is a 1-byte instruction
   - ☐ It decreases the 16-bit contents of a register pair by 1
   - ☐ The instruction set includes four instructions, as shown

DCX B
DCX D
DCX H
DCX SP

Example 7.5

Write the instruction to load the number 2050H in the register pair BC. Increment the number using the instruction INX B and illustrate whether the INX B instruction is equivalent to the instructions INR B and INR C.

FIGURE 7.7
Instructions and Register Contents for Example 7.5



Machine Code	Mnemonics
01	LXI B,2050H
50	
20	
03	INX B

Solution

Figure 7.7 shows the instructions and register contents for Example 7.5. The instruction INX B views 2050H as one 16-bit number and increases the number to 2051H. On the other hand, the instructions INR B and INR C will increase (B) and (C) separately and the contents of the BC register pair will be 2151H.

7.2.5 Review of Instructions

In this section, we examined primarily how to copy data from the microprocessor into memory and vice versa. The 8085 instruction provides three methods of copying data between the microprocessor and memory:

1. Indirect addressing using HL as a memory pointer: This is the most flexible and frequently used method. The HL register can be used to copy between any one of the registers and memory. Any instruction with the operand M automatically assumes the HL is the memory pointer.
2. Indirect addressing using BC and DE as memory pointers: The instructions LDAX and STAX use BC and DE as memory pointers. However, this method is restricted to copying from and into the accumulator and cannot be used for other registers. In addition, the mnemonics (LDAX and STAX) are somewhat misleading; therefore, careful attention must be given to their interpretion.
3. Direct addressing using LDA and STA instructions: These instructions include memory address as the operand. This method is also restricted to copying from and into the accumulator.

In addition to the above data copy instructions, we discussed two instructions, INX and DCX, concerning the register pairs. The critical feature of these instructions is that they do not affect the flags.

7.2.6 Illustrative Program: Block Transfer of Data Bytes

PROBLEM STATEMENT

Sixteen bytes of data are stored in memory locations at XX50H to XX5FH. Transfer the entire block of data to new memory locations starting at XX70H.

Data(H) 37, A2, F2, 82, 57, 5A, 7F, DA, E5, 8B, A7, C2, B8, 10, 19, 98

PROBLEM ANALYSIS

The problem can be analyzed in terms of the blocks suggested in the flowchart (Figure 7.8). The steps are as follows:

The flowchart in Figure 7.8 includes five blocks; these blocks are identified with numbers referring to the blocks in the generalized flowchart in Figure 7.3. This problem is not concerned with data manipulation (processing); therefore, the flowchart does not require Blocks 3 and 4 (data processing and temporary storage of partial results). The problem simply deals with the transferring of the data bytes from one location to another location in memory; therefore, the Store Data Byte block is equivalent to the Output block in the generalized flowchart.

Block 1 is the initialization block; this block sets up two memory pointers and one counter. Block 5 is concerned with updating the memory pointers and the counter. The

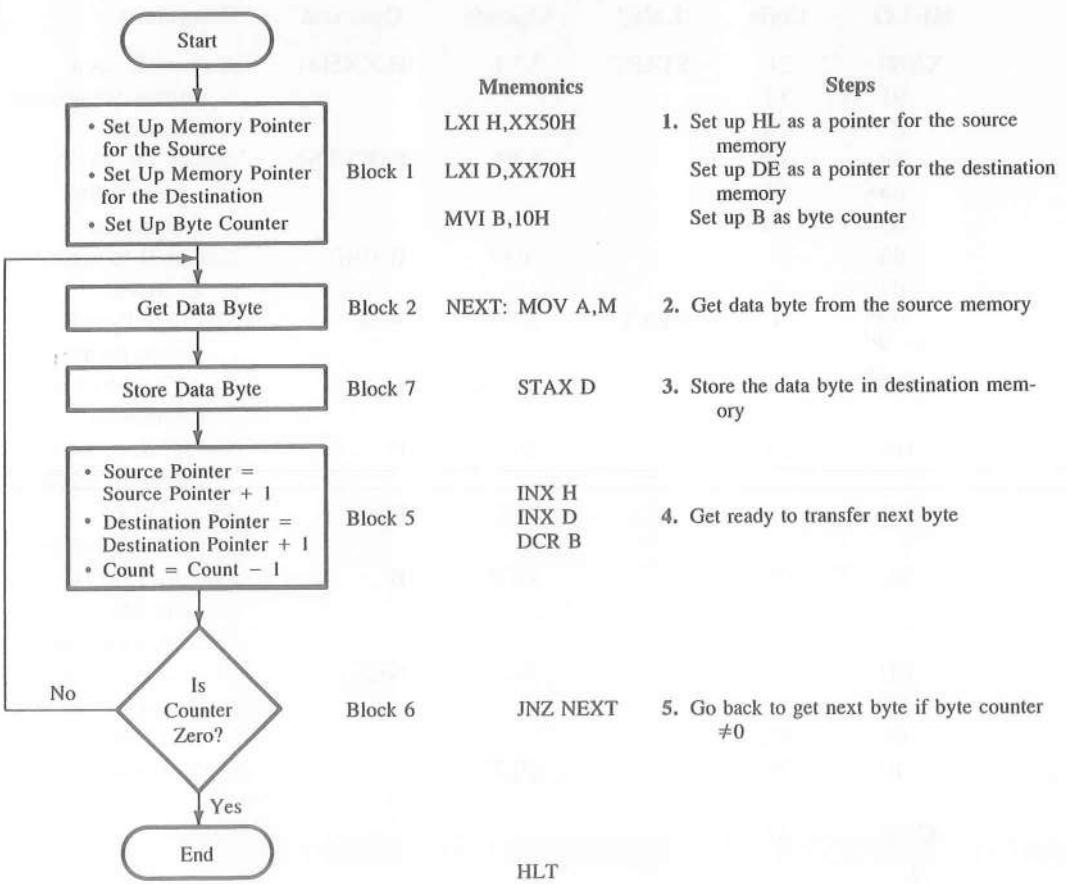


FIGURE 7.8
Flowchart for Block Transfer of Data Bytes

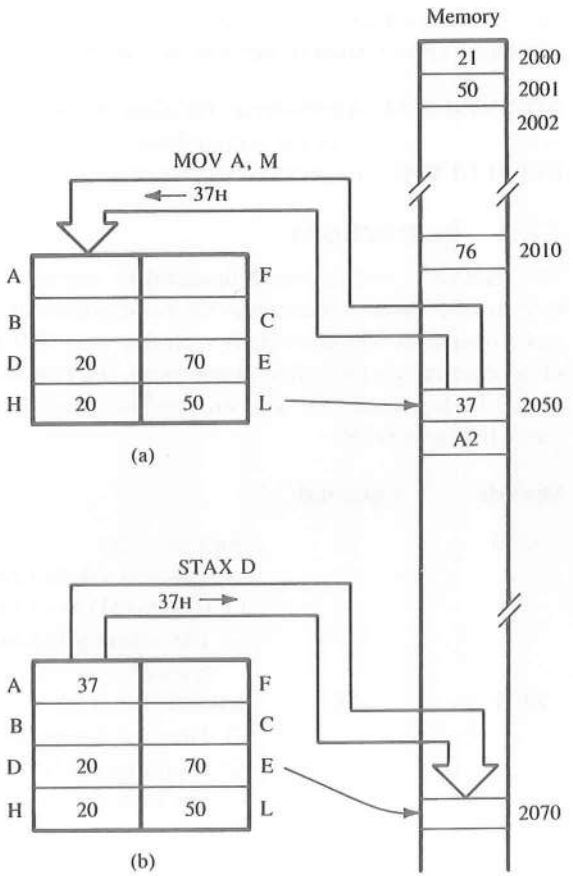
statements shown in the block appear strange if they are read as algebraic equations; however, they are not algebraic equations. The statement Pointer = Pointer + 1 means the new value is obtained by incrementing the previous value by one.

The statements in the flowchart correspond one-to-one with the mnemonics. In large programs, such details in the flowchart are impractical as well as undesirable. However, these details are included here to show the logic flow in writing programs. In Figure 7.8, some of the details can be eliminated very easily from the flowchart. For example, Blocks 2 and 7 can be combined in one statement; such as, Transfer Data Byte from Source to Destination. Similarly, Block 5 can be reduced to one statement; such as, Update memory Pointers and Counter.

PROGRAM

Memory Address HI-LO	Hex Code	Label	Instructions Opcode	Operand	Comments
XX00	21	START:	LXI	H,XX50H	;Set up HL as a
01	50				; pointer for source
02	XX				; memory
03	11		LXI	D,XX70H	;Set up DE as
04	70				; a pointer for
05	XX				; destination
06	06		MVI	B,10H	;Set up B to count
07	10				; 16 bytes
08	7E	NEXT:	MOV	A,M	;Get data byte from
					; source memory
09	12		STAX	D	;Store data byte at
					; destination
0A	23		INX	H	;Point HL to next
					; source location
0B	13		INX	D	;Point DE to
					; next destination
0C	05		DCR	B	;One transfer is
					; complete,
					; decrement count
0D	C2		JNZ	NEXT	;If counter is not 0,
0E	08				; go back to transfer
0F	XX				; next byte
10	76		HLT		;End of program
XX50	37				;Data
↓	↓				
XX5F	98				

FIGURE 7.9
Data Transfer from Memory to Accumulator (a), Then to New Memory Location (b)



PROGRAM EXECUTION AND OUTPUT

To execute the program, substitute the page number of your system's R/W memory in place of XX, enter the program and the data, and execute it. To verify the proper execution, check the memory locations from XX70H to XX7FH.

Let us assume the system R/W user memory starts at 2000H. Figure 7.9(a) shows how the contents of the memory location 2050H are copied into the accumulator by the instruction MOV A,M; the HL register points to location 2050 and instruction MOV A,M copies 37H into A. Figure 7.9(b) shows that the DE register points to the location 2070H and the instruction STAX D copies (A) into the location 2070H.

See Questions and Programming Assignments 5–21 at the end of this chapter.

7.3 ARITHMETIC OPERATIONS RELATED TO MEMORY

In the last chapter, the arithmetic instructions concerning three arithmetic tasks—Add, Subtract, and Increment/Decrement—were introduced. These instructions dealt with

microprocessor register contents or numbers. In this chapter, instructions concerning the arithmetic tasks related to memory will be introduced:

ADD M/SUB M: Add/Subtract the contents of a memory location to/from the contents of the accumulator.
INR M/DCR M: Increment/Decrement the contents of a memory location.

7.3.1 Instructions

The arithmetic instructions referenced to memory perform two tasks: one is to copy a byte from a memory location to the microprocessor, and the other is to perform the arithmetic operation. These instructions (other than INR and DCR) implicitly assume that one of the operands is (A); after an operation, the previous contents of the accumulator are replaced by the result. All flags are modified to reflect the data conditions (see the exceptions: INR and DCR).

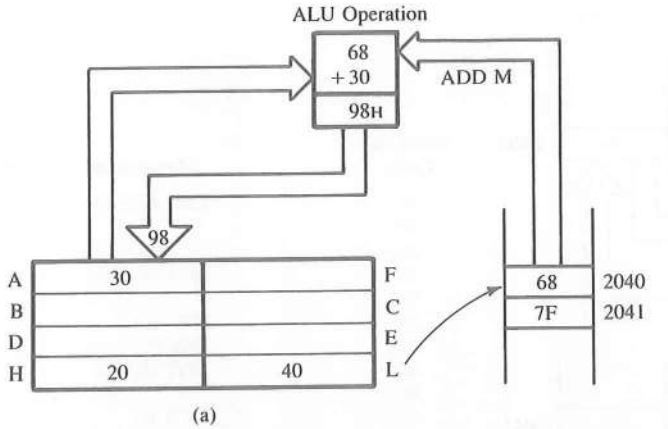
Opcode	Operand	
ADD	M	Add Memory ☐ This is a 1-byte instruction ☐ It adds (M) to (A) and stores the result in A ☐ The memory location is specified by the contents of HL register
SUB	M	Subtract Memory ☐ This is a 1-byte instruction ☐ It subtracts (M) from (A) and stores the result in A ☐ The memory location is specified by (HL)
INR	M	This is a 1-byte instruction ☐ It increments the contents of a memory location by 1, not the memory address ☐ The memory location is specified by (HL) ☐ All flags except the Carry flag are affected
DCR	M	This is a 1-byte instruction ☐ It decrements (M) by 1 ☐ The memory location is specified by (HL) ☐ All flags except the Carry flag are affected

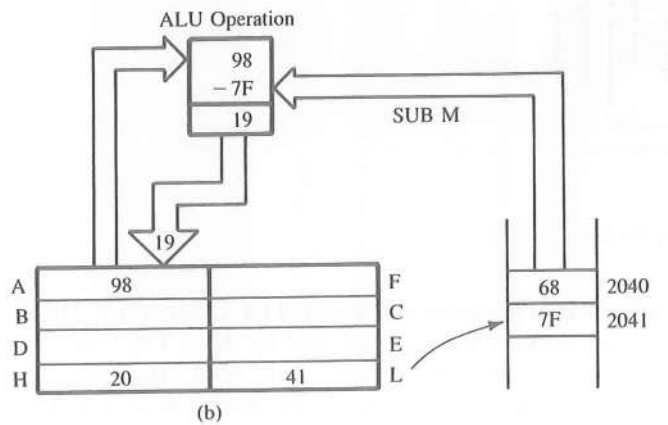
Example 7.6

Write instructions to add the contents of the memory location 2040H to (A), and subtract the contents of the memory location 2041H from the first sum. Assume the accumulator has 30H, the memory location 2040H has 68H, and the location 2041H has 7FH.

Solution

Before asking the microprocessor to perform any memory-related operations, we must specify the memory location by loading the HL register pair. In the example illustrated in Figure 7.10, the contents of the HL pair 2040H specify the memory location. The instruction ADD M adds 68H, the contents of memory location 2040H, to the contents of





Machine Code	Mnemonics
21	
40	
20	LXI H,2040H
86	ADD M
23	INX H
96	SUB M

FIGURE 7.10
Register and Memory Contents and Instructions for Example 7.6

the accumulator (30H). The instruction INX H points to the next memory location, 2041H, and the instruction SUB M subtracts the contents (7FH) of memory location 2041H from the previous sum.

Example 7.7

Write instructions to

1. load 59H in memory location 2040H, and increment the contents of the memory location.
2. load 90H in memory location 2041H, and decrement the contents of the memory location.

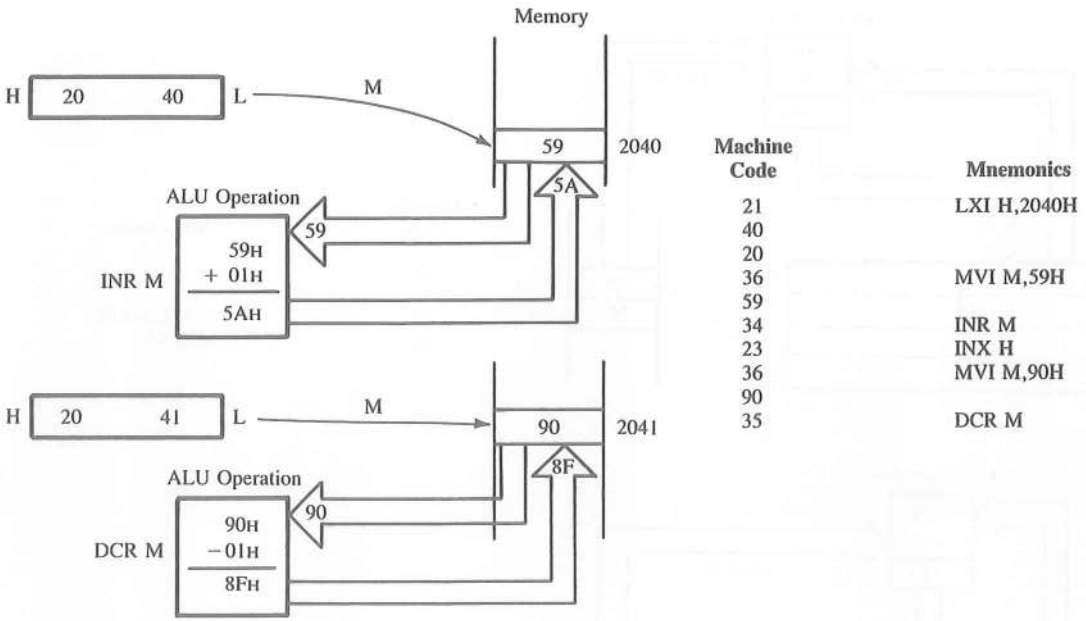


FIGURE 7.11
Register and Memory Contents, and Instructions for Example 7.7

Solution Figure 7.11 shows register contents and the instructions required for Example 7.7. The instruction MVI M loads 59H in the memory location indicated by (HL). The instruction INR M increases the contents, 59H, of the memory location to 5AH. The instruction INX H increases (HL) to 2041H. The next two instructions load and decrement 90H.

7.3.2 Illustrative Program: Addition with Carry

PROBLEM STATEMENT

Six bytes of data are stored in memory locations starting at XX50H. Add all the data bytes. Use register B to save any carries generated, while adding the data bytes. Display the entire sum at two output ports, or store the sum at two consecutive memory locations, XX70H and XX71H.

Data(H) A2, FA, DF, E5, 98, 8B

PROBLEM ANALYSIS

This problem can be analyzed in relation to the general flowchart in Figure 7.3 as follows:

1. Because of the memory-related arithmetic instructions just introduced in this section, two blocks in the general flowchart—data acquisition and data processing—can be combined in one instruction.

2. The fourth block—temporary storage of partial results—is unnecessary because the sum can be stored in the accumulator.
3. The data processing block needs to be expanded to account for carry.

In the first block (Block 1) of the flowchart in Figure 7.12, the accumulator and the carry register (for example, register B) must be cleared in order to use them for arithmetic operations; otherwise, residual data will cause erroneous results.

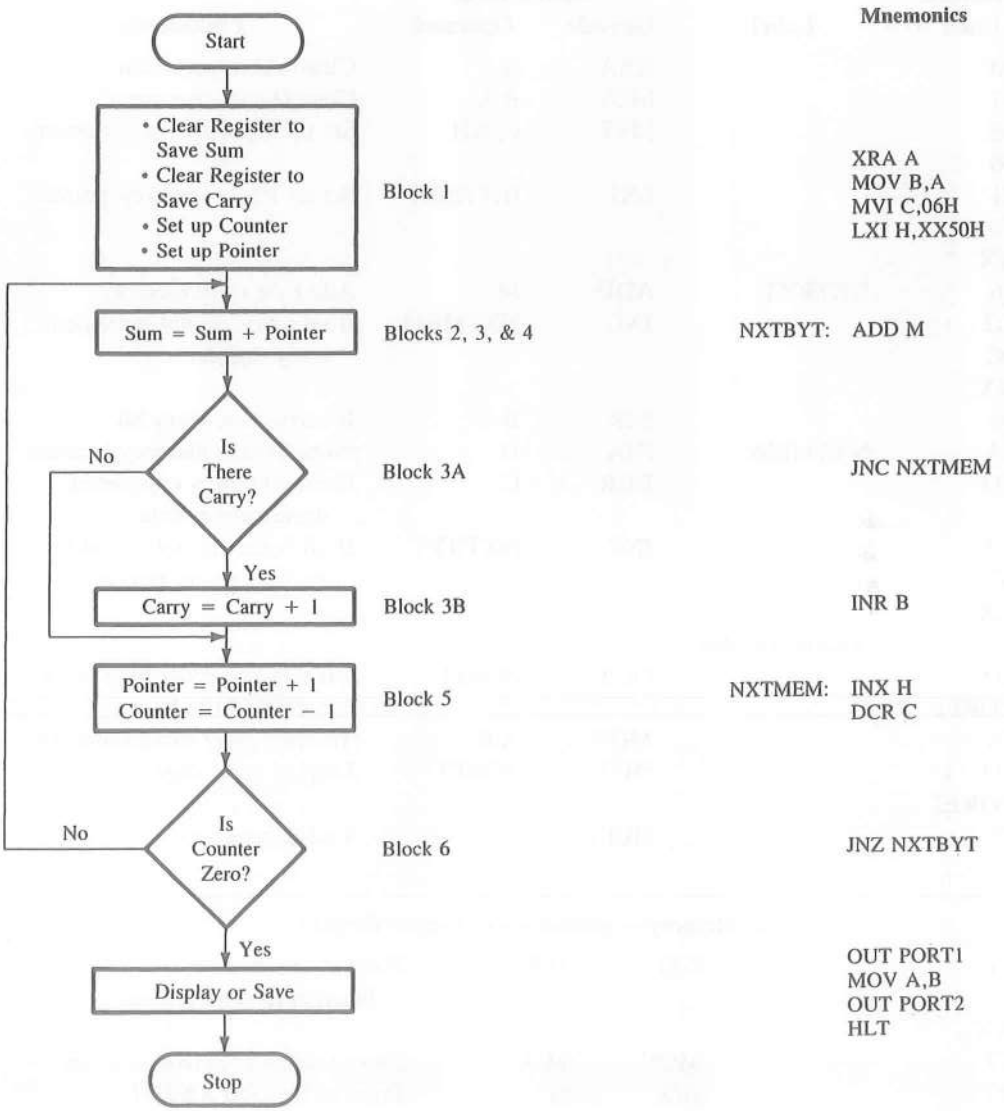


FIGURE 7.12
Flowchart for Addition with Carry

After the addition, it is necessary to check whether that operation has generated a carry (Block 3A). If a carry is generated, the carry register is incremented by one (Block 3B); otherwise, it is bypassed. The instruction ADC (Add with Carry) is inappropriate for this operation. (See Appendix F for the description of the instruction ADC.)

PROGRAM

Memory Address HI-LO	Machine Code	Label	Instructions Opcode	Operand	Comments
XX00	AF		XRA	A	;Clear (A) to save sum
01	47		MOV	B,A	;Clear (B) to save carry
02	0E		MVI	C,06H	;Set up register C as a counter
03	06				
04	21		LXI	H,XX50H	;Set up HL as memory pointer
05	50				
06	XX				
07	86	NXTBYT:	ADD	M	;Add byte from memory
08	D2		JNC	NXTMEM	;If no carry, do not increment
09	0C				; carry register
0A	XX				
0B	04		INR	B	;If carry, save carry bit
0C	23	NXTMEM:	INX	H	;Point to next memory location
0D	0D		DCR	C	;One addition is completed;
					; decrement counter
0E	C2		JNZ	NXTBYT	;If all bytes are not yet added,
0F	07				; go back to get next byte
10	XX				
		;Output Display			
11	D3		OUT	PORT1	;Display low-order byte of the
12	PORT1				; sum at PORT1
13	78		MOV	A,B	;Transfer carry to accumulator
14	D3		OUT	PORT2	;Display carry digits
15	PORT2				
16	76		HLT		;End of program

;Storing in Memory—Alternative to Output Display

11	21	LXI	H,XX70H	;Point to the memory
12	70			; location to store answer
13	XX			
14	77	MOV	M,A	;Store low-order byte at XX70H
15	23	INX	H	;Point to location XX71H
16	70	MOV	M,B	;Store carry bits
17	76	HLT		;End of program

```
50    A2        ;Data Bytes
51    FA
52    DF
53    E5
54    98
55    8B
```

PROGRAM DESCRIPTION AND OUTPUT

In this program, register B is used as a carry register, register C as a counter to count six data bytes, and the accumulator to add the data bytes and save the partial sum.

After the completion of the summation, the high-order byte (bits higher than eight bits) of the sum is saved in register B and the low-order byte is in the accumulator. Both are displayed at two different ports, or they can be stored at the memory locations XX70H and 71H.

See Questions and Programming Assignments 22–31 at the end of this chapter.

7.4 LOGIC OPERATIONS: ROTATE

In the last chapter, the logic instructions concerning the four operations AND, OR, Ex-OR, and NOT were introduced. This chapter introduces instructions related to rotating the accumulator bits. The opcodes are as follows:

- ☐ RLC: Rotate Accumulator Left
- ☐ RAL: Rotate Accumulator Left Through Carry
- ☐ RRC: Rotate Accumulator Right
- ☐ RAR: Rotate Accumulator Right Through Carry

7.4.1 Instructions

This group has four instructions; two are for rotating left and two are for rotating right. The differences between these instructions are illustrated in the following examples.

1. RLC: Rotate Accumulator Left
   - ☐ Each bit is shifted to the adjacent left position. Bit D_7 becomes D_0.
   - ☐ CY flag is modified according to bit D_7.

Example 7.8

Assume the accumulator contents are AAH and CY = 0. Illustrate the accumulator contents after the execution of the RLC instruction twice.

Solution

Figure 7.13 shows the contents of the accumulator and the CY flag after the execution of the RLC instruction twice. The first RLC instruction shifts each bit to the left by one position, places bit D_7 in bit D_0 and sets the CY flag because $D_7 = 1$. The accumulator byte AAH becomes 55H after the first rotation. In the second rotation, the byte is again AAH, and the CY flag is reset because bit D_7 of 55H is 0.

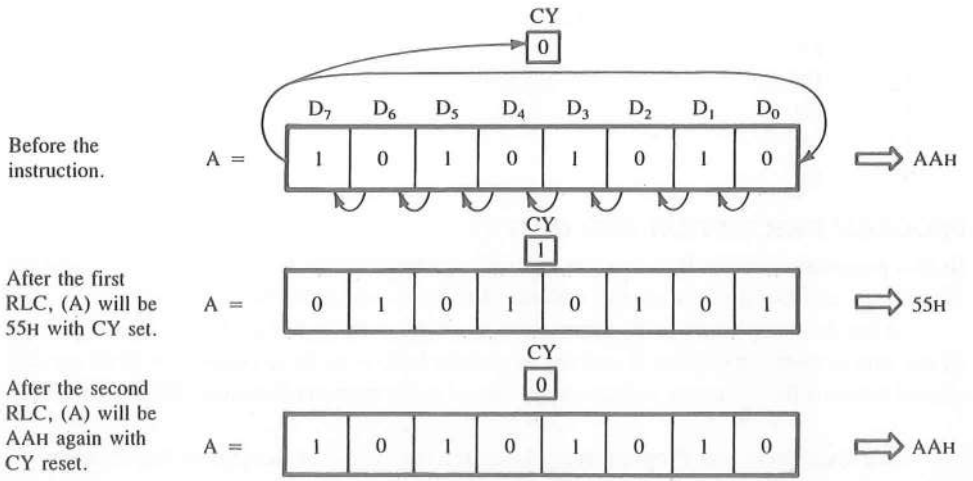


FIGURE 7.13
Accumulator Contents after RLC

2. RAL: Rotate Accumulator Left Through Carry
   - ☐ Each bit is shifted to the adjacent left position. Bit D_7 becomes the carry bit and the carry bit is shifted into D_0.
   - ☐ The Carry flag is modified according to bit D_7.

Example 7.9 Assume the accumulator contents are AAH and CY = 0. Illustrate the accumulator contents after the execution of the instruction RAL twice.

Solution Figure 7.14 shows the contents of the accumulator and the CY flag after the execution of the RAL instruction twice. The first RAL instruction shifts each bit to the left by one position, places bit D_7 in the CY flag, and the CY bit in bit D_0. This is a 9-bit rotation; CY is assumed to be the ninth bit of the accumulator. The accumulator byte AAH becomes 54H after the first rotation. In the second rotation, the byte becomes A9H, and the CY flag is reset.

Examining these two examples, you may notice that the primary difference between these two instructions is that (1) the instruction RLC rotates through eight bits, and (2) the instruction RAL rotates through nine bits.

3. RRC: Rotate Accumulator Right
   - ☐ Each bit is shifted right to the adjacent position. Bit D_0 becomes D_7.
   - ☐ The Carry flag is modified according to bit D_0.
4. RAR: Rotate Accumulator Right Through Carry
   - ☐ Each bit is shifted right to the adjacent position. Bit D_0 becomes the carry bit, and the carry bit is shifted into D_7.

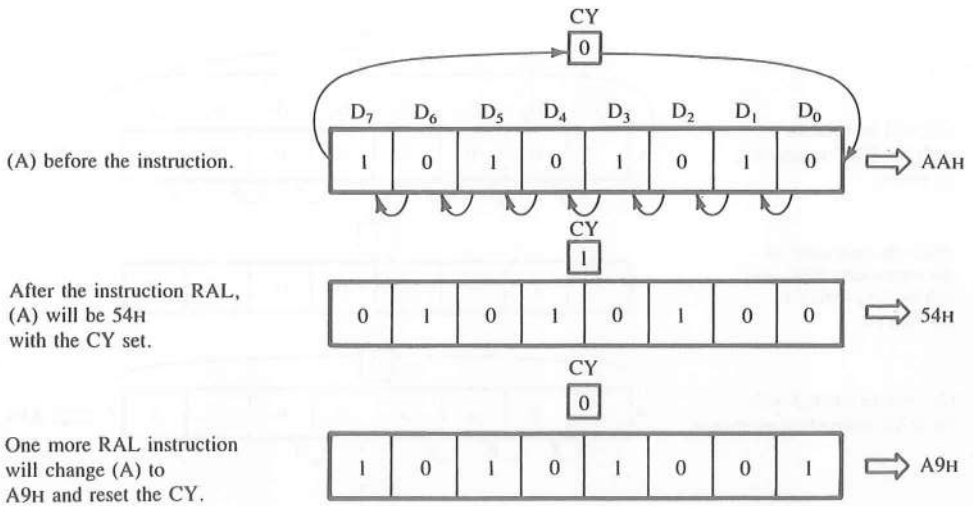


FIGURE 7.14
Accumulator Contents after RAL

Example 7.10

Assume the contents of the accumulator are 81H and CY = 0. Illustrate the accumulator contents after the RRC and RAR instructions.

Solution

Figure 7.15 shows the changes in the contents of the accumulator (81H) when the RRC instruction is used and when the RAR instruction is used. The 8-bit rotation of the RRC instruction changes 81H into C0H, and the 9-bit rotation of the RAR instruction changes 81H into 40H.

APPLICATIONS OF ROTATE INSTRUCTIONS

The rotate instructions are primarily used in arithmetic multiply and divide operations and for serial data transfer.

For example, if (A) is 0000 1000 = 08H,

- ☐ By rotating 08H right: (A) = 0000 0100 = 04H
  This is equivalent to dividing by 2
- ☐ By rotating 08H left: (A) = 0001 0000 = 10H
  This is equivalent to multiplying by 2 (10H = 16_{10})

However, these procedures are invalid when logic 1 is rotated left from D_7 to D_0 or vice versa. For example, if 80H is rotated left, it becomes 01H. Applications of serial data transfer are discussed in Chapter 16.

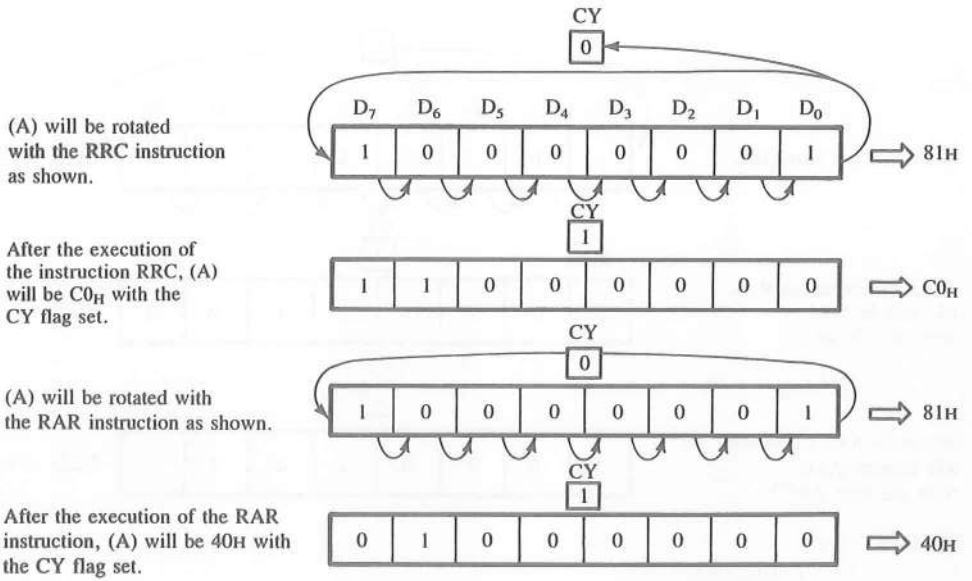


FIGURE 7.15
Rotate Right Instructions

7.4.2 Illustrative Program: Checking Sign with Rotate Instructions

PROBLEM STATEMENT

A set of ten current readings is stored in memory locations starting at XX60H. The readings are expected to be positive ($<127_{10}$). Write a program to

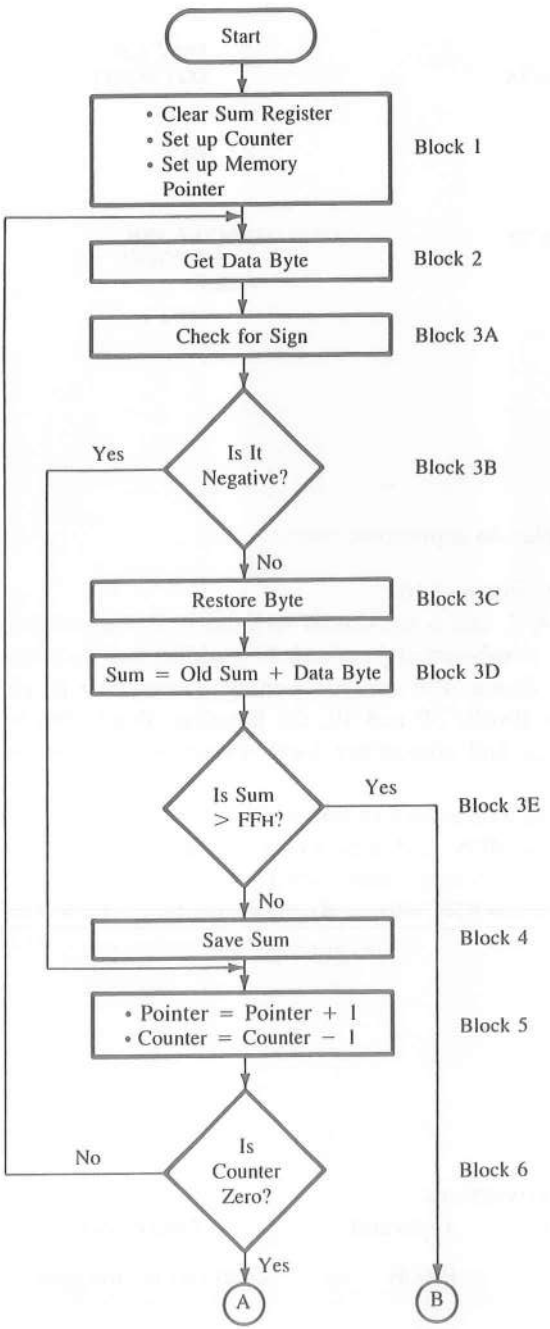
1. check each reading to determine whether it is positive or negative.
2. reject all negative readings.
3. add all positive readings.
4. output FFH to PORT1 at any time when the sum exceeds eight bits to indicate overload; otherwise, display the sum. If no output port is available in the system, go to step 5.
5. store FFH in the memory location XX70H when the sum exceeds eight bits; otherwise, store the sum.

Data(H) 28, D8, C2, 21, 24, 30, 2F, 19, F2, 9F

PROBLEM ANALYSIS

This problem can be divided into the following steps:

1. Transfer a data byte from the memory location to the microprocessor, and check whether it is a negative number. The sign of the number can be verified by rotating bit D_7 into the Carry position and checking for CY. (See Assignment 38 at the end of this chapter to verify the sign of a number with the Sign flag.)
2. If it is negative, reject the data and get the next data byte.



Mnemonics

```
        MVI B,00H
        MVI C,0AH
        LXI H,XX60H
NEXT:   MOV A,M
        RAL
        JC REJECT
        RAR
        ADD B
        JC OVRLOD
        MOV B,A
REJECT: INX H
        DCR C
        JNZ NEXT
```

FIGURE 7.16
Flowchart for Checking Sign with Rotate Instructions (*continued on next page*)

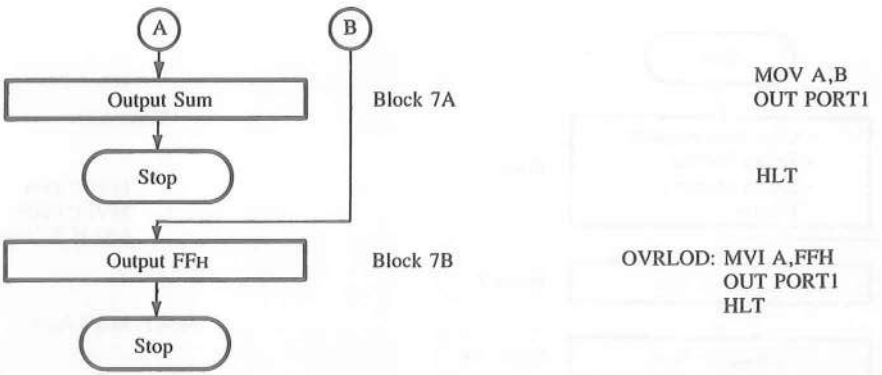


FIGURE 7.16
(continued)

3. If it is positive, add the data byte.
4. Check the sum for a carry and display an appropriate output.

The steps are shown in the flowchart in Figure 7.16.

In this flowchart, Blocks 1, 2, 4, 5, and 6 are similar to those in the generalized flowchart. Block 3—data processing—is substantially expanded by adding two decision-making blocks. This flowchart is first drawn with decision-making answers that do not change the sequence. For example, in Blocks 3B and 3E, the flowchart should first be continued with the answers NO. Then find appropriate locations where the program should be directed to the answers YES.

In this program, the sign of a data byte cannot be checked with the instruction JM (Jump On Minus) because the instruction MOV A,M does not set any flags. (However, the flags can be set by ORing the accumulator contents with itself.)

Similarly, in Block 3C, the instruction RRC (Rotate Right) cannot be used when the instruction RAL is used to rotate left. However, the program can be written using RLC and RRC in both places (3A and 3C).

PROGRAM

Memory Address HI-LO	Machine Code	Label	Instructions Opcode	Operand	Comments
XX00	06		MVI	B,00H	;Clear (B) to save sum
01	00				
02	0E		MVI	C,0AH	;Set up register C
					; as a counter

```
03      0A
04      21                      LXI     H,XX60H     ;Set up HL as memory
                                                    ;  pointer
05      60
06      XX
07      7E          NEXT:       MOV     A,M         ;Get byte
08      17                      RAL                 ;Shift D7 into CY
09      DA                      JC      REJECT      ;If D7 = 1, reject byte and
0A      12                                          ;  go to increment
                                                    ;  pointer
0B      XX
0C      1F                      RAR                 ;If byte is positive, re-
                                                    ;  store it
0D      80                      ADD     B           ;Add previous sum to (A)
0E      DA                      JC      OVRLOD      ;If sum >FFH, it is over-
                                                    ;  load;
0F      1C                                          ; turn on emergency
10      XX
11      47                      MOV     B,A         ;Save sum
12      23          REJECT:     INX     H           ;Point to next reading
13      0D                      DCR     C           ;One reading is checked;
                                                    ;  decrement counter
14      C2                      JNZ     NEXT        ;If all readings are not
15      07                                          ;  checked, go back to
                                                    ;  transfer next byte
16      XX
                ;Output Display Section
17      78                      MOV     A,B
18      D3                      OUT     PORT1       ;Display sum
19      PORT1
1A      76                      HLT                 ;End of program
1B      00                      NOP                 ;To match Jump location,
                                                    ;  OVRLOD in memory
                                                    ;  storage
1C      3E          OVRLOD:     MVI     A,FFH       ;It is an overload
1D      FF
1E      D3                      OUT     PORT1       ;Display overload signal
1F      PORT1                                       ;  at PORT1
20      76                      HLT
```

```
                    ;Storing Result in Memory—Alternative to Output Display
18      32                      STA     XX70H       ;Store sum in memory
                                                    ;  XX70H
19      70
1A      XX
1B      76
1C      3E          OVRLOD:     MVI     A,FFH       ;Store overload signal in
1D      FF                                          ;  memory XX70
1E      32                      STA     XX70H
1F      70
20      XX
21      76                      HLT
```

```
XX60    28      ;Current Readings
  61    D8
  62    C2
  63    21
  64    24
  65    30
  66    2F
  67    19
  68    F2
  69    9F
```

PROGRAM DESCRIPTION AND OUTPUT

In this program, register C is used as a counter to count ten bytes. Register B is used to save the sum. The sign of the number is checked by verifying whether D_7 is 1 or 0. If the Carry flag is set to indicate the negative sign, the program rejects the number and goes to Block 5, Getting Ready for Next Operation.

The program should reject the data bytes D8, C2, F2, and 9F, and should add the rest. The answer displayed should be E5.

See Questions and Programming Assignments 32–40 at the end of this chapter.

7.5 LOGIC OPERATIONS: COMPARE

The 8085 instruction set has two types of Compare operations: CMP and CPI.

- ☐ CMP: Compare with Accumulator
- ☐ CPI: Compare Immediate (with Accumulator)

The microprocessor compares a data byte (or register/memory contents) with the contents of the accumulator by subtracting the data byte from (A), and indicates

whether the data byte is ≥\≤ (A) by modifying the flags. However, the contents are not modified.

7.5.1 Instructions

1. CMP R/M: Compare (Register or Memory) with Accumulator
   - ☐ This is a 1-byte instruction.
   - ☐ It compares the data byte in register or memory with the contents of the accumulator.
   - ☐ If (A) < (R/M), the CY flag is set and the Zero flag is reset.
   - ☐ If (A) = (R/M), the Zero flag is set and the CY flag is reset.
   - ☐ If (A) > (R/M), the CY and Zero flags are reset.
   - ☐ When memory is an operand, its address is specified by (HL).
   - ☐ No contents are modified; however, all remaining flags (S, P, AC) are affected according to the result of the subtraction.
2. CPI 8-bit: Compare Immediate with Accumulator
   - ☐ This is a 2-byte instruction, the second byte being 8-bit data.
   - ☐ It compares the second byte with (A).
   - ☐ If (A) < 8-bit data, the CY flag is set and the Zero flag is reset.
   - ☐ If (A) = 8-bit data, the Zero flag is set, and the CY flag is reset.
   - ☐ If (A) > 8-bit data, the CY and Zero flags are reset.
   - ☐ No contents are modified; however, all remaining flags (S, P, AC) are affected according to the result of the subtraction.

Example 7.11

Write an instruction to load the accumulator with the data byte 64H, and verify whether the data byte in memory location 2050H is equal to the accumulator contents. If both data bytes are equal, jump to memory location BUFFER.

Solution

Figure 7.17 illustrates Example 7.11.

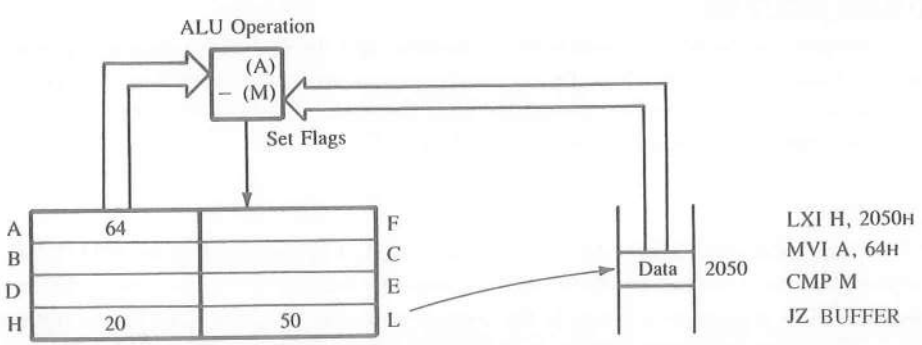


FIGURE 7.17
Compare Instructions

In these instructions, the instruction CMP M selects the memory location pointed out by the HL register (2050H) and compares the contents of that location with (A). If they are equal, the Zero flag is set, and the program jumps to location BUFFER.

Let us assume the data byte in memory location 2050H is 9AH. The microprocessor compares 64H with 9AH by subtracting 9AH from 64H as shown below.

(A) = 64H	0110 0100
	+
2's Complement of 9AH	0110 0110
CY	0 1100 1010
Complement CY	1 1100 1010

The result CAH will modify the flags as S = 1, Z = 0, and CY = 1; however, the original byte in the accumulator 64H will not be changed.

7.5.2 Illustrative Program: Use of Compare Instruction to Indicate End of Data String

PROBLEM STATEMENT

A set of current readings is stored in memory locations starting at XX50H. The end of the data string is indicated by the data byte 00H. Add the set of readings. The answer may be larger than FFH. Display the entire sum at PORT1 and PORT2 or store the answer in the memory locations XX70 and XX71H.

Data(H) 32, 52, F2, A5, 00

PROBLEM ANALYSIS

In this problem, the number of data bytes is variable, and the end of the data string is indicated by loading 00H. Therefore, the counter technique will not be useful to indicate the end of the readings. However, by comparing each data byte with 00H, the end of the data can be determined. This is shown in the flowchart in Figure 7.18.

FLOWCHART

The flowchart shows that after a data byte is transferred, it is first checked for 00H (Block 6). This operation is similar to checking for a negative number in the previous problem. However, this is also an exit point. If the byte is zero, the program goes to the output Block 7. The other significant change is in Block 5 where the unconditional Jump brings the sequence back into the program.

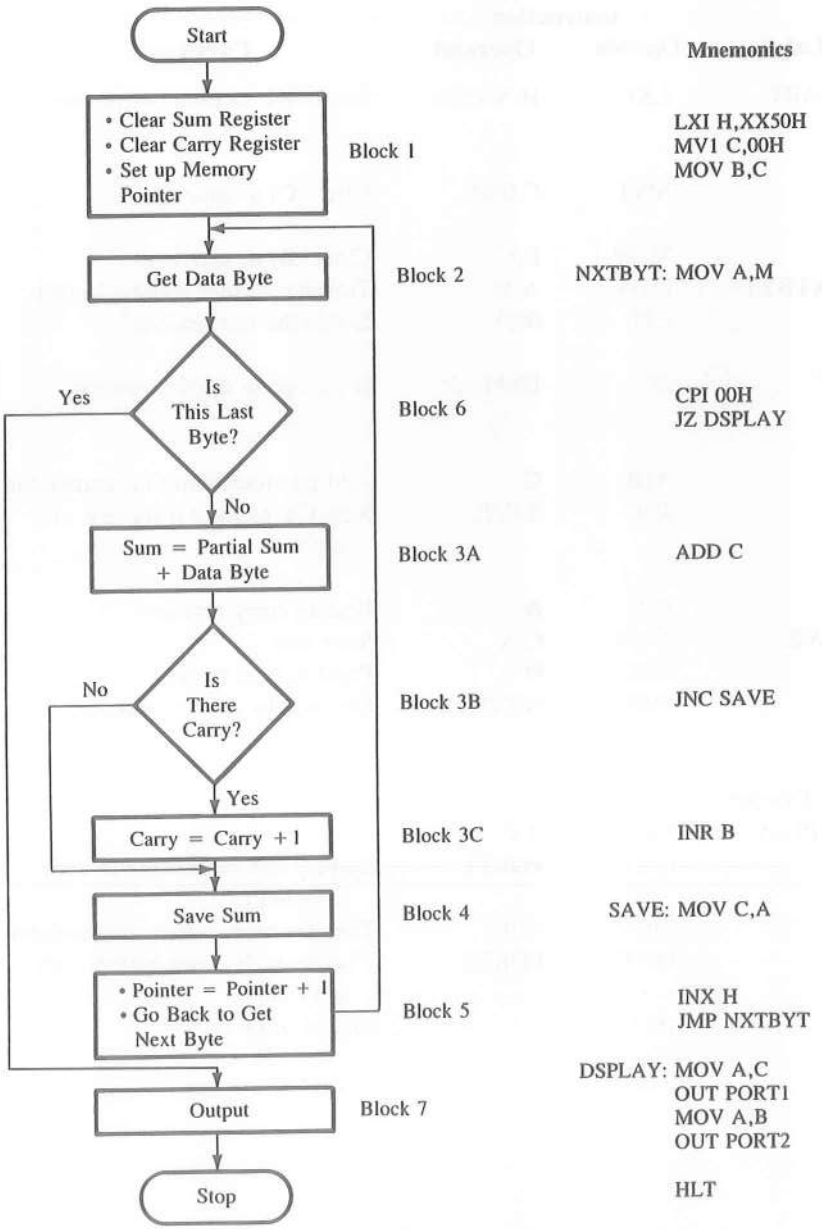


FIGURE 7.18
Flowchart for Compare Instruction to Check End of Data String

PROGRAM

Memory Address HI-LO	Machine Code	Label	Instruction Opcode	Operand	Comments
XX00	21	START:	LXI	H,XX50H	;Set up HL as memory pointer
01	50				
02	XX				
03	0E		MVI	C,00H	;Clear (C) to save sum
04	00				
05	41		MOV	B,C	;Clear (B) to save carry
06	7E	NXTBYT:	MOV	A,M	;Transfer current reading to (A)
07	FE		CPI	00H	;Is this the last reading?
08	00				
09	CA		JZ	DSPLAY	;If yes; go to display section
0A	16				
0B	XX				
0C	81		ADD	C	;Add previous sum to accumulator
0D	D2		JNC	SAVE	;Skip CY register if there is no
0E	11				; carry
0F	XX				
10	04		INR	B	;Update carry register
11	4F	SAVE:	MOV	C,A	;Save sum
12	23		INX	H	;Point to next reading
13	C3		JMP	NXTBYT	;Go back to get next reading
14	06				
15	XX				
		;Output Display			
16	79	DSPLAY:	MOV	A,C	
17	D3		OUT	PORT1	;Display low-order byte of sum
18	PORT1				; at PORT1
19	78		MOV	A,B	;Transfer carry bits to accumulator
1A	D3		OUT	PORT2	;Display high-order byte of sum
1B	PORT2				; at PORT2
1C	76		HLT		;End of program

```
                    ;Storing Result in Memory—Alternative to Output Display
16      21      DSPLAY:   LXI     H,XX70    ;Point index to XX70H location
17      70
18      XX
19      71                MOV     M,C       ;Store low-order byte of sum
                                            ;  in XX70H
1A      23                INX     H         ;Point index to XX71H
1B      70                MOV     M,B       ;Store high-order byte
1C      76                HLT               ;End of program

50      32      ;Data—Current Readings
51      52
52      F2
53      A5
54      00
```

PROGRAM DESCRIPTION AND OUTPUT

This program adds the first four readings, which results in the sum 21BH. The low-order byte 1BH is saved in register C, and the high-order byte 02H is stored in the carry register B. These are each displayed at different output ports by the OUT instruction or stored in the memory locations XX70 and XX71.

See Questions and Programming Assignments 41–56 at the end of the chapter.

7.5.3 Illustrative Program: Sorting

PROBLEM STATEMENT

A set of three readings is stored in memory starting at XX50H. Sort the readings in ascending order.

Data(H) 87, 56, 42

PROBLEM ANALYSIS

In this problem, three readings should be arranged in ascending order: 42, 56, and 87. The number of readings is kept small to simplify the explanation. However, the programming technique used here, called bubble sort, can be applied to a large set of data. Ordering data in a given sequence such as ascending, descending, or alphabetical order is a common application in programming.

The technique involved is comparing two bytes at a time and placing them in the proper sequence. For example, in our data set, we will compare the first two bytes (87H and 56H), and if the first byte is larger than the second byte, we will exchange their mem-

ory locations to arrange them in ascending order; otherwise, we will keep them in the same locations. We will follow the same procedure for the second and third bytes. The number of comparisons necessary is always $N - 1$ where N is the number of data bytes. Therefore, we need a counter of 2 for one complete comparison. Table 7.1 shows the memory locations and their contents after each pass (execution of one cycle). Two exchanges occur in the first pass, and the bytes are sequenced as 56, 42, and 87. In the second pass, only one exchange occurs. The microprocessor should continue these comparisons until no exchanges occur. We need to devise a technique or set up a reminder for the processor to recognize that no exchanges occur in a given pass. This is called setting a flag. In daily life, we write a note to remind ourselves. Similarly, we can use any register to write a binary note to the microprocessor. For example, we can write 1 (or any number such as FFH) in register D when an exchange takes place; otherwise, register D will be cleared.

PROGRAM

```
START:   LXI H, XX50H    ;Set up HL as a memory pointer for bytes
         MVI D, 00       ;Clear register D to set up a flag
         MVI C, 02       ;Set register C for comparison count
CHECK:   MOV A, M        ;Get data byte
         INX H           ;Point to next byte
         CMP M           ;Compare bytes
         JC NXTBYT       ;If (A) < second byte, do not exchange
         MOV B, M        ;Get second byte for exchange
         MOV M, A        ;Store first byte in second location
         DCX H           ;Point to first location
         MOV M, B        ;Store second byte in first location
         INX H           ;Get ready for next comparison
         MVI D, 01       ;Load 1 in D as a reminder for exchange
NXTBYT:  DCR C           ;Decrement comparison count
         JNZ CHECK       ;If comparison count ≠ 0, go back
         MOV A, D        ;Get flag bit in A
         RRC             ;Place flag bit D0 in Carry
         JC START        ;If flag is 1, exchange occurred
                         ;Start the next pass
         HLT             ;End of sorting
```

TABLE 7.1
Execution of Sort Program

Memory Locations	Initial Bytes	First Pass	Second Pass	Third Pass
XX50	87	56	42	42
XX51	56	42	56	56
XX52	42	87	87	87
Reg. D	0	1	1	0

PROGRAM DESCRIPTION AND OUTPUT

During the initialization phase, register HL is set up as a pointer where readings are stored; register D is cleared to set up a flag when an exchange occurs; and register C is initialized for a count of two because we have three data bytes to sort. Next is the comparison phase. The programs gets the first byte (87H) in A and compares that with the second byte (56H). In this comparison, the carry flag is reset; therefore, the next set of instructions places 56H in location XX50H and 87H in location XX51H. The flag in register D is set, the comparison counter in register C is decremented by 1, and the program returns to location CHECK for the next comparison. In this second comparison, 87H is compared with 42H, and again the bytes are exchanged. The flag (reminder) in D is set again, but now the comparison counter is zero. Therefore, the program falls through the first loop and checks the status of the flag in D by rotating the flag bit into Carry. Because Carry is set, the program goes back to the beginning and starts the process again from location XX50H. In the second pass, 56H and 42H are exchanged; therefore, the process is repeated a third time. In this pass, every comparison generates a carry. The program jumps to location NXTBYT, and the flag in register D remains zero. Thus, the program places the bytes in ascending order: 42H, 56H, and 87H.

7.6 DYNAMIC DEBUGGING

After you have completed the steps in the process of static debugging (described in the previous chapter), if the program still does not produce the expected output, you can attempt to debug the program by observing the execution of instructions. This is called **dynamic debugging.**

7.6.1 Tools for Dynamic Debugging

In a single-board microcomputer, techniques and tools commonly used in dynamic debugging are

- ☐ Single Step
- ☐ Register Examine
- ☐ Breakpoint

Each will be discussed below; the Single-Step and Register Examine keys were discussed briefly in the previous chapter.

SINGLE STEP

The Single-Step key on a keyboard allows you to execute one instruction at a time, and to observe the results following each instruction. Generally, a single-step facility is built with a hard-wired logic circuit. As you push the Single-Step key, you will be able to observe addresses and codes as they are executed. With the single-step technique you will be able to spot

- ☐ incorrect addresses
- ☐ incorrect jump locations for loops
- ☐ incorrect data or missing codes

To use this technique effectively, you will have to reduce loop and delay counts to a minimum number. For example, in a program that transfers 100 bytes, it is meaningless to set the count to 100 and single-step the program 100 times. By reducing the count to two bytes, you will be able to observe the execution of the loop. (If you reduce the count to one byte, you may not be able to observe the execution of the loop.) By single-stepping the program, you will be able to infer the flag status by observing the execution of Jump instructions. The single-step technique is very useful for short programs.

REGISTER EXAMINE

The Register Examine key allows you to examine the contents of the microprocessor register. When appropriate keys are pressed, the monitor program can display the contents of the registers. This technique is used in conjunction either with the single-step or the breakpoint facilities discussed below.

After executing a block of instructions, you can examine the register contents at a critical juncture of the program and compare these contents with the expected outcomes.

BREAKPOINT

In a single-board computer, the breakpoint facility is, generally, a software routine that allows you to execute a program in sections. The breakpoint can be set in your program by using RST instructions. (See "Interrupts," Chapter 12.) When you push the Execute key, your program will be executed until the breakpoint, where the monitor takes over again. The registers can be examined for expected results. If the segment of the program is found satisfactory, a second breakpoint can be set at a subsequent memory address to debug the next segment of the program. With the breakpoint facility you can isolate the segment of the program with errors. Then that segment of the program can be debugged with the single-step facility. The breakpoint technique can be used to check out the timing loop, I/O section, and interrupts. (See Chapter 12 for how to write a breakpoint routine.)

7.6.2 Common Sources of Errors

Common sources of errors in the instructions and programs illustrated in this chapter are as follows:

1. Failure to clear the accumulator when it is used to add numbers.
2. Failure to clear the carry registers or keep track of a carry.
3. Failure to update a memory pointer or a counter.
4. Failure to set a flag before using a conditional Jump instruction.
5. Inadvertently clearing the flag before using a Jump instruction.
6. Specification of a wrong memory address for a Jump instruction.
7. Use of an improper combination of Rotate instructions.

8. Specifying the Jump instruction on a wrong flag. This is a very common error with the Compare instructions.

See Questions and Programming Assignments 57–59 at the end of the chapter

SUMMARY

In this chapter, programming techniques—such as looping, counting, and indexing—were illustrated using memory-related data transfer instructions, 16-bit arithmetic instructions, and logic instructions. Techniques used commonly in debugging a program—single step, register examine, and breakpoint—were discussed; and common sources of errors were listed.

Review of Instructions

The instructions introduced and illustrated in this chapter are summarized below for an overview.

Instructions	Task	Addressing Mode
Data Transfer (Copy) Instructions		
1. LXI rp,16-bit	Load 16-bit data in a register pair.	Immediate
2. MOV R,M	Copy (M) into (R).	Indirect
3. MOV M,R	Copy (R) into (M).	Indirect
4. LDAX B/D	Copy the contents of the memory, indicated by the register pair, into the accumulator.	Indirect
5. STAX B/D	Copy (A) into the memory, indicated by the register pair.	Indirect
6. LDA 16-bit	Copy (M) into (A), memory specified by the 16-bit address.	Direct
7. STA 16-bit	Copy (A) into memory, specified by the 16-bit address.	Direct
8. MVI M,8-bit	Load 8-bit data in memory; the memory address is specified by (HL).	Indirect
Arithmetic Instructions		
1. ADD M	Add (M) to (A).	Indirect
2. SUB M	Subtract (M) from (A).	Indirect
3. INR M	Increment the contents of (M) by 1.	Indirect
4. DCR M	Decrement the contents of (M) by 1.	Indirect
Logic Instructions		
1. RLC	Rotate each bit in the accumulator to the left.	

2. RAL	Rotate each bit in the accumulator to the left through the Carry.	
3. RRC	Rotate each bit in the accumulator to the right.	
4. RAR	Rotate each bit in the accumulator to the right through the Carry.	
5. CMP R	Compare (R) with (A).	Register
6. CMP M	Compare (M) with (A).	Indirect
7. CPI 8-bit	Compare 8-bit data with (A).	Immediate

LOOKING AHEAD

The instructions that have been introduced in this and the previous chapter make up the major segment of the instruction set. Applications of these instructions in designing counters and time delays are illustrated in the next chapter. The other group of instructions critical for assembly language programming is related to the subroutine technique, illustrated in Chapter 9.

QUESTIONS AND PROGRAMMING ASSIGNMENTS

If data bytes are given in the programming assignments, enter those data bytes manually in the respective memory locations before executing the programs. If an assignment calls for an output port and your microcomputer does not have an independent output port, store the results in memory. The high-order byte of a memory address is shown as XX. Substitute the high-order byte that is appropriate to the trainer in your laboratory.

Section 7.1

1. You are given a set of 100 resistors with 10 k value and asked to test them for 10 percent tolerance. Reject all the resistors that are outside the tolerance and give the final number of resistors you found within the tolerance. Draw a flowchart.
2. You are given a big grocery list for a party and asked to buy the items starting from number 60. The maximum amount you can spend is $125. Draw a flowchart.
3. In the receiving department of a manufacturing company, auto parts are placed in a sequential order from 0000 to 4050. These parts need to be transferred to the bins on the factory floor, marked with the starting number 8000. You have an intelligent robot that can read the bin numbers and transfer the parts. Draw a flowchart to instruct the robot to transfer 277 parts starting from bin number 1025 in the receiving department to the starting bin number 8060 on the floor.

4. A newly hired librarian was given a set of new books on computers and English literature. The stack of computer books was placed above the English books. The librarian was asked to find the total amount spent on computer books; the prices were marked on the back cover of each book. Draw a flowchart representing the process.

Section 7.2

5. Specify the memory location and its contents after the following instructions are executed.

```
MVI B,F7H
MOV A,B
STA XX75H
HLT
```

6. Show the register contents as each of the following instructions is being executed.

	A	B	C	D	E	H	L

```
MVI C,FFH
LXI H,XX70H
LXI D,XX70H
MOV M,C
LDAX D
HLT
```

7. In Question 6, specify the contents of the accumulator and the memory location XX70H after the execution of the instruction LDAX D.
8. Identify the memory locations that are cleared by the following instructions.

```
MVI B,00H
LXI H,XX75H
MOV M,B
INX H
MOV M,B
HLT
```

9. Specify the contents of registers A, D, and HL after execution of the following instructions.

```
      LXI H,XX90H ;Set up register HL as a memory pointer
      SUB A       ;Clear accumulator
      MVI D,0FH   ;Set up register D as a counter
LOOP: MOV M,A     ;Clear memory
      INX H       ;Point to the next memory location
```

```
DCR D       ;Update counter
JNZ LOOP    ;Repeat until (D) = 0
HLT
```

10. Explain the result after the execution of the program in Question 9.

11. Rewrite the instructions in Question 9 using the register BC as a memory pointer.

12. Explain how many times the following loop will be executed.

```
      LXI B,0007H
LOOP: DCX B
      JNZ LOOP
```

13. Explain how many times the following loop will be executed.

```
      LXI B,0007H
LOOP: DCX B
      MOV A,B
      ORA C
      JNZ LOOP
```

14. The following instructions are intended to clear ten memory locations starting from the memory address 0009H. Explain why a large memory block will be erased or cleared and the program will stay in an infinite loop.

```
      LXI H,0009H
LOOP: MVI M,00H
      DCX H
      JNZ LOOP
      HLT
```

15. The following block of data is stored in the memory locations from XX55H to XX5AH. Transfer the data to the locations XX80H to XX85H in the reverse order (e.g., the data byte 22H should be stored at XX85H and 37H at XX80H).
Data(H) 22, A5, B2, 99, 7F, 37

16. Data bytes are stored in memory locations from XX50H to XX5FH. To insert an additional five bytes of data, it is necessary to shift the data string by five memory locations. Write a program to store the data string from XX55H to XX64H. Use any sixteen bytes of data to verify your program.

Hint: This is a block transfer of data bytes with overlapping memory locations. If the data transfer begins at location XX50H, a segment of the data string will be destroyed.

17. A system is designed to monitor the temperature of a furnace. Temperature readings are recorded in 16 bits and stored in memory locations starting at XX60H. The high-order byte is stored first and the low-order byte is stored in the next consecutive memory location. However, the high-order byte of all the temperature readings is constant.

Write a program to transfer low-order readings to consecutive memory locations starting at XX80H and discard the high-order bytes.
Temperature Readings (H) 0581, 0595, 0578, 057A, 0598

18. A string of six data bytes is stored starting from memory location 2050H. The string includes some blanks (bytes with zero value). Write a program to eliminate the blanks from the string. (*Hint:* To check a blank, set the Zero flag by using the ORA. Use two memory pointers: one to get a byte and the other to store the byte.)
Data(H) F2, 00, 00, 4A, 98, 00

19. Write a program to add the following five data bytes stored in memory locations starting from XX60H, and display the sum. (The sum does not generate a carry. Use register pair DE as a memory pointer to transfer a byte from memory into a register.)
Data(H) 1A, 32, 4F, 12, 27

20. Write a program to add the following data bytes stored in memory locations starting at XX60H and display the sum at the output port if the sum does not generate a carry. If a result generates a carry, stop the addition, and display 01H at the output port.
Data(H) First Set: 37, A2, 14, 78, 97
Second Set: 12, 1B, 39, 42, 07

21. In Assignment 20, modify the program to count the number of data bytes that have been added and display the count at the second output port.

Section 7.3

22. Specify the contents of memory locations XX70H to XX74H after execution of the following instructions.

```
        LXI H,XX70H     ;Set up HL as a memory pointer
        MVI B,05H       ;Set up register B as a counter
        MVI A,01
STORE:  MOV M,A         ;Store (A) in memory
        INR A
        INX H
        DCR B
        JNZ STORE
        HLT
```

23. Identify the contents of the registers, the memory location (XX55H), and the flags as the following instructions are executed.

	A	H	L	S	Z	CY	M XX55H
LXI H,XX55H							
MVI M,8AH							
MVI A,76H							
ADD M							
STA XX55H							
HLT							

24. Assemble and execute the instructions in Question 23, and verify the contents.
25. Identify the contents of the memory location XX65H and the status of the flags S, Z, and CY when the instruction INR M is executed.

```
LXI H,XX65H
MVI M,FFH
INR M
HLT
```

26. Repeat the Illustrative Program: Addition with Carry (Section 7.3.2) using the DE register as a memory pointer and memory location XX40H as a counter. (*Hints:* Use the instruction MVI M to load the memory location XX40H with a count, and DCR M to decrement the counter.)
27. The temperatures of two furnaces are being monitored by a microcomputer. A set of five readings of the first furnace, recorded by five thermal sensors, is stored at the memory location starting at XX50H. A corresponding set of five readings from the second furnace is stored at the memory location starting at XX60H. Each reading from the first set is expected to be higher than the corresponding reading from the second set. For example, the temperature reading at the location 54H (T_{54}) is expected to be higher than the temperature reading at the location 64H (T_{64}).

    Write a program to check whether each reading from the first set is higher than the corresponding reading from the second set. If all readings from the first set are higher than the corresponding readings from the second set, turn on the bit D_0 of the output PORT1. If any one of the readings of the first set is lower than the corresponding reading of the second set, stop the process and output FF as an emergency signal to the output PORT1.

    Data(H) First Set: 82, 89, 78, 8A, 8F
    Second Set: 71, 78, 79, 82, 7F
28. Repeat Assignment 27 with the following modification. Check whether any two readings are equal, and if they are equal, turn on bit D_7 of PORT1 and continue checking. (*Hint:* Check for the Zero flag when two readings are equal.)

    Data(H) First Set: 80, 85, 8F, 82, 87
    Second Set: 71, 74, 7A, 82, 77
29. A set of eight data bytes is stored in memory locations starting from XX70H. Write a program to add two bytes at a time and store the sum in the same memory locations, low-order sum replacing the first byte and a carry replacing the second byte. If any pair does not generate a carry, the memory location of the second byte should be cleared.

    Data(H) F9, 38, A7, 56, 98, 52, 8F, F2
30. A set of eight data bytes is stored in memory locations starting from XX70H. Write a program to subtract two bytes at a time and store the result in a sequential order in memory locations starting from XX70H.

    Data(H) F9, 38, A7, 56, 98, A2, F4, 67
31. In Assignment 30, if any one of the results of the subtraction is in the 2's complement, it should be discarded.

Section 7.4

32. Specify the contents of the accumulator and the status of the CY flag when the following instructions are executed.

a.
```
MVI A,B7H
ORA A
RLC
```

b.
```
MVI A,B7H
ORA A
RAL
```

33. Identify the register contents and the flags as the following instructions are being executed.

	A	S	Z	CY

```
MVI A,80H
ORA A
RAR
```

34. Specify the contents of the accumulator and the CY flag when the following instructions are executed.

a.
```
MVI A,C5H
ORA A
RAL
RRC
```

b.
```
MVI A,A7H
ORA A
RAR
RAL
```

35. The accumulator in the following set of instructions contains a BCD number. Explain the function of these instructions.

```
MVI A,79H
ANI F0H
RRC
RRC
RRC
RRC
```

36. Calculate the decimal value of the number in the accumulator before and after the Rotate instructions are executed, and explain the mathematical functions performed by the instructions.

a.
```
MVI A,18H
RLC
```

b.
```
MVI A,78H
RRC
RRC
```

37. Explain the mathematical function that is performed by the following instructions.

```
MVI A,07H
RLC
MOV B,A
RLC
RLC
ADD B
```

38. Repeat the Illustrative Program in Section 7.4.2 ("Checking Sign with Rotate Instructions") with the following modifications:

a. Check the sign of a number by using the instruction JM (Jump On Minus), instead of the instruction RAL. (*Hint:* Set the flags by using the instruction ORA A.)

b. If the sum of the positive readings exceeds eight bits, continue the addition, save generated carry, and display the total sum at two different ports.

Data(H) 48, 72, 8F, 7F, 6B, F2, 98, 7C, 67, 19

39. In Assignment 38, in addition to modifications a and b, count the number of positive readings in the set and display the count at PORT3.

40. A set of eight data bytes is stored in the memory location starting at XX50H. Check each data byte for bits D_7 and D_0. If D_7 or D_0 is 1, reject the data byte; otherwise, store the data bytes at memory locations starting at XX60H.

Data(H) 80, 52, E8, 78, F2, 67, 35, 62

Section 7.5

41. Identify the contents of the accumulator and the flag status as the following instructions are executed.

	A	S	Z	CY
MVI A,7FH				
ORA A				
CPI A2H				

42. Identify the register contents and the flag status as the following instructions are executed.

	A	S	Z	CY
LXI H,2070H				
MVI M,64H				
MVI A,8FH				
CMP M				

43. Identify the bytes from the following set that will be displayed at PORT1, assuming one byte is loaded into the accumulator at a time.

Data(H) 58, 32, 7A, 87, F2, D7

```
MVI A,BYTE
MVI B,64H
MVI C,C8H
CMP B
JC REJECT
CMP C
```

```
        JNC REJECT
        OUT PORT1
        HLT
REJECT: SUB A
        OUT PORT1
        HLT
```

44. In Question 43, identify the range of numbers in decimal that will be displayed at PORT1.

45. The following program reads one data byte at a time. Identify the data bytes from the following set that will transfer the program to location ACCEPT.
Data(H) 19, 20, 64, 8F, D8, F2

```
        IN PORT1
        MVI B,20H
        CMP B
        JC REJECT
        JM REJECT
        STA 2070H
        JMP ACCEPT
REJECT: JMP INVALID
```

46. In Question 45, identify the range of numbers in decimal that will transfer the program to location INVALID.

47. Repeat the Illustrative Program: Use of Compare Instruction to Indicate the End of a Data String (Section 7.5.2), but include the following modifications: Clear register D and use CMP D to check a byte in the memory location. If a byte is not zero, add the byte and continue adding; otherwise, go to the output.

48. A set of eight readings is stored in memory starting at location XX50H. Write a program to check whether a byte 40H exists in the set. If it does, stop checking and display its memory location; otherwise output FFH.
Data(H) 48, 32, F2, 38, 37, 40, 82, 8A

49. Refer to Assignment 48. Write a program to find the highest reading in the set, and display the reading at an output port.

50. Refer to Assignment 48. Write a program to find the lowest reading in the set, and display the reading at the output port.

51. A set of ten bytes is stored in memory starting with the address XX50H. Write a program to check each byte, and save the bytes that are higher than 60_{10} and lower than 100_{10} in memory locations starting from XX60H.
Data(H) 6F, 28, 5A, 49, C7, 3F, 37, 4B, 78, 64

52. In Assignment 51, in addition to saving the bytes in the given range, display at PORT1 the number of bytes saved.

53. A string of readings is stored in memory locations starting at XX70H, and the end of the string is indicated by the byte 0DH. Write a program to check each byte in the string, and save the bytes in the range of 30H to 39H (both inclusive) in memory locations starting from XX90H.
Data(H) 35, 2F, 30, 39, 3A, 37, 7F, 31, 0D, 32

54. In Assignment 53, display the number of bytes accepted from the string between 30H and 39H.

55. A bar code scanner scans the boxes being shipped from the loading dock and records all the codes in computer memory; the end of the data is indicated by the byte 00. The code 1010 0011 (A3H) is assigned to 19″ television sets. Write a program to count the number of 19″ television sets that were shipped from the following data set.
Data(H) FA, 67, A3, B8, A3, A3, FA, 00

56. Sort the following set of marks scored by ten students in a circuit course in descending order.
Data(H) 63, 41, 56, 62, 48, 5A, 4F, 4C, 56, 56

Section 7.6

57. The following program adds the number of bytes stored in memory locations starting from XX00H and saves the result in memory. Read the program and answer the questions given below.

```
        LX1 H,XX00H     ;Set up HL as a data pointer
        LX1, D,0000H    ;Set up D as a byte counter
                        ;   and E as a Carry register
NEXT:   ADD M           ;Add byte
        JNC SKIP        ;If the result has no carry, do not
                        ;   increment Carry register
        INR E
SKIP:   DCR D           ;Update byte counter
        JNZ NEXT        ;Go to next byte
        LXI H,XX90H
        MOV M,A         ;Save the result
        INX H
        MOV M,E
        HLT
```

a. Assuming the byte counter is set up appropriately, specify the number of bytes that are added by the program.
b. Specify the memory locations where the result is stored.
c. Identify the two errors in the program.

58. The following program checks a set of six signed numbers and adds the positive numbers. The numbers are stored in memory locations starting from

XX60H. The final result is expected to be less than FFH and stored in location XX70H.

Data(H) First Set: 20, 87, F2, 37, 79, 17
Second Set: A2, 15, 3F, B7, 47, 9A

```
        MVI B,00H       ;Clear (B) to save result
        MVI C,06H       ;Set up register C to count six numbers
        LXI H,XX60H     ;Set up HL as a memory pointer
NEXT:   MOV A,M         ;Get a byte
        RAL             ;Place D7 into CY flag
        JC REJECT       ;If D7 = 1, reject the byte
        RAR             ;Restore the byte
        ADD B           ;Add the previous sum
        MOV B,A         ;Save the sum
REJECT: INX H           ;Next location
        DCR B           ;Update the byte counter
        JNZ NEXT        ;Go back to get the next byte
        STA XX70H       ;Save the result
        HLT
```

a. Calculate the answers for the given data sets.
b. Assemble and execute the program for the first data set, and verify the answer.
c. Execute the program for the second data set, and verify the answer.
d. Debug the program using the Single-Step and Examine Register techniques if the result is different from the expected answer.

59. The following program adds five bytes stored in memory locations starting from 2055H. The result is stored in locations 2060H and 2061H.

Data(H) First Set: 17, 1B, 21, 7F, 9D
Second Set: F2, 87, A9, B8, C2

```
2000    21 55 20    LXI H,2055H     ;Set HL as a memory pointer
2003    AF          XRA A           ;Clear accumulator
2004    01 05 00    LXI B,0005H     ;Clear B to save carries and set up C as
                                    ;   a byte counter
2007    8E          ADC M           ;Add byte
2008    D2 0C 20    JNC 200CH       ;Skip CY register if no carry
200B    04          INR B           ;Update CY register
200C    0D          DCR C           ;Update counter
200D    23          INX H           ;Next memory location
200E    C2 20 08    JNZ 2008H       ;Go back to add next byte
2011    32 60 20    STA 2060H       ;Store low-order sum
2014    78          MOV A,B         ;Get high-order sum
2015    32 61 20    STA 2061        ;Store high-order sum
2018                HLT
```

a. Calculate the answers for the given data sets.
b. Enter the program and the first data set. Execute the program and verify the answer. Debug the program if necessary.
c. Execute the program for the second data set, and verify the answer.
d. Debug the program using the Single-Step and Examine Register techniques if the result is different from the expected answer.

8

Counters and Time Delays

This chapter deals with the designing of counters and timing delays through software (programming). Two of the programming techniques discussed in the last chapter—looping and counting—are used to design counters and time delays. The necessary instructions have already been introduced in the previous two chapters.

A counter is designed by loading an appropriate count in a register. A loop is set up to decrement the count for a down-counter* or to increment the count for an up-counter.† Similarly, a timing delay is designed by loading a register with a delay count and setting up a loop to decrement the count until zero. The delay is determined by the clock period of the system and the time required to execute the instructions in the loop.

Counters and time delays are important techniques. They are commonly used in applications such as traffic signals, digital clocks, process control, and serial data transfer.

OBJECTIVES

- ☐ Write instructions to set up time delays, using one register, a register pair, and a loop-within-a-loop technique.
- ☐ Calculate the time delay in a given loop.
- ☐ Draw a flowchart for a counter with a delay.
- ☐ Design an up/down counter for a given delay.
- ☐ Write a program to turn on/off specific bits at a given interval.

*A down-counter counts in descending order.

†An up-counter counts in ascending order.

8.1 COUNTERS AND TIME DELAYS

Designing a counter is a frequent programming application. Counters are used primarily to keep track of events; time delays are important in setting up reasonably accurate timing between two events. The process of designing counters and time delays using software instructions is far more flexible and less time consuming than the design process using hardware.

COUNTER

A counter is designed simply by loading an appropriate number into one of the registers and using the INR (Increment by One) or the DCR (Decrement by One) instructions. A loop is established to update the count, and each count is checked to determine whether it has reached the final number; if not, the loop is repeated.

The flowchart shown in Figure 8.1 illustrates these steps. However, this counter has one major drawback; the counting is performed at such high speed that only the last count can be observed. To observe counting, there must be an appropriate time delay between counts.

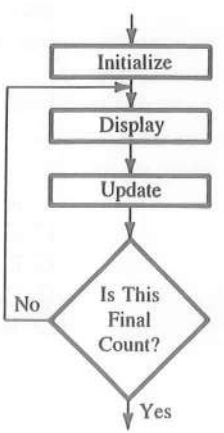
TIME DELAY

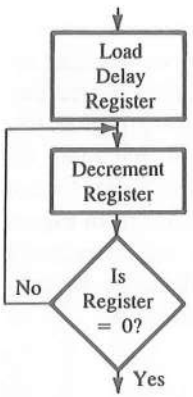
The procedure used to design a specific delay is similar to that used to set up a counter. A register is loaded with a number, depending on the time delay required, and then the register is decremented until it reaches zero by setting up a loop with a conditional Jump instruction. The loop causes the delay, depending upon the clock period of the system, as illustrated in the next sections.

8.1.1 Time Delay Using One Register

The flowchart in Figure 8.2 shows a time-delay loop. A count is loaded in a register, and the loop is executed until the count reaches zero. The set of instructions necessary to set up the loop is also shown in Figure 8.2.

FIGURE 8.1
Flowchart of a Counter





Label	Opcode	Operand	Comments	T-states
	MVI	C,FFH	;Load register C	7
LOOP:	DCR	C	;Decrement C	4
	JNZ	LOOP	;Jump back to ; decrement C	10/7

FIGURE 8.2
Time Delay Loop: Flowchart and Instructions

The last column in Figure 8.2 shows the T-states (clock periods) required by the 8085 microprocessor to execute each instruction. (See Appendix F for the list of the 8085 instructions and their T-states.) The instruction MVI requires seven clock periods. An 8085-based microcomputer with 2 MHz clock frequency will execute the instruction MVI in 3.5 μs as follows:

$$\begin{aligned}&\text{Clock frequency of the system } f = 2 \text{ MHz}\\&\text{Clock period } T = 1/f = 1/2 \times 10^{-6} = 0.5\ \mu s\\&\text{Time to execute MVI} = 7 \text{ T-states} \times 0.5\\&\qquad\qquad = 3.5\ \mu s\end{aligned}$$

However, if the clock frequency of the system is 1 MHz, the microprocessor will require 7 μs to execute the same instruction. To calculate the time delay in a loop, we must account for the T-states required for each instruction and for the number of times the instructions are executed in the loop.

In Figure 8.2, register C is loaded with the count FFH (255_{10}) by the instruction MVI, which is executed once and takes seven T-states. The next two instructions, DCR and JNZ, form a loop with a total of 14 (4 + 10) T-states. The loop is repeated 255 times until register C = 0.

The time delay in the loop T_L with 2 MHz clock frequency is calculated as

$$TL = (T \times \text{Loop T-states} \times N10)$$

where T_L = Time delay in the loop
T = System clock period
N_{10} = Equivalent decimal number of the hexadecimal count loaded in the delay register

$$\begin{aligned}T_L &= (0.5 \times 10^{-6} \times 14 \times 255)\\&= 1785\ \mu s\\&\approx 1.8\ ms\end{aligned}$$

ATILIM UNIVERSITY LIBRARY

In most applications, this approximate calculation of the time delay is considered reasonably accurate. However, to calculate the time delay more accurately, we need to adjust for the execution of the JNZ instruction and add the execution time of the initial instruction.

The T-states for JNZ instruction are shown as 10/7. This can be interpreted as follows: The 8085 microprocessor requires ten T-states to execute a conditional Jump instruction when it jumps or changes the sequence of the program and seven T-states when the program falls through the loop (goes to the instruction following the JNZ). In Figure 8.2, the loop is executed 255 times; in the last cycle, the JNZ instruction will be executed in seven T-states. This difference can be accounted for in the delay calculation by subtracting the execution time of three states. Therefore, the adjusted loop delay is

$$\begin{aligned} T_{LA} &= T_L - (3 \text{ T-states} \times \text{Clock period}) \\ &= 1785.0\ \mu s - 1.5\ \mu s = 1783.5\ \mu s \end{aligned}$$

Now the total delay must take into account the execution time of the instructions outside the loop. In the above example, we have only one instruction (MVI C) outside the loop. Therefore, the total delay is

$$\text{Total Delay} = \begin{matrix}\text{Time to execute instructions} \\ \text{outside loop}\end{matrix} + \begin{matrix}\text{Time to execute} \\ \text{loop instructions}\end{matrix}$$

$$\begin{aligned} T_D &= T_O + T_{LA} \\ &= (7 \times 0.5\ \mu s) + 1783.5\ \mu s = 1787\ \mu s \\ &\approx 1.8\ ms \end{aligned}$$

The difference between the loop delay T_L and these calculations is only 2 μs and can be ignored in most instances.

The time delay can be varied by changing the count FFH; however, to increase the time delay beyond 1.8 ms in a 2 MHz microcomputer system, a register pair or a loop within a loop technique should be used.

8.1.2 Time Delay Using a Register Pair

The time delay can be considerably increased by setting a loop and using a register pair with a 16-bit number (maximum FFFFH). The 16-bit number is decremented by using the instruction DCX. However, the instruction DCX does not set the Zero flag and, without the test flags, Jump instructions cannot check desired data conditions. Additional techniques, therefore, must be used to set the Zero flag.

The following set of instructions uses a register pair to set up a time delay.

Label	Opcode	Operand	Comments	T-states
	LXI	B,2384H	;Load BC with 16-bit count	10
LOOP:	DCX	B	;Decrement (BC) by one	6
	MOV	A,C	;Place contents of C in A	4
	ORA	B	;OR (B) with (C) to set Zero flag	4
	JNZ	LOOP	;If result ≠ 0, jump back to LOOP	10/7

In this set of instructions, the instruction LXI B,2384H loads register B with the number 23H, and register C with the number 84H. The instruction DCX decrements the entire number by one (e.g., 2384H becomes 2383H). The next two instructions are used only to set the Zero flag; otherwise, they have no function in this problem. The OR instruction sets the Zero flag only when the contents of B and C are simultaneously zero. Therefore, the loop is repeated 2384H times, equal to the count set in the register pair.

TIME DELAY

The time delay in the loop is calculated as in the previous example. The loop includes four instructions: DCX, MOV, ORA, and JNZ, and takes 24 clock periods for execution. The loop is repeated 2384H times, which is converted to decimals as

$$\begin{aligned} 2384H &= 2 \times (16)^3 + 3 \times (16)^2 + 8 \times (16)^1 + 4(16^0) \\ &= 9092_{10} \end{aligned}$$

If the clock period of the system = 0.5 μs, the delay in the loop T_L is

$$\begin{aligned} T_L &= (0.5 \times 24 \times 9092_{10}) \\ &\approx 109 \text{ ms (without adjusting for the last cycle)} \\ \text{Total Delay } T_D &= 109 \text{ ms} + T_O \\ &\approx 109 \text{ ms (The instruction LXI adds only 5 μs.)} \end{aligned}$$

A similar time delay can be achieved by using the technique of two loops, as discussed in the next section.

8.1.3 Time Delay Using a Loop within a Loop Technique

A time delay similar to that of a register pair can also be achieved by using two loops; one loop inside the other loop, as shown in Figure 8.3(a). For example, register C is used in the inner loop (LOOP1) and register B is used for the outer loop (LOOP2). The following instructions can be used to implement the flowchart shown in Figure 8.3(a).

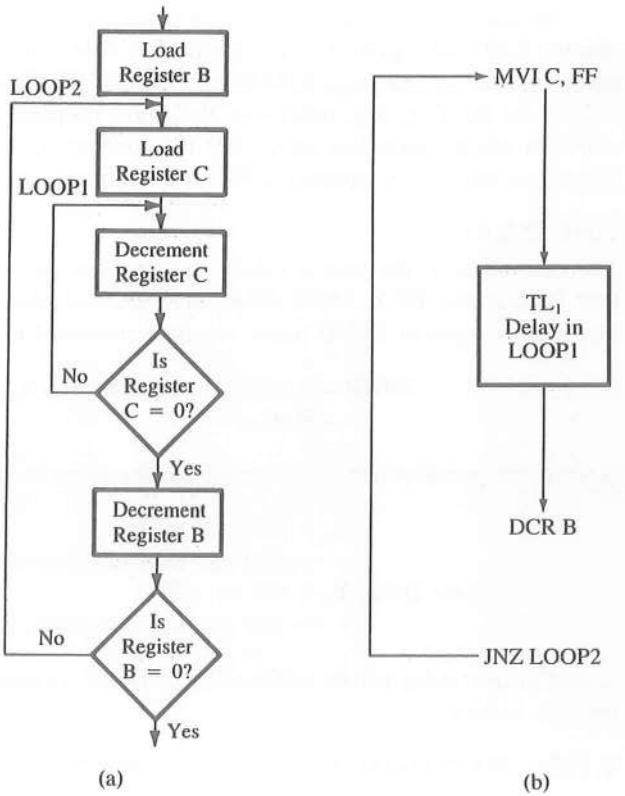
```
        MVI B,38H     7T
LOOP2:  MVI C,FFH     7T
LOOP1:  DCR C         4T
        JNZ LOOP1     10/7T
        DCR B         4T
        JNZ LOOP2     10/7T
```

DELAY CALCULATIONS

The delay in LOOP1 is T_{L1} = 1783.5 μs. These calculations are shown in Section 8.1.1. We can replace LOOP1 by T_{L1}, as shown in Figure 8.3(b). Now we can calculate the delay in LOOP2 as if it is one loop; this loop is executed 56 times because of the count (38H) in register B:

$$\begin{aligned} T_{L2} &= 56(T_{L1} + 21 \text{ T-states} \times 0.5 \text{ μs}) \\ &= 56(1783.5 \text{ μs} + 10.5 \text{ μs}) \\ &= 100.46 \text{ ms} \end{aligned}$$

FIGURE 8.3
Flowchart for Time Delay with Two Loops



The total delay should include the execution time of the first instruction (MVI B,7T); however, the delay outside these loops is insignificant. The time delay can be increased considerably by using register pairs in the above example.

Similarly, the time delay within a loop can be increased by using instructions that will not affect the program except to increase the delay. For example, the instruction NOP (No Operation) can add four T-states in the delay loop. The desired time delay can be obtained by using any or all available registers.

8.1.4 Additional Techniques for Time Delay

The disadvantages in using software delay techniques for real-time applications in which the demand for time accuracy is high, such as digital clocks, are as follows:

1. The accuracy of the time delay depends on the accuracy of the system's clock.
2. The microprocessor is occupied simply in a waiting loop; otherwise it could be employed to perform other functions.
3. The task of calculating accurate time delays is tedious.

In real-time applications, timers (integrated timer circuits) are commonly used. The Intel 8254 (described in Chapter 15) is a programmable timer chip that can be interfaced with the microprocessor and programmed to provide timings with considerable accuracy. The disadvantages of using the hardware chip include the additional expense and the need for an extra chip in the system.

8.1.5 Counter Design with Time Delay

To design a counter with a time delay, the techniques illustrated in Figures 8.1 and 8.2 can be combined. The combined flowchart is shown in Figure 8.4.

The blocks shown in the flowchart are similar to those in the generalized flowchart in Figure 7.3. The block numbers shown in Figure 8.4 correspond to the block numbers in the generalized flowchart. Compare Figure 8.4 with Figure 7.3 and note the following points about the counter flowchart (Figure 8.4):

1. The output (or display) block is part of the counting loop.
2. The data processing block is replaced by the time-delay block.
3. The save-the-partial-answer block is eliminated because the count is saved in a counter register, and a register can be incremented or decremented without transferring the count to the accumulator.

The flowchart in Figure 8.4 shows the basic building blocks. However, the sequence can be changed, depending upon the nature of the problem, as shown in Figures 8.5(a) and 8.5(b).

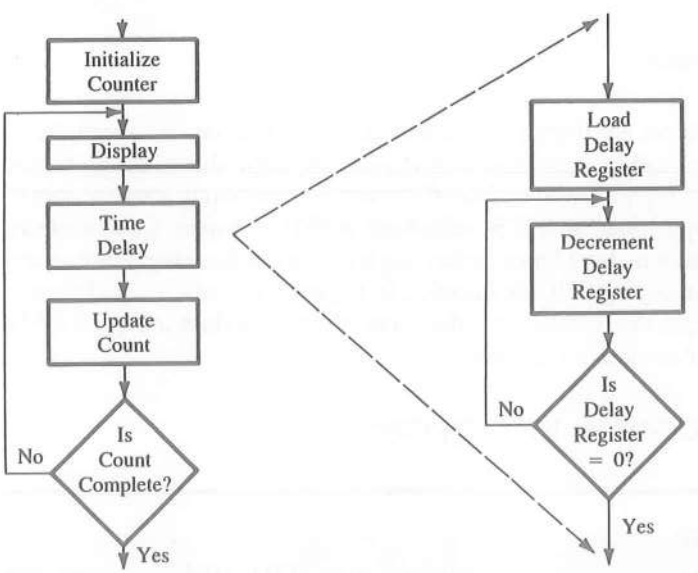


FIGURE 8.4
Flowchart of a Counter with a Time Delay

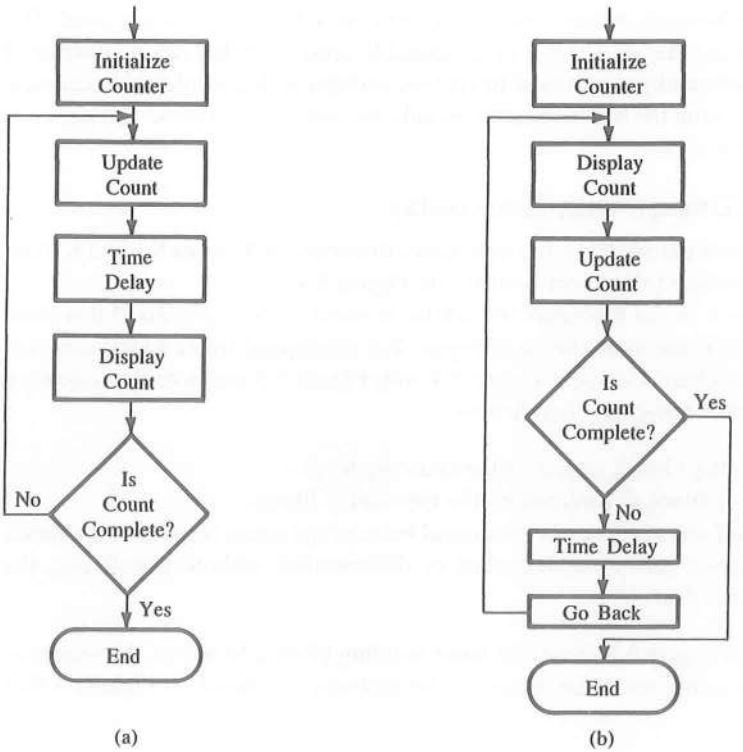


FIGURE 8.5
Variations of Counter Flowchart

The flowchart in Figure 8.4 displays the count after initialization. For example, an up-counter starting at zero can be initialized at 00H using the logic shown in Figure 8.4. However, the flowchart in Figure 8.5(a) updates the counter immediately after the initialization. In this case, an up-counter should be initialized at FFH to display the count 00H.

Similarly, the decision-making block differs slightly in these flowcharts. For example, if a counter was counting up to 9, the question in Figure 8.4 would be: Is the count 10? In Figure 8.5(a) the question would be: Is the count 9? The flowchart in Figure 8.5(b) illustrates another way of designing a counter.

8.2 ILLUSTRATIVE PROGRAM: HEXADECIMAL COUNTER

PROBLEM STATEMENT

Write a program to count continuously in hexadecimal from FFH to 00H in a system with a 0.5 μs clock period. Use register C to set up a one millisecond (ms) delay between each count and display the numbers at one of the output ports.

PROBLEM ANALYSIS

The problem has two parts; the first is to set up a continuous down-counter, and the second is to design a given delay between two counts.

1. The hexadecimal counter is set up by loading a register with an appropriate starting number and decrementing it until it becomes zero (shown by the outer loop in the flowchart, Figure 8.6). After zero count, the register goes back to FF because decrementing zero results in a (−1), which is FF in 2's complement.
2. The one millisecond (ms) delay between each count is set up by using the procedure explained previously in Section 8.1.1—Time Delay Using One Register. Figure 8.2 is identical with the inner loop of the flowchart shown in Figure 8.6. The delay calculations are shown later.

FLOWCHART AND PROGRAM

The flowchart in Figure 8.6 shows the two loops discussed earlier—one for the counter and another for the delay. The counter is initialized in the first block, and the count is displayed

Memory Address	Hex Code	Label	Mnemonics
HI-LO XX00	06		MVI B,00H
01	00		
02	05	NEXT:	DCR B
03	0E		MVI C.COUNT
04	XX		
05	0D	DELAY:	DCR C
06	C2		JNZ DELAY
07	05		
08	XX		
09	78		MOV A,B
0A	D3		OUT PORT#
0B	PORT#		
0C	C3		JMP NEXT
0D	02		
0E	XX		

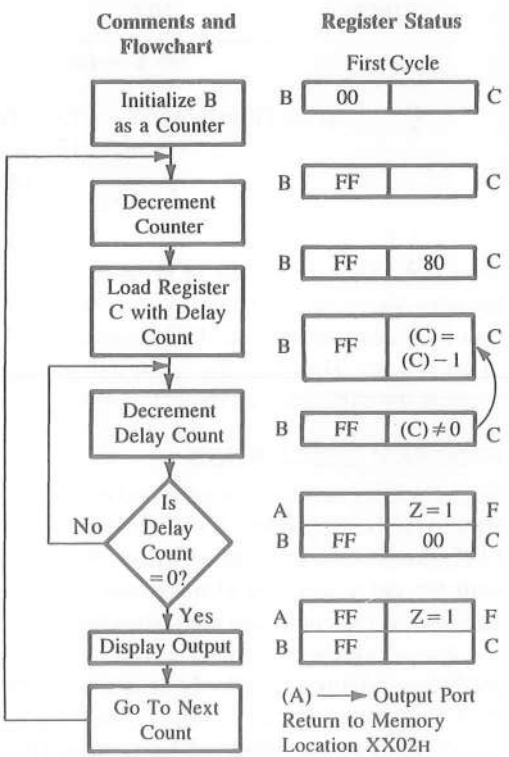


FIGURE 8.6
Program and Flowchart for a Hexadecimal Counter

in the outer loop. The delay is accomplished in the inner loop. This flowchart in Figure 8.6 is similar to that in Figure 8.5(a). You should study the flowchart carefully to differentiate between the counter loop and the delay loop because they may at first appear to be similar.

Delay Calculations The delay loop includes two instructions: DCR C and JNZ with 14 T-states. Therefore, the time delay T_L in the loop (without accounting for the fact that JNZ requires seven T-states in the last cycle) is

$$\begin{aligned} T_L &= 14 \text{ T-states} \times T \text{ (Clock period)} \times \text{Count} \\ &= 14 \times (0.5 \times 10^{-6}) \times \text{Count} \\ &= (7.0 \times 10^{-6}) \times \text{Count} \end{aligned}$$

The delay outside the loop includes the following instructions:

DCR B	4T
MVI C,COUNT	7T
MOV A,B	4T
OUT PORT	10T
JMP	10T
	35 T-states

Delay outside the loop: $T_O = 35 \text{ T-states} \times T$
$= 35 \times (0.5 \times 10^{-6})$
$= 17.5\ \mu s$

$$\text{Total Time Delay } T_D = T_O + T_L$$

$$1 \text{ ms} = 17.5 \times 10^{-6} + (7.0 \times 10^{-6}) \times \text{Count}$$

$$\text{Count} = \frac{1 \times 10^{-3} - 17.5 \times 10^{-6}}{7.0 \times 10^{-6}} \approx 140_{10}$$

Therefore, the delay count 8CH (140_{10}) must be loaded in register C to obtain 1 ms delay between each count. In this problem, the calculation of the delay count does not take into consideration that the JNZ instruction takes seven T-states in the last cycle, instead of 10 T-states. However, the count will remain the same even if the calculations take into account the difference of three T-states.

PROGRAM DESCRIPTION

Register B is used as a counter, and register C is used for delay. Register B initially starts with number 00H; when it is decremented by the instruction DCR, the number becomes FFH. (Verify this by subtracting one from zero in 2's complement.) Register C is loaded with the delay count 8CH to provide a 1 ms delay in the loop. The instruction DCR C decrements the count and the instruction JNZ (Jump On No Zero) checks the Zero flag to see if the number in register C has reached zero. If the number is not zero, instruction JNZ causes a jump back to the instruction labeled DELAY in order to decrement (C) and, thus, the loop is repeated 140_{10} times.

The count is displayed by moving (B) to the accumulator and then to the output port. The instruction JMP causes an unconditional jump for the next count in register B, forming a continuous loop to count from FFH to 00H. After the count reaches zero, (B) is decremented, becoming FFH, and the counting cycle is repeated.

PROGRAM OUTPUT

When the program is executed, the actual output seen may vary according to the device used as the output for the display. The eye cannot see the changes in a count with a 1 ms delay. If the output port has eight LEDs, the LEDs representing the low-order bits will appear to be on continuously, and the LEDs with high-order bits will go on and off according to the count. If the output port is a seven-segment display, all segments will appear to be on; a slight flicker in the display can be noticed. However, the count and the delay can be measured on an oscilloscope.

8.3 ILLUSTRATIVE PROGRAM: ZERO-TO-NINE (MODULO TEN)* COUNTER

PROBLEM STATEMENT

Write a program to count from 0 to 9 with a one-second delay between each count. At the count of 9, the counter should reset itself to 0 and repeat the sequence continuously. Use register pair HL to set up the delay, and display each count at one of the output ports. Assume the clock frequency of the microcomputer is 1 MHz.

Instructions Review the following instructions.

- ☐ LXI: Load Register Pair Immediate
- ☐ DCX: Decrement Register Pair
- ☐ INX: Increment Register Pair

These instructions manipulate 16-bit data by using registers in pairs (BC, DE, and HL). However, the instructions DCX and INX do not affect flags to reflect the outcome of the operation. Therefore, additional instructions must be used to set the flags.

PROBLEM ANALYSIS

The problem is similar to that in the Illustrative Program for a hexadecimal counter (Section 8.2) except in two respects: the counter is an up-counter (counts *up* to the digit 9), and the delay is too long to use just one register.

1. The counter is set up by loading a register with the appropriate number and incrementing it until it reaches the digit 9. When the counter register reaches the final count, it is reset to zero. This is an additional step compared to the Illustrative Program of Section 8.2 in which the counter resets itself. (Refer to the flowchart in Figure 8.7 to see how each count is checked against the final count.)

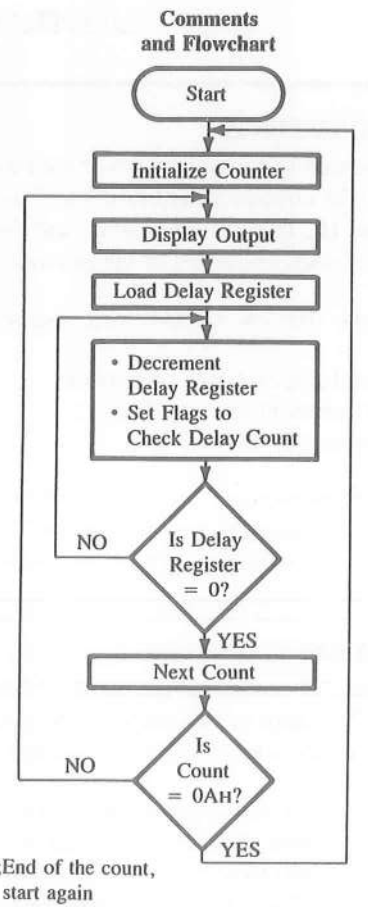
*The counter goes through ten different states (0 to 9) and is called a *modulo ten* counter.

2. The 1-second delay between each count is set up by using a register pair, as explained in Section 8.1.2. The delay calculations are shown later.

FLOWCHART AND PROGRAM

The flowchart in Figure 8.7 shows three loops: one for the counter (outer) loop on the left, the second for the delay (inner) loop on the left, and the third to reset the counter (the loop on the right). This flowchart is similar to the flowchart in Figure 8.4. The flowchart indicates that the number is displayed immediately after the initialization; this is different from the flowchart of Figure 8.6, in which the number is displayed at the end of the program.

Memory Address	Hex Code	Label	Mnemonics	T	
HI-LO XX00					
01	06	START:	MVI B,00H		
02	00				
03	D3	DSPLAY:	OUT PORT#	10	T_0
04	PORT#				
05	21		LXI H,16-Bit	10	T_0
06	LO*				
07	HI				
08	2B	LOOP:	DCX H	6	T_L: 24 T-states
09	7D		MOV A,L	4	
0A	B4		ORA H	4	
0B	C2		JNZ LOOP	10/7	
0C	08				
0D	XX†				
0E	04		INR B	4	T_0
0F	78		MOV A,B	4	
10	FE		CPI 0AH	7	
11	0A				
12	C2		JNZ DSPLAY	10/7	
13	03				
14	XX†				
15	CA		JZ START		
16	00				
17	XX				



*Enter 16-bit delay count in place of LO and HI, appropriate to the clock period in your system.
†Enter high-order address (page number) of your R/W memory.

FIGURE 8.7
Program and Flowchart for a Zero-to-Nine Counter

The counter is incremented at the end of the program and checked against count 0AH (final count + 1). If the counter has not reached number 0AH, the count is displayed (outer loop on the left); otherwise, it is reset by the loop on the right (Figure 8.7).

PROGRAM DESCRIPTION

Register B is used as a counter, and register pair HL is used for the delay. The significant differences between this program and the Illustrative Program for a hexadecimal counter (Section 8.2) are as follows:

1. Register pair HL contains a 16-bit number that can be manipulated in two ways: first, as a 16-bit number, and second, as two 8-bit numbers. The instruction DCX views the HL register as one register with a 16-bit number. On the other hand, the instructions MOV A,L and ORA H treat the contents of the HL registers as two separate 8-bit numbers.
2. In the delay loop, the sequence is repeated by the instruction JNZ (Jump on No Zero) until the count becomes zero, thus providing a delay of 1 second. However, the instruction DCX does not set the Zero flag. Therefore, the instruction JNZ would be unable to recognize when the count has reached zero, and the program would remain in a continuous loop.

   In this program, the instruction ORA is used to set the Zero flag. The purpose here is to check when the 16-bit number in register pair HL has reached zero. This is accomplished by ORing the contents of register L with the contents of register H. However, the contents of two registers cannot be ORed directly. The instruction MOV A,L loads the accumulator with (L), which is then ORed with (H) by the instruction ORA H. The Zero flag is set only when both registers are zero.
3. The instruction CPI,0AH (Compare Immediate with 0AH) checks the contents of the counter (register B) in every cycle. When register B reaches the number 0AH, the program sequence is redirected to reset the counter without displaying the number 0AH.
4. The time required to reset the counter to zero, indicated by the right-hand loop in the flowchart, is slightly different from the time delay between each count set by the left-hand outer loop.

Delay Calculations The major delay between two counts is provided by the 16-bit number in the delay register HL (the inner loop in the flowchart). This delay is set up by using a register pair, as explained in Section 8.1.2.

$$\text{Loop Delay } T_L = 24 \text{ T-states} \times T \times \text{Count}$$
$$1 \text{ second} = 24 \times 1.0 \times 10^{-6} \times \text{Count}$$
$$\text{Count} = \frac{1}{24 \times 10^{-6}} = 41666 = \text{A2C2H}$$

The delay count A2C2H in register HL would provide approximately a 1-second delay between two counts. To achieve higher accuracy in the delay, the instructions outside the loop starting from OUT PORT# to JNZ DSPLAY and the difference of three states in the last execution of JNZ must be accounted for in the delay calculations.

The instructions outside the loop are: OUT, LXI, INR, MOV, CPI, and JNZ (DSPLAY). These instructions require 45 T-states; therefore, the delay count is calculated as follows:

$$\text{Total Delay } T_D = T_O + T_L$$
$$1 \text{ second} = (45 \times 1.0 \times 10^{-6}) + (24 \times 1.0 \times 10^{+6} \times \text{Count})$$
$$\text{Count} \approx 41665$$

The difference between the two delay counts calculated above is of very little significance in many applications.

PROGRAM OUTPUT

In an LED output port, each LED will be lit according to the binary count representing 0 to 9. In a seven-segment display, the continuous sequence of the numbers from 0 to 9 can be observed very easily because of the 1-second delay between each count. However, it will be difficult to observe a steady pattern on the oscilloscope (except on a storage scope) because the reset cycle time is slightly different from the delay time. (See Assignment 3 at the end of this chapter to run this program on your system.)

8.4 ILLUSTRATIVE PROGRAM: GENERATING PULSE WAVEFORMS

PROBLEM STATEMENT

Write a program to generate a continuous square wave with the period of 500 μs. Assume the system clock period is 325 ns, and use bit D_0 to output the square wave.

Instructions The following instructions should be reviewed and the differences between various rotate instructions observed:

- ☐ RAL: Rotate Accumulator Left Through Carry
- ☐ RAR: Rotate Accumulator Right Through Carry
- ☐ RLC: Rotate Accumulator Left
- ☐ RRC: Rotate Accumulator Right

The first two instructions, RAL and RAR, use the Carry flag as the ninth bit, and the accumulator can be viewed as a 9-bit register. The last two instructions, RLC and RRC, rotate the accumulator contents through eight positions.

In addition to using these instructions, you should review the concept of masking with the ANI instruction.

PROBLEM ANALYSIS

In this problem, the period of the square wave is 500 μs; therefore, the pulse should be on (logic 1) for 250 μs and off (logic 0) for the remaining 250 μs. The alternate pattern of 0/1 bits can be provided by loading the accumulator with the number AAH (1010 1010)

and rotating the pattern once through each delay loop. Bit D_0 of the output port is used to provide logic 0 and 1; therefore, all other bits can be masked by ANDing the accumulator with the byte 01H. The delay of 250 μs can be very easily obtained with an 8-bit delay count and one register.

PROGRAM

Memory Address HI-LO	HEX Code	Label	Mnemonics	Comments
XX00	16			
01	AA		MVI D,AA	;Load bit pattern AAH
02	7A	ROTATE:	MOV A,D	;Load bit pattern in A
03	07		RLC	;Change data from AAH to
				; 55H and vice versa
04	57		MOV D,A	;Save (A)
05	E6		ANI 01H	;Mask bits D_7–D_1
06	01			
07	D3		OUT PORT1	;Turn on or off the lights
08	PORT1			
09	06		MVI B,COUNT (7T)	;Load delay count for 250 μs
0A	COUNT			
0B	05	DELAY:	DCR B (4T)	;Next count
0C	C2		JNZ DELAY (10/7T)	;Repeat until (B) = 0
0D	0B			
0E	XX			
0F	C3		JMP ROTATE (10T)	;Go back to change logic level
10	02			
11	XX			

PROGRAM DESCRIPTION

Register D is loaded with the bit pattern AAH (1010 1010), and the bit pattern is moved into the accumulator. The bit pattern is rotated left once and saved again in register D. The accumulator contents must be saved because the accumulator is used later in the program.

The next instruction, ANI, ANDs (A) to mask all but bit D_0, as illustrated below.

(A)	→1	0	1	0	1	0	1	0
After RLC	→0	1	0	1	0	1	0	1
AND with 01H	→0	0	0	0	0	0	0	1
Remaining contents	→0	0	0	0	0	0	0	1

This shows that 1 in D_0 provides a high pulse that stays on 250 μs because of the delay. In the next cycle of the loop, bit D_0 is at logic 0 because of the Rotate instruction, and the output pulse stays low for the next 250 μs.

Delay Calculations In this problem, the pulse width is relatively small (250 μs); therefore, to obtain a reasonably accurate output pulse width, we should account for all the T-states. The total delay should include the delay in the loop and the execution time of the instructions outside the loop.

1. The number of instructions outside the loop is seven; it includes six instructions before the loop beginning at the symbol ROTATE and the last instruction JMP.

$$\text{Delay outside the Loop: } T_O = 46 \text{ T-states} \times 325 \text{ ns} = 14.95 \text{ μs}$$

2. The delay loop includes two instructions (DCR and JNZ) with 14 T-states except for the last cycle, which has 11 T-states.

$$\begin{aligned}\text{Loop Delay: } T_L &= 14 \text{ T-states} \times 325 \text{ ns} \times (\text{Count} - 1) + 11 \text{ T-states} \times 325 \text{ ns} \\ &= 4.5 \text{ μs } (\text{Count} - 1) + 3.575 \text{ μs}\end{aligned}$$

3. The total delay required is 250 μs. Therefore, the count can be calculated as follows:

$$\begin{aligned}T_D &= T_O + T_L \\ 250 \text{ μs} &= 14.95 \text{ μs} + 4.5 \text{ μs } (\text{Count} - 1) + 3.575 \text{ μs} \\ \text{Count} &= 52.4_{10} = 34\text{H}\end{aligned}$$

Program Output The output of bit D_0 can be observed on an oscilloscope; it should be a square wave with a period of 500 μs.

8.5 DEBUGGING COUNTER AND TIME-DELAY PROGRAMS

The debugging techniques discussed in Chapters 6 and 7 can be used to check errors in a counter program. The following is a list of common errors in programs similar to those illustrated in this chapter.

1. Errors in counting T-states in a delay loop. Typically, the first instruction—to load a delay register—is mistakenly included in the loop.
2. Errors in recognizing how many times a loop is repeated.
3. Failure to convert a delay count from a decimal number into its hexadecimal equivalent.
4. Conversion error in converting a delay count from decimal to hexadecimal number or vice versa.
5. Specifying a wrong Jump location.
6. Failure to set a flag, especially with 16-bit Decrement/Increment instructions.
7. Using a wrong Jump instruction.
8. Failure to display either the first or the last count.
9. Failure to provide a delay between the last and the last-but-one count.

Some of these errors are illustrated in the following program.

8.5.1 Illustrative Program for Debugging

The following program is designed to count from 100_{10} to 0 in Hex continuously, with a 1-second delay between each count. The delay is set up by using two loops—a loop within a loop. The inner loop is expected to provide approximately 100 ms delay, and it is repeated ten times, using the outer loop to provide a total delay of 1 second. The clock period of the system is 330 ns. The program includes several deliberate errors. Recognize the errors as specified in the following assignment.

		Mnemonics	T-states
1.		MVI A,64H	7
2.	DSPLAY:	OUT PORT1	10
3.	LOOP2:	MVI B,10H	7
4.	LOOP1:	LXI D,DELAY	10
5.		DCX D	6
6.		NOP	4
7.		NOP	4
8.		MOV A,D	4
9.		ORA E	4
10.		JNZ LOOP1	10/7
11.		DCR B	4
12.		JZ LOOP2	10/7
13.		DCR A	4
14.		CPI 00H	7
15.		JNZ DSPLAY	10/7

DELAY CALCULATIONS

$$\text{Delay in LOOP1} = \text{Loop T-states} \times \text{Count} \times \text{Clock period } (330 \times 10^{-9})$$

$$100 \text{ ms} = 32 \text{ T} \times \text{Count} \times 330 \times 10^{-9}$$

$$\text{DELAY COUNT} = \frac{100 \times 10^{-3}}{32 \times 330 \times 10^{-9}}$$

$$= 9470$$

This delay calculation ignores the initial T-states in loading the count and the difference of T-states in the last execution of the conditional Jump instruction.

DEBUGGING QUESTIONS

1. Examine LOOP1. Is the label LOOP1 at the appropriate location? What is the effect of the present location on the program?
2. What is the appropriate place for the label LOOP1?
3. Is the delay count accurate?
4. What is the effect of instruction 8 (MOV A,D) on the count?
5. Should instruction 3 be part of LOOP2?
6. Is the byte in register B (instruction 3) accurate?
7. Calculate T-states in the outer loop using the appropriate place for the label LOOP2. (Do not include the T-states of LOOP1.)

8. Is there any need for instruction 14 (CPI)?
9. Is there any need for an additional instruction, such as number 16?
10. What is the effect of instruction 12 (JZ) on the program?
11. Calculate the total delay in LOOP2 inclusive of LOOP1 if the byte in register B = 0AH.
12. Assuming instruction 8 is necessary, make appropriate changes in instructions 1 and 13.
13. Calculate the time delay between the display of two consecutive counts.
14. Will this program display the last count, assuming the other errors are corrected?

SUMMARY

- ☐ Counters and time delays can be designed using software.
- ☐ Time delays are designed simply by loading a count in a register or a register pair and decrementing the count by setting a loop until the count reaches zero. Time delay is determined by the number of T-states in a delay loop, the clock frequency, and the number of times the loop is repeated.
- ☐ Counters are designed using techniques similar to those used for time delays. A counter design generally includes a delay loop.
- ☐ In the 8085 microprocessor, 8-bit registers can be combined as register pairs (B and C, D and E, and H and L) to manipulate 16-bit data. Furthermore, the contents of each register can be examined separately even if registers are being used as register pairs.
- ☐ Sixteen-bit instructions such as DCX and INX do not affect flags; therefore, some other technique must be used to set flags.

QUESTIONS AND PROGRAMMING ASSIGNMENTS*

1. In Figure 8.2, calculate the loop delay T_L if register C contains 00H and the system clock frequency is 3.072 MHz. Adjust the delay calculations to account for seven T-states of the JNZ instruction in the last iteration.
2. In Section 8.1.2, load register pair BC with 8000H, and calculate the loop delay T_L if the system clock frequency is 3.072 MHz (ignore three T-state difference of the last cycle).
3. In Section 8.1.2, load register pair BC with 0000H and calculate the total delay T_D if the system clock period is 325 ns (adjust for the last cycle).

*The illustrative programs shown in this chapter and these assignments can be verified on a single-board microcomputer with a display output port.

4. Calculate the delay in the following loop, assuming the system clock period is 0.33 μs:

Label	Mnemonics	8085 T-states
	LXI B,12FFH	10
DELAY:	DCX B	6
	XTHL	16
	XTHL	16
	NOP	4
	NOP	4
	MOV A,C	4
	ORA B	4
	JNZ DELAY	10/7

5. In Figure 8.3, load register C with 00H and register B with C8H. Calculate the loop delay in LOOP1 and LOOP2 (clock period = 325 ns).
6. Specify the number of times the following loops are executed.

```
a.      MVI A,17H      b.      MVI A,17H      c.      LXI B,1000H
  LOOP: ORA A            LOOP: RAL              LOOP: DCX B
        RAL                    ORA A                  NOP
        JNC LOOP               JNC LOOP               JNZ LOOP
```

7. Specify the number of times the following loops are executed.

```
a. LOOP: MVI B,64H     b.      ORA A          c.      MVI A,17H
         NOP                   MVI B,64H       LOOP: ORA A
         DCR B           LOOP: DCR B                 RRC
         JNZ LOOP              JNC LOOP               JNC LOOP
```

8. Calculate the time delay in the Illustrative Program for a hexadecimal counter (Section 8.2), assuming a count of CFH in register C.
9. Recalculate the delay in the Illustrative Program for a zero-to-nine counter (Section 8.3) using the clock frequency of your system.
10. Calculate the COUNT to obtain a 100 μs loop delay, and express the value in Hex. (Use the clock frequency of your system.)

		T-states
	MVI B,COUNT	
LOOP:	NOP	4
	NOP	4
	DCR B	4
	JNZ LOOP	10/7

11. In Section 8.1.2, calculate the value of the 16-bit number that should be loaded in register BC to obtain the loop delay of 250 ms if the system clock period is 325 ns. Does the value change if it is calculated with seven T-states for the JNZ instruction in the last cycle?

12. Calculate the 16-bit count to be loaded in register DE to obtain the loop delay of two seconds in LOOP2 (use the clock period of your system and ignore the execution time of the first instruction MVI B).

```
        MVI B,14H      (7)     ;Count for outer loop
LOOP2:  LXI D,16-bit   (10)    ;Count for LOOP1
LOOP1:  DCX D          (6)
        MOV A,D        (4)
        ORA E          (4)
        JNZ LOOP1      (10/7)
        DCR B          (4)
        JNZ LOOP2      (10/7)
```

Use the clock frequency of your system in the following assignments.

13. Write a program to count from 0 to 20H with a delay of 100 ms between each count. After the count 20H, the counter should reset itself and repeat the sequence. Use register pair DE as a delay register. Draw a flowchart and show your calculations to set up the 100 ms delay.
14. Design an up-down counter to count from 0 to 9 and 9 to 0 continuously with a 1.5-second delay between each count, and display the count at one of the output ports. Draw a flowchart and show the delay calculations.
15. Write a program to turn a light on and off every 5 seconds. Use data bit D_7 to operate the light.
16. Write a program to generate a square wave with period of 400 μs. Use bit D_0 to output the square wave.
17. Write a program to generate a rectangular wave with a 200 μs on-period and a 400 μs off-period.
18. A railway crossing signal has two flashing lights run by a microcomputer. One light is connected to data bit D_7 and the second light is connected to data bit D_6. Write a program to turn each signal light alternately on and off at an interval of 1 second.

9

Stack and Subroutines

The **stack** is a group of memory locations in the R/W memory that is used for temporary storage of binary information during the execution of a program. The starting memory location of the stack is defined in the main program, and space is reserved, usually at the high end of the memory map. The method of information storage resembles a stack of books. The contents of each memory location are, in a sense, "stacked"—one memory location above another—and information is retrieved starting from the top. Hence, this particular group of memory locations is called the *stack.* This chapter introduces the stack instructions in the 8085 set.

The latter part of this chapter deals with the subroutine technique, which is frequently used in programs. A **subroutine** is a group of instructions that performs a subtask (e.g., time delay or arithmetic operation) of repeated occurrence. The subroutine is written as a separate unit, apart from the main program, and the microprocessor transfers the program execution from the main program to the subroutine whenever it is called to perform the task. After the completion of the subroutine task, the microprocessor returns to the main program. The subroutine technique eliminates the need to write a subtask repeatedly; thus, it uses memory more efficiently. Before implementing the subroutine technique, the stack must be defined; the stack is used to store the memory address of the instruction in the main program that follows the subroutine call.

The stack and the subroutine offer a great deal of flexibility in writing programs. A large software project is usually divided into subtasks called **modules.** These modules are developed independently as subroutines by different programmers. Each programmer can use all the microprocessor registers to write a subroutine without affecting the other parts of the program. At the beginning of the subroutine module, the register contents of the main program are stored on the stack, and these register contents are retrieved before returning to the main program.

This chapter includes two illustrative programs: The first illustrates the use of the stack-related instructions to examine and manipulate the flags; and the second illustrates the subroutine technique in a traffic-signal controller.

OBJECTIVES

- ☐ Define the stack, the stack pointer (register), and the program counter, and describe their uses.
- ☐ Explain how information is stored and retrieved from the stack using the instructions PUSH and POP and the stack pointer register.
- ☐ Demonstrate how the contents of the flag register can be displayed and how a given flag can be set or reset.
- ☐ Define a subroutine and explain its uses.
- ☐ Explain the sequence of a program execution when a subroutine is called and executed.
- ☐ Explain how information is exchanged between the program counter and the stack, and identify the contents of the stack pointer register when a subroutine is called.
- ☐ List and explain conditional Call and Return instructions.
- ☐ Illustrate the concepts in the following subroutines: multiple calling, nesting, and common ending.
- ☐ Compare similarities and differences between PUSH/POP and CALL/RET instructions.

9.1 STACK

The **stack** in an 8085 microcomputer system can be described as a set of memory locations in the R/W memory, specified by a programmer in a main program. These memory locations are used to store binary information (bytes) temporarily during the execution of a program.

The beginning of the stack is defined in the program by using the instruction LXI SP, which loads a 16-bit memory address in the stack pointer register of the microprocessor. Once the stack location is defined, storing of data bytes begins at the memory address that is one less than the address in the stack pointer register. For example, if the stack pointer register is loaded with the memory address 2099H (LXI SP,2099H), the storing of data bytes begins at 2098H and continues in reversed numerical order (decreasing memory addresses such as 2098H, 2097H, etc.). Therefore, as a general practice, the stack is initialized at the highest available memory location to prevent the program from being destroyed by the stack information. The size of the stack is limited only by the available memory.

Data bytes in the register pairs of the microprocessor can be stored on the stack (two at a time) in reverse order (decreasing memory address) by using the instruction PUSH. Data bytes can be transferred from the stack to respective registers by using the instruction POP. The stack pointer register tracks the storage and retrieval of the information. Because two data bytes are being stored at a time, the 16-bit memory address in the stack pointer register is decremented by two; when data bytes are retrieved, the address is incremented by two. An address in the stack pointer register indicates that the next two memory locations (in descending numerical order) can be used for storage.

The stack is shared by the programmer and the microprocessor. The programmer can store and retrieve the contents of a register pair by using PUSH and POP instructions. Similarly, the microprocessor automatically stores the contents of the program counter when a subroutine is called (to be discussed in the next section). The instructions necessary for using the stack are explained below.

INSTRUCTIONS

Opcode	Operand	
LXI	SP,16-bit	☐ Load Stack Pointer ☐ Load the stack pointer register with a 16-bit address. The LXI instructions were discussed in Chapter 7
PUSH	Rp	Store Register Pair on Stack ☐ This is a 1-byte instruction
PUSH	B D H PSW	☐ It copies the contents of the specified register pair on the stack as described below ☐ The stack pointer register is decremented, and the contents of the high-order register (e.g., register B) are copied in the location shown by the stack pointer register ☐ The stack pointer register is again decremented, and the contents of the low-order register (e.g., register C) are copied in that location ☐ The operands B, D, and H represent register pairs BC, DE, and HL, respectively ☐ The operand PSW represents Program Status Word, meaning the contents of the accumulator and the flags
POP	Rp	Retrieve Register Pair from Stack ☐ This is a 1-byte instruction
POP	B D H PSW	☐ It copies the contents of the top two memory locations of the stack into the specified register pair ☐ First, the contents of the memory location indicated by the stack pointer register are copied into the low-order register (e.g., register L), and then the stack pointer register is incremented by 1 ☐ The contents of the next memory location are copied into the high-order register (e.g., register H), and the stack pointer register is again incremented by 1

All three of these instructions belong to the data transfer (copy) group; thus, the contents of the source are not modified, and no flags are affected.

Example 9.1

In the following set of instructions (illustrated in Figure 9.1), the stack pointer is initialized, and the contents of register pair HL are stored on the stack by using the PUSH instruction. Register pair HL is used for the delay counter (actual instructions are not

shown); at the end of the delay counter, the contents of HL are retrieved by using the instruction POP. Assuming the available user memory ranges from 2000H to 20FFH, illustrate the contents of various registers when PUSH and POP instructions are executed.

Solution

In this example, the first instruction—LXI SP,2099H—loads the stack pointer register with the address 2099H (Figure 9.1). This instruction indicates to the microprocessor that memory space is reserved in the R/W memory as the stack and that the locations beginning at 2098H and moving upward can be used for temporary storage. This instruction also suggests that the stack can be initialized anywhere in the memory; however, the stack location should not interfere with a program. The next instruction—LXI H—loads data in the HL register pair, as shown in Figure 9.1.

When instruction PUSH H is executed, the following sequence of data transfer takes place. After the execution, the contents of the stack and the register are as shown in Figure 9.2.

1. The stack pointer is decremented by one to 2098H, and the contents of the H register are copied to memory location 2098H.
2. The stack pointer register is again decremented by one to 2097H, and the contents of the L register are copied to memory location 2097H.
3. The contents of the register pair HL are not destroyed; however, HL is made available for the delay counter.

After the delay counter, the instruction POP H restores the original contents of the register pair HL, as follows. Figure 9.3 illustrates the contents of the stack and the registers following the POP instruction.

1. The contents of the top of the stack location shown by the stack pointer are copied in the L register, and the stack pointer register is incremented by one to 2098H.
2. The contents of the top of the stack (now it is 2098H) are copied in the H register, and the stack pointer is incremented by one.

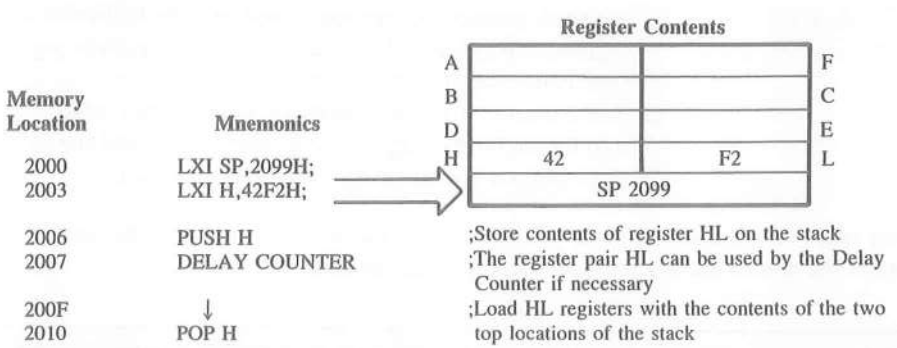


FIGURE 9.1
Instructions and Register Contents in Example 9.1

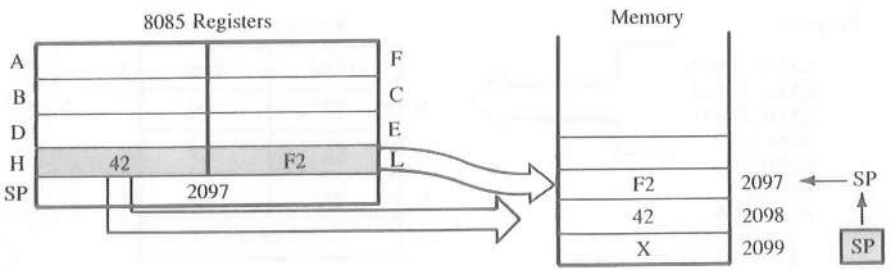


FIGURE 9.2
Contents on the Stack and in the Registers After the PUSH Instruction

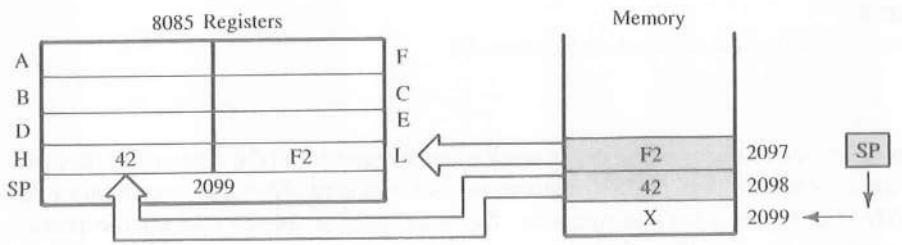


FIGURE 9.3
Contents on the Stack and in the Registers After the POP Instruction

3. The contents of memory locations 2097H and 2098H are not destroyed until some other data bytes are stored in these locations.

Example 9.2

The available user memory ranges from 2000H to 23FFH. A program of data transfer and arithmetic operations is stored in memory locations from 2000H to 2050H, and the stack pointer is initialized at location 2400H. Two sets of data are stored, starting at locations 2150H and 2280H (not shown in Figure 9.4). Registers HL and BC are used as memory pointers to the data locations. A segment of the program is shown in Figure 9.4.

1. Explain how the stack pointer can be initialized at one memory location beyond the available user memory.
2. Illustrate the contents of the stack memory and registers when PUSH and POP instructions are executed, and explain how memory pointers are exchanged.
3. Explain the various contents of the user memory.

Solution

1. The program initializes the stack pointer register at location 2400H, one location beyond the user memory (Figure 9.4). This procedure is valid because the initialized location is never used for storing information. The instruction PUSH first decrements the stack pointer register, and then stores a data byte.

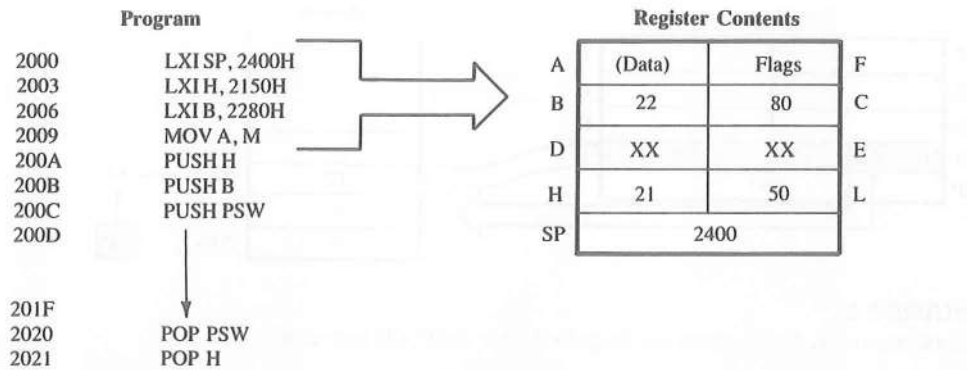


FIGURE 9.4
Instructions and Register Contents in Example 9.2

2. Figure 9.5 shows the contents of the stack pointer register and the contents of the stack locations after the three PUSH instructions are executed. After the execution of the PUSH (H, B, and PSW) instructions, the stack pointer moves upward (decreasing memory locations) as the information is stored. Thus the stack can grow upward in the user memory even to the extent of destroying the program.

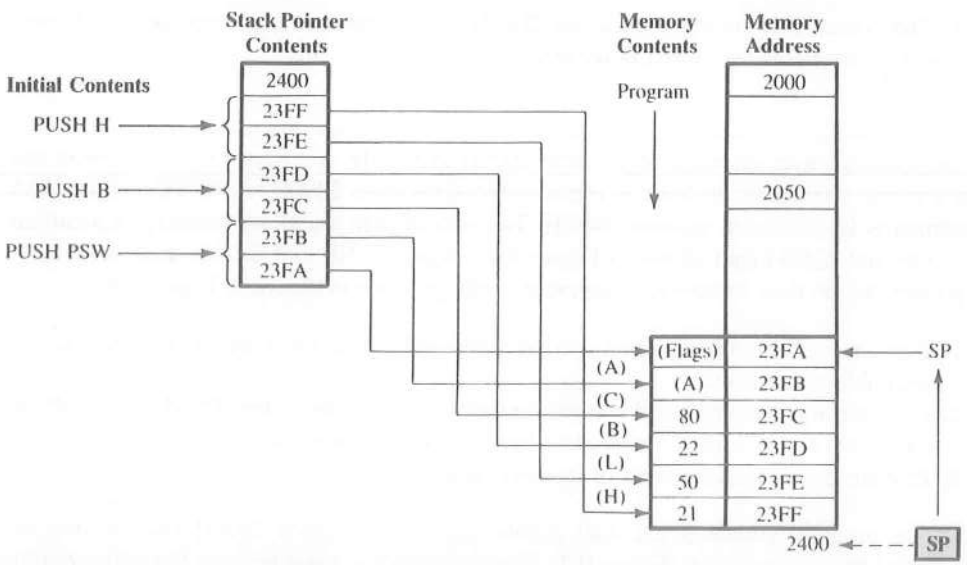


FIGURE 9.5
Stack Contents After the Execution of PUSH Instructions

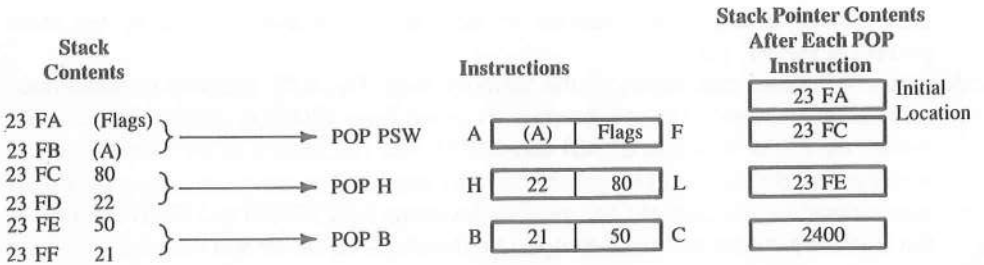
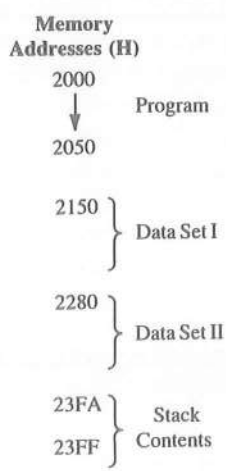


FIGURE 9.6
Register Contents After the Execution of POP Instructions

Figure 9.6 shows how the contents of various register pairs are retrieved. To restore the original contents in the respective registers, follow the sequence Last-In–First-Out (LIFO). In the example, the register contents were pushed on the stack in the order of HL, BC, and PSW. The contents should have been restored in the order of PSW, BC, and HL. However, the order is altered in this example to demonstrate how register contents are exchanged.

The instruction POP PSW copies the contents of the two top locations to the flag register and the accumulator, respectively, and increments the stack pointer by two to 23FCH. The next instruction, POP H, takes the contents of the top two locations (23FC and 23FD), and copies them in registers L and H, respectively, while incrementing the stack pointer by two to 23FEH. The instruction POP B copies the contents of the next two locations in registers C and B, incrementing the stack pointer to 2400H. By reversing the positions of two instructions, POP H and POP B, the contents of the BC pair are exchanged with those of the HL pair. It is important to remember that the instruction POP H does not restore the original contents of the HL

FIGURE 9.7
R/W Memory Contents



pair; instead it copies the contents of the top two locations shown by the stack pointer in the HL pair.

3. Figure 9.7 shows the sketch of the memory map. The R/W memory includes three types of information. The user program is stored from 2000H to 2050H. The data are stored, starting at locations 2150H and 2280H. The last section of the user memory is initialized as the stack where register contents are stored as necessary, using the PUSH instructions. In this example, the memory locations from 23FFH to 23FAH are used as the stack, which can be extended up to the locations of the second data set.

9.1.1 Review of Important Concepts

The following points can be summarized from the preceding examples:

1. Memory locations in R/W memory can be employed as temporary storage for information by initializing (loading) a 16-bit address in the stack pointer register; these memory locations are called the stack. The terms *stack* and *stack pointer* appear similar; however, they are not the same. The stack is memory locations in R/W memory; the stack pointer is a 16-bit register in the 8085 microprocessor.
2. Read/Write memory generally is used for three different purposes:
   a. to store programs or instructions;
   b. to store data;
   c. to store information temporarily in defined memory locations called the stack during the execution of the program.
3. The stack space grows upward in the numerically decreasing order of memory addresses.
4. The stack can be initialized anywhere in the user memory map. However, as a general practice, the stack is initialized at the highest user memory location so that it will be less likely to interfere with a program.
5. A programmer can employ the stack to store contents of register pairs by using the instruction PUSH and can restore the contents of register pairs by using the instruction POP. The address in the stack pointer register always points to the top of the stack, and the address is decremented or incremented as information is stored or retrieved.
6. The contents of the stack pointer can be interpreted as the address of the memory location that is already used for storage. The retrieval of bytes begins at the address in the stack pointer; however, the storage begins at the next memory location (in the decreasing order).
7. The storage and retrieval of data bytes on the stacks should follow the LIFO (Last-In–First-Out) sequence. Information in stack locations is not destroyed until new information is stored in those locations.

9.1.2 Illustrative Program: Resetting and Displaying Flags

PROBLEM STATEMENT

Write a program to perform the following functions:

1. Clear all the flags.
2. Load 00H in the accumulator, and demonstrate that the Zero flag is not affected by the data transfer instruction.
3. Logically OR the accumulator with itself to set the Zero flag, and display the flag at PORT1 or store all the flags on the stack.

PROBLEM ANALYSIS

The problem concerns examining the Zero flag after instructions have been executed. There is no direct way of observing the flags; however, they can be stored on the stack by using the instruction PUSH PSW. The contents of the flag register can be retrieved in any one of the registers by using the instruction POP, and the flags can be displayed at an output port. In this example, the result to be displayed is different from the result to be stored on the stack memory. To display only the Zero flag, all other flags should be masked; however, the masking is not necessary to store all the flags.

PROGRAM

Memory Address	Machine Code	Instructions	Comments
XX00	31	LXI SP,XX99H	;Initialize the stack
01	99		
02	XX		
03	2E	MVI L,00H	;Clear L
04	00		
05	E5	PUSH H	;Place (L) on stack
06	F1	POP PSW	;Clear flags
07	3E	MVI A,00H	;Load 00H
08	00		
09	F5	PUSH PSW	;Save flags on stack
0A	E1	POP H	;Retrieve flags in L
0B	7D	MOV A,L	
0C	D3	OUT PORT0	;Display flags
0D	PORT0		
0E	3E	MVI A,00H	;Load 00H again
0F	00		
10	B7	ORA A	;Set flags and reset CY, AC
11	F5	PUSH PSW	;Save flags on stack
12	E1	POP H	;Retrieve flags in L
13	7D	MOV A,L	
14	E6	ANI 40H	;Mask all flags except Z
15	40		
16	D3	OUT PORT1	
17	PORT1		
18	76	HLT	;End of program

Storing in Memory: Alternative to Output Display

```
XX0A   3E   MVI A,00H   ;Load 00H again
  0B   00
  0C   B7   ORA A       ;Set flags and reset CY and AC
  0D   F5   PUSH PSW    ;Save flags on stack
  0E   76   HLT         ;End of program
```

Program Description The stack pointer register is initialized at XX99H. The instruction MVI L clears (L), and (L) is placed on the stack, which is subsequently placed into the flag register to clear all the flags.

To verify the flags after the execution of the MVI A instruction, the PUSH and POP instructions are used in the same way as these instructions were used to clear the flags, and the flags are displayed at PORT0. Similarly, the Zero flag is displayed at PORT1 after the instructions MVI and ORA.

Program Output Data transfer (copy) instructions do not affect the flags; therefore, no flags should be set after the instruction MVI A, even if (A) is equal to zero. PORT0 should display 00H. However, the instruction ORA will set the Zero and the Parity flags to reflect the data conditions in the accumulator, and it also resets the CY and AC flags. In the flag register, bit D_6 represents the Z flag, and the ANI instruction masks all flags except the Z flag. PORT1 should display 40H as shown below.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	
0	1	0	0	0	1	0	0	= 40H
S	Z		AC		P		CY	

Storing Output in Memory If output ports are not available, the results can be stored in the stack memory. The machine code (F5) at memory location XX09H saves the flags affected by the instruction MVI A,00H. Then the instructions can be modified starting from memory location XX0AH. The alternative set of instructions is shown above; it sets the flags (using ORA instruction), and saves them on the stack without masking. The result (44H) includes the parity flag. The contents of the stack locations should be as shown in Figure 9.8.

Instructions	Stack Memory	Contents	
	XX99		← Stack Pointer Initialization
XX09 PUSH PSW:	XX98	(A) = 00H	
	XX97	(F) = 00H	
XX0D PUSH PSW:	XX96	(A) = 00H	
	XX95	(F) = 44H	

FIGURE 9.8
Output Stored in Stack Memory

9.2 SUBROUTINE

A **subroutine** is a group of instructions written separately from the main program to perform a function that occurs repeatedly in the main program. For example, if a time delay is required between three successive events, three delays can be written in the main program. To avoid repetition of the same delay instructions, the subroutine technique is used. Delay instructions are written once, separately from the main program, and are called by the main program when needed.

The 8085 microprocessor has two instructions to implement subroutines: CALL (call a subroutine), and RET (return to main program from a subroutine). The CALL instruction is used in the main program to call a subroutine, and the RET instruction is used at the end of the subroutine to return to the main program. When a subroutine is called, the contents of the program counter, which is the address of the instruction following the CALL instruction, is stored on the stack and the program execution is transferred to the subroutine address. When the RET instruction is executed at the end of the subroutine, the memory address stored on the stack is retrieved, and the sequence of execution is resumed in the main program. This sequence of events is illustrated in Example 9.3.

INSTRUCTIONS

Opcode	Operand	
CALL	16-bit memory address of a subroutine	Call Subroutine Unconditionally ☐ This is a 3-byte instruction that transfers the program sequence to a subroutine address ☐ Saves the contents of the program counter (the address of the next instruction) on the stack ☐ Decrements the stack pointer register by two ☐ Jumps unconditionally to the memory location specified by the second and third bytes. The second byte specifies a line number and the third byte specifies a page number ☐ This instruction is accompanied by a return instruction in the subroutine
RET		Return from Subroutine Unconditionally ☐ This is a 1-byte instruction ☐ Inserts the two bytes from the top of the stack into the program counter and increments the stack pointer register by two ☐ Unconditionally returns from a subroutine

The conditional Call and Return instructions will be described later in the chapter.

Example 9.3 Illustrate the exchange of information between the stack and the program counter for the following program if the available user memory ranges from 2000H to 23FFH.

Memory Address		
2000	LXI SP,2400H	;Initialize the stack pointer at 2400H
↓	↓	
2040	CALL 2070H	;Call the subroutine located at 2070H. This is
2041		; a 3-byte instruction
2042		
2043	NEXT INSTRUCTION	;The address of the next instruction following
↓	↓	; the CALL instruction
205F	HLT	;End of the main program
2070	First Subroutine	;Beginning of the subroutine
↓	Instruction	
	↓	
207F	RET	;End of the subroutine
2080	↓	
↓	Other Subroutines	
	↓	
2398	Empty Space	
23FF	↓	
2400		;The stack is initialized at 2400H

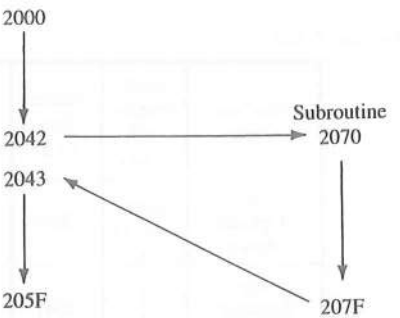
Solution After reviewing the above program note the following points:

1. The available user memory is from 2000H to 23FFH (1024 or 1K bytes); however, the stack pointer register is initialized at 2400H, one location beyond the user memory. This allows maximum use of the memory because the actual stack begins at 23FFH. The stack can expand up to the location 2398H without overlapping with the program.
2. The main program is stored at memory locations from 2000H to 205FH.
3. The CALL instruction is located at 2040H to 2042H (3-byte instruction). The next instruction is at 2043H.
4. The subroutine begins at the address 2070H and ends at 207FH.

PROGRAM EXECUTION

The sequence of the program execution and the events in the execution of the CALL and subroutine are shown in Figures 9.9 and 9.10.

FIGURE 9.9
Subroutine Call and Program Transfer



The program execution begins at 2000_H, continues until the end of CALL 2042_H, and transfers to the subroutine at 2070_H. At the end of he subroutine, after executing the RET instruction, it comes back to the main program at 2043H and continues.

CALL EXECUTION

Memory Address	Machine Code	Mnemonics	Comments
2040	CD	CALL 2070H	;Call subroutine located at the memory
2041	70		; location 2070H
2042	20		
2043	NEXT	INSTRUCTION	

The sequence of events in the execution of the CALL instruction by the 8085 is shown in Figure 9.10. The instruction requires five machine cycles and eighteen T-states. The sequence of events in each machine cycle is as follows.

1. M_1—Opcode Fetch: In this machine cycle, the contents of the program counter (2040H) are placed on the address bus, and the instruction code CD is fetched using the data bus. At the same time, the program counter is upgraded to the next memory address, 2041H. After the instruction is decoded and executed, the stack pointer register is decremented by one to 23FFH.
2. M_2 and M_3—Memory Read: These are two Memory Read cycles during which the 16-bit address (2070H) of the CALL instruction is fetched. The low-order address 70H is fetched first and placed in the internal register Z. The high-order address 20H is fetched next, and placed in register W. During M_3, the program counter is upgraded to 2043H, pointing to the next instruction.
3. M_4 and M_5—Storing of Program Counter: At the beginning of the M_4 cycle, the normal operation of placing the contents of the program counter on the address bus is suspended; instead, the contents of the stack pointer register 23FFH are placed on the address bus. The high-order byte of the program counter (PCH = 20H) is placed on the data bus and stored in the stack location 23FFH. At the same time, the stack pointer register is decremented to 23FEH.

   During machine cycle M_5, the contents of the stack pointer 23FEH are placed on the address bus. The low-order byte of the program counter (PCL = 43H) is placed on the data bus and stored in stack location 23FEH.

Instruction: CALL 2070H

Memory Address	Code (H)
2040	CD
2041	70
2042	20

Machine Cycles	Stack Pointer (SP) 2400	Address Bus (AB)	Program Counter (PCH) (PCL)	Data Bus (DB)	Internal Registers (W) (Z)
M_1 Opcode Fetch	23FF (SP–1)	2040	20 41	CD Opcode	—
M_2 Memory Read		2041	20 42	70 Operand	→ 70
M_3 Memory Read	23FF	2042	20 43	20 Operand	→ 20
M_4 Memory Write	23FE (SP–2)	23FF	20 43	20 → (PCH)	
M_5 Memory Write	23FE	23FE	20 43	43 → (PCL)	(20) (70)
M_1 Opcode Fetch of Next Instruction		2070 → (W)(Z)	→ 2071		(2070) → (W)(Z)

FIGURE 9.10
Data Transfer During the Execution of the CALL Instruction

4. Next Instruction Cycle: In the next instruction cycle, the program execution sequence is transferred to the CALL location 2070H by placing the contents of W and Z registers (2070H) on the address bus. During M_1 of the next instruction cycle, the program counter is upgraded to location 2071 (W,Z + 1).

In summary: After the CALL instruction is fetched, the 16-bit address (the operand) is read during M_2 and M_3 and stored temporarily in W/Z registers. (Examine the contents of the address bus and the data bus in Figure 9.10.) In the next two cycles, the contents of the program counter are stored on the stack. This is the address where the microprocessor will continue the execution of the program after the completion of the subroutine. Figure 9.11 shows the contents of the program counter, the stack pointer register, and the stack during the execution of the CALL instruction.

RET EXECUTION

At the end of the subroutine, when the instruction RET is executed, the program execution sequence is transferred to the memory location 2043H. The address 2043H was

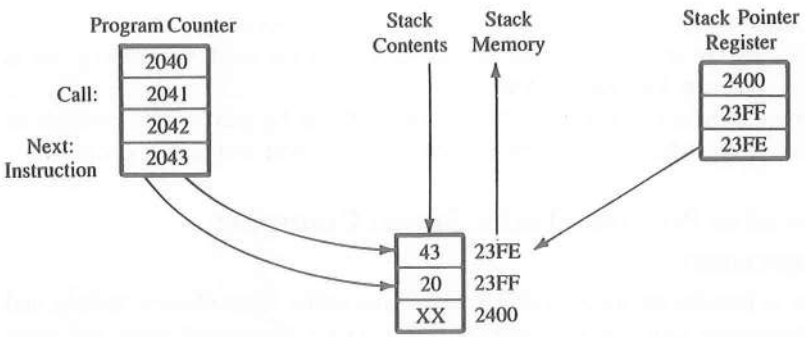


FIGURE 9.11
Contents of the Program Counter, the Stack Pointer, and the Stack During the Execution of the CALL Instruction

stored in the top two locations of the stack (23FEH and 23FFH) during the CALL instruction. Figure 9.12 shows the sequence of events that occurs as the instruction RET is executed.

M_1 is a normal Opcode Fetch cycle. However, during M_2 the contents of the stack pointer register are placed on the address bus, rather than those of the program counter. Data byte 43H from the top of the stack is fetched and stored in the Z register, and the

Instruction: RET

Memory Address	Code (H)
207F	C9

Contents of Stack Memory	
23FE	43
23FF	20

Machine Cycles	Stack Pointer (23FE)	Address Bus (AB)	Program Counter	Data Bus (DB)	Internal Registers (W) (Z)
M_1 Opcode Fetch	23FE	207F	2080	C9 Opcode	
M_2 Memory Read	23FF	23FE		43 (Stack) →	43
M_3 Memory Read	2400	23FF		20 (Stack−1) →	20
M_1 Opcode Fetch of Next Instruction		2043 (W) (Z) ←	2044		2043 (W) (Z)

FIGURE 9.12
Data Transfer During the Execution of the RET Instruction

stack pointer register is upgraded to the next location, 23FFH. During M_2, the next byte—20H—is copied from the stack and stored in register W, and the stack pointer register is again upgraded to the next location, 2400H.

The program sequence is transferred to location 2043H by placing the contents of the W/Z registers on the address bus at the beginning of the next instruction cycle.

9.2.1 Illustrative Program: Traffic Signal Controller

PROBLEM STATEMENT

Write a program to provide the given on/off time to three traffic lights (Green, Yellow, and Red) and two pedestrian signs (WALK and DON'T WALK). The signal lights and signs are turned on/off by the data bits of an output port as shown below:

Lights	Data Bits	On Time
1. Green	D_0	15 seconds
2. Yellow	D_2	5 seconds
3. Red	D_4	20 seconds
4. WALK	D_6	15 seconds
5. DON'T WALK	D_7	25 seconds

The traffic and pedestrian flow are in the same direction; the pedestrian should cross the road when the Green light is on.

PROBLEM ANALYSIS

The problem is primarily concerned with providing various time delays for a complete sequence of 40 seconds. The on/off times for the traffic signals and pedestrian signs are as follows:

Time Sequence in Seconds	DON'T WALK	WALK		Red		Yellow		Green		Hex Code
0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0		
(15) ↓										
15	0	1	0	0	0	0	0	1	=	41H
(5) ↓										
20	1	0	0	0	0	1	0	0	=	84H
(20) ↓	↓									
40	1	0	0	1	0	0	0	0	=	90H

The Green light and the WALK sign can be turned on by sending data byte 41H to the output port. The 15-second delay can be provided by using a 1-second subroutine and

a counter with a count of 15_{10}. Similarly, the next two bytes, 84H and 90H, will turn on/off the appropriate lights/signs as shown in the flowchart (Figure 9.13). The necessary time delays are provided by changing the values of the count in the counter.

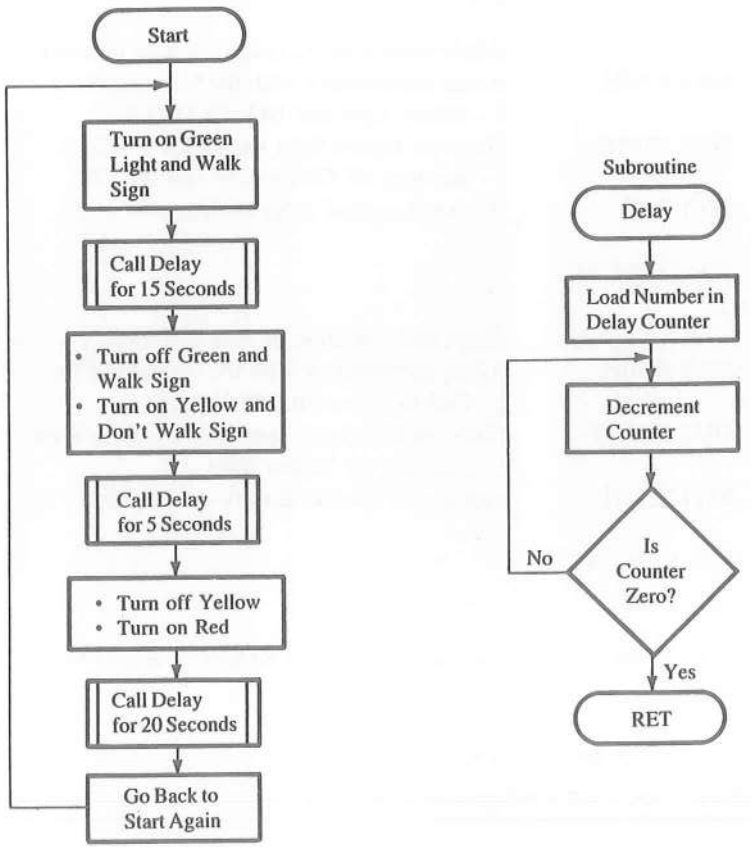


FIGURE 9.13
Flowchart for Traffic Signal Controller

PROGRAM

Memory Address	Code		Mnemonics	Comments
XX00	31			
01	99		LXI SP,XX99	;Initialize stack pointer at location XX99H
02	XX			;High-order address (page) of user memory
03	3E	START:	MVI A,41H	;Load accumulator with the bit pattern for
04	41			; Green light and WALK sign
05	D3		OUT PORT#	;Turn on Green light and WALK sign

```
06      PORT#
07      06          MVI B,0FH       ;Use B as a counter to count 15seconds.
08      0F          ;               B is decremented in the subroutine
09      CD          CALL DELAY      ;Call delay subroutine located at XX50H
0A      50
0B      XX                          ;High-order address (page) of user memory
0C      3E          MVI A,84H       ;Load accumulator with the bit pattern for
0D      84          ;               Yellow light and DON'T WALK
0E      D3          OUT PORT#       ;Turn on Yellow light and DON'T WALK
0F      PORT#       ;               and turn off Green light and WALK
10      06          MVI B,05        ;Set up 5-second delay counter
11      05
12      CD          CALL DELAY
13      50
14      XX                          ;High-order address of user memory
15      3E          MVI A,90H       ;Load accumulator with the bit pattern for
16      90          ;               Red light and DON'T WALK
17      D3          OUT PORT#       ;Turn on Red light, keep DON'T WALK on,
18      PORT#       ;               and turn off Yellow light
19      06          MVI B,14H       ;Set up the counter for 20-second delay
1A      14
1B      CD          CALL DELAY
1C      50
1D      XX
1E      C3          JMP START       ;Go back to location START to repeat the
1F      03          ;               sequence
20      XX
```

;DELAY: This is a 1-second delay subroutine that provides delay
; according to the parameter specified in register B
;*Input:* Number of seconds is specified in register B
;*Output:* None
;*Registers Modified:* Register B

```
XX50    D5      DELAY:  PUSH D          ;Save contents of DE and accumulator
51      F5              PUSH PSW
52      11      SECOND: LXI D,COUNT     ;Load register pair DE with a count for
53      LO              ;               1-second delay
54      HI
55      1B      Loop:   DCX D           ;Decrement register pair DE
56      7A              MOV A,D
57      B3              ORA E           ;OR (D) and (E) to set Zero flag
58      C2              JNZ LOOP        ;Jump to Loop if delay count is not equal to 0
59      55
5A      XX
```

```
5B    05    DCR B        ;End of 1 second delay; decrement the counter
5C    C2    JNZ SECOND   ;Is this the end of time needed? If not, go
5D    52                 ;  back to repeat 1-second delay
5E    XX                 ;High-order memory address of user memory
5F    F1    POP PSW      ;Retrieve contents of saved registers
60    D1    POP D
61    C9    RET          ;Return to main program
```

PROGRAM DESCRIPTION

The stack pointer register is initialized at XX99H so that return addresses can be stored on the stack whenever a CALL instruction is used. As shown in the flowchart this program loads the appropriate bit pattern in the accumulator, sends it to the output port, and calls the delay routine. Register B is loaded in the main program and used in the subroutine to provide appropriate timing.

The DELAY subroutine is similar to the delays discussed in Chapter 8 except it requires the instruction RET at the end of the routine.

This example illustrates the type of subroutine that is called many times from various locations in the main program, as illustrated in Figure 9.14.

In this program, the subroutine is called from the locations XX09, 0A, and 0BH. The return address, XX0C, is stored on the stack, and the stack pointer is decremented by two to location XX97H. At the end of the subroutine, the contents of the top two loca-

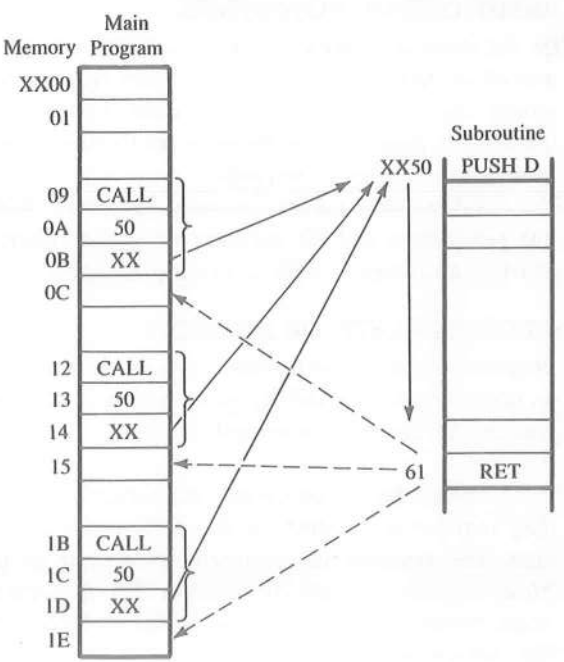


FIGURE 9.14
Multiple-Calling for a Subroutine

tions of the stack (XX0C) are retrieved, the stack pointer register is incremented by two to the original location (XX99H), and the main program is resumed. This sequence is repeated two more times in the main program, as shown in Figure 9.14. This is called a multiple-calling subroutine.

9.2.2 Subroutine Documentation and Parameter Passing

In a large program, subroutines are scattered all over the memory map and are called from many locations. Various information is passed between a calling program and a subroutine, a procedure called **parameter passing.** Therefore, it is important to document a subroutine clearly and carefully. The documentation should include at least the following:

1. Functions of the subroutine
2. Input/Output parameters
3. Registers used or modified
4. List of other subroutines called by this subroutine

The delay subroutine in the traffic-signal controller program shows one example of subroutine documentation.

FUNCTIONS OF THE SUBROUTINE

It is important to state clearly and precisely what the subroutine does. A user should understand the function without going through the instructions.

INPUT/OUTPUT PARAMETERS

In the delay subroutine illustrated in Section 9.2.1, the information concerning the number of seconds is passed from the main program to the subroutine by loading an appropriate count in register B. In this example, a register is used to pass the parameter. The parameters passed to a subroutine are listed as **inputs,** and parameters returned to calling programs are listed as **outputs.**

When many parameters must be passed, R/W memory locations are used to store the parameters, and HL registers are used to point to parameter locations. Similarly, the stack is also used to store and pass parameters.

REGISTERS USED OR MODIFIED

Registers used in a subroutine also may be used by the calling program. Therefore, it is necessary to save the register contents of the calling program on the stack at the beginning of the subroutine and to retrieve the contents before returning from the subroutine.

In the delay subroutine, the contents of registers DE, the accumulator, and the flag register are pushed on the stack because these registers are used in the subroutine. The contents are restored at the end of the routine using the LIFO method. However, the contents of registers that pass parameters should not be saved on the stack because this could cause irrelevant information to be retrieved and passed on to the calling program.

LIST OF SUBROUTINES CALLED

If a subroutine is calling other subroutines, the user should be provided with a list. The user can check what parameters need to be passed to various subroutines and what registers are modified in the process.

9.3 RESTART, CONDITIONAL CALL, AND RETURN INSTRUCTIONS

In addition to the unconditional CALL and RET instructions, the 8085 instruction set includes eight Restart instructions and eight conditional Call and Return instructions.

9.3.1 Restart (RST) Instructions

RST instructions are 1-byte Call instructions that transfer the program execution to a specific location on page 00H. They are executed the same way as Call instructions. When an RST instruction is executed, the 8085 stores the contents of the program counter (the address of the next instruction) on the top of the stack and transfers the program to the Restart location. These instructions are generally used in conjunction with the interrupt process discussed in Chapter 12. These instructions are listed here to emphasize that they are Call instructions and not necessarily always associated with the interrupts. The list of eight RST instructions is as follows:

RST 0	Call 0000H	RST 4	Call 0020H
RST 1	Call 0008H	RST 5	Call 0028H
RST 2	Call 0010H	RST 6	Call 0030H
RST 3	Call 0018H	RST 7	Call 0038H

9.3.2 Conditional Call and Return Instructions

The conditional Call and Return instructions are based on four data conditions (flags): Carry, Zero, Sign, and Parity. The conditions are tested by checking the respective flags. In case of a conditional Call instruction, the program is transferred to the subroutine if the condition is met; otherwise, the main program is continued. In case of a conditional Return instruction, the sequence returns to the main program if the condition is met; otherwise, the sequence in the subroutine is continued. If the Call instruction in the main program is conditional, the Return instruction in the subroutine can be conditional or unconditional. The conditional Call and Return instructions are listed for reference.

CONDITIONAL CALL

CC	Call subroutine if Carry flag is set (CY = 1)
CNC	Call subroutine if Carry flag is reset (CY = 0)
CZ	Call subroutine if Zero flag is set (Z = 1)
CNZ	Call subroutine if Zero flag is reset (Z = 0)
CM	Call subroutine if Sign flag is set (S = 1, negative number)
CP	Call subroutine if Sign flag is reset (S = 0, positive number)

CPE	Call subroutine if Parity flag is set (P = 1, even parity)
CPO	Call subroutine if Parity flag is reset (P = 0, odd parity)

CONDITIONAL RETURN

RC	Return if Carry flag is set (CY = 1)
RNC	Return if Carry flag is reset (CY = 0)
RZ	Return if Zero flag is set (Z = 1)
RNZ	Return if Zero flag is reset (Z = 0)
RM	Return if Sign flag is set (S = 1, negative number)
RP	Return if Sign flag is reset (S = 0, positive number)
RPE	Return if Parity flag is set (P = 1, even parity)
RPO	Return if Parity flag is reset (P = 0, odd parity)

9.4 ADVANCED SUBROUTINE CONCEPTS

In Section 9.2, one type of subroutine (multiple-calling of a subroutine by a main program) was illustrated. Other types of subroutine techniques, such as nesting and multiple-ending, are briefly illustrated below.

9.4.1 Nesting

The programming technique of a subroutine calling another subroutine is called **nesting.** This process is limited only by the number of available stack locations. When a subroutine calls another subroutine, all return addresses are stored on the stack. Nesting is illustrated in Figure 9.15.

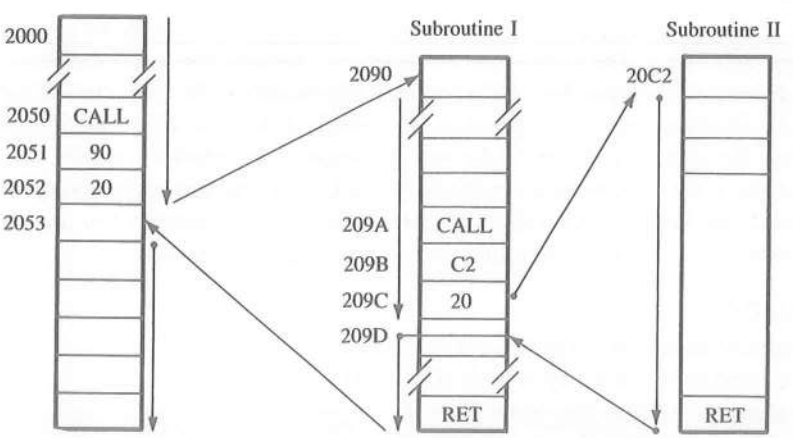
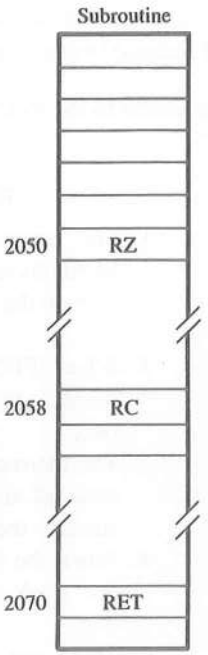


FIGURE 9.15
Nesting of Subroutines

FIGURE 9.16
Multiple-Ending Subroutine



The main program in Figure 9.15 calls the subroutine from location 2050H. The address of the next instruction, 2053H, is placed on the stack, and the program is transferred to the subroutine at 2090H. Subroutine I calls Subroutine II from location 209AH. The address 209DH is placed on the stack, and the program is transferred to Subroutine II. The sequence of execution returns to the main program, as shown in Figure 9.15.

9.4.2 Multiple-Ending Subroutines

Figure 9.16 illustrates three possible endings to one CALL instruction. The subroutine has two conditional returns (RZ—Return on Zero, and RC—Return on Carry) and one unconditional return (RET). If the Zero flag (Z) is set, the subroutine returns from location 2050H. If the Carry flag (CY) is set, it returns from location 2058H. If neither the Z nor the CY flag is set, it returns from location 2070H. This technique is illustrated in Chapter 10, Section 10.4.2.

SUMMARY

To implement a subroutine, the following steps are necessary:

1. The stack pointer register must be initialized, preferably at the highest memory location of the R/W memory.

2. The CALL (or conditional Call) instruction should be used in the main program accompanied by the RET (or conditional Return) instruction in the subroutine.

The instructions CALL and RET are similar to the instructions PUSH and POP. The similarities and differences are as follows:

CALL and RET	PUSH and POP
1. When CALL is executed, the microprocessor automatically stores the 16-bit address of the instruction next to CALL on the stack.	1. The programmer uses the instruction PUSH to save the contents of a register pair on the stack.
2. When CALL is executed, the stack pointer register is decremented by two.	2. When PUSH is executed, the stack pointer register is decremented by two.
3. The instruction RET transfers the contents of the top two locations of the stack to the program counter.	3. The instruction POP transfers the contents of the top two locations of the stack to the specified register pair.
4. When the instruction RET is executed, the stack pointer is incremented by two.	4. When the instruction POP is executed, the stack pointer is incremented by two.
5. In addition to the unconditional CALL and RET instructions, there are eight conditional CALL and RETURN instructions.	5. There are no conditional PUSH and POP instructions.

QUESTIONS AND PROGRAMMING ASSIGNMENTS

1. Check the appropriate answer in the following statements.
   - **a.** A stack is
     - (1) an 8-bit register in the microprocessor.
     - (2) a 16-bit register in the microprocessor.
     - (3) a set of memory locations in R/WM reserved for storing information temporarily during the execution of a program.
     - (4) a 16-bit memory address stored in the program counter.
   - **b.** A stack pointer is
     - (1) a 16-bit register in the microprocessor that indicates the beginning of the stack memory.
     - (2) a register that decodes and executes 16-bit arithmetic expressions.
     - (3) the first memory location where a subroutine address is stored.
     - (4) a register in which flag bits are stored.

c. When a subroutine is called, the address of the instruction following the CALL instruction is stored in/on the
   (1) stack pointer.
   (2) accumulator.
   (3) program counter.
   (4) stack.

d. When the RET instruction at the end of a subroutine is executed,
   (1) the information where the stack is initialized is transferred to the stack pointer.
   (2) the memory address of the RET instruction is transferred to the program counter.
   (3) two data bytes stored in the top two locations of the stack are transferred to the program counter.
   (4) two data bytes stored in the top two locations of the stack are transferred to the stack pointer.

e. Whenever the POP H instruction is executed,
   (1) data bytes in the HL pair are stored on the stack.
   (2) two data bytes at the top of the stack are transferred to the HL register pair.
   (3) two data bytes at the top of the stack are transferred to the program counter.
   (4) two data bytes from the HL register that were previously stored on the stack are transferred back to the HL register.

f. The instruction RST 7 is a:
   (1) restart instruction that begins the execution of a program.
   (2) one-byte call to the memory address 0038H.
   (3) one-byte call to the memory address 0007H.
   (4) hardware interrupt.

2. Read the following program and answer the questions given below.

Line No.	Mnemonics
1	LXI SP,0400H
2	LXI B,2055H
3	LXI H,22FFH
4	LXI D,2090H
5	PUSH H
6	PUSH B
7	MOV A,L
↓	↓
20	POP H

a. What is stored in the stack pointer register after the execution of line 1?

b. What is the memory location of the stack where the first data byte will be stored?
c. What is stored in memory location 03FEH when line 5 (PUSH H) is executed?
d. After the execution of line 6 (PUSH B), what is the address in the stack pointer register, and what is stored in stack memory location 03FDH?
e. Specify the contents of register pair HL after the execution of line 20 (POP H).

3. The following program has a delay subroutine located at location 2060H. Read the program and answer the questions given at the end of the program.

Memory Locations	Mnemonics	
2000	LXI SP,20CDH	;Main Program
2003	LXI H,00008H	
2006	MVI B,0FH	
2008	CALL 2060H	
200B	OUT 01H	
	DCR B	
	↓	
	CONTD	
2060	PUSH H	:Delay Subroutine
2061	PUSH B	
	MVI B,05H	
	LXI H,COUNT	
	↓	
	POP B	
	POP H	
	RET	

a. When the execution of the CALL instruction located at 2008H–200AH is completed, list the contents stored at 20CCH and 20CBH, the contents of the program counter, and the contents of the stack pointer register.
b. List the stack locations and their contents after the execution of the instructions PUSH H and PUSH B in the subroutine.
c. List the contents of the stack pointer register after the execution of the instruction PUSH B located at 2061H.
d. List the contents of the stack pointer register after the execution of the instruction RET in the subroutine.

4. Explain the functions of the following routines:

a.
```
LXI SP,209FH
MVI C,00H
PUSH B
POP PSW
RET
```

b.
```
LXI SP,STACK
PUSH B
PUSH D
POP B
POP D
RET
```

5. Read the following program and answer the questions.

```
2000 LXI SP,2100H     DELAY: 2064 PUSH H
2003 LXI B,0000H             2065 PUSH B
2006 PUSH B                  2066 LXI B,80FFH
2007 POP PSW          LOOP:  2069 DCX B
2008 LXI H,200BH             206A MOV A,B
200B CALL 2064H              206B ORA C
200E OUT 01H                 206C JNZ LOOP
2010 HLT                     206F POP B
                             2070 RET
```

   a. What is the status of the flags and the contents of the accumulator after the execution of the POP instruction located at 2007H?
   b. Specify the stack locations and their contents after the execution of the CALL instruction (not the Call subroutine).
   c. What are the contents of the stack pointer register and the program counter after the execution of the CALL instruction?
   d. Specify the memory location where the program returns after the subroutine.
   e. What is the ultimate fate of this program?

6. Write a program to add the two Hex numbers 7A and 46 and to store the sum at memory location XX98H and the flag status at location XX97H.
7. In Assignment 6, display the sum and the flag status at two different output ports.
8. Write a program to meet the following specifications:
   a. Initialize the stack pointer register at XX99H.
   b. Clear the memory locations starting from XX90H to XX9FH.
   c. Load register pairs B, D, and H with data 0237H, 1242H, and 4087H, respectively.
   d. Push the contents of the register pairs B, D, and H on the stack.
   e. Execute the program and verify the memory locations from XX90H to XX9FH.
9. Write a program to clear the initial flags. Load data byte FFH into the accumulator and add 01H to the byte FFH by using the instruction ADI. Mask all the flags except the CY flag and display the CY flag at PORT0 (or store the results on the stack). Repeat the program by replacing the ADI instruction with the INR instruction and the byte 01H with the NOP instruction. Display the flag at PORT1 (or store the results on the stack). Explain the results.
10. Write a 20 ms time delay subroutine using register pair BC. Clear the Z flag without affecting any other flags in the flag register and return to the main program.
11. Write a program to control a railway crossing signal that has two alternately flashing red lights, with a 1-second delay on time for each light.
12. Write a program to simulate a flashing yellow light with 750 ms on time. Use bit D_7 to control the light. (*Hint:* To simulate flashing, the light must be turned off.)

10

Code Conversion, BCD Arithmetic, and 16-Bit Data Operations

In microcomputer applications, various number systems and codes are used to input data or to display results. The ASCII (American Standard Code for Information Interchange) keyboard is a commonly used input device for disk-based microcomputer systems. Similarly, alphanumeric characters (letters and numbers) are displayed on a CRT (cathode ray tube) terminal using the ASCII code. However, inside the microprocessor, data processing is usually performed in binary. In some instances, arithmetic operations are performed in BCD numbers. Therefore, data must be converted from one code to another code. The programming techniques used for code conversion fall into four general categories:

1. Conversion based on the position of a digit in a number (BCD to binary and vice versa).
2. Conversion based on hardware consideration (binary to seven-segment code using table look-up procedure).
3. Conversion based on sequential order of digits (binary to ASCII and vice versa).
4. Decimal adjustment in BCD arithmetic operations. (This is an adjustment rather than a code conversion.)

This chapter discusses these techniques with various examples written as subroutines. The subroutines are written to demonstrate industrial practices in writing software, and can be verified on single-board microcomputers. In addition, instructions related to 16-bit data operations are introduced and illustrated.

OBJECTIVES

Write programs and subroutines to

- ☐ Convert a packed BCD number (0–99) into its binary equivalent.
- ☐ Convert a binary digit (0 to F) into its ASCII Hex code and vice versa.

- ☐ Select an appropriate seven-segment code for a given binary number using the table look-up technique.
- ☐ Convert a binary digit (0 to F) into its ASCII Hex code and vice versa.
- ☐ Decimal-adjust 8-bit BCD addition and subtraction.
- ☐ Perform such arithmetic operations as multiplication and subtraction using 16-bit data related instructions.
- ☐ Demonstrate uses of instructions such as DAD, PCHL, XTHL, and XCHG.

10.1 BCD-TO-BINARY CONVERSION

In most microprocessor-based products, data are entered and displayed in decimal numbers. For example, in an instrumentation laboratory, readings such as voltage and current are maintained in decimal numbers, and data are entered through a decimal keyboard. The system-monitor program of the instrument converts each key into an equivalent 4-bit binary number and stores two BCD numbers in an 8-bit register or a memory location. These numbers are called **packed BCD.** Even if data are entered in decimal digits, it is inefficient to process data in BCD numbers because, in each 4-bit combination, digits A through F are unused. Therefore, BCD numbers are generally converted into binary numbers for data processing.

The conversion of a BCD number into its binary equivalent employs the principle of *positional weighting* in a given number.

For example: $72_{10} = 7 \times 10 + 2$.

The digit 7 represents 70, based on its second position from the right. Therefore, converting 72_{BCD} into its binary equivalent requires multiplying the second digit by 10 and adding the first digit.

Converting a 2-digit BCD number into its binary equivalent requires the following steps:

1. Separate an 8-bit packed BCD number into two 4-bit unpacked BCD digits: BCD_1 and BCD_2.
2. Convert each digit into its binary value according to its position.
3. Add both binary numbers to obtain the binary equivalent of the BCD number.

Example 10.1 Convert 72_{BCD} into its binary equivalent.

Solution

$$72_{10} = 0111\ 0010_{BCD}$$

Step 1: 0111 0010 → 0000 0010 Unpacked BCD_1
→ 0000 0111 Unpacked BCD_2
Step 2: Multiply BCD_2 by 10 (7×10)
Step 3: Add BCD_1 to the answer in Step 2

The multiplication of BCD_2 by 10 can be performed by various methods. One method is multiplication with repeated addition: add 10 seven times. This technique is illustrated in the next program.

10.1.1 Illustrative Program: 2-Digit BCD-to-Binary Conversion

PROBLEM STATEMENT

A BCD number between 0 and 99 is stored in an R/W memory location called the **Input Buffer (INBUF).** Write a main program and a conversion subroutine (BCDBIN) to convert the BCD number into its equivalent binary number. Store the result in a memory location defined as the **Output Buffer (OUTBUF).**

PROGRAM

```
START:  LXI SP,STACK      ;Initialize stack pointer
        LXI H,INBUF       ;Point HL index to the Input Buffer memory location where BCD
                          ;   number is stored
        LXI B,OUTBUF      ;Point BC index to the Output Buffer memory where binary
                          ;   number will be stored
        MOV A,M           ;Get BCD number
        CALL BCDBIN       ;Call BCD to binary conversion routine
        STAX B            ;Store binary number in the Output Buffer
        HLT               ;End of program

BCDBIN: ;Function: This subroutine converts a BCD number into its binary equivalent
        ;Input: A 2-digit packed BCD number in the accumulator
        ;Output: A binary number in the accumulator
        ;No other register contents are destroyed
        PUSH B            ;Save BC registers
        PUSH D            ;Save DE registers
        MOV B,A           ;Save BCD number
        ANI 0FH           ;Mask most significant four bits
        MOV C,A           ;Save unpacked BCD1 in C
        MOV A,B           ;Get BCD again
        ANI F0H           ;Mask least significant four bits
        JZ BCD1           ;If BCD2 = 0, the result is only BCD2
        RRC               ;Convert most significant four
        RRC               ;   bits into unpacked BCD2
        RRC
        RRC
        MOV D,A           ;Save BCD2 in D
        XRA A             ;Clear accumulator
```

Example: Assume BCD number is 72:

A	0111	$0010 \rightarrow 72_{10}$
B	0111	$0010 \rightarrow 72_{10}$
A	0000	$0010 \rightarrow 02_{10}$
C	0000	$0010 \rightarrow 02_{10}$
A	0111	$0010 \rightarrow 72_{10}$
A	0111	$0000 \rightarrow 70M_{10}$
A	0000	$0111 \rightarrow 07_{10}$
D	0000	$0111 \rightarrow 07_{10}$

```
        MVI E,0AH   ;Set E as multiplier of 10
SUM:    ADD E       ;Add 10 until (D) = 0
        DCR D       ;Reduce BCD2 by one
        JNZ SUM     ;Is multiplication complete?
                    ;If not, go back and add again
BCD1:   ADD C       ;Add BCD1
        POP D       ;Retrieve previous contents
        POP B
        RET
```

```
E     0000    1010 → 0AH
Add E as many times as (D)

After adding E seven times A
contains:  0100 0110
C         +0000 0010
A          0100 1000 → 48H
```

PROGRAM DESCRIPTION

1. In writing assembly language programs, the use of labels is a common practice. Rather than writing a specific memory location or a port number, a programmer uses such labels as INBUF (Input Buffer) and OUTBUF (Output Buffer). Using labels gives flexibility and ease of documentation.
2. The main program initializes the stack pointer and two memory indexes. It brings the BCD number into the accumulator and passes that parameter to the subroutine.
3. After returning from the subroutine, the main program stores the binary equivalent in the Output Buffer memory.
4. The subroutine saves the contents of the BC and DE registers because these registers are used in the subroutine. Even if this particular main program does not use the DE registers, the subroutine may be called by some other program in which the DE registers are being used. Therefore, it is a good practice to save the registers that are used in the subroutine, unless parameters are passed to the subroutine. The accumulator contents are not saved because that information is passed on to the subroutine.
5. The conversion from BCD to binary is illustrated in the subroutine with the example of 72_{BCD} converted to binary.

The illustrated multiplication routine is easy to understand; however, it is rather long and inefficient. Another method is to multiply BCD_2 by shifting, as illustrated in Assignments 3 and 4 at the end of this chapter.

PROGRAM EXECUTION

To execute the program on a single-board computer, complete the following steps:

1. Assign memory addresses to the instructions in the main program and in the subroutine. Both can be assigned consecutive memory addresses.
2. Define STACK: the stack location with a 16-bit address in the R/W memory (such as 2099H).
3. Define INBUF (Input Buffer) and OUTBUF (Output Buffer): two memory locations in the R/W memory (e.g., 2050H and 2060H).
4. Enter a BCD byte in the Input Buffer (e.g., 2050H).
5. Enter and execute the program.

6. Check the contents of the Output Buffer memory location (2060H) and verify your answer.

See Assignments 1 through 4 at the end of this chapter.

10.2 BINARY-TO-BCD CONVERSION

In most microprocessor-based products, numbers are displayed in decimal. However, if data processing inside the microprocessor is performed in binary, it is necessary to convert the binary results into their equivalent BCD numbers just before they are displayed. Results are quite often stored in R/W memory locations called the **Output Buffer.**

The conversion of binary to BCD is performed by dividing the number by the powers of ten; the division is performed by the subtraction method.

For example, assume the binary number is

$$1\ 1\ 1\ 1\ \ 1\ 1\ 1\ 1_2\ (\text{FFH}) = 255_{10}$$

To represent this number in BCD requires twelve bits or three BCD digits, labeled here as BCD_3 (MSB), BCD_2, and BCD_1 (LSB),

$$= 0\ 0\ 1\ 0\ \ 0\ 1\ 0\ 1\ \ 0\ 1\ 0\ 1$$
$$BCD_3 \quad BCD_2 \quad BCD_1$$

The conversion can be performed as follows:

Step 1: If the number is less than 100, go to Step 2; otherwise, divide by 100 or subtract 100 repeatedly until the remainder is less than 100. The quotient is the most significant BCD digit, BCD_3.

Step 2: If the number is less than 10, go to Step 3; otherwise divide by 10 repeatedly until the remainder is less than 10. The quotient is BCD_2.

Step 3: The remainder from Step 2 is BCD_1.

Example			Quotient
255			
−100	=	155	1
−100	=	55	1
		BCD_3 =	2
55			
−10	=	45	1
−10	=	35	1
−10	=	25	1
−10	=	15	1
−10	=	05	1
		BCD_2 =	5
		BCD_1 =	5

These steps can be converted into a program as illustrated next.

10.2.1 Illustrative Program: Binary-to-Unpacked-BCD Conversion

PROBLEM STATEMENT

A binary number is stored in memory location BINBYT. Convert the number into BCD, and store each BCD as two unpacked BCD digits in the Output Buffer. To perform this task, write a main program and two subroutines: one to supply the powers of ten, and the other to perform the conversion.

PROGRAM

This program converts an 8-bit binary number into a BCD number; thus it requires 12 bits to represent three BCD digits. The result is stored as three unpacked BCD digits in three Output-Buffer memory locations.

```
START:  LXI SP,STACK    ;Initialize stack pointer
        LXI H,BINBYT    ;Point HL index where binary number is stored
        MOV A,M         ;Transfer byte
        CALL PWRTEN     ;Call subroutine to load powers of 10
        HLT

PWRTEN: ;this subroutine loads the powers of 10 in register B and calls the binary-to-BCD
        ;  conversion routine
        ;Input: Binary number in the accumulator
        ;Output: Powers of ten and stores BCD1 in the first Output-Buffer memory
        ;Calls BINBCD routine and modifies register B

        LXI H,OUTBUF    ;Point HL index to Output-Buffer memory
        MVI B,64H       ;Load 100 in register B
        CALL BINBCD     ;Call conversion
        MVI B,0AH       ;Load 10 in register B
        CALL BINBCD
        MOV M,A         ;Store BCD1
        RET

BINBCD: ;This subroutine converts a binary number into BCD and stores BCD2 and BCD3 in the
        ;  Output Buffer.
        ;Input: Binary number in accumulator and powers of 10 in B
        ;Output: BCD2 and BCD3 in Output Buffer
        ;Modifies accumulator contents

        MVI M,FFH       ;Load buffer with (0 – 1)
NXTBUF: INR M           ;Clear buffer and increment for each subtraction
        SUB B           ;Subtract power of 10 from binary number
        JNC NXTBUF      ;Is number > power of 10? If yes, add 1 to buffer memory
        ADD B           ;If no, add power of 10 to get back remainder
```

```
        INX H           ;Go to next buffer location
        RET
```

PROGRAM DESCRIPTION

This program illustrates the concepts of the **nested subroutine** and the **multiple-call subroutine.** The main program calls the PWRTEN subroutine; in turn, the PWRTEN calls the BINBCD subroutine twice.

1. The main program transfers the byte to be converted to the accumulator and calls the PWRTEN subroutine.
2. The subroutine PWRTEN supplies the powers of ten by loading register B and the address of the first Output-Buffer memory location, and calls conversion routine BINBCD.
3. In the BINBCD conversion routine, the Output-Buffer memory is used as a register. It is incremented for each subtraction loop. This step also can be achieved by using a register in the microprocessor. The BINBCD subroutine is called twice, once after loading register B with 64H (100_{10}), and again after loading register B with 0AH (10_{10}).
4. During the first call of BINBCD, the subroutine clears the Output Buffer, stores BCD_3, and points the HL registers to the next Output-Buffer location. The instruction ADD B is necessary to restore the remainder because one extra subtraction is performed to check the borrow.
5. During the second call of BINBCD, the subroutine again clears the output buffer, stores BCD_2, and points to the next buffer location. BCD_3 is already in the accumulator after the ADD instruction, which is stored in the third Output-Buffer memory by the instruction MOV M,A in the PWRTEN subroutine.

This is an efficient subroutine; it combines the functions of storing the answer and finding a quotient. However, two subroutines are required, and the second subroutine is called twice for a conversion.

See Assignments 5 through 8 at the end of this chapter.

10.3 BCD-TO-SEVEN-SEGMENT-LED CODE CONVERSION

When a BCD number is to be displayed by a seven-segment LED, it is necessary to convert the BCD number to its seven-segment code. The code is determined by hardware considerations such as common-cathode or common-anode LED; the code has no direct relationship to binary numbers. Therefore, to display a BCD digit at a seven-segment LED, the **table look-up technique** is used.

In the table look-up technique, the codes of the digits to be displayed are stored sequentially in memory. The conversion program locates the code of a digit based on its

magnitude and transfers the code to the MPU to send out to a display port. The table look-up technique is illustrated in the next program.

10.3.1 Illustrative Program: BCD-to-Common-Cathode-LED Code Conversion

PROBLEM STATEMENT

A set of three packed BCD numbers (six digits) representing time and temperature are stored in memory locations starting at XX50H. The seven-segment codes of the digits 0 to 9 for a common-cathode LED are stored in memory locations starting at XX70H, and the Output-Buffer memory is reserved at XX90H.

Write a main program and two subroutines, called UNPAK and LEDCOD, to unpack the BCD numbers and select an appropriate seven-segment code for each digit. The codes should be stored in the Output-Buffer memory.

PROGRAM

```
        LXI SP,STACK      ;Initialize stack pointer
        LXI H,XX50H       ;Point HL where BCD digits are stored
        MVI D,03H         ;Number of digits to be converted is placed in D
        CALL UNPAK        ;Call subroutine to unpack BCD numbers
        HLT               ;End of conversion
UNPAK:  ;This subroutine unpacks the BCD number in two single digits
        ;Input: Starting memory address of the packed BCD numbers in HL registers
        ;       Number of BCDs to be converted in register D
        ;Output: Unpacked BCD into accumulator and output
        ;       Buffer address in BC
        ;Calls subroutine LEDCOD
        LXI B,BUFFER      ;Point BC index to the buffer memory
NXTBCD: MOV A,M           ;Get packed BCD number
        ANI F0H           ;Masked BCD1
        RRC               ;Rotate four times to place BCD2 as unpacked single-digit BCD
        RRC
        RRC
        RRC
        CALL LEDCOD       ;Find seven-segment code
        INX B             ;Point to next buffer location
        MOV A,M           ;Get BCD number again
        ANI 0FH           ;Separate BCD1
        CALL LEDCOD
        INX B
        INX H             ;Point to next BCD
        DCR D             ;One conversion complete, reduce BCD count
        JNZ NXTBCD        ;If all BCDs are not yet converted, go back to convert next BCD
        RET
```

```
LEDCOD:  ;This subroutine converts an unpacked BCD into its seven-segment-LED code
         ;Input: An unpacked BCD in accumulator
         ;Memory address of the buffer in BC register
         ;Output: Stores seven-segment code in the output buffer
         PUSH H          ;Save HL contents of the caller
         LXI H,CODE      ;Point index to beginning of seven-segment code
         ADD L           ;Add BCD digit to starting address of the code
         MOV L,A         ;Point HL to appropriate code
         MOV A,M         ;Get seven-segment code
         STAX B          ;Store code in buffer
         POP H
         RET
CODE:    3F              :Digit 0: Common-cathode codes
         06              ;Digit 1
         5B              ;Digit 2
         4F              ;Digit 3
         66              ;Digit 4
         6D              ;Digit 5
         7D              ;Digit 6
         07              ;Digit 7
         7F              ;Digit 8
         6F              ;Digit 9
         00              ;Invalid Digit
```

PROGRAM DESCRIPTION/OUTPUT

1. The main program initializes the stack pointer, the HL register as a pointer for BCD digits, and the counter for the number of digits; then it calls the UNPAK subroutine.
2. The UNPAK subroutine transfers a BCD number into the accumulator and unpacks it into two BCD digits by using the instructions ANI and RRC. This subroutine also supplies the address of the buffer memory to the next subroutine, LEDCOD. The subroutine is repeated until counter D becomes zero.
3. The LEDCOD subroutine saves the memory address of the BCD number and points the HL register to the beginning address of the code.
4. The instruction ADD L adds the BCD digit in the accumulator to the starting address of the code. After storing the sum in register L, the HL register points to the seven-segment code of that BCD digit.
5. The code is transferred to the accumulator and stored in the buffer.

This illustrative program uses the technique of the nested subroutine (one subroutine calling another). Parameters are passed from one subroutine to another; therefore, you should be careful in using Push instructions to store register contents on the stack. In

addition, the LEDCOD subroutine does not account for a situation if by adding the register L a carry is generated. (See Assignment 12.)

See Assignments 9–12 at the end of this chapter.

10.4 BINARY-TO-ASCII AND ASCII-TO-BINARY CODE CONVERSION

The American Standard Code for Information Interchange (known as ASCII) is used commonly in data communication. It is a seven-bit code, and its 128 (2^7) combinations are assigned different alphanumeric characters (see Appendix E). For example, the hexadecimal numbers 30H to 39H represent 0 to 9 ASCII decimal numbers, and 41H to 5AH represent capital letters A through Z; in this code, bit D_7 is zero. In serial data communication, bit D_7 can be used for parity checking (see Chapter 16, Serial I/O and Data Communication).

The ASCII keyboard is a standard input device for entering programs in a microcomputer. When an ASCII character is entered, the microprocessor receives the binary equivalent of the ASCII Hex number. For example, when the ASCII key for digit 9 is pressed, the microprocessor receives the binary equivalent of 39H, which must be converted to the binary 1001 for arithmetic operations. Similarly, to display digit 9 at the terminal, the microprocessor must send out the ASCII Hex code (39H). These conversions are done through software, as in the following illustrative program.

10.4.1 Illustrative Program: Binary-to-ASCII Hex Code Conversion

PROBLEM STATEMENT

An 8-bit binary number (e.g., 9FH) is stored in memory location XX50H.

1. Write a program to
   a. Transfer the byte to the accumulator.
   b. Separate the two nibbles (as 09 and 0F).
   c. Call the subroutine to convert each nibble into ASCII Hex code.
   d. Store the codes in memory locations XX60H and XX61H.
2. Write a subroutine to convert a binary digit (0 to F) into ASCII Hex code.

MAIN PROGRAM

```
LXI SP,STACK    ;Initialize stack pointer
LXI H,XX50H     ;Point index where binary number is stored
LXI D,XX60H     ;Point index where ASCII code is to be stored
MOV A,M         ;Transfer byte
MOV B,A         ;Save byte
RRC             ;Shift high-order nibble to the position of low-order
RRC             ;  nibble
```

```
        RRC
        RRC
        CALL ASCII      ;Call conversion routine
        STAX D          ;Store first ASCII Hex in XX60H
        INX D           ;Point to next memory location, get ready to store
                        ;  next byte
        MOV A,B         ;Get number again for second digit
        CALL ASCII
        STAX D
        HLT

ASCII:  ;This subroutine converts a binary digit between 0 and F to ASCII Hex
        ;  code
        ;Input: Single binary number 0 to F in the accumulator
        ;Output: ASCII Hex code in the accumulator
        ANI 0FH         ;Mask high-order nibble
        CPI 0AH         ;Is digit less than 10(10)?
        JC CODE         ;If digit is less than 10(10), go to CODE to add 30H
        ADI 07H         ;Add 7H to obtain code for digits from A to F
CODE:   ADI 30H         ;Add base number 30H
        RET
```

PROGRAM DESCRIPTION

1. The main program transfers the binary data byte from the memory location to the accumulator.
2. It shifts the high-order nibble into the low-order nibble, calls the conversion subroutine, and stores the converted value in the memory.
3. It retrieves the byte again and repeats the conversion process for the low-order nibble.

In this program, the masking instruction ANI is used once in the subroutine rather than twice in the main program as illustrated in the program for BCD-to-Common-Cathode-LED Code Conversion (Section 10.3.1).

10.4.2 Illustrative Program: ASCII Hex-to-Binary Conversion

PROBLEM STATEMENT

Write a subroutine to convert an ASCII Hex number into its binary equivalent. A calling program places the ASCII number in the accumulator, and the subroutine should pass the conversion back to the accumulator.

SUBROUTINE

```
ASCBIN: ;This subroutine converts an ASCII Hex number into its binary
        ;  equivalent
        ;Input: ASCII Hex number in the accumulator
```

```
;Output: Binary equivalent in the accumulator

SUI 30H     ;Subtract 0 bias from the number
CPI 0AH     ;Check whether number is between 0 and 9
RC          ;If yes, return to main program
SUI 07H     ;If not, subtract 7 to find number between A and F
RET
```

PROGRAM DESCRIPTION

This subroutine subtracts the ASCII weighting digits from the number. This process is exactly opposite to that of the Illustrative Program that converted binary into ASCII Hex (Section 10.4.1). However, this program uses two return instructions, an illustration of the multiple-ending subroutine.

See Assignments 13 and 14 at the end of this chapter.

10.5 BCD ADDITION

In some applications, input/output data are presented in decimal numbers, and the speed of data processing is unimportant. In such applications, it may be convenient to perform arithmetic operations directly in BCD numbers. However, the addition of two BCD numbers may not represent an appropriate BCD value. For example, the addition of 34_{BCD} and 26_{BCD} results in 5AH, as shown below:

$$\begin{array}{rl} 34_{10} = & 0\ 0\ 1\ 1\ \ 0\ 1\ 0\ 0_{BCD} \\ +\ 26_{10} = & 0\ 0\ 1\ 0\ \ 0\ 1\ 1\ 0_{BCD} \\ \hline 60_{10} = & 0\ 1\ 0\ 1\ \ 1\ 0\ 1\ 0 \rightarrow 5AH \end{array}$$

The microprocessor cannot recognize BCD numbers; it adds any two numbers in binary. In BCD addition, any number larger than 9 (from A to F) is invalid and needs to be adjusted by adding 6 in binary. For example, after 9, the next BCD number is 10; however, in Hex it is A. The Hex number A can be adjusted as a BCD number by adding 6 in binary. The BCD adjustment in an 8-bit binary register can be shown as follows:

$$\begin{array}{rl} A = & 0\ 0\ 0\ 0\ \ 1\ 0\ 1\ 0 \\ +\ 6 = & \underline{0\ 0\ 0\ 0\ \ 0\ 1\ 1\ 0} \\ & 0\ 0\ 0\ 1\ \ 0\ 0\ 0\ 0 \rightarrow 10_{BCD} \end{array}$$

Any BCD sum can be adjusted to proper BCD value by adding 6 when the sum exceeds 9. In case of packed BCD, both BCD_1 and BCD_2 need to be adjusted; if a carry is generated by adding 6 to BCD_1, the carry should be added to BCD_2, as shown in the following example.

Example 10.2

Add two packed BCD numbers: 77 and 48.

Solution

Addition:

$$\begin{array}{rr} 77 = 0\ 1\ 1\ 1 & 0\ 1\ 1\ 1 \\ +\,48 = 0\ 1\ 0\ 0 & 1\ 0\ 0\ 0 \\ \hline 125 = 1\ 0\ 1\ 1 & 1\ 1\ 1\ 1 \end{array}$$

The value of the least significant four bits is larger than 9. Add 6.

$$\begin{array}{rr} & +\,0\ 1\ 1\ 0 \\ \hline \text{CY}\,\boxed{1} & 0\ 1\ 0\ 1 \end{array}$$

The value of the most significant four bits is larger than 9. Add 6 and the carry from the previous adjustment.

$$\begin{array}{rrr} & +\,0\ 1\ 1\ 0 & \\ \hline \text{CY}\,\boxed{1} & 0\ 0\ 1\ 0 & 0\ 1\ 0\ 1 \end{array}$$

In this example, the carry is generated after the adjustment of the least significant four bits for the BCD digit and is again added to the adjustment of the most significant four bits.

A special instruction called DAA (Decimal Adjust Accumulator) performs the function of adjusting a BCD sum in the 8085 instruction set. This instruction uses the Auxiliary Carry flip-flop (AC) to sense that the value of the least four bits is larger than 9 and adjusts the bits to the BCD value. Similarly, it uses the Carry flag (CY) to adjust the most significant four bits. However, the AC flag is used internally by the microprocessor; this flag is not available to the programmer through any Jump instruction.

INSTRUCTION

DAA: Decimal Adjust Accumulator

- ☐ This is a 1-byte instruction
- ☐ It adjusts an 8-bit number in the accumulator to form two BCD numbers by using the process described above
- ☐ It uses the AC and the CY flags to perform the adjustment
- ☐ All flags are affected

It must be emphasized that instruction DAA

- ☐ adjusts a BCD sum.
- ☐ does not convert a binary number into BCD numbers.
- ☐ works only with addition when BCD numbers are used; does not work with subtraction.

10.5.1 Illustrative Program: Addition of Unsigned BCD Numbers

PROBLEM STATEMENT

A set of ten packed BCD numbers is stored in the memory location starting at XX50H.

1. Write a program with a subroutine to add these numbers in BCD. If a carry is generated, save it in register B, and adjust it for BCD. The final sum will be less than 9999_{BCD}.

2. Write a second subroutine to unpack the BCD sum stored in registers A and B, and store them in the output-buffer memory starting at XX60H. The most significant digit (BCD_4) should be stored at XX60H and the least significant digit (BCD_1) at XX63H.

PROGRAM

```
START:  LXI SP,STACK    ;Initialize stack pointer
        LXI H,XX50H     ;Point index to XX50H
        MVI C,COUNT     ;Load register C with the count of BCD numbers to be added
        XRA A           ;Clear accumulator
        MOV B,A         ;Clear register B to save carry
NXTBCD: CALL BCDADD     ;Call subroutine to add BCD numbers
        INX H           ;Point to next memory location
        DCR C           ;One addition of BCD number is complete, decrement the counter
        JNZ NXTBCD      ;If all numbers are added go to next step, otherwise go back
        LXI H,XX63H     ;Point index to store BCD1 first
        CALL UNPAK      ;Unpack the BCD stored in the accumulator
        MOV A,B         ;Get ready to store high-order BCD—BCD3 and BCD4
        CALL UNPAK      ;Unpack and store BCD3 and BCD4 at XX61H and XX60
        HLT

BCDADD: ;This subroutine adds the BCD number from the memory to the accumulator and decimal-
        ;  adjusts it. If the sum is larger than eight bits, it saves the carry and decimal-adjusts the
        ;  carry sum
        ;Input: The memory address in HL register where the BCD number is stored
        ;Output: Decimal-adjusted BCD number in the accumulator and the carry in register B

        ADD M           ;Add packed BCD byte and adjust it for BCD sum
        DAA
        RNC             ;If no carry, go back to next BCD
        MOV D,A         ;If carry is generated, save the sum from the accumulator
        MOV A,B         ;Transfer CY sum from register B and add 01
        ADI 01H
        DAA             ;Decimal-adjust BCD from B
        MOV B,A         ;Save adjusted BCD in B
        MOV A,D         ;Place BCD1 and BCD2 in the accumulator
        RET

UNPAK:  ;This subroutine unpacks the BCD in the accumulator and the carry register and stores
        ;  them in the output buffer
        ;Input: BCD number in the accumulator, and the buffer address in HL registers
        ;Output: Unpacked BCD in the output buffer

        MOV D,A         ;Save BCD number
        ANI 0FH         ;Mask high-order BCD
```

```
MOV M,A     ;Store low-order BCD
DCX H       ;Point to next memory location
MOV A,D     ;Get BCD again
ANI F0H     ;Mask low-order BCD
RRC         ;Convert the most significant four bits into unpacked BCD
RRC
RRC
RRC
MOV M,A     ;Store high-order BCD
DCX H       ;Point to the next memory location
RET
```

PROGRAM DESCRIPTION

1. The expected maximum sum is 9090, which requires two registers. The main program clears the accumulator to save BCD_1 and BCD_2, clears register B to save BCD_3 and BCD_4, and calls the subroutine to add the numbers. The BCD bytes are added until the counter C becomes zero.
2. The BCDADD subroutine is an illustration of the multiple-ending subroutine. It adds a byte, decimal-adjusts the accumulator and, if there is no carry, returns the program execution to the main program. If there is a carry, it adds 01 to the carry register B by transferring the contents to the accumulator and decimal-adjusting the contents. The final sum is stored in registers A and B.
3. The main program calls the UNPAK subroutine, which takes the BCD number from the accumulator (e.g., 57_{BCD}), unpacks it into two separate BCDs (e.g., 05_{BCD} and 07_{BCD}), and stores them in the output buffer. When a subroutine stores a BCD number in memory, it decrements the index because BCD_1 is stored first.

See Assignments 15–16 at the end of this chapter.

10.6 BCD SUBTRACTION

When subtracting two BCD numbers, the instruction DAA cannot be used to decimal-adjust the result of two packed BCD numbers; the instruction applies only to addition. Therefore, it is necessary to devise a procedure to subtract two BCD numbers. Two BCD numbers can be subtracted by using the procedure of 100's complement (also known as 10's complement), similar to 2's complement. The 100's complement of a subtrahend can be added to a minuend as illustrated:

For example, 82 – 48 (= 34) can be performed as follows:

100's complement of subtrahend	52	(100 – 48 = 52)
Add minuend	+ 82	
	1/34	

The sum is 34 if the carry is ignored. This is similar to subtraction by 2's complement. However, in an 8-bit microprocessor, it is not a simple process to find 100's complement of a subtrahend (100_{BCD} requires twelve bits). Therefore, in writing a program, 100's complement is obtained by finding 99's complement and adding 01.

10.6.1 Illustrative Program: Subtraction of Two Packed BCD Numbers

PROBLEM STATEMENT

Write a subroutine to subtract one packed BCD number from another BCD number. The minuend is placed in register B, and the subtrahend is placed is register C by the calling program. Return the answer into the accumulator.

SUBROUTINE

```
SUBBCD:  ;This subroutine subtracts two BCD numbers and adjusts the result to
         ;  BCD values by using the 100's complement method
         ;Input: A minuend in register B and a subtrahend in register C
         ;Output: The result is placed in the accumulator
         MVI A,99H
         SUB C          ;Find 99's complement of subtrahend
         INR A          ;Find 100's complement of subtrahend
         ADD B          ;Add minuend to 100's complement of subtrahend
         DAA            ;Adjust for BCD
         RET
```

See Assignments 17 and 18 at the end of this chapter.

10.7 INTRODUCTION TO ADVANCED INSTRUCTIONS AND APPLICATIONS

The instructions discussed in the last several chapters deal primarily with 8-bit data (except LXI). However, in some instances data larger than eight bits must be manipulated, especially in arithmetic manipulations and stack operations. Even if the 8085 is an 8-bit microprocessor, its architecture allows specific combinations of two 8-bit registers to form 16-bit registers. Several instructions in the instruction set are available to manipulate 16-bit data. These instructions will be introduced in this section.

10.7.1 16-Bit Data Transfer (Copy) and Data Exchange Group

LHLD: Load HL registers direct

- ☐ This is a 3-byte instruction
- ☐ The second and third bytes specify a memory location (the second byte is a line number and the third byte is a page number)

- ☐ Transfers the contents of the specified memory location to L register
- ☐ Transfers the contents of the next memory location to H register

SHLD: Store HL registers direct

- ☐ This is a 3-byte instruction
- ☐ The second and third bytes specify a memory location (the second byte is a line number and the third byte is a page number)
- ☐ Stores the contents of L register in the specified memory location
- ☐ Stores the contents of H register in the next memory location

XCHG: Exchange the contents of HL and DE

- ☐ This is a 1-byte instruction
- ☐ The contents of H register are exchanged with the contents of D register, and the contents of L register are exchanged with the contents of E register

---

Example 10.3

Memory locations 2050H and 2051H contain 3FH and 42H, respectively, and register pair DE contains 856FH. Write instructions to exchange the contents of DE with the contents of the memory locations.

					Memory	
Before Instructions:	D	85	6F	E	3F	2050
					42	2051

Instructions

Machine Code	Mnemonics					
2A	LHLD 2050H				3F	2050
50					42	2051
20		H	42	3F	L	
EB	XCHG	D	42	3F	E	3F 2050
		H	85	6F	L	42 2051
22	SHLD 2050H				6F	2050
50					85	2051
20		H	85	6F	L	

10.7.2 Arithmetic Group

Operation: Addition with Carry

ADC R
ADC M
ACI 8-bit

- ☐ These instructions add the contents of the operand, the carry, and the accumulator. All flags are affected

Example 10.4

Registers BC contain 2793H, and registers DE contain 3182H. Write instructions to add these two 16-bit numbers, and place the sum in memory locations 2050H and 2051H.

Before instructions:

B	27	93	C
D	31	82	E

Instructions

MOV A,C	A	93		F		93H
ADD E	A	15	CY = 1	F	+	82H
MOV L,A	H		15	L		1/15H
MOV A,B						27H
ADD D						+ 31H
MOV H,A	H	59	15	L		59H
SHLD 2050H						

Operation: Subtraction with Carry

SBB R
SBB M
SBI 8-bit

- These instructions subtract the contents of the operand and the borrow from the contents of the accumulator

Example 10.5

Registers BC contain 8538H and registers DE contain 62A5H. Write instructions to subtract the contents of DE from the contents of BC, and place the result in BC.

Instructions

MOV A,C	(B)	85	38	(C)
SUB E			−	
MOV C,A	(D)	62	A5	(E)
MOV A,B		−1	1/93	
SBB D	(B)	22	93	(C)
MOV B,A				

Operation: Double Register ADD

DAD Rp
DAD B
DAD D
DAD H
DAD SP

Add register pair to register HL

- This is a 1-byte instruction
- Adds the contents of the operand (register pair or stack pointer) to the contents of HL registers
- The result is placed in HL registers
- The Carry flag is altered to reflect the result of the 16-bit addition. No other flags are affected
- The instruction set includes four instructions

Write instructions to display the contents of the stack pointer register at output ports. Example 10.6

Instructions

```
LXI H,0000H     ;Clear HL
DAD SP          ;Place the stack pointer contents in HL
MOV A,H         ;Place high-order address of the stack pointer in the accumulator
OUT PORT1
MOV A,L         ;Place low-order address of the stack pointer in the accumulator
OUT PORT2
```

The instruction DAD SP adds the contents of the stack pointer register to the HL register pair, which is already cleared. This is the only instruction in the 8085 that enables the programmer to examine the contents of the stack pointer register.

10.7.3 Instructions Related to the Stack Pointer and the Program Counter

XTHL: Exchange Top of the Stack with H and L

- ☐ The contents of L are exchanged with the contents of the memory location shown by the stack pointer, and the contents of H are exchanged with the contents of memory location of the stack pointer +1.

Write a subroutine to set the Zero flag and check whether the instruction JZ (Jump on Zero) functions properly, without modifying any register contents other than flags. Example 10.7

Subroutine

```
CHECK:   PUSH H
         MVI L,FFH      ;Set all bits in L to logic 1
         PUSH PSW       ;Save flags on the top of the stack
         XTHL           ;Set all bits in the top stack location
         POP PSW        ;Set Zero flag
         JZ NOEROR
         JMP ERROR
NOEROR:  POP H
         RET
```

The instruction PUSH PSW places the flags in the top location of the stack, and the instruction XTHL changes all the bits in that location to logic 1. The instruction POP PSW sets all the flags. If the instruction JZ is functioning properly, the routine returns to the calling program; otherwise, it goes to the ERROR routine (not shown). This example shows that the flags can be examined, and they can be set or reset to check malfunctions in the instructions.

SPHL: Copy H and L registers into the Stack Pointer Register
- ☐ The contents of H specify the high-order byte and the contents of L specify the low-order byte
- ☐ The contents of HL registers are not affected

This instruction can be used to load a new address in the stack pointer register. (For an example, see SPHL in the instruction set, Appendix F.)

PCHL: Copy H and L registers into the Program Counter
- ☐ The contents of H specify the high-order byte and the contents of L specify the low-order byte

Example 10.8 Assume that the HL registers hold address 2075H. Transfer the program to location 2075H.

Solution The program can be transferred to location 2075H by using Jump instructions. However, PCHL is a 1-byte instruction that can perform the same function as the Jump instruction. (For an illustration, see the instruction PCHL in the instruction set, Appendix F.)

This instruction is commonly used in monitor programs to transfer the program control from the monitor program to the user's program (see Chapter 17, Section 17.5).

10.7.4 Miscellaneous Instruction

CMC: Complement the Carry Flag (CY)
If the Carry flag is 1, it is reset; and if it is 0, it is set

STC: Set the Carry Flag

These instructions are used in bit manipulation, usually in conjunction with rotate instructions. (See the instruction set in Appendix F for examples.)

10.8 MULTIPLICATION

Multiplication can be performed by repeated addition; this technique is used in BCD-to-binary conversion. It is, however, an inefficient technique for a large multiplier. A more efficient technique can be devised by following the model of manual multiplication of decimal numbers. For example,

$$\begin{array}{lrr} & & 108 \\ & & \times\ \ 15 \\ \textbf{Step 1:} & (108 \times 5) = & 540 \\ \textbf{Step 2: Shift left and add} & (108 \times 1) = & +\ 108 \\ & & 1620 \end{array}$$

In this example, the multiplier multiplies each digit of the multiplicand, starting from the farthest right, and adds the product by shifting to the left. The same process can be applied in binary multiplication.

10.8.1 Illustrative Program: Multiplication of Two 8-Bit Unsigned Numbers

PROBLEM STATEMENT

A multiplicand is stored in memory location XX50H and a multiplier is stored in location XX51H. Write a main program to

1. transfer the two numbers from memory locations to the HL registers.
2. store the product in the Output Buffer at XX90H.

Write a subroutine to

1. multiply two unsigned numbers placed in registers H and L.
2. return the result into the HL pair.

MAIN PROGRAM

```
        LXI SP,STACK
        LHLD XX50H      ;Place contents of XX50 in L register and contents of
                        ;  XX51 in H register
        XCHG            ;Place multiplier in D and multiplicand in E
        CALL MLTPLY     ;Multiply the two numbers
        SHLD XX90H      ;Store the product in locations XX90 and 91H
        HLT
```

Subroutine

```
        MLTPLY: This subroutine multiplies two 8-bit unsigned numbers
        ;Input: Multiplicand in register E and multiplier in register D
        ;Output: Results in HL register

MLTPLY: MOV A,D         ;Transfer multiplier to accumulator
        MVI D,00H       ;Clear D to use in DAD instruction
        LXI H,0000H     ;Clear HL
        MVI B,08H       ;Set up register B to count eight rotations
NXTBIT: RAR             ;Check if multiplier bit is 1
        JNC NOADD       ;If not, skip adding multiplicand.
        DAD D           ;If multiplier is 1, add multiplicand to HL and place
                        ;  partial result in HL
```

```
NOADD:  XCHG        ;Place multiplicand in HL
        DAD H       ;And shift left
        XCHG        ;Retrieve shifted multiplicand
        DCR B       ;One operation is complete, decrement counter
        JNZ NXTBIT  ;Go back to next bit
        RET
```

PROGRAM DESCRIPTION

1. The objective of the main program is to demonstrate use of the instructions LHLD, SHLD, and XCHG. The main program transfers the two bytes (multiplier and multiplicand) from memory locations to the HL registers by using the instruction LHLD, places them in the DE register by the instruction XCHG, and places the result in the Output Buffer by the instruction SHLD.
2. The multiplier routine follows the format—add and shift to the left—illustrated at the beginning of Section 10.8. The routine places the multiplier in the accumulator and rotates it eight times until the counter (B) becomes zero. The reason for clearing D is to use the instruction DAD to add register pairs.
3. After each rotation, when a multiplier bit is 1, the instruction DAD D performs the addition, and DAD H shifts bits to the left. When a bit is 0, the subroutine skips the instruction DAD D and just shifts the bits.

10.9 SUBTRACTION WITH CARRY

The instruction set includes several instructions specifying arithmetic operations with carry (for example, add with carry or subtract with carry, Section 10.7.2). Descriptions of these instructions convey an impression that these instructions can be used to add (or subtract) 8-bit numbers when the addition generates carries. In fact, in these instructions when a carry is generated, it is added to bit D_0 of the accumulator in the next operation. Therefore, these instructions are used primarily in 16-bit addition and subtraction, as shown in the next program.

10.9.1 Illustrative Program: 16-Bit Subtraction

PROBLEM STATEMENT

A set of five 16-bit readings of the current consumption of industrial control units is monitored by meters and stored at memory locations starting at XX50H. The low-order byte is stored first (e.g., at XX50H), followed by the high-order byte (e.g., at XX51H). The corresponding maximum limits for each control unit are stored starting at XX90H. Subtract each reading from its specified limit, and store the difference in place of the readings. If any reading exceeds the maximum limit, call the indicator routine and continue checking.

MAIN PROGRAM

```
        LXI D, 2050H      ;Point index to readings
        LXI H, 2080H      ;Point index to maximum limits
        MVI B,05H         ;Set up B as a counter
NEXT:   CALL SBTRAC       ;Point to next location
        INX D             ;Point to next location
        INX H
        DCR B
        JNZ NEXT
        HLT
```

Subroutine

```
;SBTRAC: This subroutine subtracts two 16-bit numbers
;Input: The contents of registers DE point to reading locations
;       The contents of registers HL point to maximum limits
;Output: The results are placed in reading locations, thus destroying the initial readings
;The comment section illustrates one example, assuming the following data:
```

Memory Contents

```
The first current reading = 6790H    2050 = 90H    LSB
                                     2051 = 67H    MSB
Maximum limit = 7000 H               2090 = 00H    LSB
                                     2091 = 70H    MSB
```

```
                        ;Illustrative Example
SBTRAC:  MOV A,M;(A)    ;(A)  = 00H LSB of maximum limit
         XCHG           ;(HL) = 2050H
         SUB M          ;(A)  = 0000 0000  2's complement of 90H
                        ;(M)  = 0111 0000  Borrow flag is set to
                                1 0111 0000  indicate the result is in
                                             2's complement.
         MOV M,A        ;Store at 2050H
         XCHG           ;(HL) = 2090H
         INX H          ;(HL) = 2091H
         INX D          ;(DE) = 2051H
         MOV A,M        ;(A)  = 70H  MSB of the maximum limit
         XCHG           ;(HL) = 2051H
         SBB M          ;(A)  = 0111 0000  (70H)
                        ;(M)  = 1001 1001  2's complement of 67H
                        ;(CY) =         1  Borrow flag
         CC INDIKET     ;Call Indicate subroutine if reading is higher than the
                        ;  maximum limit
         MOV M,A
         RET
```

PROGRAM DESCRIPTION

This is a 16-bit subtraction routine that subtracts one byte at a time. The low-order bytes are subtracted by using the instruction SUB M. If a borrow is generated, it is accounted for by using the instruction SBB M (Subtract with Carry) for high-order bytes. In the illustrative example, the first subtraction (00H – 90H) generates a borrow that is subtracted from high-order bytes. The instruction XCHG changes the index pointer alternately between the set of readings and the maximum limits.

SUMMARY

The following code conversion and 16-bit arithmetic techniques were illustrated in this chapter:

- ☐ BCD-to-binary (Section 10.1)
- ☐ Binary-to-BCD (Section 10.2)
- ☐ BCD to seven-segment-LED code (Section 10.3)
- ☐ Binary-to-ASCII code and ASCII-to-binary (Section 10.4)
- ☐ BCD addition and subtraction (Sections 10.5 and 10.6)
- ☐ Multiplication of two 8-bit unsigned numbers and 16-bit subtraction with carry (Section 10.7)

Review of Instructions

The instructions introduced and illustrated in this section are summarized below.

16-Bit Data Transfer (Copy) and Data Exchange Instructions

☐ LHLD 16-bit	Load HL registers direct
☐ SHLD 16-bit	Store HL registers direct
☐ XCHG	Exchange the contents of HL with DE
☐ XTHL	Exchange the top of the stack with HL
☐ SPHL	Copy HL registers into the stack pointer
☐ PCHL	Copy HL registers into the program counter

Arithmetic Instructions Used in 16-Bit Operations

Addition: The following instructions add the contents of the operand, the carry, and the accumulator.

ADC R	Add register contents with carry
ADC M	Add memory contents with carry
ACI 8-bit	Add immediate 8-bit data with carry

Subtraction: The following instructions subtract the contents of the operand and the borrow from the contents of the accumulator.

SBB R	Subtract register contents with borrow
SBB M	Subtract memory contents with borrow
SBI 8-bit	Subtract immediate 8-bit data with borrow

LOOKING AHEAD

This chapter included various types of code conversion techniques and illustrated applications of more advanced instructions. The illustrative programs were written as independent modules, similar to the circuit boards of an electronic system. Now, what is needed is to link these modules to perform a specific task.

However, single-board microcomputer systems are unsuitable for writing and coding programs with more than 50 instructions. Coding becomes cumbersome, modifications become tedious, and troubleshooting becomes next to impossible. To write a large program, therefore, it is necessary to have access to an assembler and a disk-based system, which will be discussed in the next chapter.

QUESTIONS AND PROGRAMMING ASSIGNMENTS

Section 10.1: BCD-to-Binary Conversion

1. Rewrite the BCDBIN subroutine to include storing results in the Output Buffer. Eliminate unnecessary PUSH and POP instructions.
2. Modify the program (Section 10.1.1) to convert a set of numbers of 2-digit BCD numbers into their binary equivalents and store them in the Output Buffer. The number of BCD digits in the set is specified by the main program in register D and passed on as a parameter to the subroutine.
3. Rewrite the multiplication section using the RLC (Rotate Left) instruction. *Hints:* Rotating left once is equivalent to multiplying by two. To multiply a digit by ten, rotate left three times and add the result of the first rotation (times 10 = times 8 + times 2).
4. In Assignment 3, multiplication is performed by rotating the high-order digit to the left. However, in the BCDBIN subroutine, just before the multiplication section, the high-order digit BCD_2 is shifted right to place it in the low-order position. Rewrite the subroutine to combine the two operations. *Hint:* Rotating BCD_2 right once from the high-order position is equivalent to multiplying it by eight.

Section 10.2: Binary-to-BCD Conversion

5. Assume the STACK is defined as 20B8H in the illustrative program to convert binary to unpacked BCD (Section 10.2.1). Specify the stack addresses and their (symbolic) contents when the BINBCD subroutine is called the second time.
6. Rewrite the main program to supply the powers of ten in registers B and C and to store converted BCD numbers in the Output Buffer. Modify the BINBCD subroutine to accommodate the changes in the main program and eliminate the PWRTEN subroutine.
7. Rewrite the program to convert a given number of binary data bytes into their BCD equivalents, and store them as unpacked BCDs in the Output Buffer. The number of data bytes is specified in register D in the main program. The converted numbers should be stored in groups of three consecutive memory locations. If the number is not large enough to occupy all three locations, zeros should be loaded in those locations.
8. A set of ten BCD readings is stored in the Input Buffer. Convert the numbers into binary and add the numbers. Store the sum in the Output Buffer; the sum can be larger than FFH.

Section 10.3: BCD-to-Seven-Segment-LED Code Conversion

9. List the common-cathode and the common-anode seven-segment-LED look-up table to include hexadecimal digits from 0 to F. (See Figure 4.9 for the diagram.)
10. A set of data is stored as unpacked (single-digit) BCD numbers in memory from XX50H to XX5FH. Write a program to look up the common-cathode-LED code for each reading and store the code from XX55H to XX64H (initial data can be eliminated).
11. Design a counter to count continuously from 00H to FFH with 500 ms delay between each count. Display the count at PORT1 and PORT2 (one digit per port) with a common-anode seven-segment-LED code.
12. Modify the LEDCOD subroutine to account for a carry when the instruction ADD L is executed. For example, if the starting address of the CODE table is 02F9, the codes of digits larger than six will be stored on page 03H. The subroutine given in the illustrative program to convert BCD to LED code (Section 10.3.1) does not account for such a situation. *Hint:* Check for the carry (CY) after the addition, and increment H whenever a carry is generated.

Section 10.4: Binary-to-ASCII Code Conversion

13. A set of ASCII Hex digits is stored in the Input-Buffer memory. Write a program to convert these numbers into binary. Add these numbers in binary, and store the result in the Output-Buffer memory.
14. Extend the program in Assignment 13 to convert the result from binary to ASCII Hex code.

Section 10.5: BCD Addition

15. Write a counter program to count continuously from 0 to 99 in BCD with a delay of 750 ms between each count. Display the count at an output port.

16. Modify the illustrative program for the addition of unsigned numbers (Section 10.5.1) to convert the unpacked BCD digits located at XX60 to XX63 into ASCII characters, and store them in the Output Buffer.

Section 10.6: BCD Subtraction

17. Design a down-counter to count from 99 to 0 in BCD with 500 ms delay between each count. Display the count at an output port. *Hint:* Check for the low-order digit; when it reaches zero, adjust the next digit to nine.

18. Write a program to subtract a 2-digit BCD number from another 2-digit BCD number; the numbers are stored in two consecutive memory locations. Rather than using the 100's complement method, subtract one BCD digit at a time, and decimal-adjust each digit after subtraction. (The instruction DAA cannot be used in subtraction.) Display the result at an output port. Verify that if the subtrahend is larger than the minuend, the result will be negative and will be displayed as the 100's complement.

Section 10.7: Advanced Instructions and Applications

19. A set of 16-bit readings is stored in memory locations starting at 2050H. Each reading occupies two memory locations: the low-order byte is stored first, followed by the high-order byte. The number of readings stored is specified by the contents of register B. Write a program to add all the readings and store the sum in the Output-Buffer memory. (The maximum limit of a sum is 24 bits.)

20. In Assignment 19, save the contents of the stack pointer from the main program, point the stack pointer to location 2050H, and transfer the readings to registers by using the POP instruction. Add the readings as in Assignment 19; however, retrieve the original contents of the stack pointer after the addition is completed.

21. Assume that the monitor program stores a memory address in the DE registers. When a Hex key is pressed to enter a new memory address, the keyboard subroutine places the 4-bit code of the key pressed in the accumulator. Write a subroutine to shift out the most significant four bits of the old address and to insert the new code from the accumulator as the least significant four bits in register E.

Hint: Place the memory address in the HL registers, and use the instruction DAD H four times; this will shift all bits to the left by four positions and will clear the least significant four bits.

22. A pair of 32-bit readings is stored in groups of four consecutive memory locations; the memory location with the lowest memory address in each group contains the least significant byte. Write a program to add these readings; if a carry is generated, call an error routine.