

JR NZ, END	;Is time = 60 seconds? If not, go back to main ;program
;Section III	
LD C, 00H	;60 Seconds complete, clear "Second" regis- ;ter
INC B	;Increment "Minutes"
LD A, B	
CP 05H	;Is it five minutes?
JR NZ, END	;If not, return to main program
LD BC, 0000H	;If it is, clear minute and second registers
END: RETN	

**Program Description** The main program clears registers B and C to store minutes and seconds respectively, enables the interrupts, sets up register D to count 60 interrupts, and displays the starting time in minutes (00) and seconds (00). When the first pulse interrupts the processor, program control is transferred to memory location 0066<sub>H</sub>, as mentioned earlier, and the Jump instruction at this location, provided by the system designer, will locate the service routine labelled as "TIMER."

In the service routine (Section I), register D is decremented every 1/60th of a second and the program is returned to the main routine. This is repeated 60 times. After the 60th interrupt, the counter D goes to zero and the program enters Section II. In this section, counter D is reloaded, the "Second" register is incremented and adjusted for BCD, and the program is returned to the main routine. For the next 60 interrupts, the program remains in Section I. When Section II is repeated 60 times, the program goes to Section III, where the "Minute" register is incremented and the "Second" register is cleared. If the Minute register has not reached the five-minute period, the routine returns to the main program. When the minute register reaches a five-minute period, register pair BC is cleared and the next period begins. In this particular program, the service routine does not save any register contents by using PUSH instructions before starting the service routine. In most service routines, register contents must be saved because the interrupt is asynchronous and can occur at any time.

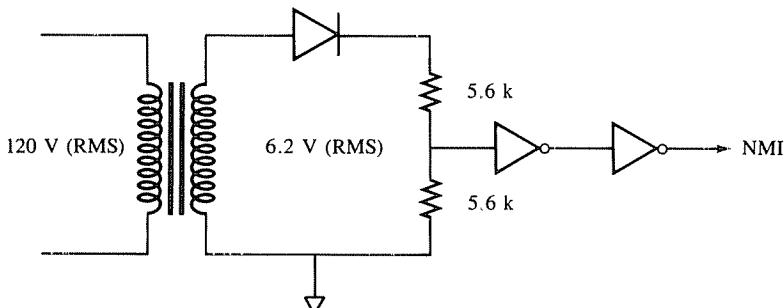


FIGURE 12.11  
Schematic for 60 Hz Powerline as an Interrupting Source

# 12.6

## MULTIPLE INTERRUPTS AND PRIORITIES

---

The Z80 microprocessor has one  $\overline{\text{INT}}$  pin for the maskable interrupt. However, Mode 0 suggests that at least eight different peripherals can be connected, and their requests can be transferred to eight restart locations. Mode 2 suggests almost unlimited possibilities to vector interrupt requests anywhere in memory. This raises two questions:

1. How do we connect more than one interrupting device to one interrupt line?
2. What happens if multiple interrupting devices request service simultaneously?

The method of connecting multiple devices to the  $\overline{\text{INT}}$  line of the Z80 is determined by the process of identifying an interrupting device. After the acknowledgment of an interrupt, the interrupting device can be identified either by the **polling method** (software technique) or by the **interrupt vector method** (hardware technique). In the polling method, the microprocessor queries each device using software instructions, identifies the device, and transfers the program to the appropriate service routine. On the other hand, in the interrupt vector method, the device identifies itself by supplying either an instruction or an address.

The next question is: What happens when devices request the service at the same time? In the polling technique, software determines the priority among the requesting devices and serves those devices in the sequence specified in the program. In the interrupt vector technique, priority is determined by the hardware. We will illustrate these techniques in the next two sections.

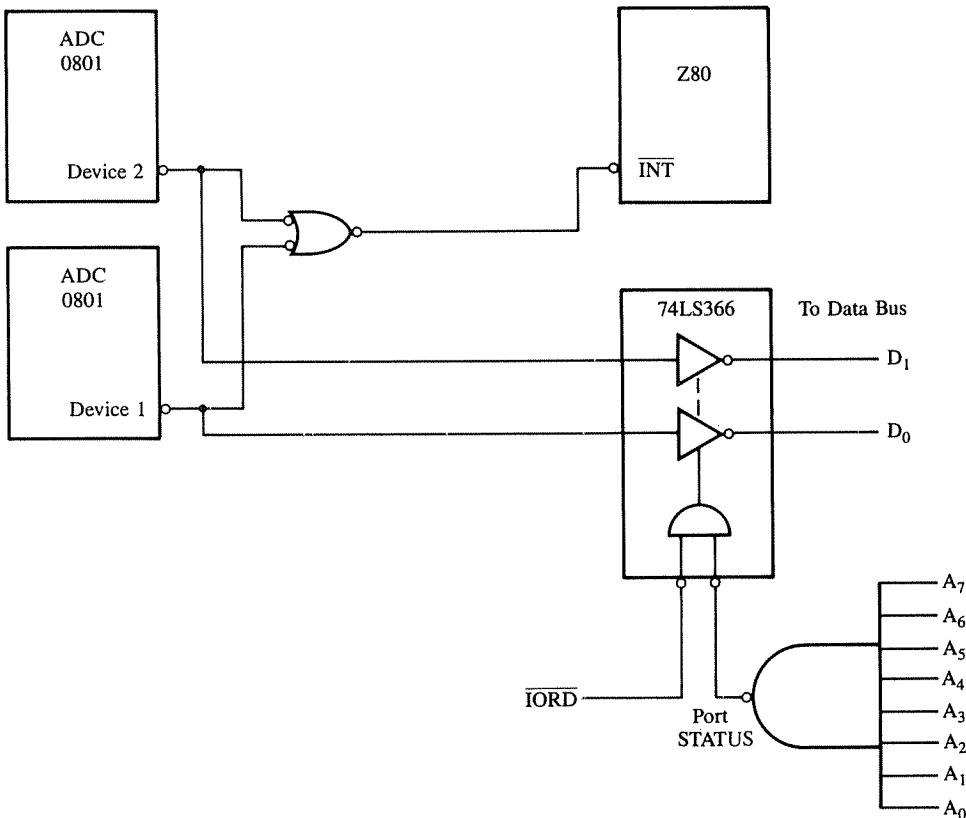
### 12.6.1 Polling Technique

Figure 12.12 shows an example in which two A/D converters are interfaced with the Z80 in interrupt Mode 1. The  $\overline{\text{INTR}}$  lines from the ADC0801 are logically ORed, and the output of the gate is connected to the  $\overline{\text{INT}}$  line of the Z80; either one or both converters can interrupt the processor. Figure 12.12 does not include any circuitry to turn off the  $\overline{\text{INT}}$  because the  $\overline{\text{INTR}}$  line of the ADC0801 goes inactive when the microprocessor reads the output.

To identify the interrupting data converters, an additional input port with the tri-state inverter (74LS366) is designed, and the  $\overline{\text{INTR}}$  lines of both converters are connected to the data bus lines ( $D_1$  and  $D_0$ ) through the 74LS366. The following subroutine identifies the interrupting devices and determines the priority between the two converters if they request the service simultaneously.

#### SERVICE ROUTINE

```
MODE1:    ;This is an interrupt service routine, written at location 0038H to
          ;respond to Mode 1 interrupt requests. It determines the priority
          ;between the two data converters and identifies them. In this
          ;routine Device 1 has higher priority than Device 2. After
```



**FIGURE 12.12**  
Multiple Interrupts with Polling Technique

```

;identifying the interrupting converter(s), it reads and stores data
;received from the converter(s), and initiates the next conversion.
PUSH AF          ;Save register contents
IN A, (STATUS)   ;Read tri-state inverter port
AND 00000011B    ;Mask data lines D7-D2
RRA              ;Place D0 in CY flag
CALL C, DVICE1   ;If D0 = 1, go to DVICE1 to read data
RRA              ;Place D1 in CY flag
CALL C, DVICE2   ;If D1 = 1, go to DVICE2 to read data
POP AF          ;Retrieve register contents
EI               ;Enable interrupt
RET

```

DVICE1:	PUSH AF	;Save interrupt status
	IN A, (ADC1)	;Read data from Device 1 and turn off INT
	LD (HL), A	;Save data in memory
	OUT (ADC1), A	;Start next conversion
	POP AF	;Retrieve register contents
	RET	
DVICE2:	PUSH AF	;Save interrupt status
	IN A, (ADC2)	;Read data from Device 2 and turn off INT
	LD (DE), A	;Save data in memory
	OUT (ADC2), A	;Start next conversion
	POP AF	;Retrieve register contents
	RET	

### PROGRAM DESCRIPTION

This service routine assumes that the main program sets up the interrupt mode and initializes memory pointers (HL and DE) to store data. Data converters are assigned port addresses ADC1 and ADC2 for both input and output, and the tri-state buffer is assigned the address STATUS as an input port.

To identify the interrupting device, the service routine reads the tri-state buffer and saves bits D<sub>1</sub> and D<sub>0</sub>. The routine checks first whether D<sub>0</sub> is at logic 1 (the 74LS366 inverts the interrupt request) by rotating D<sub>0</sub> into the CY flag because Device 1 has higher priority than Device 2. If D<sub>0</sub> is high, the routine calls DVICE1 to read data. To check whether Device 2 has also requested the service, the input reading from the STATUS port is rotated again to the right to place D<sub>1</sub> into the CY flag. If the CY is 1, the routine calls DVICE2; otherwise, it returns control to the main program.

Figure 12.12 shows only two interrupting devices; however, the polling technique and the circuit shown can be extended to include many devices. In place of the negative OR gate in Fig. 12.12, interrupt requests from many devices can be tied together using open collector logic devices. The disadvantage of the polling technique is the delayed response in servicing requests.

### 12.62 Interrupt Vector Technique

The schematic shown in Figure 12.13 implements multiple interrupting devices using the 8-to-3 priority encoder 74LS148. The encoder has eight input lines and three output lines; the output ranges from 000 to 111, thus encoding the eight inputs. However, the output is inverted. For example, when the input I<sub>7</sub> is active, the output is 000, and when the input I<sub>0</sub> is active, the output is 111. In addition to encoding the input, the encoder also determines the priorities among interrupting devices; the higher input signal has higher priority. For example, if I<sub>6</sub> and I<sub>4</sub> are active at the same time, it ignores I<sub>4</sub> and places the code of I<sub>6</sub> on the output lines. The encoder provides appropriate combinations on its output lines A<sub>0</sub>, A<sub>1</sub>, and A<sub>2</sub>, which are connected to data lines D<sub>0</sub>, D<sub>1</sub> and D<sub>2</sub>, respectively; other data lines are tied high. Even though the encoder has eight input lines, we will use only the even input lines I<sub>6</sub>, I<sub>4</sub>, I<sub>2</sub>, and I<sub>0</sub>, and they will be connected to four different interrupting devices. The reason to use only the even lines is that the Z80 expects the vector address to be even (this will be explained later).

When an interrupting device requests service, one of the input lines goes low, which makes the line GS low and interrupts the microprocessor. When the interrupt is acknowledged and the signal INTA enables the buffer 74LS366, the code corresponding to the input is placed on lines D<sub>2</sub>, D<sub>1</sub>, and D<sub>0</sub>. For example, if the interrupting device on line I<sub>0</sub> goes low, the output of the encoder will be 111. This code is inverted by the buffer 74LS366 and placed onto data lines D<sub>2</sub>, D<sub>1</sub>, and D<sub>0</sub>. Other data lines are high; thus, the byte 1111 1000 (F8<sub>H</sub>) is placed onto the data lines. Assuming that the Z80 is set up for interrupt Mode 2, and the interrupt register I is loaded with the byte 20<sub>H</sub>, the Z80 forms a vector address 20F8<sub>H</sub> for the interrupting device connected to I<sub>0</sub>. The program is transferred to location 20F8<sub>H</sub> to get the 16-bit address of the service routine stored in memory locations 20F8<sub>H</sub> and 20F9<sub>H</sub>. If the input I<sub>2</sub> goes low, the byte 1111 1010 is placed onto the data bus, and the program is transferred to location 20FA<sub>H</sub>. The Z80 expects the interrupt

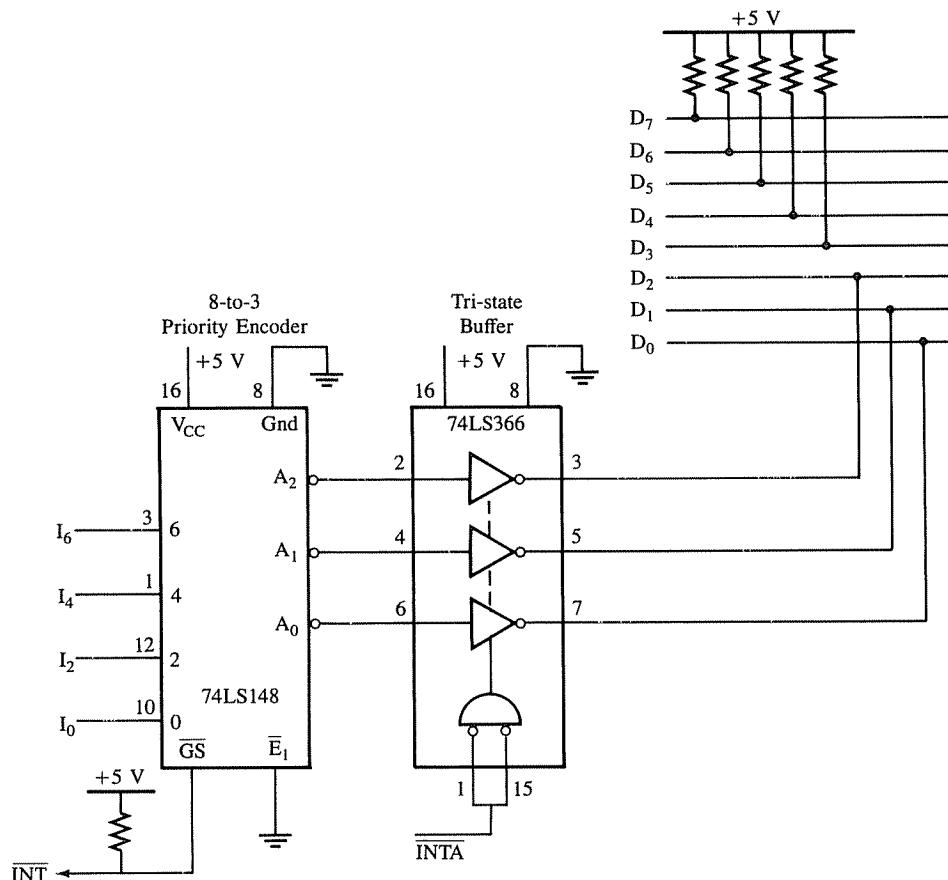


FIGURE 12.13  
Implementing Multiple Interrupts

vector to be an even memory address and uses two consecutive memory locations to obtain the address of a service routine; thus, we were unable to use the odd inputs to the encoder.

If there are simultaneous requests, the priorities are determined by the encoder; it responds to a higher level input, ignoring a lower level input. One of the drawbacks of this scheme is that the interrupting device connected to the input  $I_6$  always has the highest priority.

The interrupt scheme, similar to that illustrated in Figure 12.13, can also be implemented by using devices such as the Parallel Input/Output (PIO) controller, to be discussed in the next chapter. A programmable interrupt controller, such as the AMD AM9519A can also be used in implementing interrupts. This is quite a versatile device; it can accept interrupts from eight different devices, hold the requests, resolve priorities, and process them according to their priorities. It is also programmable, meaning its various operating modes can be determined by writing instructions into its internal registers.

## 12.7

### ILLUSTRATION: RESTART AS A SOFTWARE INSTRUCTION—IMPLEMENTATION OF BREAKPOINT TECHNIQUE

In Mode 0, external hardware is necessary to insert an RST instruction when an interrupt is acknowledged. However, the fact that RST is a software instruction is quite often overlooked or misunderstood. The RST instructions are commonly used to set up software breakpoints as a debugging technique. A breakpoint is a Restart (RST) instruction in a program where the execution of the program stops temporarily, and program control is transferred to the RST location. The program should be transferred from the RST location to a breakpoint service routine, which should allow the user to examine register or memory contents when specified keys are pressed. After the breakpoint routine, the program should return to the execution of the main program, where the breakpoint is set.

The breakpoint procedure allows the user to test programs in segments. For example, if RST 30H is written in a program, the program execution is transferred to location 0030<sub>H</sub>; it is equivalent to a 1-byte call instruction. This can be used to write a software breakpoint routine, as illustrated in the next problem.

#### 12.71 Problem Statement

Implement a breakpoint facility at RST 30H for the user. When the user writes RST 30H in the program, the program should

1. be interrupted at the instruction RST 30H.
2. display the accumulator contents and the flags when the Hex key A (1010)<sub>2</sub> is pressed.
3. exit the breakpoint routine and continue execution when the Zero key (0000)<sub>2</sub> is pressed.

Assume that the system includes a Hex keyboard and the keyboard routine KBRD is available. When it is called, it returns the binary key code of the key pressed in the accumulator. For example, if key 7 is pressed, the KBRD routine places 0111 in the accumulator.

## 12.72 Problem Analysis

The breakpoint routine is concerned with displaying the accumulator contents and the flags when the user writes the RST instruction in a program. The microprocessor executes the program until it fetches the RST instruction. Then the program is transferred to location  $0030_H$  and from  $0030_H$  to the breakpoint routine. The technique used to display register contents after executing a segment of the user's program is as follows:

1. Store the register contents on the stack and call the KBRD routine.
2. When the Hex key A is pressed, retrieve the contents of the accumulator and the flags from the stack by manipulating the stack pointer.
3. When the Zero key is pressed, return to the user's program.

## 12.73 Program: Breakpoint Subroutine

```
;BRKPNT: This is a breakpoint subroutine; it can be implemented with the
;           instruction RST 30H. It displays the accumulator and the flags when
;           the A key is pressed, and returns to the calling program when the
;           Zero key is pressed.
;Input : None
;Output : None
;Does not modify any register contents.
;Calls:    KBRD subroutine. The KBRD is a keyboard subroutine which checks a
;           key pressed. The routine identifies the key and places its binary
;           code into the accumulator.
```

BRKPNT:	PUSH AF	;Save registers
	PUSH BC	
	PUSH DE	
	PUSH HL	
KYCHK:	CALL KBRD	;Check for a key
	CP 0AH	;Is it key A?
	JP NZ, RETKY	;If not, check Zero key
	LD HL, 0007H	;Load stack pointer displacement count; see ;program description
	ADD HL, SP	;Place memory address in HL, where (A) is ;stored
	LD A, (HL)	;Get contents of A from stack
	OUT (PORT1), A	;Display accumulator contents

	DEC HL	;Point HL to the location of the flags
	LD A, (HL)	;Get flags
	OUT (PORT2), A	;Display flags
	JP KYCHK	;Go back and check next key
RETKY:	CP 00H	;Is it Zero key?
	JP NZ,KYCHK	;If not, go and check key program
	POP HL	;Retrieve registers
	POP DE	
	POP BC	
	POP AF	
	RET	

### 12.74 Program Description

The breakpoint routine saves all the registers on the stack, and the address in the stack pointer is decremented accordingly. (In this particular problem registers BC and DE need not be saved. These registers are saved to allow modifications in the program as given in the Assignments.) The accumulator contents are stored in the seventh memory location from the top of the stack, and the flags in the sixth memory location.

When key A is pressed, the HL register adds seven to the stack pointer contents and places that address in HL register (ADD HL, SP), without modifying the contents of the stack pointer. This is a critical point, because if the stack pointer is varied, appropriate contents may not be retrieved with POP and RET instructions. Finally, the subroutine displays the accumulator and the flags at the two output ports and returns to the main program.

---

## SUMMARY

---

- The interrupt is an asynchronous process of communication with the microprocessor, and is initiated by an external peripheral.
- The Z80 has two active low interrupt signals: maskable interrupt and nonmaskable interrupt (NMI). The maskable interrupt is level sensitive, and the nonmaskable is edge sensitive.
- The maskable interrupt can be enabled or disabled through the program control instructions EI and DI, and it has three operating modes: Modes 0, 1, and 2. The nonmaskable interrupt cannot be disabled.
- The Z80 instruction set includes eight RST instructions, which are equivalent to 1-byte calls to specific locations on memory page 00<sub>H</sub>. These can be used as software instructions to transfer the program control to their vectored locations on memory page 00<sub>H</sub> or can be inserted through external hardware in interrupt Mode 0.
- When the Z80 accepts a request in maskable interrupt I/O, it acknowledges the request by issuing a special M<sub>1</sub> cycle. During M<sub>1</sub>, the I/O Request (IORQ) signal goes active. The signals (M<sub>1</sub> and IORQ) are logically ANDed to generate the Inter-

**TABLE 12.2**  
Summary of Z80 Interrupt Process

Interrupts	Conditions to Accept Interrupt Requests	Software Instruction	External Hardware	Restart Locations
Nonmaskable Interrupt ( <u>NMI</u> ) <input type="checkbox"/> Edge Sensitive <input type="checkbox"/> Pin 17 (NMI)	<u>BUSRQ</u> Inactive  <u>NMI</u> Active Low	No Effect of EI or DI	Not Required	0066 <sub>H</sub>
Maskable Interrupt <input type="checkbox"/> Level Sensitive <input type="checkbox"/> Pin 16 (INT) Mode 0	<u>BUSRQ</u> Inactive  <u>NMI</u> Inactive <u>INT</u> Active Low	Must Be Enabled by EI and Can Be Disabled	RST Instruction	Eight locations: See Table 12.1
Mode 1 Mode 2		Uses I Register for High-order Byte	Not Required Low-order Byte	0038 <sub>H</sub> Any Memory Location

rupt Acknowledge (INTA) signal, which can be used to insert a hardware instruction or a byte.

- The general steps in the interrupt process are as follows. The Z80

1. disables the interrupt.
2. stores the contents of the program counter on the stack.
3. transfers the program to the memory location specified either by the external hardware or by the mode operation.
4. services the interrupt request.
5. fetches one of the return instructions, gets the address from the top of the stack, and returns to the program where the program was interrupted.

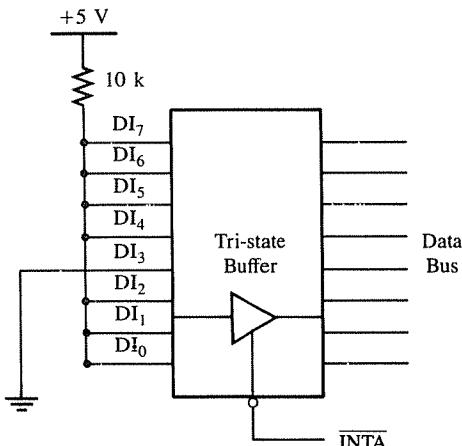
The operational details are summarized in Table 12.2.

## ASSIGNMENTS

1. Answer the following questions.
  - a. What is difference between the nonmaskable and the maskable interrupt?
  - b. When does the Z80 check the INT and NMI signals?
  - c. How is the Interrupt Acknowledge cycle differentiated from the Opcode Fetch and the I/O Read machine cycles?

- d. How is the INTA (Interrupt Acknowledge) signal generated?
- e. Specify the three conditions that are necessary to acknowledge the INT.
- f. Assuming the system's clock frequency is 2 MHz, and if the INT goes low in the first T-state of the OUT instruction (11 T-states), specify the period for which the INT has to remain active to be acknowledged.
- g. In a system with the clock frequency of 1 MHz, the Z80 begins to execute an instruction with ten T-states. If at the beginning of the instruction, the NMI goes active for 0.5  $\mu$ s, will the NMI be accepted, and if the answer is yes, when will it be accepted?
- h. The instruction CALL 2085H is written at memory locations 2017–18–19<sub>H</sub>. If it is interrupted during its execution, what is the address that is stored on the stack?
- i. The execution of the unconditional Call instruction requires 17 T-states. In a system with 2 MHz clock frequency, if the INT (set up in Mode 1) goes active at the beginning of the Call instruction and stays on for 8.5  $\mu$ s, and if the NMI goes active 2  $\mu$ s later than the INT and stays on for 1  $\mu$ s, where will the program be transferred?
- j. Does the system Reset disable the maskable interrupt?
- k. If the Z80 is initialized in the interrupt Mode 1, what is the status of the interrupt flip-flop IFF2 when the Z80 acknowledges the NMI?
- l. If the instruction RST 20H is written in a program at location 2051<sub>H</sub>, where will the program be transferred and what will be stored on the stack when the instruction is executed?
2. Identify the RST instruction shown in Figure 12.14 and answer the following questions.
- a. Specify the Restart memory location when the microprocessor is interrupted.
- b. If the instruction in the monitor program at 0030 is CALL 20BF<sub>H</sub> and the service routine is written at 20BF<sub>H</sub>, what instruction is necessary at location 0033<sub>H</sub>?

**FIGURE 12.14**  
RST Instruction for Assignment 2



3. The main program is stored beginning at  $0100_H$ . The main program has called the subroutine at  $0150_H$ , and when the microprocessor is executing the instruction at location  $0151_H$  (LD), it is interrupted. Read the program and then answer the questions that follow.

START:	0100      LD SP, 0400H
	0103      IM 1
	0104      EI
	↓
	0120      CALL 0150H
	↓
SUB:	0150      PUSH BC
	0151      LD BC, 10FFH
	0154      LD C, A
	↓
	015E      POP BC
	015F      RET

- a. Specify the contents of the stack location  $03FF_H$ .
  - b. Specify the stack locations where the contents of registers BC are stored.
  - c. When the program is interrupted, what is the memory address stored on the stack?
4. In 3, if the program is changed as follows and the circuit in Figure 12.10 is used to supply the byte, specify the location to which the program will be transferred when it is interrupted and the location of the service routine.

START:	0100      LD SP, 2400H
	0103      IM 2
	0104      LD A, 01H
	0106      LD I, A
	0107      LD HL, 01F8H
	010A      LD (HL), 80H
	010C      INC HL
	010D      LD (HL), 23H
	010F      EI
	0110      ↓

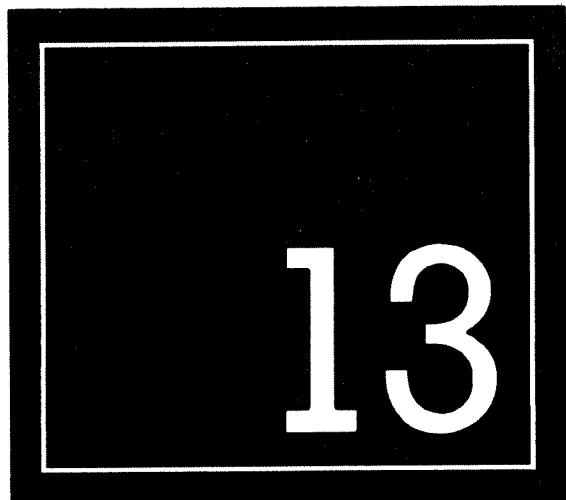
5. In Figure 12.13, if the input  $I_4$  goes active, specify the location to which the program will be transferred.
6. In Figure 12.13, if the inputs  $I_4$  and  $I_6$  go active simultaneously, specify the location to which the program will be transferred.
7. In Figure 12.13, connect the output lines  $A_2$ ,  $A_1$ , and  $A_0$  of the encoder (through the buffer 74LS366) to the data bus  $D_5$ ,  $D_4$ , and  $D_3$  and tie the remaining data lines high. If the Z80 is set up in the interrupt Mode 0, and if the input  $I_4$  goes active, where will the program be transferred?

8. A program is stored in memory from  $2000_H$  to  $205F_H$ . To check the first segment of the program up to location  $2025_H$ , a breakpoint routine call is inserted at location  $2026_H$ . (Refer to the breakpoint routine—Section 12.73.) If the stack pointer is initialized at  $2099_H$ , answer the following questions.
  - a. Specify the contents of the memory locations  $2098_H$  and  $2097_H$ .
  - b. Specify the memory locations where the accumulator content and the flags are stored when the microprocessor executes the instruction PUSH AF in the breakpoint routine.
  - c. Specify the memory locations where HL register contents are stored after execution of the instruction PUSH HL.
  - d. Specify the contents of the stack pointer when the breakpoint routine returns from the KBRD routine.
  - e. What address is placed in the program counter when the instruction RET is executed?
9. Modify the breakpoint routine (Section 12.7) to display the memory location at which the breakpoint is inserted into a program.
10. Modify the breakpoint routine to display the contents of BC, DE, and HL registers when user pushes the Hex Keys 1, 2, and 3. (The respective Hex codes are 01, 02, and 03.)

# Programmable Interface Devices

A **programmable interface device** is designed to perform various input/output functions, and these functions can be programmed into the device by writing an instruction (or instructions) in its internal register, called the control register. Functions can also be changed by writing a new instruction in the control register. These devices are flexible, versatile, and economical; they are widely used in microprocessor-based products.

In Chapter 5, we used simple integrated circuits, such as latches and tri-state buffers for I/O functions. However, they are limited in their capabilities; each device can perform only one function, and they are hardwired. In this chapter, we first discuss the basic concepts in programmable devices and then examine the Z80 **Parallel Input Output (PIO)** device in the context of these concepts. The PIO is an I/O device specially designed to function with the Z80, and it is commonly used in Z80-based systems. The PIO has two I/O ports, and it can be programmed in various modes ranging from bit mode to bidirectional data transfer mode. These modes are illustrated with several interfacing applications, such as keyboard and seven-segment display, and bidirec-



tional data transfer between two microcomputers. Finally, another widely used peripheral device, the Intel 8255, is described and compared with the Z80 PIO.

## OBJECTIVES

- List elements and characteristics of a typical programmable device.
  - Explain the functions of handshake signals.
  - List the elements of the Z80 PIO (Parallel Input Output) and explain its various operating modes.
  - Write initialization instructions to set up the PIO in a given mode.
  - Design an interfacing circuit to set up the PIO in a handshake mode, and write instructions to transfer data using the interrupt I/O.
- List the elements of the Intel 8255 Programmable Peripheral Interface and its various operating modes.
  - Write initialization instructions to set up the 8255 in a given mode.
  - Compare the features of the Z80 PIO and the 8255.

## 13.1

### BASIC CONCEPTS IN PROGRAMMABLE DEVICES

---

In Chapter 5, we discussed the interfacing of simple input (switches) and output (LEDs) devices. In the illustrations, we assumed that the I/O devices were always ready for data transfer. In fact, that assumption may not be valid in many data transfer situations. The MPU needs to check whether a peripheral is ready before it reads from or writes into a device because the execution speed of the microprocessor is much faster than the response of a peripheral such as a printer. For example, when the MPU sends data bytes (characters) to a printer, the microprocessor can execute the instructions to transfer a byte in microseconds, but the printer can take 10–25 ms to print a character. After transferring a character to the printer, the MPU should wait until the printer is ready for the next character; otherwise data will be lost. To prevent the loss of data or the MPU's reading the same data more than once, signals are exchanged between the MPU and a peripheral prior to actual data transfer; these signals are called **handshake signals**. To provide such signals in the illustrations of Chapter 5, we need to build additional logic circuitry.

In Chapter 12, we interfaced an A/D converter using the interrupt I/O; however, the interrupt signal was generated by the internal logic of the data converter. Many peripherals may not have that capability; such signals may have to be provided by the interfacing circuitry. In some applications, data flow is bidirectional (such as data transfer between two computers). In such a situation, the interfacing device should be capable of handling **bidirectional data flow**. Based on the above discussion and the illustrations of Chapters 5 and 12, we can summarize the requirements for a programmable interfacing device as follows. The device should include the following:

1. Input and output registers (a group of latches to hold data).
2. Tri-state buffers.
3. Capability for bidirectional data flow.
4. Handshake and interrupt signals.
5. Control logic.
6. Chip Select logic.
7. Interrupt control logic.

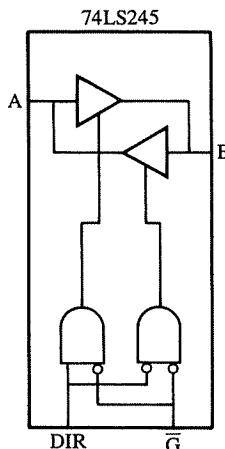
To understand the programmability of such a device, we illustrate a simple example of building a programmable device using a transceiver (bidirectional buffer) in the next section.

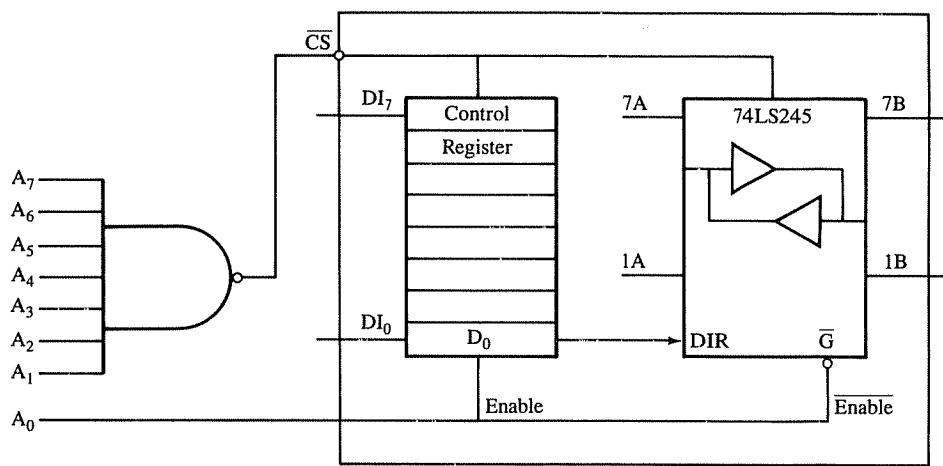
### 13.11 Making the 74LS245 Transreceiver Programmable

The 74LS245 is a bidirectional tri-state octal buffer, and the direction of the data flow is determined by the signal DIR. Figure 13.1 shows the logic diagram of the 74LS245; it shows one buffer (rather than eight) in each direction. The buffer is enabled when  $\bar{G}$  is active low; however, the direction of the data flow is determined by the DIR signal. When the DIR is high, data flow from A to B, and when it is low, data flow from B to A. In fact, this is a hardwired programmable device; the direction of the data flow is programmed through DIR. However, we are interested in a device that can be programmed by writing an instruction through the MPU. This can be accomplished by adding a register called the control register, as shown in Figure 13.2, and by connecting the DIR signal to bit  $D_0$  of the control register. When  $D_0 = 1$ , data flow from A to B as output, and when  $D_0 = 0$ , data flow in the opposite direction as input.

Now the question is: How would the MPU write into the control register? It does so the same way it would with any other I/O port, through a port address. Figure 13.2 shows that the address lines  $A_7$ – $A_1$  are used to select the chip through a NAND gate and  $A_0$  is used to differentiate between the control register and the transceiver. When  $A_0$  is high, the control register is enabled, and when  $A_0$  is low, the transceiver is enabled. Thus, the MPU could access the control register through the port address  $FF_H$ ; and the transceiver through  $FE_H$ . To set up the transceiver as an output device, the control word would be  $01_H$ , and to set it up as an input device the control word would be  $00_H$ .

**FIGURE 13.1**  
Logic Symbol of 74LS245  
Bidirectional Buffer





**FIGURE 13.2**  
Making 74LS245 Programmable

---

**Example  
13.1**

Write instructions to initialize the hypothetical chip (Figure 13.2) as an output buffer and send a byte.

---

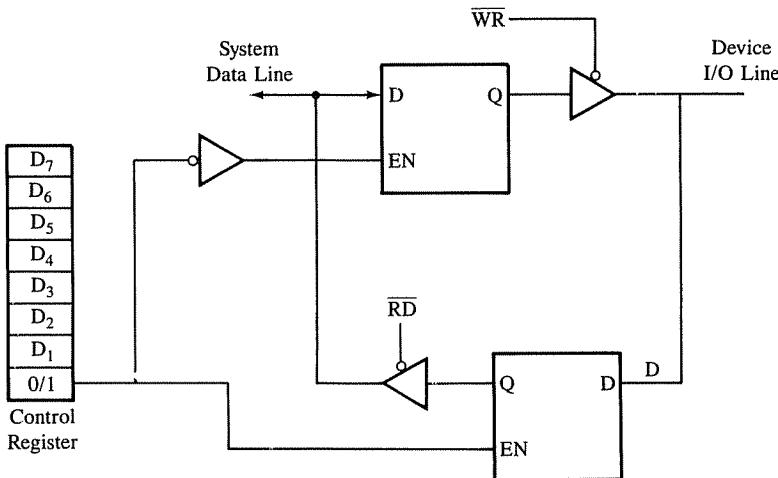
**Solution**

**Instructions**

LD A, 01H	;Set D <sub>0</sub> = 1, D <sub>1</sub> through D <sub>7</sub> are “don’t care” lines
OUT (FFH), A	;Write in the control register
LD A, BYTE1	;Load data byte
OUT (FEH), A	;Send data out

---

In the last example, we used the 74LS245 as a tri-state buffer. However, in microprocessor applications, we often need registers that can be used as I/O ports. We can build a latch with a buffer, and by controlling the enable signal of the latch, we can program it to function as an input port or an output port. Figure 13.3 shows two latches (representing eight latches in each direction); the enable signals of these latches are controlled by bit D<sub>0</sub> in the control register. If bit D<sub>0</sub> is 0, it enables the output latch, and if D<sub>0</sub> is 1, it enables the input latch. Thus, by programming bit D<sub>0</sub>, we can make the device function as an input port or an output port. When the device is programmed as an output device, the MPU can write to the port by using the WR control signal to enable the tri-state buffer and to send out a byte. When bit D<sub>0</sub> = 1, the input latch is enabled and the output latch is disabled, and the MPU can read by enabling the input buffer. If we have additional registers (or I/O ports),



**FIGURE 13.3**  
Programmable I/O Ports

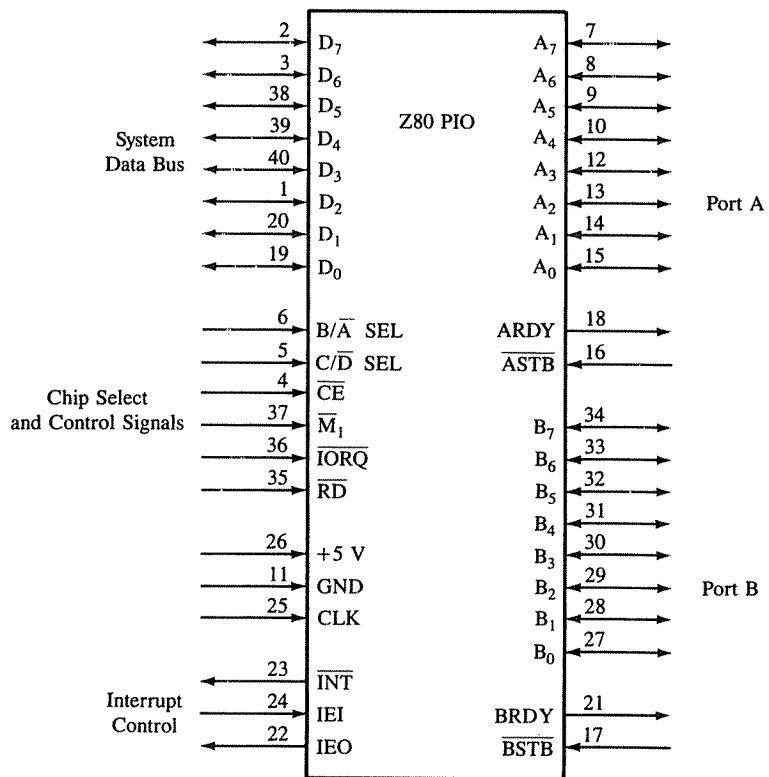
we can use other bits of the control register to define the functions of these registers. Similarly, we can build an interrupt logic and control it through a flip-flop that can in turn be enabled or disabled by a bit in the control register. Thus, we can build a programmable device to meet the requirements specified in the previous section.

## 13.2

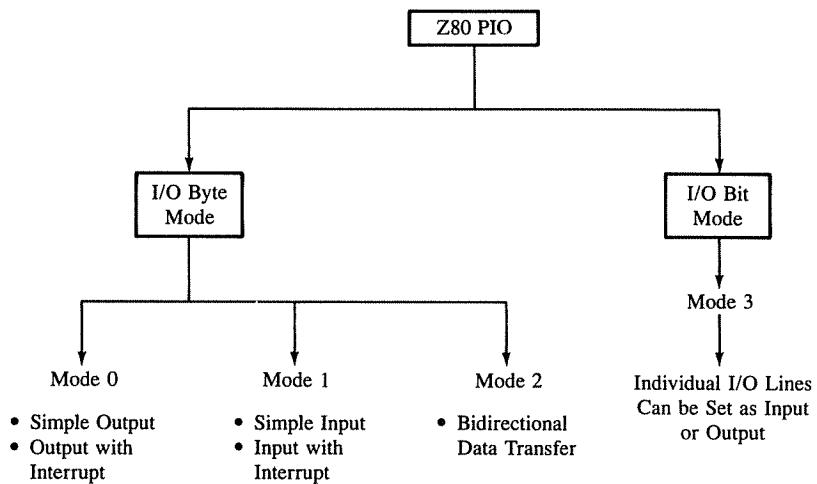
### Z80 PARALLEL INPUT/OUTPUT DEVICE (PIO)

The **Z80 PIO** is a programmable I/O interfacing device, specially designed for the Z80. It has two 8-bit I/O ports, A and B, and its signals are divided into six groups as shown in Figure 13.4; they are described in the next section. Ports A and B can be used in three different modes: byte output (Mode 0), byte input (Mode 1), and bit input/output (Mode 3) as shown in Figure 13.5. In addition, Port A can be configured in the bidirectional mode (Mode 2).

- *Modes 0 and 1.* Mode 0 is for output and Mode 1 is for input. In these modes, Ports A and B can be used in two ways: simple I/O without handshake signals or interrupt I/O with handshake signals. Each port has two handshake signals: *Strobe* and *Ready*.
- *Mode 2.* This mode specifies the bidirectional data flow. Only Port A can be configured in this mode, and it uses all four handshake signals.
- *Mode 3.* This is a bit mode whereby each bit of Port A and Port B can be configured as input or output. The handshake signals cannot be used in this mode.



**FIGURE 13.4**  
Z80 PIO Logic Pinout  
SOURCE: Courtesy of Zilog, Inc.



**FIGURE 13.5**  
Z80 PIO Modes

### 13.21 Z80 PIO Signals

As shown in Figure 13.4, the PIO signals are grouped in six categories.

1. Data bus—D<sub>7</sub>–D<sub>0</sub>: This is an 8-bit bidirectional, tri-state data bus, used to transfer information between the Z80 MPU and the PIO.
2. I/O lines—A<sub>7</sub>–A<sub>0</sub>: These are bidirectional tri-state I/O lines of Port A, used to transfer information between the PIO and a peripheral. These lines can source 250  $\mu$ A in logic 1 state and sink 2 mA in logic 0 state.  
B<sub>7</sub>–B<sub>0</sub>: These are Port B I/O lines similar to those of Port A. These lines can supply 1.5 mA at 1.5 V to drive Darlington transistors.
3. Handshake signals: The PIO has four handshake signals, two for each port. However, all of them are used for Port A when it is configured in the bidirectional mode.
  - ASTB—This is an active low Port A input signal from a peripheral to the PIO. When Port A is configured as an output port, this signal indicates the acknowledgement of the byte received by the peripheral. When Port A is configured as an input port, this signal indicates that a byte has been placed in Port A by a peripheral.
  - ARDY—This is an active high Port A output signal from the PIO to a peripheral. In the output mode, the signal indicates that a byte has been placed in the Port A register and is ready for data transfer. In the input mode, it indicates that the Port A register is empty and ready to accept the next byte from the peripheral.
  - BSTB and BRDY—These are handshake signals for Port B similar to those of Port A. However, these are used by Port A when Port A is configured in the bidirectional mode.
4. Power and clock: The PIO operates with a single power supply with + 5 V and uses a single phase system clock as an input for internal operations.
5. Interrupt control logic: The PIO has three signals to handle the interrupt I/O.
  - INT—Interrupt: This is an active low open collector output signal from the PIO; it is used to interrupt the Z80 MPU.
  - IEI—Interrupt Enable In: This is an active high input signal used to form a priority interrupt daisy chain when multiple peripherals are connected in the interrupt I/O (see Section 13.35 for the discussion of daisy chain priority interrupts). The high on this pin indicates that no other peripherals with higher priority are being serviced.
  - IEO—Interrupt Enable Out: This is an active high output signal used in daisy chain priority interrupts. This signal goes high when IEI is high and the Z80 is not servicing an interrupt from this PIO. This signal blocks lower priority devices from interrupting when a higher priority device is being serviced.
6. Control signals: The PIO has six control signals. The first three signals (CE, B/A, and C/D) determine the port addresses of the I/O registers A and B and their control registers. The remaining three signals define the type of the operation (Read or Write) being performed.

**TABLE 13.1**  
Z80 PIO Port Selection

CE	C/D	B/A	Selected Port
0	0	0	Data Port A
0	0	1	Data Port B
0	1	0	Control Register A
0	1	1	Control Register B
1	X	X	PIO Not Selected

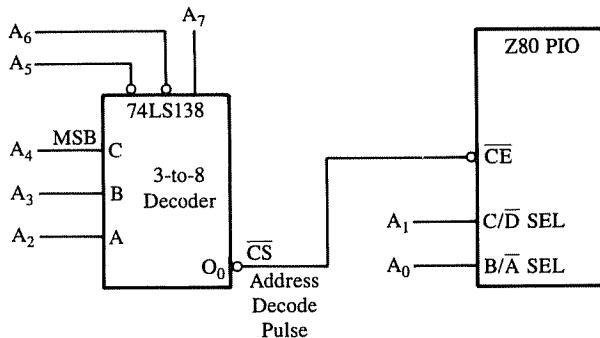
- $\overline{\text{CE}}$ —Chip Enable: This is an active low signal and is connected to a decoded address bus of the Z80.
- $\overline{\text{B/A}}$ —Port B or A Select: When this signal is high, Port B is selected, and when it is low, Port A is selected. This signal is generally connected to address line  $A_0$  of the MPU.
- $\overline{\text{C/D}}$ —Control or Data Select. When this signal is high, the control register is selected to write a command, and when it is low, the I/O register is selected to transfer data between the MPU and the PIO. This signal is generally connected to address line  $A_1$  of the MPU. The port selection is summarized in Table 13.1.
- $\overline{\text{M}_1}$ ,  $\overline{\text{RD}}$ , and  $\overline{\text{IORQ}}$ —All these signals are connected to the corresponding control signals of the Z80. The  $\overline{\text{M}_1}$  signal synchronizes the internal operation and the interrupt logic of the PIO and performs various functions in conjunction with the other two control signals as described below.
  - a. Read: When the  $\overline{\text{RD}}$  and  $\overline{\text{IORQ}}$  signals are active low, the MPU reads from the selected register.
  - b. Write: When the  $\overline{\text{IORQ}}$  is active, but the  $\overline{\text{RD}}$  is inactive, the MPU writes into the selected register. There is no specific control signal to write into register; it is a default condition.
  - c. Interrupt Acknowledge: When  $\overline{\text{M}_1}$  and  $\overline{\text{IORQ}}$  are active, the MPU acknowledges the interrupt from the PIO.
  - d. Reset: When  $\overline{\text{M}_1}$  is active and both  $\overline{\text{RD}}$  and  $\overline{\text{IORQ}}$  are inactive, the PIO is reset.

**Example  
13.2**

Figure 13.6 shows a circuit interfacing the PIO with the Z80 microprocessor. Identify the port addresses of Ports A and B and control registers.

**Solution**

In Figure 13.6, the output line  $O_0$  of the 74LS138 decoder is connected to the Chip Enable of the PIO. To assert the output line  $O_0$  of the decoder, the address line  $A_7$  should be at logic 1 and the remaining lines at logic 0. By combining these address lines with address lines  $A_1$  and  $A_0$ , the port addresses are as follows (refer to Table 13.1).

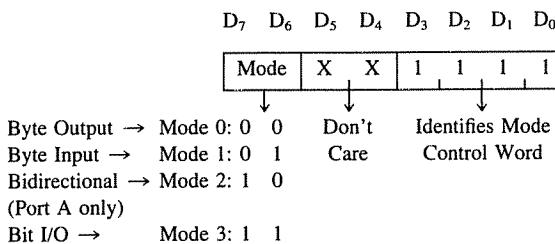


**FIGURE 13.6**  
Interfacing PIO

A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	C/D	B/A		
1	0	0	0	0	0	0	0	0	0	= 80 <sub>H</sub>	Data Port A
								0	1	= 81 <sub>H</sub>	Data Port B
								1	0	= 82 <sub>H</sub>	Control Register A
								1	1	= 83 <sub>H</sub>	Control Register B

### 13.22 Control Word

Figure 13.5 shows that the PIO can operate in four different modes. To set up an operating mode, the appropriate control word must be written in the control register of the port being used. The control word is determined by the internal logic (as discussed in Section 13.1), and it is specified by the manufacturer. The control word for the PIO to specify the modes is shown in Figure 13.7, and how to initialize a port is illustrated in Example 13.3.



**FIGURE 13.7**  
Z80 PIO Mode Word

**Example  
13.3**

In Figure 13.8, eight DIP switches are connected to Port A and eight LEDs are connected to Port B (the buffer is necessary to supply sufficient current to the LEDs). Write instructions to initialize Port A as an input port and Port B as an output port. Read Port A and turn on the corresponding LEDs in Port B. Assume that the decoding logic is the same as in Figure 13.6.

**Solution**

To initialize Port A as an input port,  $D_7 = 0$  and  $D_6 = 1$ , and to initialize Port B as an output port,  $D_7$  and  $D_6$  should be both 0. Thus the control words are

Port A as Input Port:  $0\ 1\ 0\ 0\ 1\ 1\ 1\ 1 = 4F_H$   
  
 Mode 1

Port B as Output Port:  $0\ 0\ 0\ 0\ 1\ 1\ 1\ 1 = 0F_H$   
  
 Mode 0

**Instructions**

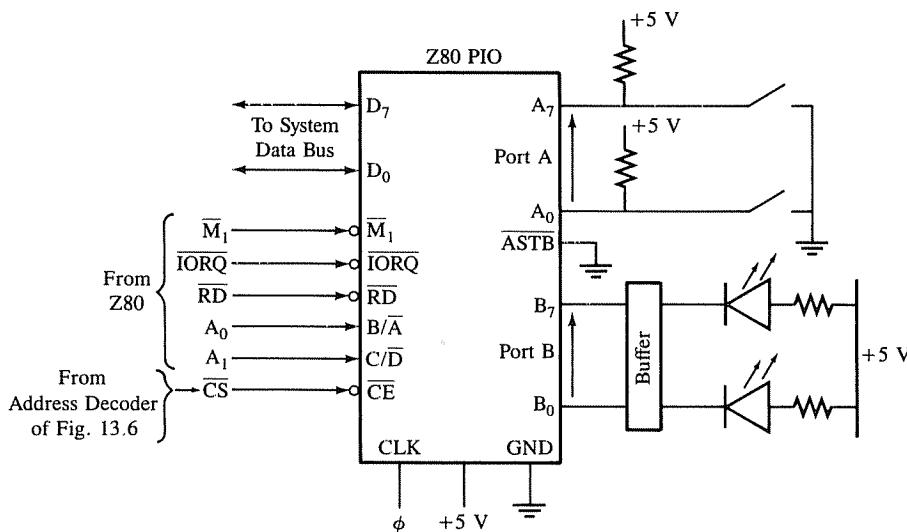
```

; The following port addresses refer to Figure 13.6
PORTA EQU 80H           ;Port A address
PORTB EQU 81H           ;Port B address
CNTRLA EQU 82H          ;Control Register A
CNTRLB EQU 83H          ;Control Register B
LD A, 01001111B         ;Control word 4FH for Port A
OUT (CNTRLA), A         ;Write in control register A
LD A, 00001111B         ;Control word 0FH for Port B
OUT (CNTRLB), A         ;Write in control register B
IN A, (PORTA)           ;Read DIP switches
OUT (PORTB), A           ;Turn on corresponding LEDs
HALT

```

**Description** Initially, all ports are defined by writing equates; these port addresses are from Figure 13.6. Then, Ports A and B are initialized by writing control words into their respective control registers. The remaining two instructions are simple I/O instructions. Note that in Figure 13.8, the handshake signal ASTB of the input Port A is grounded in order to prevent the input port from waiting for a strobe signal.

In Example 13.3, switches and LEDs are connected as simple I/O devices, similar to illustrations in Chapter 5. The only difference in writing instructions is the initialization instructions. The next section shows how to interface peripherals using the handshake signals and the interrupt I/O.



**FIGURE 13.8**  
Interfacing PIO in Mode 0 for Simple I/O

### MODES 0, 1, AND 2 WITH HANDSHAKE SIGNALS AND INTERRUPT I/O

## 13.3

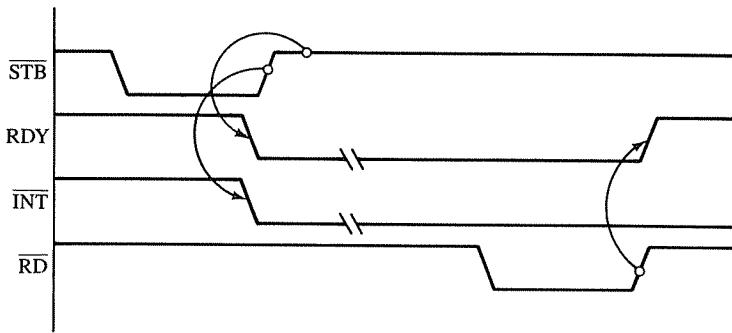
As described earlier, the PIO has two handshake signals, Strobe and Ready (STB and RDY), associated with each port and one interrupt (INT) signal. The handshake signals are used to indicate the readiness of the peripheral. The interpretation of the handshake signals is somewhat dependent on the mode being used; therefore, they will be explained separately.

The interrupt signal is used to request service from the MPU. To generate an INT signal, the interrupt flip-flop of the port being used must be enabled; each port has its interrupt enable flip-flop. (These flip-flops should not be confused with the Z80 interrupt flip-flops, IFF1 and IFF2.)

#### 13.31 Input Mode 1 and Handshake Signals

Figure 13.9 shows the sequence of events and timing when the selected port is configured as an input port and a byte is transferred from the peripheral to the PIO and then to the MPU.

1. The peripheral causes the STB (Strobe) to go low and informs the PIO that a data byte has been placed in the input register.

**FIGURE 13.9**

PIO Input Mode 1: Timing Waveforms

SOURCE: Courtesy of Zilog, Inc.

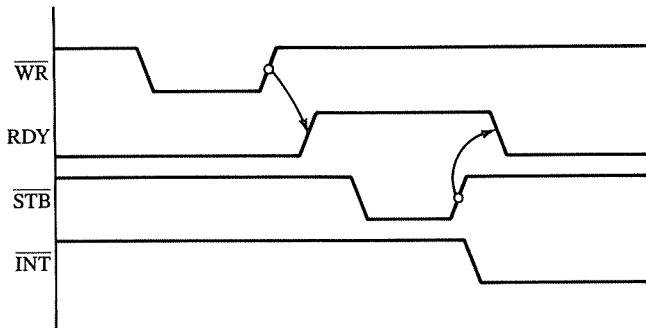
2. The rising edge of the  $\overline{STB}$  activates the interrupt ( $\overline{INT}$ ), and the other handshake signal RDY (Ready) goes inactive, indicating that the input register is full.
3. Let us assume that the Z80 and the PIO interrupt flip-flops are enabled, and that the Z80 MPU is set up in the interrupt Mode 2. When the Z80 acknowledges the interrupt request, a preprogrammed 8-bit vector is placed onto the data bus (see Section 13.33 for interrupt vector definition). This vector is combined with the byte in the interrupt register, IR, of the Z80 to form a 16-bit address, and the program is transferred to this memory address to get the address of the service routine. These interrupt activities are not shown in the timing diagram.
4. When the service routine reads the byte from the port, the RDY goes active on the rising edge of the RD signal, indicating that the PIO is ready for the next byte.

### 13.32 Output Mode 0 and Handshake Signals

Figure 13.10 shows the sequence of events and the timing when the port is configured as an output port and a data byte is transferred from the MPU to the PIO and then to the peripheral.

1. When the Z80 executes the OUT instruction, it places the byte in the PIO register and activates the RDY signal, indicating to the peripheral that a byte is available in the register.
2. The RDY signal stays high until the peripheral sends the  $\overline{STB}$  signal. The rising edge of the  $\overline{STB}$  activates the interrupt, indicating to the MPU that the byte has been received by the peripheral, and that it is ready for the next byte.
3. When the  $\overline{INT}$  is acknowledged, the program is transferred to the service routine as described in step 3 of Mode 1.

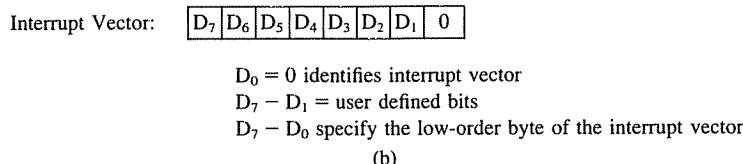
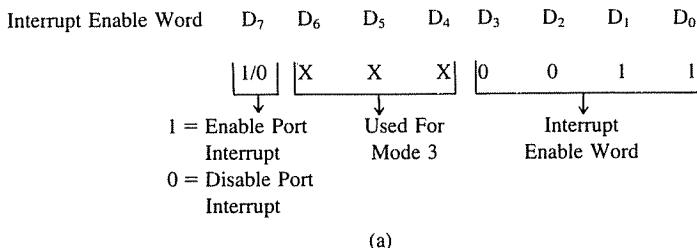
In Figure 13.10, the WR signal needs clarification because the PIO does not have a  $\overline{WR}$  pin. This signal is generated internally by the PIO when  $\overline{CE}$ ,  $C/D$ , and  $\overline{IORQ}$  are active, but  $\overline{RD}$  is inactive.



**FIGURE 13.10**  
PIO Output Mode 0: Timing Waveforms  
SOURCE: Courtesy of Zilog, Inc.

### 13.33 Interrupt Enable Word and Interrupt Vector

To set up the PIO ports in the interrupt I/O, the interrupt flip-flop of the port being used must be enabled by writing an instruction in the control register. In addition, the low-order byte must be programmed to form the interrupt vector to locate the address of the service routine. The definitions of the interrupt enable word and the interrupt vector are shown in Figure 13.11. To set up the PIO ports in the interrupt I/O, we need to write three instructions (control words) in the control register of the port being used: mode word, interrupt enable word, and interrupt vector. This is illustrated in the next example.



**FIGURE 13.11**  
Definitions of: (a) Interrupt Enable Word (b) Interrupt Vector

---

**Example  
13.4**

Assuming the same Chip Select logic as in Figure 13.6, write instructions to set up the Z80 MPU and the PIO for the interrupt I/O. Initialize Port A as an input port and Port B as an output port. The addresses of the service routines for Port A and Port B are stored in locations 2096<sub>H</sub> and 2098<sub>H</sub>, respectively.

**Solution****Initialization Instructions**

```

;Setting up Port A as input port with interrupt I/O

START: LD A, 4FH      ;Control word for Mode 1
        OUT (CNTRLA), A ;Initialize Port A as input port
        LD A, 83H      ;Interrupt enable control word
        OUT (CNTRLA), A ;Enable Port A interrupt flip-flop
        LD A, 96H      ;Low-order byte of the interrupt vector for
                         ;Port A
        OUT (CNTRLA), A ;Specify interrupt vector for Port A

;Setting up Port B as output port with interrupt I/O
LD A, OFH      ;Control word for Mode 0
OUT (CNTRLB), A ;Initialize Port B as output port
LD A, 83H      ;Interrupt enable control word
OUT (CNTRLB), A ;Enable Port B interrupt flip-flop
LD A, 98H      ;Low-order byte of the interrupt vector for
                         ;Port B
OUT (CNTRLB), A ;Specify interrupt vector for Port B

;Setting up Z80 MPU in interrupt Mode 2
LD SP, STACK   ;Initialize stack pointer
LD A, 20H      ;High-order address for interrupt vector
LD I, A        ;Initialize Z80 interrupt register
IM 2          ;Set up Z80 in interrupt Mode 2
EI            ;Enable Z80 interrupt

```

**Description** The above instructions are divided into three groups: initialization of Port A, Port B, and Z80. To initialize Port A and Port B for the interrupt I/O and in Mode 0 and 1, three control words are necessary: Mode Word, Interrupt Enable, and Interrupt Vector. These words can be executed in any sequence; however, we have used a certain sequence to clarify the concepts. Now the question is: How does the PIO differentiate these words, especially when they are written in the same control register? These words are differentiated by identifying certain bit patterns. For example, a mode word is recognized when D<sub>3</sub>–D<sub>0</sub> are all 1s, and an interrupt vector is recognized when bit D<sub>0</sub> = 0.

The instructions in the third group set up the Z80 MPU in the interrupt Mode 2, and specify the high-order byte (20<sub>H</sub>) of the interrupt vector in register I. By keeping the

instruction EI at the end, the initialization of the PIO will not be disturbed even if an interrupt were to occur in the system.

When the Z80 acknowledges an interrupt from Port A, the byte (20<sub>H</sub>) in register I is combined with the low-order byte 96<sub>H</sub>, specified as the interrupt vector for Port A, to form the 16-bit memory addresses 2096<sub>H</sub>. The program is then transferred to location 2096<sub>H</sub>. The byte stored in memory location 2096<sub>H</sub> provides the low-order address and the byte in location 97<sub>H</sub> the high-order address of the service routine for Port A. For Port B interrupt, the program is transferred to location 2098<sub>H</sub> in a similar manner.

---

### 13.34 Mode 2: Bidirectional Data Transfer

Port A of the PIO can be set up as a bidirectional input and output port. This is called Mode 2, which is the combination of Mode 0 and Mode 1. In Mode 2, Port A uses all four handshake lines; thus Port B cannot be used as an I/O port with an interrupt-generating capability. Port B must be set in Mode 3 without its logic checking capability (Mode 3 is discussed in Section 13.4). In bidirectional data transfer (Mode 2), when Port A functions as the output port, the handshake signals of Port A (ASTB and ARDY) are used, and when it functions as an input port, the handshake signals for Port B (BSTB and BRDY) are used. Similarly, the interrupt vector for the output is programmed in the control register of Port A, and the interrupt vector for the input is programmed in the control register of Port B. In this mode, the process of data transfer is as follows:

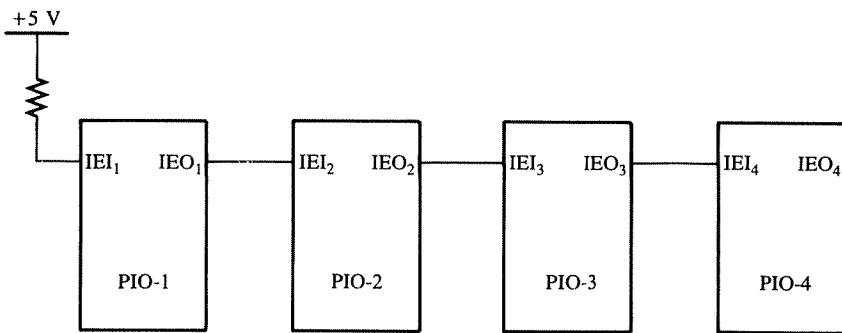
1. *Output Mode.* When the Z80 writes a byte in Port A, ARDY goes high. When the peripheral asserts ASTB and reads the byte, an interrupt is generated (if enabled) to signal the Z80 that the next byte can be sent. This is identical to Mode 0 except that a data byte is allowed onto the data bus of Port A when ASTB is active.
2. *Input Mode.* When the peripheral places a data byte on the data lines and asserts the BSTB signal, an interrupt (if enabled) is generated to indicate that the byte is placed in Port A. When the Z80 reads the byte, the BRDY goes active to indicate to the peripheral that the Z80 is ready for the next byte.

The timing and additional details of Mode 2 are further explained in Section 13.6.

### 13.35 Interrupt Priority

When multiple peripherals are interfaced with the interrupt I/O, it is essential to have a priority scheme built into the system. The PIO can be used to set the *daisy chain priority* scheme, whereby the first PIO connected to Z80 has the highest priority. Figure 13.12 shows four PIOs connected in the daisy chain format; PIO-1 has the highest priority and PIO-4 has the lowest priority. The design of the daisy chain priority scheme is based on two PIO signals IEI and IEO and the Z80 instruction RETI.

- IEI—Interrupt Enable In: This is an active high input signal to the PIO. When this signal is high, it indicates that no other PIOs of higher priority are being serviced by the



**FIGURE 13.12**  
Daisy Chain Interrupt Priority

Z80. When it is low, it indicates that a higher priority PIO is being serviced and no interrupt can be generated from this PIO.

- IEO—Interrupt Enable Out: This is an active high output signal. This remains high if IEI is high and remains low if IEI is low. When IEI is high and IEO is low, it indicates that either the PIO is being serviced or an interrupt is pending from this PIO.
- RETI—Return From Interrupt: This is a 2-byte (ED 4D) instruction, and the PIO logic can recognize it. When the PIO reads this instruction, it sets IEO high; thus indicating the end of the service routine and allowing lower priority PIOs to interrupt the processor.

To further explain the daisy chain interrupt operation, let us assume that all IEI and IEO signals in Figure 13.12 are high. When PIO-2 generates an interrupt and it is accepted, the IEO<sub>2</sub> goes low; thus causing IEI<sub>3</sub> to go low. The signal ripples through PIO-3 and PIO-4 and disables these devices. Let us assume that when PIO-2 is being serviced, it is interrupted by PIO-1, causing IEI<sub>2</sub> to go low. At the end of the service routine, PIO-1 reads the instruction RETI and sets the IEI<sub>2</sub> high; thus the service routine of PIO-2 can continue.

This daisy chain using the PIO has two drawbacks: (1) it is limited to four PIOs (it can be extended to include more PIOs by using additional logic), and (2) the priority is fixed.

## 13.4 MODE 3: BIT MODE

Mode 3 is a bit mode whereby each bit of Port A and Port B can be individually assigned input or output function. The features of this mode are as follows:

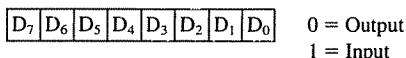
1. Each individual line of the port can be assigned either input or output function by writing a control word in the control register of the port.
2. The handshake signals are not used; Ready is kept low, and Strobe is disabled.

3. Bits are read or written into by use of the normal Read and Write functions of the I/O ports.
4. Individual bits can be masked by writing a mask word in the control register.
5. An interrupt can be generated if a predefined logical combination occurs in the input lines. The logical combination (AND/OR) can be defined by writing an interrupt control word in the control register, and the logic level can be active low or active high. For example, we can specify that bits D<sub>0</sub>–D<sub>7</sub> be inputs and two bits, D<sub>7</sub> and D<sub>6</sub>, be active low with AND function. With this specification, when the PIO reads both bits D<sub>7</sub> and D<sub>6</sub> low, an interrupt signal is generated. In OR logic function, when one of the input lines (D<sub>7</sub> or D<sub>6</sub>) is active, the interrupt is activated.

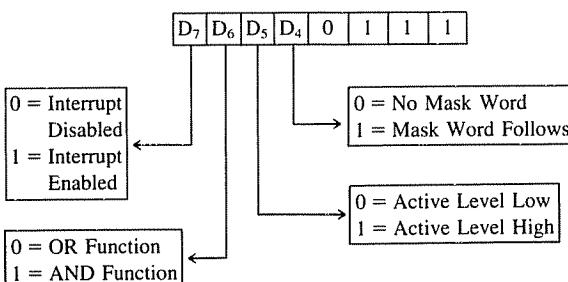
To set up the PIO in Mode 3 with the interrupt capability, four different words should be written in the control register of the port being used.

1. Mode control word for Mode 3.
2. I/O register control word to assign input or output function to individual bits.
3. Interrupt control word to define the logic conditions to generate the interrupt.
4. Mask control word to specify a mask word.

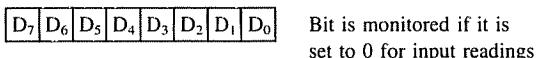
The Mode control word is already defined in Figure 13.7, and the remaining three words are defined in Figure 13.13. The use of these words is illustrated in Example 13.5.



(a) I/O Register Control Word



(b) Interrupt Control Word



(c) Mask Control Word

**FIGURE 13.13**  
Word Definitions for Mode 3

**Example  
13.5**

Port A of the Z80 PIO is set up in Mode 3 to read input switches (active low) and control the relays (Figure 13.14). Input switches are connected to bits A<sub>5</sub>, A<sub>3</sub>, and A<sub>1</sub>, and corresponding output relays are connected to bits A<sub>4</sub>, A<sub>2</sub>, and A<sub>0</sub>. Bits A<sub>7</sub> and A<sub>6</sub> are used as emergency input switches, and when both are on (active low), an interrupt is generated to turn off the entire process. Write instructions to initialize the PIO and monitor the process. Port addresses are the same as defined in Example 13.2.

**Solution**

To initialize the PIO as specified in the problem statement, we need to define the four words as follows:

Mode Word (refer to Figure 13.7)

D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>
1	1	X	X	1	1	1	1

↓  
Mode 3

I/O Register Word (refer to Figure 13.13(a)):

D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>
1	1	1	0	1	0	1	0

↓  
Emergency  
Input

↓  
Output

↓  
Output

↓  
Output

↓  
Input

↓  
Input

↓  
Input

Interrupt Control Word (refer to Figure 13.13(b))

D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>
1	1	0	1	0	1	1	1

↓  
Interrupt  
Enabled

↓  
Active  
Low

AND  
Mask  
Logic  
Follows

Mask Control Word (refer to Figure 13.13(c))

D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>
0	0	1	1	1	1	1	1

↓  
Bits to  
be monitored

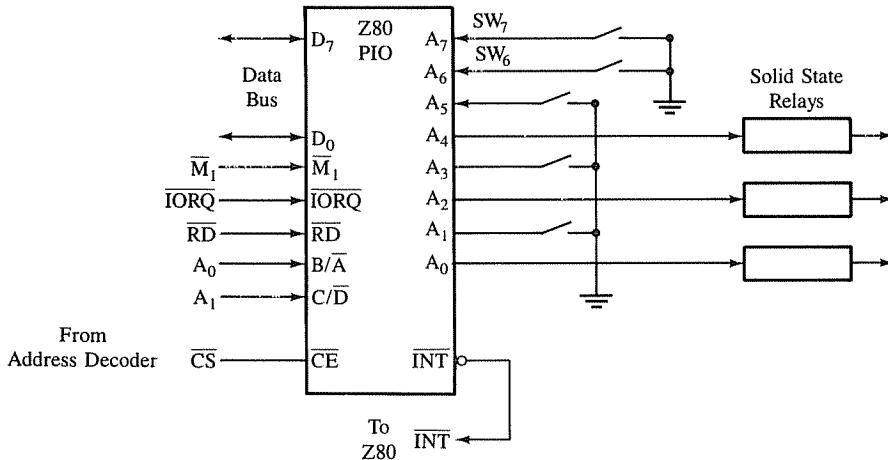


FIGURE 13.14

Interfacing PIO in Mode 3

### Instructions

;Initialization of Port A in Mode 3

START:	LD A, 0CFH	;Mode control word
	OUT (CTRLA), A	;Write in control register A
	LD A, 0EAH	;Control word to set up I/O functions
	OUT (CTRLA), A	; for bits D <sub>7</sub> -D <sub>0</sub> of port A
	LD A, 20H	;High-order address of interrupt vector
	LD I, A	;Initialize interrupt register I in Z80
	LD A, 72H	;Low-order address of interrupt vector
	OUT (CTRLA), A	
	LD A, 0D7H	;Enable interrupt with AND logic
	OUT (CTRLA), A	
	LD A, 3FH	;Mask to check D <sub>7</sub> and D <sub>6</sub>
	OUT (CTRLA), A	
READ:	IN A, (PORTA)	;Read Port A switches
	CPL	;Complement logic levels
	RRCA	;Rotate right switch readings to turn on relays
	OUT (PORTA), A	;Turn on relays
	JP READ	;Go back to monitor switches

**Program Description** The control words are already explained, and they are written in the control register of port A. The interrupt vector is specified as 2072<sub>H</sub>. The interrupt control word 1 1 0 1 0 1 1 1 (D7<sub>H</sub>) sets up the PIO to check for active low AND logic. This

is followed by the masking word 0 0 1 1 1 1 1 (3F<sub>H</sub>); this word specifies that bits D<sub>7</sub> and D<sub>6</sub> should be monitored for active low AND logic. This masking word specifies that if bits D<sub>7</sub> and D<sub>6</sub> are both logic 0, the PIO will generate an interrupt. Then the program will be transferred to location 2072<sub>H</sub> to find the address of the service routine.

After the initialization, the instructions continuously read switches of Port A. When a switch is turned on, it provides logic 0; therefore, the reading is complemented to turn on the relays. Each switch and its corresponding relay are next to each other; therefore, rotating the reading right by one position turns on appropriate relays. When the switches connected to bits D<sub>7</sub> and D<sub>6</sub> are turned on (logic 0), the PIO logic checks for that condition and generates an interrupt. Then the service routine can take an appropriate action.

## 13.5

### ILLUSTRATION: INTERFACING KEYBOARD AND SEVEN-SEGMENT DISPLAY

This illustration is concerned with interfacing a push-button keyboard and a seven-segment LED display using the PIO. The PIO is connected as a peripheral I/O as in Example 13.2, with the same port addresses. The emphasis in this illustration is not particularly on the features of PIO but on how to integrate hardware and software. When a key is pressed, the binary reading of the key has almost no relationship to what we intend to represent. Similarly, to display a number at a seven-segment LED, the binary value of the number needs to be converted into the seven-segment code, which is primarily decided by the hardware consideration. This illustration demonstrates how the microprocessor monitors the changes in hardware reading, and how we can convert the reading into appropriate binary format using the Z80 instructions.

#### 13.51 Problem Statement

A push-button keyboard is connected to Port A and a seven-segment LED is connected to Port B of the PIO, as shown in Figure 13.15. Port A should be configured in the input Mode 1 and Port B in the output Mode 0; this is a simple I/O configuration without the use of handshake signals or the interrupt.

Write a program to monitor the keyboard to sense a key pressed and display the number of the key at the seven-segment LED. For example, when the key K<sub>7</sub> is pressed, the digit 7 should be displayed at Port B.

#### 13.52 Problem Analysis

In this problem, the address decoding circuit and the port addresses are the same as in Example 13.2, and the initialization instructions for the PIO are same as in Example 13.3; therefore, these aspects of the problem will not be discussed here.

The keyboard circuit shown in Figure 13.15 is similar to that in Figure 5.6 except that the DIP switches are replaced by push-button keys and the buffer is replaced by the PIO. When a push-button key is pressed, it bounces (makes and breaks contact) a few

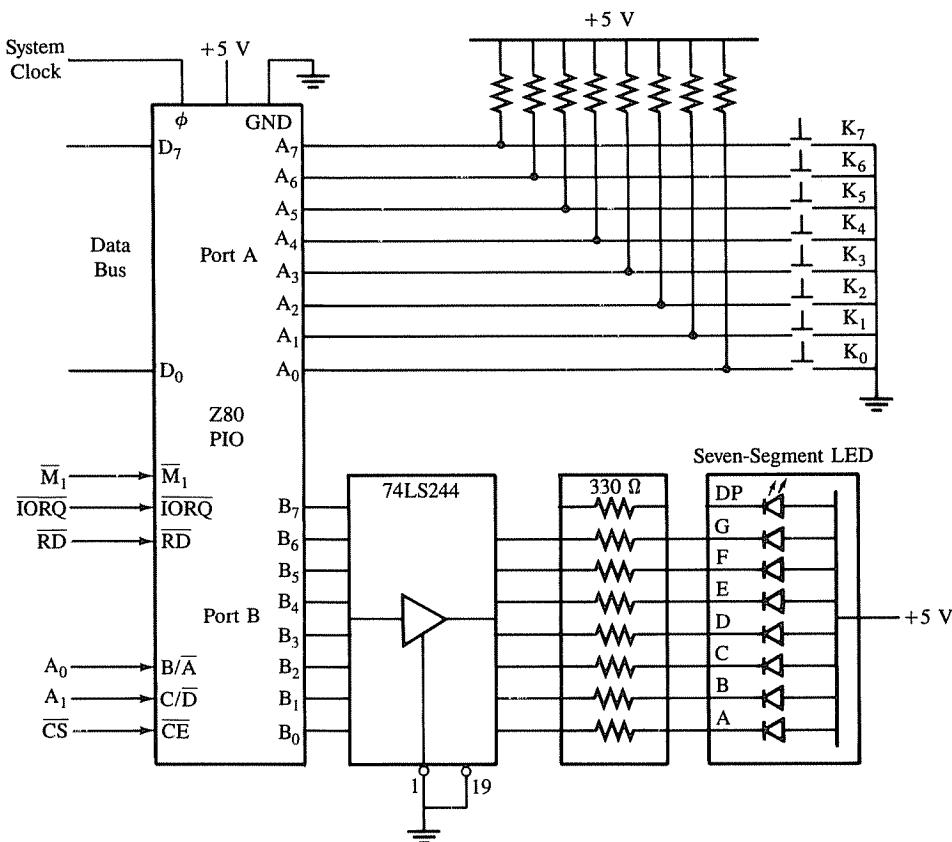


FIGURE 13.15  
Interfacing a Keyboard and a Seven-Segment LED

times before it makes a firm contact. To prevent the multiple readings of the same key, it is necessary to debounce the key. We have already discussed the hardware solution to this problem in Chapter 12. The software solution to this problem is to wait for 10–20 ms until the key is settled and then check the key again. The display circuit in Figure 13.15 uses a common anode seven-segment LED, connected to Port B of the PIO. To display a digit, it is necessary to turn on the appropriate segments of the LED. The appropriate binary code can be obtained by using the table look-up technique, described in Section 13.54. The programming of this problem can be divided into the following categories:

1. Check whether a key is pressed.
2. Debounce the key.
3. Identify and encode the key in appropriate binary format.
4. Obtain the seven-segment code and display it.

The instructions for these steps can be written in separate modules, as shown in the next section.

### 13.53 Keyboard

The keys K<sub>7</sub>–K<sub>0</sub> are tied high through 10 k resistors, and when a key is pressed, the corresponding line is grounded. When all keys are open and if the Z80 reads Port A, the reading on the data bus will be FF<sub>H</sub>. When any key is pressed, the reading will be less than FF<sub>H</sub>. For example, if K<sub>7</sub> is pressed, the output of Port A will be 0111 1111 (7F<sub>H</sub>). This reading should be encoded into the binary equivalent of the digit 7 (0 1 1 1) by using software routines. The following subroutines—KYCHK and KYCODE—accomplish the tasks of checking a key pressed and encoding the key in appropriate binary format.

;KYCHK: This subroutine first checks whether all keys are open.  
;Then, it checks for a key closure, debounces the key, and places  
;the reading in the accumulator. See Figure 13.16 for flowchart.

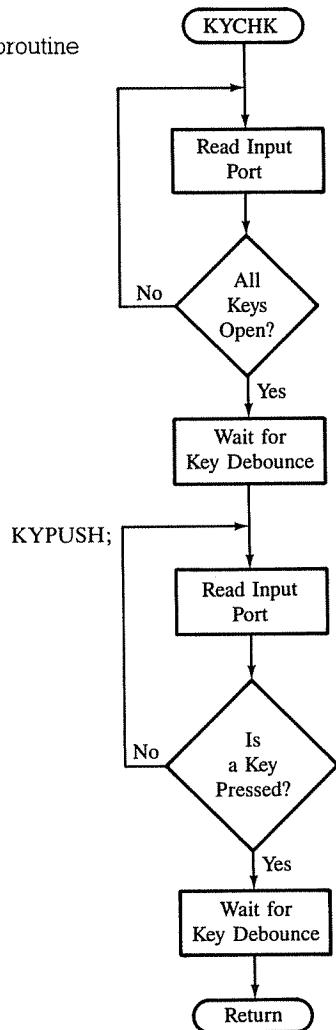
```
KYCHK: IN A, (PORTA)    ;Read keyboard
        CP 0FFH          ;Are all keys open?
        JP NZ, KYCHK     ;If not, wait in loop
        CALL DBONCE      ;If yes, wait 20 ms

KYPUSH: IN A, (PORTA)    ;Read keyboard
        CP 0FFH          ;Is key pressed?
        JP Z, KYPUSH     ;If not, wait in loop
        CALL DBONCE      ;If yes, wait 20 ms
        CPL              ;Set 1 for key closure
        OR A              ;Set 0 flag for an error
        JP Z, KYPUSH     ;It is error, check again
        RET
```

**Description** This subroutine is based on hardware; when all keys are open the keyboard reading is FF<sub>H</sub>, and when a key is pressed, the reading is less than FF<sub>H</sub>. The routine begins with the loop to check whether all keys are open, and it stays in the loop until all keys are open. This prevents a reading of the same key repeatedly if someone were to hold the key for a long time. When it finds that a key has been released, it waits for 20 ms for a key debounce.

The loop starting at KYPUSH checks whether a key is pressed. When a key is pressed, the reading is less than FF<sub>H</sub>; thus, the compare instruction does not set the Z flag and the program goes to the next instruction for a key debounce. The CPL instruction complements the accumulator reading; thus, the reading of the key pressed is set to 1, and other bits are set to 0. The next two instructions check for an error. If it is momentary contact (false alarm), all bits are 0s. The OR instruction sets the Z flag, and the Jump instruction takes the program back to checking keys.

**FIGURE 13.16**  
Flowchart: Key Check Subroutine



KYCODE: ;This routine converts (encodes) the binary hardware reading of the key ; pressed into appropriate binary format according to the number of the ; key.

LD C, 08H	;Set code counter
NEXT: DEC C	;Adjust key code
RLCA	;Place MSB in CY
JP NC, NEXT	;If bit = 0, go back to check next bit
LD A, C	;Place key code into the accumulator
RET	

**Description** Conceptually, this is an important routine; it establishes the relationship between the hardware and the number of the key. For example, if key K<sub>7</sub> is pressed, the reading from the routine KYCHK in the accumulator will be 1 0 0 0 0 0 0 0 (the reading is already complemented). The KYCODE routine sets register C for the count of eight and immediately decrements the count to seven. The instruction RLCA places bit D<sub>7</sub> into the CY flag, and the next instruction checks for the CY flag. If it is set, the key K<sub>7</sub> must be pressed, and the key code (digit 7) is in register C. If CY = 0, the program loops back to check the next bit (D<sub>6</sub>). The loop is repeated until 1 is found in CY, and at every iteration of the loop, the key code in register C is adjusted for the next key. If more than one key is pressed, this routine ignores the low-order key. Finally, the subroutine places the key code into the accumulator and returns.

```
DBONCE: ;This is a 20 ms delay routine.
        ;The delay COUNT should be calculated based on system frequency.
        ;This does not destroy any register contents.
        ;Input and Output = None
        PUSH BC      ;Save register contents
        LD BC, COUNT ;Load delay count
LOOP:   NOP          ;Add delay
        DEC C        ;Repeat until C = 0
        JR NZ, LOOP  ;Loop until B = 0
        DJNZ LOOP    ;Loop until B = 0
        POP BC       ;Restore register contents
        RET
```

**Description** This is a simple delay routine similar to the delay routines discussed in Chapter 9. The first instruction loads the BC register with a 16-bit number, and the DJNZ instruction decrements the number in B and repeats the loop until B = 0. The NOP instruction is included in the loop to increase the delay. The inner loop is repeated until C = 0. In this routine, the 16-bit number (COUNT) should be calculated based on the clock frequency of the system and the T-states in the loop (see Chapter 9 for details).

### 13.54 Seven-Segment Display

Figure 13.15 shows that a common anode seven-segment LED is connected to Port B through the driver 74LS244. The driver is necessary to increase the current capacity of Port B; each LED segment requires 15–20 mA of current.

A seven-segment LED consists of seven light-emitting diodes (A through G) and one diode (DP) for the decimal point; these LEDs are physically arranged as shown in Figure 13.17(a). A common anode LED can be logically represented as shown in Figure 13.17(b), and a common cathode LED as in Figure 13.17(c). The segments, A through G, are usually connected to data lines D<sub>0</sub> through D<sub>6</sub>, respectively. If the decimal point is being used, data line D<sub>7</sub> is connected to DP; otherwise it is left open. The binary code required to display a digit is determined by the type of the seven-segment LED (common cathode or common anode) and the connections of the data lines. For example, to display digit 7 at Port B (Figure 13.15), segments A, B, and C should be turned on, and in a

common anode LED, these segments are turned on with logic 0. Therefore, the binary code should be  $78_{\text{H}}$  as follows:

Data Lines	D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>	
Logic	X	1	1	1	1	0	0	0	= 78 <sub>H</sub>
Segments	NC	G	F	E	D	C	B	A	

The code for each Hex digit from 0 to F can be determined by examining the connections of the data lines to the segments and their logic requirements.

The driver 74LS244 (Figure 13.15) is an octal noninverting driver with tri-state output and the current sinking capacity of 24 mA. It has two active low enable lines, and the driver is permanently enabled by grounding these lines. In this circuit, the driver functions simply as a current amplifier; whatever logic is at Port B will be at the output of the driver.

To display the number of the key pressed, a routine is necessary that will send an appropriate code to Port B. The routine KYCODE supplies the binary number of the key pressed; however, there is no relationship between the binary value of a digit and its seven-segment code. Therefore, the table look-up technique (Section 10.7) will have to be used to find the code for the digit supplied by KYCODE; this is shown in the next routine, DISPLAY.

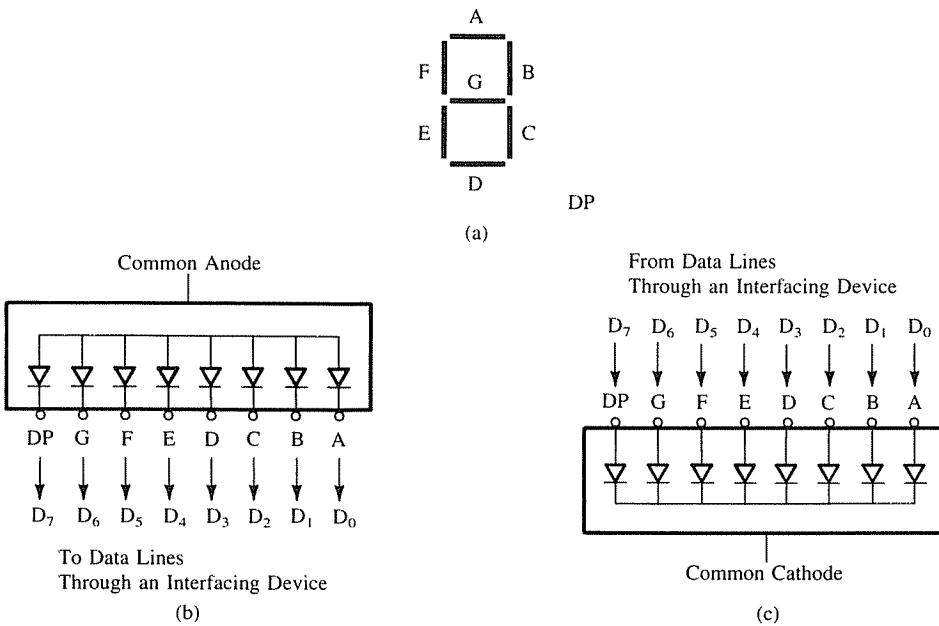


FIGURE 13.17

Seven-Segment LED: (a) LED segments (b) Common Anode LED (c) Common Cathode LED

DISPLAY: ;This routine takes the binary number and converts it into its common  
;anode seven-segment LED code. The codes are stored in memory  
;sequentially starting from the address CACODE.  
;Input: Binary number in accumulator  
;Output: None  
;Modifies contents of HL and A  
LD HL, CACODE ;Load starting address of code table in HL  
ADD A, L ;Add digit to low-order address in L  
LD L, A ;Place code address in L  
LD A, (HL) ;Get code from memory  
OUT (PORTB), A ;Send code to Port B  
RET

CACODE: ;Common anode seven-segment codes are stored sequentially in memory  
DB 40H, 79H, 24H, 30H, 19H, 12H ;Codes for digits from 0 to 5  
DB 02H, 78H, 00H, 18H, 08H, 03H ;Codes for digits from 6 to B  
DB 46H, 21H, 06H, 0EH ;Codes for digits from C to F

**Description** In this routine the HL register is used as a memory pointer to code location. The digit to be displayed is in the accumulator, supplied by the routine KYCODE, and the seven-segment code is stored sequentially in memory starting from location CACODE. The basic concept in this routine is to modify the memory pointer by adding the value of the digit to the base address and get the code location. For example, let us assume that the starting address of CACODE is  $2050_H$  and the digit 7 is in the accumulator. The code for digit 0 is in location  $2050_H$ ; consequently, the code for digit 7 is in location  $2057_H$ . Therefore, to display digit 7, the routine adds the contents of the accumulator (7) to the low-order byte  $50_H$  in register L, resulting in the sum  $57_H$ . Thus, the memory pointer in HL is modified to  $2057_H$ , and the code for digit 7 is obtained by using this memory pointer.

### 13.55 Program

To monitor the keyboard and display the key pressed, we need to initialize the PIO ports and combine the software modules discussed previously.

KYBORD: ;This program first initializes the PIO ports; Port A in Mode 1 and  
; Port B in Mode 0 and then, calls the subroutine modules discussed  
; previously to monitor the keyboard.

PORATA	EQU 80H	;Port A address
PORTB	EQU 81H	;Port B address
CNTRLA	EQU 82H	;Control register A
CNTRLB	EQU 83H	;Control register B
WORDA	EQU 4FH	;Mode 1 control word
WORDB	EQU 0FH	;Mode 0 control word
STACK	EQU 20A7H	;Beginning stack address

PIO:	LD SP, STACK LD A, WORDA OUT (CNTRLA), A LD A, WORDB OUT (CNTRLB), A	;Set up Port A in Mode 1  ;Set up Port B in Mode 0
NEXTKY:	CALL KYCHK CALL KYCODE CALL DSPLAY JP NEXTKY	;Check if a key is pressed ;Encode the key ;Display key pressed ;Check the next key pressed

**Description** This is the main program, which involves the initialization of the PIO and the stack pointer. The port addresses defined here are from Example 13.3 (Figure 13.8), and the address of STACK (stack pointer initialization) is shown as an illustration; it has no specific significance. Because the problem is divided into small modules, the main program consists primarily of calling these modules.

### 13.56 Comments and Alternative Approaches

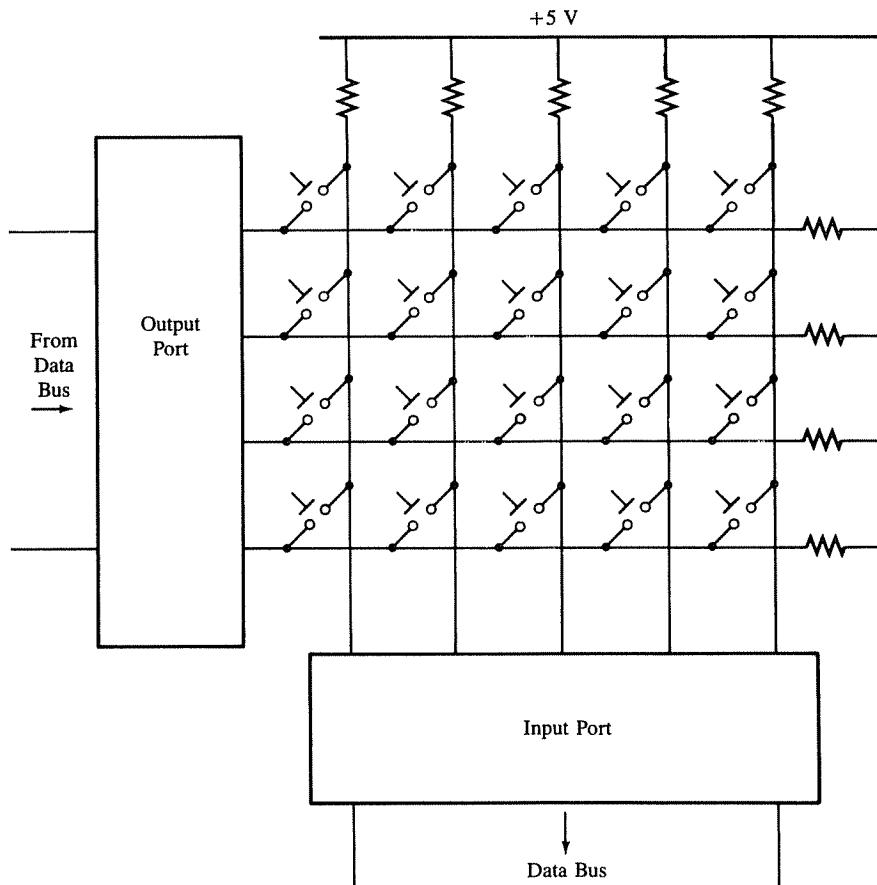
The interfacing of the push-button keyboard and seven-segment display is a simplified illustration of industrial applications. The illustration is deliberately kept simple to emphasize the conceptual framework between hardware and software. However, as an application, it has several limitations.

1. This method of connecting the keyboard limits the number of keys in proportion to the number of I/O ports; only eight keys can be connected to an 8-bit port. Generally, keys are connected in a matrix format (Section 13.57). For example, in the matrix format, 16 keys can be connected to one 8-bit port or 64 keys can be connected to two 8-bit ports.
2. This method of connecting a seven-segment LED needs excessive hardware, one port and a driver per seven-segment LED. Furthermore, it consumes large current (100–150 mA per display). To minimize hardware and power consumption, the technique of multiplexing is generally used (Section 13.58).

In this illustration, the primary emphasis is on software. For example, in the keyboard, the debouncing and encoding is performed by using instructions. However, interfacing chips that can sense a key closure, debounce the key, and encode the key are currently available commercially. These chips can also generate an interrupt signal when a key is pressed. Similarly, in the seven-segment display, the table look-up can be replaced by a decoder/driver. However, the hardware approach increases unit price. On the other hand, the software approach involves considerable labor (programming and debugging) cost. The choice is generally determined by the production volume and the total unit price.

### 13.57 Matrix Keyboard

In a matrix keyboard, keys are arranged in a matrix form, as shown in Figure 13.18. It has 20 keys, arranged in four rows and five columns. When a key is pressed, it shorts one row



**FIGURE 13.18**  
Interfacing a Matrix Keyboard

and one column; otherwise, the row and column do not have any contact. This keyboard requires nine data lines instead of the 20 required if the keys are connected as in Figure 13.15.

The interfacing of a matrix keyboard requires two I/O ports, one output port and one input port. Rows are connected to the output port, and the columns are connected to the input port. To sense a key closure, we can use either the software approach or the hardware approach.

A software technique called matrix scan is used to sense a key closure. In this technique, rows are grounded by sending 0s to all the rows through the output port, and a key closure is checked by reading the data on columns through the input port. If no key is pressed, all bits of the input reading are high, and if a key is pressed, one of the bits will be 0. Then the program grounds one row at a time, locates the key pressed, and encodes the

key. The basic concepts in interfacing a matrix keyboard are similar to those discussed in the above illustration, except that the software is somewhat complex. An illustration of interfacing a matrix keyboard is discussed in Chapter 17.

The hardware approach is to use a commercially available chip, as shown in Figure 13.19 (National Semiconductor MM54/74C923); this is a key encoder for a 20-key matrix. Figure 13.19 shows the block diagram of the key encoder including the connections to a keyboard. In this circuit, when a key is pressed, the internal circuit senses a key closure, debounces and encodes the key, and generates an interrupt to inform the MPU that a key has been pressed. For example, when the "A" key is pressed, the output of the key encoder will be 0 1 0 1 0. The task for software is to read the code.

### 13.58 Multiplexing and Scanned Display

The display technique in the above illustration is quite limited. It needs one I/O port and a driver for one seven-segment LED; this technique can be quite costly for multiple-digit display. The number of hardware chips needed for multiple-digit display can be minimized by using the technique called multiplexing, whereby the data lines are time-shared by various seven-segment LEDs.

Figure 13.20 shows a block diagram for a multiplexed display. The diagram has two output ports: one port  $P_A$  to drive LED segments, and a second port  $P_B$  to turn on the corresponding cathodes. The output lines of port  $P_A$  are connected to seven segments of each LED, and the output lines of port  $P_B$  are connected to the cathodes of each LED. To display a digit, the code is sent to the segments through port  $P_A$ , and a LED is turned on by sending a bit to the appropriate cathode through port  $P_B$ . To display a four-digit number,

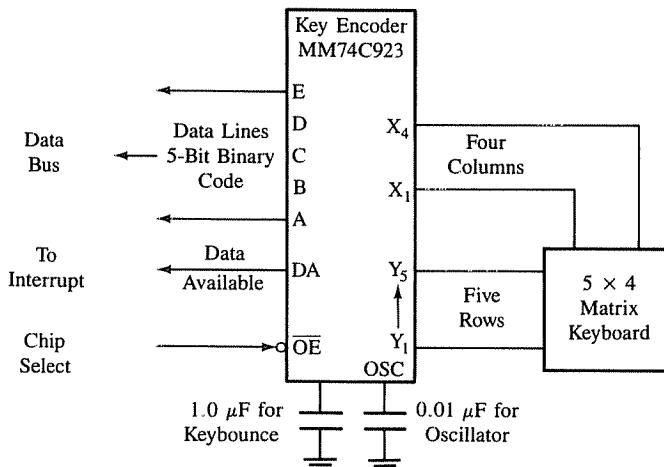
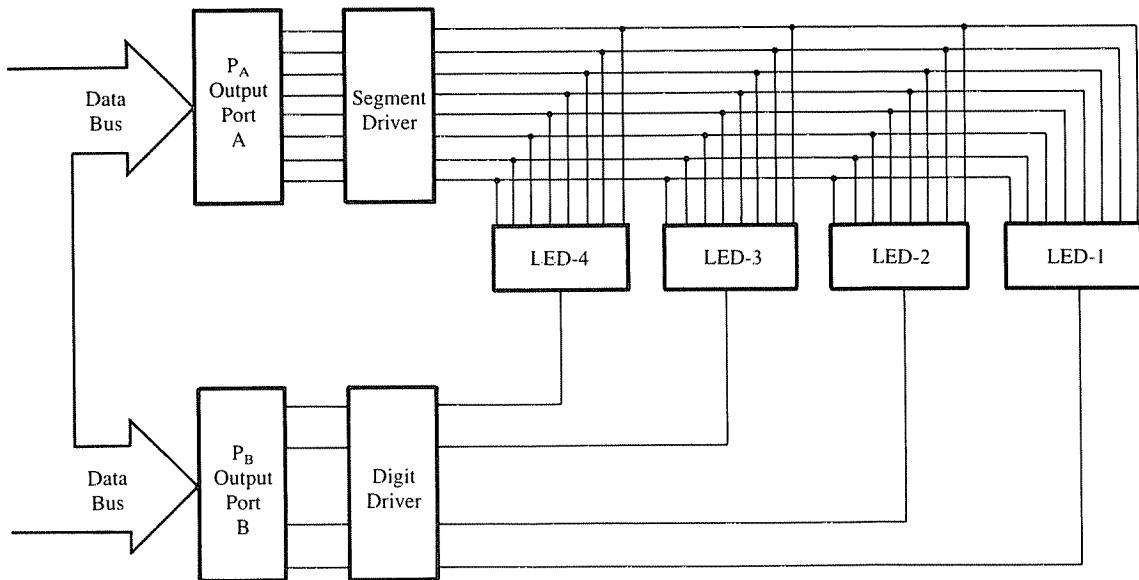


FIGURE 13.19

Logic Diagram of MM74C923 20-Key Encoder Interfacing a Matrix Keyboard



**FIGURE 13.20**  
Block Diagram of Multiplexed Output Display

each seven-segment LED is turned on and off sequentially. For example, to display 1987, the code for digit 7 is sent first and LED-1 is turned on. Next, the code for digit 8 is placed on the segment data lines; simultaneously, LED-1 is turned off and LED-2 turned on. The cycle is repeated fast enough that the display appears stable. This multiplexing technique reduces the power consumption and the number of chips.

Two ports shown in Figure 13.20 are not capable of driving eight seven-segment LEDs. In a common cathode seven-segment LED, all segments are driven by the output lines, which should supply at least 10–15 mA of current to each segment. The cathode should sink seven or eight times that current. The I/O ports of the PIO are limited in current capacity; therefore, additional transistors or ICs, called segment and digit drivers, are required, as shown in Figure 13.20. An illustration of multiplexed scanned display is shown in Chapter 17. Another approach is to replace software by hardware, as shown in Figure 13.21. It has two types of displays; Port A is connected to a HP 5082/7340 and Port B has seven-segment LEDs with a Hex decoder/driver. The HP 5082/7340 display has an internal decoder/driver; thus, two digits per port can be displayed. Port B has a separate decoder/driver; however, both ports are functionally similar. In this approach, the task of the software instructions is reduced simply to outputting the byte to be displayed to the port. For example, to display  $87_H$ , the Z80 needs only to output  $87_H$  to that port. Replacing software with hardware can increase the unit price of a display; for example, in Figure 13.21 we can connect only four LEDs. In addition, this technique increases the power consumption.

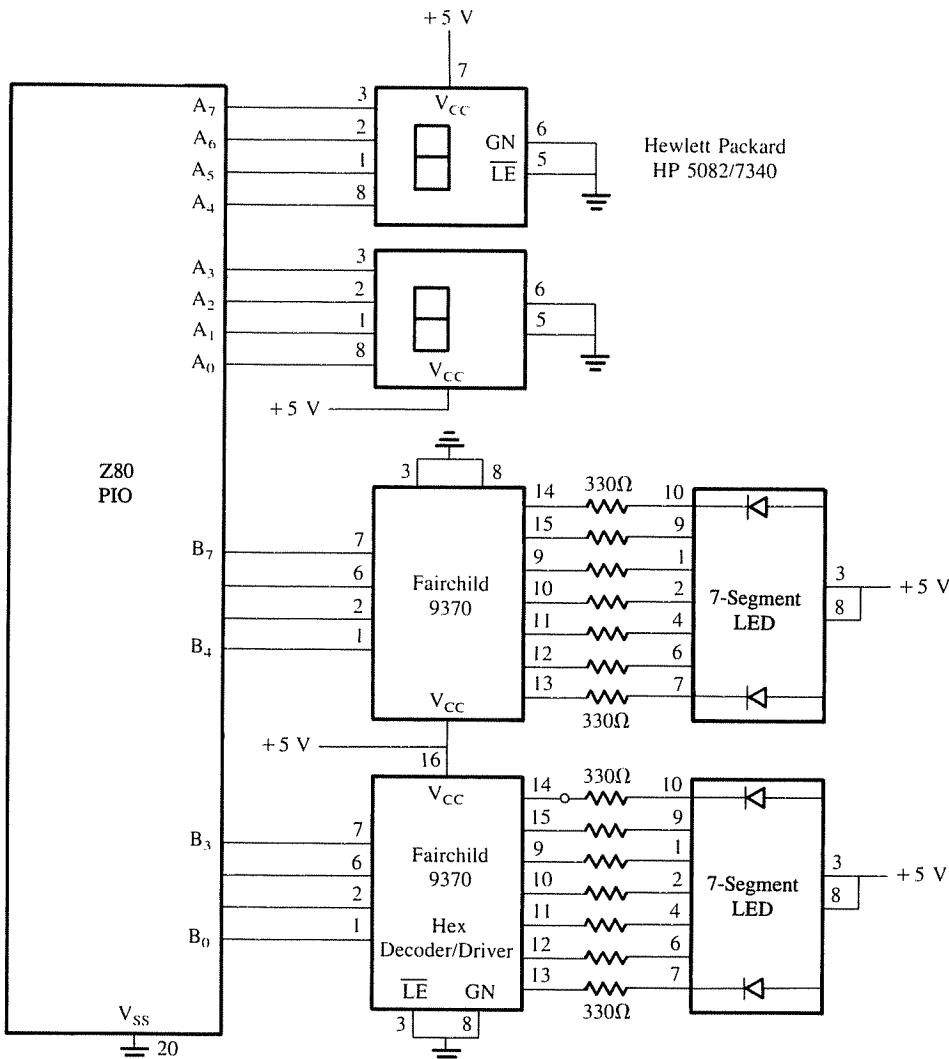


FIGURE 13.21

Hardware Alternatives to Multiplexed Display

---

**ILLUSTRATION: BIDIRECTIONAL DATA TRANSFER BETWEEN  
TWO MICROCOMPUTERS USING PIO IN MODE 2**

13.6

The bidirectional data transfer is a common occurrence in the computer world. Typical examples include data transfer between two microcomputers or between a floppy disk and

a microcomputer. The bidirectional communication between two microcomputers can be accomplished using the PIO in Mode 2.

### 13.61 Problem Statement

Design an interfacing circuit to set up bidirectional data communication between two Z80 microcomputers: Micro-1 and Micro-2. Use the PIO in Mode 2 as the interfacing device with Micro-1, and the interrupt technique for data transfer. Set up Micro-2 using a tri-state buffer as the interfacing device and implement data transfer under program control (status check). Write necessary software to transfer a block of data from Micro-1 to Micro-2.

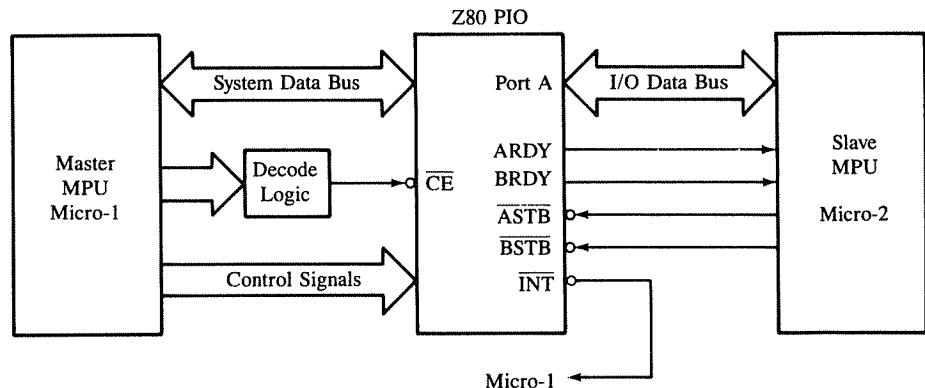
### 13.62 Problem Analysis

Figure 13.22 shows a block diagram to set up the bidirectional communication between two microcomputers. The block diagram shows two bidirectional data buses—system data bus and I/O data bus—interconnected through the PIO, which serves as the interfacing device of Micro-1. Port A of the PIO is used for bidirectional data transfer. Micro-1 uses the handshake signals of Port A (ASTB and ARDY) for output control, and the handshake signals of Port B (BSTB and BRDY) for input control. The communication process is the combination of Mode 0 and Mode 1.

Both microcomputers require I/O ports to read and write data and to check the status of handshake signals. Therefore, it is necessary to analyze carefully these I/O functions between the MPUs. Data transfer for Micro-2 is to be accomplished under program control with the status check and through the interrupt process for Micro-1. The steps in the data transfer operations between the two MPUs and the timing are as follows.

#### Data Output from Micro-1 to Micro-2.

1. The Micro-1 writes data into Port A and causes the signal ARDY to go high, indicating to Micro-2 that a byte is available to be read (Figure 13.23). This is an output function for Micro-1.



**FIGURE 13.22**

Block Diagram: Bidirectional Communication between Two Microcomputers Using PIO

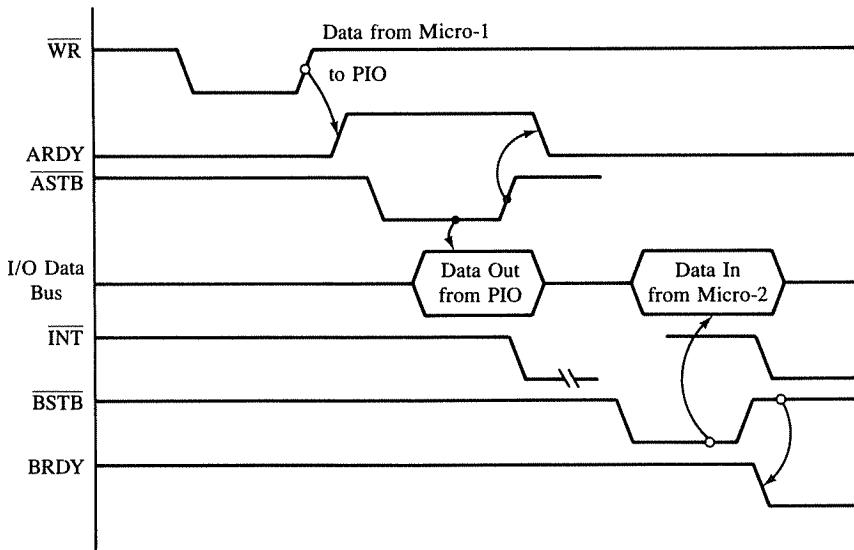


FIGURE 13.23

Timing Waveforms for Bidirectional Data Transfer

2. The Micro-2 continues to check ARDY, and when it goes high, it asserts the ASTB signal low. This places the data byte onto the data bus (Figure 13.23). For Micro-2, checking ARDY is an input function.
3. Once the data byte is on the bus, it can be latched or read by Micro-2 on the rising edge of the ASTB signal. Therefore, the asserting of ASTB low and reading the byte can be performed by a Read operation of Micro-2. Thus, asserting the ASTB is an input operation of Micro-2.
4. When the byte is read by the Micro-2, ARDY goes low and the INT is generated to indicate to Micro-1 that a next byte can be sent.

#### Data Input to Micro-1 from Micro-2

1. Micro-2 places a data byte onto the I/O data bus as it asserts the BSTB signal. This is an output function for Micro-2.
2. On the rising edge of the BSTB, the INT is generated to inform Micro-1 that a byte is available in Port A to be read. Similarly, when BSTB goes high, it causes BRDY to go low, indicating to Micro-2 to wait until a byte is read, so Micro-2 continues to read BRDY. This is an input function for Micro-2.
3. When an interrupt is generated, Micro-1 reads the byte, and the RD signal causes the BRDY to go high, indicating to Micro-2 that next byte can be sent.

This analysis leads to certain hardware requirements, which are discussed in the next section.

### 13.63 Hardware Description

To clarify the hardware requirements, we can summarize the bidirectional data transfer operations as follows.

**1. To send a byte from Micro-1 and to receive it in Micro-2:**

- Micro-1 writes a byte into Port A of the PIO whenever an interrupt is generated.
- Micro-2 reads ARDY until it goes high, and then reads the byte by asserting the ASTB.

**2. To receive a data byte by Micro-1:**

- Micro-1 performs an input operation whenever an interrupt is generated.
- Micro-2 performs two operations: one input operation to monitor BRDY and one output operation to write a data byte and assert the BSTB.

Thus, for Micro-1, Port A of the PIO needs to be set up in the bidirectional mode, and the Z80 in the interrupt Mode 2. Micro-2 needs one input port to monitor ARDY and BRDY, another input port to read a byte and assert ASTB, and one output port to send a byte and assert BSTB.

Figure 13.24 shows the complete schematic of the necessary ports and their decoding logic. The decoding logic for Port A of the PIO is the same as in Example 13.2; thus, port addresses range from  $80_H$  to  $83_H$ . All the handshake signals are being used for bidirectional data transfer by Port A; the INT signal of the PIO is connected to the INT signal of the Z80 MPU of Micro-1, and Port B is not being used in this illustration.

The two handshake signals—ARDY and BRDY—are tied, respectively, to bits D<sub>7</sub> and D<sub>0</sub> of the I/O data bus through a tri-state buffer so that they can be monitored by Micro-2; this port is labeled as input port STATUS. The signal ASTB is asserted by reading PORTIN, and the signal BSTB is asserted by writing in PRTOUT. The decode logic for these ports is generated by using the 74LS138 (3-to-8) decoder. Assuming the “don’t care” address lines (A<sub>4</sub> and A<sub>3</sub>) are at logic 0, the port addresses are as follows:

	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	
Status Port:	0	1	0	X	X	1	1	1	= $47_H \rightarrow$ STATUS
Input Port:						1	0	1	= $45_H \rightarrow$ PORTIN
Output Port:						0	0	0	= $40_H \rightarrow$ PRTOUT

Two output lines of the decoder are combined with the  $\overline{IORD}$  control signal of Micro-2 to generate two input device select pulses ( $45_H$  and  $47_H$ ). Port  $47_H$  is used to read status on the data lines D<sub>7</sub> and D<sub>0</sub>, and Port  $45_H$  is used to assert the ASTB signal. The decoder line with the address  $40_H$  is combined with the  $\overline{IOWR}$  signal of Micro-2 to generate the BSTB signal.

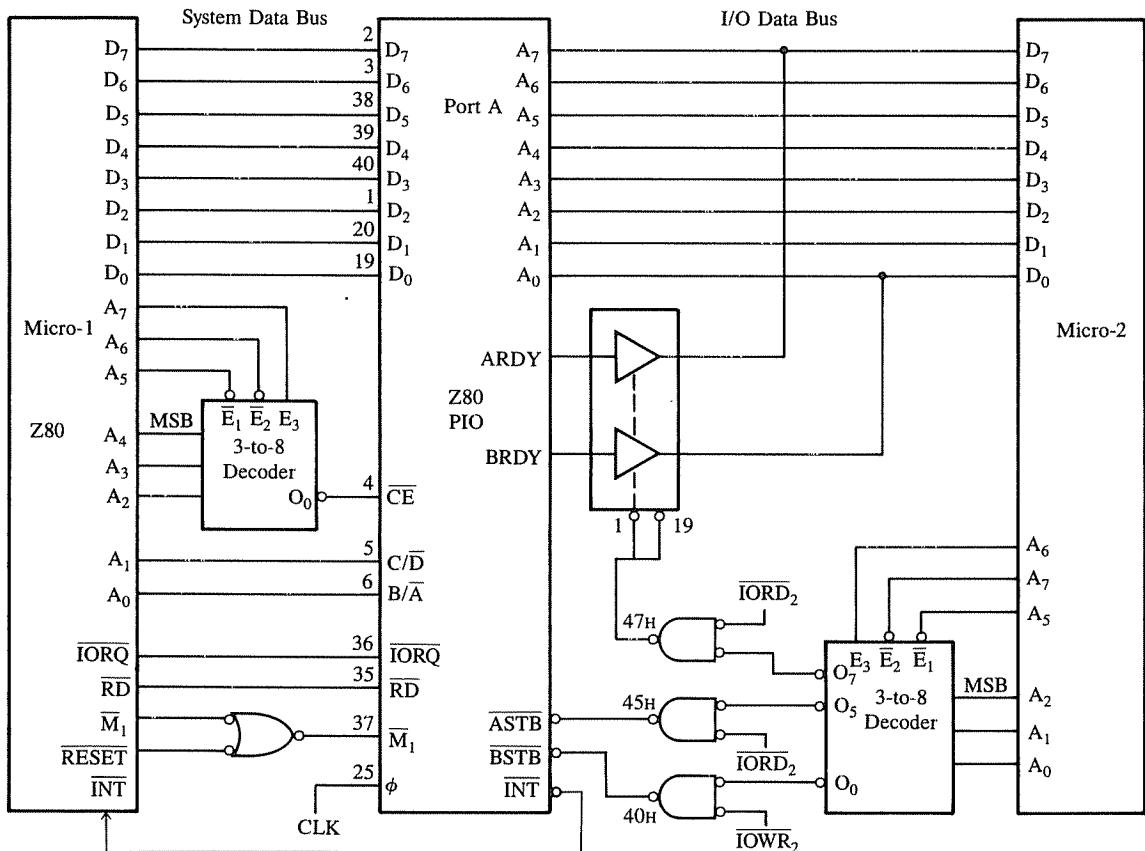


FIGURE 13.24  
Interfacing Schematic for Bidirectional Data Transfer between Two Microcomputers

### 13.64 Program

This illustration requires two programs: one for Micro-1 to send a block of data bytes and another for Micro-2 to receive those data bytes. These programs will be similar, but will vary in details because Micro-1 uses the PIO with the interrupt I/O while Micro-2 transfers data under program control. To transfer data from Micro-2 to Micro-1 is not included in the problem statement; it is left as an assignment.

#### Micro-1 Program

```
;This program initializes Z80 in interrupt Mode 2 and the PIO in the
; bidirectional data transfer Mode 2 and sends out a block of data bytes to Micro-2.
;The interrupt vector for Port A is 2090H and for Port B is 2092H.
```

```

MICRO1: LD A, 1000111B ;Control word (8FH) for Mode 2
        OUT (CNTRLA), A ;Set up Port A for bidirectional data transfer
        LD A, 10000011B ;Interrupt enable word (83H)—Figure 13.11(a)
        OUT (CNTRLA), A ;Enable interrupt for Port A
        OUT (CNTRLB), A ;Enable interrupt for Port B
        LD A, 90H         ;Interrupt vector for Port A
        OUT (CNTRLA), A ;Write interrupt vector to Port A
        LD A, 92H         ;Interrupt vector for Port B
        OUT (CNTRLB), A ;Write interrupt vector to Port B
        LD SP, STACK1    ;Initialize stack for Z80 of Micro-1
        IM 2             ;Set up Z80 in interrupt Mode 2
        LD HL, DATA      ;Set up HL as memory pointer where data
                           ; bytes are located
        LD B, BYTES      ;Load number for bytes to be transferred
        CALL SEND        ;Send first to get things started
LOOP:   JP LOOP          ;Wait for an interrupt
SEND:   ;This is a service routine for Port A to send data to Micro-2.
        ;The routine outputs one byte at a time to PIO Port A until all
        ; bytes are transmitted.
        LD A, (HL)        ;Get byte from memory
        OUT (PORTA), A   ;Send byte to Port A of PIO
        INC HL            ;Next byte location
        DEC B             ;Decrement byte counter
        EI                ;Enable Z80 interrupt
        RET NZ            ;Return if byte counter ≠ 0
        JP END            ;Jump to End message subroutine

```

#### Micro- 2 Program (See Figure 13.25)

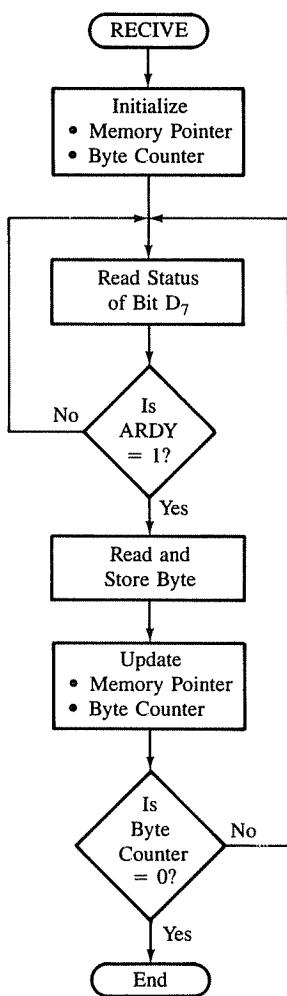
;This program receives the block of data from Micro-1 under program control.  
;It checks the status of ARDY. When ARDY is high, it reads PORTIN.

```

RECEIVE: LD SP, STACK 2 ;Initialize stack pointer for Micro-2
         LD HL, STORE  ;Point index to first memory location where data
                           ; bytes should be stored
         LD B, BYTES    ;Specify number of bytes to be received
         IN A, (PORTIN) ;This is dummy read to generate interrupt for PIO
ARDY:   IN A, (STATUS) ;Check ARDY
         RLA            ;Place bit D7 into Carry
         JP NC, ARDY   ;If ARDY is low, wait in loop
         IN A, (PORTIN) ;Read data
         LD (HL), A    ;Store data byte
         INC HL        ;Next memory location
         DJNZ NZ, ARDY ;Go back to read next byte if all bytes are not
                           ; yet received
HALT

```

**FIGURE 13.25**  
Flowchart: Program to Receive  
Data Bytes by Micro-2



### PROGRAM DESCRIPTION

1. To transfer a block of data from Micro-1 to Micro-2, both programs need to be executed at the same time.
2. The program for Micro-1 waits in a loop after the initialization; ordinarily, in real application, the program would continue to perform other tasks until an interrupt is generated.
3. The program RECEIVE for Micro-2 performs a dummy read operation to start the data transfer and continues to check ARDY. When Micro-2 reads PORTIN the first time, the ASTB signal goes low and an interrupt is generated for Micro-1.

4. In Micro-1, because of the interrupt the program is transferred to the service routine. It writes a byte in the PIO, causing ARDY to go high; decrements the byte counter; and if the counter is not zero, returns to the main program to wait in the loop.
5. When ARDY goes high, bit D<sub>7</sub> goes high because the ARDY is tied to data line D<sub>7</sub>. The program RECIVE reads the byte; this causes ASTB to go low, and an interrupt is generated for the next byte.
6. Every time a byte is transferred by Micro-1 and received by Micro-2, the respective counters (registers B) are decremented, and the data transfer continues until the counters go to 0. In Micro-1, when the byte counter is 0, the program control jumps to an End message routine. It is expected that the message routine will include the instructions EI and RET.
7. The programs given can transfer a block of data from Micro-1 to Micro-2, but not vice versa. To transfer a block of data from Micro-2 to Micro-1, additional routines are necessary (see Assignment 24). The service routine for Micro-1 will involve reading PORTA whenever an interrupt is generated. For Micro-2, an additional set of instructions is necessary. These instructions will monitor the BRDY signal. When the BRDY goes high, it means PORTA is ready to receive a byte. Then Micro-2 can write a byte causing BSTB to go low; an interrupt will be generated and Micro-1 can read the byte.

## 13.7

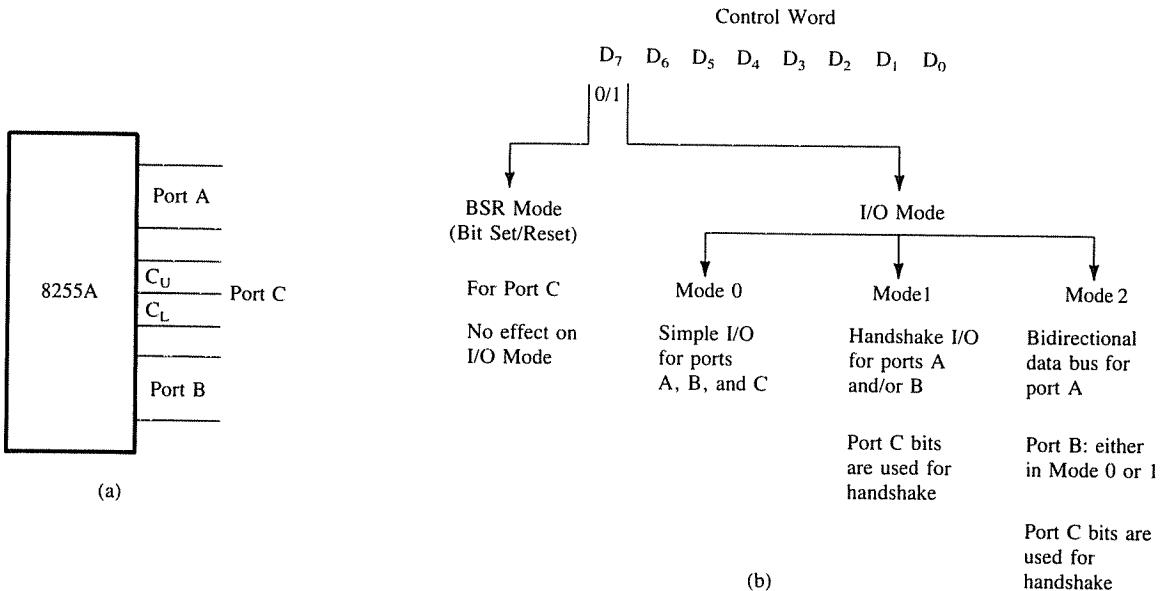
### THE 8255A PROGRAMMABLE PERIPHERAL INTERFACE

---

The Intel 8255A is another widely used, programmable, parallel I/O device, similar to the Z80 PIO. It is the revised version of Intel's 8255 and is commonly referred to as the 8255 rather than the 8255A. It can be programmed to transfer data under various conditions—from simple I/O to interrupt I/O. It is flexible, versatile, and economical, but somewhat complex. It is a general purpose I/O device and can be used with almost any microprocessor. Because of its wide use in industry it is discussed here briefly. (See Gaonkar, *Microprocessor Architecture*, 1984, for a full description.)

The 8255A has 24 I/O pins, and they can be grouped into two 8-bit parallel ports, A and B, and an 8-bit port C. The eight-bits of port C can be used as individual bits or grouped into two 4-bit ports: C<sub>U</sub> and C<sub>L</sub> (Figure 13.26(a)). Ports A and B of the 8255A are similar to ports A and B of the Z80 PIO, and port C is similar to the bit mode of the PIO. The functions of these ports are defined by writing a control word in the control register.

Figure 13.26(b) shows all the functions of the 8255A. They are classified according to two modes: the Bit Set/Reset (BSR) mode and the I/O mode (byte mode). The **BSR mode** is used to set or reset the bits in port C. The I/O mode is further divided into three modes: Mode 0, Mode 1, and Mode 2. In Mode 0, all ports function as simple I/O ports. Mode 1 is a **handshake mode**, whereby Ports A and/or B use bits from Port C as handshake signals. In the handshake mode, two types of I/O data transfer can be implemented:

**FIGURE 13.26**

(a) 8255A I/O Ports and (b) Their Modes

status check under program control and interrupt. In Mode 2, port A can be set up for bidirectional data transfer using handshake signals from port C, and port B can be set up in either Mode 0 or Mode 1. The definitions of Mode 0 and Mode 1 in the 8255A are quite different from those in the PIO and should not be confused.

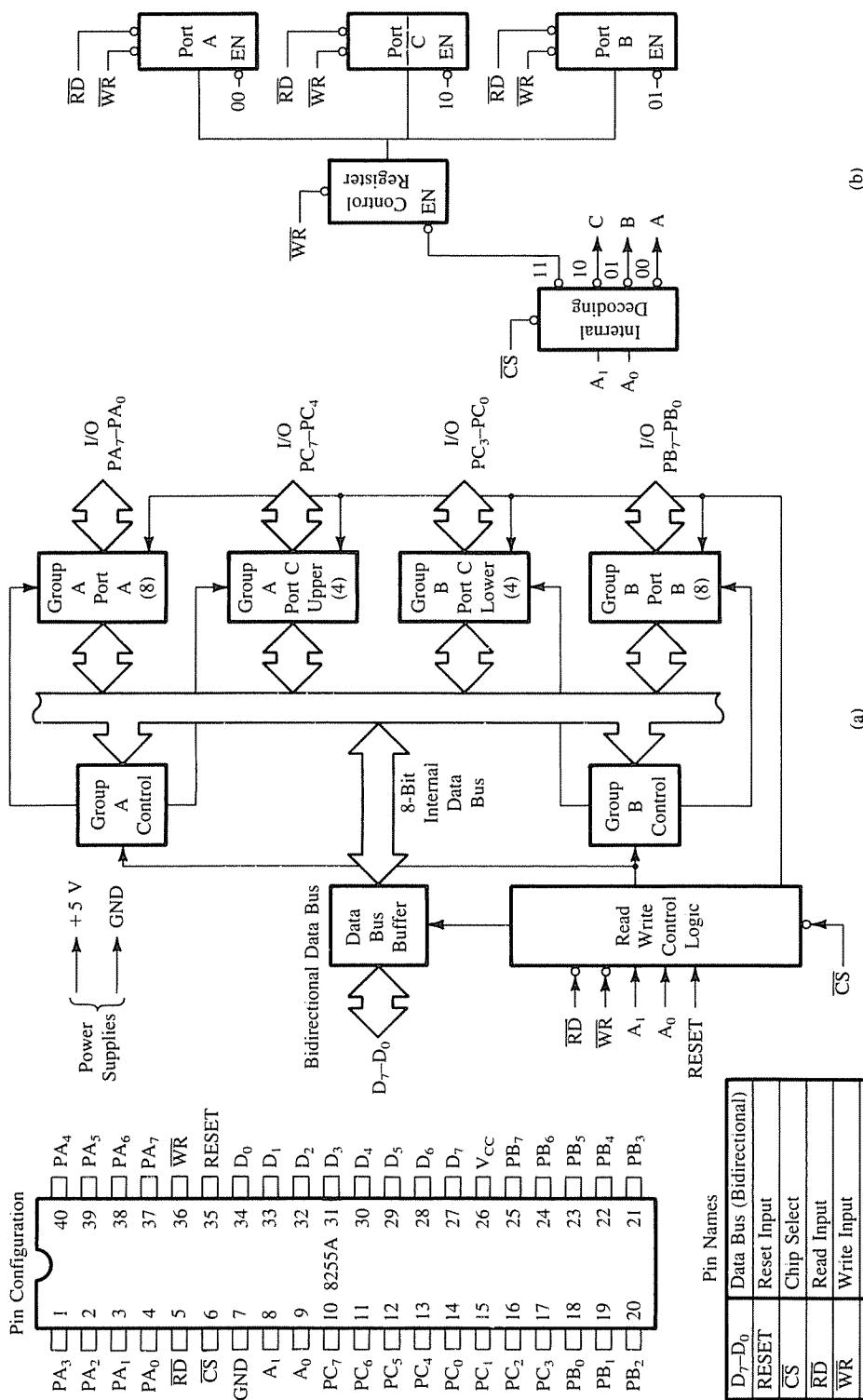
### 13.71 Block Diagram of the 8255A

The block diagram in Figure 13.27(a) shows two 8-bit ports (A and B), two 4-bit ports ( $C_U$  and  $C_L$ ), the data bus buffer, and control logic. Figure 13.27(b) shows a simplified but expanded version of the internal structure, which includes a control register. This block diagram includes all the elements of a programmable device; Port C performs functions similar to that of the status register.

#### CONTROL LOGIC

The control section has six lines. Their functions and connections are as follows:

- RD—Read: This control signal enables the Read operation. When the signal is low, the MPU reads data from a selected I/O port of the 8255A.
- WR—Write: This control signal enables the Write operation. When the signal goes low, the MPU writes into a selected I/O port or the control register.
- RESET—Reset: This is an active high signal and clears all the registers of the 8255A.

**FIGURE 13.27**

(a) 8255A Block Diagram (b) Expanded Version of the Control Logic and I/O Ports  
SOURCE: (a) Reprinted by permission of Intel Corporation, copyright 1979.

**(b)**

- $\overline{\text{CS}}$ ,  $A_0$ , and  $A_1$ —Chip Select Signals: These signals are used for selecting the device.  $\overline{\text{CS}}$  is connected to a decoded address, and  $A_0$  and  $A_1$  are generally connected to the system address lines  $A_0$  and  $A_1$ , respectively.

The  $\overline{\text{CS}}$  signal is the master Chip Select, and  $A_0$  and  $A_1$  specify one of the I/O ports or the control register as shown.

$\overline{\text{CS}}$	$A_1$	$A_0$	Selected
0	0	0	Port A
0	0	1	Port B
0	1	0	Port C
0	1	1	Control Register
1	X	X	8255 is not selected.

As an example, the port addresses in Figure 13.28 are determined by the  $\overline{\text{CS}}$ ,  $A_0$ , and  $A_1$  lines. The  $\overline{\text{CS}}$  line goes low when  $A_7=1$  and  $A_6$  through  $A_2$  are at logic 0. Combining these signals with  $A_0$  and  $A_1$  yields port addresses ranging from  $80_{\text{H}}$  to  $83_{\text{H}}$ , as shown in Figure 13.28(b).

### CONTROL WORD

The 8255A has one control register, and the contents of this register, called the control word, specify an I/O function for each port. This register can be accessed to write a control word when  $A_0$  and  $A_1$  are at logic 1, as mentioned previously. The register is not accessible for a Read operation.

Bit  $D_7$  of the control register specifies either the I/O function or the Bit Set/Reset function, as shown in Figure 13.26(b). If bit  $D_7 = 1$ , bits  $D_0$ – $D_6$  determine I/O functions in various modes, as shown in Figure 13.29. If bit  $D_7 = 0$ , port C operates in the Bit Set/Reset (BSR) mode. The BSR control word does not affect the functions of ports A and B; the BSR mode will be described later.

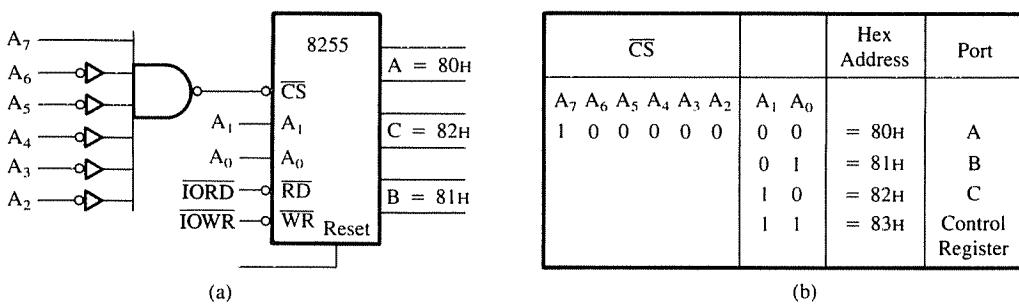
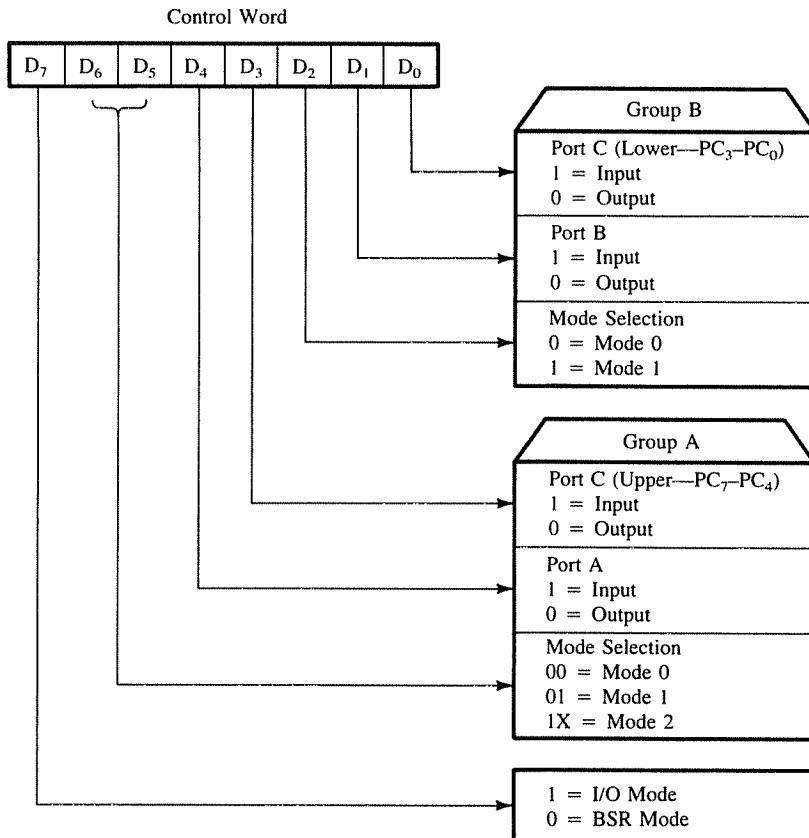


FIGURE 13.28

(a) 8255A Chip Select Logic and (b) I/O Port Addresses

**FIGURE 13.29**

8255A Control Word Format for I/O Mode

SOURCE: Adapted from Intel Corporation, copyright 1981.

To communicate with peripherals through the 8255A, the following three steps are necessary:

1. Determine the addresses of ports A, B, and C and of the control register according to the Chip Select logic and the address lines A<sub>0</sub> and A<sub>1</sub>.
2. Write a control word in the control register.
3. Write I/O instructions to communicate with peripherals through ports A, B, and C.

### 13.72 Mode 0: Simple Input or Output

In this mode, ports A and B function as two 8-bit I/O ports and port C as two 4-bit ports. Each port (or half port, in the case of C) can be programmed to function as simply an input

or output port. The input/output features in Mode 0 are as follows:

1. Outputs are latched.
2. Inputs are not latched.
3. Ports do not have handshake interrupt capability.

### 13.73 BSR (Bit Set/Reset) Mode

The BSR mode is concerned only with the 8 bits of port C, which can be set or reset by writing an appropriate control word in the control register. A control word with bit D<sub>7</sub> = 0 is recognized as a BSR control word, and it does not alter any previously transmitted control word with bit D<sub>7</sub> = 1; thus the I/O operations of ports A and B are not affected by a BSR control word. In the BSR mode, individual bits of port C can be used for applications such as an on/off switch.

#### BSR CONTROL WORD

This control word, when written in the control register, sets or resets one bit at a time, as shown in Figure 13.30.

### 13.74 Illustration: Interfacing an A/D Converter Using the 8255A in Mode 0 and BSR Mode.

#### PROBLEM STATEMENT

Design an interfacing circuit to read data from an A/D converter, using the 8255 in the peripheral-mapped I/O.

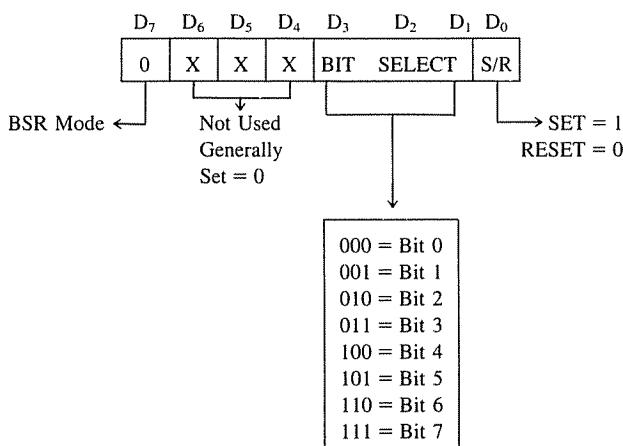


FIGURE 13.30  
8255A Control Word Format in the BSR Mode

1. Set up port A to read data.
2. Set up bit  $PC_0$  to start conversion and bit  $PC_7$  to read the ready status of the converter.

### PROBLEM ANALYSIS

The Chip Select logic in Figure 13.31 is similar to that in the previous examples. The port addresses can be obtained by examining the decoding logic of the  $\overline{CS}$  signal and combining that with the  $A_1$  and  $A_0$  signals. The port addresses range from  $20_H$  to  $23_H$ , as shown.

$A_7$	$A_6$	$A_5$	$A_4$	$A_3$	$A_2$	$A_1$	$A_0$	
0	0	1	0	0	0	0	0	$= 20_H$
							1	$= 21_H$
						1	0	$= 22_H$
						1	1	$= 23_H$
								Control Register

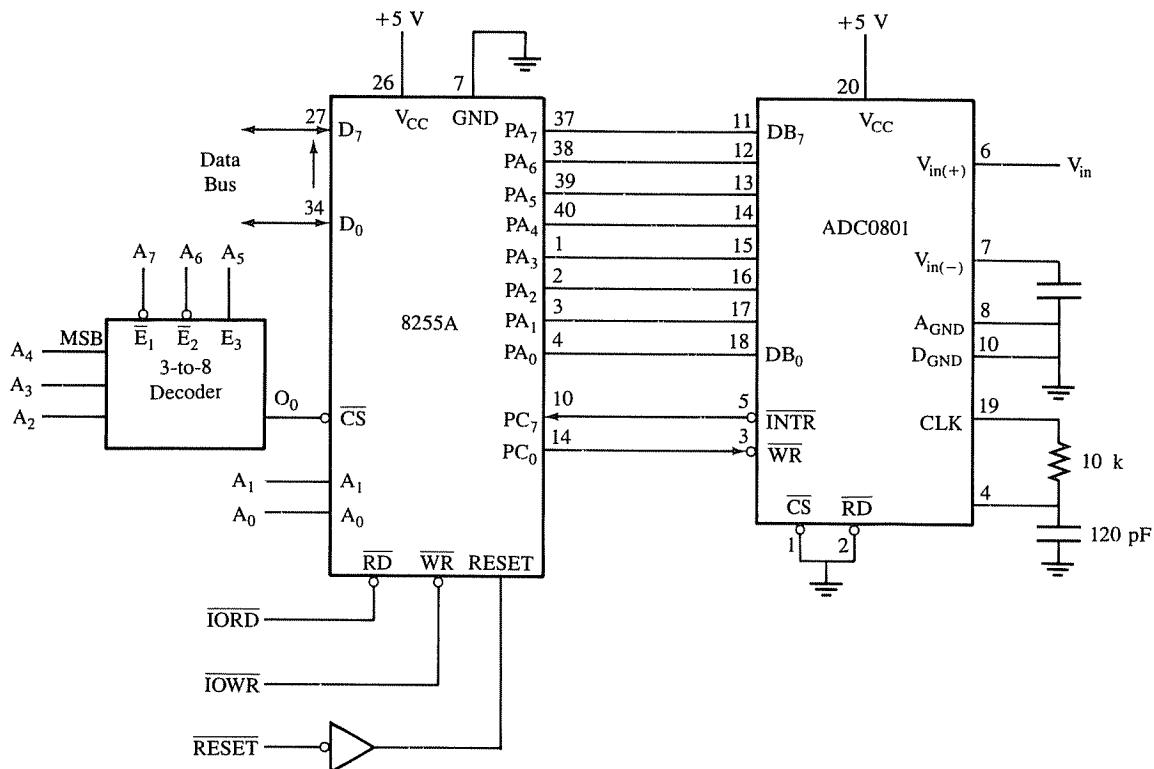


FIGURE 13.31

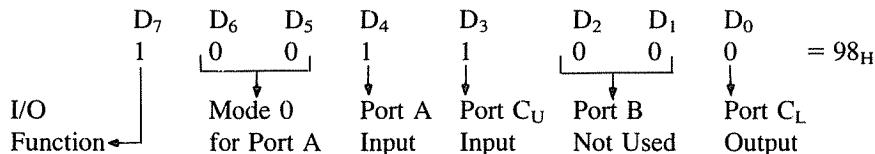
Schematic: Interfacing the A/D Converter ADC0801 Using the 8255A in Mode 0 and BSR Mode

**MODE 0 CONTROL WORD**

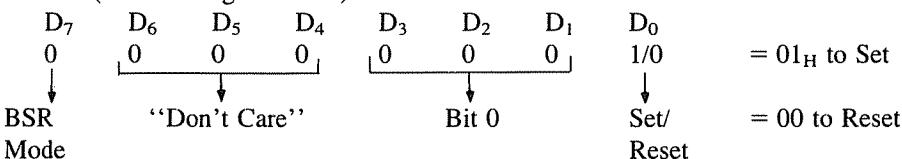
The configuration of the ports is specified as follows:

- Port A as an input port.
- Port C<sub>L</sub> as an output port because bit PC<sub>0</sub> is being used to start conversion.
- Port C<sub>U</sub> as an input port to read the status at PC<sub>7</sub>.
- Port B not being used.

Therefore, the control word necessary to meet the requirements is 98<sub>H</sub> as shown.

**BSR CONTROL WORD FOR START PULSE**

Bit PC<sub>0</sub> is being used as a start pulse. To set and reset PC<sub>0</sub>, the BSR control word is as follows (Refer to Figure 13.30):

**SUBROUTINE**

```

PORTA    EQU 20H          ;Port A address
PORTC    EQU 22H          ;Port C address
CNTRL   EQU 23H          ;Control register address
A/D:     LD C, 23H        ;Set up Z80 C register as pointer to 8255A
                  ;control register
                  LD A, 10011000B ;Load the mode 0 control word (98H)
                  OUT (C), A    ;Write in the control register to set up A and
                  ;CU as inputs
                  LD A, 00H        ;Load BSR control word to reset PC0
                  OUT (C), A    ;Send WR pulse
                  CALL DELAY      ;Wait for sufficient pulse width
                  LD A, 01H        ;Load BSR control word to set PC0
                  OUT (C), A    ;Start conversion
READ:    IN A, (C)        ;Read bit PC7
                  RLA            ;Place PC7 in the carry
                  JP C, READ      ;Wait in the loop until the end of conversion
                  IN A, (20H)     ;Read A/D converter
                  RET

```

### PROGRAM DESCRIPTION

The 8255A ports are initialized by placing the control word 98<sub>H</sub> into the control register. To provide a start pulse to the converter, logic 0 is sent to bit PC<sub>0</sub> in port C, and after a sufficient delay, bit PC<sub>0</sub> is set to start the conversion. The end of conversion is checked by verifying the status of line PC<sub>7</sub>. When PC<sub>7</sub> goes low, the instruction LD A, (20H) reads and places data into the accumulator.

### 13.75 Mode 1: Input or Output with Handshake

In Mode 1, handshake signals are exchanged between the MPU and peripherals prior to data transfer. The features of this mode are as follows:

1. Two ports—A and B—function as 8-bit I/O ports. They can be configured either as input or output ports.
2. Each port uses three lines from port C as handshake signals. The remaining two lines of port C can be used for simple I/O functions.
3. Input and output data are latched.
4. Interrupt logic is supported.

In the 8255A, the specific lines used from port C for handshake signals vary according to the I/O function of a port. Therefore, input and output functions in Mode 1 are discussed separately.

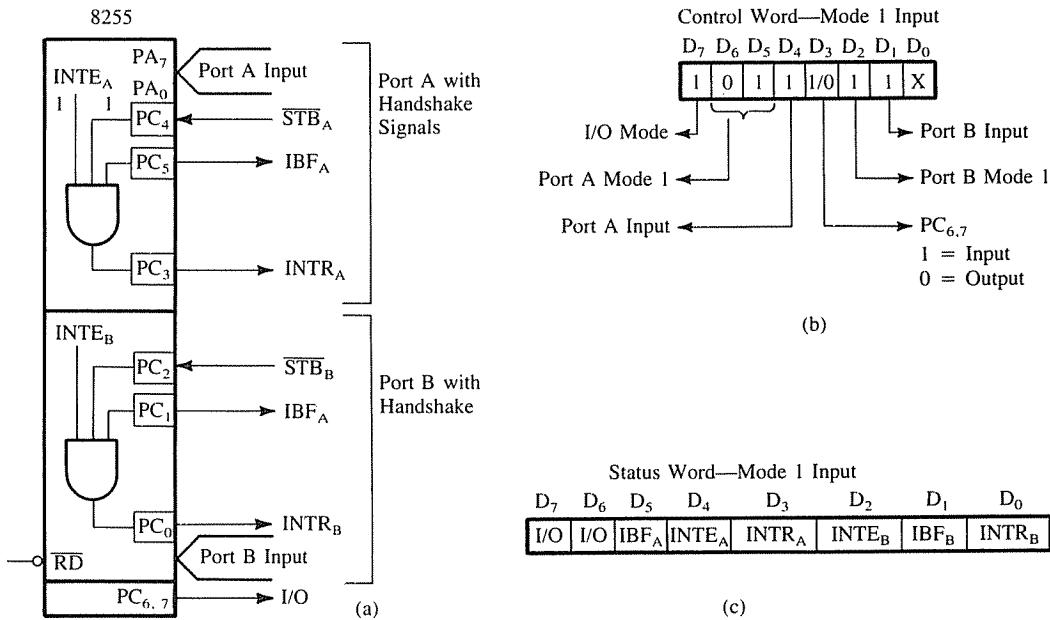
#### MODE 1: INPUT CONTROL SIGNALS

Figure 13.32(a) shows the associated control signals used for handshaking when ports A and B are configured as input ports. Port A uses the upper three signals—PC<sub>3</sub>, PC<sub>4</sub>, and PC<sub>5</sub>—while Port B uses PC<sub>2</sub>, PC<sub>1</sub>, and PC<sub>0</sub>. The functions of these signals are:

- STB (Strobe Input): This signal (active low) is generated by a peripheral device to indicate that it has transmitted a data byte. The 8255A, in response to STB, generates IBF and INTR, as shown in Figure 13.33.
- IBF (Input Buffer Full): This signal is an acknowledgment by the 8255A to indicate that the input latch has received the data byte. This is reset when the MPU reads the data (Figure 13.33).
- INTR (Interrupt Request): This is an output signal that may be used to interrupt the MPU. This signal is generated if STB, IBF, and INT<sub>E</sub> (internal flip-flop) are all at logic 1, and is reset by the falling edge of the RD signal (Figure 13.33).
- INT<sub>E</sub> (Interrupt Enable): This is an internal flip-flop used to enable or disable the generation of the INTR signal. The two flip-flops INT<sub>E</sub><sub>A</sub> and INT<sub>E</sub><sub>B</sub> are set/reset through the BSR mode. The INT<sub>E</sub><sub>A</sub> is enabled/disabled through PC<sub>4</sub>, and INT<sub>E</sub><sub>B</sub> is enabled/disabled through PC<sub>2</sub>.

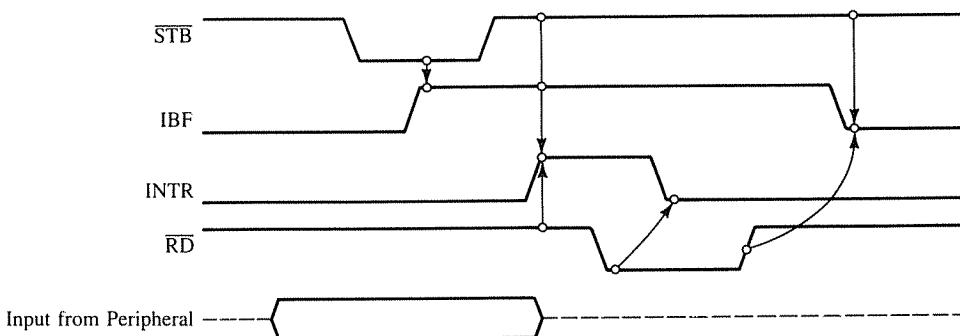
#### CONTROL AND STATUS WORDS

Figure 13.32(b) uses control words derived from Figure 13.29 to set up port A and port B as input ports in Mode 1. Similarly, Figure 13.32(c) shows the status word, which will be placed into the accumulator if port C is read.



**FIGURE 13.32**  
8255A Mode 1: Input Configuration

SOURCE: Adapted from Intel Corporation, copyright 1981.



**FIGURE 13.33**  
8255A Mode 0: Timing Waveforms for Strobed Input (with Handshake)

SOURCE: Adapted from Intel Corporation, copyright 1981.

### PROGRAMMING THE 8255A IN MODE 1

The 8255A can be programmed to function using either status check I/O or interrupt I/O. Figure 13.34(a) shows a flowchart for the status check I/O. In this flowchart, the MPU continues to check data status through the IBF line until it goes high. This is a simplified flowchart and does not show how to handle data transfer if two ports are being used. The technique is similar to that of Mode 0 combined with the BSR mode. The disadvantage of the status check I/O with handshake is that the MPU is tied up in the loop.

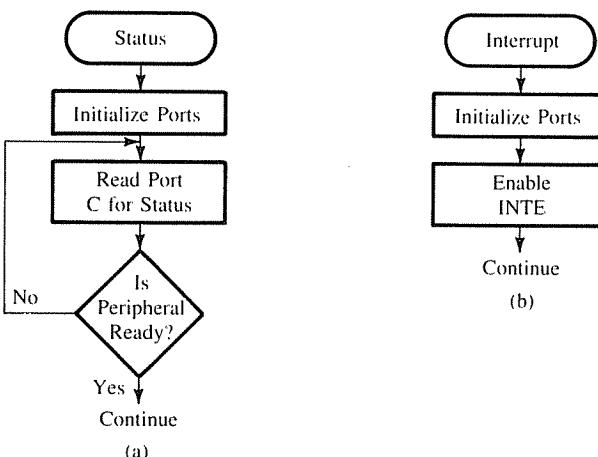
The flowchart in Figure 13.34(b) shows the steps required for the interrupt I/O, assuming that vectored interrupts are available. The confusing step in the interrupt I/O is to set INT<sub>E</sub> for either port A or port B. Figure 13.33(a) shows that the STB signal is connected to pin PC<sub>4</sub> and the INT<sub>E</sub><sub>A</sub> is also controlled by the pin PC<sub>4</sub>. (In port B, the pin PC<sub>2</sub> is used for the same purposes.) However, the INT<sub>E</sub><sub>A</sub> is set or reset in the BSR mode and the BSR control word has no effect when ports A and B are set in Mode 1.

In case the INTR line is used to implement the interrupt, it may be necessary to read the status of INT<sub>R</sub><sub>A</sub> and INT<sub>R</sub><sub>B</sub> to identify the port requesting an interrupt service and to determine the priority through software, if necessary.

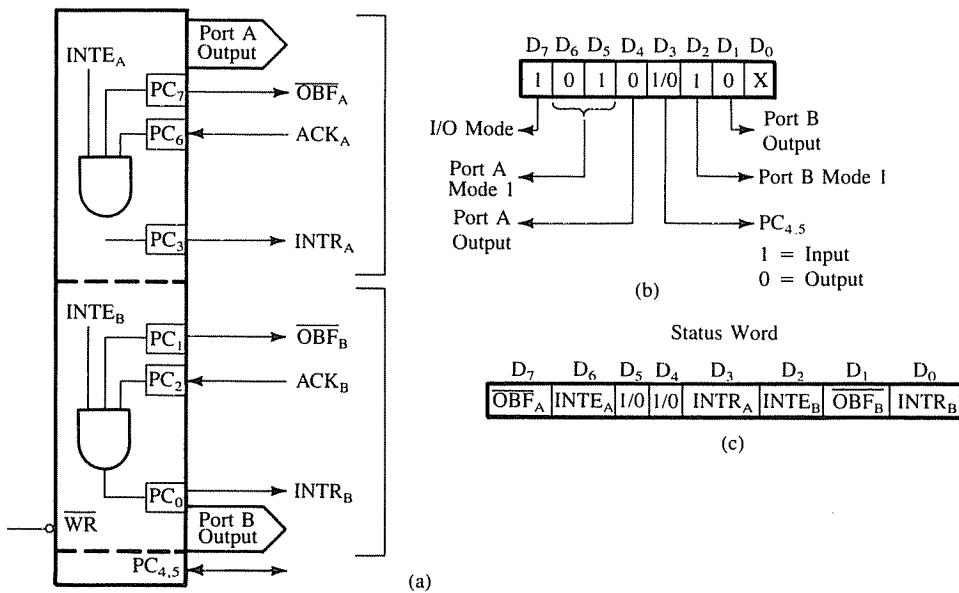
### MODE 1: OUTPUT CONTROL SIGNALS

Figure 13.35 shows the control signals when ports A and B are configured as output ports. These signals are defined as follows:

- OBF (Output Buffer Full): This is an output signal that goes low when the MPU writes data into the output latch of the 8255A. This signal indicates to an output peripheral that new data are ready to be read (Figure 13.36). It goes high again after the 8255A receives an ACK (Acknowledge) from the peripheral.

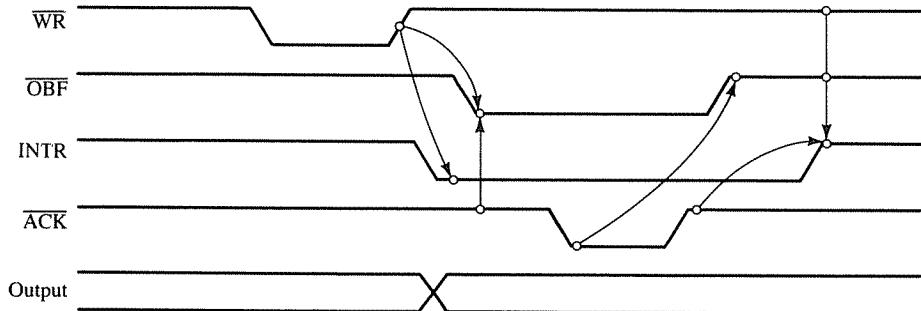


**FIGURE 13.34**  
Flowcharts: (a) Status Check I/O (b) Interrupt I/O



**FIGURE 13.35**  
8255A Mode 1: Output Configuration

SOURCE: Adapted from Intel Corporation, copyright 1981.



**FIGURE 13.36**  
8255A Mode 1: Timing Waveforms for Strobed Input (with Handshake)

SOURCE: Adapted from Intel Corporation, copyright 1981.

- ACK (Acknowledge): This is an input signal to the 8255A from a peripheral, which must output a low when the peripheral receives the data from the 8255A ports (Figure 13.36).
- INTR (Interrupt Request): This is an output signal, and it is set by the rising edge of the ACK signal. This signal can be used to interrupt the MPU to request the next data byte

for output. The INTR is set when  $\overline{OBF}$ ,  $\overline{ACK}$  and INTE are all one (Figure 13.36) and reset by the falling edge of WR.

- INTE (Interrupt Enable): This is an internal flip-flop to a port and needs to be set to generate the INTR signal. The two flip-flops INTE<sub>A</sub> and INTE<sub>B</sub> are controlled by bits PC<sub>6</sub> and PC<sub>2</sub>, respectively, through the BSR mode.
- PC<sub>4</sub> and PC<sub>5</sub>: These lines can be set up as either input or output.

### CONTROL AND STATUS WORDS

Figure 13.35(b) shows the control word needed to set up port A and port B as output ports in Mode 1. Similarly, Figure 13.35(c) also shows the status word, which will be placed into the accumulator if port C is read.

### 13.76 Mode 2: Bidirectional Data Transfer

This mode is used primarily in applications such as data transfer between two microcomputers or floppy disk controller interface. In this mode, port A can be configured as the bidirectional port and port B in either Mode 0 or Mode 1. Port A uses five signals from port C as controls signals for data transfer. The remaining three signals from port C can be used either as simple I/O or as handshake signals for port B. Figure 13.37 shows two configurations of Mode 2.

### 13.77 Comparison of the Z80 PIO and the 8255

The Z80 PIO and the Intel 8255A are two widely used peripheral interface devices, and they are designed to serve similar functions. However, the 8255A is a general-purpose

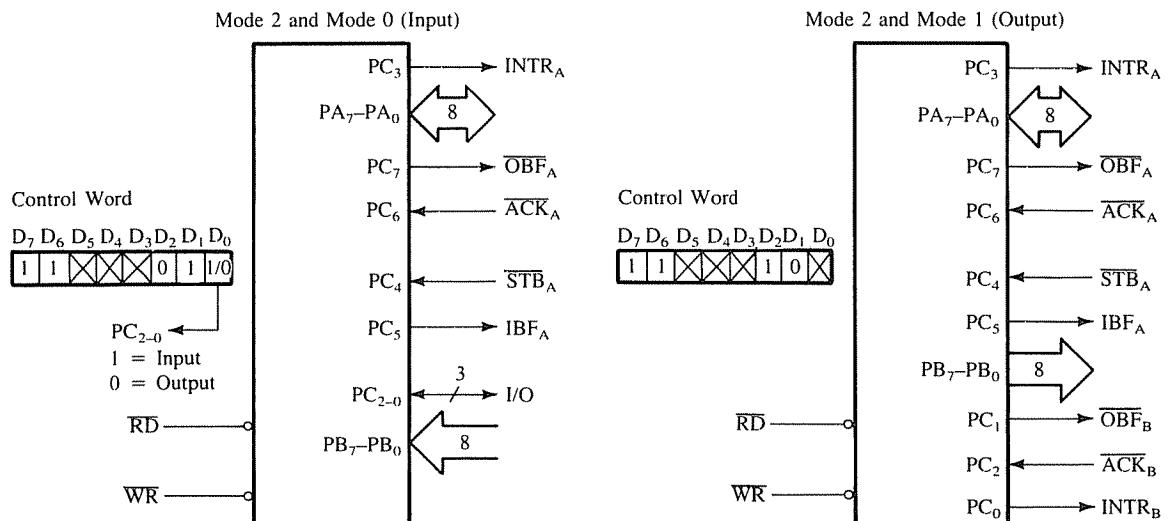


FIGURE 13.37

8255A Mode 2. Bidirectional Input/Output

SOURCE: Adapted from Intel Corporation, copyright 1981.

**TABLE 13.2**  
Comparison of the Z80 PIO and the 8255

	PIO	8255A
1. Number of 8-bit Ports	Two: A and B	Three: A, B, and C Port C Consists of Two 4-bit Ports.
2. I/O Lines in Bit Mode	16 Lines	8 Lines in Addition to Ports A and B
3. Handshake and Interrupt Signals	Ports A and B Separate 4 Lines	Ports A and B Uses Lines of Port C
4. Bidirectional Mode	Port A	Port A
5. Status Check of Handshake Signals	Not Available	Available through Port C
6. Interrupt I/O	Ports A and B	Ports A and B
7. Logic Check and Interrupt	Bit Mode	Not Available
8. Daisy Chain Interrupt Priority	Available	Not Available

device and can be used with any microprocessor. On the other hand, the PIO is specifically designed to work with the Z80 microprocessor. Therefore, the PIO can offer some special features, such as daisy chain interrupt priority, based on its ability to recognize the instruction RETI. The PIO has some other attractive features, such as its ability to recognize predefined logic conditions in the bit mode. The advantages of the 8255A over the PIO are that the 8255A has three I/O ports and the status of the handshake signals can be monitored by reading port C. In the 8255A, data transfer can be set up under the program control or the interrupt control; the PIO is better suited for the interrupt control. The comparison of these two devices is shown in Table 13.2.

## SUMMARY

This chapter has been concerned with the basic concepts (such as control register, control logic, and handshake signals) underlying a programmable device. Based on these concepts, the Z80 PIO (Programmable Input/Output) device was discussed in details with illustrative applications. Finally, another widely used peripheral interfacing device, the Intel 8255A, was discussed and compared with the PIO. The important points can be summarized as follows:

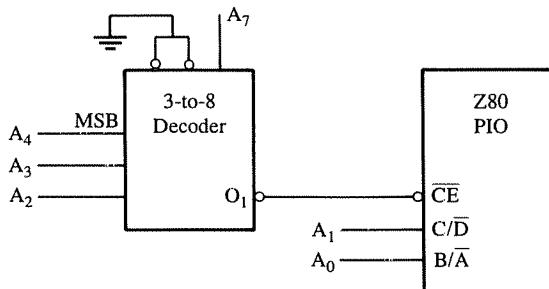
- A programmable interface device is designed to perform various I/O functions, and these functions can be specified by writing an appropriate control word (or words) into its control registers.

- A programmable I/O device generally includes multiple I/O ports, control register(s), handshake signals, and interrupt capability.
- The signals that are exchanged between the MPU and peripherals prior to data transfer are called handshake signals. These signals check whether a peripheral is ready for data transfer and inform the MPU accordingly.
- The Z80 PIO is a programmable I/O device with two I/O ports (A and B), and it has four operating modes: Mode 0 (output), Mode 1 (input), Mode 2 (bidirectional), and Mode 3 (bit mode).
- Each PIO port has two handshake signals, and each port can be used to transfer data under interrupt control.
- In PIO, only Port A can be used for bidirectional data transfer, and all handshake lines are used for this data transfer. Port A handshake lines are used for output control and Port B handshake lines for input control.
- Both PIO ports can be set up in the bit mode, whereby each line can be assigned either input or output function. For input lines, AND or OR logic function can be specified in the control register, and an interrupt can be generated when the conditions exist.
- The PIO has two signals—IEI and IEO—which are used to set up daisy chain priority.
- The Intel 8255A is a general purpose programmable interfacing device, and it has three ports. It can also operate in various modes similar to the PIO.

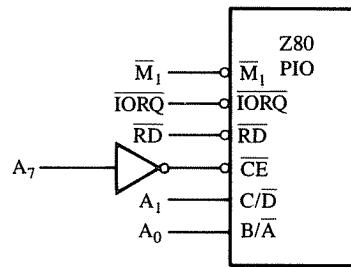
## ASSIGNMENTS

---

1. List the internal components found in a typical programmable device.
2. Explain the functions of handshake signals.
3. List the operating modes of the PIO and their features.
4. If the PIO does not include the Write signal, specify the control signals necessary to perform a Write operation.
5. Write instructions to set up Port A in Mode 0 and Port B in Mode 1.
6. In Figure 13.6, identify the addresses of Port A and Port B and their control registers if the output line  $O_7$  of the decoder is connected to the  $\overline{CE}$  signal of the PIO.
7. In Figure 13.6, identify the addresses of ports A and B and their control registers if the address lines  $A_0$  and  $A_1$  are interchanged.
8. In Figure 13.6, exchange the address lines  $A_7$  and  $A_5$  of the decoder, and identify port addresses.
9. In Figure 13.38, address lines  $A_6$  and  $A_5$  are “don’t care.” Specify the multiple addresses that can access ports A and B.
10. Identify the addresses of ports A and B and their control registers in Figure 13.39, assuming all “don’t care” lines at logic 0.



**FIGURE 13.38**  
PIO Interfacing for Assignment 9



**FIGURE 13.39**  
PIO Interfacing for Assignment 10

11. Port A of the PIO is initialized in Mode 3 with bits D<sub>7</sub>–D<sub>5</sub> as input and D<sub>4</sub>–D<sub>0</sub> as output. The PIO generates an interrupt when bits D<sub>7</sub> and D<sub>6</sub> are both at logic 0. Write initialization instructions.
12. Explain the functions of the handshake signals ASTB and ARDY if PIO Port A is initialized as an input port.
13. List the control words that need to be written into the control register to set up PIO Port B in Mode 1 for interrupt control I/O.
14. If PIO Port B is initialized as an output port, list the sequence of events that occurs when a data byte is transferred to a peripheral under interrupt I/O.
15. The PIO ports A and B are initialized as output ports to transfer data under interrupt control. Write instructions for the Z80 MPU to initialize the stack at 20A7<sub>H</sub> and the interrupt register at 20<sub>H</sub>. Assume that the service routine for Port A is at 2072<sub>H</sub> and for port B at 2097<sub>H</sub>, and that the interrupt vectors are located at 2048<sub>H</sub> and 204A<sub>H</sub> for port A and B, respectively.
16. Write a service routine to output a byte to the peripheral in Assignment 15. Show the memory addresses where the service routine is to be stored.
17. In 15, an interrupt occurs from Port A when the Z80 MPU is executing a 3-byte instruction located at 2022<sub>H</sub>–2024<sub>H</sub>. List the stack addresses and their contents when the interrupt is acknowledged and the program control is transferred to the service routine.
18. In Figure 13.12, if PIO-3 is being serviced, specify the status of the pins IEI and IEO of PIO-2 and PIO-3.
19. The keyboard routine for Figure 13.15 gives the priority from key K<sub>7</sub> to key K<sub>0</sub> in that sequence. Modify the subroutine to change the priority sequence so that key K<sub>0</sub> has the highest priority.
20. When two keys are pressed simultaneously (Figure 13.15), the subroutine recognizes only the higher priority key. Modify the subroutine to recognize both keys.
21. Redraw Figure 13.15 to replace the PIO with an octal buffer (such as the 74LS240) and an octal latch (such as the 74LS373).

22. Modify Figure 13.18 to show how a  $4 \times 4$  matrix keyboard can be connected to Port A of the PIO. Write initialization instructions to set up Port A in Mode 3 to generate an interrupt whenever a key is pressed.
23. Write initialization instructions to set up Port A in the bidirectional mode and Port B in Mode 3. Assign lines  $B_7-B_4$  as input and  $B_3-B_0$  as output. Can you write the control word to generate an interrupt when the lines  $B_7$  and  $B_6$  are at logic 1? Explain your answer.
24. Write necessary software to transfer 100 bytes of data from Micro-2 to Micro-1 (Figure 13.24).
25. List the operating modes of the 8255A Programmable Peripheral Interface.
26. Specify the bit of a control word for the 8255A that differentiates between the I/O mode and the BSR mode.
27. Write initialization instructions for the 8255A to set up Ports A and B in the handshake mode with the interrupt I/O.
28. Figure 13.40 shows an interfacing of the data converter ADC0801 using Port A of the Z80 PIO; Port A is set up in Mode 1. The handshake signal ARDY is used to start the conversion by connecting it to WR through the one-shot multivibrator 74121, and ASTB is used to detect the end of the conversion by monitoring the line INTR of the converter. To start the conversion, one dummy Read instruction IN A, (PIOA) is executed. Explain the need for the one-shot multivibrator. (Hint: The data converter needs a pulse transition low-to-high to start the conversion.)

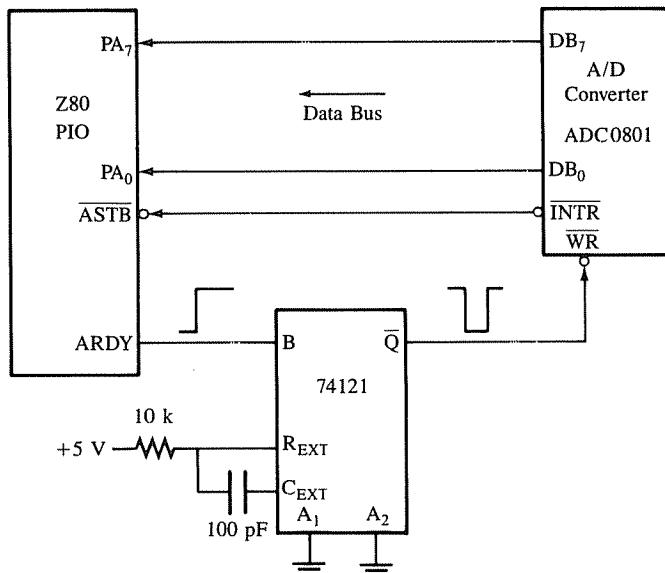
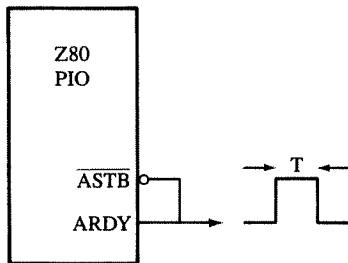


FIGURE 13.40

Interfacing the ADC0801 Using the Z80 PIO in Mode 1 with Handshake

29. Write instructions to set up the Z80 PIO Port A in Mode 1 and the Z80 in interrupt Mode 2 with the interrupt vector  $2070_{H}$ . Initialize the memory pointer at  $2050_{H}$  to store data and the counter to record ten readings, and start the conversion. Write also an interrupt service routine to read the port, store data in memory (for ten readings), start the conversion again, and enable the interrupt.
30. In Figure 13.41, Port A of the Z80 PIO is initialized in Mode 0, and its hand-shake signals ARDY and ASTB are tied together. When we execute the instruction OUT (PIOA), A, a pulse equal to one system clock period is generated at the output of ARDY that can be used as a strobe to start a process at a peripheral. Explain the output pulse of ARDY. (Hint: In Mode 0, the transition low-to-high of the STB signal turns off the RDY signal—see Figure 13.10.)

**FIGURE 13.41**  
Generating a Strobe Pulse Using  
PIO Handshake Lines





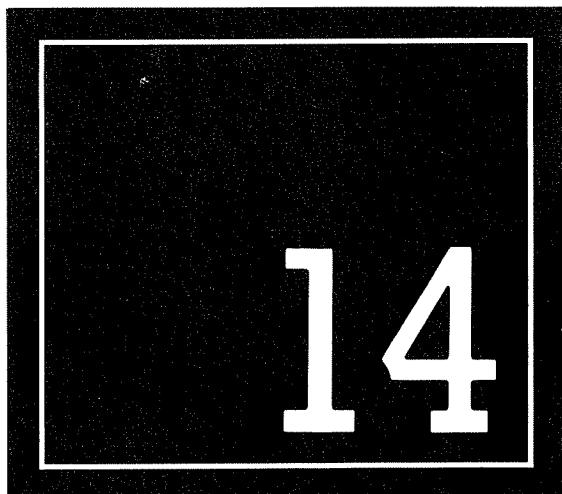
# Programmable Timers and Counters

A programmable timer/counter device is designed to generate accurate time delays using the system's clock and to count occurrences of external events. It can be used for applications such as a real-time clock, an event counter, and a signal generator.

This chapter is concerned primarily with the Z80 CTC—Counter/Timer Circuit—and its applications. The CTC has four timer/counter channels and can be programmed to function as timers or counters by the user's writing control words into appropriate internal registers. The CTC is also capable of generating interrupt signals at a specified time delay or count. The CTC is widely used as a timer/counter and has become an integral part of Z80-based microprocessor systems. This chapter also includes the description of another widely used timer/counter—the Intel 8253—and its features are compared with those of the CTC.

## OBJECTIVES

- List the elements of the block diagram of the Z80 CTC (Counter/Timer Circuit) and explain functions of each element.
- List the operating modes of the CTC and explain the differences in these modes.
- Explain how the CTC operates as a counter and a timer and its interrupt capability.
- Identify port addresses of each counter channel in a given circuit.
- Write initialization instructions to set up the CTC in either the counter or the timer mode.
- Design a circuit to interface the CTC for given port addresses and write instructions to set up the CTC as a timer or a counter.
- List the elements of the Intel 8253 Programmable Interval Timer and its operating modes.
- Design a circuit to interface the 8253 for given port addresses and write instructions to set up the device for a given mode.



# 14.1

## Z80 CTC—COUNTER/TIMER CIRCUIT

---

The **Z80 CTC (Counter/Timer Circuit)** is a 28-pin programmable chip, specially designed to work with the Z80 microprocessor. The block diagram (Figure 14.1) shows that the chip includes four counter/timer channels, control logic, interrupt control, and the system data bus. The CTC requires +5 V power supply and the Z80 system clock (CLK).

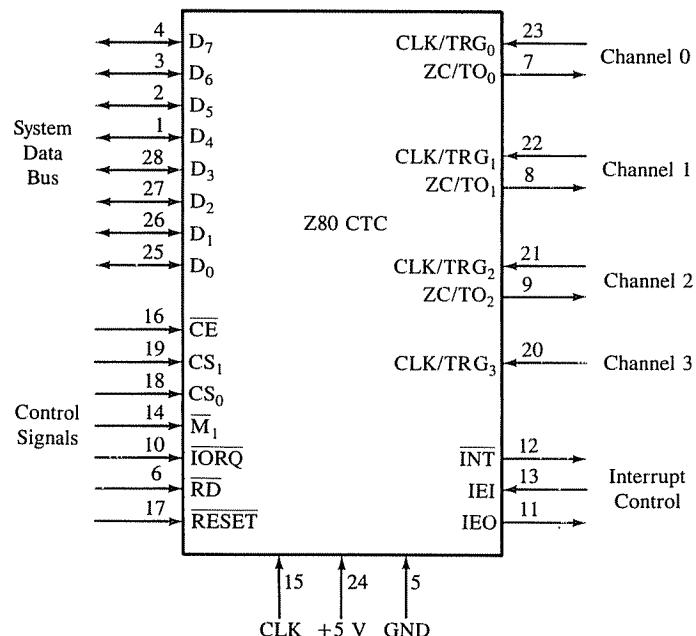
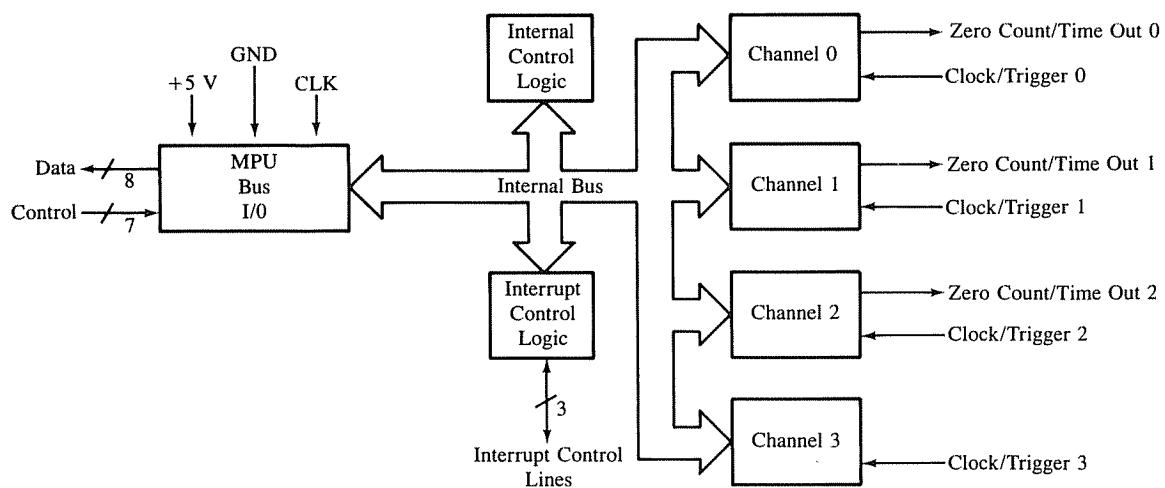
The system data bus, D<sub>7</sub>–D<sub>0</sub>, consists of bidirectional tri-state lines and is connected to the data bus of the Z80 microprocessor. These bus lines transfer all data and commands between the Z80 and the CTC.

The four channels can be independently programmed in either the timer or the counter mode. An 8-bit number (time constant) is loaded into the register of a channel, and is decremented on every clock pulse. At the end of the count, it generates a pulse at the ZC/TO (Zero Count/Time Out) pin; reloads the time constant into the register, and begins the next operation. If the CTC is programmed to use the interrupt, it generates an interrupt pulse at the end of the count, and its IEI and IEO signals can be used to set up the daisy chain interrupt priorities.

### 14.11 Interfacing the CTC

The CTC has an 8-bit bidirectional data bus, one output signal per channel (except Channel 3), and seven control signals including the chip enable signal, as shown in Figure 14.1. These signals are connected to the appropriate signals of the Z80 microprocessor.

- D<sub>0</sub>–D<sub>7</sub> (Data Bus): This is a tri-state bidirectional data bus that transfers data and commands between the Z80 and the CTC.
- CLK (Clock): This is an input from the Z80 system clock.
- CLK/TRG (Clock/Trigger): This is an external input signal. In the counter mode, it is used to count external events, and its active level (high or low) is specified in the control word. In the timer mode, an active edge starts the timer.
- ZC/TO (Zero Count/Time-Out): This is an active high output, generated when the down-counter has been decremented to zero.
- IORQ (I/O Request), M<sub>1</sub> (Machine Cycle 1), and RD (Read): These are three active low signals and perform Read/Write/Interrupt Acknowledge operations as described below.
- Read: This operation is performed when IORQ and RD are active low and M<sub>1</sub> is high. In this operation, the contents of the counter of the selected channel can be read.
- Write: There is no separate signal for the Write operation. When RD is high, the CTC generates the Write signal internally. In this operation, control word and time constant can be written in the selected channel.
- Interrupt Acknowledge: The Z80 acknowledges the interrupt by asserting two control signals (IORQ and M<sub>1</sub>) low, and the highest priority interrupting channel places its interrupt vector on the data bus. (This operation is discussed in detail later in Section 14.13 and 14.14).

**FIGURE 14.1**

Z80 CTC: Block Diagram and Logic Pinout

SOURCE: Courtesy of Zilog, Inc.

**TABLE 14.1**  
Control Operations and Signals

Operations	$\overline{IORQ}$	$\overline{RD}$	$\overline{M_1}$
Read	Low	Low	High
Write	Low	High	High
Interrupt	Low	High	Low
Acknowledge			

- The control operations and the active level of the associated control signals are summarized in Table 14.1.
- RESET (Reset): This is an active low signal that terminates the counting operation and disables the interrupts; all outputs go inactive and the data bus  $D_7-D_0$  goes to the high impedance state.
- $\overline{CE}$  (Chip Enable): This is an active low signal connected to the decoded low-order address bus of the Z80. When this signal is active, the CTC is selected.
- $\overline{CS}_0$  and  $\overline{CS}_1$  (Channel Select): These two lines are generally connected to the address lines  $A_0$  and  $A_1$ , respectively, and the logic combination of these lines (as shown in the following list) selects one of the four channels of the CTC to write into or read from. The decoded address of the Chip Enable and the logic levels of  $\overline{CS}_0$  and  $\overline{CS}_1$  determine the port address of the selected counter channel.

$\overline{CE}$	$\overline{CS}_1$	$\overline{CS}_0$	Selected Channel
0	0	0	Channel 0
0	0	1	Channel 1
0	1	0	Channel 2
0	1	1	Channel 3

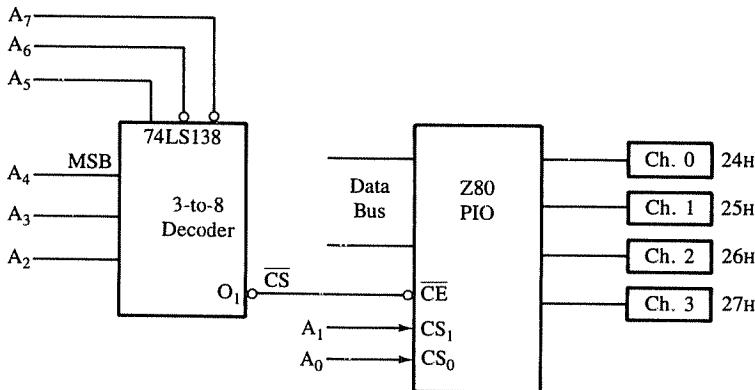
Example 14.1 illustrates the use of these signals in interfacing the CTC.

**Example  
14.1**

Determine the port addresses of the CTC channels shown in Figure 14.2.

**Solution**

To select the CTC, the output line  $O_1$  of the decoder should go low. Therefore the logic levels of the address lines  $A_7-A_2$  should be as shown. Combining the logic levels of  $A_7-A_2$  with those of  $A_1$  and  $A_0$  gives us port addresses of the counter channels ranging from  $24_H$  to  $27_H$ .



**FIGURE 14.2**  
Interfacing the CTC

A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	= 24 <sub>H</sub>	Channel 0
0	0	1	0	0	1	0	0	= 25 <sub>H</sub>	Channel 1
↓ Decoder Enable			↓ Decoder Input			1	0	= 26 <sub>H</sub>	Channel 2
						1	1	= 27 <sub>H</sub>	Channel 3
Channel Select									

## 14.12 Programming the CTC

The CTC can be programmed to operate in either the timer mode or the counter mode. Each channel consists of *channel control logic*, *time constant register*, and the *down-counter*, as shown in Figure 14.3. To program the CTC, a control word should be written into the channel, and it must be followed by an 8-bit time constant value loaded into the time constant register. The control word determines such parameters as the operating mode, the active trigger level (falling or rising edge), and the interrupt logic (see Figure 14.4). The time constant (count), which can be from 1 to 256 (0 = 256), is loaded into the down-counter and decremented according to the specified mode operation. When the count reaches zero, it is automatically reloaded into the register. Figure 14.3 also shows a block called prescaler. This is used only in the timer mode; it divides the system clock frequency by either 16 or 256. The output of the prescaler decrements the down-counter in the timer mode.

The channel control word is shown in Figure 14.4 and is illustrated in Example 14.2.

FIGURE 14.3

Internal Architecture of a Channel  
SOURCE: Courtesy of Zilog, Inc.

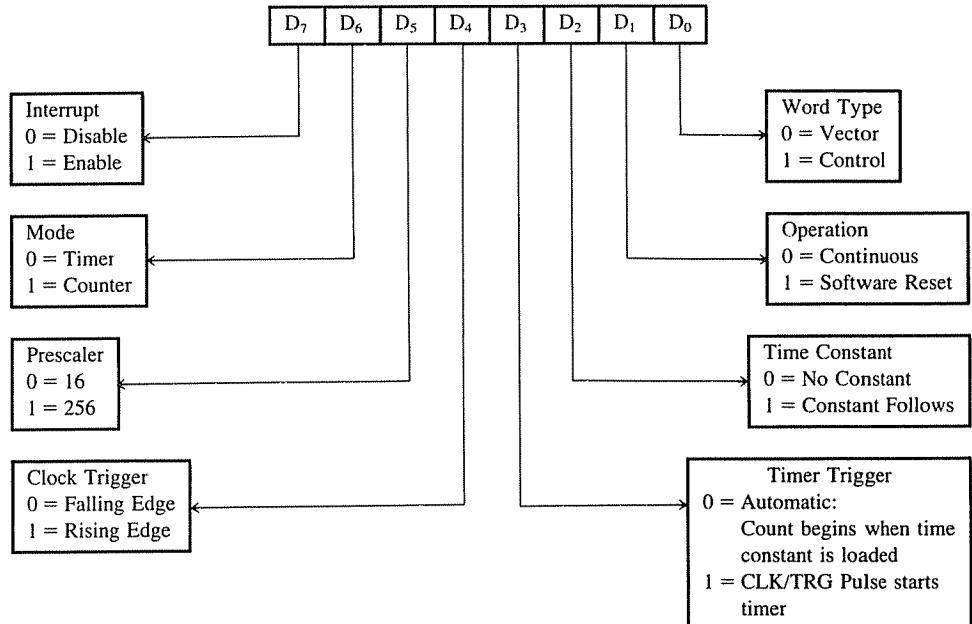
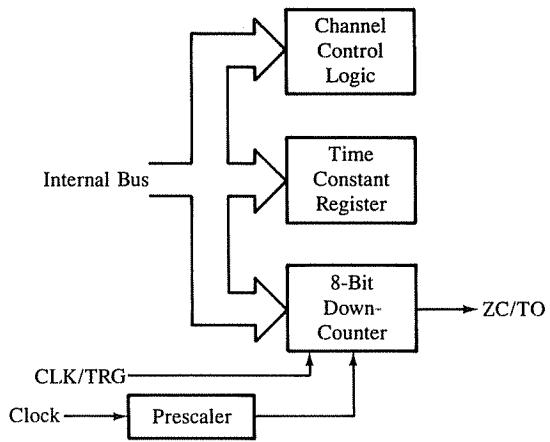


FIGURE 14.4

Channel Control Word

Write instructions to program Channel 0 of the CTC (Figure 14.2) in the timer mode to provide a pulse every 20 ms if the system clock is 1 MHz.

**Example  
14.2**

Channel 0 of the CTC in Figure 14.2 has the port address 24<sub>H</sub>. To program the channel in the timer mode, we need to send two words to the channel port: a channel control word and a proper count to provide 20 ms delay. Assuming the prescaler is 256, the system clock frequency will be divided by 256; in other words, the clock period will be multiplied by 256. Therefore the total delay between two consecutive outputs is

$$T_d = T_c \times PS \times TC$$

where  $T_c$  = System clock  
 $PS$  = Prescaler  
 $TC$  = Time constant (Count).

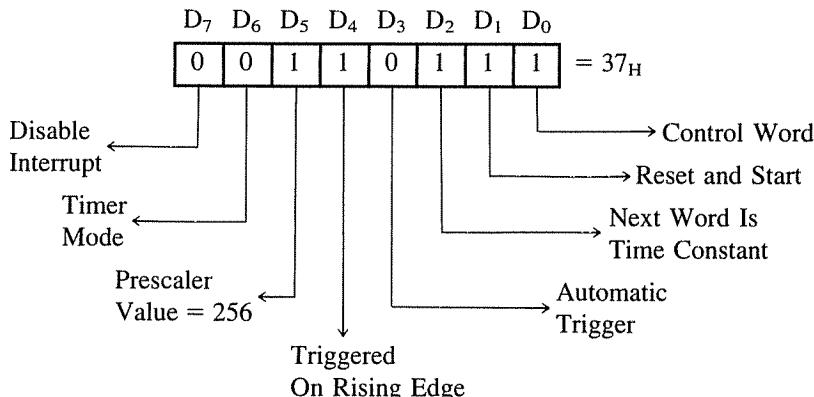
**Solution**

The system has a 1  $\mu$ s clock period; therefore, the number in the time constant register will be decremented every 256  $\mu$ s. The count necessary to obtain a 20 ms delay is

$$20 \text{ ms} = 1 \text{ } \mu\text{s} \times 256 \times TC \text{ (Count)}$$

$$TC = \frac{20 \text{ ms}}{256 \text{ } \mu\text{s}} = 78.125 \cong 78 = 4E_H.$$

Assuming the counter will begin at the rising edge of the system clock as soon as the count is loaded into the register, the control word is as follows:



**Instructions** The following instructions will set up the CTC in the timer mode with the specified parameters.

CNTRL	EQU 37H	;Defines channel control word
COUNT	EQU 4EH	;Time constant
PORT0	EQU 24H	;Port address of Channel 0

```

SETUP: LD A, CNTRL      ;Channel control word
        OUT (PORT0), A   ;Send control word to Channel 0
        LD A, COUNT       ;Time constant
        OUT (PORT0), A   ;Load time constant in Channel 0
        HALT

```

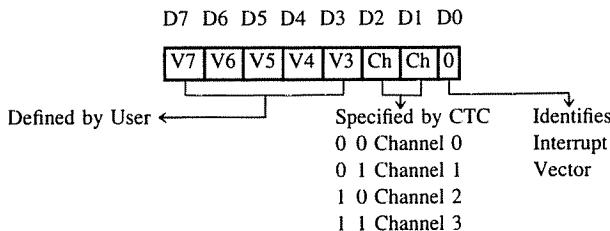
**Description** The first instruction writes the channel control word in Channel 0 (Port 24<sub>H</sub>), and it is followed by the time constant (COUNT = 4E<sub>H</sub>). As soon as the time constant is loaded into the channel, the down-counter begins. Because the prescaler divides the system clock by 256, the count is decremented every 256 µs. The countdown continues until the time constant (4E<sub>H</sub>) goes to zero, and at the end of the count, a high pulse, approximately 1.5 times the system clock period in width, is generated at the output (ZC/TO = Zero Count/Time Out) of Channel 0. The channel word has specified continuous operation; therefore, the time constant is automatically loaded again into the channel register and the down-counter continues generating a pulse every 20 ms.

### 14.13 Using the CTC Interrupts

The CTC architecture includes the interrupt control logic shown in the block diagram (Figure 14.1). This logic is used to generate an interrupt request pulse and to determine the priorities among the four channels as well as among various CTC devices if the system includes more than one CTC. The CTC has three signals associated with the interrupt: INT̄, IEI, and IEO. These signals are functionally similar to those of the Z80 PIO described in the last chapter. The INT̄ signal is again described here, and the other two are described in Section 14.14.

- Int (Interrupt Request): This is an active low output signal, and if the interrupt is enabled in the channel control word, it goes low when the down-counter reaches zero.

To use the interrupt process, the Z80 should be set in the interrupt Mode 2, and the CTC should be programmed to supply the low-order byte of the interrupt vector. The CTC generates an interrupt request (INT) when the down-counter of a channel reaches zero. When the Z80 acknowledges the interrupt request, the Interrupt register I of the Z80 supplies the high-order byte and the CTC supplies the low-order byte of the interrupt vector address. The low-order byte of the interrupt vector and the identification of the channel requesting the interrupt are defined as shown in Figure 14.5. Bits D<sub>7</sub>–D<sub>3</sub> are defined by the user; bits D<sub>2</sub>–D<sub>1</sub> are supplied by the CTC to identify the channel that has reached the count of zero, and bit D<sub>0</sub> must be zero to differentiate the interrupt vector from the control word. To form the interrupt vector, initially, bits D<sub>2</sub> and D<sub>1</sub> can be at any logic level; they are specified by the CTC when a channel requests an interrupt. Once the interrupt vector address of Channel 0 is defined, the remaining vector addresses are automatically defined; they are consecutive addresses with two memory locations for each channel.



**FIGURE 14.5**  
Bit Definition of Interrupt Vector

Modify the instructions in Example 14.2 to program Channel 0 of the CTC (Figure 14.2) in the timer mode to provide a pulse and generate an interrupt every 20 ms. The address of the interrupt service routine is at memory locations  $2050_{\text{H}}$  and  $2051_{\text{H}}$ .

**Example  
14.3**

To program the CTC, we need to load three words into Channel 0.

**Solution**

1. Channel control word: The word is similar to the word in Example 14.2 except that bit D<sub>7</sub> should be 1 to enable the interrupt. Therefore, the word should be changed from  $37_{\text{H}}$  to  $B7_{\text{H}}$ .
2. Time constant: It is the same as in Example 14.2.
3. Interrupt vector for location  $2050_{\text{H}}$ : The high-order byte should be loaded into the interrupt vector register I of the Z80 and the low-order byte into Channel 0.

### Instructions

CNTRL	EQU B7H	;Channel control word
COUNT	EQU 4EH	;Time constant
PORT0	EQU 24H	;Channel 0 port address
SETUP:	DI	;Disable interrupt
	IM 2	;Set up Z80 in interrupt Mode 2
	LD A, 20H	;High-order byte of interrupt vector
	LD I, A	;Load interrupt register with high-order byte
	LD A, CNTRL	;Channel control word = B7H
	OUT (PORT0), A	;Initialize Channel 0
	LD A, COUNT	;Time constant = 4EH as in Example 14.2
	OUT (PORT0), A	;Load time constant into Channel 0
	LD A, 50H	;Low-order byte of interrupt vector
	OUT (PORT0), A	;Load interrupt vector in CTC
	EI	;Enable Z80 interrupt
	↓	;Continue with program

**Description** The first instruction DI disables, and the last instruction EI enables, the Z80 interrupt. This is necessary to avoid any false interrupts when the CTC is being initialized. The interrupt vector for Channel 0 is initialized at  $2050_H$ ; therefore, the interrupt service routine address for Channel 0 must be stored in locations  $2050_H$  and  $2051_H$ . The interrupt vector addresses for Channels 1 through 3 are automatically defined; they range from  $2052_H$  to  $2057_H$ . The interrupt service routine should be terminated by the instruction RETI (Return from Interrupt). The CTC is designed to recognize the RETI instruction, and when it does so, the interrupt request is automatically removed.

#### 14.14 Interrupt Priorities

Among the four CTC channels, Channel 0 has the highest priority and Channel 3 has the lowest priority. When multiple CTC devices are used in a system, they can be connected in the daisy chain format; the CTC has two signals (IEI and IEO) to set up the daisy chain priorities among CTC devices.

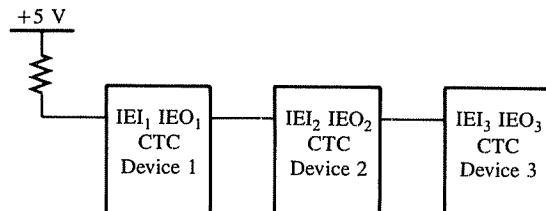
- IEI (Interrupt Enable In): This is an active high input signal, used to set up daisy chain interrupt priorities. A high level on this pin indicates that no other interrupting devices of higher priority in the daisy chain are being serviced by the Z80.
- IEO (Interrupt Enable Out): This is an output signal, used in conjunction with the IEI signal to set up the daisy chain priority in a system. The IEO signal remains high if the IEI is high and the Z80 is not servicing any interrupt from any CTC channel. This signal blocks lower priority devices from interrupting while a higher priority interrupting device is being serviced.

Figure 14.6 shows three CTC devices connected in the daisy chain format. The IEI of Device #1 is tied to +5 V, and its IEO signal is fed to Device #2. In this schematic, Device #1 has the highest priority, and Device #3 has the lowest priority, and within each device, the priority goes from Channel 0 to Channel 3.

#### 14.15 Counter and Timer Applications

The CTC operations in the counter mode and the timer mode appear to be similar; this apparent similarity can cause confusion in applications. For example, either of the modes can be used to design a clock. Therefore, it is necessary to discuss the differences between these two operations.

**FIGURE 14.6**  
CTCs Connected in Daisy Chain Format



The counter mode is used to count external events indicated by the CLK/TRG pulse. When an external circuit causes the CLK/TRG pin to go active, the down-counter in the CTC is decremented. Thus, the down-counter continues to count the events until it reaches zero, and then the output (ZC/TO) pulsed high indicates the end of the count. The down-counter is automatically loaded again, and the next cycle of counting continues.

On the other hand, the timer mode is used to provide time delays and is based on the internal clock (CLK). However, the clock frequency is divided by a number called prescaler (16 or 256), which is specified in the control word. Thus the down-counter is decremented every 16th or 256th clock pulse. When the down-counter reaches zero, an output pulse is generated, similar to that in the counter mode, and the counter is loaded again for the next time delay.

The CTC can be used in the timer mode to design a clock to indicate the time of the day; the accuracy is determined primarily by the system clock. Similarly, the CTC can be used in the counter mode to count pulses from a 60 Hz power line, and a clock can be designed as illustrated in Section 14.3.

---

## ILLUSTRATION: DESIGNING A BAUD (RATE) GENERATOR USING THE CTC IN THE TIMER MODE

---

# 14.2

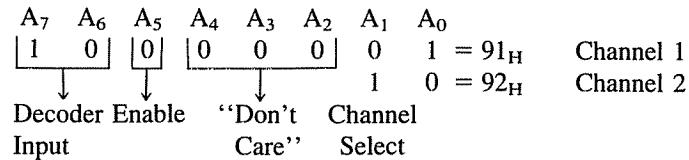
A typical application of the CTC in the timer mode is a programmable **baud (rate) generator** for serial I/O data communication (see Chapter 15). The baud generator is a frequency generator that provides a pulse at a predetermined frequency. In serial I/O, data bits are generally transmitted from 110 bits to 9,600 bits per second. For example, to send data over telephone lines, the transmission rate ranges from 300 to 2,400 bits per second. The clock frequency for serial I/O circuitry is generally 16 to 64 times the transmission rate. This illustration is concerned with designing a baud generator using the CTC.

### 14.21 Problem Statement

Design a programmable baud (frequency) generator using the CTC to provide two frequencies:  $300 \times 16$  (= 4.8 kHz) and  $1200 \times 16$  (= 19.2 kHz); the system clock is 3.6864 MHz. Identify the port addresses of Channels 1 and 2 and program Channel 1 to generate 4.8 kHz and Channel 2 for 19.2 kHz.

### 14.22 Problem Analysis

The port addresses of the channels can be obtained by analyzing the decoding logic in Figure 14.7. The circuit uses the 74LS139 2-to-4 decoder; it has address lines  $A_7$  and  $A_6$  as inputs,  $A_5$  as the enable line (active low), and lines  $A_4$ ,  $A_3$ , and  $A_2$  as “don’t care.” The address lines  $A_1$  and  $A_0$  are connected to the CTC, and they determine the channel selection. Assuming the “don’t care” lines at logic 0, Channel 1 can be accessed with the port address  $91_H$  and Channel 2 with the port address  $92_H$ , as shown.



To program the channels for the specified frequencies, we need to initialize the channels in the timer mode and calculate the time constant; this step is similar to Example 14.2. Let us assume the prescaler is 16; for Channel 1, the output or the baud (rate) should be 4.8 kHz.

$$\text{Baud} = \frac{\text{System Clock Frequency}}{\text{Prescaler} \times \text{Time Constant}} = \frac{f_s}{16 \times T_c}$$

$$4.8 \text{ kHz} = \frac{3.6864 \text{ MHz}}{16 \times T_c} \quad T_c = 48$$

Similarly, the time constant for 19.2 kHz = 12.

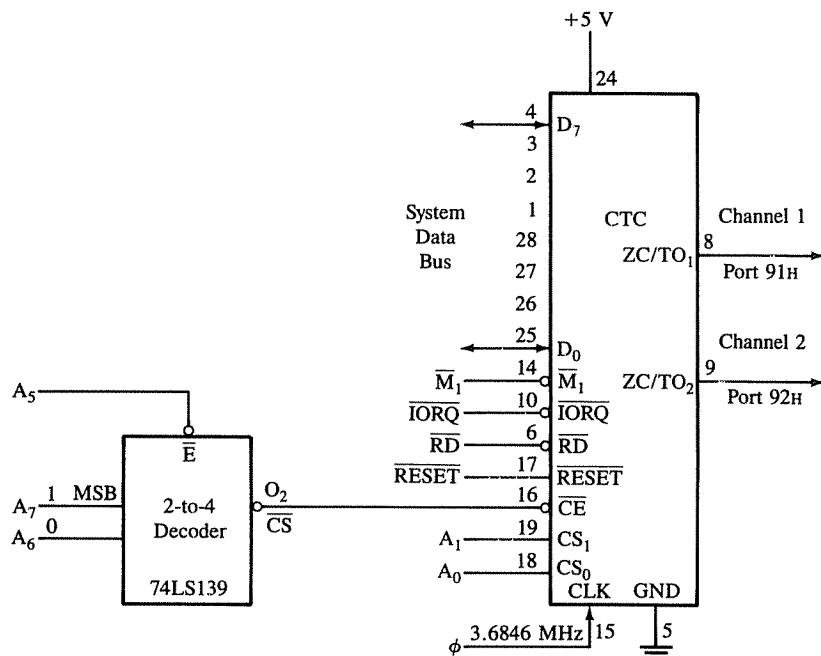
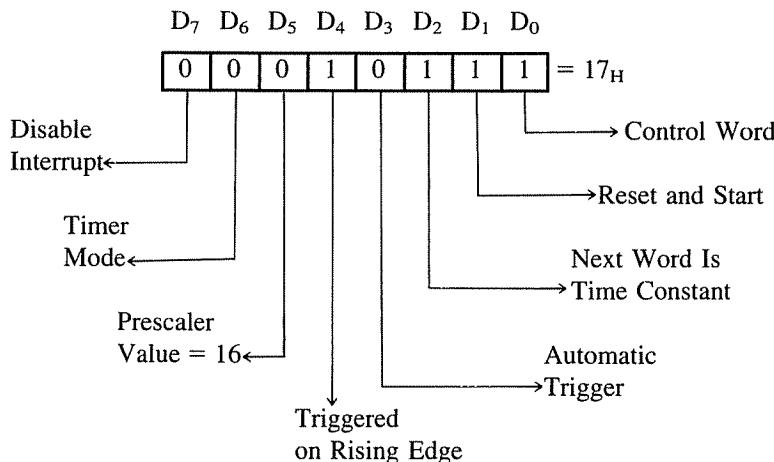


FIGURE 14.7

Schematic: Baud Generator Using CTC in the Timer Mode

The channel control word for both channels is  $17_H$  as shown in the following diagram; this is a continuous operation without the interrupt capability.



**Instructions** The following instructions will set up the CTC in the timer mode: Channel 1 for 4.8 kHz output and Channel 2 for 19.2 kHz.

```

CNTRL EQU 17H      ;Defines channel control word
COUNT1 EQU 30H      ;Time constant 48 for Channel 1
COUNT2 EQU 0CH      ;Time constant 12 for Channel 2
CTC1 EQU 91H        ;Port address of Channel 1
CTC2 EQU 92H        ;Port address of Channel 2
SETUP: LD A, CNTRL ;Channel control word
       OUT (CTC1), A ;Send control word to Channel 1
       LD A, COUNT1   ;Time constant for Channel 1
       OUT (CTC1), A ;Load time constant into Channel 1
       LD A, CNTRL   ;Channel control word
       OUT (CTC2), A ;Send control word to Channel 2
       LD A, COUNT2   ;Time constant for Channel 2
       OUT (CTC2), A ;Load time constant into Channel 2
       HALT
    
```

**Description** The first two instructions write the channel control word into Channel 1 (Port  $91_H$ ), and it must be followed by the time constant (COUNT1 =  $30_H$ ). This operation is repeated for Channel 2 by the subsequent instructions. When the time constant is loaded, it resets the channel and begins the countdown at the rising edge of the system clock. When the count reaches zero, a high pulse equal to 1.5 times ( $0.41 \mu s$ ) the system clock period is generated at the output (ZC/TO = Zero Count/Time Out) of the channel; thus, Channel 1 provides a 4.8 kHz clock and Channel 2 provides a 19.2 kHz clock.

# 14.3

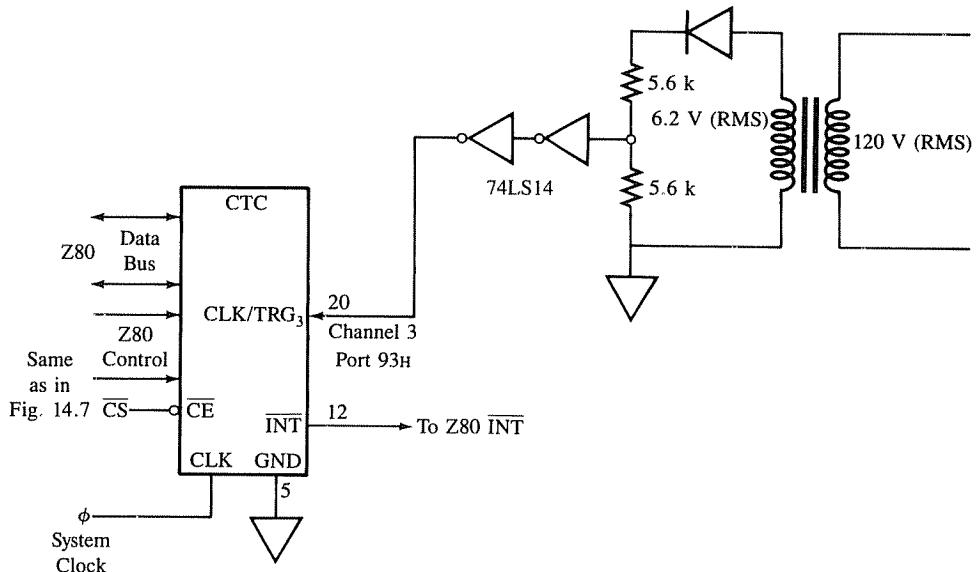
## ILLUSTRATION: USING THE CTC IN THE COUNTER MODE WITH INTERRUPT

In the counter mode, the CTC counts external events whenever the input signal at the CLK/TRG pin of a channel goes active. The CLK/TRG pin can be activated by either a leading edge or trailing edge pulse input; it is specified by the channel control word. The channel operation is similar to that in the timer mode; an 8-bit count is loaded into the channel register, and the count is decremented whenever the CLK/TRG input goes active. When the count reaches zero, the output ZC/TO goes active for approximately 1.5 times the clock period and the count is reloaded into the register.

This illustration concerns designing a clock by counting a 60 Hz power line, which provides an accurate time base.

### 14.31 Problem Statement

Design a minute timer using a 60 Hz powerline as an external trigger to the CLK/TRG pin of Channel 3 as shown in Figure 14.8. The CTC should interrupt the Z80 MPU every second to update the seconds display, and at the end of 60 minutes the clock should be reset and start again. The vector address for Channel 3 = 2056<sub>H</sub>.



**FIGURE 14.8**  
Schematic: CTC in the Counter Mode

### 14.32 Problem Analysis

This problem has three parts:

1. Getting an appropriate pulse from the 60 Hz power line.
2. Setting up the CTC in the counter mode with the interrupt capability.
3. Writing subroutines to upgrade the displays of seconds and minutes.

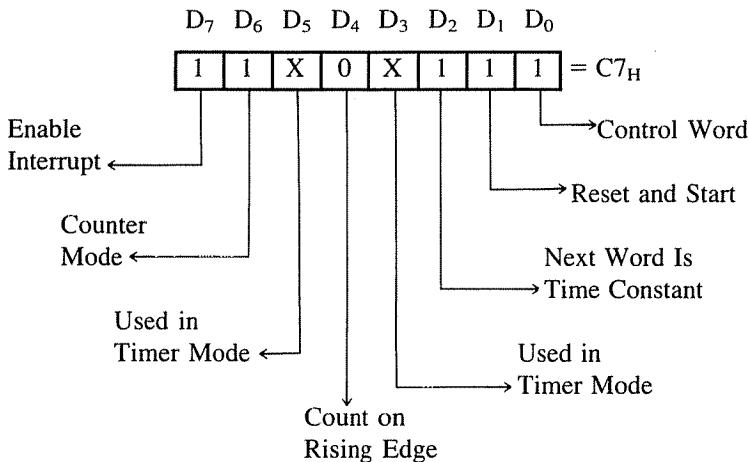
The AC power line provides 120 V (RMS) with 60 Hz frequency. Thus, it can provide a signal with a 16.6 ms period; however, the voltage should be converted into a 5 V pulse to be compatible with TTL logic. Figure 14.8 shows a step-down transformer with a rectifier; the output of the rectifier will be approximately + 10 V, and the resistor divider network adjusts the output to + 5 V. The inverters are used as a wave-shaping circuit to convert the sine wave into a square wave pulse. The output of the inverters will be a 5 V square wave with a 16.6 ms period, connected to the CLK/TRG pin of the CTC Channel 3. This pulse triggers the CTC channel 60 times per second.

The CTC channel counter should be set up to count these pulses, and at the end of the count, the CTC should generate an interrupt. Thus, the interrupts are generated every second, and the interrupt service subroutine should upgrade the seconds display.

### 14.33 Initializing the CTC in the Counter Mode

To initialize the CTC in the counter mode with interrupt capability, we need to send three words: channel control word and count (time constant) to Channel 3, and interrupt vector to Channel 0.

The channel control word is as follows (assuming “don’t care” bits at logic 0):



The time constant = 3C<sub>H</sub> to count 60 pulses. The low-order vector address = 50<sub>H</sub> for Channel 0; the high-order address is supplied by the interrupt register I (see Figure 14.5).

### 14.34 Program

In this illustration, the program can be divided into three sections: CTC initialization, the main program to display minutes and seconds, and the interrupt service routine to update the timer registers.

**CTC Initialization** The following instructions are written as a subroutine to set up the CTC Channel 3 in the counter mode with interrupt enable.

```

CNTRL EQU C7H      ;Defines channel control word
COUNT3 EQU 3CH      ;Time constant 60 for 1 second
LOVECT EQU 50H      ;Low-order address of interrupt vector
       CTC3 EQU 93H      ;Port address of Channel 3
       CTC0 EQU 90H      ;Port address of Channel 0.
SETCTC: LD A, CNTRL    ;Channel control word
        OUT (CTC3), A   ;Send control word to Channel 3
        LD A, COUNT3    ;Time constant to count 60 pulses
        OUT (CTC3), A   ;Load time constant into Channel 3
        LD A, LOVECT    ;Low-order interrupt vector
        OUT (CTC0), A   ;Load interrupt vector address into Channel 0
        RET

```

**Main Program** The main program initializes the stack pointer, the interrupt register, and registers for minutes and seconds. It calls the SETCTC subroutine, enables the Z80 interrupt, and stays in the display loop.

```

START: LD SP, STACK    ;Initialize stack pointer
       IM 2           ;Set up Z80 in interrupt Mode 2
       LD A, 20H        ;Load high-order byte of interrupt vector
       LD I, A          ;Load interrupt register
       CALL SETCTC     ;Initialize CTC
       LD BC, 0000      ;Set B for minutes and C for seconds
       EI              ;Enable Z80 interrupts
DISPLAY: LD A, B        ;Display minutes
        OUT (PORT1), A
        LD A, C          ;Display seconds
        OUT (PORT2), A
        JP DISPLAY

```

**Interrupt Service Routine** This routine is concerned primarily with updating the registers for seconds and minutes and decimal adjusting the values in the registers for BCD display. When register C reaches 60 seconds, it clears the register and increments register B. Register B is incremented until it reaches 60 minutes, and then the timer is reset to start again.

```

    TIMER: PUSH AF      ;Save (A) (F)
            LD A, C      ;Get previous reading
            ADD 01H       ;Update seconds
            DAA          ;Decimal adjust seconds
            LD C, A      ;Save BCD value of seconds
            CP 60H       ;Is time = 60 seconds?
            JR NZ, GOEND ;If not, go to end and return
            LD C, 00      ;If yes, clear seconds
            LD A, B      ;Get previous minutes
            ADD A, 01H     ;Update minutes
            DAA          ;Decimal adjust minutes
            LD B, A      ;Save BCD value of minutes
            CP 60H       ;Are minutes = 60?
            JR NZ, GOEND ;If not, go to end and return
            LD B, 00      ;If yes, clear minutes

    GOEND: POP AF
           EI
           RETI

```

### PROGRAM DESCRIPTION

The main program initializes the CTC Channel 3 and remains in the DISPLAY loop displaying the contents of registers B and C at output ports. The CTC channel register is loaded with the count 60, and whenever the powerline source triggers the CLK/TRG pin, the CTC register is decremented. When the register reaches zero, the interrupt request ( $\overline{INT}$ ) goes active and interrupts the Z80; thus, the Z80 is interrupted every second. Channel 3 does not have a ZC/TO output signal; therefore, the interrupt request  $\overline{INT}$  must be used to indicate that the register has reached zero.

When the Z80 acknowledges the interrupt request, the CTC supplies the low-order address ( $56_H$ ) of the interrupt vector, and it is combined with the high-order address ( $20_H$ ) from the Z80 interrupt register. The program execution is transferred to location  $2056_H$ , where the address of the service routine is stored in two consecutive memory locations ( $2056_H$  and  $2057_H$ ), and then the program is transferred to the service routine TIMER.

The service routine increments the seconds in register C, adjusts the value for BCD, and checks whether the number has reached 60. If it has not, the routine jumps to the end to enable the interrupt and returns to the main program. When the instruction RETI (Return from Interrupt) is executed, it is recognized by the CTC, which clears the  $\overline{INT}$  signal. When register C does eventually reach 60, the routine clears register C and increments the minutes in register B. When register B reaches the count of 60 minutes, register B is cleared, and the timer is reset to start all over.

---

### THE 8253 PROGRAMMABLE INTERVAL TIMER

14.4

The Intel 8253 is another widely used general purpose programmable interval timer/counter, and it is in many ways similar to the Z80 CTC. The 8253 includes three identical

16-bit counters which can operate independently in any one of six modes (described later). It is packaged in a 24-pin DIP and requires a single +5 V power supply. For operation as a counter, a 16-bit count is loaded in its register, and on command, it begins to decrement the count until it reaches zero. At the end of the count, it generates a pulse which can be used to interrupt the MPU. The counter can count in either binary or BCD. In addition, a count can be read by the MPU while the counter is decrementing.

#### 14.41 Block Diagram of the 8253

Figure 14.9 shows the block diagram of the 8253; it includes three 16-bit counters (0, 1, and 2). Each counter has two input signals—clock (CLK) and GATE—and one output signal—OUT. GATE can be used to initiate, enable, or disable counting. The diagram also shows three blocks: data bus buffer, Read/Write control logic, and a control word register.

**Data Bus Buffer** This is a tri-state 8-bit, bidirectional buffer connected to the data bus of the MPU.

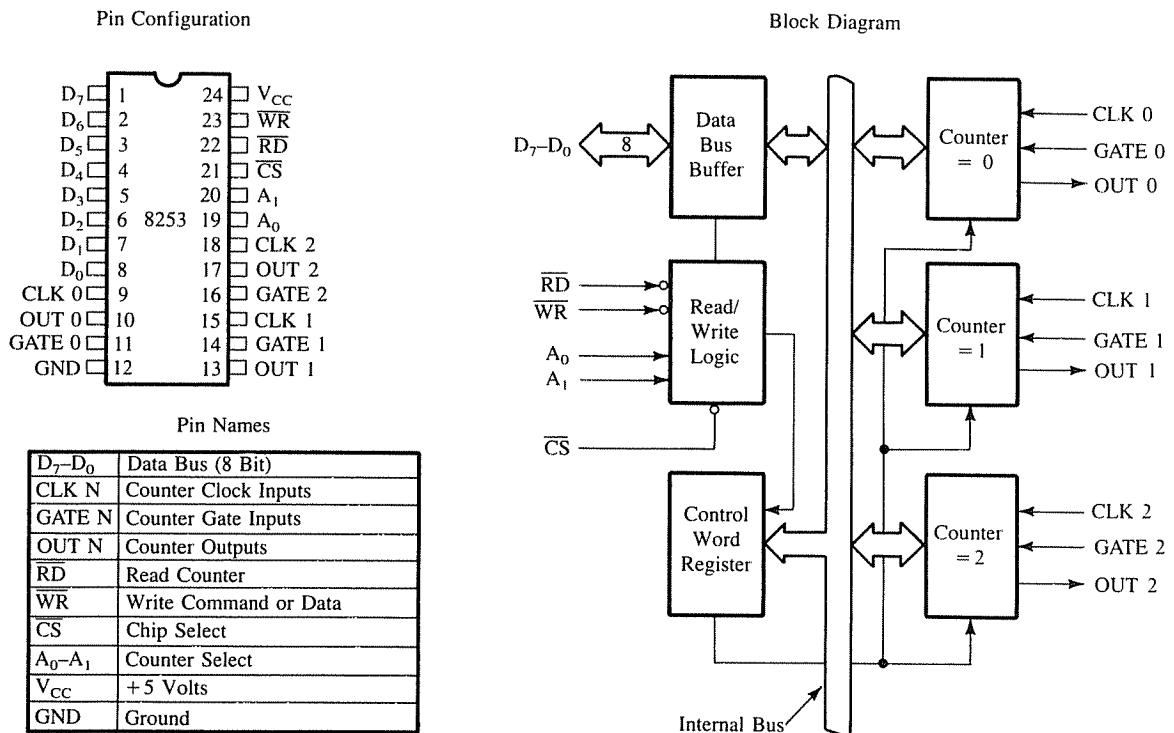


FIGURE 14.9  
8253 Block Diagram  
SOURCE: Reprinted by permission of Intel Corporation, copyright 1979.

**Control Logic** This control section has five signals:  $\overline{RD}$  (Read),  $\overline{WR}$  (Write),  $\overline{CS}$  (Chip Select), and the address lines  $A_0$  and  $A_1$ . In the peripheral I/O mode, the RD and WR signals are connected to  $\overline{IORD}$  and  $\overline{IOWR}$ , respectively. In memory-mapped I/O, these are connected to  $\overline{MEMRD}$  (Memory Read) and  $\overline{MEMWR}$  (Memory Write). Address lines  $A_0$  and  $A_1$  of the MPU are usually connected to lines  $A_0$  and  $A_1$  of the 8253, and  $\overline{CS}$  is tied to a decoded address.

The control word register and counters are selected according to the signals on lines  $A_0$  and  $A_1$ , as shown.

$A_1$	$A_0$	Selection
0	0	Counter 0
0	1	Counter 1
1	0	Counter 2
1	1	Control Word Register

**Control Word Register** This register is accessed when lines  $A_0$  and  $A_1$  are at logic 1. It is used to write a command word which specifies the counter to be used, its mode, and either Read or Write operation. However, the control word register is not available for a Read operation. The control word format is shown in Figure 14.10.

**Mode** The 8253 can operate in six different modes, as shown in Figure 14.11. The gate of a counter is used either to disable or enable counting, as shown in Figure 14.12.

#### 14.42 Programming the 8253

The 8253 can be programmed to provide various types of outputs (Figure 14.11) through Write operations, or to check a count while counting through Read operations. The details of these operations are given below.

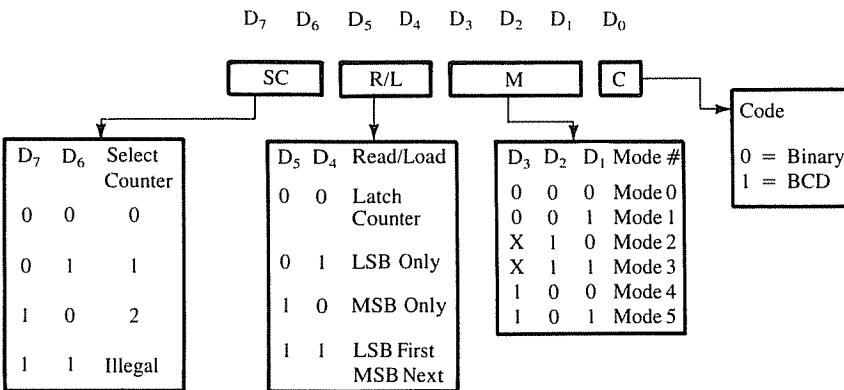
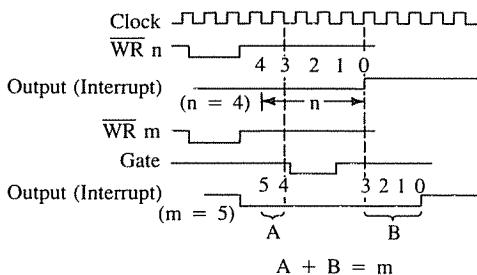


FIGURE 14.10

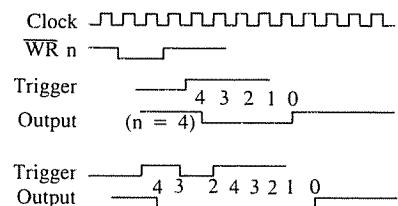
8253 Control Word Format

SOURCE: Adapted from Intel Corporation, copyright 1978.

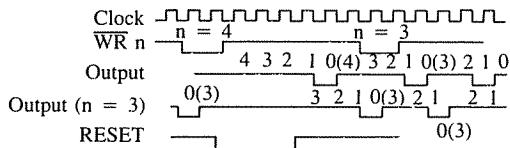
Mode 0: Interrupt on Terminal Count



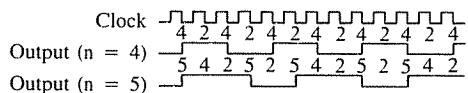
Mode 1: Programmable One-Shot



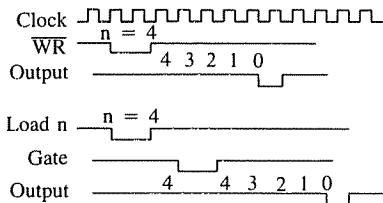
Mode 2: Rate Generator Clock



Mode 3: Square Wave Generator



Mode 4: Software Triggered Strobe



Mode 5: Hardware Triggered Strobe

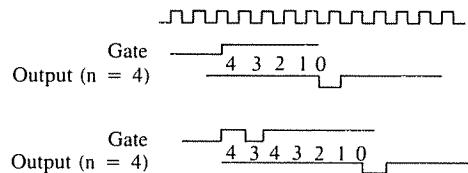


FIGURE 14.11

8253 Operating Modes

SOURCE: Reprinted by permission of Intel Corporation, copyright 1981.

**Write Operations** To initialize a counter, the following steps are necessary.

1. Write a control word into the control register.
2. Load the low-order byte of a count into the counter register.
3. Load the high-order byte of a count into the counter register.

With a clock and an appropriate gate signal to one of the counters, the above instructions should be able to start the counter and provide appropriate output according to the control word.

**Read Operations** In some applications, especially in event counters, it is necessary to read the value of the count in progress. This can be done by one of two methods. One

Modes	Signal Status	Low or Going Low	Rising	High
0		Disables counting	—	Enables counting
1		—	(1) Initiates counting (2) Resets output after next clock	—
2		(1) Disables counting (2) Sets output immediately high	(1) Reloads counter (2) Initiates counting	Enables counting
3		(1) Disables counting (2) Sets output immediately high	Initiates counting	Enables counting
4		Disables counting	—	Enables counting
5		—	Initiates counting	—

**FIGURE 14.12**

Gate Settings of a Counter

SOURCE: Reprinted by permission of Intel Corporation, copyright 1981

method involves reading a count after inhibiting (stopping) the counter to be read. The second method involves reading a count while counting is in progress (reading on the fly).

In the first method, counting is stopped (or inhibited) by controlling the gate input or the clock input of the selected counter, and two I/O Read operations are performed by the MPU. The first I/O operation reads the low-order byte, and the second reads the high-order byte.

In the second method, an appropriate control word is written into the control register to latch a count in the output latch, and two I/O Read operations are performed by the MPU.

### 14.43 The 8253 as a Square Wave Generator

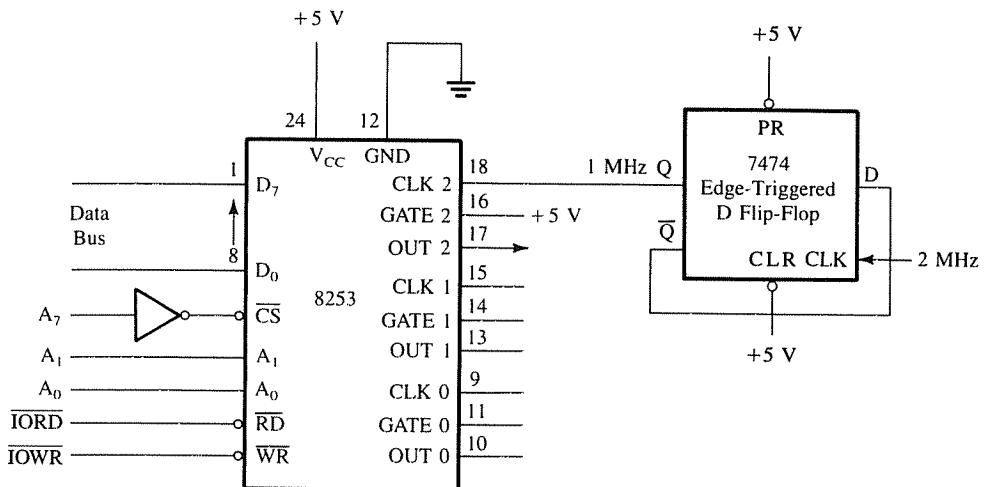
One of the attractive features of the 8253 is that it has several modes that can be used for various purposes whereas the CTC is primarily restricted to two modes: counter and timer. The following example illustrates how to set up the 8253 as a square wave generator.

1. Identify the port addresses of the control register and the counter 2 in Figure 14.13.
2. Calculate the count necessary to obtain a 20 kHz square wave if the clock frequency is 1 MHz and the counter is set up in Mode 3.
3. Write instructions to initialize Counter 2 in Mode 3 to obtain a 20 kHz square wave.

---

**Example**  
**14.4**


---



**FIGURE 14.13**  
Schematic: Interfacing the 8253

**Solution**

### 1. Port Addresses

The Chip Select is enabled when  $A_7 = 1$  (see Figure 14.13), and the control register is selected when  $A_1$  and  $A_0 = 1$ . Similarly, Counter 2 is selected when  $A_1 = 1$  and  $A_0 = 0$ . Assuming that the unused address lines  $A_6$  to  $A_2$  are at logic 0, the port addresses will be as follows:

$$\begin{aligned} \text{Control Register} &= 83_{\text{H}} \\ \text{Counter 2} &= 82_{\text{H}}. \end{aligned}$$

### 2. Count to Generate 20 kHz in Mode 3

The system clock frequency is 2 MHz, which is divided by 2 using a D flip-flop. The output of the flip-flop changes on the rising edge; therefore, the output frequency fed to CLK 2 is 1 MHz, with the period of 1  $\mu\text{s}$  (see Figure 14.14).

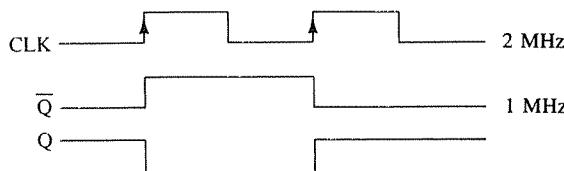
In Mode 3, the output remains high for half the count and low for the remaining half of the count. This is accomplished by decrementing the count by two at every falling edge of the clock. Therefore, with 1 MHz clock frequency, the count will be decremented by two every microsecond. To obtain the square wave with 20 kHz frequency (50  $\mu\text{s}$  period), the output should remain high for 25  $\mu\text{s}$  and low for 25  $\mu\text{s}$ .

$$\frac{\text{Count} \times \text{Clock Period}}{2} = \text{Half Period of Square Wave}$$

$$\text{Count} = 2 \times \frac{25 \mu\text{s}}{1 \mu\text{s}} = 50$$

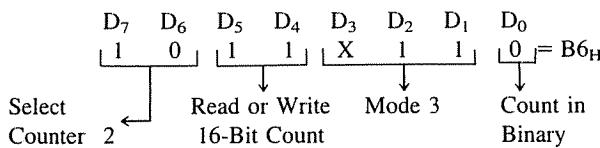
**FIGURE 14.14**

Clock Frequency Waveform



### 3. Control Word and Instructions to Initialize the Counter

To initialize the 8253 for Counter 2 in Mode 3, the following control word is necessary.



#### Instructions

COUNTR:	LD A, 10110110B	;Control word (B6H) to initialize Counter 2.
	OUT (83H), A	;Write control word in control register
	LD A, 50	;Low-order byte (50 = 32H) of the count
	OUT (82H), A	;Load Counter 2 with the low-order byte
	LD A, 00	;High-order byte of the count
	OUT (82H), A	;Load Counter 2 with the high-order byte

**Description** The first instruction loads the control word into the control register of the 8253 to set up Counter 2 in Mode 3. The subsequent instructions load the count 0032H (50<sub>10</sub>) to obtain a 20 kHz square wave. In Figure 14.13, the gate is tied high; therefore, the counter begins as soon as the count is loaded.

### 14.44 Comparison of Z80 CTC and Intel 8253

The Intel 8253 is a general purpose programmable timer/counter. It includes three 16-bit counters. Its control logic requires RD/WR control signals and the decoded address. Each counter can have its own independent clock, and the counter operation can be controlled using the gate input. The 8253 can operate in six different modes to provide various types of outputs, such as a single pulse at the end of the count or a square wave output.

On the other hand, the CTC is specially designed to work with the Z80; it requires M<sub>1</sub> and IORQ control signals in addition to RD signal. The CTC has four independent channels with an 8-bit time constant register. The counting is initiated by software instructions; it does not have a gate input signal. It can operate only in two modes: counter and timer. However, the CTC has a very powerful interrupt scheme; it can generate an interrupt request signal at the end of the count and set up the daisy chain priority scheme.

## SUMMARY

---

Programmable timer/counter circuits are designed to provide accurate time delays and to count external events. These integrated circuits are used in various applications such as time delays, counters, one-shot, and waveform generation. This chapter has been concerned with applications of programmable timer/counter devices, specifically, the Z80 Counter/Timer Circuit (CTC). The CTC was described in detail, and it was illustrated with two applications: baud generator and timer (clock) design using a 60 Hz power line signal. In addition, the Intel 8253 timer was described and compared with the CTC. A summary of the important features of the programmable timer circuits is as follows.

- A programmable timer/counter device generally includes multiple timer/counter circuits and can operate in various modes.
- The control logic of the timer includes a control register and a time constant register. The operation mode and the selection of a timer is specified by writing a control word into the control register, and an appropriate count is loaded into the time constant register.
- The Z80 CTC has two modes of operation: the timer mode and the counter mode.
- In the timer mode, the CTC provides the time delay based on the frequency of the system clock, the prescaler specified in the control word, and the time constant loaded into the counter register. After the time constant (count) is loaded, the counter can be initiated either automatically or by an external pulse. Once the operation is initiated, the down-counter is decremented at every clock pulse, and when it reaches zero, an active high pulse is generated at the output. The count is automatically loaded again, and the operation continues.
- In the counter mode, the CTC counts external events indicated by the input from an external circuit. In this mode, the counter is decremented whenever it receives an active pulse from the external event, and when the counter reaches zero, an active high pulse is generated at the output and the count is automatically reloaded into the counter register.
- The Z80 CTC is capable of generating an interrupt request pulse. If the interrupt is enabled, the CTC generates an interrupt request pulse when the count reaches zero. The CTC also includes interrupt control logic to set up several CTCs in a daisy chain priority.
- The Intel 8253 is also a widely used general purpose timer/counter circuit. It has three 16-bit counters that can operate independently in six different modes.

## ASSIGNMENTS

---

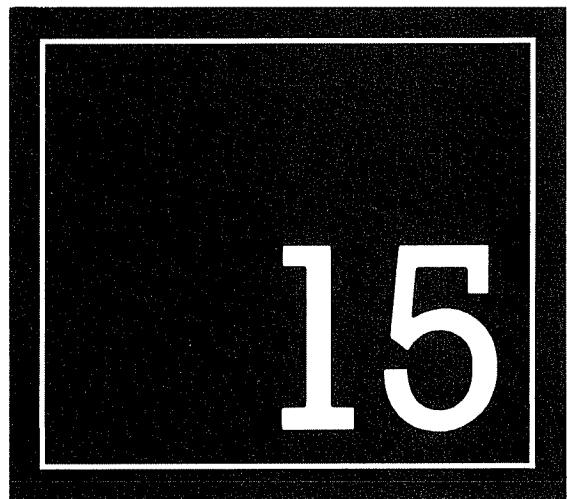
1. Explain the operations of the Z80 CTC in the counter mode and the timer mode.
2. What is the function of the prescaler when the CTC is set up in the timer mode?
3. Calculate the time delay for the following parameters: System clock frequency = 4 MHz, prescaler = 256, and the time constant = 150.
4. Calculate the time constant to obtain a pulse every 400  $\mu$ s if the system clock is 4 MHz and the prescaler value is 16.
5. Identify the port addresses of Channel 1 and Channel 2 in Figure 14.2 if address lines A<sub>7</sub> and A<sub>4</sub> are interchanged.
6. Specify the control word for Channel 1 (Figure 14.7) to set up the CTC in the timer mode with the automatic trigger on the falling edge and the interrupt disabled.
7. Write instructions to initialize Channel 2 (Figure 14.7) in the timer mode to provide a pulse approximately every 5 ms.
8. Specify the control word to set up Channel 2 in the counter mode with the interrupt enabled.
9. Write instructions to set up Channel 2 (Fig. 14.8) in the counter mode to count 120 events. At the end of 120 events, the CTC should generate an interrupt request. After the interrupt acknowledge, the program should be transferred to the location 209F<sub>H</sub>, where the service routine is located. The interrupt vector address for Channel 0 is specified as 2048<sub>H</sub>.
10. Show output connections of Channel 1 and Channel 2 (Fig. 14.8) to count 1000 events. Initialize the channels with appropriate time constants.
11. Design a five minute timer using two channels of the Z80 CTC. Assuming the system clock to be 2 MHz, set up one channel as a timer and another as a counter. Connect the output of the timer to CLK/TRG input of the counter. The counter channel should generate an interrupt request every second, and the service routine should count the seconds until the total time is five minutes.
12. Write initialization instructions to set up Channel 0 as a timer with the prescaler equal to 16 and the time constant equal to 256. If the clock frequency is 4 MHz, calculate the delay obtained at the output of Channel 0.
13. Set up Channel 1 as a counter with the maximum count (256) and trigger the CLK/TRG1 from the output of Channel 0 in Assignment 12. Calculate the delay interval at ZC/TO1.
14. Set up Channels 2 and 3 as counters and generate an interrupt after one hour by using the output of Assignment 13.



# Serial I/O and Data Communication

The Z80 microprocessor is a parallel device; it transfers data bits simultaneously over its eight data lines. This is called the **parallel I/O mode**, as discussed in previous chapters. However, in many situations, the parallel I/O mode is either impractical or impossible. For example, parallel data communication over a long distance can become very expensive. Similarly, devices such as a CRT terminal and a cassette tape are not designed for parallel I/O. In these instances, the **serial I/O mode** is used, whereby one bit at a time is transferred over a single line.

In *serial transmission* (from the MPU to a peripheral), an 8-bit parallel word should be converted into a stream of eight serial bits; this is known as **parallel-to-serial conversion**. After the conversion, one bit at a time is transmitted over a single line at a given rate called the **baud** (bits per second). In *serial reception*, on the other hand, the MPU receives a stream of eight bits, and they should be converted into an 8-bit parallel word; this is known as **serial-to-parallel conversion**. In addition to the conversion, information such as the beginning and the end of transmission and error check is necessary in serial communication. This chapter first discusses these basic concepts in serial data communication and explains how serial communication can be implemented using microprocessor instructions



(software). However, in industrial applications, the hardware approach through programmable devices is generally used. This chapter illustrates applications with two such devices: the Intel 8251 and the Z80 SIO (and DART).

## OBJECTIVES

- Explain how data transfer occurs in the serial I/O mode and how it differs from the parallel I/O mode.
- Explain the terms: synchronous and asynchronous transmission; simplex, and half and full duplex transmission; baud, and parity check.
- Explain how data bits are transmitted (or received) in the asynchronous format, and calculate the delay required between two successive bits for a given baud.
- Explain the RS-232C serial I/O standard and compare it with the RS-422 and -423 standards.
- Explain how serial I/O communication can be implemented in the asynchronous format using software.
- Explain the block diagram and the functions of each block of the Intel 8251 USART (Programmable Communication Interface).
- Design an interfacing circuit using the 8251, and write initialization instructions to set up data communication between a microcomputer and a serial peripheral.
- Explain the block diagram and the functions of each block of the Z80 SIO (Serial Input/Output Controller) and the DART (Dual Asynchronous Receiver and Transmitter).
- Write initialization instructions to set up the Z80 SIO (or DART) for given specifications to implement the asynchronous communication.
- Write interrupt service routines to implement communication between the Z80 MPU and a terminal when the Z80 SIO is set up in the interrupt mode.

# 15.1 BASIC CONCEPTS IN SERIAL I/O

---

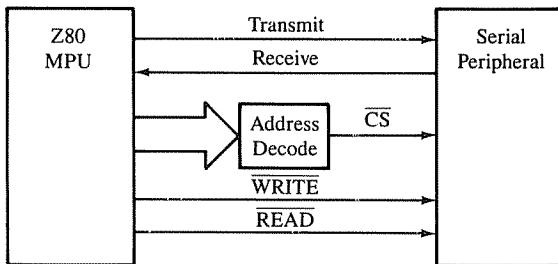
The basic concepts concerning the serial I/O mode can be classified into the following categories.

1. Interfacing requirements
2. Serial I/O format and requirements
3. Error checks in data communication
4. Data communication over long distance
5. Standards in serial I/O
6. Software versus programmable hardware approaches

### 15.11 Interfacing Requirements

The interfacing requirements for a serial I/O peripheral are the same as those of a parallel I/O device. The microprocessor identifies the peripheral through a port address and enables it using Read or Write control signal. The primary difference between the parallel I/O and the serial I/O is in the number of lines used for data transfer—the parallel I/O uses the entire data bus and the serial I/O uses only one data line. Figure 15.1 shows a typical configuration of serial I/O transmission; the MPU selects the peripheral through Chip Select and uses the control signals READ to receive data and WRITE to transmit data. The serial peripheral can be interfaced either under program control (status check) or interrupt control.

**FIGURE 15.1**  
Block Diagram: Serial I/O Interfacing



## 15.12 Serial I/O Format and Requirements

The serial transmission format is concerned with such issues as synchronization between a receiver and a transmitter, direction of data flow, and rate or speed of transmission. These topics are briefly described below. Topics concerning errors in transmission and data communication over long distances are discussed in the subsequent sections.

**Synchronous versus Asynchronous Transmission** Serial communication occurs in one of two formats: synchronous or asynchronous. In the synchronous format, a receiver and a transmitter are synchronized with the same frequency, and a block of characters is transmitted along with the synchronization (Sync) characters, as shown in Figure 15.2(a). Error check characters called CRC (Cyclic Redundancy Check—discussed later), are also included. This format is generally used for high speed transmission (more than 20 k bits/second).

The asynchronous format is character-oriented as shown in Figure 15.2(b). The asynchronous transmission, as the name suggests, can occur any time; it is unpredictable in relation to time. Therefore, each character must carry information about when the transmission begins and when it ends; this information is included in each character by adding the Start and the Stop bits (Figure 15.2(b)). When no data are being transmitted, a receiver stays in the high state. Transmission begins with a low Start bit, followed by a character and one or two high Stop bits. This is also known as **framing**. Figure 15.2(b) shows the transmission of 11 bits for an ASCII character in the asynchronous format: one Start bit, eight character bits, and two Stop bits. In serial I/O, logic 1 is known also as **Mark** and logic 0 as **Space**. The format shown in Figure 15.2(b) is similar to Morse Code, but dots and dashes are replaced by logic 0s and 1s. The asynchronous format is generally used in low speed transmission (less than 20 k bits/second).

**Simplex and Duplex Transmission** Serial communication can also be classified according to the direction and simultaneity of data flow.

In **simplex transmission**, data are transmitted in only one direction. An example is the transmission from a microcomputer to a printer.

In **duplex transmission**, data flow in both directions. However, if the transmission is one way at a time, it is called *half duplex*; if it is both ways simultaneously, it is called

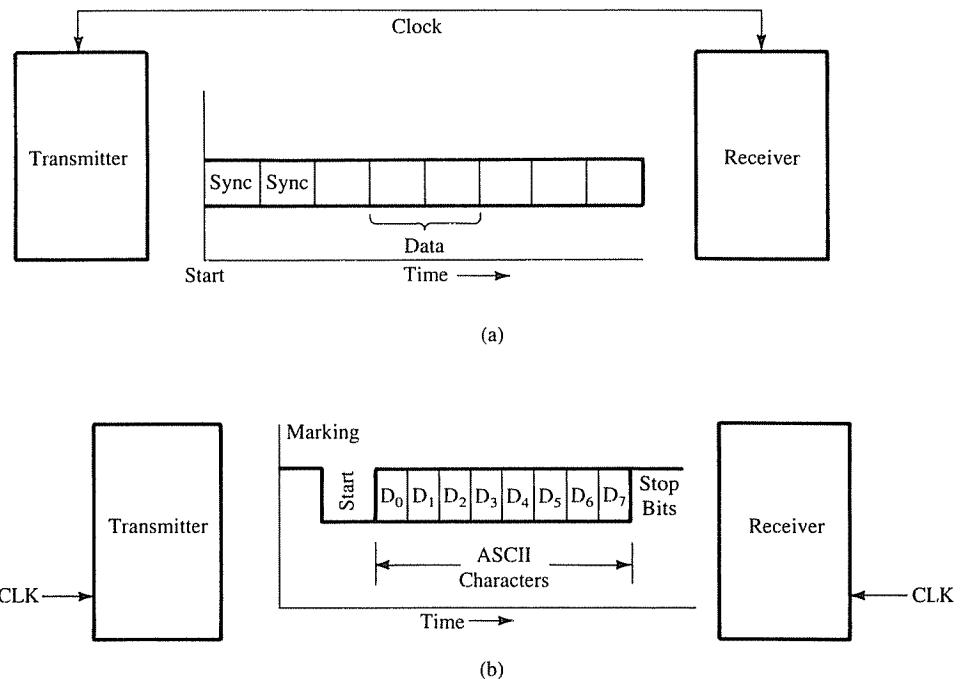


FIGURE 15.2  
Transmission Format: (a) Synchronous (b) Asynchronous

*full duplex*. Generally, transmission between two computers or between a computer and a terminal is full duplex.

**Rate of Transmission** In parallel I/O, data bits are transferred when a control signal enables the interfacing device; the transfer takes place in less than three T-states. However, in serial I/O, one bit is sent out at a time; therefore, how long the bit stays on or off is determined by the speed at which the bits are transmitted. Furthermore, the receiver should be set up to receive the bits at the same rate as the transmission; otherwise, the receiver may not be able to differentiate between two consecutive 0s or 1s.

The rate at which the bits are transmitted—bits/second—is called a **baud**; however, technically, it is defined as the number of signal changes/second. Each piece of equipment has its own baud requirement. For example, a teletype (TTY) generally runs on a 110 baud. However, in most terminals and printers, the baud is adjustable, typically ranging from 50 to 9,600 baud. Figure 15.3 shows how the ASCII character I ( $49_{16}$ ) will be transmitted with 1,200 baud with the framing information of one Start and two Stop bits. The transmission begins with active low Start bit, followed by the LSB D<sub>0</sub>. The bit time—the delay between any two successive bits—is 0.83 ms; this is determined by the baud as shown in Figure 15.3.

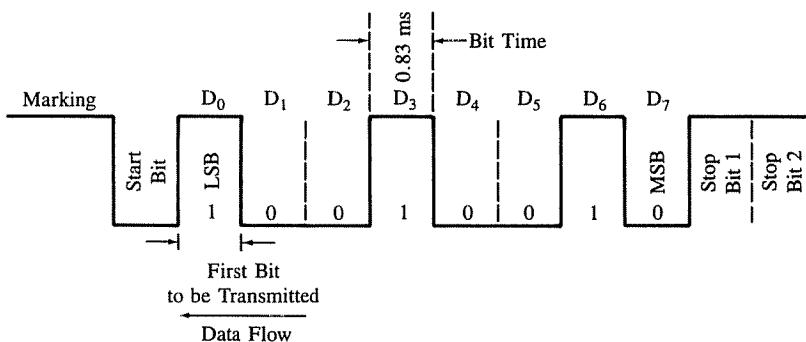


FIGURE 15.3

Serial Bit Format for ASCII Character "I" at 1,200 Baud

$$\begin{aligned} 1,200 \text{ bits} &= 1 \text{ second} \\ \text{For 1 bit} &= 1/1,200 = 0.83 \text{ ms} \end{aligned}$$

Therefore, to transmit one character, a parallel byte ( $49_{\text{H}}$ ) should be converted into a stream of 11 bits by adding framing bits (one Start and two Stop bits), and each bit must be transmitted at the interval of 0.83 ms. This can be implemented either through software or through programmable hardware chips. To receive a character in the serial mode, the process is reversed—bits are received one at a time and converted into a parallel word.

### 15.13 Error Checks in Data Communication

During transmission, various types of errors can occur. For example, data bits may change because of noise or can be misunderstood by the receiver because of differences in receiver and transmitter clocks. These errors need to be checked; therefore, additional information for error checking is sent during the transmission. The receiver can check the received data against the error check information, and if an error is detected, the receiver can request the retransmission of that data segment. Three methods used in common practice are **parity check**, **checksum**, and **cyclic redundancy check**.

#### PARITY CHECK

This is used to check each character by counting the number of 1s in the character, and in the ASCII code transmission, bit  $D_7$  is used to transmit parity check information. The technique is based on the principle that in a given system, each character is transmitted with either an even number or an odd number of 1s.

In an even parity system, when a character has an odd number of 1s, the bit  $D_7$  is set to 1 and an even number of 1s is transmitted. For example, the code for the character I is  $49_{\text{H}}$  (01001001) with three 1s. When the character I is transmitted in an even parity system, the transmitter will set the bit  $D_7$  to 1, making the code  $C9_{\text{H}}$  (1100 1001). On the

other hand, in an odd parity system, the character I is transmitted by keeping bit D<sub>7</sub> at 0; thus, the code remains 49<sub>H</sub>.

In the Z80 system, the parity check is easy to implement and detect because the Z80 has the parity flag, and this flag can be used to check parity information in each character. However, the parity check cannot detect multiple errors in any given character if the number of errors is even.

### CHECKSUM

The checksum technique is used when blocks of data are transferred. It involves adding all the bytes in a block without carries. Then, the 2's complement of the sum (negative of the sum) is transmitted as the last byte. The receiver adds all the bytes, including the 2's complement of the sum; thus, the result should be zero if there is no error in the block.

### CYCLIC REDUNDANCY CHECK (CRC)

This technique is commonly used when data are transferred from and to a floppy disk and in a synchronous data communication. The technique is based on mathematical relationships of polynomials. A stream of data can be represented as a polynomial that is divided by a constant polynomial called the generator polynomial. The remainder of the division is sent out as a check for errors. The receiver checks the remainder to detect an error in the transmission. The mathematical details are as follows.

1. A stream of data bits can be represented as

$$M(x) = b_nx^n + b_{n-1}x^{n-1} + \dots + b_0x^0, \text{ where}$$

$b_0$  = least significant bit

$b_n$  = most significant bit.

For example, the polynomial of the Hex number 8A<sub>H</sub> (1 0 0 0 1 0 1 0) is

$$\begin{aligned} M(x) &= 1x^0 + 0x^1 + 0x^2 + 0x^3 + 1x^4 + 0x^5 + 1x^6 + 0x^7 \\ &= 1x^0 + 1x^4 + 1x^6 = x^6 + x^4 + 1. \end{aligned}$$

2. Let us assume the length of the CRC code is four bits, although normally it is 16 bits. To obtain proper division, the polynomial is first multiplied by the power of the CRC code length (in our example it is  $x^4$ ) and divided by the agreed-upon generator polynomial  $G(x)$ . The formula is

$$\frac{M(x) \times x^4}{G(x)} = Q(x) + R(x)$$

where  $Q(x)$  is the quotient obtained by Modulo-2 arithmetic (see Appendix B for Modulo-2 arithmetic) and  $R(x)$  is the remainder.

3. Assuming  $G(x) = x^4 + 1$

$$\frac{M(x) \times x^4}{G(x)} = \frac{x^{10} + x^8 + x^4}{x^4 + 1} = (x^6 + x^4 + x^2) + x^2 \quad (\text{see Appendix B}).$$

Quotient  $Q(x)$       Remainder  $R(x)$

4. This remainder is added to the modified polynomial

$$M(x) \times x^4 + R(x) = x^{10} + x^8 + x^4 + x^2 \text{ and transmitted as}$$

	$x^{10}$				$x^8$				$x^4$				$x^2$				
	0	1	0	1	0	0	0	0	1	0	1	0	1	0	0	0	0
D <sub>0</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>	D <sub>5</sub>	D <sub>6</sub>	D <sub>7</sub>	D <sub>8</sub>	D <sub>9</sub>	D <sub>10</sub>	D <sub>11</sub>						
Data = 8A <sub>H</sub>										CRC							

The transmitted stream of bits includes the original byte 8A<sub>H</sub> in reverse order, appended by the CRC bits at the end.

5. The receiver divides the transmitted polynomial by  $G(x)$ , and if the remainder is 0, it indicates no error (divides  $x^{10} + x^8 + x^4 + x^2$  by  $x^4 + 1$  and checks the answer).

The CRC check is a somewhat complex technique; it is discussed briefly here because the Z80 SIO is capable of generating and checking the CRC code if the SIO is used in the synchronous mode.

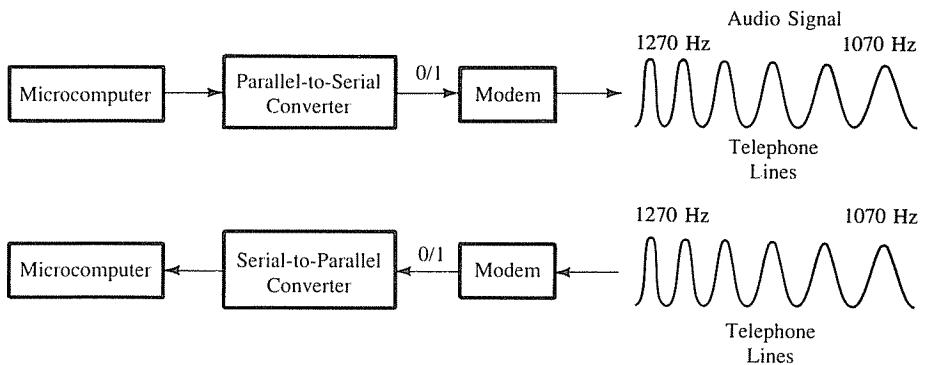
### 15.14 Data Communication Over Telephone Lines

The serial I/O technique can be used to send data over long distance through telephone lines. However, telephones lines are designed to handle voice; the bandwidth of telephone lines ranges from 300 Hz to 3,300 Hz, while a digital signal, with rise time in nanoseconds, requires a bandwidth of several megahertz. Therefore, data bits are converted into audio tones using modems.

A **modem** (Modulator/Demodulator) is a circuit that translates digital data into audio tone frequencies for transmission over telephone lines and converts audio frequencies into digital data for reception. The modems are designed to transfer data at rates of 300–2,400 bps (bits per second). Generally, two types of modulation techniques are used: *frequency shift keying* (FSK) for low-speed modems and *phase shift keying* (PSK) for high-speed modems.

Computers can exchange information over telephone lines by using two modems—one on each side (Figure 15.4). A calling computer (or terminal), also known as the originator, contacts a receiving (answering) computer through a telephone number, and a communication link is established after control signals have been exchanged between computers and modems.

A typical process of communication for a 300 bps modem is shown in Figure 15.4. A parallel word is converted into serial bits; in turn, the originator modem generates two audio frequencies — 1,070 Hz for logic 0 (Space) and 1,270 Hz for logic 1 (Mark). These audio frequencies are transmitted over telephone lines. At the answering end, audio frequencies are converted back into 0s and 1s, and serial bits are converted into a parallel word that can be read by the computer. When the answering computer needs to transmit, it transmits on 2,025 Hz (Space) and 2,225 Hz (Mark).



**FIGURE 15.4**  
Communication over Telephone Lines Using Modems

### 15.15 Standards in Serial I/O

The serial I/O technique is commonly used to interface terminals, printers, and modems. These peripherals and computers are designed and manufactured by various companies. Therefore, a common understanding must exist among various manufacturing and user groups that can ensure compatibility among different equipment. When this understanding is defined and generally accepted in industry (and by users), it is known as a standard. A standard is normally defined by a professional organization (such as IEEE—Institute of Electrical and Electronics Engineers); however, a widespread practice can occasionally become a de facto standard. A standard may include such items as assignment of pin positions for signals, voltage levels, speed of data transfer, length of cables, and mechanical specifications.

In serial I/O, data can be transmitted either as current or voltage. Typically, 20mA (or 60mA) current loops are used in teletype equipment. When a teletype is Marking or at logic 1, current flows; when it is at logic 0 (or Space), the current flow is interrupted. The advantage of the current loop method is that signals are relatively noise-free.

When data are transmitted as voltage, the commonly used standard is known as **RS-232C**. It is defined in reference to Data Terminal Equipment (DTE) and Data Communication Equipment (DCE)—terminal and modem—as shown in Figure 15.5(a); however, its voltage levels are not compatible with TTL logic levels. The rate of data transmission in RS-232C is restricted to a maximum of 20 kbaud and the distance is limited to 50 ft. For high-speed data transmission, two new standards—RS-422A and RS-423A—have been developed in recent years; however, they are not yet widely used.

To appreciate the difficulties and confusion in this standard, one has to examine its historical background. The RS-232 Standard was developed during the initial days of computer timesharing, long before the existence of TTL logic, and its primary focus was to have compatibility between a terminal and a modem. However, the same standard is now being used for communications between computers and peripherals, and the roles of a

data terminal and a modem have become ambiguous. Should a computer be considered a terminal or a modem? The answer is that it can be either. Therefore, the lines used for transmission and reception will differ, depending on how the manufacturer designs the equipment.

### RS-232C

Figure 15.5(b) shows the RS-232C 25 pins and associated signals. The signals are divided into four groups: **data signals**, **control signals**, **timing signals**, and **grounds**. For data

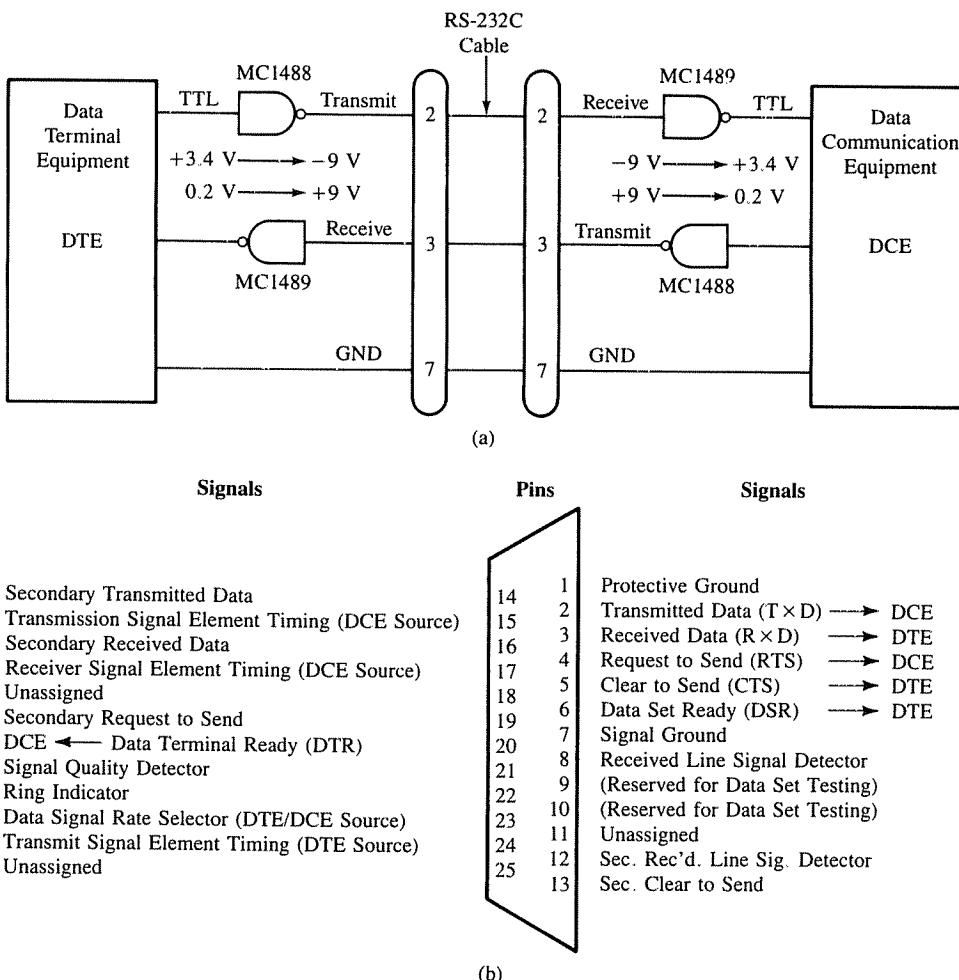


FIGURE 15.5

(a) Minimum Configuration of RS-232C Signals and Voltage Levels (b) RS-232C Signal Definitions and Pin Assignments

SOURCE: Courtesy of Electronic Industries Association.

lines, the voltage level from +3 V to +15 V is defined as logic 0, and from -3 V to -15 V is defined as logic 1 (normally, voltage levels are  $\pm 12$  V). This is negative true logic. Because of incompatibility with TTL logic, voltage translators, called **line drivers** and **line receivers**, are required to interface TTL logic with the RS-232 signals, as shown in Figure 15.5(a). The line driver, MC1488, converts logic 1 into approximately -9 V and logic 0 into +9 V (Figure 15.5(a)). Before the signal is received by the DCE, it is again converted by the line receiver, MC1489, into TTL compatible logic. This raises the question: If the received signal is to be converted back to the TTL levels, what is the reason to convert the transmitted signal to higher voltages in the first place? The primary reason is that the standard was defined before the TTL levels came into existence; before 1960, most equipment was designed to handle higher voltages. The other reason is that this standard provides a higher level of noise margin, from -3 V to +3 V.

The minimum interface requires three lines: pins 2, 3, and 7 as shown in Figure 15.5(a). These lines are defined in relation to the DTE; the terminal transmits on pin 2 and receives on pin 3. On the other hand, the DCE transmits on pin 3 and receives on pin 2. Typically, data transmission with a handshake requires eight lines. Specific functions of handshake lines differ in different peripherals, and therefore, should be referred to in the manufacturer's manuals.

For high-speed transmission, the standards RS-422A and RS-423A are used. These standards use differential amplifiers to reject noise levels and can transmit data at a higher speed with longer cable. The RS-422A allows the maximum speeds of 10 Mbaud for a 40 ft distance and 100 kbaud at 4,000 ft. The RS-423A is limited to 100 kbaud for a 30 ft distance and 10 kbaud at 300 ft. See Table 15.1 for comparison of the three standards: RS-232C, RS-422A, and RS-423A.

### 15.16 Review of Serial I/O Concepts and Approaches to Implementation

Serial data transmission can be implemented through either software or programmable I/O devices. The software and the hardware approaches are conceptually similar. In asynchronous data transmission, the steps can be summarized as follows:

1. Inform the receiver that the transmission is beginning with the Start pulse.
2. Convert a parallel word into a stream of serial bits.
3. Transmit data one bit at a time with appropriate time delay using one data line of an output port. The time delay is determined by the speed of the transmission.
4. Transmit parity check bit.
5. Inform the receiver that transmission is ending by sending Stop bits.

In data reception, the process is reversed. The receiver needs to

1. Recognize the beginning of the transmission.
2. Receive data one bit at a time and convert them into a parallel byte.
3. Check for errors and recognize the end of the transmission.

**TABLE 15.1**  
Comparison of Serial I/O Standards

Specifications	RS-232C	RS-422A	RS-423A
Speed	20 kbaud	10 Mbaud—40 ft 100 kbaud—4,000 ft	100 kbaud—30 ft 1 kbaud—4,000 ft
Distance	50 ft	4,000 ft	4,000 ft
Logic 0	> +3 to +25 V	B > A	+4 to +6 V
Logic 1	< -3 to -25 V	B < A	-4 to -6 V
Receiver Input	± 15 V	± 7 V	± 12 V
Voltage			

NOTE: B and A are differential input to the op amp.

In the software approach, the speed of transmission is set up by using an appropriate delay between the transmission of two consecutive bits, and the entire word is converted into a serial stream by rotating the byte and outputting one bit at a time using one of the data lines of an output port. The software provides the time delay between the two consecutive bits and adds framing bits and the parity bit; this is discussed in Section 15.2.

In the hardware approach, the above functions are performed by a programmable device (chip). The device contains a parallel-to-serial register and 1-bit output port for transmission, and a serial-to-parallel register and 1-bit input port for reception. The rate of transmission and reception is determined by the clock. The programmable chip also includes a control register that can be programmed to add framing and error-check information, and to specify the number of bits to be transferred. The microprocessor transfers a parallel byte using the data bus, and the programmable chip performs the remaining functions for serial I/O.

The software approach is suitable for slow-speed asynchronous data communication where timing requirements are not critical. The approach is simple and inexpensive. The hardware approach is suitable for both asynchronous and synchronous formats. The approach is flexible, and chips can be programmed to accommodate changing requirements. In industrial and commercial products, the hardware approach has become almost universal. This chapter includes the detailed discussion and illustrations of two programmable chips: the Intel 8251 and the Z80 SIO. However, for learning the basic concepts in serial I/O, the software approach is more suitable than the hardware approach; thus, the software approach is described here prior to the discussion of the programmable serial I/O devices. We will limit our discussion to the asynchronous communication mode, which is commonly used in the microcomputer. Synchronous data communication is a specialized technique and will not be discussed here.

The basic concepts concerning serial I/O discussed in the previous sections are summarized in Table 15.2.

**TABLE 15.2**

Summary of Synchronous and Asynchronous Serial Data Communication

Format	Synchronous	Asynchronous
Data Format	Groups of Characters	One Character at a Time
Speed	High (20 k bits/second or Higher)	20 k bits/second or Lower
Framing Information	Sync Characters Are Sent with Each Group	Start and Stop Bits with Every Character
Implementation	Hardware	Hardware or Software
Data Direction	Simplex, Half and Full Duplex	Simplex, Half and Full Duplex

## 15.2

### SOFTWARE-CONTROLLED ASYNCHRONOUS SERIAL I/O

In the software-controlled asynchronous serial mode, the program should perform the following tasks.

1. Output a Start bit.
2. Convert the character to be sent into a stream of serial bits with appropriate delay.
3. Add parity information if necessary.
4. Output one or two Stop bits.

Figure 15.6 shows the accumulator with the code for the ASCII character “I,” and it is converted into a stream of 11 bits, including one Start bit and two Stop bits. After the Start bit, the character bits are transmitted with bit D<sub>0</sub> first and bit D<sub>7</sub> last; for ASCII characters, bit D<sub>7</sub> can be used to add parity information. The bit time—the delay between two successive bits—is determined by the transmission baud. Figure 15.6 shows the transmission with 1,200 baud; the delay between the two consecutive bits is thus 0.83 ms.

Data reception in the serial mode involves the reverse process: receiving one bit at a time and forming an 8-bit parallel word. The receiving program should continue to read the input port until it receives the Start bit, and then begin to count character bits with appropriate delay.

#### 15.21 Serial Data Transmission

Figure 15.7(a) shows a flowchart to transmit an ASCII character, and it can be explained in the context of the block diagram shown in Figure 15.6(a). When no character is being transmitted, the transmit line of the output port stays high in the Mark position. The transmission begins with the Start bit, active low. The initialization block of the flowchart

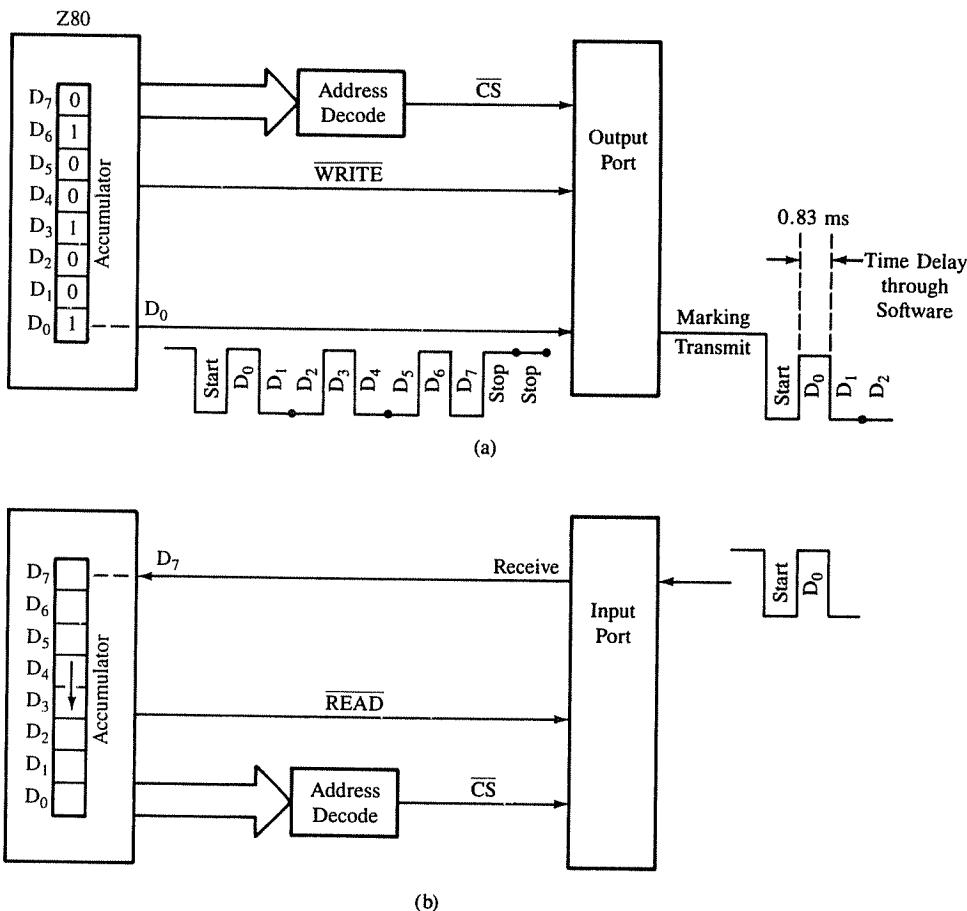
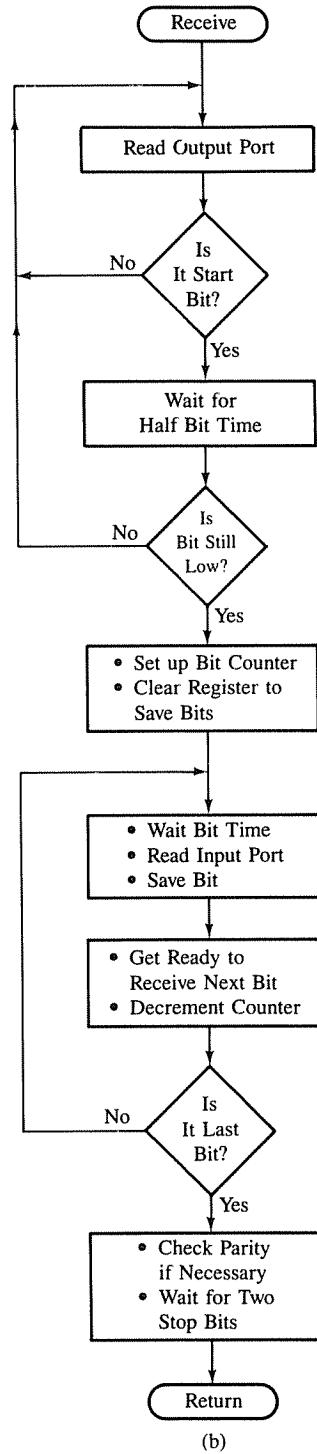
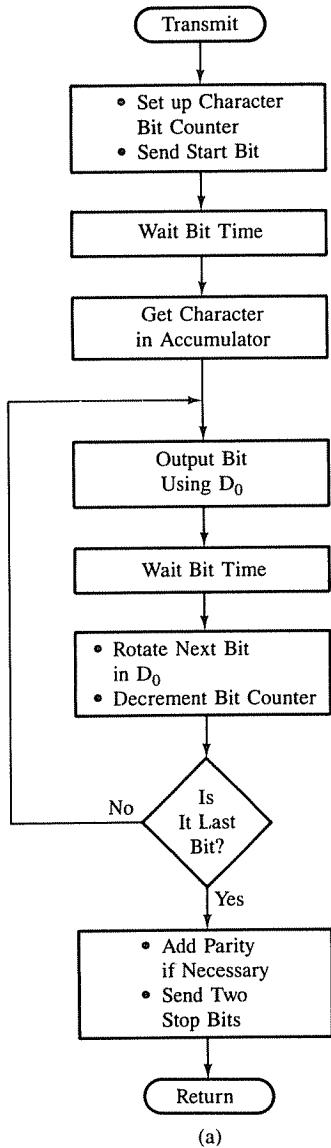


FIGURE 15.6

(a) Serial Data Transmission and (b) Serial Data Reception under Software Control

includes setting up a counter to count eight character bits; Start and Stop bits are sent out separately. The program waits for bit time—0.833 ms for 1,200 baud—and begins to send one character-bit at a time over the data line D<sub>0</sub> at the interval of 0.83 ms. To get ready for the next bit, the program rotates the bits—for example, D<sub>1</sub> into D<sub>0</sub>. It repeats the loop eight times, and finally sends out two Stop bits. Assuming that the character is being sent to a printer, the printer waits until it receives all the bits serially, checks the parity if necessary, forms a character, and prints it during the Stop bits. The Stop bits perform two functions: They allow sufficient time for the printer to print the character and leave the transmit line in the Mark position at the end of the character.



**FIGURE 15.7**  
Flowcharts: (a) Transmission and (b) Reception of an ASCII Character

### 15.22 Serial Data Reception

In serial data reception, the program begins by reading the input port. When no character is being received, the input line stays high. The program stays in the loop and continues to read the port until the Start bit (active low) is received, as shown in the flowchart (Figure 15.7(b)).

When the Start bit is received, the program waits for half the bit time and samples the character bits in the middle of the pulse rather than at the beginning to avoid errors in transition. Then it checks again to confirm that it is really a Start bit and not a false start such as a noise spike. In the next block, it initializes the counter to count eight bits and clears a register to save the partial readings. The program reads the input port at the interval of bit time until it reads all the character bits, checks the parity bit, and ignores the last two bits by just waiting for bit times. The character reception also begins with the LSB; that is, the microprocessor will receive bit  $D_0$  first. In Figure 15.6(a), the data line  $D_7$  is used for the reception. The line  $D_7$  provides some programming convenience for serial-to-parallel conversion; the word can be formed by shifting bits to the right whenever a bit is read, and eventually the LSB will reach its proper position.

## PROGRAMMABLE COMMUNICATION INTERFACE— INTEL 8251A: HARDWARE APPROACH TO SERIAL I/O

# 15.3

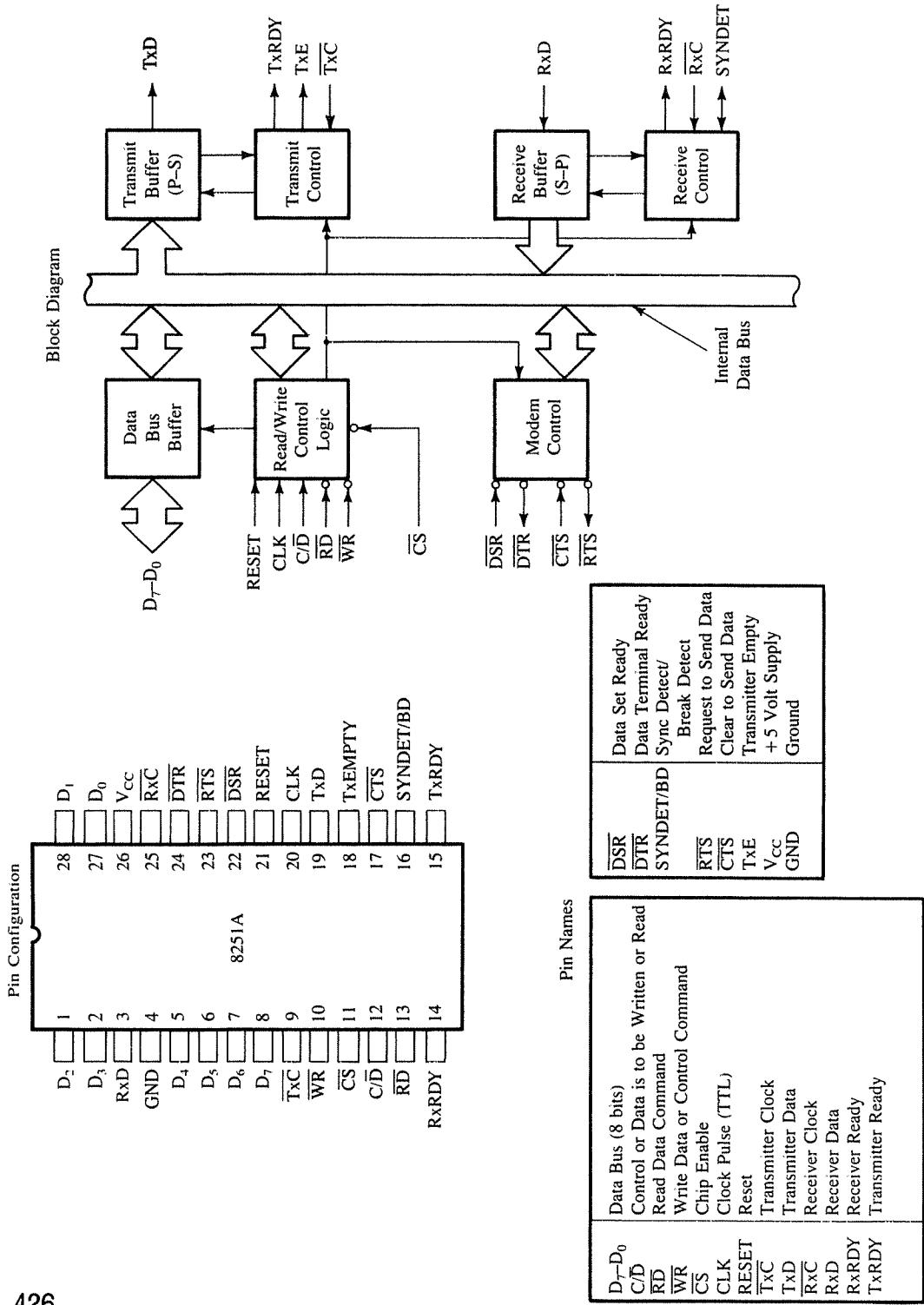
The hardware approach to serial I/O incorporates the same basic principles and requirements necessary for the software approach. The various functions performed separately under software control must be performed by the hardware, designed in an integrated circuit. Such a device should

1. Transfer a parallel word between the microprocessor and the device.
2. Have an input port to receive and an output port to transmit serial data, both with one I/O line each.
3. Perform parallel-to-serial and serial-to-parallel conversion.
4. Provide framing and error-check information.
5. Transmit (or receive) serial data according to the clock connected to the device.

The integrated circuit that meets these requirements is generally called a USART (Universal Synchronous/Asynchronous Receiver/Transmitter). Because of technological advances in IC fabrication, such devices have become quite inexpensive and are commonly used for serial I/O. We will focus on two widely used devices: the Intel 8251A and the Z80 SIO. The 8251A is discussed in this section and the Z80 SIO is described in Section 15.5. The discussion of the 8251A is included prior to the discussion of the SIO because the initialization of the 8251A is easier than that of the SIO.

### 15.31 The 8251A Programmable Communication Interface: Overview

The 8251A is a programmable chip designed for synchronous and asynchronous serial data communication, packaged in a 28-pin DIP. The 8251A is the enhanced version of its

**FIGURE 15.8**

The 8251A: Block Diagram, Pin Configuration, and Description  
SOURCE: Adapted from Intel Corporation, copyright 1979.

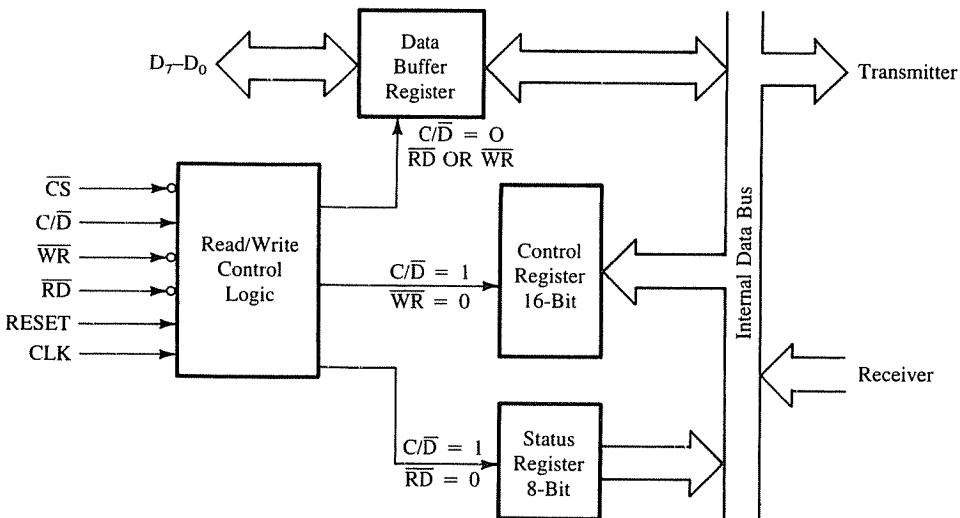
predecessor, the 8251, and is compatible with the 8251. Figure 15.8 shows the block diagram of the 8251A. It includes five sections—Read/Write Control Logic, Transmitter, Receiver, Data Bus Buffer, and Modem Control.

The control logic interfaces the chip with the MPU, determines the functions of the chip according to the control word in its register (explained later), and monitors the data flow. The transmitter section converts a parallel word received from the MPU into serial bits and transmits them over the TxD line to a peripheral. The receiver section receives serial bits from a peripheral, converts them into a parallel word, and transfers the word to the MPU. The modem control is used to establish data communication through modems over telephone lines. The 8251A is a complex device, capable of performing various functions. For the sake of clarity, this chapter focuses only on the asynchronous mode of serial I/O and excludes any discussion of the synchronous mode and the modem control. The asynchronous mode is commonly used for data communication between the MPU and such serial peripherals as terminals and floppy disks.

Figure 15.9 shows an expanded version of the 8251A block diagram. The block diagram shows all the elements of a programmable chip; it includes the interfacing signals, the control register, and the status register. The functions of various blocks are described next.

### 15.32 Read/Write Control Logic and Interfacing

This section has six input signals, control logic, and three buffer registers: data register, control register, and status register.



**FIGURE 15.9**

The 8251A: Expanded Block Diagram of Control Logic and Registers

### Input Signals

- **$\overline{CS}$** —Chip Select: When this signal goes low, the 8251A is selected by the MPU for communication. This is usually connected to a decoded address bus.
- **$C/D$** —Control/Data: When this signal is high, the control register or the status register is addressed, and when it is low the data buffer is addressed. The control register and the status register are differentiated by **WR** and **RD** signals, respectively.
- **$\overline{WR}$** —Write: When this signal goes low, the MPU either writes in the control register or sends output to the data buffer. This is connected to **IOWR** or **MEMWR**.
- **$\overline{RD}$** —Read: When this signal goes low, the MPU either reads a status from the status register or accepts (inputs) data from the data buffer. This is connected to either **IORD** or **MEMRD**.
- **RESET**—Reset: A high on this input resets the 8251A and forces it into the idle mode.
- **CLK**—Clock: This is the clock input, usually connected to the system clock; the clock frequency must be between 740/kHz and 3.125/MHz. This clock does not control either the transmission or the reception rate. This clock is necessary to synchronize internal operations of the 8251A.

**Control Register** This 16-bit register for a control word consists of two independent bytes; the first byte is called the **mode instruction** (word), and the second byte is called the **command instruction** (word). This register can be accessed as an output port when the **C/D** pin is high.

**Status Register** This input register checks the ready status of a peripheral. This register is addressed as an input port when the **C/D** pin is high; it has the same port address as the control register.

**Data Buffer** This bidirectional register can be addressed as an input and an output port when the **C/D** pin is low. Table 15.3 summarizes all the interfacing and control signals.

**TABLE 15.3**  
Summary of Control Signal for the 8251A

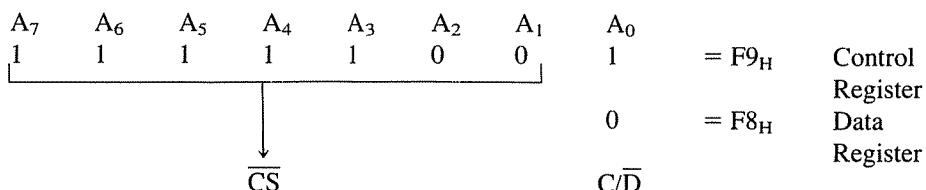
<b>CS</b>	<b>C/D</b>	<b>RD</b>	<b>WR</b>	<b>Function</b>
0	1	1	0	MPU Writes Instructions in the Control Register
0	1	0	1	MPU Reads Status from the Status Register
0	0	1	0	MPU Outputs Data to the Data Buffer
0	0	0	1	MPU Accepts Data from the Data Buffer
1	X	X	X	USART Is Not Selected

**Example  
15.1**

Identify port addresses of the control register and the data register in Figure 15.10 and explain the functions of each control register.

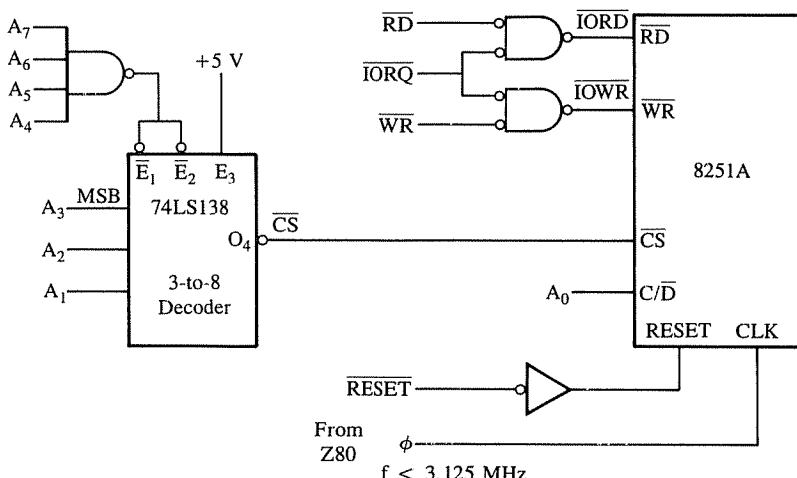
**Solution**

In Figure 15.10, the address line  $A_0$  is connected to the  $C/\bar{D}$  pin of the 8251A. When  $A_0$  is high, the MPU communicates with the control register, and when it is low, it selects the data register. The 8251A is selected when the output line  $O_4$  of the decoder goes low. Therefore, the control register is accessed with the port address  $F9_H$ , and the data register is accessed with the port address  $F8_H$  as shown.



The control register is an output port and the status register is an input port; they are identified by  $\overline{WR}$  and  $\overline{RD}$  signals even if their port addresses are the same.

The data register is selected when the  $C/\bar{D}$  line goes low. Thus, the port address of the data register is  $F8_H$ . The register is bidirectional, and the same address is used to receive or transmit data. The input and output functions are identified by  $\overline{RD}$  and  $\overline{WR}$  signals.



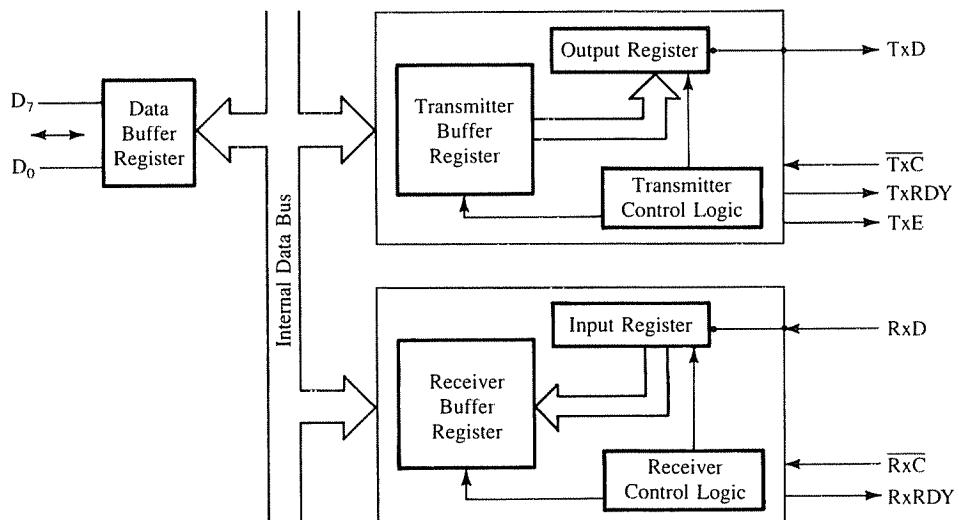
**FIGURE 15.10**  
Interfacing the 8251A

The 8251A does not have IORQ signal; it has RD and WR signals. However, the Z80 identifies its I/O operation using the IORQ signal. Therefore, the control signals IORD and IOWR are generated by ANDing (physically OR gate) RD and WR with the IORQ signal as shown in Figure 15.10. The RESET signal from the Z80 can reset the 8251A, thus clearing all the previous commands, and the Z80 clock signal is used by the 8251A to perform internal functions.

### 15.33 Transmitter Section

The transmitter accepts parallel data from the MPU and converts them into serial data. It has two registers: a buffer register to hold 8 bits and an output register to convert 8 bits into a stream of serial bits (Figure 15.11). The MPU writes a byte in the buffer register, and whenever the output register is empty, the contents of the buffer register are transferred to the output register. This section transmits data on the TxD pin with the appropriate framing bits (Start and Stop). Three output signals and one input signal are associated with the transmitter section.

- TxD—Transmit Data: Serial bits are transmitted on this line.
- TxC—Transmitter Clock: This input signal controls the rate at which bits are transmitted by the USART. The clock frequency can be 1, 16, or 64 times the baud.
- TxRDY—Transmitter Ready: This is an output signal. When it is high, it indicates that the buffer register is empty and the USART is ready to accept a byte. It can be used



**FIGURE 15.11**

The 8251A: Expanded Block Diagrams of Transmitter and Receiver Sections

either to interrupt the MPU or to indicate the status. This signal is reset when a data byte is loaded into the buffer.

- **TxE**—Transmitter Empty: This is an output signal. Logic 1 on this line indicates that the output register and the buffer register are empty. This signal is reset when a byte is transferred from the buffer to the output register.

### 15.34 Receiver Section

The receiver accepts serial data on the RxD line from a peripheral and converts them into parallel data. The section has two registers: the receiver input register and the buffer register (Figure 15.11). When the RxD line goes low, the control logic assumes it is a Start bit, waits for half a bit time, and samples the line again. If the line is still low, the input register accepts the next bits, forms a character, and loads it into the buffer register. Subsequently, the parallel byte is transferred to the MPU when requested. In the asynchronous mode, two input signals and one output signal are necessary.

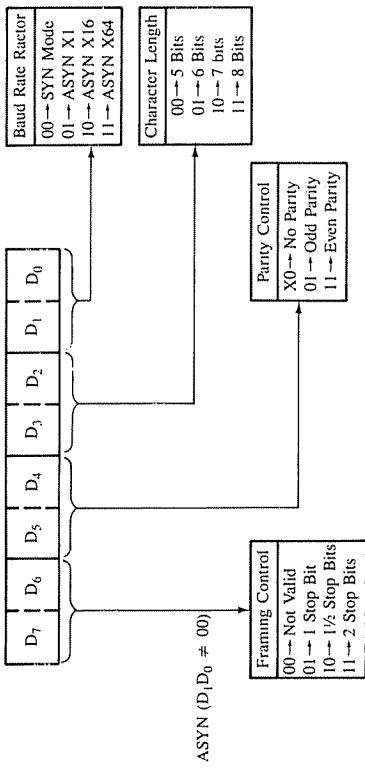
- **RxD**—Receive Data: Bits are received serially on this line and converted into a parallel byte in the receiver input register.
- **RxC**—Receiver Clock: This is a clock signal that controls the rate at which bits are received by the USART. In the asynchronous mode the clock can be set to 1, 16, or 64 times the baud.
- **RxRDY** (Receiver Ready): This is an output signal. It goes high when the USART has a character in the buffer register and is ready to transfer it to the MPU. This line can be used either to indicate the status or to interrupt the MPU.

### 15.35 Programming the 8251A

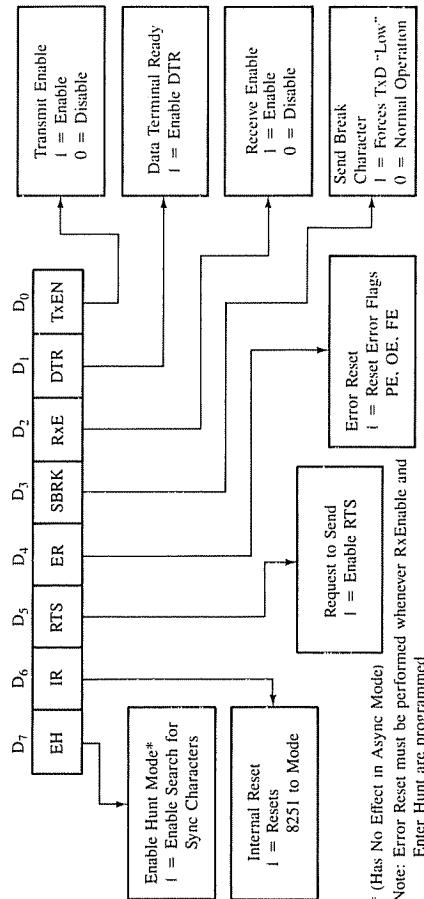
To implement serial communication, the MPU must inform the 8251A of all details such as mode, baud, Stop bits, and parity. Therefore, prior to data transfer, a set of control words must be loaded into the 16-bit control register of the 8251A. In addition, the MPU must check the readiness of a peripheral by reading the status register. The control words are divided into two formats: mode word and command word. The mode word specifies the general characteristics of operation (such as baud, parity, number of Stop bits), and the command word enables data transmission and/or reception. The status word, which can be read by the MPU, provides information concerning register status and transmission errors. Figure 15.12 shows the definitions of these words.

To program the 8251A in the asynchronous mode, the following sequence of steps must be followed.

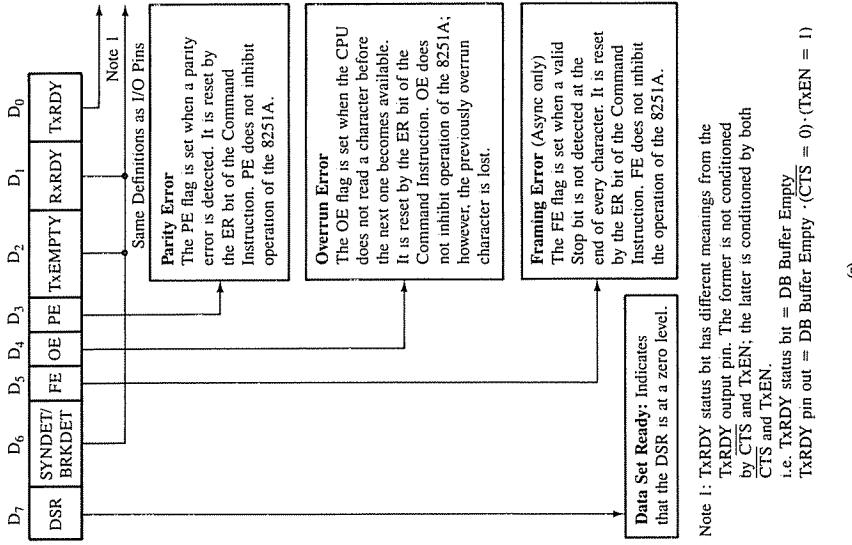
1. Reset the 8251A. This can be done either through a system reset (RESET signal) or software reset of the 8251A (bit D<sub>6</sub> in the command word).
2. Write a mode word in the control register to specify baud, parity, number of stop bits (see Figure 15.12(a) for definition). This word must be followed by a command word (see Step 3).
3. Write a command word in the control register to enable data transfer (see Figure



(a)



(b)



(c)

**FIGURE 15.12**  
 (a) Mode Word, (b) Command Word, and (c) Status Word Format  
 SOURCE: (a and c) Reprinted by permission of Intel Corporation, copyright 1981.

15.12(b) for definition). The command word can be changed anytime during the operation without resetting the 8251A.

4. To modify the mode word, the 8251A must be reset prior to writing a new mode word. This can be accomplished by setting bit D<sub>6</sub> to logic 1 in the command word.

The 8251A is designed to recognize the first word written to the control register as a mode word. However, the software reset is accomplished by the bit D<sub>6</sub> in the command word. If we were to load a command word with bit D<sub>6</sub> = 1, it would be interpreted as the mode word. The solution to this dilemma is to send the first word as a dummy mode word, followed by the command word that resets the 8251A. Therefore, the sequence of programming the 8251 is as follows:

1. Send dummy mode word (00) to the control register.
2. Send command word with bit D<sub>6</sub> = 1 to reset the 8251A.
3. Send mode word to specify communication parameters (baud, parity, etc.).
4. Send command word to enable transmission/reception.

After this sequence, any word sent to the control register will be interpreted as a command word until the 8251A is reset. The above sequence is adequate in most situations; however, on rare occasions the 8251A may be set up for synchronous mode by a random word when the system is turned on. In such a case, it would interpret the next two words written in the control register as Sync characters. Therefore, to deal with the worst possible situation at power turned on, three dummy words are sent to the control register, followed by the command word to reset the 8251A.

Figure 15.10 shows a schematic of interfacing the 8251A with the Z80 MPU.

**Example  
15.2**

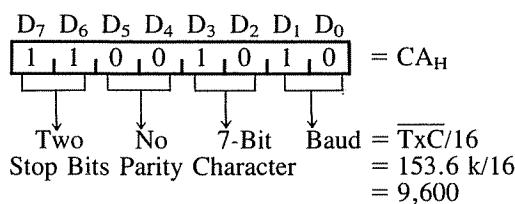
1. Identify the mode and the command words to initialize the 8251A to transmit data with the following requirements:

- Asynchronous mode with 9,600 baud.
- Character length: seven bits and two Stop bits.
- No parity check.

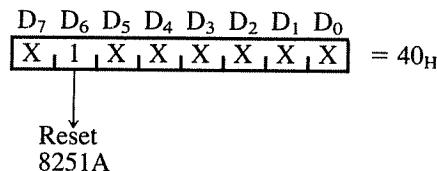
2. Write a subroutine SETUP to initialize the 8251A.

**Solution**

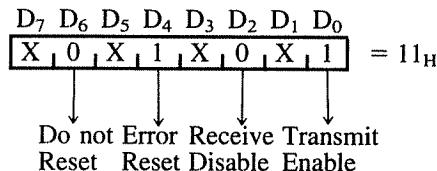
1. The mode word to meet the requirements is as follows (refer to Figure 15.12(a)):



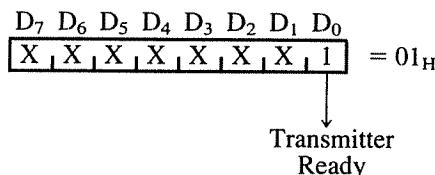
Reset command word—Figure 15.12(b):



Command word to enable transmitter—Figure 15.12(b):



Status Word—Figure 15.12(c):



## 2. Initialization Subroutine:

```

CNTRL EQU F9H ;8251A control port address
PORT EQU F8H ;8251A data port address
RESET EQU 40H ;Reset command
MODE EQU CAH ;Mode word
COMAND EQU 11H ;Command word to enable transmitter
SETUP: LD C, CNTRL ;Load control port address into C
       LD B, 00 ;Load dummy word
       OUT (C), B ;Send dummy word three times
       OUT (C), B
       OUT (C), B
       LD B, RESET ;Load reset command
       OUT (C), B ;Reset 8251A
       LD B, MODE ;Load mode word = CAH
       OUT (C), B ;Specify communication parameters
       LD B, COMAND ;Load command word (11H) to enable transmission
       OUT (C), B ;Enable transmitter
       RET
    
```

As the power is turned on, the 8251A can come up in either the asynchronous or the synchronous mode. If it is set up in the synchronous mode, it expects the next two words as Sync pulses. Therefore, this initialization routine sends three dummy words at the beginning; the first two will be interpreted as Sync characters and the third as the Mode word, which is followed by the Reset command word. Then the Mode and Command words are sent in a sequence. If the 8251A comes up in the asynchronous mode, it will interpret the first dummy word as the Mode word, followed by Command and Mode words; thus, it will interpret the Reset as a Command and not as a Mode word.

---

### ILLUSTRATION: INTERFACING A RS-232 TERMINAL USING THE 8251A IN THE POLLED MODE

---

## 15.4

CRT terminals are serial I/O devices, generally connected using the RS-232C standard. The terminal has two sections: the ASCII keyboard as an input port and the video screen as an output port. To transmit data using the RS-232C standard, the TTL logic levels should be converted into RS-232C levels by using line drivers, and to receive data, line receivers should be used to convert back into TTL logic levels.

#### 15.41 Problem Statement

1. Figure 15.13 shows a schematic of interfacing a CRT terminal using RS-232C. The port decoding logic is the same as in Figure 15.10 with the control port address  $F9_H$  and the data port address  $F8_H$ . Explain the RS-232C signals and the operations of the line driver (MC 1488) and the line receiver (MC 1489).
2. Write a program to transmit a message from a Z80 single-board microcomputer to a CRT terminal under program control (status check). The requirements are as follows:
  - A message is stored in ASCII characters (without parity) in memory locations starting at  $XX70_H$ .
  - The message specifies the number of characters (excluding the first byte) to be transmitted in the first byte and concludes with the characters for the carriage return and the line feed.
  - Use subroutine SETUP to initialize the 8251A as in Example 15.2.
  - Explain the necessary modifications in the program to receive data into the Z80 system.

#### 15.42 RS-232C Signals

In Figure 15.13, the addresses of the control port and the data port are  $F9_H$  and  $F8_H$ , respectively; these addresses are from Figure 15.10. The transmit and receive signals are connected to the terminal using the line driver (MC 1488) and the line receiver (MC 1489), as shown in Figure 15.13. Line drivers and receivers are integrated circuits designed to

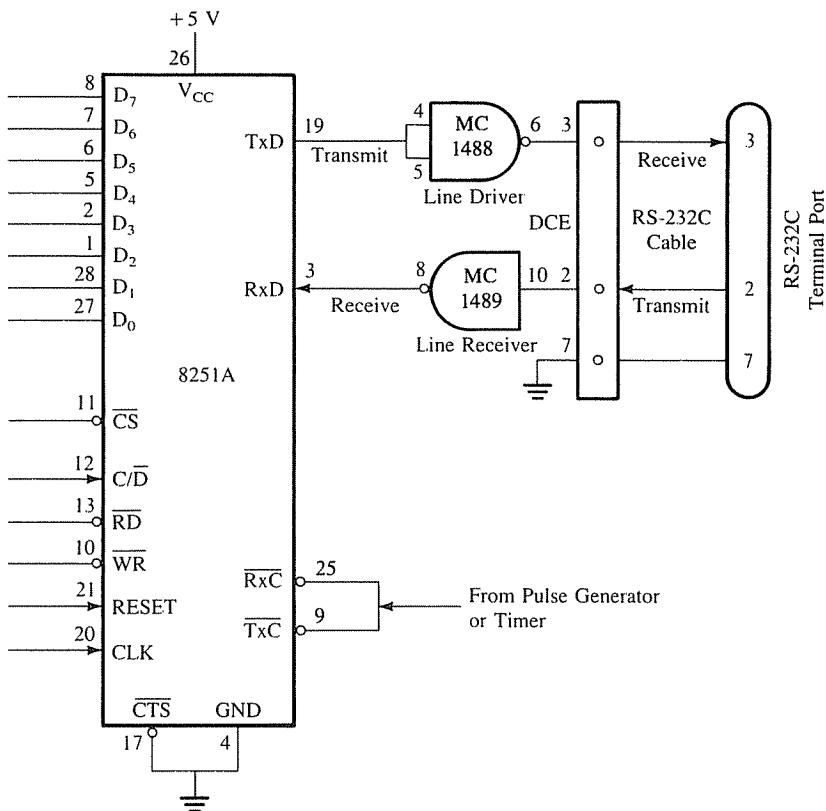


FIGURE 15.13

Schematic: Interfacing a CRT Terminal Using RS-232 with the 8251A in the Polled Mode

interface TTL logic levels with the RS-232C signal levels. They are used primarily in interfacing data terminal equipment (DTE) with data communication equipment (DCE).

**Line Driver: MC 1488** This is a quad line driver which converts TTL input levels to a maximum  $+15\text{ V}_{\text{DC}}$  output signal. Typically, for logic 0 input ( $< +0.8\text{ V}_{\text{DC}}$ ) the output is around  $+10\text{ V}$ , and for logic 1 input ( $> +2.4\text{ V}_{\text{DC}}$ ) the output is around  $-10\text{ V}$ ; thus, the positive true logic is converted into negative true logic for RS-232C signals. The internal circuit of the MC 1488 functions much like a comparator. For an input lower than the threshold voltage, the output approaches positive power supply voltage, and for an input higher than the threshold voltage, the output approaches negative power supply voltage.

**Line Receiver: MC 1489** This a quad line receiver which converts high voltage signals (typically  $+15\text{ V}$ ) into TTL logic levels. Output voltages usually range from  $0.2\text{ V}$  (low)

to 4.0 V (high). The internal circuit functions as an on/off transistor. When the transistor base has a negative input voltage, the transistor is turned off and the collector voltage (the output of the MC 1489) is high. When the transistor base has a positive input voltage, the transistor is driven into saturation to 0.2 V.

### 15.43 Program

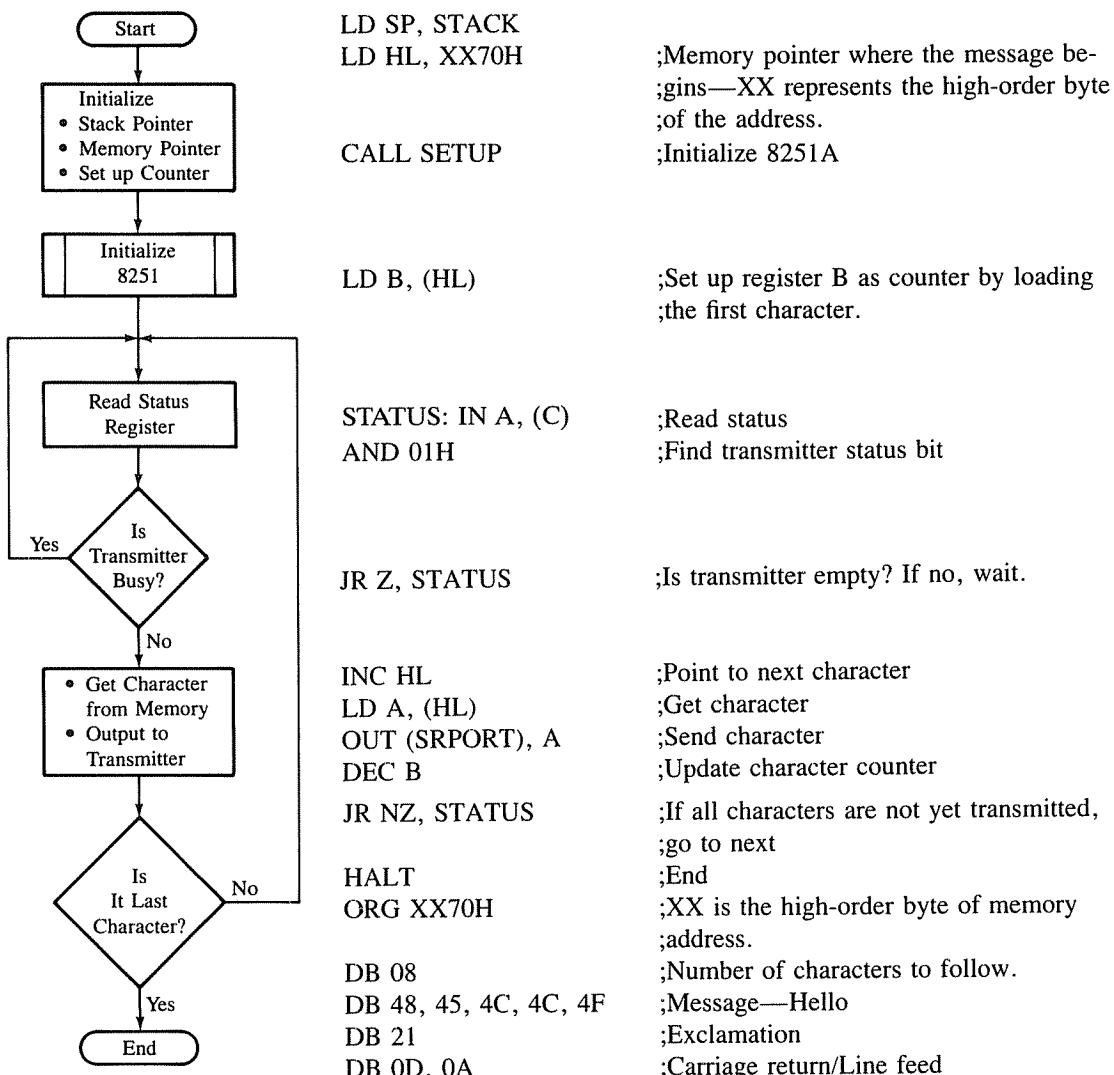


FIGURE 15.14

Flowchart: Transmission from Microcomputer to Terminal

### PROGRAM DESCRIPTION

According to the problem statement, the first character of the message specifies the number of characters to be transmitted. Therefore, the instruction LD B, (HL) loads the first character (in this case 08<sub>H</sub>) into register B and sets that register as counter. The subroutine SETUP (described earlier in Example 15.2) initializes the 8251A for the given specifications.

The next group of instructions, starting with the label STATUS, continues to read the status port and check bit D<sub>0</sub> until it is 1. Bit D<sub>0</sub> indicates the status of the transmitter; logic 1 indicates that the transmitter buffer is empty and ready for the next character. When bit D<sub>0</sub> goes to logic 1, the program points to the next character, loads it into the accumulator, and sends it to the transmitter. This loop is repeated until the counter B becomes 0, indicating the completion of the message.

### 15.44 Data Reception

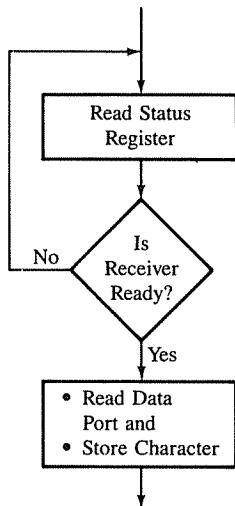
To receive data from the terminal, the command word should be modified to enable the receiver section of the 8251A by setting D<sub>2</sub> = 1. Thus, to enable only the receiver, the command word is as follows:

D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>	=	14 <sub>H</sub>
X	0	X	1	X	1	X	0		

↓            ↓            ↓            ↓            ↓  
 Do not      Error      Receiver    Transmitter  
 Reset        Reset      Enable      Disable

The program for data reception (Figure 15.15) should begin with reading the status port as in the transmission program and checking bit D<sub>1</sub> for receiver ready, instead of bit D<sub>0</sub> for

**FIGURE 15.15**  
Flowchart: Data Reception



transmitter ready (Figure 15.12(c)). When a character is placed into the receiver buffer by the terminal, bit D<sub>1</sub> is set to logic 1, and then the program should read the data port for the character.

## SERIAL INPUT/OUTPUT CONTROLLERS: Z80 SIO AND DART

# 15.5

The **Z80 Serial Input/Output Controller (SIO)** is another commonly used programmable device in serial communication. It has two channels (equivalent of two devices), and both channels can be used for asynchronous or synchronous communication. It is functionally similar to the Intel 8251A (Section 15.3), but includes two channels and additional features such as CRC error check. It is a versatile device, but requires many more programming instructions to set up than does the 8251A. The **Z80 DART (Dual Asynchronous Receiver/Transmitter)** is similar to the SIO, but it is designed to handle only asynchronous serial communication. It also has two channels, and they are identical to the asynchronous sections of the SIO.

In 8-bit microprocessors, serial communication is generally asynchronous; thus, we will focus on applications using the DART or the asynchronous section of the SIO. The SIO can be set up to communicate data in three different ways: program control (status check), interrupt control, and block transfer. In the last section, we illustrated the 8251A under program control; in this section, we focus on interrupt control.

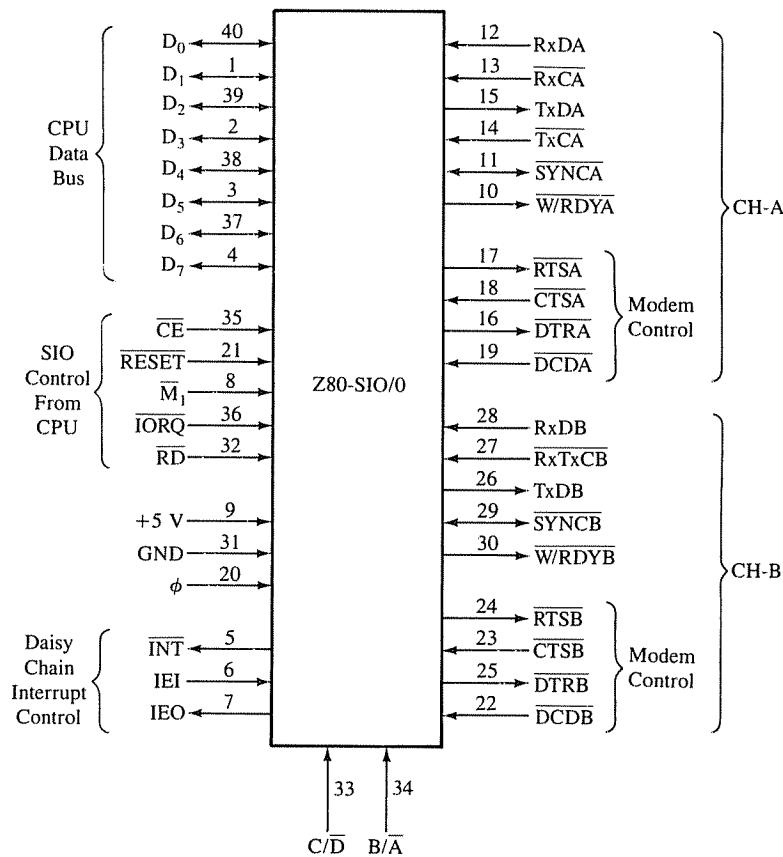
### 15.51 The Z80 SIO and DART: Overview

The Z80 SIO is a dual channel interfacing device in a 40-pin package as shown in Figure 15.16(a). The logic pin-out shows seven sections: Channel A and Channel B, modem control for each channel, parallel data bus, control signals for interfacing, and interrupt control.

Channel A and Channel B are two independent channels and can support both asynchronous and synchronous communications. The SIO has a versatile interrupt structure and can be used to set up daisy-chained interrupt priority. The parallel data bus and the control signals are used to interface the device with the Z80, and the modem control signals are used for communication through telephone lines.

The SIO has three versions: **SIO/0**, **SIO/1**, and **SIO/2**. The internal structure of these versions is identical; however, to provide all the necessary signals for two independent channels, 41 pins are necessary. To accommodate 41 signals in a 40-pin package, one function must be restricted or combined with some other pin. For example, channel B of the SIO/0 has a common clock for the receiver and the transmitter; thus, the clock frequency for the receiver and the transmitter must be the same.

The DART is designed to support only asynchronous communication; it is functionally and architecturally identical with the asynchronous section of the SIO. Figure 15.16(c) shows the pin-out of the DART. It is pin compatible with the Z80 SIO/0 except for two pins, which can be used for general-purpose inputs or ignored.



(a) Z80-SIO/0 Pin Configuration

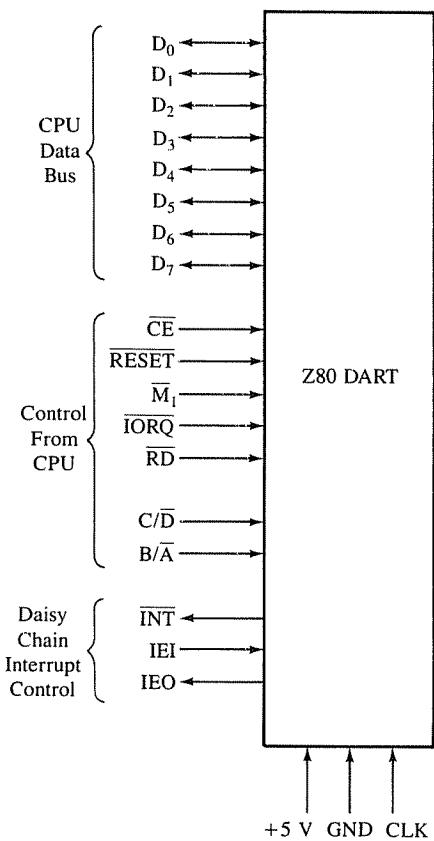
**FIGURE 15.16**

Z80 SIO/0 and DART Logic and Pin Configurations

SOURCE: Courtesy of Zilog, Inc.

Figure 15.17 shows the expanded block diagram of the DART (or the asynchronous section of the SIO) with Channel A only. The Read/Write control section and the data bus are used for interfacing the device with the MPU. The device includes several control and status registers shown as the Control/Status block. The parameters of communications, such as number of bits per character, parity, and transmission rate in relation to the clock have to be specified or programmed. Prior to implementing communication, these control registers must be programmed.

To transmit a byte, the MPU accesses the device through a port address and sends a parallel byte. The control and status circuitry of Channel A frames the byte by adding a Start bit, Stop bits and a parity bit, places it into the register to transmit data, and then



(b) Z80 DART Pin Functions

D <sub>1</sub>	1	40	D <sub>0</sub>
D <sub>3</sub>	2	39	D <sub>2</sub>
D <sub>5</sub>	3	38	D <sub>4</sub>
D <sub>7</sub>	4	37	D <sub>6</sub>
INT	5	36	IORQ
IEI	6	35	CE
IEO	7	34	B/A
M <sub>1</sub>	8	33	C/D
V <sub>DD</sub>	9	32	RD
W/RDYA	10	31	Z80 DART
RIA	11	30	W/RDYB
RxD A	12	29	RIB
RxC A	13	28	RxD B
TxA C	14	27	RxTxCB
TxD A	15	26	TxD B
DTRA	16	25	DTRB
RTSA	17	24	RTSB
CTS A	18	23	CTSB
DCDA	19	22	DCDB
CLK	20	21	RESET

(c) Pin Assignments

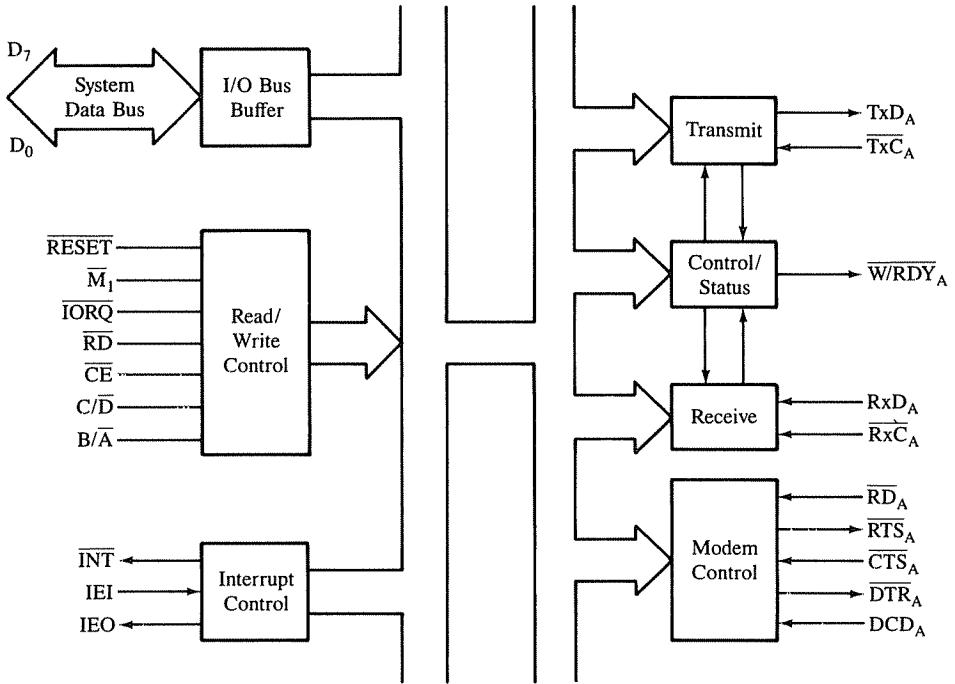
FIGURE 15.16 (continued)

transmits one bit at a time by using the shift register. The baud is determined by the transmitter clock and multiplying factor specified in the control register. To receive data, the process is reversed. In addition, the data go through error logic to check errors. The following sections describe various signals and illustrate the serial communication process.

## 15.52 Read/Write Control Logic and Interfacing

To interface the DART (or the SIO) with the Z80, the signals are as follows:

- D<sub>0</sub>-D<sub>7</sub>—Data Bus: This is a tri-state bidirectional data bus that transfers data and commands between the Z80 and the SIO.
- B/A—Channel Select: This signal selects either channel A or channel B for communication. When it is logic 1, channel B is selected, and when it is logic 0, channel A is selected. Address line A<sub>0</sub> from the Z80 is generally connected to this signal.



**FIGURE 15.17**  
Z80 DART: Expanded Block Diagram of Channel A

- **C/D**—Control or Data: This signal selects either the control register or the data register of the selected channel. When this signal is logic 1, the MPU communicates with the control register either to write a command or read the status. When this signal is logic 0, the MPU either writes a byte in the data register to transmit or reads a byte from the receiver. Address line A<sub>1</sub> from the Z80 is generally connected to this signal.
- **CE**—Chip Enable: When this signal goes low, it indicates that the SIO has been selected for communication. The decoded address bus is usually connected to this signal. The address lines connected to signals **CE**, **C/D**, and **B/A** determine the port addresses of the control register and the data register. Table 15.4 summarizes the active levels of these signals and the control signals.
- **φ**—System Clock: This is a single phase clock input from the system; it synchronizes the internal operations of the SIO.
- **RESET**—Reset: This is an active low signal that disables all the SIO operations, including interrupts. After a reset, the control registers must be rewritten to transmit or receive.
- **IORQ** (I/O Request), **M<sub>1</sub>** (Machine Cycle 1), and **RD** (Read): These are three active low

signals that perform Read/Write/Interrupt Acknowledge operations as described below:

1. *Read*. This operation is performed when  $\overline{IORQ}$  and  $\overline{RD}$  are active low and  $\overline{M}_1$  is high. In this operation, the Z80 either receives a data byte or reads a status from the status register.
2. *Write*. This operation is performed when  $\overline{IORQ}$  is active low, but  $\overline{RD}$  and  $\overline{M}_1$  are high; the SIO does not have a separate signal for the Write operation. In this operation, the Z80 either writes a command in the control register or transfers a byte for transmission.
3. *Interrupt Acknowledge*. The Z80 acknowledges an interrupt by asserting  $\overline{IORQ}$  and  $\overline{M}_1$  signals low, and the SIO places its interrupt vector on the data bus.

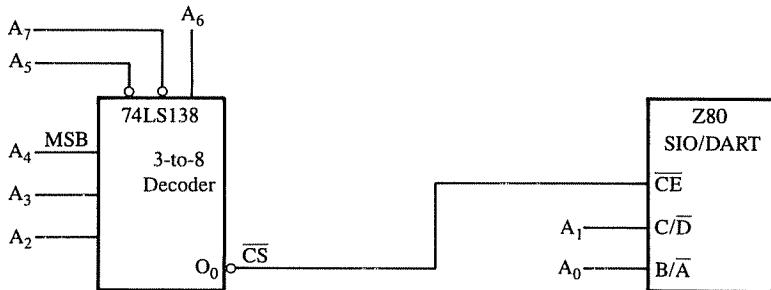
The control operations, the active level of the associated signals, and port selection are summarized in Table 15.4; Example 15.3 illustrates interfacing of the SIO with the Z80.

**TABLE 15.4**  
Summary of Control Signals and Port Selection of Z80 SIO/DART

<b>IORQ</b>	<b>RD</b>	<b><math>M_1</math></b>	<b>CE</b>	<b>C/D</b>	<b>B/A</b>	<b>Function and Port Selection</b>
0	1	1	0	0	0	Z80 Selects Channel A for Data Transmission
			0	0	1	Z80 Selects Channel B for Data Transmission
			0	1	0	Z80 Writes Command in Channel A Control Register
			0	1	1	Z80 Writes Command in Channel B Control Register
0	0	1	0	0	0	Z80 Selects Channel A for Data Reception
			0	0	1	Z80 Selects Channel B for Data Reception
			0	1	0	Z80 Reads Channel A Status Register
			0	1	1	Z80 Reads Channel B Status Register
		1	X	X		SIO Is Not Selected
0	1	0				SIO acknowledges an interrupt and a vector address is placed on the data bus

**Example  
15.3**

Determine the port addresses of the Z80 SIO control registers and data registers shown in Figure 15.18.



**FIGURE 15.18**  
Interfacing Z80 SIO/DART

**Solution**

According to Table 15.4, the SIO is selected when the  $\overline{CE}$  signal goes low, and the  $\overline{CE}$  goes low when the output line  $O_0$  of the decoder is asserted active low. By combining the logic levels of  $A_7$ – $A_2$  with those of  $A_1$  and  $A_0$ , the port addresses range from  $40_H$  to  $43_H$  as shown.

$A_7 \ A_6 \ A_5 \ A_4 \ A_3 \ A_2 \ A_1 \ A_0$

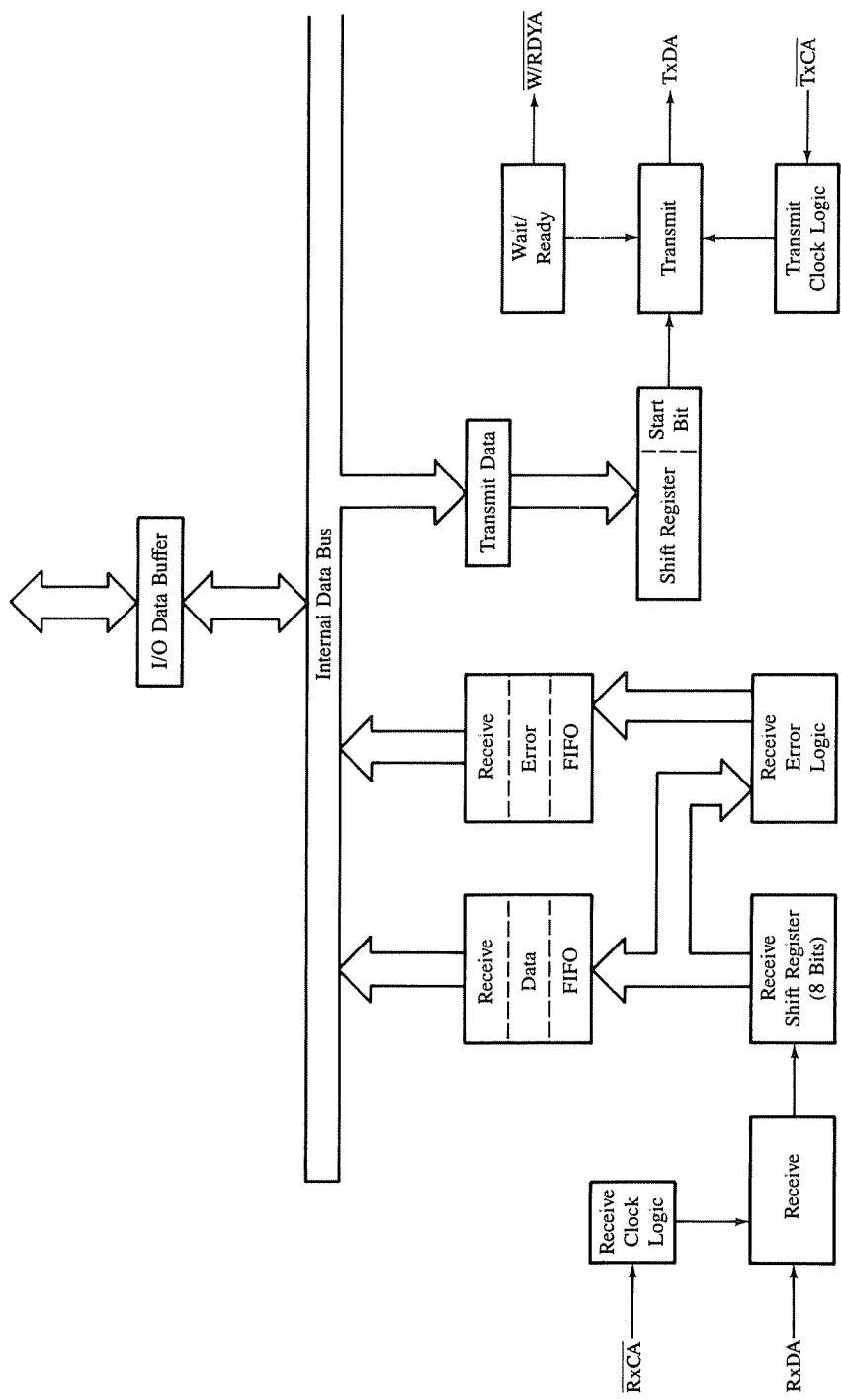
$C/D \ B/A$

$0 \ 1 \ 0$	$0 \ 0 \ 0$	$0 \ 0 = 40_H$	Channel A data register
$\downarrow$ Decoder Enable	$\downarrow$ Decoder Input	$0 \ 1 = 41_H$	Channel B data register
		$1 \ 0 = 42_H$	Channel A control register
		$1 \ 1 = 43_H$	Channel B control register

### 15.53 Transmitter and Receiver Sections

When the MPU writes into the channel A data register, the byte is placed into the Transmit Data register of the transmitter section (Figure 15.19). The byte is properly framed by adding Start and Stop bits according to the instructions written into the control register during the initialization of the device. Then the framed byte is transmitted one bit at a time over the  $TxD_A$  line by use of the shift register. The rate of transmission is determined by the transmitter clock and the scaling factor specified in the control register. The data transmission rate can be specified as 1, 1/16, 1/32, or 1/64 of the clock.

The receiver section has one 8-bit Receive Shift register and three buffer registers (Figure 15.19) arranged in the FIFO (first-in-first-out) format. The shift register receives bits over the  $RxD_A$  line and converts the bits into a parallel word. This word is placed in



**FIGURE 15.19**  
Z80 DART<sup>®</sup>: Receive and Transmit Internal Block Diagram  
SOURCE: Courtesy of Zilog, Inc.

the buffer register that can be read by the MPU. Three buffer registers can store three bytes, thus allowing the MPU sufficient time for an interrupt service if needed. The incoming data also go through the error logic section, which also has three registers. Status information associated with each byte and errors such as parity, framing error, and overrun are stored in Receive Error registers. The rate of reception is again determined by the receiver clock and the scaling factor. The line called W/RDYA shown in Figure 15.19 synchronizes the data transfer (described later in more detail).

Although Figure 15.19 shows only the internal structure of Channel A of the DART, the discussion is equally valid for Channel B and the asynchronous section of the SIO. Five signals associated with this section are described below:

- TxDA—Transmit Data: This is an output signal, and serial bits are transmitted on this line.
- TxCA—Transmitter Clock: This is an input clock signal, and the clock frequency can be 1, 16, 32, or 64 times the transmission data rate. However, this scaling factor must be the same for both the transmitter and the receiver.
- RxDA—Receive Data: Bits are received serially on this line.
- RxCA—Receiver Clock: This is an input clock signal for the receiver, and the clock frequency can be 1, 16, 32, or 64 times the data rate.
- W/RDYA—Wait/Ready: This is an output signal defined during the initialization of the device and used for two functions. When this line is defined as a Wait function, it has the characteristics of the open drain logic, and it synchronizes the data transfer between the device and the MPU by adding Wait states. When it is defined as a Ready function, it can be driven high or low, and it is used in conjunction with the DMA (Direct Memory Access) controller.

Channel B includes all the above signals except that it has only one clock signal used for both transmission and reception. The SIO has one additional signal called SYNC that is used in the synchronous communication.

### 15.54 Programming the SIO and DART

In the asynchronous format, the SIO and the DART can be set up to handle serial data transfer in three modes: polling, interrupt, and block transfer. However, prior to implementing data transfer, a series of commands must be issued to define various communication parameters such as number of bits in a character, number of Stop bits, and parity. These commands are issued using the **Write Registers** (WR) in the control section. Similarly, to synchronize the communication, the MPU needs to check the readiness of the peripheral, examine error conditions, and obtain information concerning the interrupt vector. The MPU performs these tasks by reading the status from **Read Registers** (RR). The DART has six Write Registers (WR0–WR5), shown in Figure 15.20, which are used to specify the communication parameters. It has three Read Registers (RR0–RR2), shown in Figure 15.21, which are used for status information. The SIO has two additional Write Registers (WR6–WR7) that are used only for synchronous communication. This type of architecture raises the question: How does the MPU write in these six (or eight in the SIO)

Write Register 0

D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>
----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------

					0	0	0	Register 0
					0	0	1	Register 1
					0	1	0	Register 2
					0	1	1	Register 3
					1	0	0	Register 4
					1	0	1	Register 5
					0	0	0	Null Code
					0	0	1	Not Used
					0	1	0	Reset Ext/Status Interrupts
					0	1	1	Channel Reset
					1	0	0	Enable INT On Next Rx Character
					1	0	1	Reset TxINT Pending
					1	1	0	Error Reset
					1	1	1	Return From INT (CH-A Only)
								Not Used

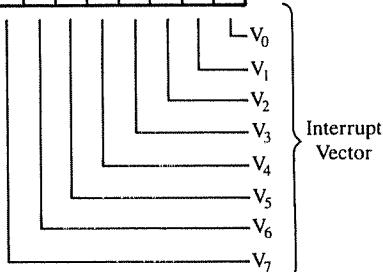
Write Register 1

D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>
----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------

						Ext INT Enable	
						TxINT Enable	
						Status Affects Vector (Ch. B Only)	
					0	0	RxINT Disable
					0	1	RxINT on First Character
					1	0	INT on All Rx Characters (Parity Affects Vector)
					1	1	INT on All Rx Characters (Parity Does not Affect Vector)
							WAIT/READY on R/T
							WAIT/READY Function
							WAIT/READY Enable

Write Register 2 (Channel B Only)

D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>
----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------



Write Register 3

D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>
----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------

						Rx Enable	
						Not Used	
						Auto Enables	
					0	0	Rx 5 Bits/Character
					0	1	Rx 7 Bits/Character
					1	0	Rx 6 Bits/Character
					1	1	Rx 8 Bits/Character

Write Register 4

D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>
----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------

						Parity Enable	
						Parity Even/Odd	
					0	0	Not Used
					0	1	1 Stop Bit/Character
					1	0	1½ Stop Bits/Character
					1	1	2 Stop Bits/Character
							Not Used
					0	0	x1 Clock Mode
					0	1	x16 Clock Mode
					1	0	x32 Clock Mode
					1	1	x64 Clock Mode

Write Register 5

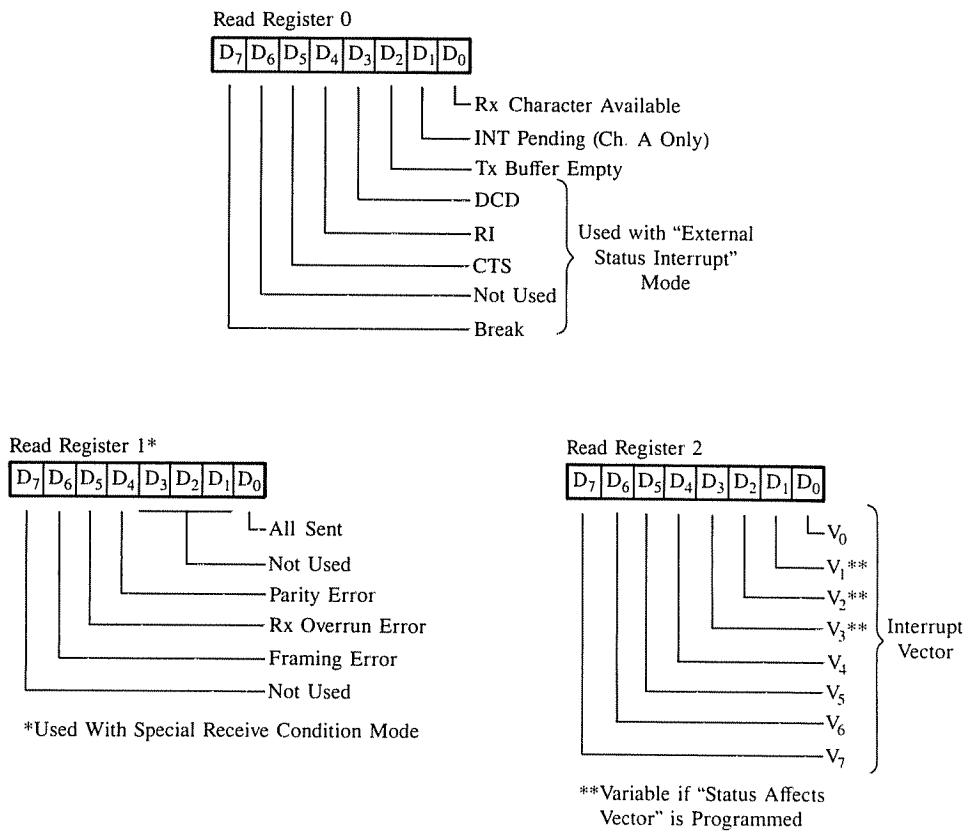
D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>
----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------

						Not Used	
						RTS	
						Not Used	
						Tx Enable	
						Send Break	
					0	0	Tx5 Bits (or Less)/Character
					0	1	Tx7 Bits/Character
					1	0	Tx6 Bits/Character
					1	1	Tx8 Bits/Character
							DTR

FIGURE 15.20

Z80 DART: Write Registers

SOURCE: Courtesy of Zilog, Inc.



**FIGURE 15.21**  
Z80 DART: Read Registers  
SOURCE: Courtesy of Zilog, Inc.

registers and read three different status registers using one port address of the control register? This dilemma is resolved by using three bits (D<sub>2</sub>-D<sub>0</sub>) of the WR0 register as a pointer to the remaining registers; thus, each command issued through WR1-WR5 requires two bytes: one to the WR0 register to set up the pointer, and the other to write a command into the specified register. The Read and Write operations are differentiated by the Read and Write control signals of the MPU.

Both channels A and B have identical architecture, except that register WR2 is included only in Channel B. When the device is set up to handle the interrupt mode, register WR2 is used to specify the low-order address of the interrupt vector, and this address can be used by both channels. Figure 15.20 shows bit definitions for Write Registers and Figure 15.21 shows the bit definitions of the Read (status) Registers. The

functions of the Write Registers are briefly described here and illustrated in Examples 15.4 and 15.5. The Read (status) Registers are explained later.

### WRITE REGISTERS

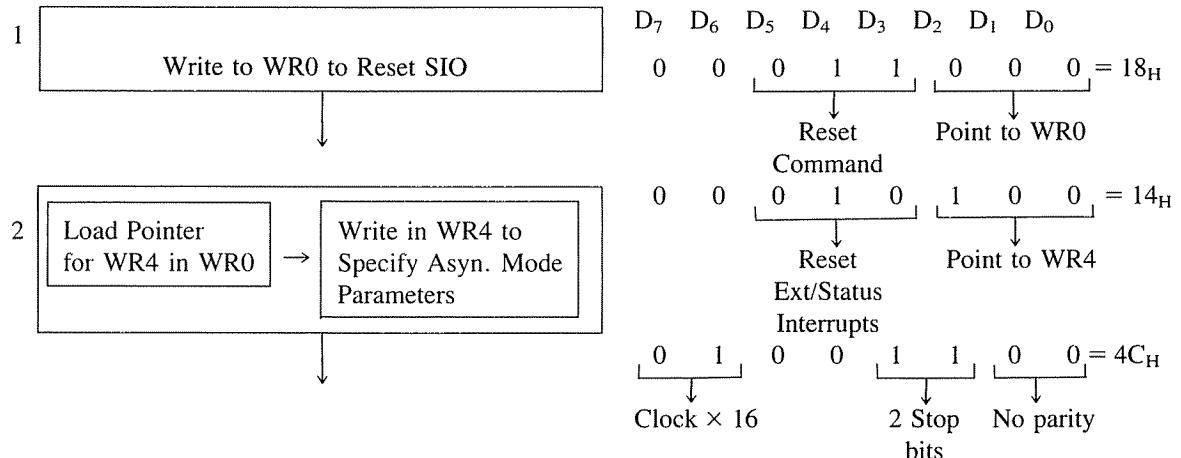
- WR0 (Write Register 0): This is a special register; bits D<sub>0</sub>–D<sub>2</sub> are used as pointers to the remaining registers and bits D<sub>3</sub>–D<sub>5</sub> are used to issue initialization commands. Bits D<sub>7</sub>–D<sub>6</sub> should be 00; they are used in Synchronous communication only.
- WR1 (Write Register 1): The bits in this register define various interrupt and Wait/Ready modes.
- WR2 (Write Register 2—Channel B only): This register is used to store the low-order address of the interrupt vector, and it can be used by the interrupts in both channels. The vector is automatically modified based on the status of bit D<sub>2</sub> in register WR1.
- WR3 (Write Register 3): The bits in this register define receiver parameters and enable the receiver.
- WR4 (Write Register 4): This register contains bits that define communication parameters such as Stop bits, parity, and baud multiplying factor.
- WR5 (Write Register 5): The bits in this register define transmitter parameters and enable the transmitter.
- WR6–WR7 (Write Registers 6–7): These registers are included in the SI0 only and used for synchronous communication.

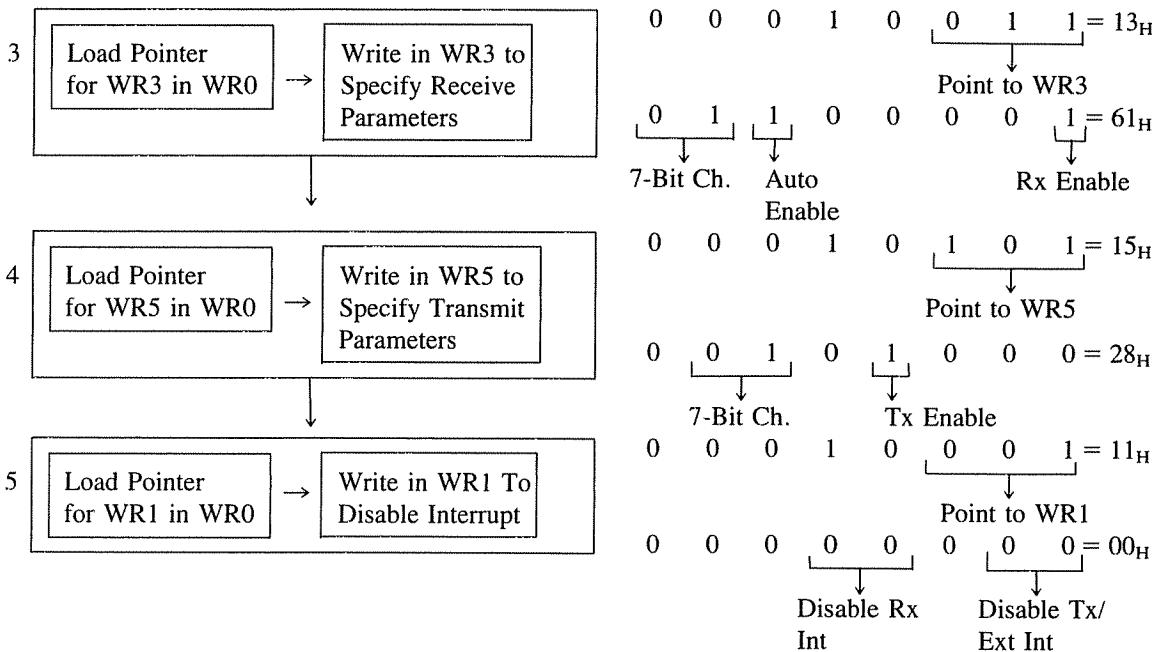
Draw a flowchart and give bit definitions to initialize the Z80 SI0 or DART to meet the following specifications:

**Example  
15.4**

1. Asynchronous transmission/receiver format without interrupts.
2. 7-bit character, two Stop bits, no parity, frequency multiplying factor = 16.

To initialize the SI0 in the asynchronous mode, register WR0 is used to reset the device and as a pointer to other registers. The steps are as follows:



**Example 15.5**

Write a subroutine to initialize channel A of the DART shown in Figure 15.18 to meet the specifications of Example 15.4.

**Solution**

As shown in Example 15.3, the channel A control port address is 42<sub>H</sub>. Referring to bit definitions in Example 15.4, we can write the instructions using register C as follows. However, the coding can be substantially reduced by using the instruction OTIR (see Assignment 29).

DATAA	EQU 40H	;Channel A data port address
CNTRLA	EQU 42H	;Channel A control port address
DART:	LD C, CNTRLA	;Load control port address into register C
	LD A, 18H	;Reset byte
	OUT (C), A	;Reset Channel A
	LD A, 14H	;Pointer for WR4
	OUT (C), A	
	LD A, 4CH	;Bit definitions for asynchronous communication
	OUT (C), A	;Specify parity, stop bits, clock multiplier
	LD A, 13H	;Pointer to WR3
	OUT (C), A	

LD A, 61H	;Bit definitions for receiver
OUT (C), A	;Enable receiver to receive 7-bit character
LD A, 15H	;Point to WR5
OUT (C), A	
LD A, 28H	;Bit definition for transmitter
OUT (C), A	;Enable transmitter to transmit 7-bit character
LD A, 11H	;Point to WR1
OUT (C), A	
LD A, 00H	:Byte to disable interrupts
OUT (C), A	;Disable interrupts
RET	;Initialization complete

### READ REGISTERS

The SIO (DART) has three registers (RR0, and RR1, and RR2) that can be read by the MPU to obtain receiver/transmitter status information. Registers RR0 and RR1 are included in both channels; however, register RR2 is only in Channel B. The RR0 register can be read directly by accessing the control port; however, the remaining two registers have to be read by using the pointer bits in Write register WR0. Figure 15.21 shows bit definitions of the Read Registers. These registers are briefly described here and illustrated in the following examples.

- RR0 (Read Register 0): This register provides the status information of the receiver, transmitter, and the interrupts. Bit D<sub>0</sub> is set when a character is received, and bit D<sub>2</sub> is set when the transmitter buffer is empty.
- RR1 (Read Register 1): This register monitors errors in receiving data; bits in this register identify various types of the errors, such as parity, framing, and overrun. To read the information from this register, the pointer must be written in register WR0.
- RR2 (Read Register 2—Channel B only): The interrupt vector written in register WR2 of Channel B is available through this register for interrupts in both channels. If bit D<sub>2</sub> of Write Register WR1 is set, the vector address can be modified according to the interrupting source, and the modified address is returned to the MPU when the interrupt is acknowledged.

Assuming Channel A of the SIO or DART is initialized for asynchronous serial I/O with the polled mode, write a subroutine to receive a character and check for errors. The character should be stored in memory location INBUF, and if there are any errors, the contents of the status register RR1 should be stored in memory location ERRCHK.

**Example 15.6**

The availability of a character is indicated by bit D<sub>0</sub> in RR0. Therefore, in the polled mode, the program should continue to check bit D<sub>0</sub> until it is set.

**Solution**

READ:	→IN A, (CNTRLA)	;Read status register RR0
	BIT 0, A	;Check bit D <sub>0</sub>
	JR Z, READ	;Wait until character is available
	IN A, (PORTA)	;Read character
	LD (INBUF), A	;Store character
CHECK:	LD A, 01H	;Load pointer for RR1
	OUT (CNTRLA), A	;Select status register RR1
	IN A, (CNTRLA)	;Read error flags in register RR1
	AND 70H	;Mask all bits except D <sub>6</sub> , D <sub>5</sub> , D <sub>4</sub>
	LD (ERRCHK), A	;These are error flags (refer to Figure 15.21)
	RET Z	;Store error status.
RESET:	LD A, 30H	;No errors
	OUT (CNTRLA), A	;Reset command
	RET	;Reset all error flags

**Description** The first instruction (IN) of this subroutine places the contents of the status register RR0 in the accumulator by reading the control port of Channel A. To read register RR0 does not require a pointer in WR0. Bit D<sub>0</sub> of the register RR0 indicates the availability of a character. Thus, the routine continues to check bit D<sub>0</sub> until it is set. Once a character is available, it is stored in memory location INBUF.

If there are errors in the received character, bits D<sub>6</sub> (framing), D<sub>5</sub> (overrun), and D<sub>4</sub> (parity) of register RR1 are set. However, to read register RR1, the pointer needs to be loaded into Write Register WR0. The subroutine reads RR1, logically ANDs its contents with 01110000, and saves the result for further action in location ERRCHK. If there are no errors, error bits (D<sub>6</sub>, D<sub>5</sub>, and D<sub>4</sub>) are reset, the Z flag of the Z80 is set, and the program returns on the Z flag. If there are errors, the error bits are reset by sending the reset instruction to the control port.

**Example  
15.7**

Assuming the SIO (DART) Channel A is initialized for the polled I/O, write a subroutine to send a character stored in memory location OUTBUF.

**Solution**

The status of the transmitter section is indicated by bit D<sub>2</sub> in register RR0. Therefore, the subroutine should continue to check bit D<sub>2</sub> in RR0 until it is set, at which time it should transmit the character.

STATUS:	→IN A, (CNTRLA)	;Read register RR0
	BIT 2, A	;Check transmitter buffer if it is empty
	JR Z, STATUS	;Wait until buffer is empty
	LD A, (OUTBUF)	;Get character from memory
	OUT (PORTA), A	;Send the character
	RET	

### 15.55 The SIO (DART) In the Interrupt Mode

The SIO has a powerful interrupt scheme that can respond quickly to various sources of interrupts. In the previous examples of data transmission and reception in the polled mode, we have seen that the MPU is kept occupied continuously in polling, even if there is no character to transmit or receive. In the interrupt mode, the SIO will interrupt the MPU only on a need basis—such as when it has received a character, when the transmitter is empty, or under some special error conditions. The MPU is thus free to perform other functions.

Figure 15.22 shows three sources of interrupts: transmit, receive, and external status. In transmission, when the buffer becomes empty and the SIO is ready for the next character, an interrupt is generated. The external status interrupt is generated whenever there is a transition in status lines and when transmission conditions such as Transmit Underrun and Break occur.

The receive interrupt is somewhat complex in that it has two categories: one for the first character received and the second for every character received. The second category is similar to the transmit interrupt; whenever a character is received, an interrupt is generated. The first category is used primarily for the block transfer. In this case, when the first character arrives, an interrupt is generated, and then no other interrupt is generated until

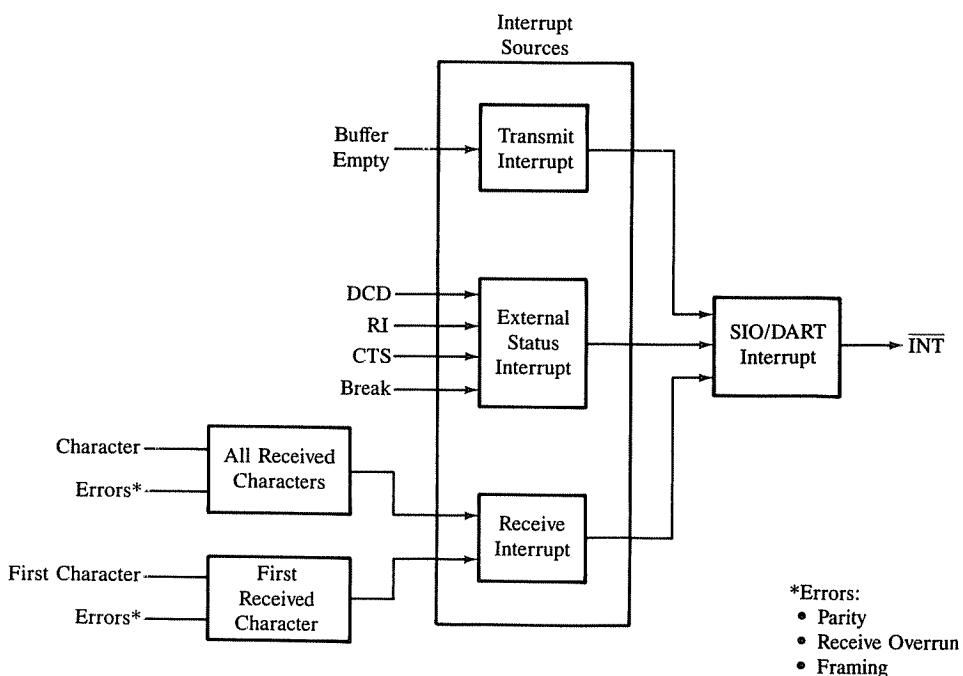


Figure 15.22  
Z80 DART: Interrupt Structure

the block transfer is completed or a higher priority source requests a service. The receive interrupt can also be generated when errors occur. Once an interrupt is generated, it is necessary to identify the source of the interrupt. Two procedures can be used: polling and modifying the interrupt vector. For both methods, an interrupt vector must be loaded into register WR2 in Channel B during the initialization. In the polling method, once an interrupt is acknowledged, the program control will be transferred to the vector location. Then the service routine will identify the source of the interrupt by checking the status registers RR0 and RR1. In the other method, the interrupt vector in register WR2 is modified according to the interrupting source and eight different addresses can be generated, as shown in Table 15.5. The vectors shown in Table 15.5 are arranged according to their interrupt priorities, starting with the lowest priority 0; Channel B has lower priority than Channel A, and within each channel the interrupt generated by the receive errors (Special Receive Condition) has the highest priority. This is further illustrated in Example 15.8.

The Z80 SIO (DART) has three signals to handle the interrupts: one to generate an interrupt and the other two to set the priorities among various devices using the daisy chain scheme. The signals are as follows:

- INT—Interrupt Request: This is an open drain active low signal, and it is generated to acknowledge the conditions previously discussed.
- IEI—Interrupt Enable IN and IEO—Interrupt Enable Out: These signals are used to determine the priorities among devices in the system when multiple devices are connected in the interrupt driven mode. The functions of these signals are identical to those of IEI and IEO signals in the Z80 PIO and CTC described in the previous chapters.

---

**Example 15.8**

Add the necessary instructions in the initialization subroutine SETUP in Example 15.4 to include the interrupt mode with the interrupt vector at  $80_H$ . The vector should be modified according to the interrupting source. Assuming the interrupt register in the Z80 contains the address  $20_H$ , specify the vector addresses for the transmit interrupt in Channel A and the interrupt due to the parity error (refer to Table 15.5).

**TABLE 15.5**  
Interrupt Vectors

	Type of Interrupt	D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>	Hex	Priority
Channel B:	External/status Transition	0	0	0	0	0	0	1	0	= 02	0
	Transmitter Buffer Empty	0	0	0	0	0	0	0	0	= 00	1
	Received Character	0	0	0	0	0	1	0	0	= 04	2
	Special Receive Condition	0	0	0	0	0	1	1	0	= 06	3
Channel A:	External/status Transition	0	0	0	0	1	0	1	0	= 0A	4
	Transmitter Buffer Empty	0	0	0	0	1	0	0	0	= 08	5
	Received Character	0	0	0	0	1	1	0	0	= 0C	6
	Special Receive Condition	0	0	0	0	1	1	1	0	= 0E	7

Referring to the subroutine DART in Example 15.4, we find that three commands must be added to initialize the DART in the interrupt mode: one command in WR2 to specify the vector  $80_H$  (Channel B), the second command to turn on bit  $D_2$  in WR1 (Channel B) to enable modification of the interrupt vector, and the third command in WR1 to enable the interrupts. The second command is additional here because we are using only Channel A, and the third command replaces the last disable command in the previous routine. To load these commands, the register WR0 must be used as a pointer. The instructions are as follows:

LD A, 12H	;Pointer to WR2
OUT (CNTRLB), A	;Set up pointer to WR2
LD A, 80H	;Low-order vector address
OUT (CNTRLB), A	;Load vector into register WR2—Channel B
LD A, 11H	;Point to WR1
OUT (CNTRLB), A	
LD A, 04H	;Bit $D_2 = 1$ (status affects vector)
OUT (CNTRLB), A	;Set bit $D_2$ in Channel B
LD A, 12H	;Byte to enable Tx, Rx (every received char.)
	; and parity error interrupt.
OUT (C), A	;Enable interrupts
RET	;Initialization complete

By referring to Table 15.4 and using  $20_H$  as the high-order address of the interrupt vector, we can identify the memory locations where the pointers are stored for various interrupts.

	$D_3 \quad D_2 \quad D_1$
1. Transmitter Buffer Empty	1 0 0 0   1 0 0   0 = $20\ 88_H$
2. Character Received	1 0 0 0   1 1 0   0 = $20\ 8C_H$
3. Parity Error	1 0 0 0   1 1 1   0 = $20\ 8E_H$

These vectors are situated two memory locations apart ( $208A_H$  is not included in this problem); thus can 16-bit addresses of various service routines be stored.

Solution

### ILLUSTRATION: INTERFACING A RS-232 TERMINAL USING DART (SIO) IN THE INTERRUPT MODE

15.6

The DART requirements for the interrupt-driven serial I/O have been discussed in the previous sections. Examples 15.6 and 15.7 illustrated how to receive and transmit a character using the polled mode, and Example 15.8 illustrated how to initialize the SIO for the interrupt mode. We will now use these concepts to interface a RS-232 terminal to a Z80 system using the DART in the interrupt mode.

The microprocessor views the ASCII terminal as two different peripherals: the keyboard as an input and the CRT as an output display. When a key is pressed, the DART receives a serial stream of bits that can be read by the microprocessor. However, to display that character on the screen, the same byte should be sent out as an output to the screen; these are two distinct processes. The following illustration shows how ASCII characters are received and displayed on the screen.

### 15.61 Problem Statement

1. Interface an ASCII terminal with the Z80 system using the DART in the interrupt mode. Use the same decoding logic as in Example 15.3 to assign port addresses  $40_H$  to the Channel A data port and  $42_H$  to the Channel A control port.
2. Initialize the DART for the interrupt mode to meet the following specifications: (a) asynchronous format, (b) seven-bit character with even parity, (c) transmit and receive frequency = 16 times baud, (d) interrupts should be generated on Rx (All Received Characters), Tx, and parity error.
3. Write a service routine to receive a character when a key is pressed.
4. Write a service routine to transmit (echo) the received character to the CRT screen.

### 15.62 Problem Analysis

1. Figure 15.23 shows the interfacing circuit using the DART in the interrupt mode. The decoding logic is the same as in Example 15.3 with Channel A port addresses PORTA =  $40_H$  and CNTRLA =  $42_H$ . In addition, INT of the DART is connected to the INT signal of the Z80, and the IEI line of the DART is tied high with the assumption there are no other devices in the system.
2. The specifications are the same as in Example 15.8, with the interrupt vector at location  $80_H$ .
3. Main Program: This program is quite simple; it involves initializing the stack and the high-order interrupt vector and calling the initialization subroutine SETUP. To illustrate how characters are received or transmitted in the interrupt-driven mode, the program should have an endless loop. In a real-life example, the MPU would be free to perform other tasks.

POR TA	EQU 40H	;Channel A: data port address
CNTRLA	EQU 42H	;Channel A: control port address
STACK	EQU 2100H	;Stack address
SIO:	LD SP, STACK	;Initialize the stack
	IM 2	;Set up Z80 in interrupt Mode 2
	LD A, 20H	;High-order interrupt vector
	LD (I), A	;Load vector into Interrupt Register
	CALL SETUP	;Initialize DART as in Example 15.8
WAIT:	JR WAIT	;Wait here until an interrupt occurs

4. The following service routine receives a character when the receive interrupt occurs and transmits the same character to the CRT by checking the status of bit D<sub>2</sub> in register RR0.

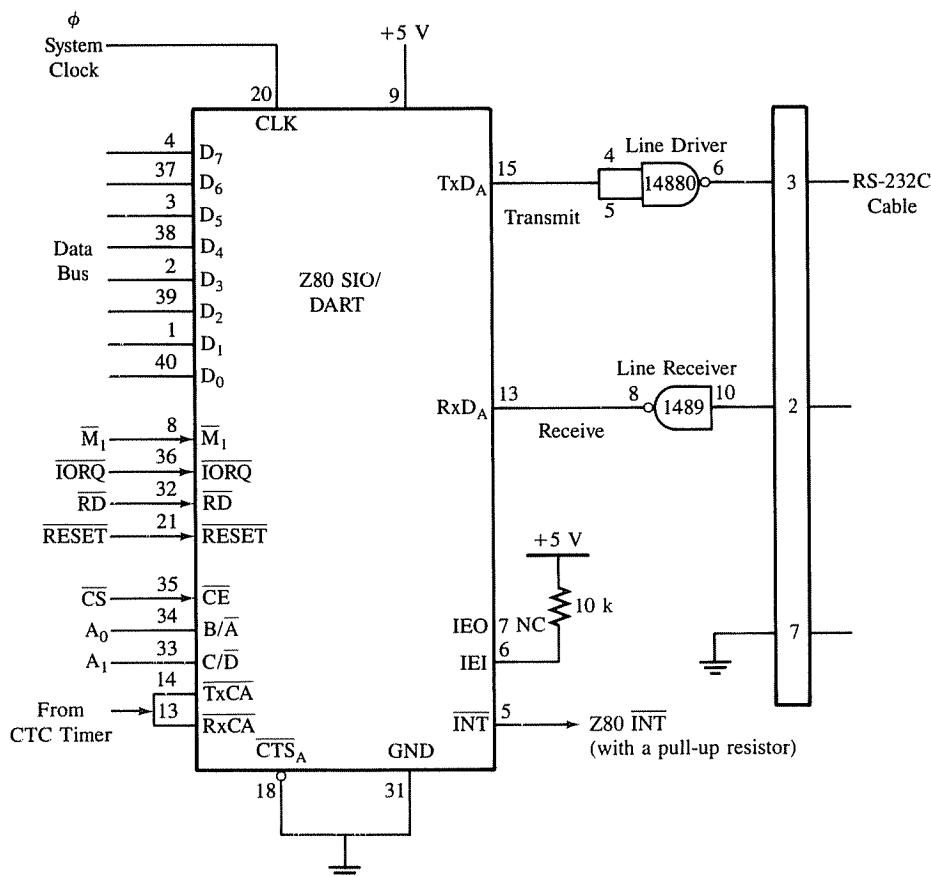


FIGURE 15.23

Interfacing a RS-232C Terminal with a Z80 System Using the DART in the Interrupt Mode

ECHO:	EX AF, AF' PUSH BC LD C, PORTA IN B, (C)	;Save accumulator and flags ;Save contents of BC ;Load PORTA address into register C ;Read received character in register B
STATUS:	IN A, (CNTRLA) BIT 2, A JR Z, STATUS OUT (C), B EX AF, AF' POP BC EI RETI	;Read status register RRO ;Is transmitter buffer empty? ;Wait until transmitter is ready ;Send character to CRT ;Retrieve register contents ;Enable interrupts

## SUMMARY

---

In this chapter, we discussed the technique of serial I/O for data communication, whereby one bit is transferred over one line rather than using eight data lines to transfer a byte. The serial I/O technique is necessary for certain types of equipment and media such as magnetic tapes and telephone lines. The rate of data transfer in serial I/O is determined by the time delay between two successive bits. Therefore, a host of issues such as error check and synchronization of data transfer between the transmitter and the receiver need to be resolved. The serial I/O data transfer can be implemented using software techniques; however, programmable devices called USART (Universal Synchronous/Asynchronous Receiver/Transmitter) are commonly used in industrial and commercial products. Two such devices, the Intel 8251A and the Z80 SIO (DART), are discussed with illustrations in this chapter. The basic concepts involved in serial I/O can be summarized as follows:

1. In serial I/O communication, a word is transmitted one bit at a time over a single line by converting a parallel word into a stream of serial bits. On the other hand, a word is received by converting a stream of bits into a parallel word.
2. Serial data communication can be either synchronous or asynchronous. The synchronous mode is used for high-speed and the asynchronous mode for low-speed data communications.
3. The MPU identifies a serial peripheral through a decoded address and an appropriate control signal. Data transfer can be implemented using such methods as polling (status check), and interrupt.
4. In software-controlled serial transmission, the MPU converts a parallel word into serial bits by using time delays and transmits one bit at a time over one data line of an output port.
5. In software-controlled serial reception, the MPU converts a serial word into a parallel word by using time delays and receives bits over one data line.
6. The Intel 8251A is a programmable serial I/O device known as a USART, which can perform all the functions of software techniques and is commonly used in synchronous and asynchronous data communication.
7. The Z80 SIO (Serial Input/Output Controller) and DART (Dual Asynchronous Receiver/Transmitter) are also commonly used in serial I/O communication.

---

## DEFINITION OF TERMS

---

- **ASCII** (American Standard Code for Information Interchange). A 7-bit alphanumeric code commonly used in computers.
- **EBCDIC** (Extended Binary Coded Decimal Interchange Code). An 8-bit alphanumeric code used primarily in IBM large computers.

- **Asynchronous Serial Data Transmission.** In this format, the transmitter is not synchronized with the receiver by the same master clock. A transmitted character includes information concerning the starting and ending of the character.
- **Synchronous Serial Data Transmission.** In this format, the transmitter is synchronized with the receiver by a common clock.
- **Simplex Transmission.** One-way data communication.
- **Duplex Transmission.** Two-way data communication. Full duplex is simultaneous in both directions and half duplex is one direction at a time.
- **Baud (Rate).** The number of signal changes per second. In serial I/O, it is equal to bits per second, the rate of data transmission.
- **Current Loop.** The transmission of serial data bits as current signals.
- **RS-232C.** A data communications standard that defines voltage signals in reference to data terminal equipment and data communication equipment.
- **RS-422A and -423A.** Data communication standards for high-speed data transmission.
- **USART (Universal Synchronous/Asynchronous Receiver/Transmitter).** A programmable chip designed for synchronous/asynchronous serial data communication.

## ASSIGNMENTS

---

1. Explain the difference between asynchronous and the synchronous data transmission.
2. Explain the terms odd and even parity.
3. Calculate the bit time for 9,600 baud.
4. Sketch the serial output waveform for the ASCII character “A” when it is transmitted with 9,600 baud and even parity.
5. What is the Hex code necessary to transmit the ASCII character “H” with odd parity?
6. Sketch the serial output waveform for the ASCII sign “+” when it is transmitted with 2,400 baud and odd parity.
7. Explain the RS-232C serial I/O standard and specify the signals used in the minimum configuration.
8. Is a microcomputer connected as a DTE or DCE in the RS-232C standard serial I/O communication?
9. Show the RS-232C cable connections in the minimum configuration when a microcomputer and a printer are connected as DCEs.
10. In Figure 15.10, specify the control port and data port addresses if the address lines  $A_7$  and  $A_0$  are interchanged.
11. In Example 15.2 (Figure 15.10), change the mode word from  $CA_H$  to  $CB_H$ , and calculate the clock frequency for 1,200 baud.

12. In Example 15.2, change the mode word to meet the following specifications: 8-bit character, even parity, 1½ Stop bits, and 2,400 baud ( $\text{Tx}C = 153.6 \text{ kHz}$ ).
13. In Example 15.2, explain the consequences if the command word to enable the transmitter is changed to  $51H$ .
14. Write a program to transmit letters A to Z from the MPU to the terminal in Figure 15.13.
15. Write a subroutine to accept a letter from the CRT terminal (Figure 15.13).
16. In Figure 15.13, specify the command word to transmit and to receive characters.
17. Write a subroutine to check and identify an error in the received character by reading the status register (Figure 15.13).
18. In Figure 15.24, identify the addresses of the control and data ports (assume all “don’t care” lines are at logic 0).
19. In Figure 15.24, if the following instructions are executed and the program is transferred to location CHECK, explain the possible reasons for such a transfer.

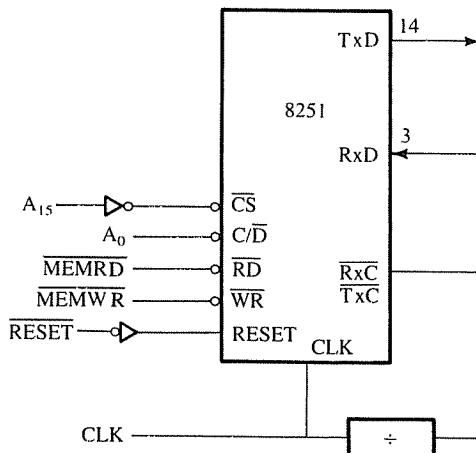
```

LD A, (8001H)      ;Read status register
AND 38H
JP NZ, CHECK

```

20. Write instructions to check a parity error when a character is received (Figure 15.24). If an error occurs, write a command word to disable the receiver, reset the error, and call the Error routine.
21. Specify the mode word and the command word for data communication having the following specifications (Figure 15.24): (a) asynchronous mode, (b) 1,200 baud ( $\text{Tx}C = \text{Rx}C = 76.8 \text{ kHz}$ ), (c) 8-bit character, (d) even parity, (e) 2 Stop bits.

**FIGURE 15.24**  
Schematic for Assignments 18–21



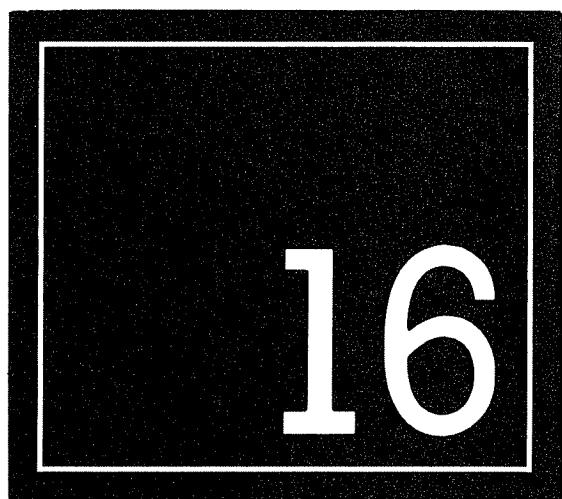
22. Write an interrupt service routine to receive a character from the keyboard and store the character in memory location INBUF.
23. How does the MPU write into the Z80 SIO without the WR signal pin on the SIO?
24. Specify the logic levels of the SIO control signals when the Z80 acknowledges an interrupt.
25. In Figure 15.18, identify the addresses of the control and the data ports if the address lines A<sub>7</sub> and A<sub>0</sub> are interchanged.
26. Write initialization instructions for the Z80 SIO to meet the following specifications: (a) asynchronous format to receive characters under program control, (b) 8-bit character with 2 Stop bits and even parity, (c) frequency multiplying factor 64.
27. Modify the instructions in the previous assignment to include the interrupt control and the low-order interrupt vector at 50<sub>H</sub> that can be modified if an error occurs.
28. In assignment 27, specify the 16-bit address of the interrupt vector if the parity error occurs in Channel B and the Interrupt Register I contains 24<sub>H</sub>.
29. Rewrite the initialization instructions in Example 15.5 using the instruction OTIR.



# Advanced Topics in Memory Design and DMA Concepts

In Chapter 4, we discussed memory interfacing; however, to maintain the clarity in our discussion, we avoided the details of memory access time and its effect on interfacing. Similarly, we did not discuss the need for Wait states in interfacing slow memory devices (peripherals) or the implementation of high-speed data transfer by giving control of system buses to external peripherals. We introduced the topic of dynamic memory, but did not discuss its interfacing. In this chapter, we discuss these timing-related topics and how they affect interfacing and design processes.

We will begin by examining the memory access time in the context of the microprocessor execution time and determine the need for Wait states in interfacing slow memory devices. We will discuss the structure and the requirements of dynamic memory and illustrate the interfacing of a dynamic memory chip. Then we will examine the same memory-related topics from a design point of view and illustrate a memory design using industrial practices. The chapter concludes with the discussion of high-speed data transfer using the Direct Memory Access (DMA) technique.



## OBJECTIVES

- Define memory access time and explain how it relates to the microprocessor machine cycles.
- Determine the need for Wait states in interfacing slow memory devices and explain how to generate Wait states.
- Explain the internal structure, interfacing requirements, and the need for refreshing dynamic memory.
- Illustrate the interfacing of a 16K dynamic memory chip with the Z80.
- Explain address decoding using a decoder and a PROM programmer.
- Design a memory system for given specifications.
- Explain the need for data transfer using the Direct Memory Access technique and the functions of BUSRQ (Bus Request) and BUSAK (Bus Acknowledgment) signals of the Z80.
- Explain the differences between the three modes of DMA data transfer: byte, burst, and continuous.
- Explain the block diagram of the Z80 DMA controller and its interfacing with the Z80.

## 16.1 INTERFACING MEMORY USING WAIT STATES

---

In interfacing memory with the microprocessor, the interfacing circuit must satisfy the timing requirements of both the microprocessor and the memory chip. In Chapter 4, we assumed that memory response can match the execution speed of the microprocessor, but this assumption is invalid in some situations. Because of cost considerations, memory chips with slow response time are occasionally used in microprocessor-based systems. Therefore, it is necessary to synchronize the execution speed of the microprocessor with the response time of memory. This can be accomplished by using the Wait signal input to the Z80 microprocessor.

- WAIT (Wait)—This is an active low signal, input to the Z80 as an external request from a slow peripheral (or memory), to indicate that the peripheral is not yet ready for data transfer. The Z80 samples the WAIT line at the falling edge of  $T_2$  of each machine cycle, and if the WAIT line is low, it adds one T-state ( $T_w$ ) as a Wait state to its machine cycle. Then it samples the  $T_w$  state and adds an additional Wait state if the WAIT signal is still low and continues to add Wait states until the WAIT signal becomes inactive. During this time, the Z80 extends the time of control signals and preserves the contents of all the buses. Thus, the WAIT signal can be used to synchronize the response time of any type of peripheral.

To ascertain whether a given memory chip is too slow in comparison with the execution speed of the microprocessor and needs Wait states to synchronize the data transfer, we must examine the timing requirements of the microprocessor and the response time of the memory.

### 16.11 Z80 Machine Cycles and Memory Access Time

In Chapter 3, we examined three Z80 machine cycles: Opcode Fetch, Memory Read, and Memory Write. The Opcode Fetch machine cycle is shown here again in Figure 16.1 with

precise timing for the Z80 with a 2.5 MHz clock. The read timing of the Opcode Fetch is the most restrictive because the microprocessor begins to read the data byte at the rising edge of  $T_3$ ; in the Memory Read cycle, it reads the byte at the falling edge of  $T_3$ . If the memory chip can respond adequately in the Opcode Fetch cycle, we need not be concerned with the Memory Read cycle.

Figure 16.1 shows the time interval  $T_{AD}$ ; this is the interval between the time the Z80 places the address on the address bus and the time it has to read the byte. In a 2.5 MHz system (400 ns clock period), the  $T_{AD}$  is 605 ns. This is calculated by subtracting the output address delay  $T_{D(AD)}$  and the data set-up time  $T_{SD}$  on the data bus from the two clock periods (see Figure 16.1).

Allowable time interval for Z80 to read =  $2 \times$  Clock Period – Address Delay  
 data after placing the address on the – Data Set-Up Time  
 address bus

$$\begin{aligned} T_{AD} &= (2 \times T) - T_{D(AD)} - T_{SD} && \text{(Eq. 16.1)} \\ &= 2 \times 400 \text{ ns} - 145 \text{ ns} - 50 \text{ ns} \\ &= 605 \text{ ns*} \end{aligned}$$

\*These specifications are obtained from Z80 AC characteristics.

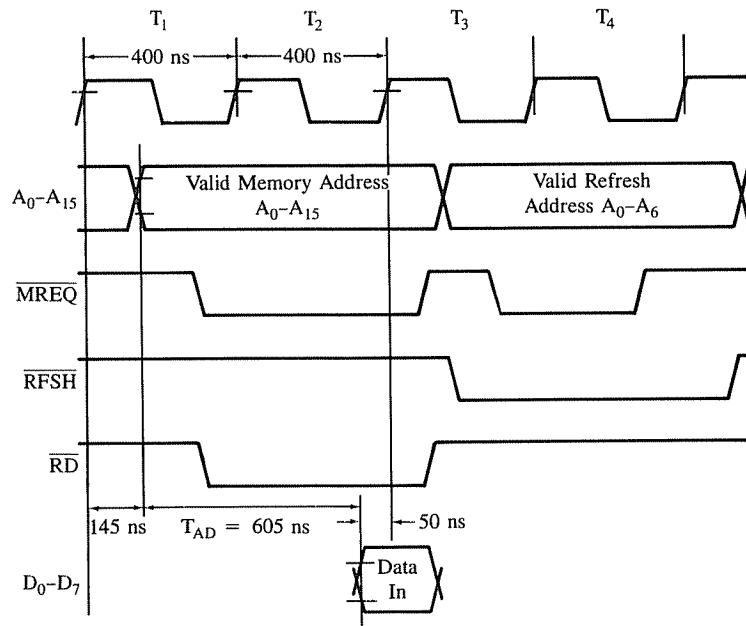


FIGURE 16.1

Z80 Opcode Fetch Machine Cycle

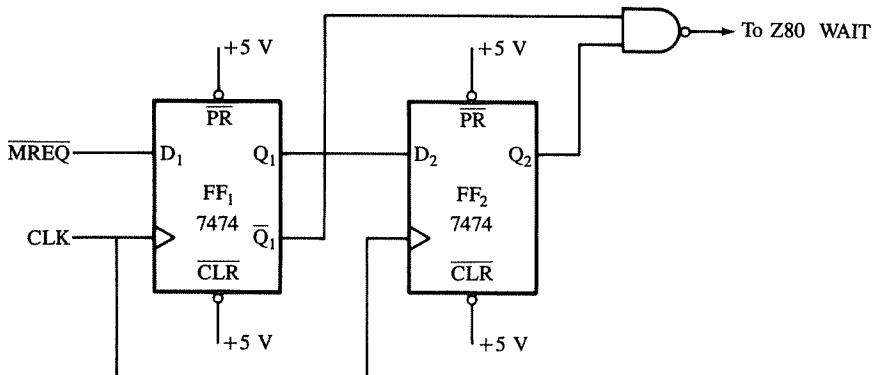
SOURCE: Adapted from *Memory Data Book and Designer's Guide*, p. x-40; courtesy of Mostek Corporation.

The memory response is defined in terms of the memory access time  $T_{AC}$ ; this is the delay between the time the memory address is placed on the address bus and the data byte is placed on the data bus. The memory access time plus the delay in the address decoding network must be less than  $T_{AD}$ ; if it is more than  $T_{AD}$ , we must add Wait states.

### 16.12 Generating Wait States

In the definition of the WAIT signal, the only requirement to add Wait states is to keep the WAIT signal low when the Z80 samples it during the  $T_2$  state. Figure 16.1 shows that the Memory Request (MREQ) goes active after the falling edge of  $T_1$ . The MREQ signal can be used, after the delay of one clock period, to activate the WAIT signal, as shown in Figure 16.2.

Figure 16.2 shows that a WAIT signal can be generated by ANDing the outputs of the two edge-triggered flip-flops and using the MREQ as an input to the flip-flop FF<sub>1</sub>. The MREQ stays high unless the microprocessor is accessing memory; therefore, we can safely assume that Q<sub>2</sub>, the output of FF<sub>2</sub>, is initially high. The MREQ goes low after the falling edge of T<sub>1</sub> (Figure 16.1); thus, Q<sub>1</sub> goes high in the next clock period at the rising edge of T<sub>2</sub> (the flip-flop is positive-edge-triggered) and generates the Wait signal, which should be connected to the WAIT pin of the Z80. Because WAIT is low at T<sub>2</sub>, the Z80 extends the machine cycle by one clock period,  $T_w$ . At the next cycle, Q<sub>2</sub> goes low (Q<sub>1</sub> was low in the previous cycle), causing the WAIT signal to go inactive. The circuit shown in Figure 16.2 adds one Wait state to all memory access cycles in the system. To use the Wait state for a particular memory chip in the system, the input MREQ should be ANDed with the decoded address of the chip as shown in the next example.

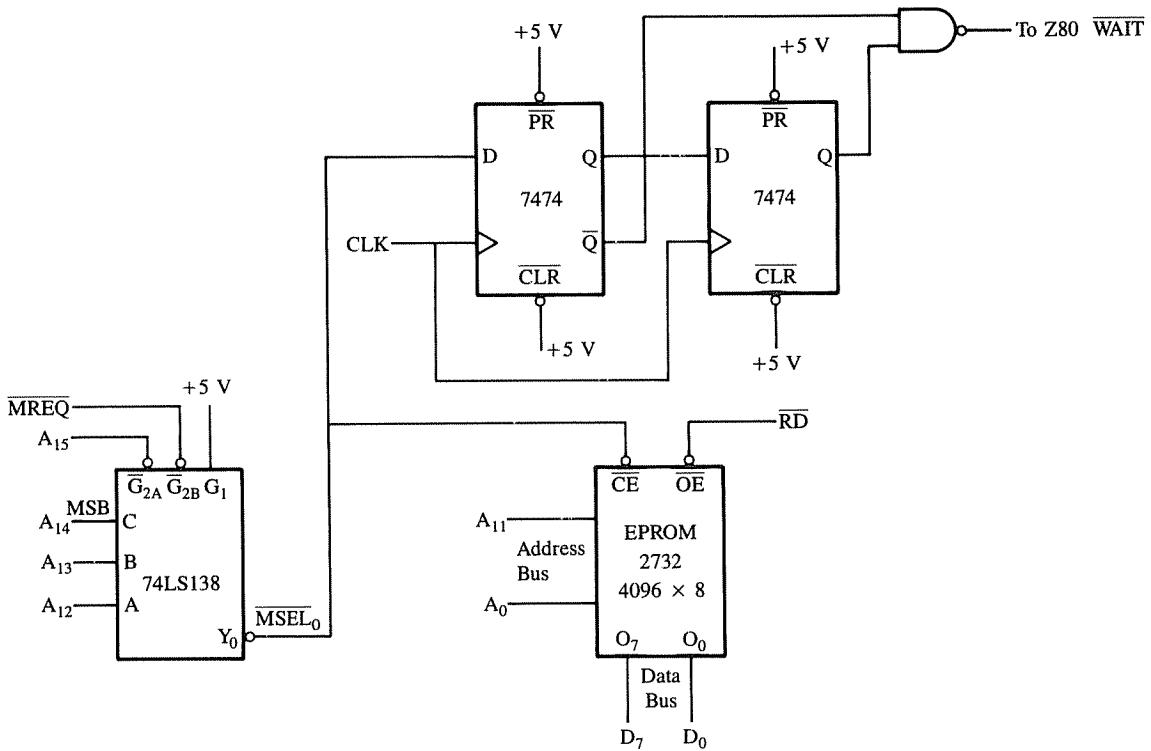


**FIGURE 16.2**  
Generating One Wait State

---

**Example  
16.1**

Figure 16.3 shows the circuit (from Figure 4.7) interfacing the 2732 EPROM with a 4 MHz Z80A system; the circuit includes a Wait state circuit. The Z80A address delay is 110 ns and data set-up time is 35 ns. The memory access time for the EPROM is given as



**FIGURE 16.3**  
Interfacing EPROM 2732 With a Wait State

450 ns, and the delay in the decoder is 40 ns. Explain the timing, and calculate the number of Wait states required.

In this system, the clock period is 250 ns. In the Opcode Fetch cycle, the Z80 reads the data byte at the rising edge of  $T_2$  cycle. Therefore, the allowable microprocessor read time is (see Eq. 16.1)

$$T_{AD} = 2 \times 250 \text{ ns} - 110 \text{ ns} - 35 \text{ ns} = 355 \text{ ns}$$

The EPROM would require

$$\begin{aligned} \text{Memory Read Time} &= \text{Memory Access Time} + \text{Address Decoding Delay} \\ &= 450 \text{ ns} + 40 \text{ ns} = 490 \text{ ns} \end{aligned}$$

These calculations show that the memory requires 490 ns and that the microprocessor, without the Wait states, would begin to read in 355 ns after placing the address on the bus. After adding one Wait state, the microprocessor read time is extended to 605 ns (355 ns +

**Solution**

250 ns). This leaves 115 ns (605 ns – 490 ns) as a safety margin for the memory read time.

The interfacing circuit shown in Figure 16.3 is a combination of two circuits: Generating one Wait State (Figure 16.2) and the memory interfacing circuit (Figure 4.7). However, in this circuit, the input to the flip-flop of the Generating Wait State circuit is modified; the input is the gated signal of the MREQ and the decoded address from the 3-to-8 decoder. Thus, whenever the Z80 accesses the EPROM, this circuit will add one Wait state to the memory machine cycles without affecting the performance of other memory chips in the system.

---

## 16.2 INTERFACING DYNAMIC MEMORY

---

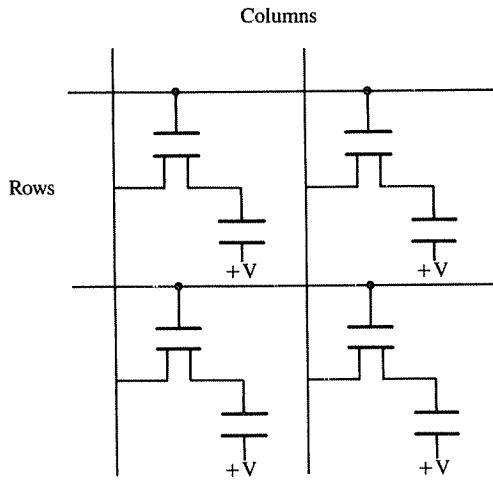
The dynamic Read/Write memory stores bit information in the form of a capacitive charge, and its internal structure differs from that of static memory. It is organized in the square matrix format of rows and columns, and its address lines are multiplexed. Therefore, to interface dynamic memory with the microprocessor, additional circuitry must be designed to address rows and columns separately. Furthermore, each memory cell needs to be refreshed at least every two milliseconds to retain the stored information. The circuitry necessary for refreshing the dynamic memory is built-in Z80 architecture; in other microprocessors, the refreshing is performed by external logic or LSI devices such as a Dynamic Memory Controller.

In the following sections, we will examine the internal structure of dynamic memory and its timing requirements, and design an interfacing circuit.

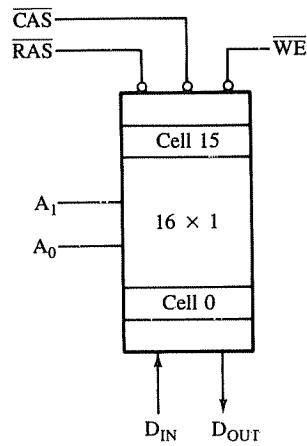
### 16.21 Dynamic Memory: Structure and Addressing

The dynamic memory consists of MOS transistors, which store information as capacitive charge and are internally arranged in the matrix format. Figure 16.4(a) shows four such cells with two rows and two columns. Figure 16.4(b) shows the representation of 16 cells with four rows and four columns with 2-to-4 decoders and latches. Thus, two input lines  $R_1$  and  $R_2$  can identify four rows when strobe line  $\overline{RAS}$  (Row Address Strobe) is active, and two lines  $C_1$  and  $C_2$  can identify four columns when the strobe line  $\overline{CAS}$  (Column Address Strobe) is active. The lines  $R_1$ ,  $R_2$ ,  $C_1$ , and  $C_2$  are connected together, and in turn connected to two address lines  $A_1$  and  $A_0$ . Thus, with two address lines and two strobe lines, we can select any of the cells. For example, by sending address 11 and asserting the  $\overline{RAS}$  line, the address 11 will be latched and row 3 will be selected, and if we follow that by sending the address 01 and asserting the  $\overline{CAS}$  line, we select column 1. Therefore, by sending the address 01 11, we can select cell number 7. Of course, we need to know the timings of when to assert  $\overline{RAS}$  and  $\overline{CAS}$  strobes. After selecting the cell, if we want to write into the cell, the Write signal must be asserted.

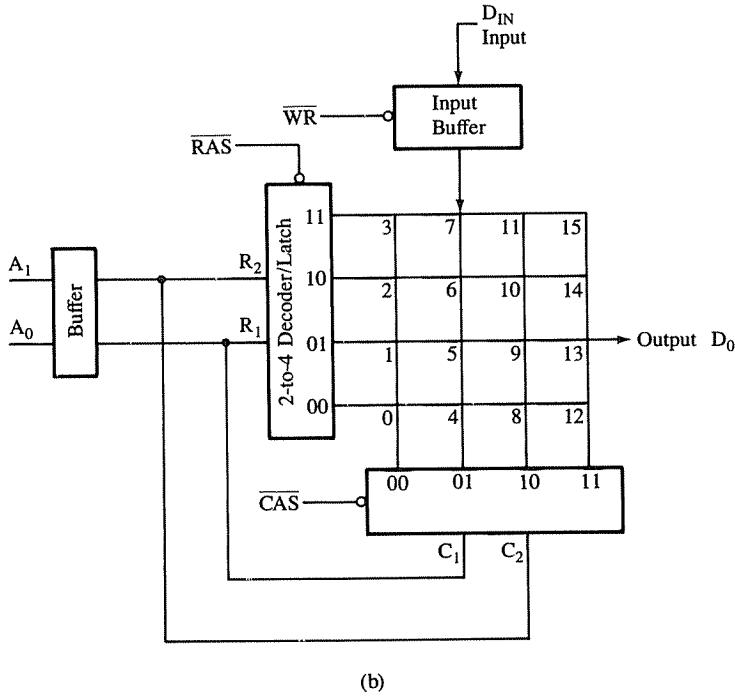
The memory structure shown in Figure 16.4(b) has one input line and one output line; thus, we can write 16 bits or read 16 bits from this memory. This memory is repre-



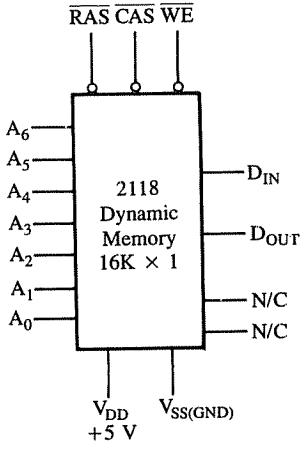
(a)



(c)



(b)



(d)

**FIGURE 16.4**

(a) Dynamic Memory Cells (b) Dynamic Memory: Internal Structure (c) Logic Symbol for 16 × 1 Dynamic Memory (d) Logic Symbol: Intel 2118 Dynamic Memory in a 16-pin Package

sented by the logic symbol, shown in Figure 16.4(c) with two address lines, two strobe lines, one control signal WE, one input line, and one output line; the size of this chip is  $16 \times 1$ . An 8-bit microprocessor such as the Z80 requires an 8-bit memory word; thus, we will need eight chips to have memory of 16 bytes. Figure 16.4(d) shows the logic symbol of the Intel 2118 family of  $16,384 \times 1$  dynamic memory. To address 16K memory, we need 14 address lines,  $A_{13}-A_0$ —seven rows and seven columns; however, the chip shows only seven lines  $A_6-A_0$ . These lines are multiplexed; the low-order address  $A_6-A_0$  is used for rows and the high-order address  $A_{13}-A_7$  is used for columns.

To appreciate the complexities in interfacing this dynamic memory chip, let us first recall the steps for interfacing a static memory in relation to the Z80 Read/Write operations. The Z80

1. places a 16-bit address on the address bus.
2. sends the MREQ signal to indicate that the 16-bit address is on the address bus.
3. sends the control signal RD or WR.

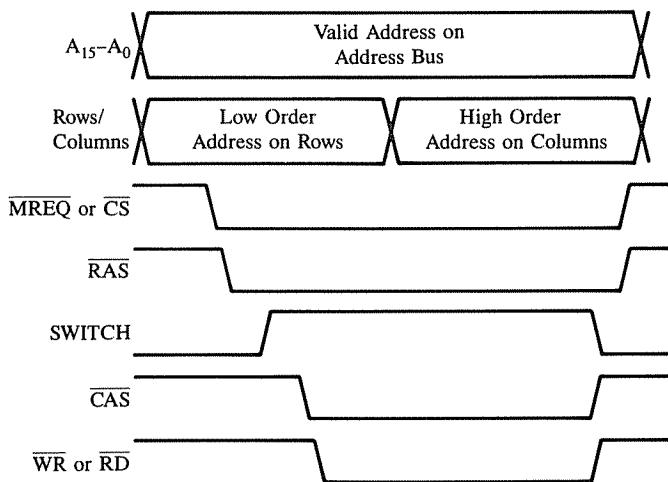
Therefore, to interface a static memory, the designer has to perform the following three steps:

1. Generate the CS signal by decoding the high-order bus and connect it to the CE pin of the memory chip.
2. Generate the MEMRD signal by ANDing RD and MREQ and connect it to the RD pin of the memory chip.
3. Generate the MEMWR signal by ANDing WR and MREQ and connect it to the WR pin of the memory chip.

In interfacing dynamic memory, the Z80 operations remain the same; it is the responsibility of the designer to generate the signals that are necessary for the dynamic memory chip. The steps and the issues involved are as follows:

1. Generate the CS signal to identify this chip among many memory chips in the system. However, there is no CS pin on the memory chip.
2. Isolate the low-order address from the high-order address.
3. Place the low-order address on the memory address lines and generate RAS (Row Address Strobe) to inform the memory that the row address is on the address lines.
4. Switch from the low-order address to the high-order address and place the high-order address (column address) on the memory address lines.
5. Generate the CAS (Column Strobe Signal) to inform the memory that the high-order address is on the address lines.
6. Use the WR signal to write into the memory or RD signal to read from memory. However, there is no RD pin on the memory chip.

The timing diagram for the memory is shown in Figure 16.5. How to generate these signals is discussed in the next section.



**FIGURE 16.5**  
Timing Requirements to Read from or Write into Dynamic Memory

## 16.22 Designing Circuits and Generating Timing Signals for Dynamic Memory

To generate timing signals shown in Figure 16.5, we will refer to the six steps listed above and use the memory chip 2118 as an illustration.

### Step 1: Generating the $\overline{CS}$ signal

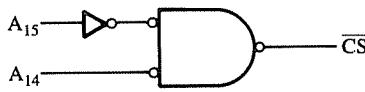
To address 16K memory, we need 14 address lines  $A_{13}-A_0$ ; only two address lines  $A_{15}-A_{14}$  remain. We can use a 2-to-4 decoder or a simple two-input gate with an inverter as shown in Figure 16.6. The  $\overline{CS}$  is generated when  $A_{15}$  is logic 1 and  $A_{14}$  is logic 0.

### Step 2: Isolating the low-order address from the high-order address

This can be accomplished by using the 74LS157 multiplexer shown in Figure 16.7(a). The device has two control signals: SELECT and  $\overline{STROBE}$ , and the  $\overline{STROBE}$  must be active (low) for the device to function. When the SELECT is low, the data on input lines 1A–4A are available on the output lines, and when the SELECT is high, the data on lines 1B–4B are available on the output lines.

To separate the low-order address from the high-order address, the low-order address lines can be connected to the A input lines of the multiplexer and the high-order

**FIGURE 16.6**  
Address Decoding to Generate Chip Select Signal



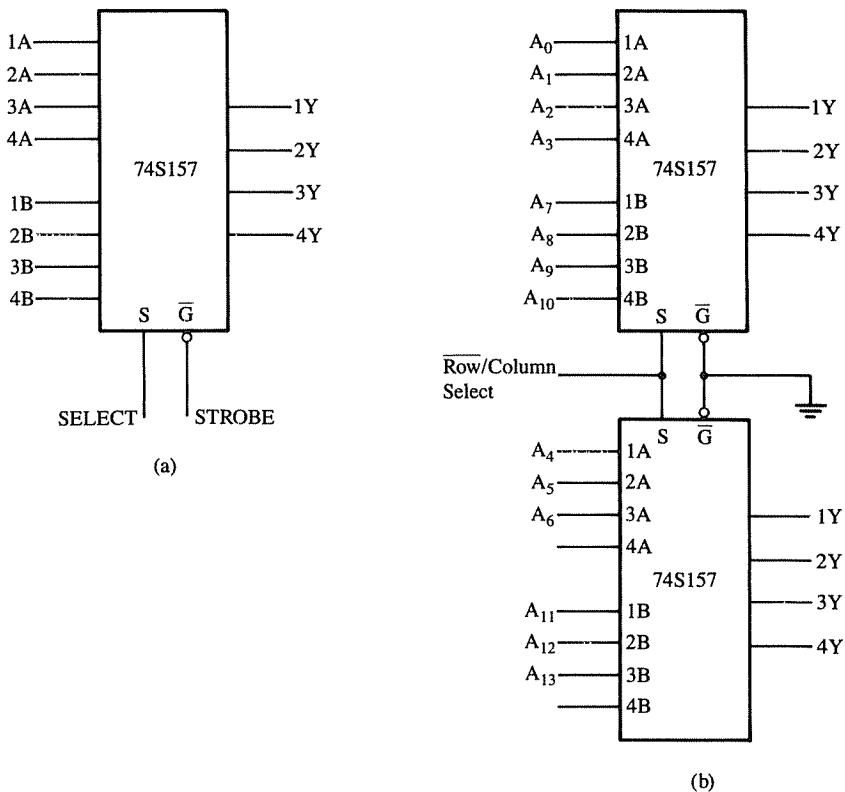


FIGURE 16.7

(a) Logic Symbol: 74S157 Multiplexer (b) Z80 Address Lines to Multiplexer

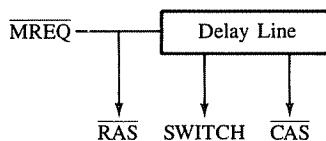
address lines can be connected to the B input lines. The Intel 2118 memory chip has seven multiplexed address lines for 14-bit addresses; we therefore need to use two multiplexers as shown in Figure 16.7(b). When the SELECT is low, the address A<sub>6</sub>–A<sub>0</sub> is placed on the output lines, and when the SELECT is high, the address A<sub>13</sub>–A<sub>7</sub> is placed on the output lines.

**Step 3:** Generating RAS (Row Address Strobe) and placing the low-order address on the memory address lines

The RAS informs the memory that the row address is on the address lines. In the Z80, assertion of the MREQ (Memory Request) signal indicates that the 16-bit address is on the address bus. Therefore, the MREQ can be used as the RAS signal, and by keeping the SELECT low, the low-order address can be placed on the memory address lines. On the falling edge of the RAS, the row address is latched by the memory.

**Step 4:** Switching from the row address to the column address

**FIGURE 16.8**  
Generating  $\overline{\text{RAS}}$  and  $\overline{\text{CAS}}$  Signals  
Using a Delay Line



This is accomplished by asserting the SELECT line high. The multiplexer can be switched by using the  $\overline{\text{MREQ}}$  signal with appropriate delay. This delay can be generated by using a delay line circuit as shown in Figure 16.8 or using propagation delays in logic gates. When the delay line, connected to the SELECT pin of the multiplexer, is asserted, the column address ( $A_{13}-A_7$ ) is placed on the memory address lines.

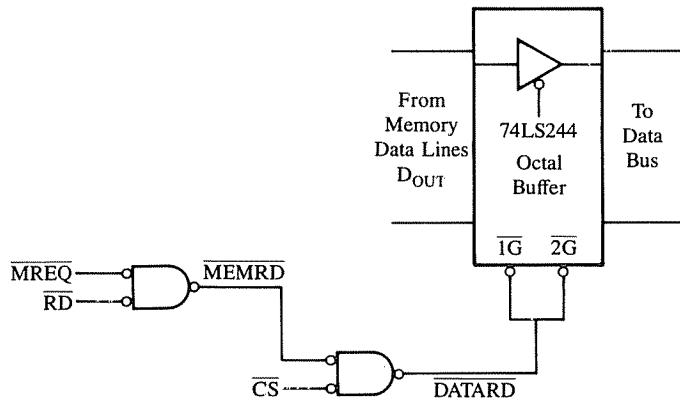
#### Step 5: Generating the $\overline{\text{CAS}}$ (Column Address Strobe) signal

The  $\overline{\text{CAS}}$  signal can be generated by delaying the  $\overline{\text{MREQ}}$  signal as shown in Figure 16.8. When the  $\overline{\text{CAS}}$  is asserted, the column address is latched by the memory chip.

#### Step 6: Writing into or reading from a memory cell

To write into a memory cell, three signals— $\overline{\text{RAS}}$ ,  $\overline{\text{CAS}}$ , and  $\overline{\text{WR}}$ —must be asserted. The  $\overline{\text{WR}}$  signal can be asserted before the  $\overline{\text{CAS}}$  signal (the early-write technique) or after the  $\overline{\text{CAS}}$  signal. When these three signals are active, the data bit on the input line  $D_{\text{IN}}$  is latched into the internal register.

To read from a memory cell, the cell must be selected by asserting the  $\overline{\text{RAS}}$  and  $\overline{\text{CAS}}$  signals. After the selection of the cell, the Z80 can assert its  $\overline{\text{RD}}$  signal to read from the cell; however, the memory chip does not have a  $\overline{\text{RD}}$  pin. Therefore, the Read operation requires a buffer that can be enabled to read the data bit as shown in Figure 16.9.



**FIGURE 16.9**  
Output Port to Read from Dynamic Memory

### 16.23 Refreshing Dynamic Memory

As mentioned earlier, the bits are stored as capacitive charges in the dynamic memory cells and the charge leaks. Therefore, to retain information, the cells must be refreshed every 2 ms. A memory cell is automatically refreshed simply by accessing its row or by reading from or writing into the cell. However, there is no guarantee that each cell can be refreshed within a 2 ms period during the normal execution of a program. Therefore, additional circuitry must be used to refresh the cells in dynamic memory.

The commonly used technique to refresh dynamic memory is to place a row address on the memory address lines and assert the RAS signal without the RD, WR, or the CAS signals; this is called RAS-only refresh. The 2118 memory chip has seven rows with 128 cells that can be refreshed by using a 7-bit counter. The counter provides a 7-bit address, and by cycling the counter through 128 row addresses within 2 ms, the entire memory chip can be refreshed. The Z80 has a specially built register R which is used as a refresh counter and a signal called RFSH (Refresh).

Figure 16.1 shows the timing of the Z80 Opcode Fetch cycle. The time during  $T_3$  and  $T_4$ , when the Z80 decodes the instruction internally and the address bus would otherwise be idle, is used in Z80-based systems for refreshing the dynamic memories. This process is called transparent refresh. During  $T_3$ , the RFSH signal goes low, indicating that the lower seven bits of the address bus contain a refresh address, and the MREQ is asserted low again. To latch the row address and refresh the cell, the RAS signal must be generated when the RFSH is active; this can be done by ANDing the MREQ and the RFSH signals. In addition, the CAS signal must be kept inactive during this period or the contents of the memory cell may be lost. The row address to refresh the cell is supplied by the contents of Refresh register R, and the address is incremented automatically after each Opcode Fetch cycle. The refresh process is automatic; the programmer can load an address into register R for testing purposes, but it is not necessary.

The refreshing technique discussed here can refresh 128 rows of a 16K dynamic memory. This technique is not, however, necessarily limited to 16K memory. If the system includes four 2118s (64K memory), the same refresh address can be used to refresh one row in each of the four memory chips if the RAS is generated without the Chip Select. We will combine the circuits discussed in this section in Section 16.3, Interfacing the 2118 Dynamic Memory with the Z80.

This refreshing technique is ideal in that it does not slow down the microprocessor operations. However, it has some limitations. It is based on the  $M_1$  cycle's being executed at least 128 times within 2 ms. This may not be possible if the external Reset, Wait, and Hold signals are active more than 1 ms. Even if this technique can refresh several 16K memory chips, it cannot be used for a memory chip larger than 16K. Then, LSI devices such as Dynamic RAM Controllers are commonly used. These controllers not only refresh the dynamic memories but also provide such necessary timing signals as RAS and CAS.

## ILLUSTRATION: INTERFACING THE 2118—16K R/W DYNAMIC MEMORY—WITH THE Z80

# 16.3

This illustration is concerned with interfacing a 16K R/W dynamic memory with the Z80. The illustration synthesizes the concepts we discussed in the previous section.

### 16.31 Problem Statement

Design a circuit to interface the 2118—16K dynamic memory—with the Z80 for the memory map starting at  $8000_H$ . Generate the necessary timing signals assuming the system clock is 2.5 MHz, and use the Z80 Refresh signal and register R to refresh the memory cells within 2 ms.

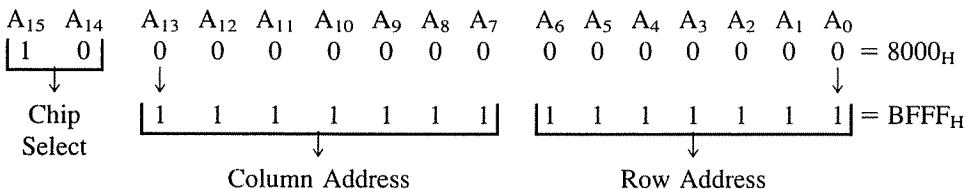
### 16.32 Problem Analysis

The interfacing of dynamic memory involves three aspects: assigning the memory map, generating timing signals, and refreshing memory cells. The steps are as follows:

1. Decode the high-order address lines to assign the map.
2. Generate RAS, CAS, and SELECT signals with appropriate delays.
3. Use the RAS signal to latch the row address, the SELECT signal to switch from the RAS to the CAS signal, and the CAS signal to latch the column address.
4. Use the Z80 WR signal to write into the selected cell.
5. Generate the Memory Read signal to enable a buffer as an input port.
6. Refresh the memory cell during  $T_3$  and  $T_4$  states of the Opcode Fetch cycle.

### 16.33 Interfacing Circuit and Its Operation

In this circuit design, the assignment of the memory map with the starting address  $8000_H$  is easy. The 16K memory requires 14 address lines  $A_{13}$ – $A_0$  to address memory cells; thus, only two address lines,  $A_{15}$  and  $A_{14}$ , remain. These address lines can be decoded either using a simple gate and an inverter or a 2-to-4 decoder. Figure 16.10 shows that the CS signal is generated using the gate 74LS32 with an inverter. The memory map of this circuit ranges from  $8000_H$  to  $BFFF_H$  as shown.



### GENERATING THE $\overline{\text{RAS}}$ SIGNAL

The  $\overline{\text{RAS}}$  signal should be active when the row address is placed on the memory lines and when the  $\overline{\text{RFSH}}$  signal is asserted to refresh the memory cells. However, the  $\overline{\text{CS}}$  signal is not necessary for refreshing. Therefore, the  $\overline{\text{CS}}$  and  $\overline{\text{RFSH}}$  signals are logically ORed as shown in Figure 16.10. The  $\overline{\text{RAS}}$  is generated by ANDing  $\overline{\text{CS}}$  or  $\overline{\text{RFSH}}$  with the  $\overline{\text{MREQ}}$ . The time delay in generating the  $\overline{\text{RAS}}$  signal in relation to the  $\overline{\text{MREQ}}$  will be 37 ns, the sum of the delays in 74LS04 and NAND gate. The delay in generating the  $\overline{\text{CS}}$  is irrelevant because the address is placed on the address bus 300 ns before the  $\overline{\text{MREQ}}$  is asserted.

### CONNECTING THE ROW AND COLUMN ADDRESSES

Figure 16.11 shows that the row address lines  $A_6$ – $A_0$  are connected to A input and the column address lines  $A_{13}$ – $A_7$  are connected to B input of the two multiplexers 74S157. The seven output lines of the multiplexers are connected to the multiplexed address lines  $A_6$ – $A_0$  of the memory chip, and the eighth output line of the multiplexer is connected to the  $\overline{\text{CAS}}$  signal of the memory chip. The input lines 4A and 4B of the second multiplexer are tied to +5 V and ground, respectively. When the SELECT is switched from low to high, the eighth output line goes from high to low (from +5 V to ground), and the  $\overline{\text{CAS}}$  signal is asserted (active low). Figure 16.11 shows the details of only one memory chip, rather than eight chips, to avoid clutter.

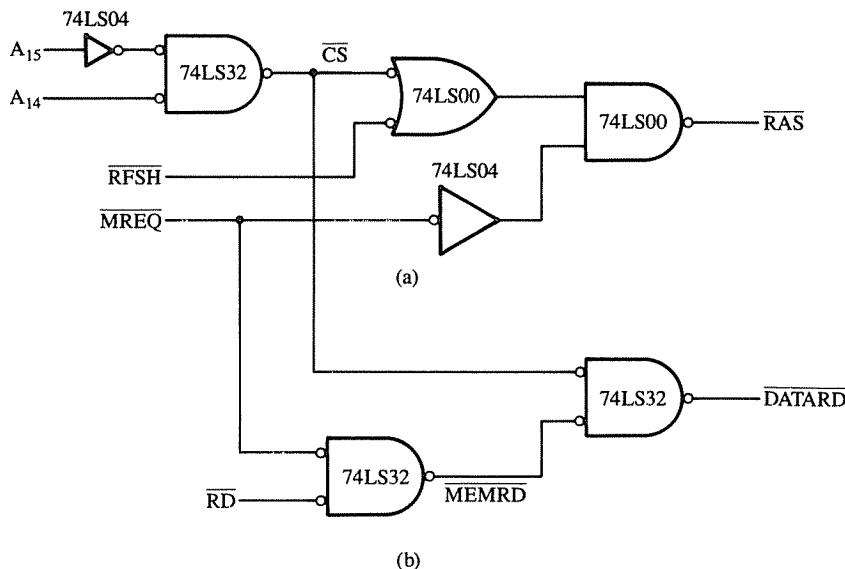
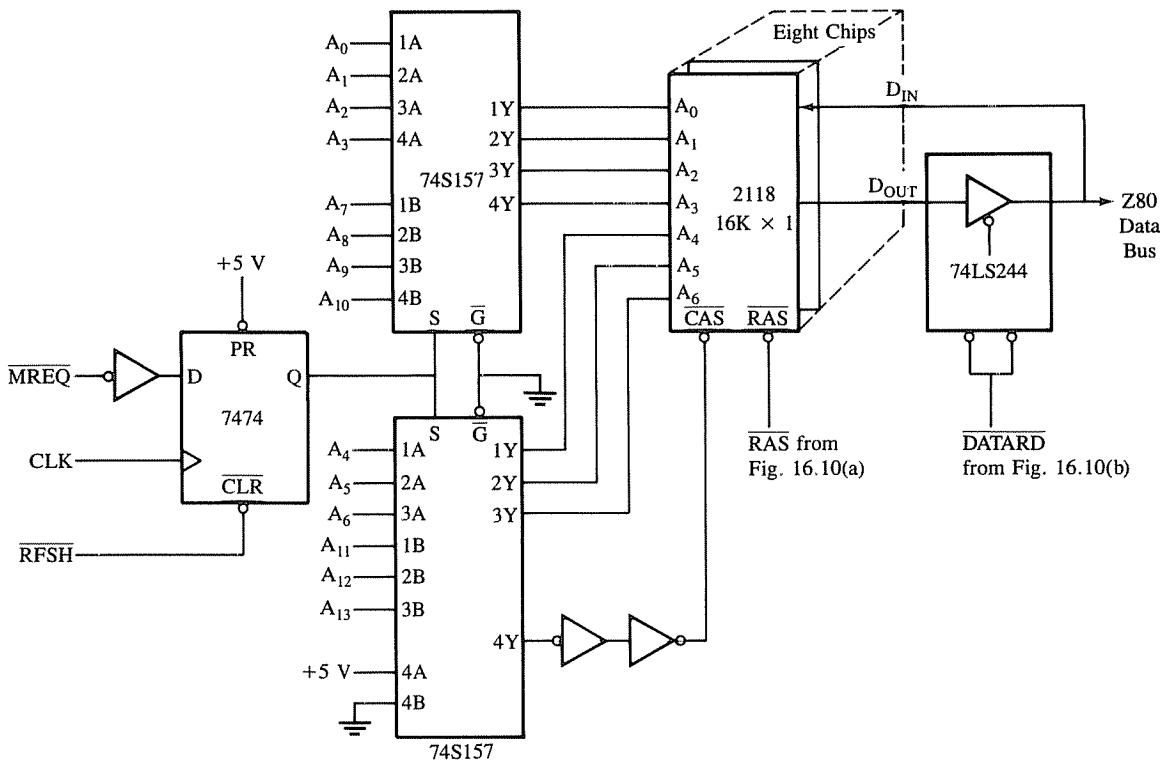


FIGURE 16.10

(a) Generating  $\overline{\text{RAS}}$  Signal When  $\overline{\text{CS}}$  or  $\overline{\text{RFSH}}$  Is Active (b) Generating  $\overline{\text{MEMRD}}$  and  $\overline{\text{DATARD}}$  to Read from Memory



**FIGURE 16.11**  
Interfacing Dynamic Memory

SOURCE: Adapted from *Memory Data Book and Designer's Guide*, p. x-45; courtesy of Mostek Corporation.

#### PLACING THE ADDRESSES AND SWITCHING FROM RAS TO CAS

The SELECT signal of the multiplexer is controlled by the MREQ through the edge-triggered flip-flop 7474 (Figure 16.11). At the beginning of the Opcode Fetch cycle, the MREQ is high and the output of the flip-flop is low. Thus, the SELECT signal is low, and the row address is placed on the output lines of the multiplexer. When the MREQ is asserted, the RAS is generated 37 ns later because of the delay in the inverter and the NAND gate (Figure 16.10), and the row address is latched by the memory. The MREQ also changes the input to the flip-flop; however, the output of the flop-flop changes on the rising edge of the clock  $T_2$ . When the output of the flip-flop goes high, the SELECT is switched from low to high, and the column address  $A_{13}-A_7$  is placed on the address lines. The CAS is delayed by 37 ns by using two inverters, thus allowing sufficient time for the column address to settle. When the CAS is asserted, the column address is latched.

### **READING FROM AND WRITING DATA INTO MEMORY**

The Memory Read signal is generated by ANDing  $\overline{CS}$ ,  $\overline{MREQ}$ , and  $\overline{RD}$  (Figure 16.10(b)); this is similar to generating the  $\overline{MEMRD}$  signal for static R/W memory. However, this memory chip does not have a  $\overline{RD}$  pin; therefore, the data bit is read by using the buffer 74LS244 as a memory-mapped input port. Figure 16.11 shows one input and one output data line; the other seven data lines are not shown here.

To write data into a selected cell, the  $\overline{WR}$  signal, which is connected to  $\overline{WR}$  pin of the memory, must be asserted. The  $\overline{WR}$  signal can also be asserted before the  $\overline{CAS}$  signal (early-write); we have not used that technique in this illustration.

### **REFRESHING THE MEMORY CELL**

During the  $T_3$  state of the Opcode Fetch cycle, the  $\overline{RFSH}$  is asserted and the address from the  $R$  register is placed on the address bus. The  $\overline{RAS}$  signal is asserted again when the  $MREQ$  goes low, the row address is latched by the memory, and the cell is refreshed. However, the  $CAS$  is held inactive during the refresh by using the  $\overline{RFSH}$  signal to reset the flip-flop. Thus, this operation is not confused with a Read or Write operation.

### **TIMING CALCULATIONS AND MICROPROCESSOR READ TIME**

In Section 16.1 we calculated that the Z80 with a 2.5 MHz clock begins to read data 450 ns after asserting the  $\overline{MREQ}$ . In dynamic memory, the sum of the memory access time and the delays in generating various signals must be less than the microprocessor read time. In Figures 16.10 and 16.11, the delays in reference to  $\overline{MREQ}$  can be calculated as follows:

1. $\overline{MREQ}$ to $\overline{RAS}$ : Delay in 7404 and 7400	= 37 ns
2. $\overline{RAS}$ to $\overline{SELECT}$ : Rising clock + flip-flop delay	= 73 ns
3. $\overline{SELECT}$ to $\overline{CAS}$ : Multiplexer and inverter delays	= 52 ns
4. $\overline{CAS}$ Access Time:	= 165 ns
5. $\overline{CAS}$ to Octal Buffer:	= 12 ns
	Total Delay
	<u>339 ns</u>

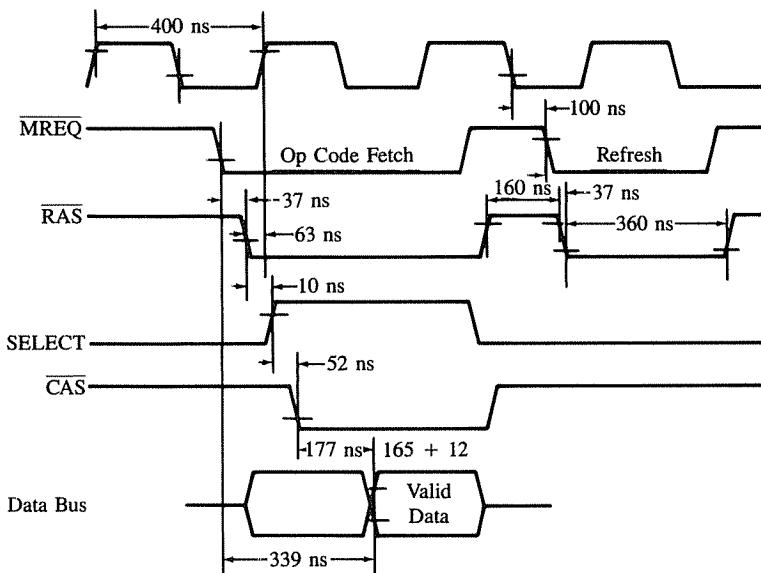
The timings are shown in Figure 16.12. The total delay is 339 ns, and the microprocessor would begin to read at 450 ns; thus, we have a 111 ns safety margin.

## **16.4 DESIGNING MEMORY SYSTEMS**

---

In designing memory systems for microprocessor-based products, several critical issues must be considered at the beginning of the design cycle. These issues are

1. Cost effectiveness
2. Ease of converting a product from design to production
3. Design flexibility and future upgrading

**FIGURE 16.12**

Timing: Dynamic Memory

SOURCE: Adapted from *Memory Data Book and Designer's Guide*, p. x-46; courtesy of Mostek Corporation.

The first issue of cost effectiveness involves the unit price of the memory chips to be used in the system and the available space on the printed circuit board (commonly known as the real estate of a PC board). Ideally, the designer would like to use the minimum space and components with the lowest unit price, and at times, these requirements may conflict.

In terms of selecting the memory chips, the memory map must be divided between R/W memory and Read Only Memory; the decision is generally based on the requirements of data retention. In the design stage, these requirements are generally not known; however, compatible memory chips are currently available so that RAM, ROM, and EPROM can be interchanged in the same socket. The next decision concerning the R/W memory is to select either static memory or dynamic memory. For small systems with 8K or less memory, the static memory has cost advantage. When memory size begins to approach 64K, dynamic memory has a distinct price advantage in spite of the additional cost of the refresh circuitry. For memory size between 8K and 32K, the integrated R/W memory (iRAM) appears to have cost advantage. Similarly, for Read Only Memory, we need to choose from among masked ROM, PROM, and EPROM. In the design stage, the EPROM is the best choice until the system is completely developed and debugged. The masked ROM becomes cost effective only for large production quantities.

The remaining two issues—the ease of converting the product from design to production and future expansion—can be solved by using Mostek's BYTEWYDE concept (also known as “byte-wide universal site”), whereby a designer can use the 28-pin DIP socket that can accommodate compatible RAMs, ROMs, EPROMs, or E<sup>2</sup>PROMs from memory size  $1K \times 8$  to  $32K \times 8$ , as shown in Figure 16.13(a). This socket can accept a 24-pin or a 28-pin memory chip; a 24-pin memory chip is inserted into the lower portion of the socket, leaving the top 4 pins unused. In this socket, 21 of the lower 24 pins are defined: ten address lines  $A_9$ – $A_0$ , eight data lines  $D_7$ – $D_0$ , the Chip Enable  $\overline{CE}$ , the Output Enable  $\overline{OE}$ , and the ground. The remaining pins are either defined by using a jumper or left as no connections. Figure 16.13(b) shows how pins are connected for  $1K \times 8$  and  $8K \times 8$  memory chips. This BYTEWYDE concept provides flexibility during the design stage, ease in changing over from a laboratory design to a production unit, and future expandability.

After the general considerations discussed above, we need to examine how to mix different sizes of memory chips in a system; this is illustrated in the next example.

## 16.41 Memory Design: Problem Statement

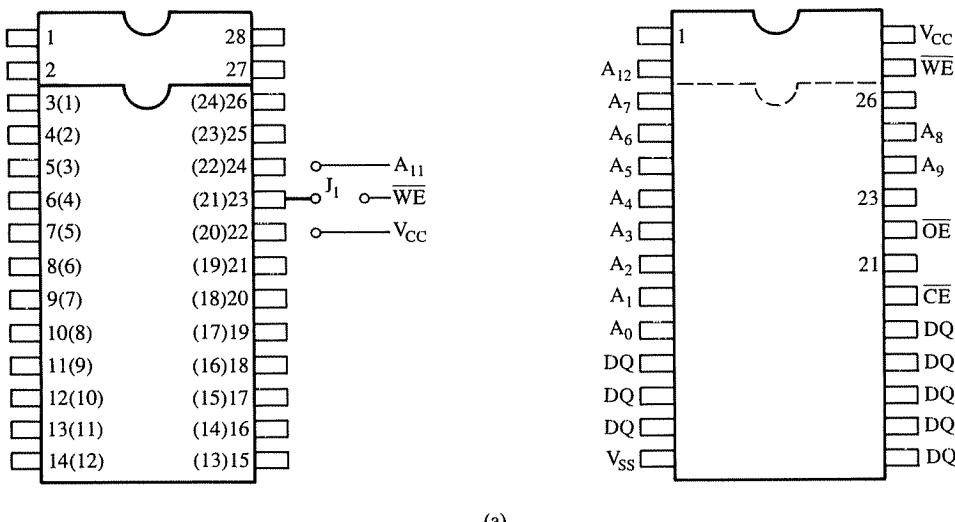
Design a memory system using the following memory devices. The memory map of ROM devices should begin at 0000, and the entire address range for ROMs should be continuous. The R/W memory address should be placed at  $8000_H$  or beyond. Illustrate the memory address decoding schemes using (a) a 3-to-8 decoder and (b) a  $256 \times 4$  PROM.

1. Two ROMs : MOSTEK MK37000 (8K × 8)
  2. One EPROM : 2732 (4K × 8)
  3. One R/W memory: CMOS 6116 (2K × 8)

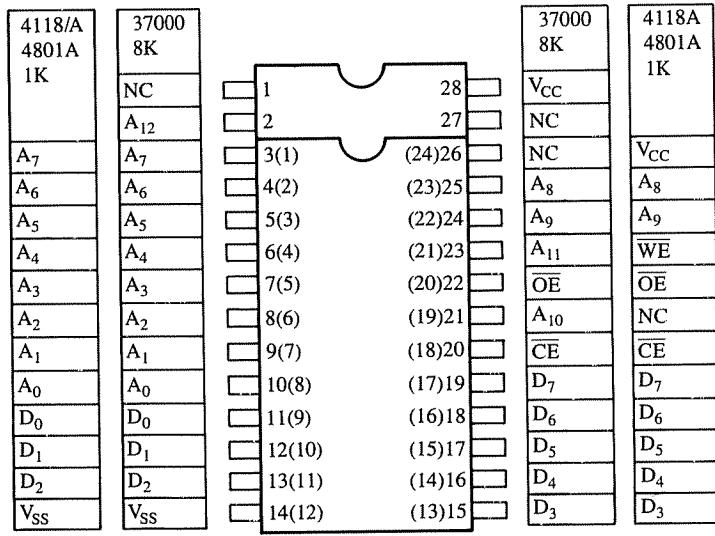
## 16.42 Address Decoding Using a 3-to-8 Decoder

This is the simplest technique to use for address decoding, and the memory map is largely influenced by the largest memory chip, ROM, which must be mapped at address 0000. The MK37000 has 8K registers (memory locations); thus, it would require 13 address lines for the memory chip. The remaining three address lines  $A_{13}$ – $A_{15}$  can be connected to the decoder, and all of them must be at logic 0 to place the beginning address at 0000. Figure 16.14 shows that the output  $Y_0$  of the decoder is connected to the  $\overline{CE}$  signal of the first MK37000, and to keep the memory map continuous, the  $Y_1$  output of the decoder is connected to the  $\overline{CE}$  of the second ROM chip. The memory addresses of the ROM chips are as follows:

$A_{15} \quad A_{14} \quad A_{13}$	$A_{12} \quad A_{11} \quad A_{10} \quad A_9 \quad A_8 \quad A_7 \quad A_6 \quad A_5 \quad A_4 \quad A_3 \quad A_2 \quad A_1 \quad A_0$	$0 \quad 0 = 0000$
$\boxed{0 \quad 0 \quad 0}$	$\downarrow$	$1 \quad 1 = 1FFF$
chip select		
$\boxed{0 \quad 0 \quad 1}$	$0 \quad 0 = 2000$	$\boxed{0 \quad 0 \quad 1} = 3FFF$
chip select	$\downarrow$	$1 \quad 1 = 3FFF$
		ROM <sub>1</sub>



(a)

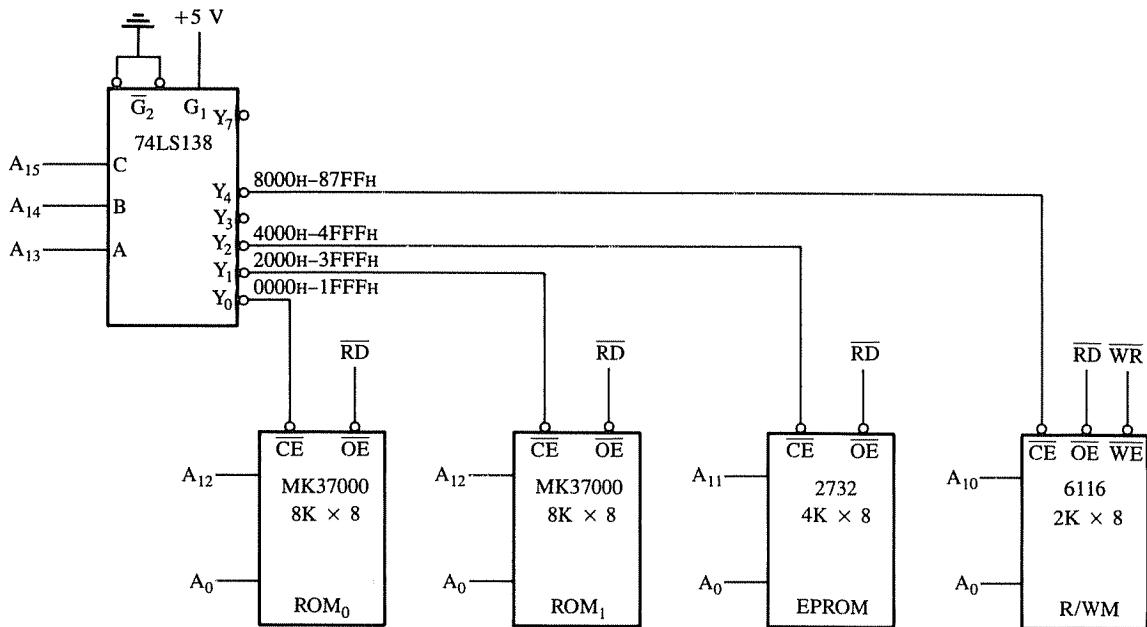


(b)

FIGURE 16.13

Bytewide Concept (a) 28-Pin Socket and Related Jumpers, and (b) Used to Connect 8K and 1K Memory in a 28-Pin Socket

SOURCE: Courtesy of Mostek Corporation.

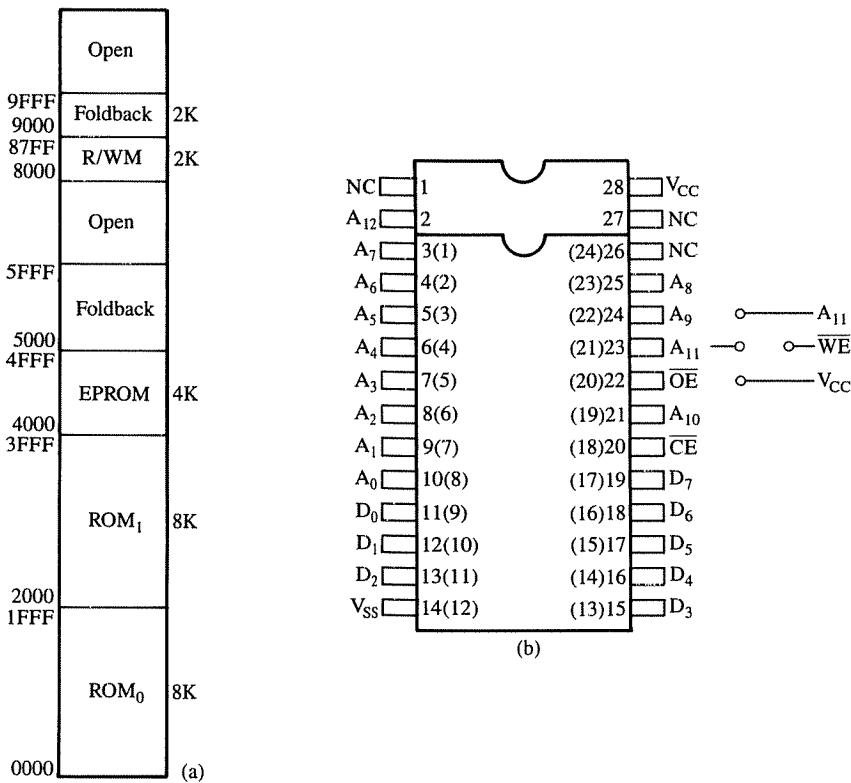


**FIGURE 16.14**  
Memory Board Design

The next memory chip is the EPROM with 4K registers; thus, only 12 address lines are required for the memory chip. Therefore, one address line, A<sub>12</sub>, must be left as “don’t care.” As a result, this memory chip will have a foldback memory of 4K. This conclusion can be drawn simply by examining the decoder circuit. Each output line of the decoder in this circuit can have a memory address block of 8K; therefore, any memory chip with less than 8K memory will have foldback memory addresses. Assuming the “don’t care” address line at logic 0, the memory map of the EPROM ranges from 4000<sub>H</sub> to 4FFF<sub>H</sub>, and the foldback addresses range from 5000<sub>H</sub> to 5FFF<sub>H</sub> as shown.

A <sub>15</sub>	A <sub>14</sub>	A <sub>13</sub>	A <sub>12</sub>	A <sub>11</sub>	A <sub>10</sub>	A <sub>9</sub>	A <sub>8</sub>	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>
0	1	0	(X)	0	0	0	0	0	0	0	0	0	0	0	0 = 4000
															↓
chip select															EPROM
0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0 = 5000
															↓
chip select															EPROM Foldback

The size of the R/W memory chip is 2K, and it must have the beginning address 8000<sub>H</sub> or higher. Therefore, it should be selected when A<sub>15</sub> = 1, and this can be accom-

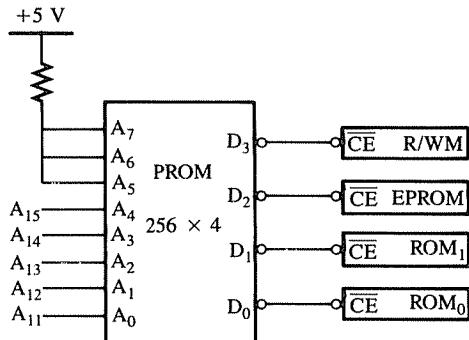


**FIGURE 16.15**  
 (a) Memory Map (b) Using Bytewide Concept

plished by connecting the Y<sub>4</sub> output of the decoder to the  $\overline{CE}$  of the memory chip. This memory chip requires 11 address lines; thus, two address lines, A<sub>12</sub> and A<sub>11</sub>, will assume “don’t care” logic status. The address of the R/W memory will range from 8000<sub>H</sub> to 87FF<sub>H</sub> assuming the “don’t care” lines at logic 0. The remaining 6K (of the decoder’s 8K block) from 8800<sub>H</sub> to 9FFF<sub>H</sub> will be the foldback memory. Figure 16.15(a) shows the entire memory map, and Figure 16.15(b) shows how to connect different sizes of memory chips using the 28-pin package and the BYTEWYDE concept. For example, to connect a 1K R/W memory chip, pin 23 is connected to  $\overline{WE}$  using the jumper, and pin 21 has no connection.

One of the drawbacks of this type of decoding technique is that it generates foldback memory addresses; thus, it wastes memory space and limits further expansion. However, in small systems this is not a serious problem. Another approach that avoids the foldback memory addresses is to use a separate decoder for each different size memory chip, but this increases the cost of the system and the size of the PC board.

**FIGURE 16.16**  
Address Decoding Using a PROM



### 16.43 Address Decoding Using a PROM

Another technique, commonly used in industrial products, is to decode the address bus using a PROM. Figure 16.16 shows a PROM (256 × 4) with eight address lines and four data lines. We can connect each data line to the Chip Enable  $\overline{CE}$  signals of memory devices; thus, we can interface four memory devices. Now we can program the PROM in such a way that for a given range of the address, only one data line will go active (low) while the others stay high.

In this memory design problem, the smallest memory size is 2K, which requires 11 address lines; the remaining five address lines must be decoded. The PROM has eight address lines; we can connect the five address lines A<sub>15</sub>–A<sub>11</sub> to the address lines A<sub>4</sub>–A<sub>0</sub> of the PROM, and the address lines A<sub>7</sub>–A<sub>5</sub> can be tied high. The five address lines A<sub>4</sub>–A<sub>0</sub> of the PROM can have 32 ( $2^5$ ) combinations, and we can determine the data output as shown in Table 16.1.

Table 16.1 shows that for the first 8K memory block of ROM<sub>0</sub>, the data line D<sub>0</sub> is active; other data lines are high. Similarly, for the second 8K memory block of ROM<sub>1</sub>, the data line D<sub>1</sub> is active while others are high. The PROM is decoded for 2K resolution; thus, every combination on A<sub>4</sub>–A<sub>0</sub> of the PROM provides a 2K memory map. Therefore, for 8K memory, we need D<sub>0</sub> and D<sub>1</sub> to be active for four combinations each. Similarly, D<sub>2</sub> is active for the next two combinations, and D<sub>3</sub> is active when the address begins at 8000<sub>H</sub>. For the open memory space, all data lines are high.

The decoding technique using the PROM has several advantages: (1) it is programmable, so the map can be altered or expanded by just reprogramming the PROM, and (2) the entire memory space can be utilized without the foldback memory addresses.

## 16.5

### DIRECT MEMORY ACCESS (DMA) AND THE Z80 DMA CONTROLLER

Direct Memory Access is a commonly used I/O technique for high-speed data transfer (for example, data transfer between system memory and a floppy disk). In polling and interrupt I/O, data transfer is relatively slow because each byte must be read and then written to its

**TABLE 16.1**  
**PROM Decoding**

destination; thus, two instructions per byte are required. In DMA, the MPU releases the control of the buses to a device called a DMA controller. The controller manages data transfer between memory and a peripheral under its control, thus bypassing the MPU. Conceptually, this is an important I/O technique that requires two signals available on the Z80—BUSRQ (Bus Request) and BUSAK (Bus Acknowledge).

- ❑ **BUSRQ**—This is an active low input signal to the Z80 from another device requesting the use of the address and the data buses, and control signals. After receiving the Bus Request, the MPU relinquishes the buses in the following machine cycle. All buses are tri-stated, so the Bus Acknowledge (**BUSAK**) signal is sent out. The MPU regains the control of the buses after the **BUSRQ** goes high.
  - ❑ **BUSAK**—Bus Acknowledge. This is an active low output signal indicating that the MPU has completed its current machine cycle and has relinquished control of the buses.

A DMA controller plays two roles in this type of data transfer: one as a peripheral to the Z80 and the other as a data transfer processor. The DMA controller uses the signals just discussed as if it were a peripheral requesting control of the buses from the MPU. The MPU communicates with the controller by using the Chip Select line, buses, and control signals. However, once the controller has gained control, it plays the role of a special-purpose processor for data transfer; it uses the Z80 buses and the control signals to transfer data directly between memory and an I/O device. To perform this function the DMA controller must have the following:

1. Data bus
2. Address bus
3. Chip Enable and Read/Write control signals
4. Signals to communicate when it functions as a peripheral and when as a processor.

Typically, the microprocessor accesses the DMA controller as an I/O device and writes the necessary initialization instructions in the DMA control registers. These instructions include the mode of data transfer (discussed later), the source and the destination addresses, and the number of bytes to be transferred. Figure 16.17 shows a block diagram of the DMA data transfer. When a peripheral is ready for data transfer, it sends the Ready signal to the DMA controller, and in turn, the DMA controller sends the Bus Request (BUSREQ) signal to the MPU. The MPU completes the execution of the present machine cycle, acknowledges the request by sending the Bus Acknowledge (BUSAK) signal, and releases the control of the buses to the DMA controller.

The DMA controller can transfer data either sequentially or simultaneously. In the **sequential transfer**, the Read cycle is followed by the Write cycle. In the **simultaneous transfer**, each byte is read from the source and written into the destination simultaneously; the Read and Write cycles are active at the same time. Figure 16.18 shows these two types

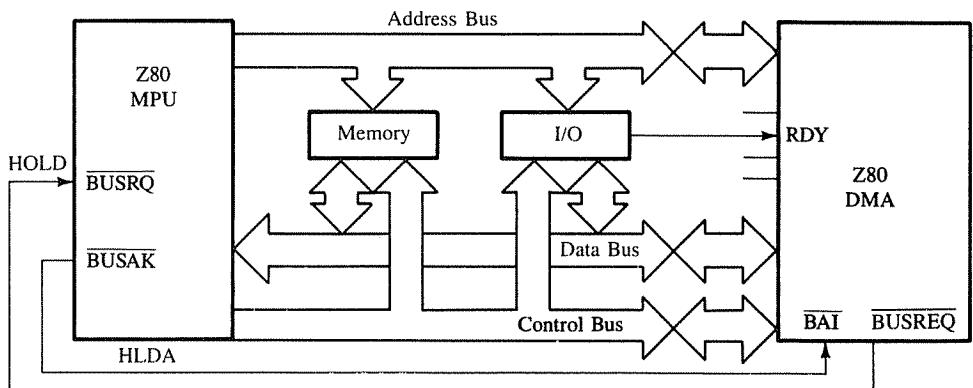


FIGURE 16.17

Block Diagram: DMA Data Transfer

SOURCE: Reprinted by permission of Intel Corporation, Copyright 1979.

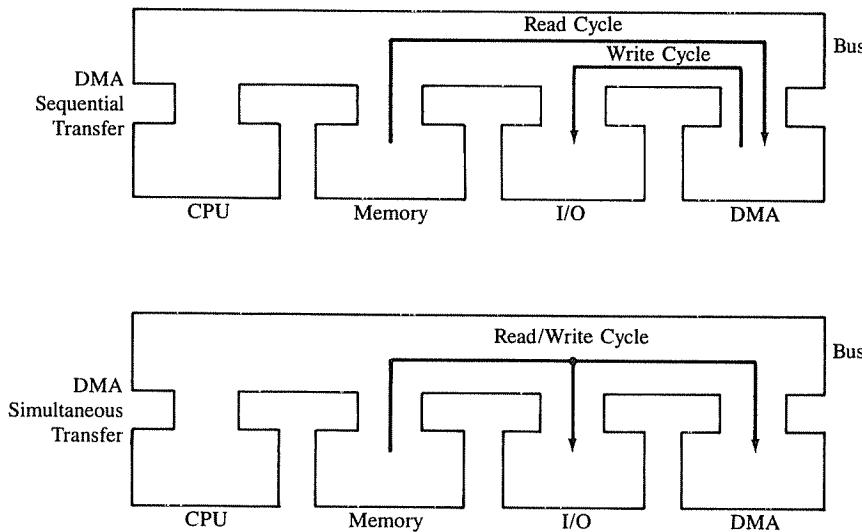


FIGURE 16.18

DMA Data Transfer: Sequential and Simultaneous

SOURCE: Courtesy of Zilog, Inc.

of data transfer. In addition, for each type of data transfer, the DMA controller can be set up in one of three modes: *byte*, *burst*, and *continuous (block)*. In the byte mode, the controller transfers one byte and releases the bus control back to the MPU. In the burst mode, the data transfer continues until the Ready goes inactive, and then the bus control is released. In the continuous transfer, the controller does not release control of the buses until the entire block of data transfer is completed.

The DMA data transfer has the highest priority in the system; no interrupt, not even the NMI (non-maskable interrupt), can be acknowledged during the DMA data transfer. One of the major disadvantages of the DMA process is that the refreshing of the dynamic memory is suspended during the data transfer; this suspension can be detrimental to the system, especially in the block mode. The process of DMA data transfer and the interfacing are discussed in the following sections in the context of the Z80 DMA controller.

### 16.51 The Z80 DMA Controller

The Z80 DMA controller is a programmable device, capable of transferring a block of 64K bytes or searching for a particular 8-bit maskable byte. It can also combine data transfers with simultaneous search. The ability to search for a byte is generally not found in other controllers. Figure 16.19 shows the logic pinout of the device; it is similar to a processor, and designed to be compatible with the Z80 control signals. It includes 16 address lines, eight data lines, control signals compatible with the Z80, interrupt control, and the signals to communicate with a peripheral and the MPU for the DMA data transfer. Some of these

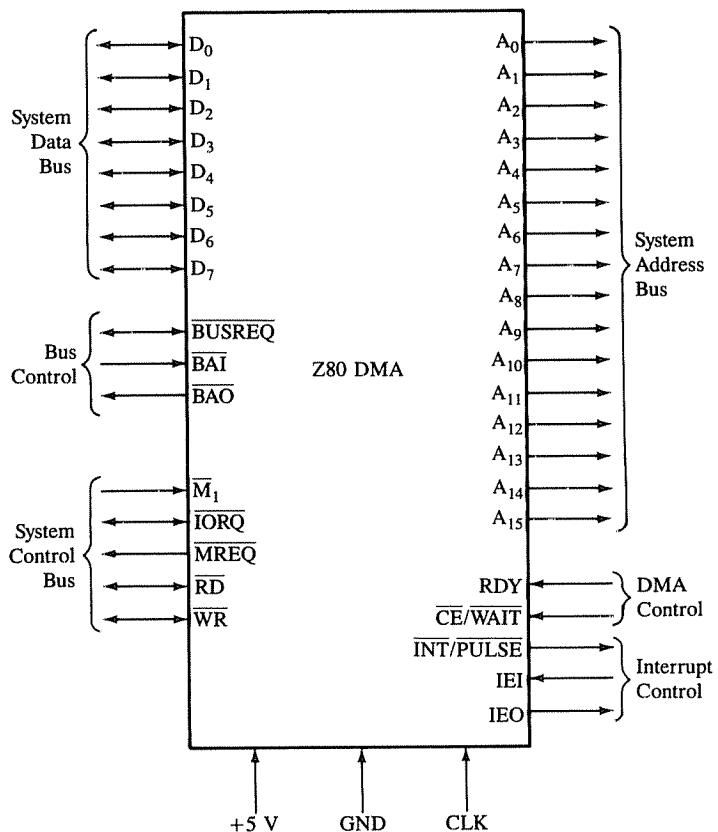


FIGURE 16.19

Z80 DMA: Logic Pinout

SOURCE: Courtesy of Zilog, Inc.

signals are bidirectional, and they perform different functions depending upon whether the DMA is in the peripheral mode or in the processor (master) mode. The signals that perform some special functions or are unique to the DMA controller are described as follows.

- **CE/WAIT**—(Chip Enable and Wait): Generally, this is used as a Chip Enable signal through which the MPU can access the DMA as a peripheral and write instructions in the control register or read status registers. However, it can be also used as a Wait line. After the DMA receives the Bus Acknowledge ( $\overline{BUSAk}$ ) signal and when it is in the processor mode, this line can be used as a Wait line by memory or I/O to slow down the speed of the DMA to match with memory or I/O.
- **BAI**—(Bus Acknowledge In): This is an active low input signal and is generally connected to the  $\overline{BUSAk}$  signal of the Z80. When this signal is active, it indicates that the

buses have been released by the MPU. In a multiple-DMA system, this signal is connected to the BAO (Bus Acknowledge Out) signal of the next higher priority DMA to form a daisy chain; the BAI of the highest priority is connected to the BUSAK of the Z80.

- BAO—Bus Acknowledge Out: This is an active low signal and is used in a multiple-DMA system to form the daisy chain priority.
- RDY—(Ready): This is an input signal and can be programmed as active low or high. When a peripheral is ready for data transfer, this signal goes active. When the DMA is in the processor mode, this signal controls the activities of the DMA.
- IORQ, RD, and WR—These are three bidirectional control signals. When the Z80 communicates with the DMA as a peripheral, these signals are input to the DMA. When the DMA functions as a processor, these signals are output signals and used to communicate with other memory or I/Os.
- MREQ—This is an output signal used as a control signal to communicate with memory.
- BUSREQ—(Bus Request): This is a bidirectional active low signal. As an output, it is connected to the BUSRQ of the Z80; it requests the use of the buses to the MPU. In a multiple-DMA system, it senses whether any other DMA is using the buses, and it refrains from requesting the buses until the other DMA operation is completed.

### 16.52 Interfacing the Z80 DMA Controller

The interfacing of the Z80 DMA is similar to that of any other peripheral. Figure 16.20 shows a schematic of interfacing the Z80 DMA with the Z80. The output line  $Y_1$  of the decoder is connected to the  $\overline{CE}/\overline{WAIT}$  line of the DMA; thus, the DMA is assigned the port address  $F9_H$ . The Bus Request and Bus Acknowledge signals of the Z80 are connected to the respective signals (BUSREQ and BAI) of the DMA, and the RDY signal of the DMA

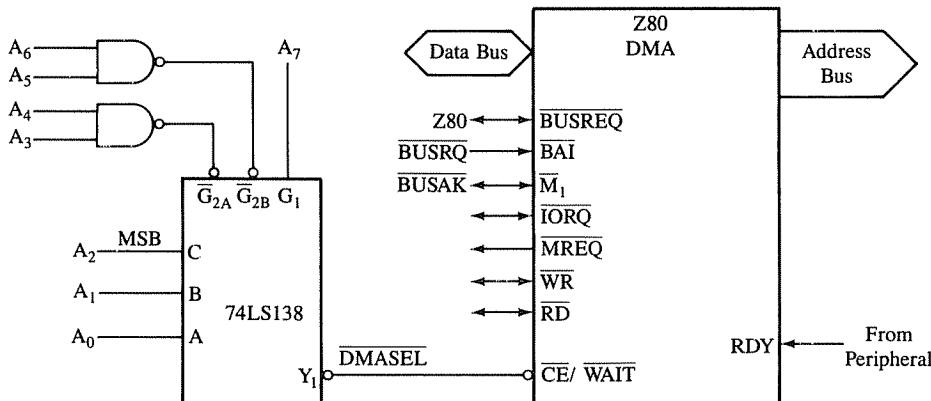


FIGURE 16.20  
Interfacing DMA

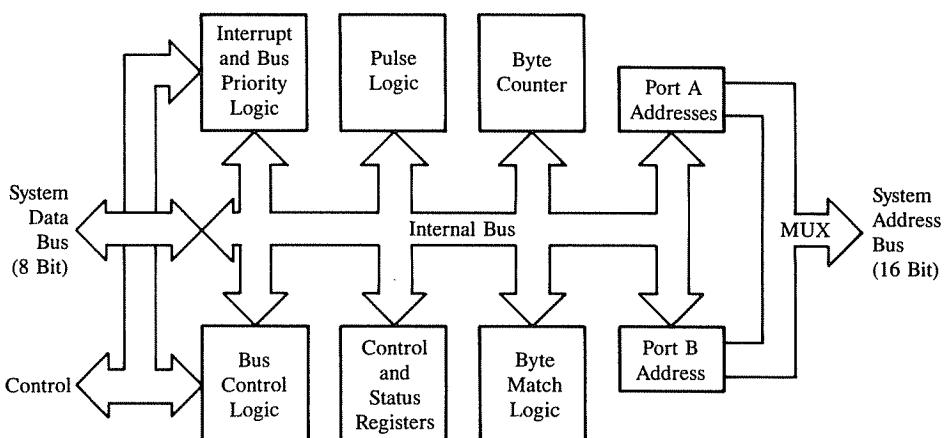
is connected to a peripheral. The remaining control signals of the Z80 ( $\overline{M}_1$ ,  $\overline{RD}$ ,  $\overline{WR}$ ,  $\overline{IORQ}$ , and  $\overline{MREQ}$ ) are connected to the respective control signals of the DMA; this is similar to interfacing any other programmable I/O. However, three control signals ( $\overline{RD}$ ,  $\overline{WR}$ ,  $\overline{IORQ}$ ) of the DMA are bidirectional, and the  $MREQ$  is an output signal. When the DMA is in the peripheral mode, the Z80 communicates with the DMA using the  $\overline{RD}$ ,  $\overline{WR}$ , and  $\overline{IORQ}$ . When the DMA is in the processor mode, the DMA can communicate with the system memory or I/O (such as a floppy disk controller) using all four control signals ( $IORQ$ ,  $MREQ$ ,  $RD$ , and  $WR$ ).

### 16.53 Programming the DMA Controller

The Z80 DMA is a versatile device and offers many options; thus, it requires a series of instructions to program the device for given specifications. However, we will discuss only the important features of this device. Figure 16.21 shows the internal structure of the DMA, which includes two port addresses, one byte counter, control and status registers, and control logic.

To program the DMA, we must write starting addresses of source and destination in Port A and Port B; either port can be used for source or destination. The addresses can be for memory or I/O; they can also be either fixed or variable, and if they are variable, they can be incremented or decremented. We must also specify the block length (the number of bytes to be transferred), type of operation (transfer, search, or search/transfer), and the mode of data transfer (byte, burst, or continuous).

The Z80 DMA has 21 Write registers to write control words and seven Read registers to read the status of an operation. The Write registers are organized in seven base register groups, and each group includes several control registers. The steps in writing



**FIGURE 16.21**  
Z80 DMA Block Diagram  
SOURCE: Courtesy of Zilog, Inc.

control words into these registers are similar to those of writing into Z80 SIO registers; they involve writing to a base register and using the base register as a pointer to other registers. The Read registers provide status information of the DMA, including the number in the byte counter and addresses in Ports A and B. For the programming details of these registers, refer to the Technical Manual of the Z80 DMA included in the References.

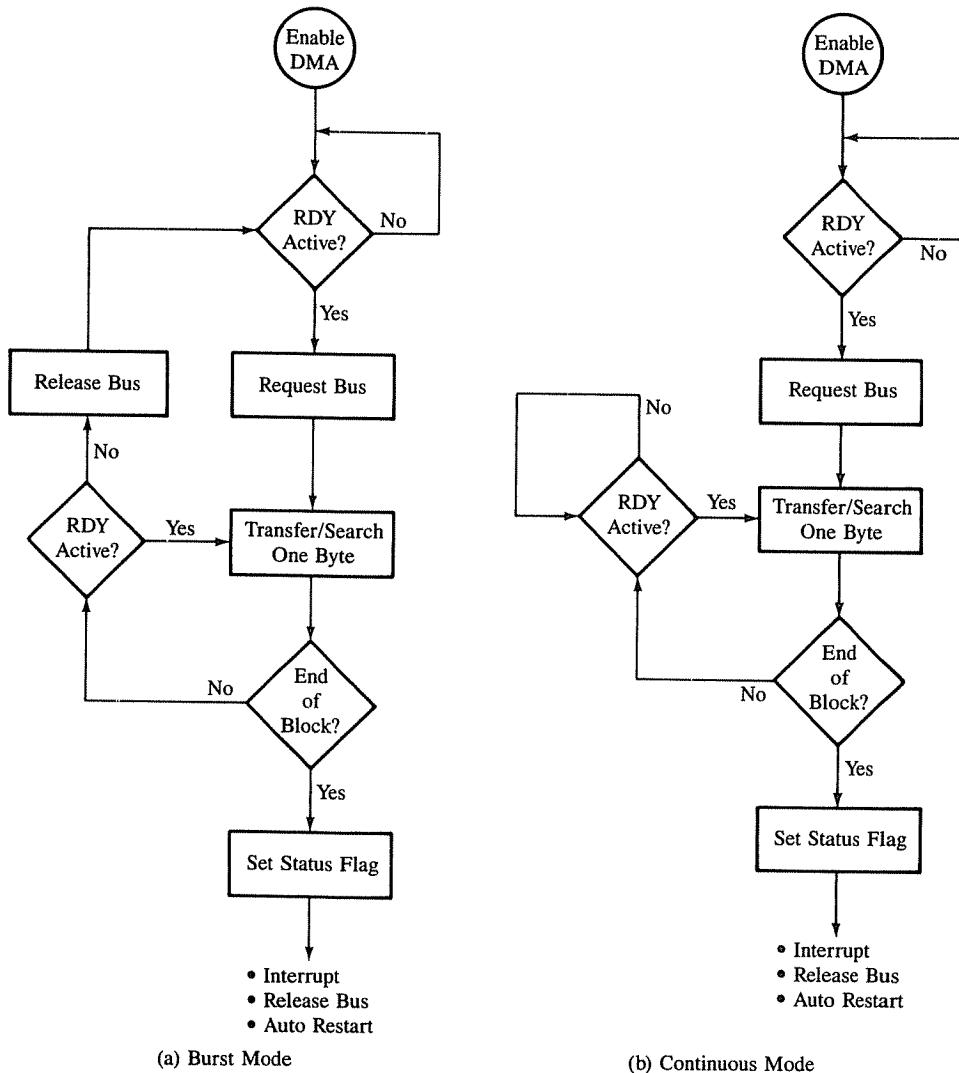


FIGURE 16.22

Flowchart: DMA Data Transfer in (a) Burst Mode, and (b) Continuous Mode  
SOURCE: Courtesy of Zilog, Inc.

### 16.54 Process of DMA Data Transfer

In a Z80 system, the DMA is normally connected as a peripheral, and control bytes must be written at the power-up initialization. Then the DMA begins to monitor the RDY line from the peripheral (such as a floppy disk controller, a printer, or a SIO). When the RDY line goes active, indicating that the peripheral is ready for data transfer, the DMA asserts the  $\overline{\text{BUSREQ}}$ . The Z80 MPU completes the present machine cycle, acknowledges the request by asserting the  $\overline{\text{BUSAK}}$ , which is connected to the  $\overline{\text{BAI}}$  of the DMA, and releases the control of the buses. When the  $\overline{\text{BAI}}$  signal goes low, the DMA assumes the role of the processor and begins the data transfer according to the specified mode. This can also be accomplished by generating an interrupt when the RDY goes active and then enabling the DMA data transfer.

If the DMA is set up in the byte mode, it transfers one byte and releases control of the buses to the MPU. The buses are requested again for each succeeding byte transfer. If the DMA is programmed in the burst mode, the DMA transfers a byte and checks the block length. If it is not the end of the block and the RDY is still active, the data transfer continues. When the RDY line goes inactive, the  $\overline{\text{BUSREQ}}$  signal goes high, and the DMA releases bus control to the MPU.

If the DMA is programmed in the continuous (block) mode, the data transfer continues until the end of the block. During the transfer, if the RDY goes inactive, the DMA does not release the control of the buses; it waits until the RDY goes active again and continues the data transfer until the end of the block. Figure 16.22 shows two flowcharts, one for the burst mode and the other for the continuous (block) mode.

---

## SUMMARY

---

In this chapter, we have discussed applications of two external request signals: Wait and Bus Request. The Wait signal is used to provide additional time to a slow peripheral so that data transfer is properly synchronized between the Z80 microprocessor and slow peripherals. On the other hand, the Bus Request is used to implement high-speed data transfer without the intervention of the microprocessor. In addition, interfacing of dynamic memory, address decoding using a PROM, and memory design were illustrated. The important concepts discussed in this chapter can be summarized as follows.

- The Z80 includes the “WAIT” signal, which can be used as an input from slow peripherals to add clock cycles in a given operation. The Z80 samples the WAIT line during  $T_2$  of each machine cycle, and if it is asserted low, the Z80 adds an additional clock period, thus providing extra time to slow peripherals.
- If memory access time is too slow in comparison with the execution speed of the microprocessor, the WAIT line can be used to synchronize the data transfer between the memory and the Z80.

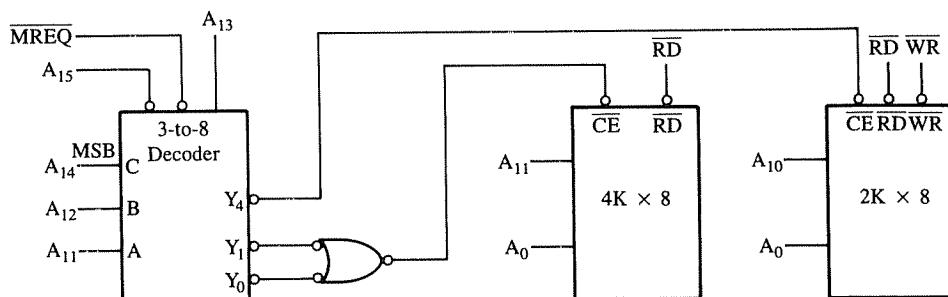
- Dynamic memory stores a bit as a capacitive charge, which has a tendency to leak; therefore, all cells must be refreshed every two milliseconds.
- In a dynamic memory chip, the memory cells are organized in a square matrix format, and the row and the column address lines are multiplexed. Therefore, the row address must be placed on the memory address lines first, followed by the column address with an appropriate delay.
- Dynamic memory includes two signals,  $\overline{\text{RAS}}$  (Row Address Strobe) and  $\overline{\text{CAS}}$  (Column Address Strobe), which are used to latch the row address and the column address.
- The Z80 architecture includes a refresh register R that can be used as a 7-bit counter to refresh 128 rows every two milliseconds.
- In memory design, two techniques used for address decoding are called incomplete decoding and absolute decoding. The incomplete decoding technique is generally less expensive, but generates foldback memory addresses.
- Memory address decoding can be achieved by using various devices such as a decoder, a PROM programmer, or logic gates.
- Direct memory access (DMA) is a commonly used I/O technique for high-speed data transfer.
- The Z80 includes two signals— $\overline{\text{BUSRQ}}$  (Bus Request) and  $\overline{\text{BUSAK}}$  (Bus Acknowledge)—which are used in the DMA data transfer.
- When the DMA controller sends the  $\overline{\text{BUSREQ}}$  signal, the MPU acknowledges the request by asserting the  $\overline{\text{BUSAK}}$  signal at the end of the machine cycle being executed and releases bus control to the DMA. The DMA uses the buses to transfer data, and then releases bus control back to the MPU.
- The DMA has three modes of data transfer: byte, burst, and block (continuous). In the byte mode, the DMA transfers one byte and releases the control back to the MPU. In the burst mode, the DMA continues to transfer data until the Ready signal of the DMA is inactive. In the block (continuous) mode, the DMA does not release the control of the buses until the entire block of the data transfer is complete.

## ASSIGNMENTS

---

1. Define the memory access time.
2. Explain the condition that must be satisfied to add a Wait state.
3. Explain the need to refresh the dynamic memory cells.
4. Explain why the refresh circuitry is unnecessary in static R/W memory.
5. What are the advantages of using a PROM as a decoder?
6. Explain the DMA technique and the functions of the Z80  $\overline{\text{BUSRQ}}$  and  $\overline{\text{BUSAK}}$  signals.
7. List the steps involved in the DMA controller's transfer of data using the  $\overline{\text{BUSRQ}}$  and  $\overline{\text{BUSAK}}$  signals.
8. In the DMA, what is the difference between the sequential and the simultaneous data transfer?

9. Explain the difference between the three modes of DMA data transfer: byte, burst, and block.
10. Calculate the time available for the Z80 to read data after an address is placed on the address bus in a 6 MHz system if the address delay is 90 ns and the data set-up time is 30 ns.
11. In 10, can the memory chip with the access time 300 ns be used without a Wait state?
12. The Mostek MK4164 is a 64K memory chip. How many multiplexed address lines are necessary for this chip?
13. Figure 16.4(d) shows the logic pinout of the 16K memory chip, in which two pins are without any connections. If these two pins are used as address lines to design a new chip, what will be the memory size of this chip?
14. How does the MPU read data from the 2118 memory if the memory chip does not have the RD signal?
15. Specify the signals necessary to generate the RAS signal.
16. Explain how the CAS signal is generated in Figure 16.11.
17. In Figure 16.11, when the RFSH signal clears the flip-flop 7474, specify the logic levels of the signal STROBE (S) of the 74S157, the CAS, and the contents of the memory address lines.
18. Specify the memory map of the 4K EPROM and 2K R/W memory in Figure 16.23.
19. In 18, is there foldback memory because memory chips of two different sizes are decoded by the same decoder? Explain your answer.
20. In Figure 16.16, change the contents of the PROM to obtain the memory map as follows: 4K EPROM from 0000 to 1FFF<sub>H</sub>, 2K R/WM from 2000<sub>H</sub> to 27FF<sub>H</sub>, and ROMs from 8000<sub>H</sub>.
21. In Figure 16.16, how many address lines of the PROM should be used if the R/WM is of size 4K?
22. Explain the detrimental effect of the continuous (block) mode in DMA if the system includes dynamic memory.

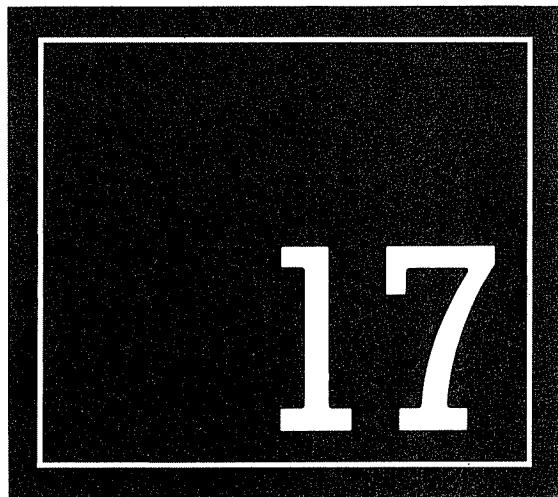


**FIGURE 16.23**  
Schematic for Assignment 18

# Designing Microprocessor- Based Products

In Chapter 1, we began with an overview of microprocessor-based products and microcomputer systems. In subsequent chapters, we examined the architecture of the Z80 microprocessor and interfacing of memory and I/Os. An overview of the Z80 instruction set was given in Chapter 6, and Chapters 7 through 11 were devoted to the discussion of various programming techniques, applications of the Z80 instruction set, and familiarization with operating systems. Similarly, we examined processes of data transfer such as interrupts, serial I/O, and DMA using programmable devices. This chapter is concerned with integrating or synthesizing the concepts of the microprocessor architecture, software, and interfacing discussed previously, by designing a microprocessor system.

Designing a single-board microcomputer is the best possible choice, since it can incorporate all the important concepts related to the microprocessor-based design. Furthermore, it allows expansion to include various types of interfacing. This chapter deals primarily with designing such a single-board computer. The chapter also includes troubleshooting techniques using an in-circuit emulator, a logic analyzer, and a signature analyzer.



## OBJECTIVES

- Design modules (sub-systems) of a single-board microcomputer based on the Z80 microprocessor and Z80 family of programmable interfacing devices.
- Illustrate the interfacing of scanned display, and list the advantages.
- Illustrate the interfacing of a matrix keyboard using software.
- Illustrate the interfacing of a matrix keyboard using a keyboard encoder.
- Draw flowcharts to illustrate the software design of a Key Monitor program and related sub-modules.
- List the primary features of the in-circuit emulator and explain its applications in troubleshooting microprocessor-based systems.
- Explain the functions of a logic analyzer and a signature analyzer as troubleshooting instruments.

## 17.1 PROJECT STATEMENT: DESIGNING A MICROCOMPUTER SYSTEM

---

Design a single-board microcomputer to meet the following specifications:

- Input: Hex keyboard with minimum of 20 keys.
- Output: Six seven-segment LEDs to display memory address and data.  
: Two seven-segment LEDs to display results.
- Memory: Minimum 2K of EPROM—2716 ( $2048 \times 8$ ) or 2732 ( $4096 \times 8$ ).  
: 2K of R/W static memory—6116 ( $2048 \times 8$ ) or equivalent.
- Microprocessor: Z80.
- System Frequency: 2 MHz.
- Suggested Interfacing Devices: Z80 PIOs, bus drivers, 3-to-8 decoders, key encoder, segment and digit drivers, and Hex decoder/drivers.

The system should allow a user to enter and execute programs, and the buses should have enough driving capacity to interface with additional peripherals. While machine codes are being entered, the memory address and data should be displayed by seven-segment LEDs. A two-digit seven-segment LED port should be available as an output port to display the results when programs are executed.

### 17.11 Project Analysis

In analyzing the specifications of a microprocessor-based product it is essential to consider both hardware and software, simultaneously. Both are interrelated, and each will have an impact on the other. However, in this project we will not focus on one approach, but will explore alternatives. We will design various modules and leave the final decisions with the user.

The functions of the single-board microcomputer according to the specifications (given above) can be classified into three categories as follows:

1. Check the keyboard for data or functions.
2. Display memory address, data, and results.
3. Execute programs.

**Keyboard** The keyboard in this design is an input port with keys arranged in the matrix format. When a key is pressed, the keyboard routine should provide a binary equivalent of the key. This can be accomplished various ways: one is a software approach whereby a key closure is sensed, debounced, identified, and the key code obtained by using the software. The other is the hardware approach whereby all these key functions are performed through a programmable keyboard encoder.

The keys are divided into two groups: One group is for Hex digits from 0 to F, and the second is concerned with various functions. There are two basic approaches to entering data and specifying a function. One approach is to begin with Hex keys, identify the memory address, and then specify a function such as Examine Memory or Execute. In the second approach, a function is specified first and then Hex keys are entered.

**Display** This project has two types of display: the system display and the user display. The system display consists of four seven-segment LEDs for memory address and two seven-segment LEDs for data. The user display consists of two seven-segment LEDs for results. We can explore both hardware and software approaches to designing output ports for these displays and suggest a way of combining the user display and the data display.

**Execute** This is the simplest function among all three and can be performed with one instruction: JP (HL). When the user wants to execute a program, he or she provides the starting memory address where the program is stored and presses the Execute key. Assuming the memory address is stored in the HL register, the instruction JP (HL) simply loads the program counter with the specified memory address, and the program control is transferred from the monitor program to the user's program.

Figure 17.1 shows the block diagram of a single-board microcomputer, and we can divide the project design into the following sections:

1. Z80 MPU design.
2. Memory design.
3. Display design.
4. Keyboard interfacing.
5. System software.

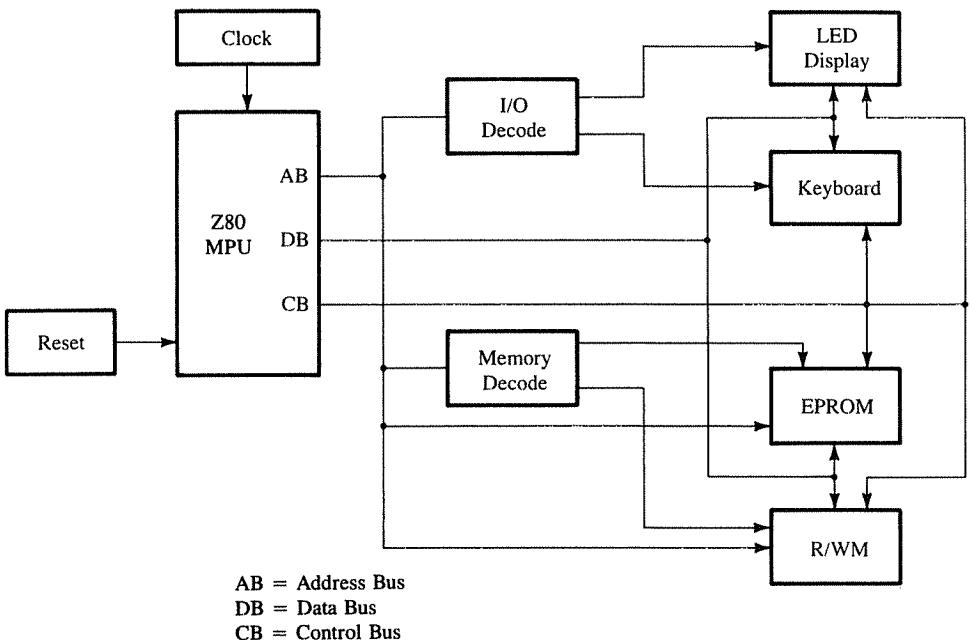
---

## Z80 MPU DESIGN

17.2

---

This single-board microcomputer is designed around the Z80 microprocessor, and the MPU should provide necessary buses with appropriate driving capacity. The MPU design can be divided into the following segments:

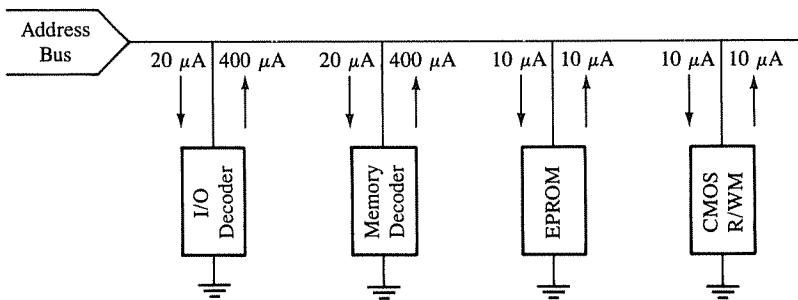


**FIGURE 17.1**  
Block Diagram of a Single-Board Microcomputer

1. Address bus.
2. Data bus.
3. Control signals.
4. Frequency and power requirements.
5. Externally triggered signals (Reset, Interrupts, etc.).

### 17.21 Address Bus

The Z80 has 16 address lines  $A_{15}-A_0$ ; this is a unidirectional bus with driving current capacity of  $I_{OH} = 250 \mu A$  and sinking capacity  $I_{OL}$  of 1.8 mA. At this point, we do not know the total load on the address bus, but by examining the block diagram, we can make some reasonable estimates of the load on the address bus. Figure 17.1 shows that the address bus will drive two decode circuits (I/O and memory decoders) and two memory chips (CMOS 6116 and EPROM 2716). We can calculate the bus loading as follows (see Figure 17.2):



**FIGURE 17.2**  
Loading on the Address Bus

#### High-level input currents $I_{IH}$

$$\begin{array}{l} \text{Two Decoders} = 20\mu\text{A} \times 2 = 40\mu\text{A} \\ \text{R/W Memory} = 10\mu\text{A} \\ \text{2732 EPROM} = 10\mu\text{A} \\ \hline 60\mu\text{A} \end{array}$$

#### Low-level input currents $I_{IL}$

$$\begin{array}{l} 400\mu\text{A} \times 2 = 800\mu\text{A} \\ = 10\mu\text{A} \\ = 10\mu\text{A} \\ \hline 820\mu\text{A} \end{array}$$

By examining these load currents, we can conclude that the bus driver is unnecessary for the address bus; we can even add a few decode circuits or gates. However, this single-board microcomputer is expected to be used for general-purpose interfacing; therefore, as a precaution we will use the 74LS244 as a bus driver to increase the driving capacity. The 74LS244 is an octal buffer/driver, capable of sourcing 15 mA and sinking 24 mA of current. Figure 17.3 shows two octal buffers for 16 address lines; the Enable lines of these buffers are active low, and they are permanently enabled. Thus, the Z80 address bus can drive additional devices (decoders, gates, etc.) without excessive loading.

## 17.22 Data Bus

The Z80 data bus has eight bidirectional lines with driving capacity similar to that of the address bus. Because the data bus is bidirectional, the loading on the bus varies considerably. When the Z80 is reading from memory, the memory chip that is enabled becomes the driving source and the microprocessor becomes the load, and when the Z80 is writing to an output port, the microprocessor is the source and the latches of the output port constitute the load. An octal latch, such as the 74LS363, requires a 400 μA input current at the low level logic; on the other hand, the 7475 requires 3 to 6 mA. Therefore, as a precaution, we will use a bidirectional buffer as a data bus driver.

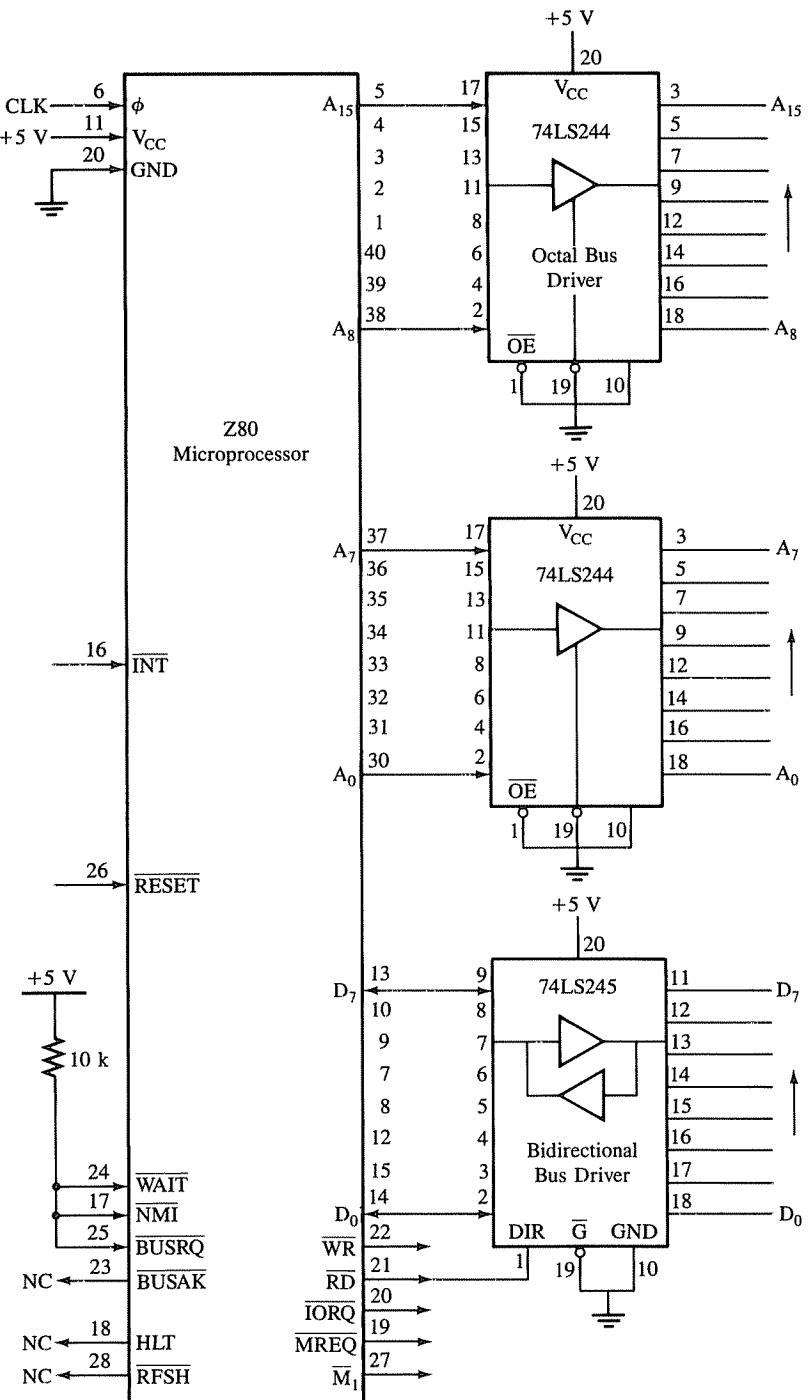


FIGURE 17.3  
Z80 MPU with Bus Drivers

Figure 17.3 shows the 74LS245 as an 8-bit bidirectional bus driver to increase the driving capacity of the data bus. The 74LS245 can sink 24 mA and source 15 mA of current. The 74LS245 has eight bidirectional data lines; the direction of the data flow is determined by the direction control line (DIR). Figure 17.3 shows that the bus driver is enabled by grounding the Enable ( $\bar{G}$ ) signal. The direction of the data flow is determined by connecting the RD signal from the Z80 to the DIR signal. When the Z80 is writing to peripherals, the RD is high and data flow from the Z80 to peripherals. When it is reading from peripherals, the RD is low and data flow toward the microprocessor.

### 17.23 Control Bus

The Z80 provides five active low signals— $\bar{M}_1$ ,  $\bar{IORQ}$ ,  $\bar{MREQ}$ ,  $\bar{RD}$ , and  $\bar{WR}$ —which can be combined to generate necessary control signals. The commonly used control signals are  $\bar{IORD}$  (I/O Read),  $\bar{IOWR}$  (I/O Write),  $\bar{MEMRD}$  (Memory Read),  $\bar{MEMWR}$  (Memory Write), and  $\bar{INTA}$  (Interrupt Acknowledge); they can be generated by the logic combinations shown in Figure 17.4(a). However, in memory interfacing the RD and WR signals are generally connected to the memory chip directly and  $\bar{MREQ}$  is combined with the address-decoding scheme. Figure 17.4(b) shows another scheme to generate the I/O control signals and the INTA signal by using a 3-to-8 decoder.

The driving capacity of these control signals is determined by the circuits used in generating them. If necessary, these control signals can be buffered by using the 74LS244 or the Hex drivers 74LS367. Generally, the circuits shown in Figure 17.4 will have sufficient drive so that buffers may not be needed.

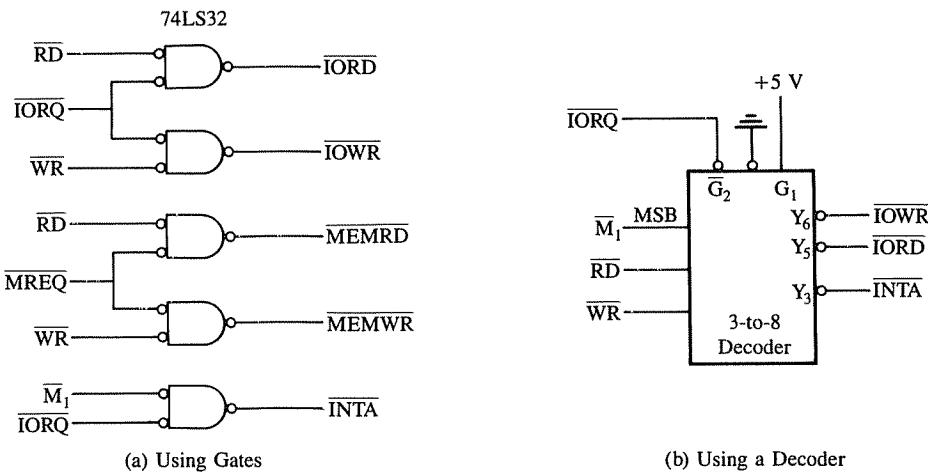


FIGURE 17.4  
Generating Control Signals

## 17.24 Frequency and Power Requirements

The clock circuitry is of critical importance in designing Z80 systems. The Z80 does not have an oscillator circuit on its chip; therefore, a separate oscillator circuit needs to be built. The Z80 requires a single-phase TTL level clock with a maximum 30 nsec rise/fall time, and the voltage levels should be between ( $V_{cc} - 0.6V$ ) and 0.8 V. The Z80 microprocessor has four versions of the chip operating at different maximum frequencies: the Z80 operates at 2.5 MHz, the Z80A at 4 MHz, the Z80B at 6 MHz, and Z80H at 8 MHz clock.

Figure 17.5(a) shows a typical oscillator circuit with two inverters and RC network; the 330 ohm pull-up resistor is necessary to obtain TTL voltage level within 0.6 V of  $V_{cc}$ . However, this type of circuit is somewhat unstable because of variances in the components. If the microprocessor is not operating at the specified maximum frequency, the circuit shown in Figure 17.5(a) can function very well. Figure 17.5(b) shows a circuit that is generally used in industrial products. The circuit uses a crystal to stabilize the frequency. The output of the oscillator circuit is fed to a flip-flop that divides the frequency in half; the flip-flop provides a 50 percent duty cycle for the clock. In addition to these circuits, several manufacturers (for example, the Motorola K1160 series) offer oscillator/driver circuits on a chip.

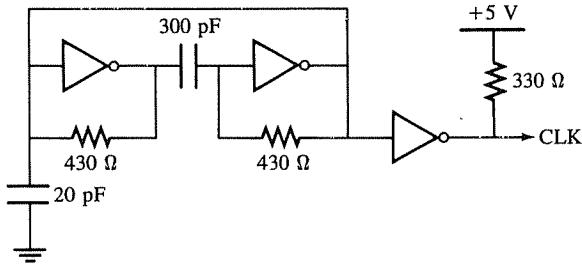
The Z80 and other components used in this system require one power supply with +5 V. The current requirement of the power supply is determined primarily by the display load and the peripherals of the system; the MPU and memory components of the system require less than 400 mA.

## 17.25 External Trigger Signals

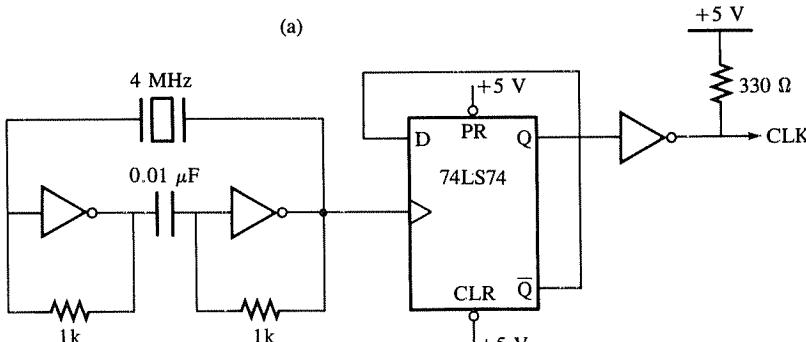
As discussed in Chapter 3, the Z80 has provision for five external input signals: Reset (RESET), Interrupt (INT), Nonmaskable Interrupt (NMI), Wait (WAIT), and Bus Request (USRQ). Of these five input signals, the  $\overline{\text{RESET}}$  and the  $\overline{\text{INT}}$  are used in this system, and the others are disabled by connecting them to +5 V (see Figure 17.3).

**Reset Circuit** The  $\overline{\text{RESET}}$  signal in the Z80 is active low; when this signal goes low, the system is reset. The reset forces the program counter to zero, disables the interrupt flip-flop, clears registers I and R, and sets the Z80 in the interrupt Mode 0.

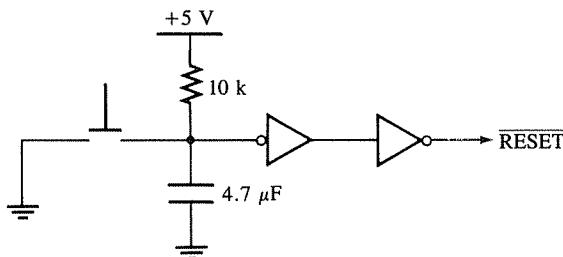
The reset circuit shown in Figure 17.5(c) is an RC network with a time constant around 50 ms. When the reset key is pushed, the  $\overline{\text{RESET}}$  goes low and slowly rises to +5 V, providing sufficient time for the MPU to reset the system; two inverters provide a sharp pulse. Some systems include a circuit called *power-on reset*, as shown in Figure 17.5(d). As power is turned on, the voltage across the capacitor does not change instantaneously; therefore, the voltage at the junction of the resistor and the capacitor goes low and slowly rises to +5 V. This pulse can be used through two inverters as in Figure 17.5(c) to reset the system.



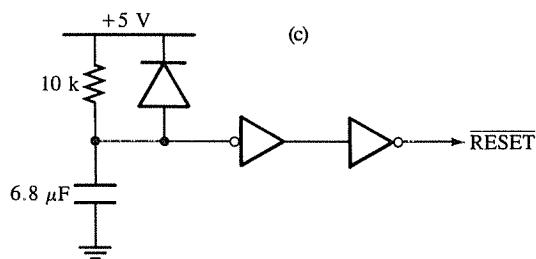
(a)



(b)



(c)



(d)

**FIGURE 17.5**

- (a) RC Network Oscillator Circuit  
 (b) Oscillator with Crystal  
 (c) Manual Reset Circuit  
 (d) Power-on Reset Circuit

**Interrupts** The Z80 has two interrupt signals,  $\overline{\text{INT}}$  and  $\overline{\text{NMI}}$ , both active low. In this system, we will use the  $\overline{\text{INT}}$  signal and tie the  $\overline{\text{NMI}}$  signal high; the floating interrupt pin can cause malfunction in the system.

**Bus Request** This is an active low signal, generally used in the DMA data transfer. We will not be using this signal in our system; this signal also must be connected to + 5V.

**Wait** If the  $\overline{\text{WAIT}}$  signal goes low during  $T_2$  of any machine cycle, the MPU enters the Wait state for an integral number of clock cycles until  $\overline{\text{WAIT}}$  goes high, and then the MPU completes the Read or Write cycle. This signal is used primarily to synchronize slow peripherals with the MPU. If we were to use the Z80A and increase the system frequency to 4 MHz, the  $\overline{\text{WAIT}}$  signal might be necessary for memory devices used in this project (see Example 16.1 for Wait calculations). In this project, to prevent the MPU from entering the Wait state, this pin is tied high.

## 17.3 MEMORY DESIGN

---

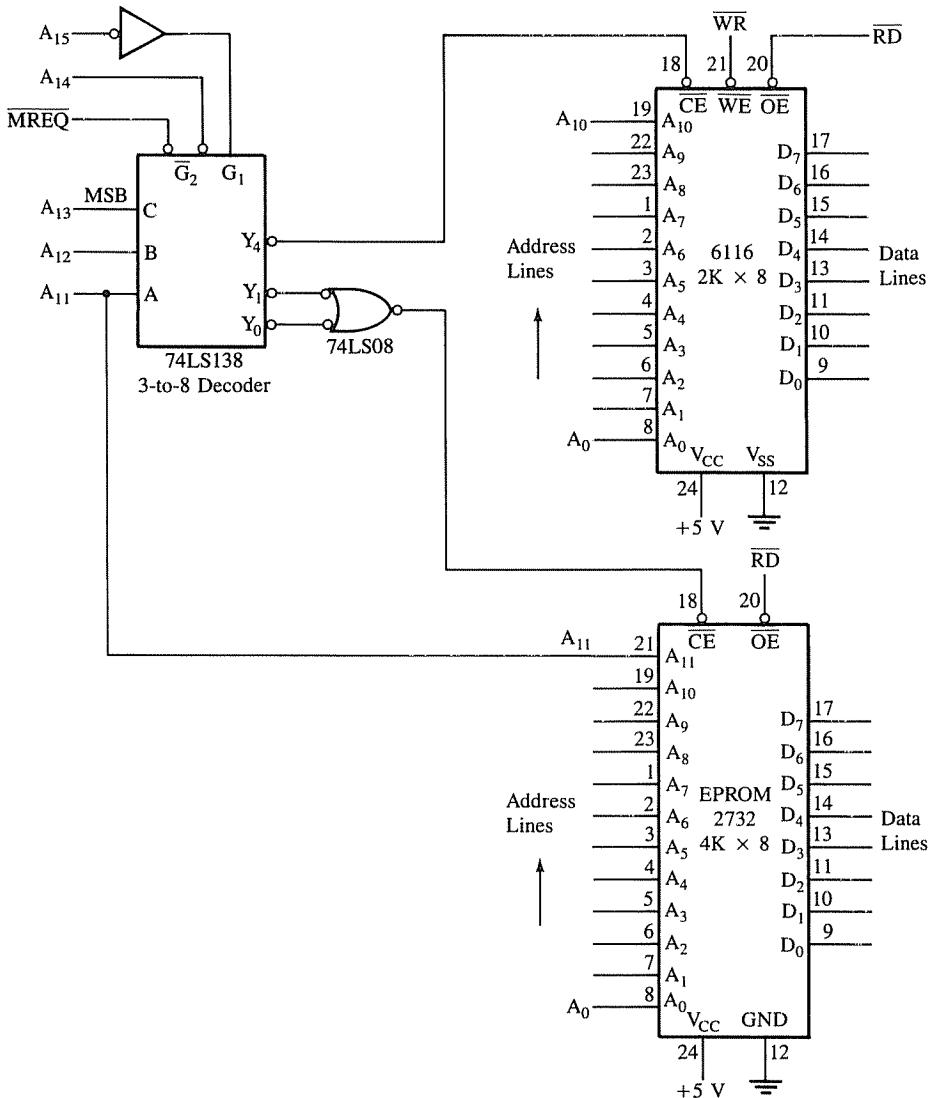
This single-board microcomputer includes two types of memory: EPROM and R/W memory. In memory design, we should be concerned about the *size* of the memory chips required and their memory maps, future *expandability*, and *access time*.

The first consideration of the memory design is the memory size and the memory map. For our design, the monitor program can be very easily stored in 2K memory, such as the 2716 EPROM. However, the price difference between the easily available 2732 EPROM (4K) and the 2716 EPROM is negligible, and both of them are pin-compatible. Therefore, we will use the 2732 EPROM ( $4\text{K} \times 8$ ) in our design. The memory map of this EPROM should begin at memory address  $0000_H$  because the program counter is cleared to that address whenever the system is reset. The memory map of EPROM with 4K bytes of memory should be placed in the range from  $0000_H$  to  $0FFF_H$  as shown in Figure 17.6. However, there are no such restrictions for the memory map of R/W memory; it can be mapped anywhere so long as it does not overlap with the map of the EPROM. The other consideration is an appropriate decoding technique for memory devices with different sizes; in this system, we are using 4K of EPROM and 2K of R/W memory. If we use the same decoding network for both devices, such as a 3-to-8 decoder, the 2K R/W memory will be left with one “don’t care” address line; thus, it will have foldback memory addresses. However, in a small system, the foldback memory is not a serious concern. Figure 17.6 shows how to avoid the foldback memory by using a negative NOR logic gate and a 3-to-8 decoder (explained in section 17.32).

The next consideration is future expandability. The 2732 requires a 24-pin socket; however, its pinout is designed in such a way that it can use a 28-pin socket and be compatible with larger memory chips. By using a 28-pin socket with additional DIP switches, the system can be expanded to accommodate the 2764 (8K), 27128 (16K), and

27256 (32K) memory chips. However, this type of expansion cannot be easily accomplished by the decoding network shown in Figure 17.6; to accommodate larger memory chips, we will have to use a PROM, which can be reprogrammed for the decoding.

The last consideration is the memory access time and whether we need any Wait states in interfacing these memories. In the last decade, the memory access time has



**FIGURE 17.6**  
Schematic: Memory Design

improved considerably; memory devices with access time around 200 ns are commonly available. If the clock frequency in our system is 2 MHz, we can conclude from the calculations shown in the last chapter that Wait states will be unnecessary in this system. However, the circuit shown in Figure 17.6 may not function properly with a 4 MHz system; the total access time including the delay in the decoder may exceed the microprocessor read time. One of the solutions to reduce the total delay is to use gates to generate control signals as shown in Figure 17.4(a).

### 17.31 EPROM Memory

Figure 17.6 shows the design of EPROM using the 2732 (4,096 × 8) and the 74LS138 (3-to-8 decoder). The twelve address lines ( $A_{11}$ – $A_0$ ) of the MPU from the bus drivers are directly connected to pins  $A_{11}$ – $A_0$  of the 2732 to decode 4,096 memory locations. The rest of the address lines ( $A_{15}$ – $A_{12}$ ) and  $A_{11}$  are decoded by the 74LS138; this provides a 2K decoding resolution for each output line of the decoder. The address line  $A_{11}$  is connected to the input of the decoder to avoid its being “don’t care” for the R/W memory (this is discussed in the next section). However, to address 4K of the EPROM memory locations, two output lines  $Y_1$  and  $Y_0$  are logically ORed and used for the Chip Enable CE line of the memory chip. To enable the memory chip, the address lines  $A_{15}$  to  $A_{12}$  should be at logic 0, and  $A_{11}$  can be at 0 or 1 because either of the output lines  $Y_0$  and  $Y_1$  can select the memory chip. By combining the address lines  $A_{10}$ – $A_0$  with the decoding lines, we can obtain the memory address of EPROM ranging from  $0000_H$  to  $07FF_H$  when  $A_{11}$  is 0 and from  $0800_H$  to  $0FFF_H$  when  $A_{11}$  is 1, as shown.

$A_{15}$	$A_{14}$	$A_{13}$	$A_{12}$	$A_{11}$	$A_{10}$	$\dots$	$A_0$	Address
0	0	0	0	0	0	-----	0	$0000_H$
					↓	-----	↓	
0	0	0	0	0	1	-----	1	$07FF_H$
						-----		
0	0	0	0	1	0	-----	0	$0800_H$
					↓	-----	↓	
0	0	0	0	1	1	-----	1	$0FFF_H$
						-----	↓	
<u>Decoder</u>		<u>Decoder</u>		Lines to decode 2048 memory bytes				
<u>Enable</u>		<u>Input</u>						
Active Low								

### 17.32 R/W Memory

The system R/W memory (2K bytes) is designed with 6116 (2,048 × 8) memory chips. Figure 17.6 shows that 11 address lines  $A_{10}$  to  $A_0$  from the MPU are connected to the 11 address pins  $A_{10}$  to  $A_0$  of the memory chips to decode the 2,048 memory locations. The Chip Select CS logic is generated from the same decoder as that for EPROM. When the  $Y_4$  line of the decoder goes low, this memory chip is enabled; thus, the memory map of this R/W memory ranges from  $2000_H$  to  $27FF_H$ , as shown.

$A_{15}$	$A_{14}$	$A_{13}$	$A_{12}$	$A_{11}$	$A_{10} - \dots - A_0$	Address
0	0	1	0	0	0 - - - - 0	$2000_H$
0	0	1	0	0	↓ - - - - ↓	$27FF_H$

## DESIGNING SCANNED DISPLAYS

**17.4**

In this system, two types of displays are specified: the system display and the user display. In Chapter 13, we discussed the interfacing of seven-segment LED displays; the approach was primarily software dependent based on the table look-up technique. In addition, we briefly introduced alternative approaches: the scan technique and hardware decoding. We will illustrate both of these approaches in this design: the scan technique for the system display and the hardware decoding for the user display. We will use Z80 PIOs as the interfacing devices.

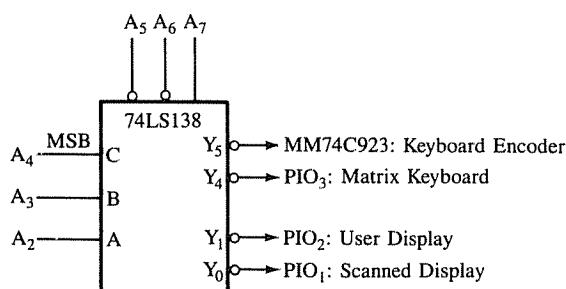
### 17.41 Basic Concepts

The basic concepts in scanned display were discussed in Section 13.58 (Figure 13.20). The display involves two output ports: One port is used to send seven-segment binary code, and the other port is used to turn seven-segment LEDs off or on in a sequence. The program repeats the sequence of code continuously; thus, the user can see a stable display.

### 17.42 Interfacing Circuit

Figure 17.7 shows the address-decoding network using the 74LS138 3-to-8 decoder. The address lines  $A_7-A_2$  are connected to the decoder and the remaining two address lines  $A_1$  and  $A_0$  will be connected directly to the Z80 PIO. The output lines of this decoder will be used for other displays and the matrix keyboard. The decode logic of the PIO is identical to that of Figure 13.6. The Z80 PIO is selected when the the output line  $Y_0$  of the decoder (Figure 17.7) is asserted. Therefore, the port addresses of the PIO (Figure 17.8) are as follows:

**FIGURE 17.7**  
I/O Decoding Circuit



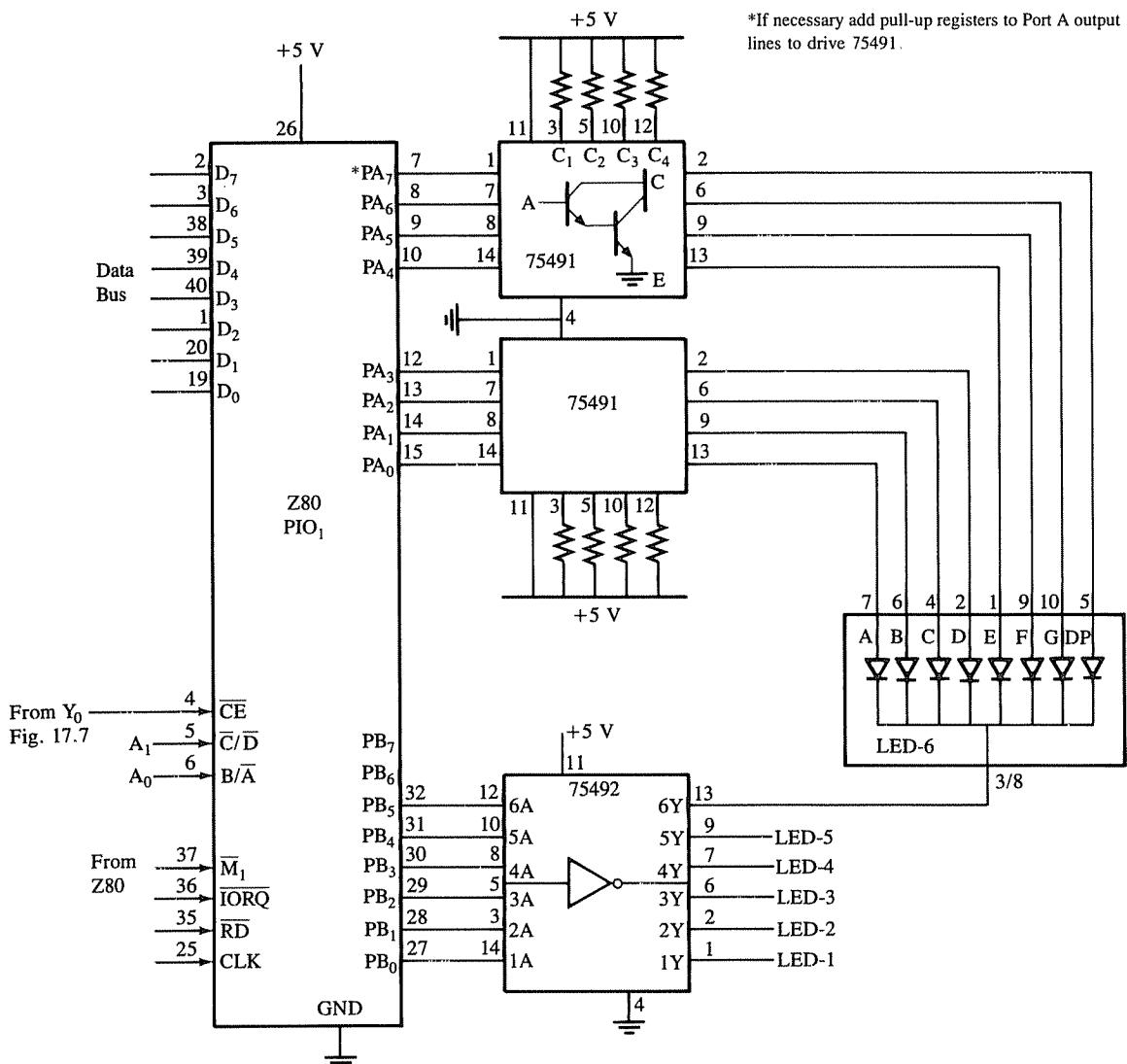


FIGURE 17.8  
Schematic: Scanned Display

A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	= 80 <sub>H</sub> Port A
1	0	0	0	0	0	0	0	= 81 <sub>H</sub> Port B
Decoder Enable Lines			Decoder Inputs			1	0	= 82 <sub>H</sub> Control Register A
						1	1	= 83 <sub>H</sub> Control Register B

Figure 17.8 shows the schematic of a scanned display; it has six common cathode seven-segment LEDs, one Z80 PIO, and two drivers. Of the six LEDs, four are used for memory address and two for data. Both ports of the PIO are set up as output ports: Port A with the address 80<sub>H</sub> for segment codes and Port B with the address 81<sub>H</sub> for digits to turn LEDs on or off. The SN 75491 and the SN 75492 are used as the segment code driver and the digit driver, respectively, to increase the current capacity in the circuit.

**SN 75491—Segment Driver** The SN 75491 is a quad device that has four Darlington pair transistors in a package; to drive eight data lines, we need two devices, as shown in Figure 17.8. The SN 75491 can source or sink 50 mA current (approximately 12.5 mA/pair). Pin A, the base of the transistor, is connected to one of the data lines of the output port and emitter E is connected to one of the LED segments.

**SN 75492—Digit Driver** The SN 75492 has six Darlington pairs in a package and can sink 250 mA of total current. Each collector (pins 6Y–1Y) is connected to the common cathode of its respective LED, and the data lines from the port are connected to the base of the transistor to turn the LEDs on or off.

To display a digit, the seven-segment code for the digit is sent to Port A, and the corresponding cathode is turned on and off in sequence; the loop is repeated continuously.

### 17.43 Program

```
;The following program initializes the Z80 PIO ports A and B as output ports
; and displays a constant message stored at memory location SYSRDY (System
; Ready). The message has six codes: uP-rdy (microprocessor ready). The
; code for the right-most letter "y" is stored at the first location SYSRDY,
; and the scanning begins at that location.
```

SEGMNT	EQU 80H	;Port address-Segment Driver
DIGIT	EQU 81H	;Port address-Digit Driver
PIO1A	EQU 82H	;Control Port A
PIO1B	EQU 83H	;Control Port B
PIO1:	LD A, 00001111B	;PIO control word 0FH for Mode 0
	OUT (PIO1A), A	;Initialize Port A
	OUT (PIO1B), A	;Initialize Port B
READY:	LD B, 00000001B	;Initialize digit code
	LD C, 06	;Initialize counter for six LEDs
	LD HL, SYSRDY	;Use HL as memory pointer for message
NEXT:	LD A, (HL)	;Get segment code
	OUT (SEGMNT), A	;Output segment code
	LD A, B	;Get digit code
	OUT (DIGIT), A	;Turn on one LED
	CALL DELAY1	;Wait 1 millisecond
	XOR A	;Code to turn off segments
	OUT (SEGMNT), A	;Clear segments

```

    RLC B           ;Shift digit code to turn on next LED
    INC HL          ;Point to next code
    DEC C           ;Next LED count
    JR Z, NEXT
    RET

SYSRDY: DB 5EH, 50H,      ;Message codes
        DB 40H, 73H, 1CH ;y d r - P u

```

### PROGRAM DESCRIPTION

This routine initializes ports A and B of PIO1 as output ports by sending the word 00001111B ( $0F_H$ ) to the control registers of Port A and Port B (see Figure 13.7 for the definition of the control word). The next instruction initializes the scan routine by placing the digit code 00000001 into register B; this code will turn on LED-1 (the first LED at the right). By rotating bit  $D_0$  (logic 1) to the left, the next LED is turned on and the LED presently being displayed is turned off; thus, only one LED is on at a time. Register C is set up as a counter to scan six LEDs, and the HL register is used as a memory pointer to point to where the message is stored.

The scanning begins by sending the first code (the last letter “y” in the message) to Port A, and LED-1 is turned on by sending the digit code. This LED is kept on for approximately 1 ms by calling the delay routine, and the entire display is turned off by clearing the segment code; this eliminates the flicker and the ghost images. The segment codes are sent in a sequence as they are stored in memory, and the corresponding LED is turned on until the counter reaches zero. To keep the display on, the routine should be called repeatedly.

**Comments** In the scanned display, the hardware is minimized. With two output ports, this scheme can scan eight LEDs. In addition, current consumption is considerably reduced. However, the major disadvantage is that the MPU is kept occupied in scanning the display continuously. To relieve the MPU from the continuous scanning task, the Intel 8279—programmable keyboard/display interface device—can be used (discussed in the Section 17.5).

### 17.44 Hardware Approach for the User Display Using a Hex Decoder/Driver

The user display requires two seven-segment LEDs; they are interfaced with the MPU using the Z80 PIO2 and 9370 Hex decoder drivers (Figure 17.9). The 9370 has four data lines as binary input and seven output lines. The binary input is decoded internally, and the corresponding seven-segment code is placed on the output lines. To display two bytes, we need one port of the PIO and two 9370s. The PIO2 is accessed by connecting the decoder output  $Y_1$  (Figure 17.7) to the  $\overline{CE}$  signal of the PIO; thus, the port addresses range from  $84_H$  to  $87_H$ .

In this design, we will use Port A and leave Port B for additional displays. Because the decoding is performed internally using hardware, the software required to use this

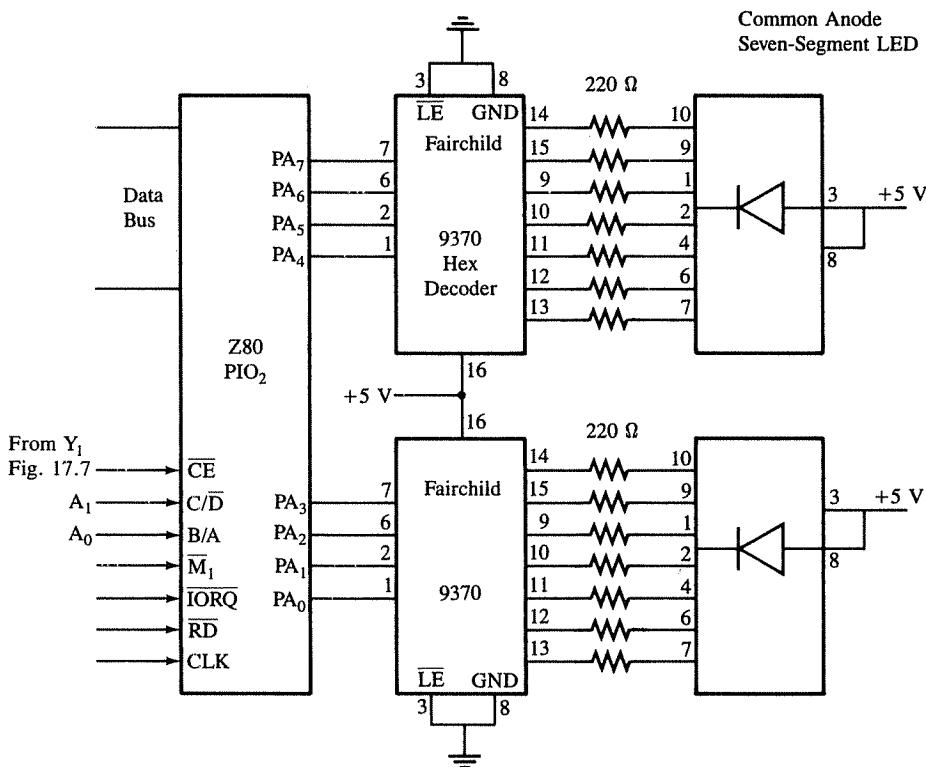
display is simple; it involves initializing Port A as an output port in Mode 0 and outputting the byte to be displayed to Port A. The instructions to display a byte at this port are as follows:

```

PIO2A EQU 86H      ;Control Port A
LEDPRT EQU 84H      ;LED port address
LD A, 00001111B    ;PIO control word for Mode 0
OUT (PIO2A), A      ;Initialize Port A of PIO2
LD A, BYTE          ;Load byte to be displayed
OUT (LEDPRT), A      ;Display BYTE at Port A

```

**Comments** In this hardware approach, the number of components increase in proportion with the numbers of LEDs; this can be quite expensive. The power consumption also increases in the same proportion.



**FIGURE 17.9**  
Schematic: Interfacing Seven-Segment LEDs Using a Hex Decoder

## 17.5

### INTERFACING A MATRIX KEYBOARD

A matrix keyboard is a commonly used input device when more than eight keys are necessary, rather than a row of keys as discussed in Chapter 13. A matrix keyboard reduces the number of connections, thus the number of interfacing devices required. For example, a keyboard with 20 keys, arranged in a  $5 \times 4$  (five rows and four columns) matrix, requires nine lines from the microprocessor to make all the connections instead of twenty lines needed if the keys are connected in a linear format.

In interfacing a matrix keyboard, the major task is to identify which key is pressed and decode the key in terms of its binary value. This task can be accomplished through either software or hardware. In this section we explore both methods: First we discuss the basic concepts in interfacing a matrix keyboard, and then write subroutines to check, identify, and decode (interpret) the key pressed. Finally, we illustrate how these functions can be replaced by a hardware device, such as the National Semiconductor keyboard encoder MM74C923.

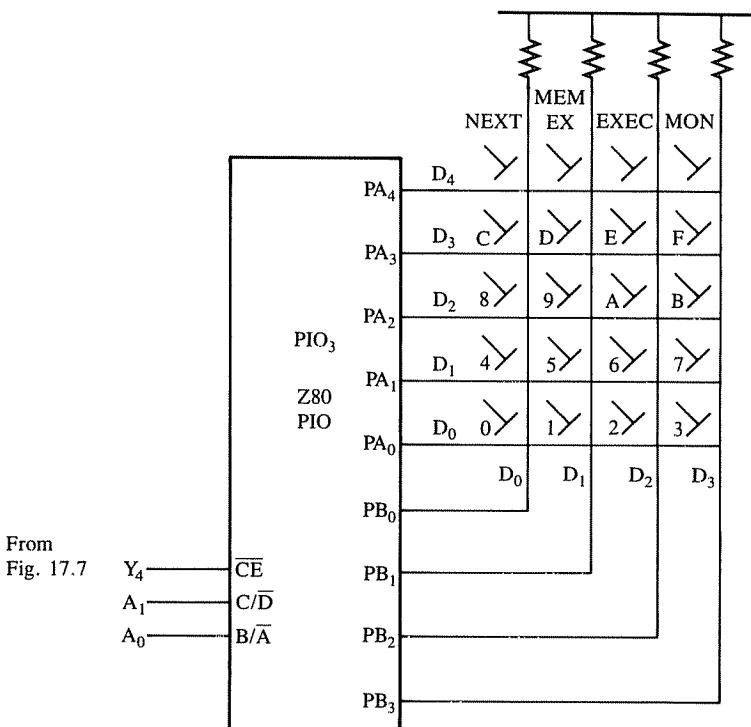


FIGURE 17.10  
Interfacing a Matrix Keyboard

### 17.51 Basic Concepts

Figure 17.10 shows a matrix keyboard with 20 keys; the keyboard has five rows and four columns. The first sixteen keys in a sequence represent data 0 to F in Hex, and the remaining four will represent various functions such as Store and Execute. The circuit includes two I/O ports: one output port and one input port. Rows are connected to the output port and columns to the input port. The columns and rows make contact only when a key is pressed; otherwise, they remain high (+5 V). When a key is pressed, the key must be identified by its column and the row, and the intersection of the column and row must change from high to low. This can be accomplished as explained in the following steps.

1. Ground all the rows by sending logic 0 through the output port.
2. Check the columns by reading the input port. If no key is pressed, all columns remain high. Continue to repeat Steps 1 and 2 until the reading indicates a change.
3. When one of the keys is pressed, the corresponding column goes low; at that point, identify and decode the key.

### 17.52 Interfacing Circuit

Figure 17.10 shows an interfacing circuit of a 20-key matrix keyboard using a Z80 PIO, identified as PIO3. This circuit uses the decoding network of Figure 17.7; the output line  $Y_4$  of the decoder is connected to the  $\overline{CE}$  line of PIO3. Therefore, the port addresses of the PIO3 range from  $90_H$  to  $93_H$  as follows:

$A_7$	$A_6$	$A_5$	$A_4$	$A_3$	$A_2$	$A_1$	$A_0$	
1	0	0	1	0	0	0	0	= $90_H$ Port A
						0	1	= $91_H$ Port B
Decoder Enable Lines			Decoder Inputs			1	0	= $92_H$ Control Register A
						1	1	= $93_H$ Control Register B

In Figure 17.10, the rows are connected to Port A, and the columns are connected to Port B; therefore, Port A should be initialized as an output port and Port B as an input port. To initialize the PIO3, the instructions are as follows:

```

PIO3A EQU 92H          ;Port A control register
PIO3B EQU 93H          ;Port B control register
PIO2: LD A, 00001111B   ;Mode 0 control word (0FH)
      OUT (PIO3A), A    ;Initialize Port A as an output port
      LD A, 01001111B   ;Mode 1 control word (4FH)
      OUT (PIO3B), A    ;Initialize Port B as an input port

```

### 17.53 Program

The matrix keyboard routine is conceptually important because it illustrates how to set up relationships between hardware binary readings and expected codes. For example, when

key “0” is pressed, the input reading at Port B will be 1 1 1 0 ( $D_3-D_0$ ); however, the binary code for that key must be 0 0 0 0 0 0 0. This conversion is performed by the software routines which are illustrated in this section. Similarly, when key “NEXT” is pressed, the input reading will be the same as for the “0” key (1 1 1 0). The software routines will have to differentiate between data and function keys.

This matrix keyboard problem can be divided into four steps (Figure 17.11).

**Step 1:** Check whether all keys are open.

In this step, the program grounds all the rows by sending 0s to the output port. It reads the input port to check the key release, and debounces the key release by waiting for 10 ms. This step is necessary to avoid misinterpretation if a key is held for a long time.

**Step 2:** Check a key closure.

In this step, the program checks for a key closure by reading the input port. If all keys are open, the input reading on data lines  $D_3-D_0$  should be 1 1 1 1, and if one of the keys is closed, the reading will be less than 1 1 1 1. (Data lines  $D_7-D_4$  are not connected; therefore, the data on these lines should be masked.)

**Step 3:** Identify the key.

This is a somewhat complex procedure. Once a key closure is found, the key should be identified by grounding one row at a time and checking each column for zero. Figure 17.11 (Step 3) shows that two loops are set up: The outer loop grounds one row at a time, and the inner loop checks each column for zero.

**Step 4:** Find the binary key code for the key.

The binary key code is identified through the counter procedure. For each row, the inner loop is repeated four times to check four columns, and for every column check, the counter is incremented. For five rows, the inner loop is repeated twenty times, and the counter is incremented from 0 to  $13_H$ —thus maintaining the binary code in the counter. Once the key is identified, the code is transferred from the counter to the accumulator. The codes 0 to F are used for data keys and the remaining codes  $10_H$  to  $13_H$  are assigned various functions as shown in Figure 17.10.

### KEYBOARD SUBROUTINE

;This subroutine checks a key closure in the keyboard, identifies the key, and ;supplies the corresponding binary code in the accumulator. It does not modify ;any register contents.

;Input: None

;Output: Binary key code in the accumulator

;Calls DBONCE, a 10 ms delay subroutine

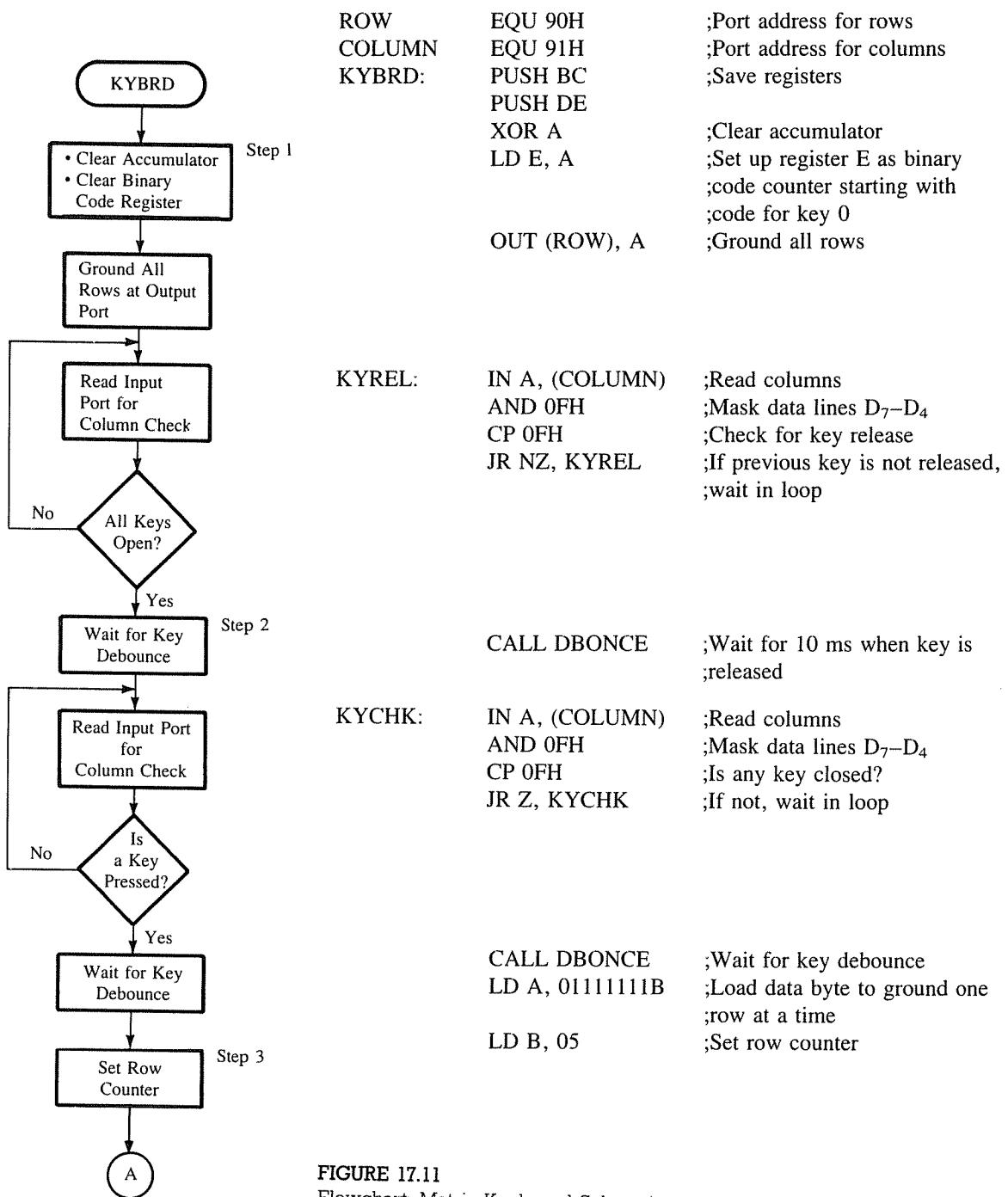
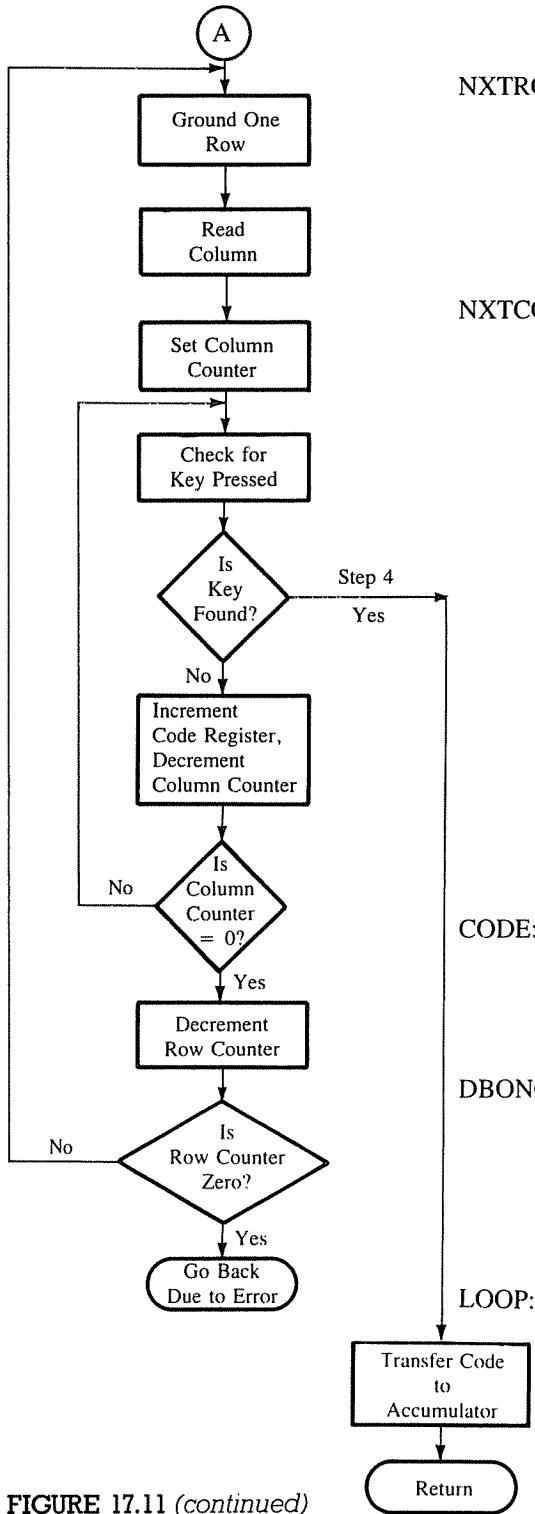


FIGURE 17.11  
Flowchart: Matrix Keyboard Subroutine



NXTROW:	RLCA LD D, A	;Move zero into one of the rows ;Save data byte to ground next ;row
NXTCOLM:	OUT (ROW), A IN A, (COLUMN) AND 0FH LD C, 04 RRA JR NC, CODE	;Ground one of the rows ;Read columns ;Mask D <sub>7</sub> -D <sub>4</sub> ;Set column counter ;Move D <sub>0</sub> into CY ;Key closure is found if zero ;is in CY
	INC E	;Increment binary code for ;next key
	DEC C	;No key closure found yet, ;decrement column counter
	JR NZ, NXTCOLM	;Check for key closure in ;next column
	LD A, D	;Load data byte to ground next ;row
CODE:	DEC B	;No key closure found in col- ;umns, get ready to ground ;next row
DBONCE:	JR NZ, NXTROW JR KYCHK LD A, E POP DE POP BC RET	;No key closure yet, go back ;to ground next row ;No key closure found, it was ;an error ;Load accumulator with binary ;code from code counter ;Retrieve data from stack ;Return to main program
LOOP:	LD BC, COUNT DEC BC LD A, C OR B JR NZ, LOOP POP AF POP BC RET	;This is a 10 ms. delay routine, does not ;destroy any register contents ;Input: None ;Output: None ;Save registers ;Load 10 ms delay count ;Repeat loop for delay ;Set zero flag if BC = 0

FIGURE 17.11 (continued)

### PROGRAM DESCRIPTION

This keyboard routine saves register contents of the calling program and clears registers A and E. Register E is used as a binary code counter for the keys; it begins with the code of “0” key. The OUT instruction grounds all the rows, and the IN instruction reads the columns. The AND instruction masks the data on lines D<sub>7</sub>–D<sub>4</sub> because they are not being used for this keyboard.

The next instruction, CP 0FH, checks whether the previous key pressed has been released; this is a precautionary step against someone holding a key for a long time. If all keys are open, D<sub>3</sub>–D<sub>0</sub> will be high, the reading will be 0F<sub>H</sub>, and the Compare instruction will set the zero flag; otherwise, the routine stays in the loop KYREL until all keys are open. The subroutine DBONCE eliminates the key bounce by waiting for 10 ms.

Once all keys are open, the routine reads the columns to check for a key closure. If any of the keys is closed, one of the columns will be at logic 0, and the routine will skip the KYCHK loop. The DBONCE routine will debounce the key closure. At this point, a key closure is found, but the key is not identified. For example, if the reading on data lines D<sub>3</sub>–D<sub>0</sub> is 1 1 1 0, any of the keys in Column 0 may have been pressed. Therefore, the next step is to identify the key.

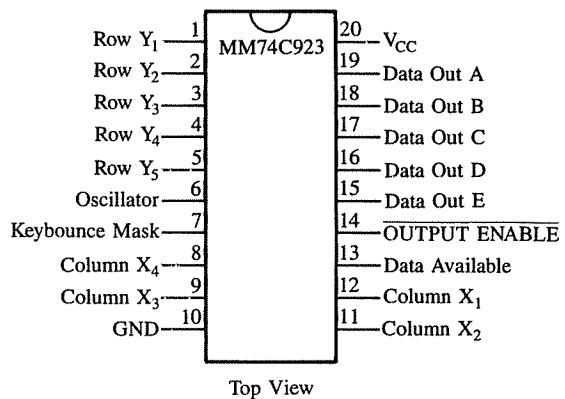
To identify the key pressed, one row is grounded at a time, beginning at Row 0. The byte 0 1 1 1 1 1 1 (7F<sub>H</sub>) is loaded into the accumulator and rotated left (RLCA) by one position; the byte is thus converted to 1 1 1 1 1 1 0. This byte is sent to Port A to ground Row 0. Then, Port B is read, and each column is checked for logic 0 by rotating the reading into the CY flag. Register C is set up to count four columns, and by rotating the byte to the left four times, each column is checked for logic 0 in the loop labelled as NXTCOLM. As each column is being checked, the code counter (Register E) is incremented at each iteration. For example, when Row 0 is grounded, four keys, 0 through 3, are checked, and the code counter is incremented from 00 to 03<sub>H</sub>.

After checking the columns in Row 0, the program loops back to location NXTROW and grounds the next row by sending the code which was previously saved in register D. Register B is set up as a row counter to count five rows. For each row, the loop NXTCOLM is repeated four times; thus, all twenty keys are checked, and for each iteration the code counter is incremented. When columns are checked, each reading is rotated into the CY flag; the key closure is found and the key identified when CY is reset. The program jumps to location CODE. The routine copies the key code into the accumulator and returns to the calling program.

### 17.54 Using the Matrix Keyboard in the Project

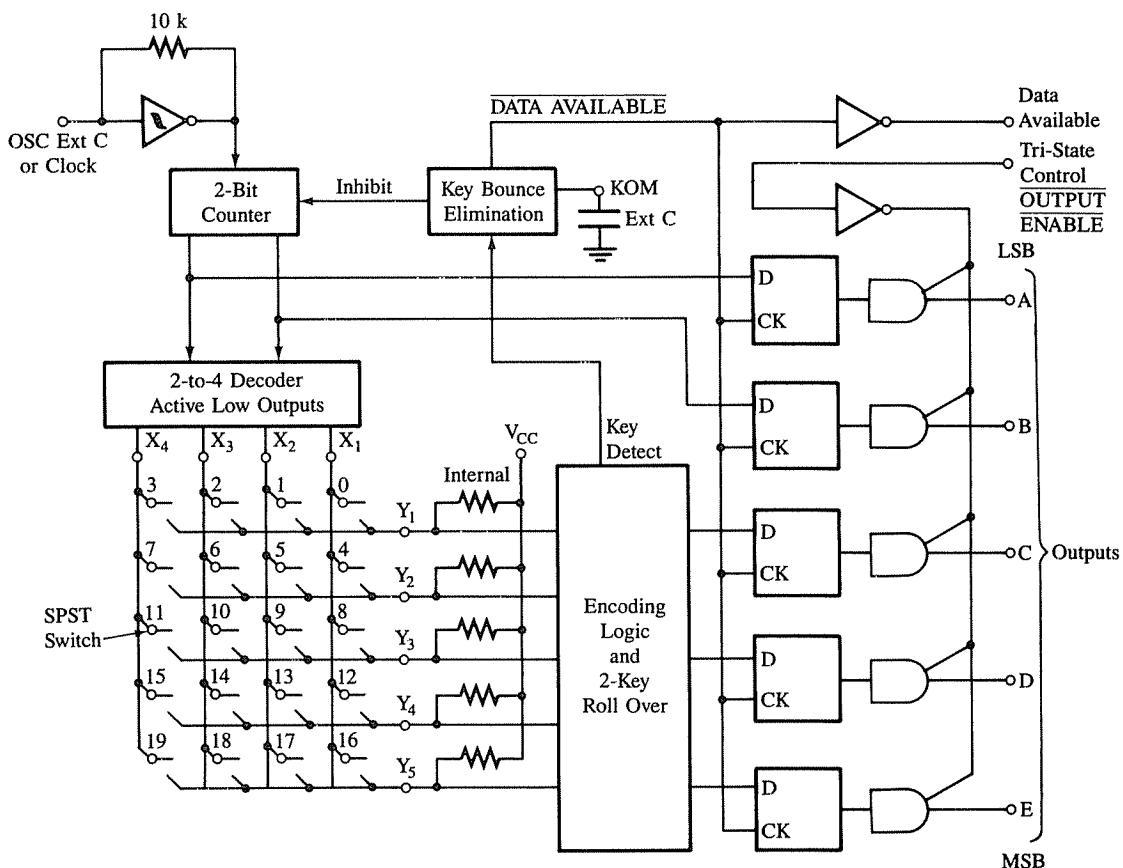
In our single-board microcomputer project, the keyboard monitor program from the previous section can be used in conjunction with the user display where the output is latched. If we were to use the scanned display (as specified) with the software-driven matrix keyboard, the keyboard subroutine would have to be coupled with the scanned display; otherwise, the display may go off. For example, when the subroutine is waiting for a key to be pressed, the scanned display cannot be refreshed by turning on and off digits in a sequence at a regular interval. Therefore, the program must alternate between refreshing

Dual-In-Line Package



Top View

(a) Pinout



(b) Block Diagram

**FIGURE 17.12**

Keyboard Encoder MM74C923 (National Semiconductor) (a) Pinout (b) Block Diagram

SOURCE: Reprinted with permission of National Semiconductor Corporation

the display and checking a keyboard to find a key pressed. Another approach is to interface the keyboard using the interrupt technique. In this approach, the program continues to scan the display until the interrupt signal is received, and then the program checks the keyboard, processes the key, and goes back to scanning the display.

### 17.55 Hardware Approaches to Interfacing Matrix Keyboard and Scanned Display

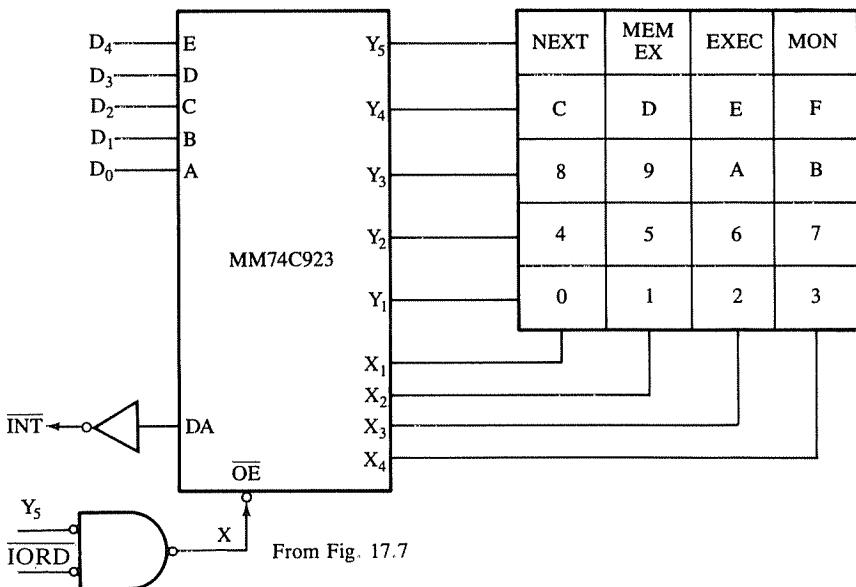
The hardware approach reduces the software and allows the MPU to perform other tasks; however, it may increase the unit cost of the product. One of the approaches is to use a logic device, such as the National Semiconductor MM74C923 keyboard encoder. This keyboard encoder can sense a key closure, debounce the key, provide the binary code of the key, and generate an interrupt. Another approach is to use a programmable device, such as the Intel 8279 keyboard/display interface. This interface device performs two tasks: One task is to detect and encode a key (this is the same task as that of the National Semiconductor keyboard encoder), and the other is to refresh a scanned display. It is capable of displaying 16 bytes. The 8279 is a complex device and will not be discussed here. However, we will illustrate how to interface a matrix keyboard using the MM74C923 keyboard encoder.

#### MM74C923 KEYBOARD ENCODER

This is a 20-key encoder with four columns and five rows (Figure 17.12). The respective columns and rows of a matrix keyboard must be connected to the columns and rows of the encoder. The encoder includes Chip Select and Interrupt logic. The decoded address line (I/O Select) is connected to the  $\overline{OE}$  pin of the encoder; it does not require a PIO or an input buffer. It has five output data lines that provide the binary code of a key closure from 00000 to 10011 (0–19).

Figure 17.13 shows the schematic for interfacing a 20-key matrix keyboard using the encoder. The keyboard is assigned the port address by connecting the output  $Y_5$  of the decoder from Figure 17.7. Thus, the keyboard can be accessed by any one of the port addresses  $94_H$  to  $97_H$ ; the address lines  $A_1$  and  $A_0$  are left as “don’t care.”

When a key is pressed, the encoder debounces the key and checks again for a valid key. If a valid key is detected, the encoder generates an interrupt and places the binary code of the key into the internal latches, and the code can be read by enabling  $\overline{OE}$ . When the MPU acknowledges the interrupt and reads the binary code, the encoder turns off the interrupt. In this interfacing, the keyboard routine is reduced to a few instructions of a service routine, which reads the keyboard and stores the code in the input buffer. This technique reduces considerable software overhead for the MPU. Therefore, the MPU can continue to scan the display until an interrupt request is received. Then, it can process the key and go back to the scanned display routine.



**FIGURE 17.13**  
Interfacing 20-Key Matrix Keyboard Using the MM74C923

devoted to writing and troubleshooting software. Software design is done neither in isolation from nor after the completion of hardware design; these are concurrent processes. In the previous sections, we discussed approaches to interfacing displays and matrix keyboards. Now we need to make some decisions based on the cost and the specifications; several alternatives are suggested here.

**Alternative 1** Figure 17.14 shows a way of combining the matrix keyboard and the scanned display. The digit code driver of the scanned display can be used to connect the rows of the matrix keyboard. The program scans the display once, turns off the display temporarily by sending logic 0 to the LED segments, and then grounds rows of the keyboard and checks the columns for a key closure. This approach requires two PIOs; three ports can be used for the display and the keyboard, and the fourth port can be used for the user display. Now the question is: Why do we need a separate user display? Can we use the data LEDs of the system display as the user display? The scanned display is not convenient for the user to use as an output port; the user can use that port only by calling routines in the monitor program. To have a separate latched output port is a worthwhile feature for beginning users.

This approach requires 14 components (two PIOs, two drivers, two Hex decoders, and eight LEDs) and a keyboard. This design is heavily software-dependent, and the microprocessor is kept totally busy in scanning the keyboard and display.

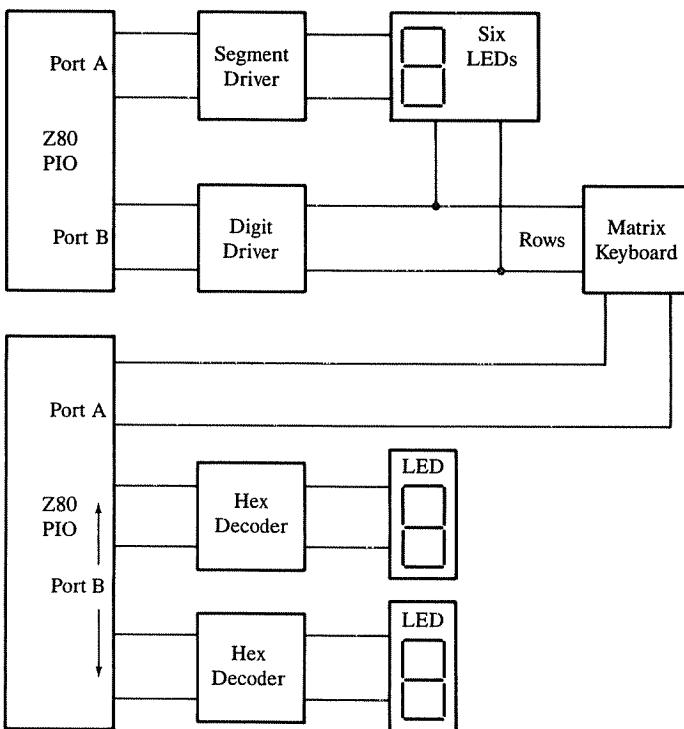
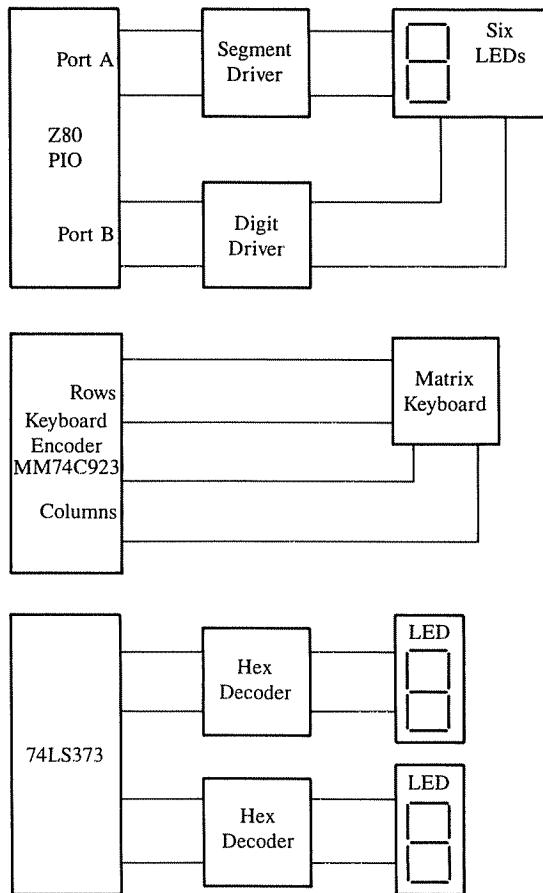


FIGURE 17.14  
Combining Scanned Display and Matrix Keyboard—System Design Alternative 1

**Alternative 2** The second alternative is to use four LEDs in the scanned display; these can be used to display a memory address. The two-LED-latched output port can be used for system data and by the user as well. When the user program is being executed, the program control is transferred to the user program; therefore, the system display will not be used. Furthermore, when a latched port is used for data, the software overhead is reduced; there is no need to continuously scan the data port. The block diagram for this alternative is similar to the first alternative.

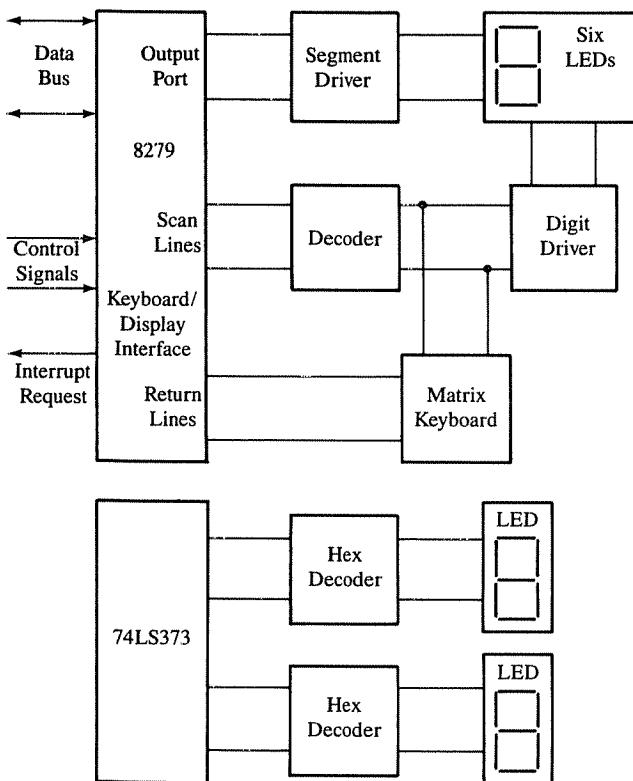
**Alternative 3** The third alternative is to replace the keyboard routine with a keyboard encoder (MM74C923). This approach eliminates the software related to checking and encoding a key. The program has to continuously scan the display, and when a keyboard encoder generates an interrupt, the MPU can read the keyboard, process the key, and go back to scanning the display. In this approach, we need one PIO for the system display, and the other PIO can be replaced by an octal latch such as the 74LS373 as shown in Figure 17.15. The block diagram shows that the design requires 15 components (in addition to a keyboard, which is common to all circuits).

**FIGURE 17.15**  
System Design Using the  
MM74C923 Keyboard Encoder—  
System Design Alternative 3



**Alternative 4** The fourth alternative is to replace the display scanning and keyboard checking with a programmable keyboard encoder, such as the Intel 8279. This device relieves the burden of scanning the display and checking the keyboard from the MPU. When a key is pressed, the 8279 generates an interrupt to inform the MPU. When the MPU reads the keyboard, it places the code in the encoder memory, and informs the encoder how many LEDs to scan. This simplifies the software necessary for the keyboard monitor and allows the MPU to perform other tasks.

Figure 17.16 shows the logic diagram of the 8279, which includes interfacing logic, control signals, system data bus, eight data lines to drive LED segments, eight return lines, and four scan lines. The 8279 is functionally equivalent to the circuit shown in Figure 17.14 without the user display. To scan more than four LEDs, additional scan lines can be generated by using a decoder.



**FIGURE 17.16**  
System Design Using the 8279 Keyboard/Display Interface—Alternative 4

The design shown in Figure 17.16 requires 15 components. This is a hardware approach to keyboard and display scanning at a somewhat higher unit cost.

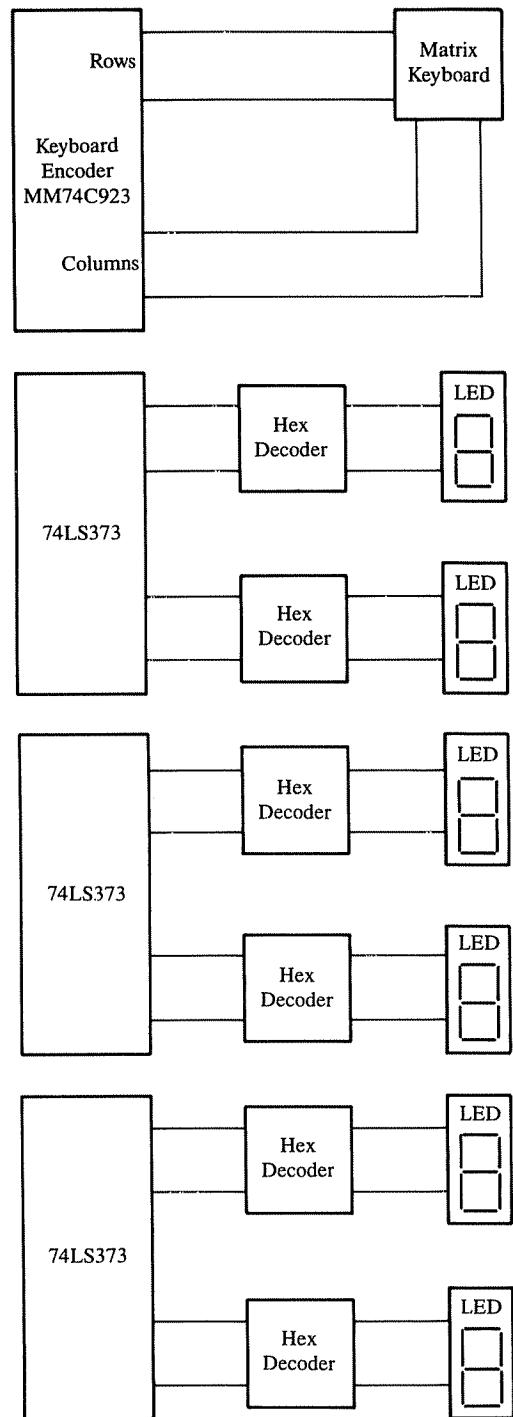
**Alternative 5** The fifth alternative is to use a keyboard encoder, such as the National Semiconductor MM74C923, for the keyboard and use the latched LED ports as displays. This will simplify the software considerably.

Figure 17.17 shows that three octal latches are used as output ports to interface six LEDs. One of the output ports can be used as the user port; thus, the total number of components required in this design is 16. This approach increases the power consumption in LEDs, but relieves the MPU from the scanning task.

We have suggested various alternatives, and now we will discuss software routines for some of these approaches. The most puzzling aspect of software design is where to begin and how to synthesize all functions into one program. The place to begin is the list of the functions to be performed. In the project analysis section, three functions are listed: check keyboard, display, and execute. The next place to look for clues is hardware.

**FIGURE 17.17**

System Design Using the  
MM74C923 Keyboard Encoder and  
Octal Latches



Examination of the hardware design reveals the following:

1. Program should begin at location  $0000_H$ .
2. "Low" memory locations should be reserved for interrupt restarts.
3. Programmable peripherals need initialization instructions.
4. As the system is turned on, a message should be displayed.
5. Four keys are available to identify functions, and 16 keys are used as Hex digits from 0 to F.

By combining the functions to be performed and the clues obtained from hardware design, the task can be divided into the following steps:

1. Initialize programmable peripherals.
2. Display the sign-on message to indicate that the system is ready.
3. Scan the display once and jump to check a key closure. If no key is pressed, go back and scan the display.
4. When a key closure is found, read the binary code of the key to check whether it is an appropriate key.
5. If it is an appropriate key, turn off the sign-on message, process the key, and return to the display loop to display the new key; otherwise, blank the display.
6. If it is not an appropriate key, return to the display loop to indicate the error message.
7. If the keyboard is interrupt driven, stay in the display loop until an interrupt is generated (this is an alternative to Steps 3–6).

The first three steps are fairly simple. The initialization is determined by peripheral devices and their decode logic (discussed previously). The display involves scanning the sign-on message.

The fourth step—the determination of an appropriate key—is critical to the software design, and the appropriateness depends upon how the user is allowed to enter and execute a program. There are two basic approaches: One approach is to begin with a function key and then enter Hex digit keys, and the second is to enter a memory address and then specify the function to be performed. In addition to the Reset key, at least three keys are required: MEMEX (Memory Examine), NEXT (Next Memory Location), and EXEC (Execute). The MEMEX key allows the user to enter the memory address, examine the data stored in that memory, and enter new data. The NEXT key stores the new data byte and increments to the next memory location. The EXEC key allows the user to execute a program. With this minimum configuration, if an inappropriate sequence of keys is pressed, and the program displays the error message, it can be terminated only by the Reset key. Therefore, a key called MON (Monitor) is added to terminate a program.

In the first approach, whereby the user begins with a function, we will use four keys: MON (Monitor), MEMEX (Memory Examine), NEXT (Next Address), and EXEC (Execute). If the MON function is selected, the program goes to the beginning and displays a Ready message, and if the EXEC function is selected, the program transfers the control to the user program. If the MEMEX key is pressed, the program displays the contents of the

memory, and if the NEXT key is pressed, the program increments to the next memory location. After the MEMEX key, the user is allowed to enter new data or go to any one of the functions.

In the second approach, the user must begin with a memory address and then specify a function; otherwise, an erroneous result will be displayed. Now we can make some hardware design decisions and illustrate an approach to software design.

Among the five design alternatives we have discussed, it appears that Alternative 1 (Figure 17.14) is the least expensive in unit price, but it is extensively dependent on software. Alternatives 4 and 5 (Figures 17.16 and 17.17) are hardware oriented and may be more expensive in unit price. We prefer to select Alternative 3 (Figure 17.15) based on instructional rather than commercial reasons. Our primary reason for the selection is that the approach illustrates important concepts without being excessively dependent on software. The specifications are as follows:

1. Memory Map: EPROM 0000<sub>H</sub> to 07FF<sub>H</sub> (Figure 17.6)  
R/WM 2000<sub>H</sub> to 27FF<sub>H</sub> (Figure 17.6).
2. System Display: Scanned display with six LEDs using PI01  
Port addresses—80<sub>H</sub> to 83<sub>H</sub> (Figure 17.8).
3. User Display: Latched LED port using the octal latch 74LS373.
4. Keyboard: 20-key matrix keyboard using the encoder MM74C923  
Port is interrupt driven with address 94<sub>H</sub> (Figure 17.13)  
Four function keys—MON, MEMEX, NEXT, and EXEC.

Based on these specifications, we will illustrate an approach to software design in the next section.

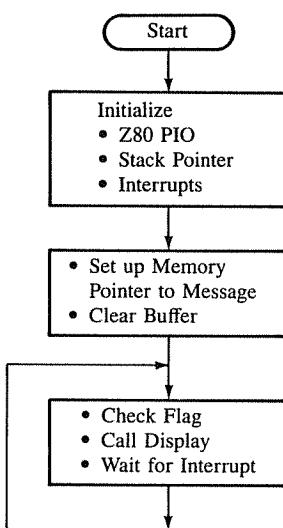
## 17.7 DESIGNING SOFTWARE MODULES

---

The proposed system includes a scanned display that needs refreshing at a regular interval and a matrix keyboard that is interrupt driven. Therefore, the main program revolves primarily around refreshing the display and waiting for an interrupt to occur, as shown in Figure 17.18. The primary task of the interrupt service routine is to read and process the key pressed and perform the designated function of the key. We can divide the software design into the following modules:

1. Initialization.
2. Display the sign-on message and wait for an interrupt.
3. When an interrupt occurs, read the key.
4. Decode the key and jump to the appropriate function.
5. Perform the function and return to scanning the display.

**FIGURE 17.18**  
Flowchart: Main Program



### 17.71 Initialization

When a system is reset, the Z80 clears the program counter, and the program execution begins at location  $0000_H$ . If the system includes several sources of interrupts, the initial memory locations can be used for Mode 0 interrupts. However, in the Z80, Mode 2 interrupts can be placed anywhere in the memory map; therefore, there is no compelling reason to save these interrupt call locations. The initialization program module can be written in the beginning segment of the 2732 EPROM.

This module must initialize the programmable devices (PIOs) and the stack pointer, enable the interrupts, and set up the interrupt mode. In Section 17.4, we have already written the initialization instructions for the Z80 PIOs. The stack pointer is generally initialized at the top of the R/W memory; however, in this project, we will need six top locations as a display buffer and one location to save input data. After reserving top locations (for example, from  $27F9_H$  to  $27FF_H$ ) for the buffer (explained in the next section), the stack pointer can be initialized. In the Z80, Mode 1 interrupt is the simplest to implement because it does not require any external hardware. When an interrupt is generated by the keyboard encoder and accepted by the MPU, the program is automatically transferred to location  $0038_H$ . Thus, the keyboard service routine must begin at  $0038_H$  or a Jump instruction must be written at  $0038_H$  to locate the start of the interrupt service routine.

### 17.72 Display Module

We have already discussed the scanned display routine in Section 17.4. That routine can be used for any fixed message such as the sign-on message or the error message; the codes

for these messages can be stored permanently in the EPROM. However, to display memory address and data that change with key strokes, we need to reserve memory locations: four for memory address and two for data in the R/W memory. These locations are called the *display buffer*. For example, in this single-board microcomputer, the R/W memory ranges from  $2000_H$  to  $27FF_H$ ; we can reserve the last six locations,  $27FA_H$  to  $27FF_H$ , as the display buffer. The display routine must be modified to scan the display buffer and get the segment code by using the table look-up technique, output the code, and turn on the corresponding digit (Figure 17.8). In addition, the routine must be informed of the number of digits to be scanned. For example, when a memory address is entered, four digits are displayed, and when the MEMEX key is pressed, six digits are displayed.

Now we need to find a way to inform the routine of the number of digits to be displayed and how to differentiate between data keys and address keys. This can be accomplished by using the flag concept. The routine that calls the codes to be displayed sets a flag when four digits are to be displayed. For example, bit  $D_7$  in register B can be used as a flag. When  $D_7$  is 0, the routine scans four memory digits, and when it is 1, it scans the six digits. The CY can also be used to perform the same function. In block 3 of the main program, this flag concept is used. Before calling the Display routine, the program checks the flag to determine whether it should scan four locations or six locations of the buffer.

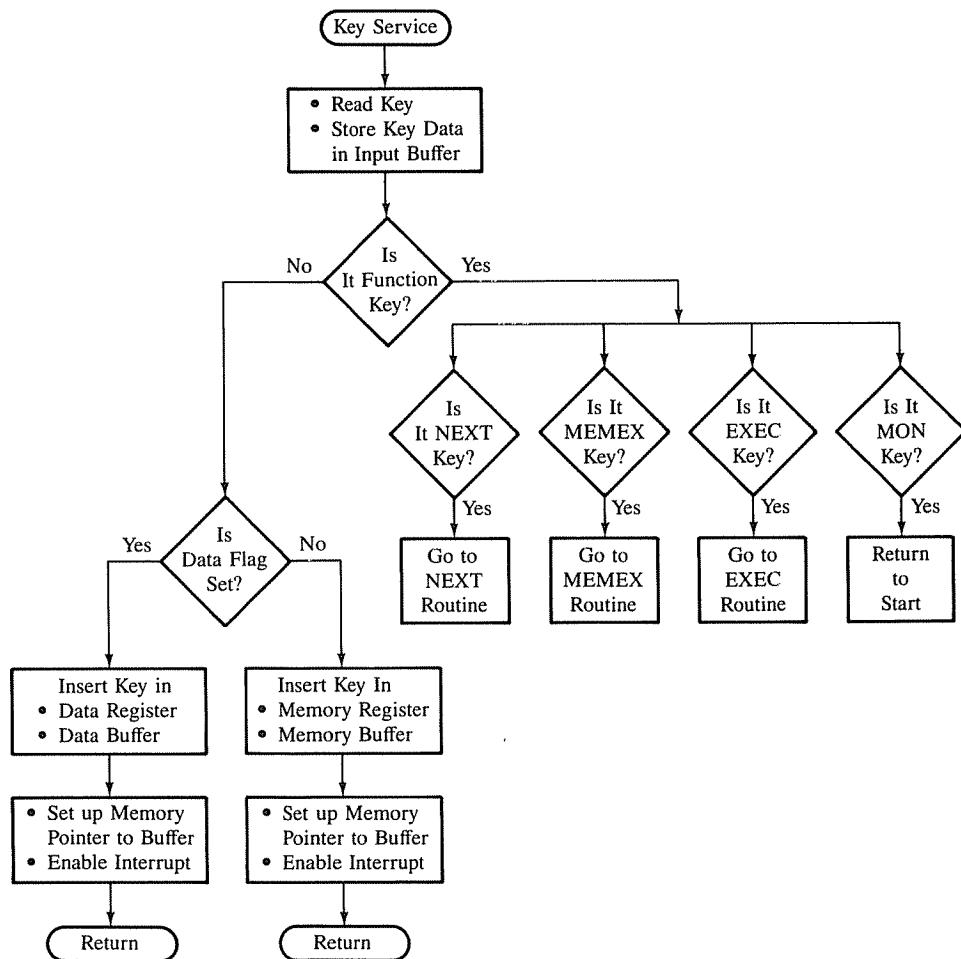
### 17.73 Reading the Keyboard and Placing the Byte In the Buffer

When an interrupt request is generated by the keyboard encoder, the MPU should read the keyboard and save the reading in the input buffer (location  $27F9_H$ ). If it is a Hex key of a memory address or of a data byte, the MPU should place the binary value of the new key in the buffer. In addition, a register pair such as DE can be used as the memory register to save the memory address and register C can be used as the data register. However, before the key code is placed in the buffer, the previous codes must be shifted by one location, and in the process, the MSD (Most Significant Digit) must be discarded (see Example 11.4).

Figure 17.19 shows the partial flowchart of the interrupt service routine, which begins with reading a key and saving the data in the input buffer. Then the program checks whether it is a function key or a Hex digit key; the keys with binary code 00 to  $0F_H$  are Hex digit keys and with binary code higher than  $0F_H$  are function keys. If a function key is pressed, the program determines whether it is the MEMEX, NEXT, MON, or EXEC key and then jumps to appropriate locations. If it is a Hex digit key, it checks the flag (bit  $D_7$ ) in register B and identifies the key that is a part of a memory address or of a data byte. Then the program inserts the code of the new key as the least significant four bits in the display buffer and in the memory register or data register and returns from the interrupt service routine.

### 17.74 Function Module

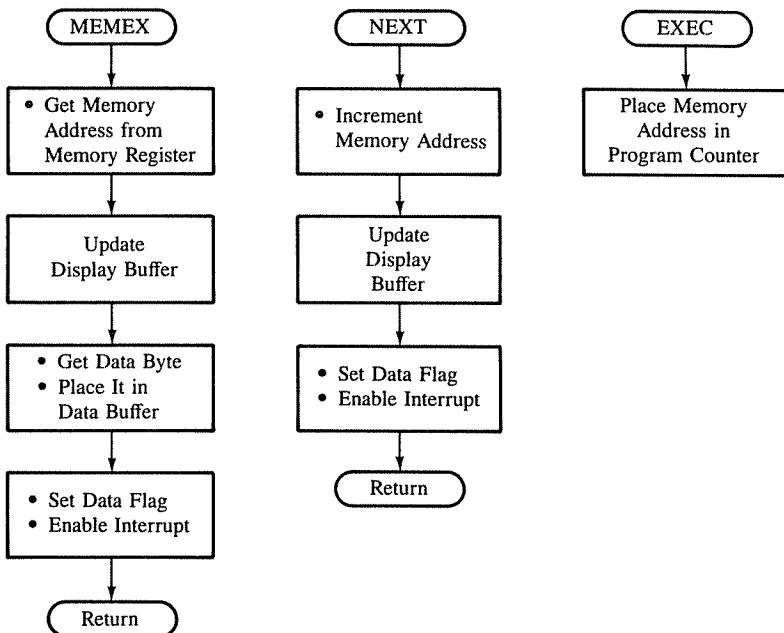
The user must enter four Hex digits as a memory address. If more than four Hex keys are entered, the binary code of the last four is saved in register DE, and the display buffer is updated accordingly. If a function key is pressed after an address is entered, the program checks which function key is pressed. If the MEMEX key is pressed, the memory address



**FIGURE 17.19**  
Flowchart: Key Service Routine

is already in the DE register. This address can be used as a memory pointer, and data from that memory location can be retrieved and placed in the display buffer (Figure 17.20). Similarly, when the NEXT key is pressed, the address in the DE register is incremented, the data byte in that location is obtained, and all six digits are placed in the display buffer.

When a data byte is being entered, we can use register C to save data keys, as mentioned earlier. When a new key is pressed, the most significant nibble can be discarded, and the binary code of the new key can be entered as a least significant nibble (see Assignment 3).



**FIGURE 17.20**  
Function Routines

If the key is EXEC, the memory address where the execution should begin is already in the DE register pair. The program places the memory address in the program counter, and the control is transferred to the user program. If the key is MON, the program returns to the beginning and displays the system-ready message.

In writing this monitor program, the critical issue to remember is that the system uses the scanned display and needs continuous refreshing. Therefore, the main program consists primarily of calling the display routine. The next steps are to code this program in Z80 assembly language and to test it on prototype hardware, using such debugging tools as an in-circuit emulator and a logic analyzer (discussed later).

### PROGRAM CODING

Assuming that program coding is to be performed by a team, it is necessary to break down the task into small, manageable, and independent modules. It is not always possible to break logic flow into independent subroutine modules. However, it is necessary to agree on symbols or labels that might be used by various members of the team; these are called global symbols.

### 17.75 Prototype Building and Testing

Microprocessor-based products are hardly ever completely built and tested as complete systems during the initial stages of design. If a system is completely built, it is difficult to

troubleshoot. Traditional approaches, such as signal injection and isolation of trouble spots, are ineffective for troubleshooting bus-oriented systems. Therefore a system is built and tested in stages. Each subsystem, such as keyboard, displays, and memory, should be built and tested separately as an independent module. Now the question is: How do we test a module without building a system? An answer can be found in such everyday incidents as testing a light bulb or starting a car with a dead battery. The light bulb can be tested by plugging it into a working socket, and the car can be started with a jumper cable and another battery. There are two principles involved in these examples: (1) borrowing resources from a working system, and (2) substitution. These principles can be used in testing each separate subsystem of a microprocessor-based product. What is needed is a working system that can create an environment similar to the complete prototype system, and that is generous enough to share its resources with hardware modules to be built. Such a working system is called an in-circuit emulator, and is described in Section 17.8 and shown in Figure 17.21.

Assuming that such an in-circuit emulator is available, each subsystem of the single-board microcomputer can be built and tested one at a time. Similarly, as software modules are being written, they can be tested first on a software development system (discussed in Chapter 7). Finally, hardware and software can be integrated and tested using an in-circuit emulator.



**FIGURE 17.21**  
Photograph: In-Circuit Emulator—Applied Micro Systems  
SOURCE: Courtesy of Applied Microsystems

## 17.8 DEVELOPMENT AND TROUBLESHOOTING TOOLS

---

In bus-oriented systems, a constant flow of data changes logic states continuously. The flow of data is controlled by software instructions. Therefore, to examine what is happening inside the system, special instruments capable of capturing data in relation to instructions are required. Three such instruments—in-circuit emulator, logic state analyzer, and signature analyzer—are discussed briefly in the next sections.

### 17.8.1 In-Circuit Emulator

The in-circuit emulation technique has become an essential part of the design process for microprocessor-based products. In-circuit emulation is the execution of a prototype software program in prototype hardware under the control of a software development system. First, the microprocessor is removed from the prototype design board, and a 40-pin cable from an in-circuit emulator is plugged into the socket previously occupied by the microprocessor. The in-circuit emulator performs all the functions of the replaced microprocessor; in addition, it allows the prototype hardware to share all its resources, such as software, memory, and I/Os. It provides a window for looking into the dynamic, real-time operation of the prototype hardware. At present, a wide variety of in-circuit emulators is available, ranging from universal emulators with complete software development systems to stand-alone microprocessor units. Figure 17.21 shows a stand-alone in-circuit emulator (MT-180) designed by Applied Micro Systems.

**Emulation Process** To test subsystems (such as I/O and memory) using an in-circuit emulator, the minimum prototype hardware required is a 40-pin microprocessor socket (without the microprocessor), a power supply, and a system clock. All other resources can be borrowed from the in-circuit emulator. As more and more prototype hardware is built, fewer and fewer resources from the in-circuit emulator will be required. In the final stage, total software and hardware are integrated for testing. A hardware prototype can be viewed as a fetus growing in stages in the womb of an in-circuit emulator; until the fetus is fully developed and functioning independently, the in-circuit emulator provides the necessary environment and resources.

**Features of the In-Circuit Emulator** An in-circuit emulator is a software/hardware troubleshooting instrument. It can be a stand-alone unit or part of a software development system. A small program can be entered directly into the emulator, or a program can be transferred into the emulator from a host computer system through an RS-232 serial link. Once a program is loaded, a user can interact with the emulator through its keyboard or a terminal. The emulator has its own software commands to perform various debugging functions. The main capabilities of an in-circuit emulator can be listed as follows:

- Downloading:** Facilities to transfer programs between a software development system or a host computer and the in-circuit emulator.

- **Resource Sharing:** The in-circuit emulator allows the system being tested to share its memory and I/O ports. The memory and I/O ports of the in-circuit emulator can be assigned any addresses, which will avoid conflict with memory and I/O of the prototype; this is called memory and I/O mapping.
- **Debugging Tools:**
  - Real-Time Trace
    - Breakpoints
    - Mnemonic Display
    - In-Line Assembly
    - Register Display/Modification
    - Disassembly

## DEBUGGING TOOLS

The debugging tools listed are used in troubleshooting programs. Single-stepping and setting breakpoints have already been discussed in Chapters 7 and 8. The others are briefly described as follows.

**Real-Time Trace** The in-circuit emulator has R/W memory used as a buffer to store the last several (128, for example) bus operations, and these can be displayed on the screen. This display is like a snapshot of all the bus operations in real time. The user can specify several requirements, such as a memory address and certain data conditions for recognizing an event, in order to trigger and display a trace. Similarly, a trace can be observed between two breakpoints or at a specified delay after a certain event. The real-time trace is a very valuable tool in debugging microprocessor-based products.

**In-Line Assembly** This allows the user to change data or instructions while software is in the in-circuit emulator.

**Disassembly** After instructions are changed in the in-circuit emulator, this facility can write mnemonics in software.

**Register Display** This displays register contents after the execution of instructions.

### 17.82 Logic State Analyzer

The logic state analyzer, also known as the logic analyzer, is a multitrace digital oscilloscope specially designed to use with microprocessor-related products. In a multitrace scope, the timing relationships of several signals can be observed with respect to some triggering event or events. For example, a four-trace scope can show the timing relationships of four signals. In a microprocessor-related product the user is interested in observing digital signals on the address bus, the data bus, the control bus, and possibly on an external instrument, relative to a specified triggering event or events. Furthermore, data display should be in a conveniently readable format, such as Hex or binary. The logic analyzer performs these functions.

A typical logic analyzer designed primarily to work with microprocessors has a 40-pin probe plus an auxiliary probe to gather external information. It includes Read-Only

Memory (ROM) to store instructions related to the analyzer, R/W buffer memory to store data from a product under test, a microprocessor to monitor data gathering, and a keyboard to specify operations and enter data in Hex or octal format. The analyzer can be triggered to gather information at a specified event related to the microprocessor in the product under test or in relation to an external word. The analyzer in a trace mode takes a snapshot of real-time information at a specific trigger, stores it in its buffer memory, and displays it on its CRT.

The in-circuit emulator is a very valuable tool in the initial stages of product development, and in later stages the logic analyzer can perform some of the troubleshooting functions.

### 17.83 Signature Analyzer

The signature analyzer is an instrument used in troubleshooting microprocessor products either in the field or during production. This instrument converts the complex serial data stream present at the intersections of logic circuits, called nodes, into a four-digit pattern called a signature. Conceptually, a signature is similar to a voltage level specified on the schematic of an analog product. To troubleshoot an analog product, voltages are measured at various locations until a mismatch is found between the measured reading and the specified reading to isolate the trouble. The signature analyzer is used in the same manner.

---

## SUMMARY

---

In this chapter, various techniques of interfacing the scanned display and the matrix keyboard were illustrated, and the trade-offs between hardware and the software were discussed. Then we used the scanned display and keyboard illustrations in designing a single-board microcomputer. In addition, debugging tools such as the in-circuit emulator, the logic analyzer, and the signature analyzer were introduced.

The design of a single-board microcomputer integrates all the concepts of microprocessor architecture, software, and interfacing discussed throughout this text. In this chapter, we discussed the necessary steps in designing hardware and software. The necessary software modules were illustrated with flowcharts; however, the coding of these modules has been reserved for Assignments.

---

## ASSIGNMENTS

---

1. Draw a schematic to interface a 16-key matrix keyboard using one PIO port. Explain how the PIO should be initialized.

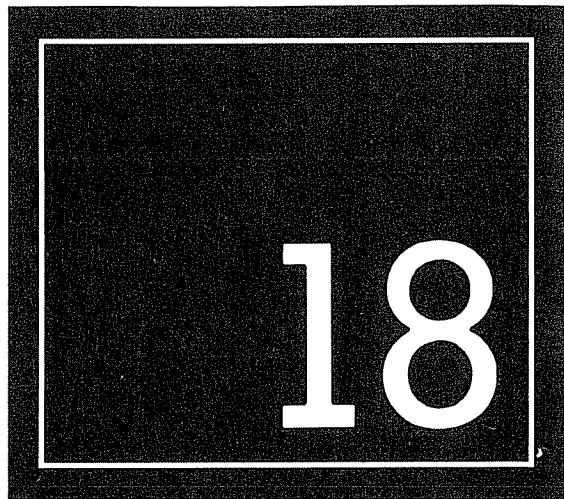
2. Draw a schematic to interface a 30-key matrix board and a six-LED scanned display using three PIO ports. Combine the matrix columns and the digit drivers, and explain why it is possible.
3. In a key monitor program, register C is used to save 4-bit codes of two data keys. Write a subroutine to insert a new 4-bit key code that is available in the accumulator; the new code must be inserted as a low-order nibble, and the most significant nibble in register C must be discarded.
4. Write instructions to unpack the data keys in 3 and place the codes in two different memory locations of the output buffer.
5. In a monitor program, register DE is used to save a 16-bit memory address. Write instructions to insert a 4-bit code of a new key in DE as a least significant nibble.
6. In 5, unpack all the codes and store them in four memory locations in the output buffer.
7. In Section 17.5, modify the matrix keyboard routine to accommodate 30 keys (six rows and five columns).
8. Modify the program in Section 17.43 to display an error message as Err and blanks.
9. Write instructions for the EXEC module assuming the memory address is in register DE.
10. Write a subroutine to transfer a 16-bit address from register DE and a data byte from register C into the display buffer ( $20FA_H$  to  $20FF_H$ ); the least significant nibble of the memory address should be placed in location  $20FA_H$  and the least significant nibble of the data byte in location  $20FE_H$ .
11. Write a Display subroutine that takes the unpacked memory address and the byte from the buffer, looks up the seven-segment code, sends the segment codes to the segment driver, and scans the digit code in a sequence to display the address and the byte.



# Trends in Microprocessor Technology

The advent of the microprocessor is having an impact on industries as diversified as machine tools, chemical processes, medical instrumentation, and sophisticated guidance control. Some applications require simple timing and bit set/reset functions, while others require high-speed data processing capability. Therefore, different microprocessor families are being designed to meet these diversified requirements. In addition to general-purpose 8-bit microprocessors, microprocessor technology has evolved in several directions as follows: (1) the complete microcomputer, commonly known as a microcontroller, on a single chip geared toward specific application; (2) the 16-bit and 32-bit microprocessors with general-purpose capability similar to mini- and mainframe computers, and (3) the integrated super 8-bit chips.

This chapter includes brief descriptions of single-chip microcomputers, 16-bit and 32-bit microprocessors, and integrated 8-bit chips; in addition, it examines recent trends in this fast-changing technology and their implications for industry.



## OBJECTIVES

- List the elements of a single-chip microcomputer (microcontroller), and compare the characteristics of Zilog Z8 and Intel MCS-51 microcontrollers.
- Describe important features of 16-bit microprocessors, and explain the concepts of memory segmentation, parallel processing, queueing, and coprocessing.
- Compare the features of the Intel 8086, the Zilog Z8000, and the Motorola MC68000 16-bit microprocessors.
- Explain the differences between the operating environments of single-user systems and multi-user systems.
- Explain how 32-bit microprocessors differ from 16-bit microprocessors.
- Explain the features of integrated super 8-bit chips, such as Zilog Z280 and Toshiba HD64180.

## 18.1 SINGLE-CHIP MICROCOMPUTERS (MICROCONTROLLERS)

---

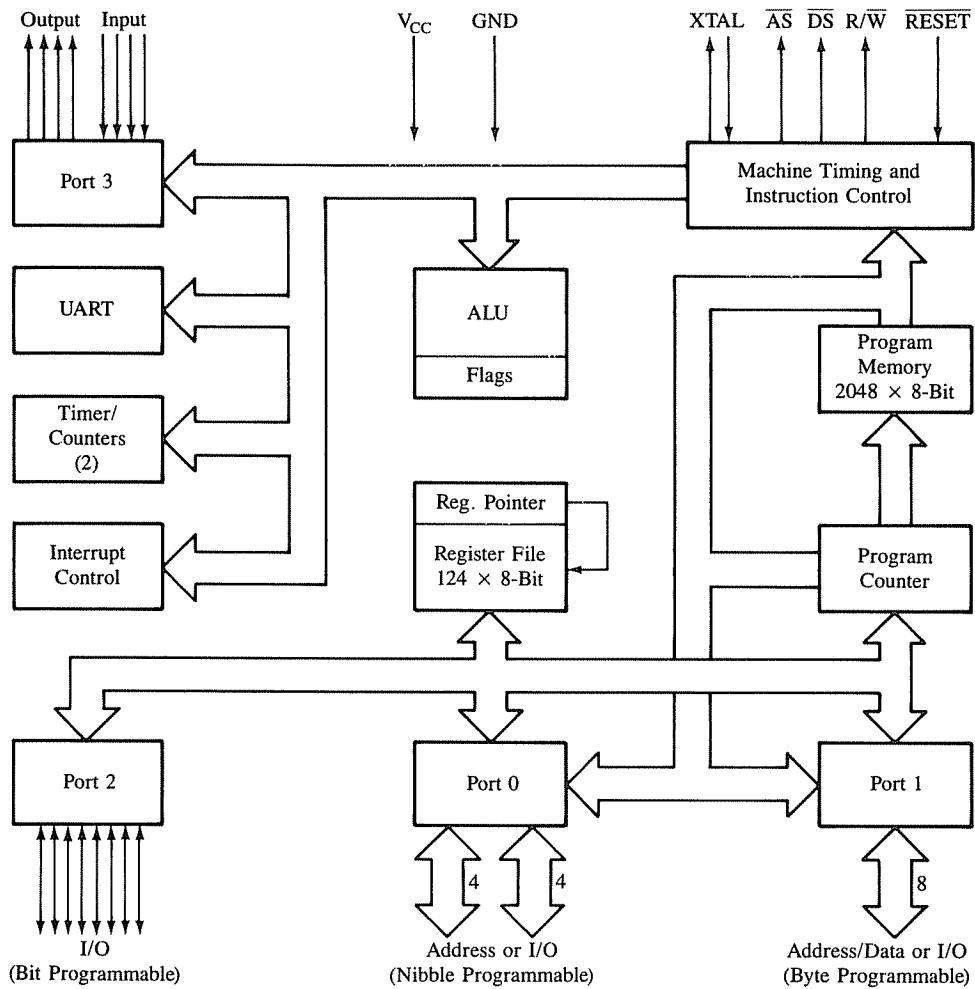
Single-chip microcomputers, also known as microcontrollers, are used primarily to perform dedicated functions. They are used as independent controllers in machines or dedicated to perform specialized functions in a larger system. Generally, they include all the essential elements of a computer on a single chip: MPU, R/W memory, ROM, and I/O lines. Examples of the single-chip microcomputers are the Zilog Z8; the Intel MCS-48, MCS-51, and MCS-96 families; the Motorola 6801; and the Fairchild F8(two-chip).

Most of these microcontrollers have an 8-bit word size (except the MCS-96, with a 16-bit word size), at least 64 bytes of R/W memory, and 1K bytes of ROM. The range of I/O lines varies from 16 to 32 lines. However, most of these devices cannot be easily programmed in college laboratories except those with EPROM on the chip, such as the Intel 8748 and 8751. A variety of single-chip microcomputers is available on the market to meet diversified industrial needs. To illustrate the trend, we will describe the Zilog Z8, and the Intel MCS-51 and 2920.

### 18.11 Zilog Z8 Microcomputer

The Z8 microcomputer is a versatile and powerful 8-bit single-chip microcontroller, used primarily in dedicated control applications. The Z8 family includes three versions: the 40-pin with ROM, the 40-pin with EPROM, and the 64-pin version, and they can operate with 8 MHz frequency.

Figure 18.1 shows the block diagram of the Z8 microcomputer. It includes four I/O ports (32 I/O lines), 2K ROM or EPROM, 128 bytes of R/W memory, two 8-bit timer/counters, and one serial I/O port (UART). It has 144 registers, including 124 general-purpose registers which can function as accumulators, address pointers, or index registers. It is capable of addressing 124K bytes of external memory. The Z8 has six interrupts, and each of the interrupts has a 16-bit vector that can point to its service routine. These interrupts can be prioritized through programming. The instruction set is quite

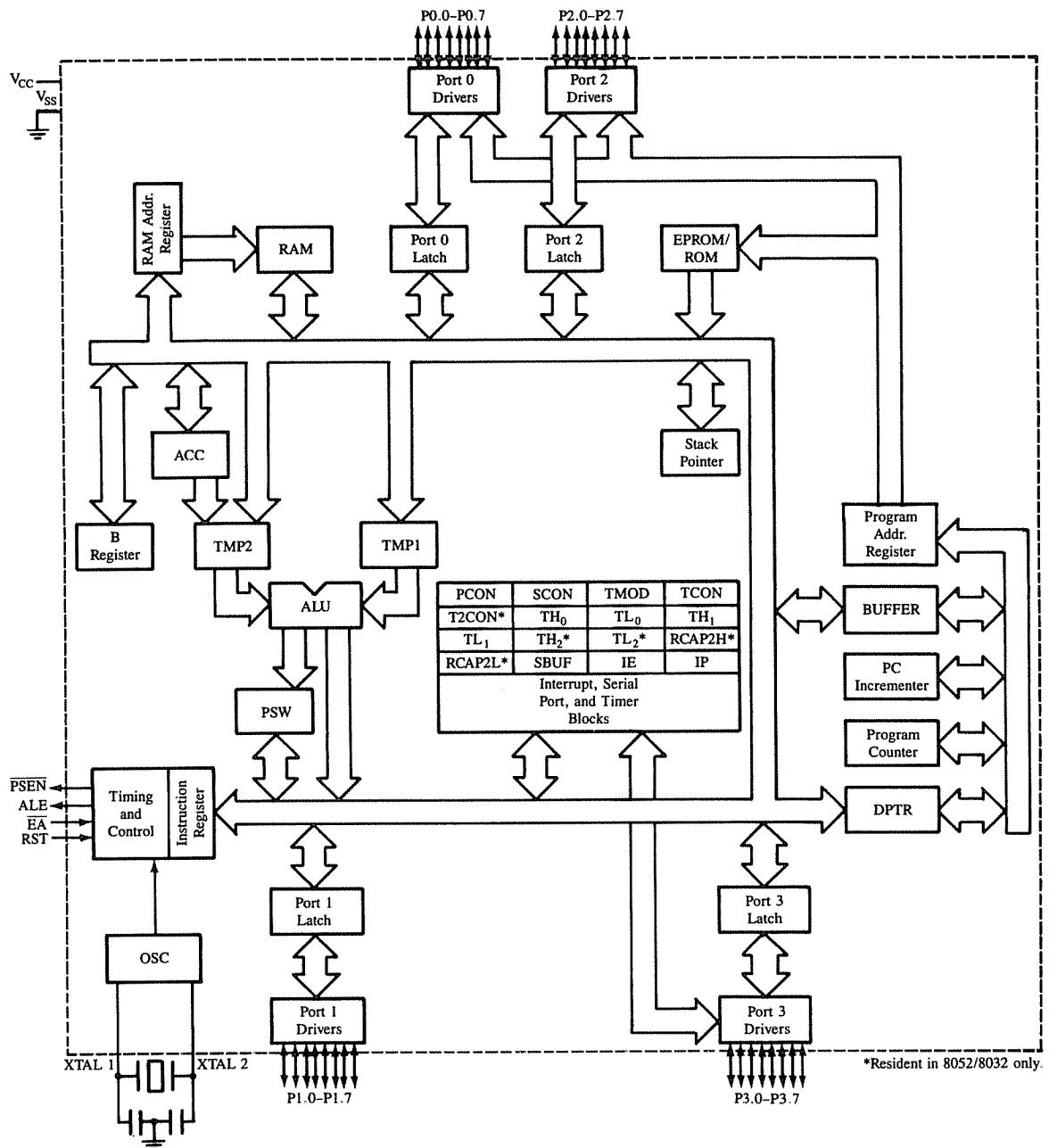


**FIGURE 18.1**  
Z8 Block Diagram  
SOURCE: Courtesy of Zilog, Inc.

powerful and especially suited for control applications. It has 46 instruction types that include bit manipulation, BCD operations, conditional and relative branching, and block transfer.

### 18.12 Intel MCS-51 Single-Chip Family

This is one of the Intel's single-chip microcomputer families, at the high end of the single-chip device spectrum in terms of its capability and versatility. It is designed for use



**FIGURE 18.2**  
MCS-51 Architectural Block Diagram  
SOURCE: Courtesy of Intel Corporation.

in sophisticated real-time instrumentation and industrial control. It can operate with a 12 MHz clock and has a very powerful instruction set.

Figure 18.2 shows the block diagram of the chip; its architecture is in many ways similar to the Zilog Z8 microcomputer. It includes the following features:

1. 4K bytes of ROM or EPROM.
2. 256 bytes of R/W memory, which includes 128 8-bit, special-function registers (SFR).
3. Four programmable I/O ports.
4. Two 16-bit timer/event counters.
5. A serial I/O port with a UART.
6. Five interrupt lines: two for external signals and three for internal operations.

The MCS-51 is known as a *bit and byte processor*. The instruction set includes binary and BCD arithmetic operations, bit set/reset functions, and all logical functions. However, its real power comes from its ability to handle Boolean functions. On any addressable bit, the processor can perform such functions as Set, Clear, Complement, Jump If Set or Not Set, and Jump If Set Then Clear. It can also perform logical functions with two bits and place the result in the carry flag.

The MCS-51 can use its 32 I/O lines as 32 individual bits or as four 8-bit parallel ports. It can serve five interrupts: two external, two from the counters, and one from the serial I/O port. The chip includes two 16-bit counters, which can operate in three different modes, and a serial I/O port, which can operate in full duplex mode.

### 18.13 Analog Signal Processor: the Intel 2920

This is a single-chip microcomputer specially designed to process analog signals. Most microprocessors are not able to process high speed analog signals because of their slow response. The 2920 is designed with special architecture and an instruction set suitable for handling high-speed signal processing. In addition to the MPU and memory, the chip includes all such necessary devices as A/D and D/A converters, a multiplexer to handle four different inputs, a sample and hold circuit, and a demultiplexer.

The 2920 is widely used for acquiring and processing analog signals. Typical applications are in such areas as telecommunications, signal processing, guidance and control, speech processing, and industrial automation.

## 16- AND 32-BIT MICROPROCESSORS

## 18.2

The 16-bit microprocessor families are designed primarily to compete in the territory of minicomputers, and are oriented towards high-level languages. Their applications may overlap the high end of 8-bit microprocessor applications and may compete with mainframe computers. They have powerful instruction sets and are capable of addressing megabytes of memory. Examples of widely used 16-bit microprocessors include Intel 8086/

8088, Zilog Z8001/8002, Digital Equipment LSI-11, Motorola 68000, and National Semiconductor NS16000. Apart from design concepts and instruction sets, a critical factor that decides the capability of the microprocessor is the number of pins available. One trend is to stay within the 40-pin package size and take advantage of existing production and testing facilities. The 40-pin package either limits the size of the memory that can be addressed or necessitates multiplexing of several functions. Intel, Zilog (for Z8002), and Digital Equipment have stayed with the 40-pin package. Another trend is to go beyond the 40-pin limit, either to a 48-pin size or to a 64-pin size. National Semiconductor (for NS16000) and Zilog (for Z8001) have chosen the 48-pin size package. Motorola and Texas Instruments have selected the 64-pin size package. The primary objectives of these 16-bit microprocessors can be summarized as follows:

1. Increase memory-addressing capacity.
2. Increase execution speed.
3. Provide a powerful instruction set.
4. Facilitate programming in high-level languages.

These objectives can be met by using various design concepts. To illustrate differences in design philosophies, the next two sections will briefly describe three 16-bit microprocessors: Intel 8086/8088, Zilog Z8001/Z8002, and Motorola MC68000.

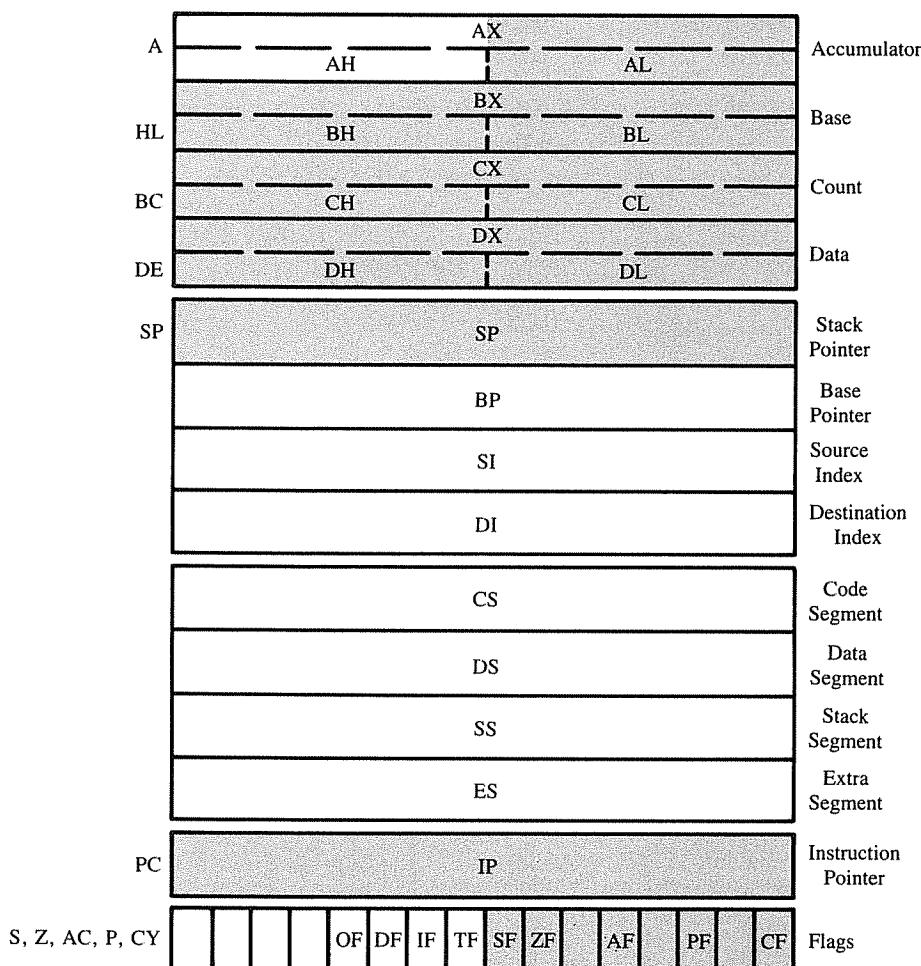
### 18.21 Intel 8086/8088

This is a 16-bit microprocessor housed in a 40-pin package and capable of addressing 1 megabyte of memory. Various versions of this chip can operate with clock frequencies from 4 MHz to 8 MHz. Figure 18.3 shows internal registers; the shaded portions of the figure are identical with the 8085/8080A registers. This microprocessor includes fourteen 16-bit registers, of which the top four registers (AX, BX, CX, and DX) are used as general purpose accumulators. These four can also be used as 8-bit registers. The next four 16-bit registers are used primarily as memory pointers and index registers; they hold part of a 20-bit memory address, as described under **Memory Segmentation**. They can also be used as general-purpose registers. The next four 16-bit registers are used to specify a segment of the 1-megabyte memory. The last two registers are similar to the program counter and flag register in the 80805/8080A, but have four additional flags.

The 8088 is functionally similar to the 8086, except that it has an 8-bit data bus. Its internal architecture and instruction set are essentially identical with those of the 8086. The only difference is that a 16-bit data word must be transferred in two segments in the 8088. The 8088 can be viewed as an 8-bit microprocessor with the execution power of a 16-bit microprocessor. The next few paragraphs describe the features of the 8086 architecture that meet the objectives described.

#### MEMORY SEGMENTATION

To increase the memory addressing capacity, the concept of memory segmentation is employed in this device. This concept involves combining the addresses from two 16-bit

**FIGURE 18.3**

The 8086 Programming Registers

SOURCE: Reprinted by permission of Intel Corporation, copyright 1981.

registers to form a 20-bit effective address. A segment register provides a base address, and another register supplies an offset address. For example, to fetch an instruction from the 256th location on page 0, the address can be formed as follows:

1. Define the memory segment by loading  $0000_H$  into the Code Segment Register.
2. The Instruction Pointer should hold  $00FF_H$  as the offset.

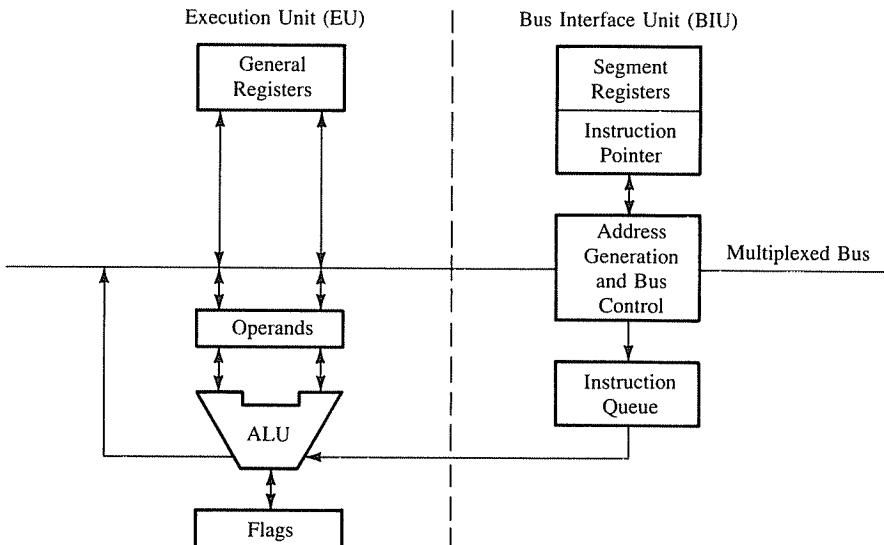
3. The processor shifts the address in the Code Segment Register by four bits to the left and adds the content of the Instruction Pointer to form the 20-bit address:

Code Segment:	0 0 0 0
Instruction Pointer:	0 0 F F
Effective Address:	0 0 0 F F

The same address can be obtained by redefining the address in the Code Segment Register and using an appropriate count from the Instruction Pointer. By having four segment registers, the 1-megabyte memory space can be conveniently divided into different sections such as program, data, and stack.

### SIMULTANEOUS PROCESSING

The 8086 includes two processors called Execution Unit and Bus Interface Unit, as shown in Figure 18.4. They speed up execution by employing the concept of dividing work between two processors and processing it simultaneously. The execution process in the 8086 is similar to that of the Z80: fetch, decode, and execute. However, in the 8-bit processor, the buses are idle during the execution cycle. This idle time is avoided in the 8086 by assigning execution to the Execution Unit and fetching to the Bus Interface Unit. When an instruction is being executed, the Bus Interface Unit fetches instructions and places them on the queue, as shown in Figure 18.4; this is also known as *pipelining* the instructions.



**FIGURE 18.4**  
Execution and Bus Interface Units (EU and BIU) of the 8086  
SOURCE: Reprinted by permission of Intel Corporation, copyright 1981.

## COPROCESSING

In addition to the 8086, Intel has designed a series of such special function devices as the 8089 (I/O Processor) and the 8087 (Numeric Processor). These processors are compatible with the 8086 in the master-slave relationship. They are designed with additional instructions and can be assigned dedicated functions to increase the overall execution speed in large systems.

## INSTRUCTION SET

The 8086 has a large instruction set, consisting of 135 basic instructions, which can operate on individual bits, bytes, 16-bit words, and 32-bit double words. The set includes such instructions as multiply, divide, and bit and string manipulation.

## MODULAR PROGRAMMING

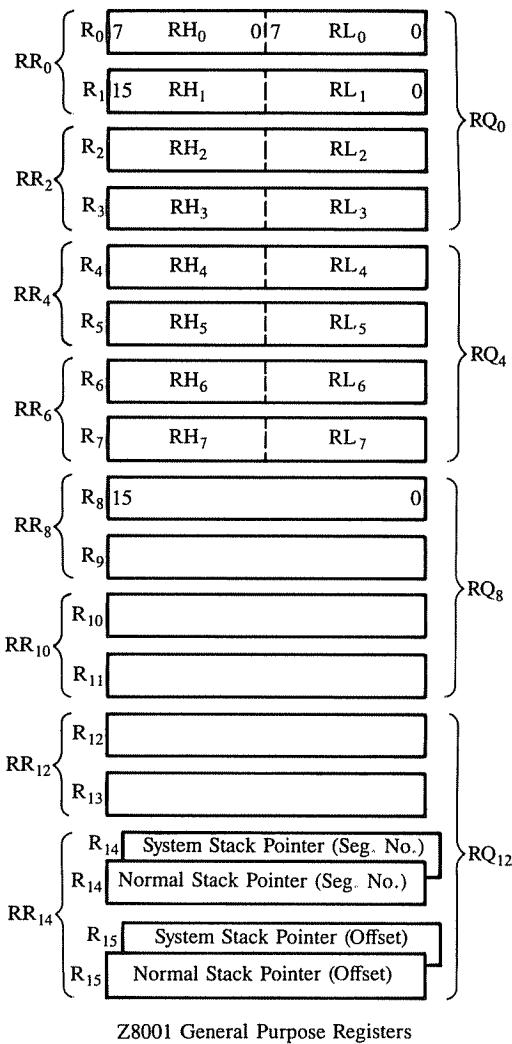
In addition to the powerful instruction set, the chip design is oriented toward modular programming, which is highly desirable for high-level languages. The memory-segmentation concept facilitates programming of independent modules that can communicate with each other as well as share common data.

### 18.22 Zilog Z8000

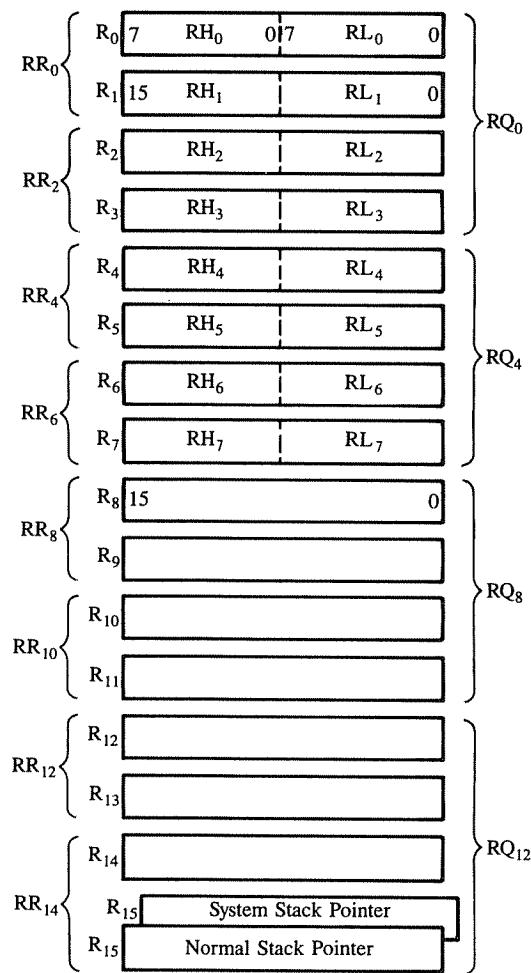
The Zilog Z8000 is a 16-bit microprocessor with two versions—Z8001 and Z8002. The Z8002 is a 40-pin device capable of addressing 64K non-segmented memory. The Z8001 is a 48-pin device, almost identical to the Z8002, capable of directly addressing eight megabytes of memory. Like the 8086, it can also use the concept of segmented memory, but unlike the 8086, it requires an additional device called the Memory Management Unit, and the memory addressing can be extended to 48 megabytes.

The overall architectural philosophy of the Z8000 is similar to that of the 8086. The Z8000 is a register-oriented microprocessor; the Z8002 version has twenty-one 16-bit registers, of which 16 are general-purpose registers (Figure 18.5). Any of these general-purpose registers, with the exception of the register R0, can be used as an accumulator, index register, memory pointer, or stack pointer. This is unlike the 8086, in which most registers have designated functions. In addition to 16 general-purpose registers, the Z8000 includes five registers: program counter, flag register, status pointer, instruction register, and refresh counter. To speed up the execution, it uses the prefetched pipeline technique and has a powerful interrupt structure. One of the unique features of the Z8000 is that it provides a refresh counter to refresh dynamic memory. It can operate in either the system or normal mode. The system mode permits privileged operations, thereby facilitating multi-user systems.

The Z8000 has a very powerful instruction set of 110 instruction types; it has seven addressing modes and can operate on a bit, byte, 16-bit word, 32-bit long word, and 64-bit quad word. The instruction set includes instructions such as multiply/divide, block transfer, and string manipulations.



Z8001 General Purpose Registers



Z8002 General Purpose Registers

FIGURE 18.5

Z8001/Z8002 General-Purpose Registers

SOURCE: Courtesy of Zilog, Inc.

### 18.23 Motorola MC68000

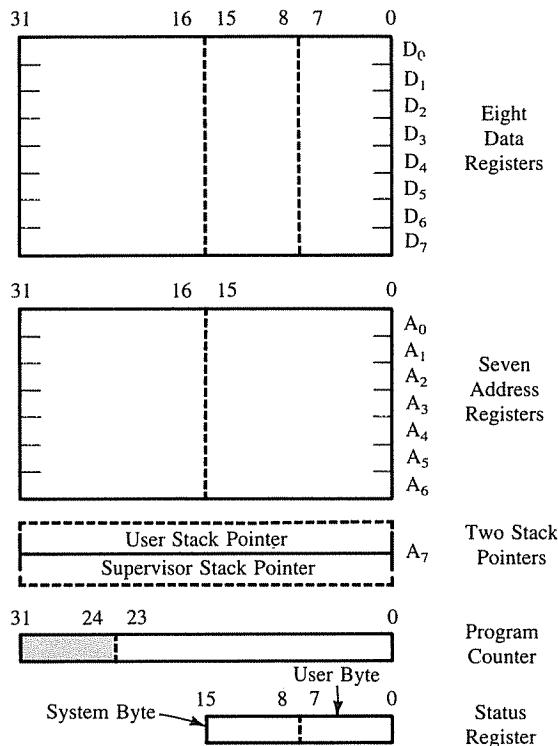
This is a 16-bit microprocessor with a 32-bit internal architecture housed in a 64-pin package. It is capable of addressing 16 megabytes of memory, and the clock frequency ranges from 4 MHz to 10 MHz for different versions of the chip.

Figure 18.6 shows the internal architecture of the device. It includes seventeen 32-bit, general-purpose registers, a 32-bit program counter, and a 16-bit status register.

**FIGURE 18.6**

Programming Registers of the 68000

SOURCE: Courtesy of Motorola, Inc.



The general-purpose registers are divided into three groups: eight data registers, seven address registers, and two stack pointers. The contents of the data registers can be accessed as bytes, 16-bit words, or 32-bit words, and the contents of the address registers can be accessed as 16-bit or 32-bit addresses. The 68000 can operate in two different modes: the user mode and the supervisor mode. The supervisor mode is designed primarily for operating systems; in this mode, some privileged system control instructions can be used. Some of its other features can be described as follows.

### NONSEGMENTED MEMORY

To increase the memory addressing capacity, Motorola has increased the number of pins in its package. The chip is designed with 23 separate lines to address eight megawords (16 megabytes). Similarly, its program counter is 32 bits long; however, only the low-order 24 bits are necessary to address the entire memory map.

### INSTRUCTION SET

The 68000 has one of the most powerful and simple instruction sets. It includes 56 basic instructions and can operate on five different types of data: bit, byte, BCD, 16-bit word, and 32-bit word. It has only memory-mapped I/O but includes 14 memory addressing modes. To cite one example of its powerful set, its MOV instruction can transfer data from

any source to any destination. It includes such instructions as Multiply and Divide and special instructions to deal with numbers longer than 32 bits. Its orientation towards high-level languages comes primarily from its instruction set.

### ASYNCHRONOUS AND SYNCHRONOUS CONTROL LINES

The 68000 has a special way of handling slow and fast peripherals. It has two sets of control signals, called asynchronous and synchronous signals. Communication with asynchronous peripherals is handled through the control lines called Upper Data Strobe (UDS), Lower Data Strobe (LDS), and Data Acknowledge (DACK). The DACK signal is similar to a handshake line; until the signal DACK is received, the bus cycle is not terminated. The 6800 family offers some synchronous peripherals, and communication with these peripherals is handled through the control signals called Valid Peripheral Address (VPA), Valid Memory Address (VMA), and Enable(E).

### 18.24 Intel 80186 and 80286

The Intel 80186 and -286 are 16-bit microprocessors, extended versions of the 8086. One of the critical barriers among the Intel's earlier microprocessors was the 40-pin package. Once that barrier was broken, it became easier to address large memory. These recent microprocessors are housed in 68-pin packages. These microprocessors, generally, use the concepts of prefetched pipeline structure, parallel processing, and memory management.

The 80186 is an improved version of the 8086; it is available in two speeds: 6 MHz and 8 MHz. It is an integrated device designed to reduce the chip count, rather than to increase the memory addressing capacity. It has multiplexed address and the data buses, and the additional lines of the bigger package are used to include devices such as a clock generator, interrupt controller, timers, DMA controller, and a chip-select unit.

The 80286 is also a 16-bit microprocessor, an improved version of the 8086 but with different architectural philosophy. It has eliminated the multiplexing of the buses; it has a linear address bus with 24 address lines that can address 16 megabytes of memory directly. It can also support a memory management unit, and through the memory management unit it can address 1 gigabyte of memory, also known as virtual memory. The processor includes various built-in mechanisms that can protect system software from user programs, protect users' programs, and restrict access to some regions of memory. The 80286 is designed for a multi-user system in an environment similar to that of minicomputers and mainframe computers; its architectural philosophy is closer to the Intel's 80386 32-bit microprocessor, which is described in the next section.

### 18.25 32-bit Microprocessors

At the high end of the microprocessor range, we now have 32-bit microprocessors available; examples include such microprocessors as Intel 80386, Zilog Z80,000, National Semiconductor NS32032, and Motorola MC68020. We are interested not in discussing the details of these microprocessors, but in exploring trends in the microprocessor technology.

These microprocessors are not merely more of the same, except bigger and faster; they offer some unique features not available in 16-bit microprocessors. The applications and the environments in which they operate are far different from those of the 8-bit microprocessor and the 16-bit microprocessors. It appears that two trends are evolving: One is multi-user, multi-tasking, time-sharing environments, and the other is distributed processing, interconnected with networks. As soon as we move away from the single-user system, the demands on these microprocessors change dramatically; the environment is more like that of minicomputers or mainframe computers. These microprocessors should not be viewed as programmable logic devices.

In a single-user system, the user has an unlimited access to all aspects of the system. The user need not be concerned with sharing the time or the resources of the system, but can schedule various tasks according to his or her convenience. The user has access to the operating system, can tamper with the system to include some personal conveniences, or in the process can lock up the system. However, the multi-user system cannot afford to provide the luxuries of unlimited access to all users. Some of the requirements of the multi-user system are as follows:

1. Higher speed of execution.
2. Ability to handle different types of tasks efficiently.
3. Large memory space that can be shared by multiple users.
4. Appropriate memory allocations and the management of memory access.
5. Data security and data access.
6. Limited and selected access to part of the system.
7. Resource (printer, hard disk, etc.) sharing and management.

Some of these requirements must be managed by a multi-user operating system, and some should be facilitated by the architectural design of the microprocessors. The 32-bit microprocessors are designed to work in this type of environment. Some of the important features of Intel 80386 and Zilog Z80,000 are described in this section as representative samples of 32-bit microprocessor technology.

The 80386 is a 32-bit microprocessor with a nonmultiplexed 32-bit address bus and can operate at 16 MHz clock. It is capable of addressing 4 gigabytes of physical memory, and through its memory management unit, it can address 64 terabytes ( $2^{46}$ ) of memory. The 80386 has 32-bit registers, and it is upward software compatible with the 8086. The execution of instructions is highly pipelined, and it is designed to operate in a multi-user and multitasking environment. It has the protection mechanism that is necessary for this type of environment.

The Z80,000 is also a 32-bit microprocessor with architecture similar to that of the 80386. It can directly address 4 gigabytes of memory, and it can extend the memory addressing capacity similar to the 80386 by using the built-in memory management unit. It is designed to operate in two primary modes—system and normal—supported by separate stacks. The normal mode is for user programs, and the system mode is used for some of the critical functions of an operating system, such as protecting the operating system from user access.

In summary, these 32-bit microprocessors are oriented toward high-level languages. They can address a large memory space, execute instructions with high speed, and perform arithmetic operations with high precision. These microprocessors suggest a trend toward replacing software functions with hardware. They are designed to perform the functions normally found in mainframe computers.

### 18.26 Contemporary 8-Bit Microprocessors and Technology Trends

The Intel 8008, which was later superseded by the Intel 8080A, was the first 8-bit microprocessor. Just about the same time (1974), Motorola brought out the MC6800 as an improvement over the first 8008, but with substantially different architecture. Within a few years, Zilog designed the Z80, and Intel came up with the 8085 as an improvement over the 8080A. Both are upward machine-language compatible with the 8080A. Other popular microprocessors are those of the MOS Technology (now part of Rockwell) MCS6500 series, which was designed as an improvement over the Motorola 6800; however, they are neither hardware nor software compatible. In recent years, Motorola came up with the MC6809, a vastly improved version of the MC6800. These contemporary 8-bit microprocessors were discussed briefly in Chapter 2. Now the question is: What is the role of these general-purpose 8-bit microprocessors in the fast changing microprocessor technology?

Along with the development of general-purpose 8-bit microprocessors, the single-chip microcontrollers began to assume a major role in the area of dedicated functions. Examination of the three examples discussed in Section 18.1 shows that the single-chip microcomputer plays a vital role in control applications and is an important segment of microprocessor technology. These devices are designed for special-purpose applications, and the circuitry on the chips varies according to the objectives. Applications range from bit set/reset functions to processing high-speed analog signals.

At the other end of the application spectrum, 16-bit microprocessors have begun to dominate the microcomputer industry. The 8086 and the Z8002 have employed several new architectural concepts, such as memory segmentation, parallel processing, queueing, and coprocessing. In addition, the Motorola MC68000 and the Zilog Z8001 broke the barrier of the 40-pin package. These processors are oriented toward high-level languages and will perform some functions of minicomputers and mainframe computers. Recent microprocessors, such as Intel 80286 and -386, and Zilog Z80,000 have begun to accept the challenges of multi-user and multi-tasking environments.

Now the question is: will these general-purpose 8-bit microprocessors disappear because of the competition from highly sophisticated single-chip microcontrollers at one end and high speed and powerful 16-bit microprocessors at the other end? We think not. Each group has carved out its own share of applications. This is similar to the automobile industry; there is room for subcompacts as well as luxury sports cars.

The 16-bit microprocessors are too powerful to perform the functions of general purpose 8-bit microprocessors; therefore, they are less likely to replace 8-bit processors. Competition for the general-purpose 8-bit microprocessors will come from the other direc-

tion: the single-chip microcomputers. However, the single-chip microcomputers do not lend themselves as suitable learning vehicles for basic concepts, and the 16-bit microprocessors are too complex and cumbersome for instructional purposes.

In 8-bit general-purpose microprocessors, the trend seems to be toward integrated devices that reduce the chip count; these devices are known as integrated super 8-bit MPUs. Two examples of such devices are Hitachi HD64180 and Zilog Z280; these are described briefly here to indicate the trend.

**Hitachi HD64180** This is an 8-bit, high-integration CMOS microprocessor in a 64-pin package, designed for applications with low power consumption, and it can operate with a 6 MHz clock. It includes a clock generator, an interrupt controller, and a memory management unit (MMU) as support devices for the microprocessor (Figure 18.7). It has 19 address lines that can address 512K bytes of physical memory, and the MMU translates internal 64K logical addressing into appropriate physical addressing. The interrupt controller is capable of handling four external and eight internal interrupting sources.

Figure 18.7 also shows that the HD64180 includes four I/O related devices: DMA controller (DMAC—two channels), asynchronous serial communication interface (ASCI—two channels), clocked serial I/O port (CSI/O), and programmable reload timer (PRT—two channels). The DMAC has two channels that support high-speed data transfer of 64K bytes per channel anywhere in the physical space of 512K bytes of memory. The ASCI has two separate channels for full-duplex communication, and the CSI/O provides a half-duplex communication; it is used primarily for simple high-speed connection between microcomputers. Similarly, the timer has two channels with 16-bit counters, and one of the channels can be used for waveform generation.

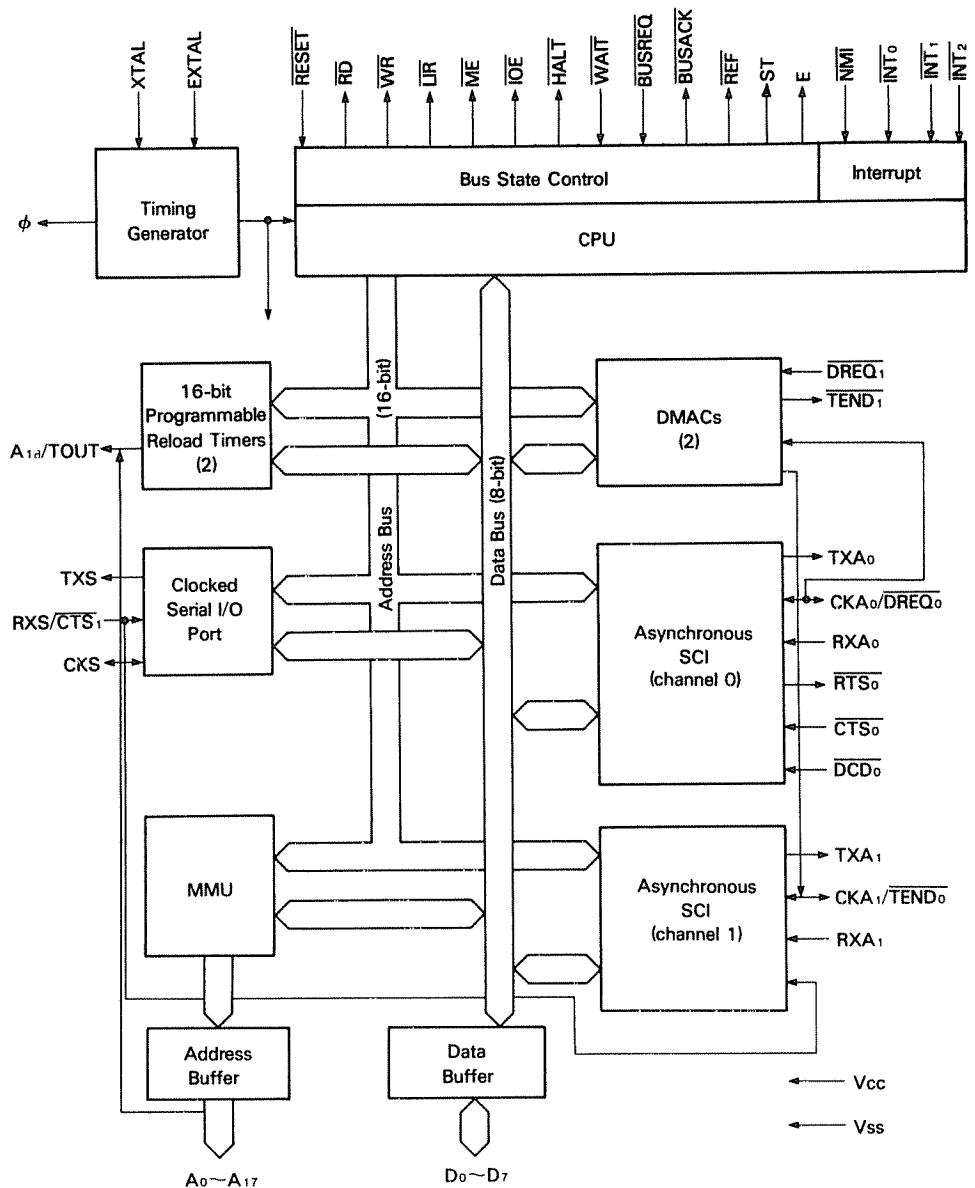
The instruction set of the HD64180 is upward compatible with the Z80 instruction set. The HD64180 has seven additional instructions, including 8-bit Multiply and Sleep. The Sleep instruction reduces the power consumption to 19 mW. One of the powerful features of this device is that the Opcode Fetch cycle of an instruction consists of three T-states versus four T-states in the Z80, resulting in faster program execution.

**Zilog Z280** This is also an 8-bit, high-integration CMOS microprocessor in a 68-pin package, and it can operate with a 20\* MHz clock. It includes a clock generator, refresh address generator, 256 bytes of on-chip memory, and a memory management unit (MMU) as support devices for the microprocessor operations (Figure 18.8). The MMU enables the microprocessor to address 16M bytes of memory, and the refresh address generator provides a 10-bit address that is used in refreshing dynamic memory. The on-chip memory allows programs to run significantly faster by reducing the number of external bus accesses.

Figure 18.8 also shows that the Z280 includes four I/O related devices: DMA (four channels), Universal Asynchronous Receiver/Transmitter (UART), and Counter/Timer (three channels). The DMA has four channels that can transfer data between any two ports (source and destination), including memory-to-I/O, I/O-to-memory, memory-to-memory,

---

\*Note: The preliminary specifications indicate that the clock frequencies can go as high as 50 MHz.



**FIGURE 18.7**

Block Diagram: HD64180

SOURCE: Courtesy of Hitachi America, Ltd., Semiconductor & I.C. Division.

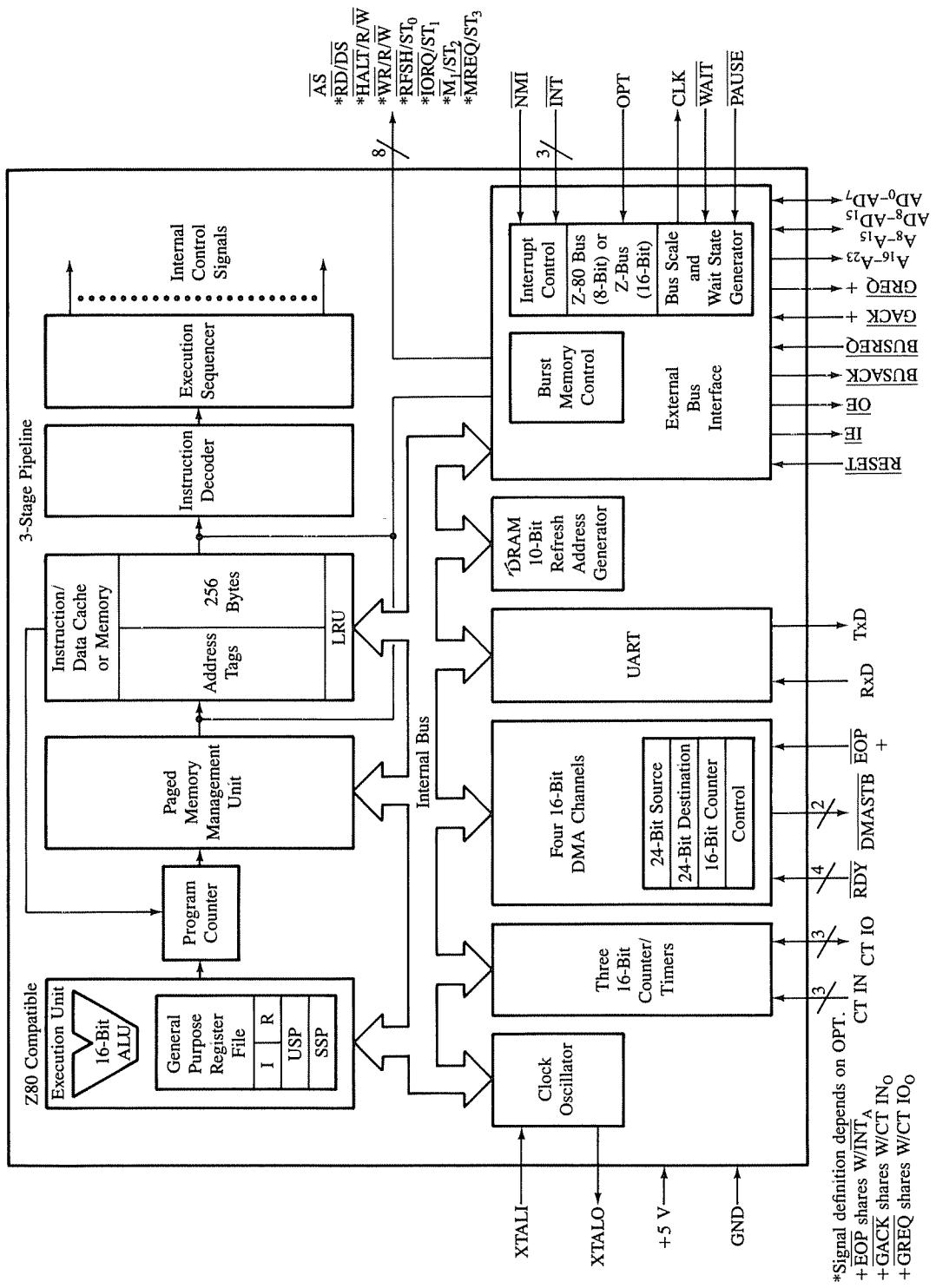


FIGURE 18.8  
Block Diagram: Z280  
SOURCE: Courtesy of Zilog, Inc.

and I/O-to-I/O. The UART is capable of handling any full-duplex asynchronous data communication. Similarly, the Z280 has three channels of Timer/Counter with 16-bit time constant; these can be used for event counting, interrupt and interval timing, and general clock generation.

The Z280 can operate in either user or system mode, and each has separate stacks. System mode is intended for the functions of an operating system, and user mode is intended for application programs. Thus, the sensitive and critical functions of the operating system are protected from the user interference. The instruction set of the Z280 retains compatibility with that of the Z80 and includes additional instructions such as 8- and 16-bit signed and unsigned multiply and divide. It has a powerful interrupt structure which has four modes of operation; the first three modes are similar to those of the Z80.

After examining these two devices, it appears that the Z80-type microprocessors have begun to reassert their presence in the form of integrated devices in industrial and computer applications.

## 18.3 BUS INTERFACE STANDARDS

---

The microcomputer is a bus-oriented system whereby subsystems or peripherals are interconnected through the bus architecture. The design approach should be such that systems are modular, expandable, and multipurpose. For example, a microcomputer with 32K memory should be expandable to 64K memory without any design changes as the user's needs change. The user should be able to select a peripheral from any manufacturer and plug it into the system. Similarly, the user should be able to print out programs as well as collect data from various instruments. To design microcomputers with such features, a common understanding of equipment specifications among manufacturers is needed; this is known as **defining standards**. In the field of electronics, these standards are generally defined by professional organizations such as IEEE (Institute for Electrical and Electronics Engineers) and EIA (Electronic Industries Association); sometimes the standards are forced upon the industry either by a dominant manufacturer or by common practice. The need for expandability and modularity gave rise to various bus standards, as listed in Table 18.1. The bus S-100, the Standard Bus, IBM PC Bus, the Multibus, and the GPIB are described here; others were described in Chapter 15.

### 18.31 S-100 (IEEE 696), Standard, IBM PC Bus

The primary force behind the development of these buses is their expandability. This type of bus allows the user to plug in additional peripherals (including memory) without any design changes and also facilitates troubleshooting. The bus design is based on the concept of the "mother board," a printed circuit board with parallel foil strips. Several edge connectors, connected to the foil strips, are included in a system, and some edge connectors are left empty for the user.

**TABLE 18.1**  
Bus Standards

Buses	I/O Mode	Applications/Description
1. S-100 (IEEE-696)	Parallel	To interconnect various components within the microcomputer. It has 100 signals.
2. STD Bus	Parallel	This is a bus competing with S-100. It has 56 signals.
3. IBM PC Bus	Parallel	It has 62 signal lines and is used to connect peripherals.
4. Multibus (IEEE-796)	Parallel	Interface between modules with the master/slave type communication. It has 86 signals.
5. GPIB (IEEE-488)	Parallel	Interface between the microcomputer and measurement equipment such as a voltmeter. It has 24 signals.
6. RS-232C	Serial	Interface between the microcomputer and serial peripherals such as a terminal and a printer (see Chapter 15).
7. RS-422 and RS-423	Serial	High-speed serial communication for distances longer than 20 meters.

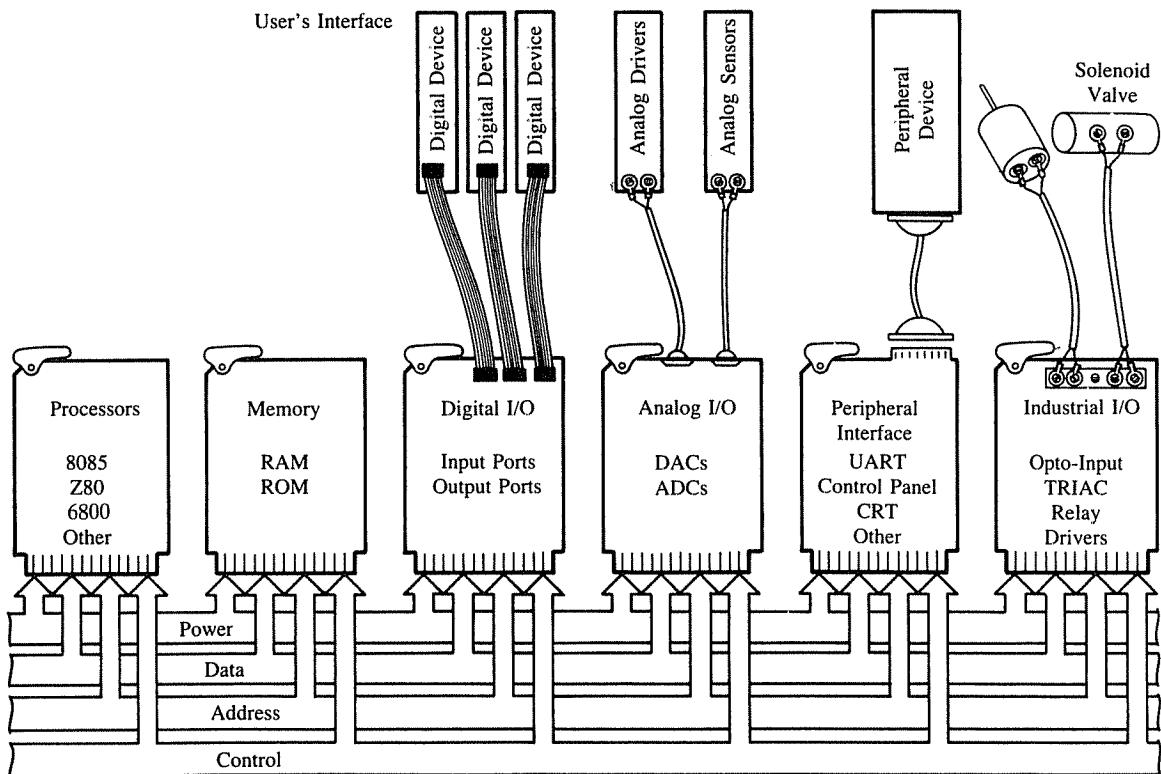
### BUS INTERFACE

The bus signals are divided into four groups, as shown in Figure 18.9: power, data, address, and control. The S-100 bus has 100 signals, the STD bus has 56 signals, and the IBM PC bus has 62 signals.

The S-100 bus was originally developed by MITS and IMASAI in 1975, even before the existence of a 16-bit microprocessor. The bus quickly became popular with hobbyists and it became the de-facto industry standard. Initially, several lines were undefined, causing contradictory uses by manufacturers. Eventually, IEEE adopted the S-100, with some modifications, as the IEEE-696 standard. This bus has 24 address lines, 16 data lines, 11 interrupts, and provision for multiprocessing.

The STD bus was recently (1981) developed by Pro-Log Corporation and MOS-TEK, as a simple bus structure for 8-bit microprocessors. It is a 56-pin bus with eight data lines, 16 address lines, 22 control lines, and ten power lines. It is a better-defined bus, an improvement over the original S-100 bus. However, the S-100 is so widely used that the STD bus may have difficulty demonstrating its superiority to industry.

The IBM PC bus is a set of 62 signal lines, specifically designed for IBM personal computers; the bus is also known as I/O channel. It is used to connect peripherals such as floppy disk controllers, serial and parallel I/O devices, and video display controllers to the system board. In addition, it can also be used to expand memory capacity and connect special-purpose I/O devices. The bus includes 20 address lines, eight data lines, seven power and frequency related lines ( $\pm 12$  V,  $\pm 5$  V, GND, system clock, and high frequency clock), and one unspecified line; the remaining 26 lines can be classified as control and status lines. The control and status group consists of Read/Write signals, interrupt request signals, DMA request and acknowledge signals, and status (such as address latch enable, I/O check, Ready) signals.

**FIGURE 18.9**

Bus Interface

SOURCE: Courtesy of Pro-Log Corporation, Monterey, Calif.

The IBM PC design is divided primarily into two segments: system board and various I/O channels with these 62 lines. Peripheral boards are simply plugged into these channels. For example, a floppy disk controller is specifically assigned three control signals: IRQ6 (Interrupt Request 6), DRQ2 (DMA Request 2), and DACK2 (DMA Acknowledge 2), and the disk controller board is designed using these lines. The usage and popularity of this bus is dependent on the IBM PC and its compatible systems; the recent models of IBM PCs (Personal System/2) do not use this bus.

### 18.32 Multibus (IEEE 796)

The multibus is an 86-line (optional 60-line) bus developed by Intel Corporation to connect independent modules (or microcomputers) of various microprocessors to share resources. Each board can operate independently using its own memory and I/Os for dedicated functions. They can access a system's resources such as a hard disk or printer through the multibus.

The bus includes 20 address lines, 16 bidirectional data lines, numerous control lines, and multiple power and ground lines; the bus control can be managed by IC devices such as the Intel 8218 or 8288. A multibus system can have many masters; therefore, control logic is required for arbitration and data flow when multiple requests are made to access the bus. Simultaneous requests to access the bus are handled through either serial or parallel techniques by control signals.

### 18.33 GPIB Interface Bus (IEEE 488)

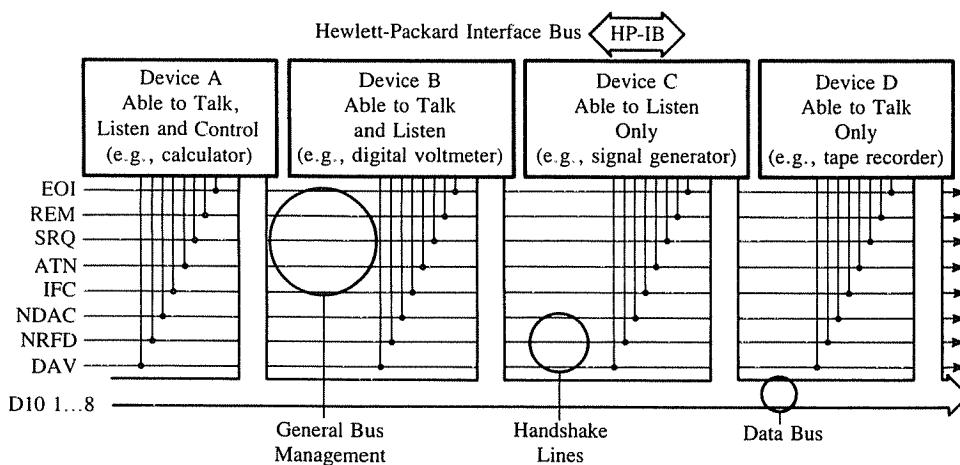
This bus was developed to facilitate interfacing of programmable instruments (such as printers, digital voltmeters, and digital tape recorders) with computers. Initially, the bus was developed by Hewlett-Packard; later, it was accepted as the IEEE 488 Interface Standard. The bus standard is also known as the General Purpose Interface Bus (GPIB) or the Hewlett-Packard Interface Bus (HP-IB).

Some of the features of this bus are the following:

1. Data transfer among the interconnected devices is digital.
2. Fifteen devices may be connected to one continuous bus.
3. Total transmission path is limited to twenty meters or two meters per device.
4. Data rate on any signal line is limited to 1Mbyte/sec.

#### BUS SIGNALS

The bus has 24 signals: eight bidirectional data lines, five general bus management lines, three handshake lines, and eight grounds. Figure 18.10 shows four types of devices that can be connected to the bus. These devices are classified as follows:



**FIGURE 18.10**  
GPIB (IEEE 488) Bus Interface

SOURCE: Hewlett-Packard Company, *Tutorial Descriptions of the Hewlett-Packard Interface Bus*. (Palo Alto, CA: Author, 1980), p. 8. Reproduced with permission.

1. *Listener* is a device capable of receiving data when addressed; 14 devices can listen at a time. Examples include printers and display devices.
2. *Talker* is a device capable of transmitting data when addressed; only one device can be active at a time. Examples include tape readers and voltmeters.
3. *Listener/Talker* is a device that can receive as well as transmit data over the interface. A programmable digital voltmeter (DVM) is a listener/talker device.
4. *Controller* is a device that controls signals and specifies which device can talk and which device can listen. A microcomputer with an appropriate I/O card can serve as a controller.

## SUMMARY

---

In this chapter, various microprocessors—from 8-bit to 32-bit—and single-chip microcontrollers were discussed, compared, and contrasted in terms of their characteristics and applications. Future trends in microprocessor technology were suggested.

Single-chip microcomputers (also known as microcontrollers) and their various applications were discussed in Section 18.1. These microcontrollers are specially designed for specific applications, and their characteristics differ according to their areas of applications.

Microprocessors with 16- and 32-bit words were discussed in Sections 18.2. These are designed to facilitate the use of high-level languages, and are expected to compete with functions of minicomputers and mainframe computers. New architectural concepts such as memory segmentation, parallel processing, and queueing were employed in designing some of these processors, and some are designed with pin package larger than 40. In 8-bit general-purpose microprocessors, Z80-type microprocessors have begun to appear in the form of integrated devices that include a microprocessor, memory management unit, DMA controllers, UARTS, and timers on one chip.

The chapter was concluded with the discussion of various bus standards.

# References

- Analog Devices Inc. *Data Acquisition Components and Subsystems*. Norwood, MA: Author, 1980.
- Bates, Paul. *Practical Digital and Data Communications with LSI Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1987.
- Brey, Barry. *8086/8088 Microprocessor Architecture, Programming, and Interfacing*. Columbus, OH: Merrill Publishing Co., 1987.
- CAMI Research Inc. *The Micro-Trainer Manual*. Arlington, MA: Author, 1983.
- Coffron, James. *Practical Hardware Details for 8080, 8085, Z80, and 6800 Microprocessor Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1981.
- Coffron, James. *Z80 Applications*. Berkeley, CA: Sybex Inc., 1983.
- Electronic Industries Association. *Interface Between Data Terminal Equipment and Data Communication Equipment Employing Serial Binary Data Interchange: RS-232-C Standard*. Washington, DC: Author, 1969. Reaffirmed in June 1981.
- Fairchild. *TTL Data Book*. Mountainview, CA: Author, 1978.
- Floyd, Thomas. *Digital Fundamentals*, 3rd ed. Columbus, OH: Merrill Publishing Co., 1986.
- Gaonkar, Ramesh. "Data Conversion, Chapter 13." In *Integrated Circuits Applications Handbook*, Arthur Seidman, Ed. New York: Wiley, 1983.
- Gaonkar, Ramesh. *Microprocessor Architecture, Programming, and Applications with the 8085/8080A*. Columbus, OH: Merrill Publishing Co., 1984.
- Goldsbrough, Paul. *Microcomputer Interfacing with the 8255 PPI Chip*. Indianapolis, IN: Howard W. Sams, 1979.
- Hall, Douglas. *Microprocessors and Interfacing*. New York: McGraw-Hill, 1986.
- Hewlett-Packard Company. *Tutorial Description of the Hewlett-Packard Interface Bus*. Palo Alto: Author, 1980.
- Hitachi America Ltd. *HD64180 8-Bit High Integration CMOS Microprocessor User's Manual*. San Jose, CA: Author, 1985.
- Hogan, Thom. *CP/M User's Guide*. Berkeley, CA: Osborne/McGraw-Hill, 1981.
- Intel Corporation. *Component Data Catalog*. Santa Clara, CA: Author, 1982.
- Intel Corporation. *iAPX 86,88 User's Manual*. Santa Clara, CA: Author, 1981.
- Intel Corporation. *Memory Components Handbook*. Santa Clara, CA: Author, 1984.
- Intel Corporation. *Microcontroller Handbook*. Santa Clara, CA: Author, 1984.
- Intel Corporation. *Peripheral Design Handbook*. Santa Clara, CA: Author, 1981.
- Intel Corporation. *2920 Analog Signal Processor Design Handbook*. Santa Clara, CA: Author, 1980.
- Khambata, Adi. *Introduction to the Z80 Microcomputer*. New York: Wiley, 1982.
- Leibson Steve. "The Input/Output Primer, Part 4: The BCD and Serial Interfaces." *Byte* (May 1982): 202–220.

- Mostek. *BYTEWYDE Memory Data Book*. Carrollton, TX: Author, 1980.
- Mostek. *Memory Data Book and Designer's Guide*. Carrollton, TX: Author, 1980.
- Mostek. *Z80 Microcomputer Data Book*. Carrollton, TX: Author, 1981.
- Multitech Electronics, Inc. *Micro-Professor User's Manual*. San Jose, CA: Author, 1981.
- National Semiconductor Inc. *Linear Data Handbook*. Santa Clara, CA: Author, 1982.
- Norton, Peter. *The Peter Norton Programmer's Guide To The IBM-PC*. Bellevue, Washington, Microsoft Press, 1985.
- Osborne, Adam, and Kane, Gerry. *4- & 8-bit Microprocessor Handbook*. Berkeley, CA: Osborne/McGraw-Hill, 1981.
- Pro-Log Corporation. *Series 7000 STD Bus Technical Manual and Product Catalog*. Monterey, CA: Author, 1981.
- Rony, Peter. "Interfacing Fundamentals: Bidirectional I/O Using Two Semaphores." *Computer Design* (April 1981): 184-188.
- Rony, Peter. "Interfacing Fundamentals: A Comparison of Block Diagrams for I/O Techniques." *Computer Design* (February 1982): 175-177.
- Rony, Peter. "Interfacing Fundamentals: Conditional I/O Using a Semaphore." *Computer Design* (April 1980): 166-167.
- Rony, Peter. "Interfacing Fundamentals: Conditional I/O Using Two Microcomputers." *Computer Design* (August 1980): 136-138.
- Rony, Peter. "Interfacing Fundamentals: 2-Wire Handshake Using Two Microcomputers." *Computer Design* (June 1981): 156-160.
- Terrell, David. *Microprocessor Technology*. Reston, VA: Reston Publishing Company, 1982.
- Texas Instruments Inc. *The TTL Data Book for Design Engineers*, 2nd ed. Dallas TX: Author, 1976.
- Titus, Christopher, et al. *16-Bit Microprocessors*. Indianapolis, IN: Howard W. Sams, 1981.
- Toomey, Paul. *The AD7574 Analog to Microprocessor Interface, Application Note*. Norwood, MA: Analog Devices, 1982.
- Uffenback, John. *Microcomputers and Microprocessors: The 8080, 8085, and Z80 Programming, Interfacing, and Troubleshooting*. Englewood Cliffs, NJ: Prentice-Hall, 1985.
- Zaks, Rodney. *Programming the Z80*, 3rd Rev. Ed. Berkeley, CA: Sybex Inc., 1982.
- Zilog Inc. *Components Data Book*. Campbell, CA: Author, 1982, 1983, 1985.
- Zilog Inc. *Microprocessor Applications Reference Book*, vol. 1. Cupertino, CA: Author, 1981.
- Zilog Inc. *Microprocessor Applications Reference Book*, vol. 2. Campbell, CA: Author, 1983.
- Zilog Inc. *Z80-Assembly Language Programming Manual*. Campbell, CA: Author, 1980.
- Zilog Inc. *Z80-CPU Z80A-CPU Technical Manual*, Campbell, CA: Author, 1978.
- Zilog Inc. *Z80 CTC Counter/Timer Circuit Technical Manual*, Campbell, CA: Author, 1982.
- Zilog Inc. *Z80 DMA Technical Manual*, Cupertino, CA: Author, 1981.
- Zilog Inc. *Z80 PIO Z80A PIO Technical Manual*, Campbell, CA: Author, 1978.
- Zilog Inc. *Z80 SIO Technical Manual*, Campbell, CA: Author, 1978.
- Zilog Inc. *Z280 MPU Microprocessor Unit, Preliminary Product Specifications*. Campbell, CA: Author, 1987.

# Z80 Instruction Set

Appendix A describes each Z80 instruction fully in terms of its operation and the operand, including details such as number of bytes, machine cycles, T-states, Hex code, and affected flags. The instructions appear in alphabetical order and are illustrated with examples.

The following abbreviations and symbols are used in the description of the instruction set.

r = Z80 Registers

rp = Register Pair

rx = Index Registers

r' = Z80 Alternate Registers

m = Memory Location

r<sub>s</sub> = Register Source

r<sub>d</sub> = Register Destination

( ) = Contents of

d = 7-bit Displacement (Expressed in  
2's Complement for Backward  
Displacement)

b = Bit from 0 to 7

MC = Machine Cycles

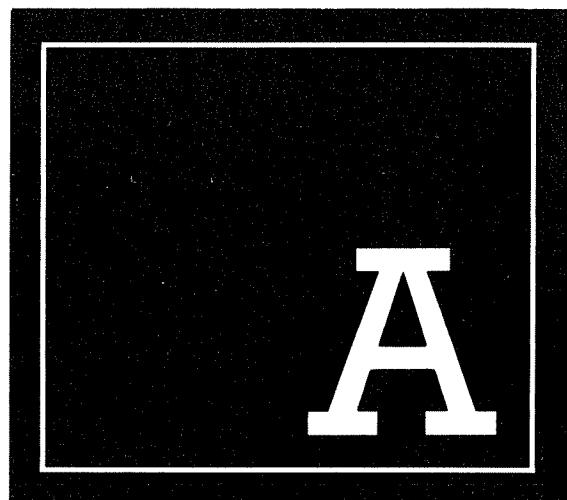
1 = Flag Set

0 = Flag Reset

✓ = Flag Affected

[Blank] = No Effect On Flag

? = Flag Indeterminate



**Flags**

S = Sign  
 Z = Zero  
 H = Half-Carry  
 P/V = Parity/Overflow  
 N = Add/Subtract  
 C = Carry (In the description of an instruction, the abbreviation CY is used instead of C to avoid confusion with register C.)

cc = Flag Condition Code

P = Plus  
 M = Minus  
 Z = Zero  
 NZ = No Zero  
 PE = Parity Even  
 PO = Parity Odd  
 C = Carry  
 NC = No Carry

**ADC A, r: ADD REGISTER TO ACCUMULATOR WITH CARRY**  
**ADC A, 8-BIT: ADD 8-BIT TO ACCUMULATOR WITH CARRY**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
--------	---------	-------	--------------	-----------

ADC	r	1	1 /4	A    B    C    D    E    H    L
				8F    88    89    8A    8B    8C    8D

8-Bit	2	2 /7(4,3)	CE 8-Bit
-------	---	-----------	----------

**Description** The contents of the operand (register or 8-bit data) and the carry flag CY are added to the contents of the accumulator and the result is placed in the accumulator.

Flags	S	Z	H	P/V	N	C
	✓	✓		✓		✓

**Example** The register BC contains  $2498_H$  and the register DE contains  $54A1_H$ . Add (BC) and (DE) and store the result in BC.

- Step 1:** Copy (B) into A and add (D) using ADD instruction.  $(C) \rightarrow (A): 98_H$   
**(E): A1H**  
**(A): 1/39H**
- Step 2:** Save the sum ( $39_H$ ) in register C.
- Step 3:** Copy (B) into A and add (D) using the ADC instruction to account for the carry from the previous sum.  $(B) \rightarrow (A): 24_H$   
**(D): 54H**
- Step 4:** Save the sum ( $79_H$ ) in B. The ADC instruction resets the previous CY flag. **Carry: 1**  
**79H**

**Comments** The instruction is generally used in 16-bit addition or multi-byte number; the carry generated by bit  $D_7$  is added to bit  $D_0$  of the next addition. This instruction should not be used to account for carries generated in summing 8-bit numbers.

**ADC A, (HL)** : ADD THE CONTENTS OF MEMORY AND CARRY TO ACCUMULATOR

**ADC A, (IX + d):**

**ADC A, (IY + d):**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
ADC	A, (HL)	1	2 /7(4,3)	8E
ADC	A, (IX + d)	3	5 /19(4,4,3,5,3)	DD 8E d
	A, (IY + d)			FD 8E d

**Description** The contents of memory specified by the operand (HL, IX + d, or IY + d) and the carry flag are added to the contents of the accumulator.

	S	Z	H	P/V	N	C
Flags	✓	✓		✓	✓	0 ✓

**ADC HL, rp: ADD REGISTER PAIR TO HL WITH CARRY**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
ADC	HL, rp	2	2 /15(4,4,4,3)	BC DE HL SP ED 4A 5A 6A 7A

**Description** The contents of the operand (BC, DE, HL, or SP) and the carry flag are added to the contents of the HL register and the result is stored in the HL register.

	S	Z	H	P/V	N	C
Flags	✓	✓	?	✓	0	✓

**Example** The HL register contains  $8200_H$ , the DE register contains  $F850_H$ , and the carry flag CY is set. Add the contents of HL and DE with carry.

Mnemonics: ADC HL, DE      Hex Code: ED 5A

Addition with Carry							
(DE):	1111	1000	0101	0000	(F850 <sub>H</sub> )		
(HL):	1000	0010	0000	0000	(8200 <sub>H</sub> )		
CY :				1			
	1/0111	1010	0101	0001	(7A51 <sub>H</sub> )		

Register Contents After Instruction

	S	Z	H	VNC	
A	X	0	0	1	0,1
D	F8				50
H	7A				51

**ADD A, r: ADD REGISTER TO ACCUMULATOR**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
--------	---------	-------	--------------	-----------

ADD	A, r	1	1 /4	A B C D E H L 87 80 81 82 83 84 85
-----	------	---	------	---------------------------------------

**Description** The contents of the register are added to the contents of the accumulator, and the result is stored in the accumulator.

Flags	S	Z	H	P/V	N	C
	✓	✓	✓	✓	0	✓

**Example** Register B has  $51_H$  and the accumulator has  $47_H$ . Add (B) to (A).

Mnemonics: ADD A, B      Hex Code: 80

Addition				Register Contents After Execution			
(B)	0101	0001	(51 <sub>H</sub> )	S	Z	H	VNC
(A)	0100	0111	(47 <sub>H</sub> )	A	98	1,0,0,0,0,0	F
(A)	1001	1000	(98 <sub>H</sub> )	B	51	X	C

**ADD A, 8-BIT: ADD 8-BIT TO ACCUMULATOR**

Opcode	Operand	Bytes	MC /T-States	Hex Code
--------	---------	-------	--------------	----------

ADD	8-Bit	2	2 /7(4,3)	C6 8-Bit
-----	-------	---	-----------	----------

**Description** The operand byte (8-bit data) is added to the contents of the accumulator, and the result is placed in the accumulator.

Flags	S	Z	H	P/V	N	C
	✓	✓	✓	✓	0	✓

**Example** The accumulator contains  $4A_H$ . Add the data byte  $59_H$  to the contents of the accumulator.

Mnemonics: ADD A, 59H      Hex Codes: C6 59

Addition				Register Contents After Execution			
(A)	0100	1010	(4A <sub>H</sub> )	S	Z	H	VNC
8-bit	0101	1001	(59 <sub>H</sub> )	A	A3	1,0,1,1,0,0	
	1010	0011	(A3 <sub>H</sub> )				

**ADD A, (HL)** : ADD CONTENTS OF MEMORY TO ACCUMULATOR

**ADD A, (IX + d):**

**ADD A, (IY + d):**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
ADD	A, (HL)	1	2 /7(4,3)	86
ADD	A, (IX + d)	3	5 /19(4,4,3,5,3)	DD 86 d
	A, (IX + d)			FD 86 d

**Description** The contents of the accumulator are added to the contents of memory location shown by the address in HL registers, and the result is stored in the accumulator. When the operand is an index register, the memory address is calculated by adding the index register and the displacement byte d.

	S	Z	H	P/V	N	C
Flags	✓	✓	✓	✓	0	✓

**Example** The accumulator contains the byte  $76_{H}$ , and the HL pair contains  $2050_{H}$ . Add the byte  $A2_{H}$  which is stored in memory location  $2050_{H}$  to the contents of the accumulator.

Mnemonics: ADD A, (HL)      Hex Code:86

Addition			Register Contents After Execution				
			S	Z	H	VNC	
(A) 0111	0110	( $76_{H}$ )					
(2050)Mem 1010	0010	( $A2_{H}$ )	A	18	0,0,0,0,0,1	F	
1 0001	1000		H	20	50		L
CY							

**Example** Use the index register IX as a memory pointer in Example A to add the contents of the accumulator and memory location  $2050_{H}$ . Assume that the index register IX contains  $2035_{H}$ . The displacement byte d ( $1B_{H}$ ) is calculated by subtracting the index address  $2035_{H}$  from the memory address  $2050_{H}$ .

Mnemonics: ADD A, (IX + 1B)      Hex Code: DD 86 1B

**ADD HL, rp: ADD REGISTER PAIR TO HL**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
ADD	HL, rp	1	3 /11(4,4,3)	BC DE HL SP 09 19 29 39

**Description** The contents of the specified register rp (BC, DE, HL, or SP) are added to the contents of HL and the result is placed in HL.

Flags	S	Z	H	P/V	N	C
			✓		0	✓

C = 1 if bit D<sub>15</sub> generates carry; otherwise it is reset.

H = 1 if bit D<sub>11</sub> generates carry; otherwise it is reset.

**Example** The HL register contains 2900<sub>H</sub> and the DE register contains F895<sub>H</sub>. Add (HL) and (DE).

Mnemonics: ADD HL, DE      Hex Code: 19

Addition				Register Contents After Instruction			
(HL): 0010 1001 0000 0000	(2900 <sub>H</sub> )			D	F8	95	E
(DE): 1111 1000 1001 0101	(F895 <sub>H</sub> )			H	21	95	L
(HL): 1/0010 0001 1001 0101	(2195 <sub>H</sub> )						
CY = 1, H = 1							

#### ADD IX, rp: ADD REGISTER PAIR TO INDEX REGISTER

#### ADD IY, rp:

Opcode	Operand	Bytes	MC /T-States	Hex Codes
ADD	IX, rp	2	4 /15(4,4,4,3)	BC DE IX/IY SP
				IX: DD 09 19 29 39
				IY: FD 09 19 29 39

**Description** The contents of the specified register pair rp are added to the contents of the index register and the result is placed in the index register.

1. This is a 2-byte instruction. The first byte specifies the index register (IX or IY), and the second byte specifies the register pair to be added to the index register.
2. This instruction cannot add (HL) to an index register or add (IX) and (IY). If the second byte is 29<sub>H</sub>, the contents of the index register specified adds to its own contents.

Flags	S	Z	H	P/V	N	C
			?		0	✓

#### AND r: LOGICALLY AND REGISTER WITH ACCUMULATOR

Opcode	Operand	Bytes	MC /T-States	Hex Codes
AND	r	1	1 /4	A B C D E H L A7 A0 A1 A2 A3 A4 A5

**Description** The contents of the specified register are ANDed with the contents of the accumulator and the result is placed in the accumulator.

	S	Z	H	P/V	N	C	
Flags	✓	✓	1		✓	0	0

**Example** The contents of the accumulator and register B are  $54_H$  and  $82_H$  respectively. Logically AND (B) with (A) and show the flags and the contents of each register after ANDing.

Mnemonics: AND B      Hex Code: A0

Logical AND				Register Contents After Instruction			
(A)	1 0 0 0 0 0 1 0	(82 <sub>H</sub> )	S Z H PNC	A	00	0,1,1,1,0,0	F
(B)	0 1 0 1 0 1 0 0	(54 <sub>H</sub> )	B	54	X		C
(A)	0 0 0 0 0 0 0 0	(00 <sub>H</sub> )					

#### AND 8-BIT: LOGICALLY AND 8-BIT WITH ACCUMULATOR

Opcode	Operand	Bytes	MC /T-States	Hex Codes
AND	8-Bit	2	2 /7(4,3)	E6 8-Bit

**Description** The contents of the operand (8-bit data) are logically ANDed with the contents of the accumulator and the result is placed in the accumulator.

	S	Z	H	P/V	N	C	
Flags	✓	✓	1		✓	0	0

**Example** The accumulator contains  $A3_H$ . AND byte  $97_H$  with (A).

Mnemonics: AND 97H      Hex Code: E6 97

Logical AND				Register Contents After Instruction			
(A)	:1 0 1 0 0 0 1 1	(A3 <sub>H</sub> )	S Z H PNC	A	83	1,0,0,0,0,0	F
(Data):	1 0 0 1 0 1 1 1	(97 <sub>H</sub> )					
(A)	:1 0 0 0 0 0 1 1	(83 <sub>H</sub> )					

#### AND (HL) : LOGICALLY AND CONTENTS OF MEMORY WITH ACCUMULATOR

**AND (IX + d):**

**AND (IY + d):**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
AND	(HL)	1	2 /7(4,3)	A6
AND	(IX + d)	3	5 /19	DD A6 d
	(IY + d)		(4,4,3,5,3)	FD A6 d

**Description** The contents of memory are ANDed with the contents of the accumulator. The memory address is specified by the contents of the HL register or an index register with a displacement byte d.

	S	Z	H	P/V	N	C
Flags	✓	✓	1		✓	0 0

**Example** Write mnemonics to AND the contents of memory location 2070<sub>H</sub> with the contents of the accumulator, assuming index register IY contains address 2000<sub>H</sub>.

Mnemonics: AND (IY + 70H)      Hex Code: FD A6 70

---

#### BIT b, r: TEST BIT IN REGISTER

Opcode	Operand	Bytes	MC /T-States	Hex Codes
--------	---------	-------	--------------	-----------

BIT	b, r	2	2 /8(4,4)	Source Register
(Bit)				A B C D E H L
CB (7)				7F 78 79 7A 7B 7C 7D
CB (6)				77 70 71 72 73 74 75
CB (5)				6F 68 69 6A 6B 6C 6D
CB (4)				67 60 61 62 63 64 65
CB (3)				5F 58 59 5A 5B 5C 5D
CB (2)				57 50 51 52 53 54 55
CB (1)				4F 48 49 4A 4B 4C 4D
CB (0)				47 40 41 42 43 44 45

**Description** This instruction tests the specified bit in a given register r and sets the Z flag if bit is zero; otherwise, Z flag is reset. The register r can be any one of the registers: A, B, C, D, E, H, L.

	S	Z	H	P/V	N	C
Flags	?	✓			?	0

**Example** Register B has 1000 0111(87<sub>H</sub>). Test bit D<sub>3</sub>.

Mnemonics: BIT 3, B      Hex Code: CB 58

This instruction tests bit D<sub>3</sub> and sets the Z flag because D<sub>3</sub> = 0.

---

#### BIT b, (HL) : TEST BIT IN MEMORY LOCATION

**BIT b, (IX + d):**

**BIT b, (IY + d):**

Opcode	Operand	Bytes	MC /T-States	(BIT)	7	6	5	4	3	2	1	0
BIT	b, (HL)	2	3 /12(4,4,4,)	CB	[7E	76	6E	66	5E	56	4E	46]
BIT	b, (IX + d)	4	5 /20	DD CB d	[7E	76	6E	66	5E	56	4E	46]
BIT	b, (IY + d)		(4,4,3,5,4)	FD CB d	[7E	76	6E	66	5E	56	4E	46]

**Description** This instruction tests the bit in the specified memory location and sets Z flag if the bit is zero. The memory location is specified by the contents of HL or index registers (plus displacement).

	S	Z	H	P/V	N	C		
Flags	[?]	✓	[ ]	1	[ ]	[?]	0	[ ]

### CALL 16-BIT: CALL SUBROUTINE SPECIFIED BY OPERAND

Opcode	Operand	Bytes	MC /T-States	Hex Code
CALL	16-Bit	3	5 /17(4,3,4,3,3)	CD 16-bit

**Description** The program execution is transferred to the subroutine address specified by the operand. Before the transfer, the address of the opcode following the CALL (the contents of the program counter) is stored on the stack. The sequence of events is described in the example below. This instruction should be accompanied by one of the return (RET or conditional RET) instructions in the subroutine.

**Flags** No flags are affected.

**Example** Write the instruction to call the subroutine located at memory location 2050<sub>H</sub>. Explain how the contents of the program counter are stored on the stack if the stack pointer is at location 2099<sub>H</sub>.

Instruction: CALL 2050H      Hex Code: CD 50 20

As an example, this machine code can be stored as follows:

Memory Address	Hex Code	Mnemonics
2010	CD	CALL 2050H
2011	50	
2012	20	

Make a note of the difference between writing a 16-bit address as mnemonics and machine code. In the code, the low-order byte (50<sub>H</sub>) is entered first, followed by the

high-order byte ( $20_H$ ). However, in mnemonics the bytes are shown in the proper sequence. If an assembler is used to obtain the codes, it will automatically reverse the sequence of the mnemonics.

When the last machine code ( $20_H$ ), located at  $2012_H$ , is fetched by the microprocessor, the program counter holds the address  $2013_H$ . This address is placed on the stack as follows.

1. Stack pointer is decremented to  $2098_H$  and the MSB is stored.
2. Stack pointer is decremented to  $2097_H$  and the LSB is stored.
3. Call address ( $2050_H$ ) is temporarily stored in internal registers and placed on the bus for the fetch cycle.

	2097	13
↑	2098	20
SP	2099	XX

#### **CALL cc, 16-BIT: CALL SUBROUTINE IF CONDITION IS TRUE**

Opcode	Operand	Bytes	MC /T-States
--------	---------	-------	--------------

CALL	cc, 16-Bit	3	5 /17(4,3,4,3,3); If condition is true 3 /10(4,3,3) ; If condition is false
------	------------	---	--

Condition Flags	NZ	Z	NC	C	PO	PE	P	M
Hex Codes	C4	CC	D4	DC	E4	EC	F4	FC

**Description** The program execution is transferred to the subroutine address specified by the 16-bit of the operand if the flag condition is true. If the condition is false, the program continues without calling the subroutine.

**Flags** No flags are affected.

**Example** Write two conditional Call instructions: one with Carry set (C) and the other with Zero flag not set (NZ).

Instructions:	1) CALL C, 2050H	Hex Codes:
	2) CALL NZ, 2070H	DC 50 20 C4 70 20

#### **CCF: COMPLEMENT CARRY FLAG**

Opcode	Operand	Bytes	MC /T-States	Hex Code
CCF		1	1 /4	3F

**Description** The Carry flag is complemented.

Flags	S	Z	H	P/V	N	C
	[ ]	[ ]	[ ? ]	[ ]	[ 0 ]	[ ✓ ]

**CP r : COMPARE REGISTER WITH ACCUMULATOR**  
**CP 8-Bit: COMPARE 8-BIT DATA WITH ACCUMULATOR**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
CP	r	1	1 /4	A B C D E H L BF B8 B9 BA BB BC BD
CP	8-bit	2	2 /7(4,3)	FE 8-bit

**Description** The operand is compared with the accumulator by subtracting the contents of the operand from the contents of the accumulator. None of the contents are altered and the comparison is shown by setting the flags as follows:

- If  $(A) < (r/8\text{-bit})$ : Carry flag is set and Zero flag is reset.
- If  $(A) = (r/8\text{-bit})$ : Zero flag is set and Carry flag is reset.
- If  $(A) > (r/8\text{-bit})$ : Carry and Zero flags are reset.

**Flags** In addition to C and Z, the other flags are also modified to reflect the result of the operation.

S	Z	H	P/V	N	C
✓	✓		✓	✓	1 ✓

**Example** Register B contains data byte  $62_H$  and the accumulator contains data byte  $57_H$ . Compare (B) with (A).

Mnemonics: CP B      Hex Code: B8

Register Contents Before Instruction			Register Contents After Instruction		
			S	Z	H VNC
A	57	X X	F	A	57 1 0 0 0 1 1
B	62	X X	C	B	62 X X

- No contents are changed.
- Carry flag is set because  $(A) < (B)$ .
- Other flags are also modified as shown.

**CP (HL) : COMPARE MEMORY CONTENTS WITH  
ACCUMULATOR**

**CP (IX + d):**  
**CP (IY + d):**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
CP	(HL)	1	2 /7(4,3)	BE
CP	(IX + d)	3	5 /19(4,4,3,5,3)	DD BE 8-bit
CP	(IY + d)			FD BE 8-bit

**Description** The memory is compared with the accumulator by subtracting the contents of the memory from the contents of the accumulator. None of the contents are altered and the comparison is shown by setting the flags as follows. The memory address is specified by the contents of the HL register or index register.

- If  $(A) < (M)$ : Carry flag is set and Zero flag is reset.
- If  $(A) = (M)$ : Zero flag is set and Carry flag is reset.
- If  $(A) > (M)$ : Carry and Zero flags are reset.

**Flags** In addition to CY and Z, the other flags are also modified to reflect the result of the operation.

S	Z	H	P/V	N	C
/	/		/	/	/

**Example** The memory location  $2050_H$  contains  $64_H$ , the accumulator contains  $64_H$ , and the HL register holds the address  $2050_H$ . Write the instruction to compare the contents of the accumulator with the contents of the memory location  $2050H$  and show the status of the flags.

Mnemonics: CP (HL)      Hex Code: BE

Register Contents Before Instruction			Memory Contents		Register Contents After Instruction				
A	64	X X	F	204F	XX	A	64	0,1,0,0,1,0	F
H	20	50	L	2050	64	H	20	50	L

- No contents are changed.
- Zero flag is set because  $(A) = (M)$ .
- Other flags are also modified as shown.

#### CPD: COMPARE MEMORY WITH ACCUMULATOR, AND DECREMENT MEMORY ADDRESS AND BYTE COUNTER

Opcode	Operand	Bytes	MC /T-States	Hex Code
CPD		2	4 /16(4,4,3,5)	ED A9

**Description** The contents of the memory location addressed by the HL register are compared with the contents of the accumulator, and the flags are set as follows without altering the contents. The HL and BC registers are decremented. Register BC can be used as a byte counter.

- If  $(A) < (M)$ : Sign flag is set and Zero flag is reset.
- If  $(A) = (M)$ : Zero flag is set and Sign flag is reset.
- If  $(A) > (M)$ : Sign and Zero flags are reset.

**Flags** In addition to S and Z, the other flags are also modified to reflect the result of the operation.

S	Z	H	P/V	N	C	
✓	✓		✓		0/1	1
P/V = 0 if BC = 0						
= 1 if BC ≠ 0						

**CPDR: COMPARE MEMORY WITH ACCUMULATOR, AND  
DECREMENT MEMORY ADDRESS AND BYTE COUNTER UNTIL  
CONTENTS ARE EQUAL OR COUNTER IS ZERO**

Opcode	Bytes	MC /T-States	Hex Code
CPDR	2	5 /21(4,4,3,5,5) 4 /16(4,4,3,5)	if BC ≠ 0 and (A) ≠ (HL) if BC = 0 or (A) = (HL)
			ED B9

**Description** The contents of the memory location addressed by the HL register are compared with the contents of the accumulator, and HL and BC registers are decremented. The instruction is repeated until either BC = 0 or (A) = (HL). Register BC is used as a byte counter.

	S	Z	H	P/V	N	C	
Flags	✓	✓		✓		0/1	1
	Z := 1 if (A) = (HL)						P/V = 0 if BC = 0
	= 1 if BC ≠ 0						<i>TP N2, B1.C</i>

**Example** The contents of the registers are (A) = 9F<sub>H</sub>, BC = 000F<sub>H</sub>, and (HL) = 2099<sub>H</sub>. The memory location 2090<sub>H</sub> has the byte 9F. Specify the contents of the registers after the execution of the instruction CPDR.

Instruction: CPDR      Hex Code: ED B9

The instruction begins its search from the location 2099<sub>H</sub>, and it will be repeated ten times until the memory location 2090<sub>H</sub>, where (A) = (HL). The contents of the registers at the end of the search will be as follows:

	S	Z	H	P/VNC	
A	9F	0	1	1	1, 1, 0 F
B	00			05	C
H	20			8F	L

**CPI: COMPARE MEMORY WITH ACCUMULATOR,  
INCREMENT MEMORY ADDRESS, AND DECREMENT BYTE  
COUNTER**

Opcode	Operand	Bytes	MC /T-States	Hex Code
CPI		2	4 /16(4,4,3,5)	ED A1

**Description** The contents of the memory location addressed by the HL register are compared with the contents of the accumulator, and the flags are set as follows without altering the contents. The HL register is incremented and the BC register is decremented. Register BC can be used as a byte counter.

- If  $(A) < (M)$ : Sign flag is set and Zero flag is reset.
- If  $(A) = (M)$ : Zero flag is set and Sign flag is reset.
- If  $(A) > (M)$ : Sign and Zero flags are reset.

**Flags** In addition to S and Z, the other flags are also modified to reflect the result of the operation.

S	Z	H	P/V	N	C
/	/	/	0/1	1	

$$\begin{aligned} P/V &= 0 \text{ if } BC = 0 \\ &= 1 \text{ if } BC \neq 0 \end{aligned}$$

**CPIR: COMPARE MEMORY WITH ACCUMULATOR,  
INCREMENT MEMORY ADDRESS, AND DECREMENT BYTE  
COUNTER UNTIL CONTENTS ARE EQUAL OR COUNTER IS  
ZERO**

Opcode	Bytes	MC /T-States	Hex Code
CPIR	2	5 /21(4,4,3,5,5) if $BC \neq 0$ and $(A) \neq (HL)$ 4 /16(4,4,3,5) if $BC = 0$ or $(A) = (HL)$	ED B1

**Description** The contents of the memory location addressed by the HL register are compared with the contents of the accumulator, and the HL register is incremented and the BC register decremented. The instruction is repeated until either  $BC = 0$  or  $(A) = (HL)$ . Register BC is used as a byte counter.

Flags	S	Z	H	P/V	N	C
	/	/	/	0/1	1	

$$\begin{aligned} Z &= 1 \text{ if } (A) = (HL) \\ P/V &= 0 \text{ if } BC = 0 \\ &= 1 \text{ if } BC \neq 0 \end{aligned}$$

**Example** The contents of the registers are  $(A) = 9F_H$ ,  $BC = 000F_H$ , and  $(HL) = 2090_H$ . None of the memory locations between  $2090_H$  and  $209F_H$  has the byte  $9F$ . Specify the contents of the registers after the execution of the instruction CPIR.

Instruction: CPIR      Hex Code: ED B1

The instruction begins its search from the location  $2090_H$ , and it will be repeated fifteen times until the byte counter BC is zero. The contents of the registers at the end of the search will be as follows:

	S	Z	H	PNC	
A	9F	0	0	1	F S and H flags will be determined by the last comparison.
B	00		00		C
H	20		A0		L

#### CPL: COMPLEMENT ACCUMULATOR

Opcode	Operand	Bytes	MC /T-States	Hex Code
CPL		1	1 /4	2F

**Description** The contents of the accumulator are complemented (inverted or 1's complement).

	S	Z	H	P/V	N	C
Flags				1		1

**Example** The accumulator has  $89_H$ . Show the contents after the execution of the instruction CPL.

Mnemonics: CPL      Hex Code: 2F

Before Instruction			After Instruction		
A	<span style="border: 1px solid black; padding: 2px;">1 0 0 0 1 0 0 1</span>	$(89H)$	A	<span style="border: 1px solid black; padding: 2px;">0 1 1 1 0 1 1 0</span>	$(76H)$

#### DAA: DECIMAL ADJUST ACCUMULATOR

Opcode	Operand	Bytes	MC /T-States	Hex Code
DAA		1	1 /4	27

**Description** If this instruction is used after an addition or subtraction of two BCD numbers, the result is adjusted for BCD values. This instruction uses the Half Carry (H) flag internally to convert the binary result into BCD values shown as follows.

After an addition of two BCD numbers

1. If the value of the low-order four bits ( $D_3-D_0$ ) in the accumulator is greater than 9 or if H flag is set, the instruction adjusts the low-order bits by adding 06 (0 1 1 0) to  $D_3-D_0$ .
2. If the value of the high-order bits ( $D_7-D_4$ ) is greater than 9 or if CY flag is set, the instruction adjusts the high-order bits by adding 60 (0 1 1 0) to  $D_7-D_4$ .

After a subtraction of two BCD numbers, the above procedure is also valid, except the instruction adds 2's complement of 06 or 60 to the respective group of digits.

	S	Z	H	P/V	N	C
Flags	✓	✓	✓	✓	✓	✓

**Example** The accumulator contains  $85_{BCD}$  and register B contains  $68_{BCD}$ . Add the two numbers and adjust the result for the BCD value.

Mnemonics:	ADD	B	Hex Code:	80	
	DAA			27	
(A)	1	0	0	$0\ 1\ 0\ 1$	$(85)_{BCD}$
(B)	+0	1	1	$0\ 1\ 0\ 0$	$(68)_{BCD}$
	<hr/>				$(153)_{BCD}$

The binary sum is  $ED_H$ , and the values of both low- and high-order four bits are higher than 9 (1 0 0 1). The instruction adds 6 (0 1 1 0) to both groups as shown below:

(A)	1	1	1	0	1	$1\ 1\ 0\ 1$	$(ED)$
	$+0\ 1\ 1\ 0\ 0\ 1\ 1\ 0$				$(66)$		
	<hr/>				$1\ 53_{BCD}$		
					CY		

The accumulator contains 53 and the CY flag is set to indicate that the sum is larger than eight bits. The program should keep track of the carry; otherwise it may be altered by subsequent instructions.

**Example** The accumulator contains  $97_{BCD}$  and register B contains  $39_{BCD}$ . Subtract (B) from (A) and adjust the result for decimal numbers.

Mnemonics:	SUB	B	Hex Codes:	90
	DAA			27

The subtraction is performed in 2's complement as follows:

(A)	1	0	0	1	0	1	1	1	97
+ (B)	1	1	0	0	0	1	1	1	2's Comp. (39)
	<hr/>								$1\ 0\ 1\ 0\ 1\ 1\ 1\ 0 \rightarrow 5E_H$

After the subtraction, the low-order byte is larger than (1 0 0 1). The instruction adjusts

the result by adding 2's complement of 06 (1 1 1 1 1 0 1 0) as shown.

$$\begin{array}{r}
 \text{(A)} \quad 0 \ 1 \ 0 \ 1 \quad 1 \ 1 \ 1 \ 0 \quad 5E \\
 + \ 1 \ 1 \ 1 \ 1 \quad 1 \ 0 \ 1 \ 0 \quad \text{2's Comp. (06)} \\
 \hline
 1 \ 0 \ 1 \ 0 \ 1 \quad 1 \ 0 \ 0 \ 0 \rightarrow 58_{BCD}
 \end{array}$$


---

**DEC r** : DECREMENT REGISTER CONTENTS

**DEC (HL)** : DECREMENT MEMORY CONTENTS

**DEC (IX + d):**

**DEC (IY + d):**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
DEC	r	1	1 /4	A B C D E H L 3D 05 0D 15 1D 25 2D
DEC	(HL)	1	3 /11(4,4,3)	35
DEC	(IX + d)	3	6 /23(4,4,3,5,4,3)	DD 35 d
DEC	(IY + d)			FD 35 d

**Description** The contents of the designated register/memory location are decremented by 1. If the operand is a memory location, it is specified by the contents of HL or index registers.

Flags	S	Z	H	P/V	N	C
	✓	✓		✓		1

---

**DEC rp: DECREMENT REGISTER PAIR OR INDEX REGISTER**

**DEC rx:**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
DEC	rp	1	1 /6	BC DE HL SP 0B 1B 2B 3B
DEC	IX	2	2 /10(4,6)	DD 2B
DEC	IY			FD 2B

**Description** The contents of the specified register are decremented by 1; the contents are viewed as a 16-bit number.

**Flags** No flags are affected.

**Example** Register HL contains  $2000_H$ . Specify the contents of the entire register after the instruction DEC HL.

Mnemonics: DEC HL Hex Code: 2B

The contents of the HL register will be  $1FFF_H$ .

**DI: DISABLE INTERRUPTS**

Opcode	Operand	Bytes	MC /T-States	Hex Code
DI		1	1 /4	F3

**Description** This instruction resets the interrupt enable flip-flops (IFF1 and IFF2) and disables maskable interrupts.

**Flags** No flags are affected.

**DJNZ d: JUMP RELATIVE IF B IS NOT ZERO**

Opcode	Operand	Bytes	MC /T-States	Hex Code
DJNZ	d	2	3 /13(5,3,5) if B ≠ 0	10 d
			2 /8(3,5) if B = 0	

**Description** Register B is decremented, and if B ≠ 0, the program execution is transferred to the memory location by adding displacement byte to the program counter +2.

**Flags** No flags are affected.

**EI: ENABLE INTERRUPTS**

Opcode	Operand	Bytes	MC /T-States	Hex Code
EI		1	1 /4	FB

**Description** This instruction sets the interrupt enable flip-flops (IFF1 and IFF2) to logic 1 and enables the maskable interrupts.

**Flags** No flags are affected.

**Comments** After the system reset or the acknowledgment of an interrupt, the interrupt enable flip-flops are reset, thus disabling the interrupts. This instruction must be executed to reenable the maskable interrupts.

**EX AF, AF': EXCHANGE ACCUMULATOR AND FLAGS WITH  
ALTERNATE ACCUMULATOR AND FLAGS**

Opcode	Operand	Bytes	MC /T-States	Hex Code
EX	AF, AF'	1	1 /4	08

**Description** The contents of the accumulator and the flag register are exchanged with their respective alternate registers.

**Flags** All flags are affected.

---

#### EX DE, HL: EXCHANGE HL AND DE REGISTERS

Opcode	Operand	Bytes	MC /T-States	Hex Code
EX	DE, HL	1	1 /4	EB

**Description** The contents of register H are exchanged with the contents of register D, and the contents of register L are exchanged with the contents of register E.

**Flags** No flags are affected.

---

#### EX (SP), HL: EXCHANGE CONTENTS OF REGISTERS WITH TOP OF STACK

**EX (SP), IX:**

**EX (SP), IY:**

Opcode	Operand	Bytes	MC /T-States	Hex Code
EX	(SP), HL	1	5 /19(4,3,4,3,5)	E3
EX	(SP), IX	2	6 /23(4,4,3,4,3,5)	DD E3
EX	(SP), IY			FD E3

**Description** The contents of the low-order register (L or the low-order byte of an index register) are exchanged with the contents of the memory location pointed to by the stack pointer. The contents of the high-order register or the high-order byte of an index register are exchanged with the contents of the next memory (stack) location ( $SP + 1$ ).

**Flags** No flags are affected.

**Example** The contents of registers and stack location are as follows:

Register Contents			Stack Contents		
H	80	FF	L	2097	32
SP	20	97		2098	A2
				2099	

After the execution of the instruction EX (SP), HL, the contents of registers and stack

locations will be as follows:

Register Contents			Stack Contents	
H	A2	32	L	2097 FF
SP	20	97		2098 80
				2099

---

#### EXX: EXCHANGE REGISTERS WITH ALTERNATE REGISTERS

Opcode	Operand	Bytes	MC /T-States	Hex Code
EXX		1	1 /4	D9

**Description** The contents of the general-purpose registers BC, DE, and HL are exchanged with the contents of their respective alternate registers BC', DE' and HL'.

**Flags** No flags are affected.

---

#### HALT: SUSPEND OPERATIONS

Opcode	Operand	Bytes	MC /T-States	Hex Code
HALT		1	1 /4	76

**Description** This instruction suspends (halts) all operations, and the microprocessor waits until an interrupt or the reset is received. During the halt, the microprocessor continues to execute NOP instruction to maintain memory refresh cycles.

**Flags** No flags are affected.

---

#### IM 0: SET UP INTERRUPT MODE 0

Opcode	Operand	Bytes	MC /T-States	Hex Code
IM	0	2	2 /8(4,4)	ED 46

**Description** This instruction sets up the microprocessor in Interrupt Mode 0. In this mode, the interrupting device can insert any instruction onto the data bus to restart the MPU execution; the first byte must be inserted during the interrupt acknowledge cycle.

**Flags** No flags are affected.

---

#### IM 1: SET UP INTERRUPT MODE 1

Opcode	Operand	Bytes	MC /T-States	Hex Code
IM	1	2	2 /8(4,4)	ED 56

**Description** This instruction sets up the microprocessor in Interrupt Mode 1. In this mode, the MPU responds to an interrupt by executing the restart at location 0038<sub>H</sub>.

**Flags** No flags are affected.

---

#### IM 2: SET UP INTERRUPT MODE 2

Opcode	Operand	Bytes	MC /T-States	Hex Code
IM	2	2	2 /8(4,4)	ED 5E

**Description** This instruction sets up the microprocessor in Interrupt Mode 2. In this mode, the MPU responds to an interrupt by executing an indirect call to the specified 16-bit address of a memory location. The low-order 8-bit address is supplied by the interrupting device, and the high-order address is supplied by the contents of the interrupt vector register I.

**Flags** No flags are affected.

---

#### IN A, (8-BIT): INPUT DATA TO ACCUMULATOR FROM A PORT WITH 8-BIT ADDRESS

Opcode	Operand	Bytes	MC /T-States	Hex Code
IN	8-bit	2	3 /11(4,3,4)	DB 8-bit

**Description** The contents of the input port specified in the operand are read and placed in the accumulator.

**Flags** No flags are affected.

---

#### IN r, (C): INPUT DATA TO REGISTER FROM A PORT WITH ADDRESS IN C

Opcode	Operand	Bytes	MC /T-States	Hex Codes
IN	r, (C)	2	3 /12	A B C D E H L ED 78 40 48 50 58 60 68

**Description** The contents of the input port with the address in register C are read and placed in the specified register.

Flags	S	Z	H	P/V	N	C
	✓	✓		✓	✓	0

**INC r: INCREMENT REGISTER**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
INC	r	1	1 /4	A B C D E H L 3C 04 0C 14 1C 24 2C

**Description** The contents of the specified register are incremented by 1.

Flags	S	Z	H	P/V	N	C
	✓	✓	✓	✓	0	

**Example** The accumulator contains  $FF_H$ . Specify the contents of the accumulator and the status of CY and Z flags after the INC instruction.

Mnemonics: INC A Hex Code: 3C

$$\begin{array}{r}
 (A) = \begin{array}{cccccc} 1 & 1 & 1 & 1 & 1 & 1 \end{array} \\
 + \begin{array}{cccccc} 0 & 0 & 0 & 0 & 0 & 1 \end{array} \\
 \hline
 (A) = \begin{array}{cccccc} 1 & 0 & 0 & 0 & 0 & 0 \end{array}
 \end{array}$$

After the byte FF has been incremented, the sum should be 00 with carry. However, the INC instruction does not set the CY flag even though it affects all other flags. If ADD instruction is used, the accumulator contents will be 00 with the CY flag set.

**INC rp: INCREMENT REGISTER PAIR OR INDEX REGISTER****INC rx:**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
INC	rp	1	1 /6	BC DE HL SP 03 13 23 33
INC	IX	2	2 /10(4,6)	DD 23
INC	IY			FD 23

**Description** The contents of the specified register pair or the index register are incremented by 1.

**Flags** No flags are affected.

**INC (HL): INCREMENT MEMORY CONTENTS****INC (IX + d):****INC (IY + d):**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
INC	(HL)	1	3 /11(4,4,3)	34
INC	(IX + d)	3	6 /23(4,4,3,5,4,3)	DD 34 d
INC	(IY + d)			FD 34 d

**Description** The contents of the specified memory are incremented by 1. The memory location is specified either by the contents of HL register or the contents of an index register plus the displacement.

	S	Z	H	P/V	N	C
Flags	✓	✓	✓	✓	0	

**Example** The memory register at  $2040_H$  contains the byte  $FF_H$ . The index register IX holds the address  $2050_H$ . Write the instruction to increment the contents of the memory register and specify the status of the flags.

Mnemonics: INC (IX + F0H)      Hex Codes: DD 34 F0

The displacement is calculated by taking 2's complement of  $10_H$ , the difference between  $2050_H$  and  $2040_H$ . After the execution of the instruction, the memory contents will be 00. The flag status will be as follows:

	S	Z	H	V	N	C
	0	1		1	0	0

Note: This instruction does not affect the C flag.

#### IND: INPUT DATA TO MEMORY AND DECREMENT BYTE COUNTER AND MEMORY POINTER

Opcode	Operand	Bytes	MC /T-States	Hex Code
IND		2	4 /16(4,5,3,4)	ED AA

**Description** This instruction reads the input port specified by the contents of register C, and the reading is stored in the memory location specified by the contents of register HL. Register B is used as a byte counter, and both B and HL are decremented by one.

	S	Z	H	P/V	N	C
Flags	?	0/1	?	?	1	

$$\begin{aligned} Z &= 1 \text{ if } B = 0 \\ &= 0 \text{ if } B \neq 0 \end{aligned}$$

#### INDR: INPUT DATA TO MEMORY AND DECREMENT MEMORY POINTER AND BYTE COUNTER UNTIL BYTE COUNTER IS ZERO

Opcode	Operand	Bytes	MC /T-States	Hex Code
INDR		2	5 /21(4,5,3,4,5) if $B \neq 0$	ED BA
		4	/16(4,5,3,4) if $B = 0$	

**Description** This instruction reads the input port specified by the contents of register C, and the reading is stored in the memory location specified by the contents of register HL. Register B is used as a byte counter, both B and HL are decremented by one, and the instruction is repeated until B = 0.

	S	Z	H	P/V	N	C
Flags	?	1	?	?	1	

**Example** The contents of the registers are  $HL = 2070_H$  and  $BC = 0401_H$ . Show the contents of memory locations and registers after the execution of the instruction INDR.

Instruction: INDR      Hex Code: ED BA

The instruction reads the data at the input port  $01_H$  four times until register B = 0 and stores the data in memory starting from  $2070_H$ . The contents of registers and the memory locations are as follows:

Register Contents Before Instruction		Memory Contents		Register Contents After Instruction	
B	04	01	C	206C	
H	20	70	L	206D	BYTE4
				206E	BYTE3
				206F	BYTE2
				2070	BYTE1

**INI:** INPUT DATA TO MEMORY, DECREMENT BYTE COUNTER, AND INCREMENT MEMORY POINTER

Opcode	Operand	Bytes	MC /T-States	Hex Code
JNI		2	4 /16(4,5,3,4)	ED A2

**Description** This instruction reads the input port specified by the contents of register C, and the reading is stored in the memory location specified by the contents of register HL. The contents of register B are decremented and those of register HL are incremented by one.

	S	Z	H	P/V	N	C
Flags	?	0/1	?	?	1	

$$Z = \begin{cases} 1 & \text{if } B = 0 \\ 0 & \text{if } B \neq 0 \end{cases}$$

**INIR: INPUT DATA TO MEMORY, INCREMENT MEMORY  
POINTER, AND DECREMENT BYTE COUNTER UNTIL BYTE  
COUNTER IS ZERO**

Opcode	Operand	Bytes	MC /T-States	Hex Code
INIR		2	5 /21(4,5,3,4,5) if B≠0 4 /16(4,5,3,4) if B = 0	ED B2

**Description** This instruction reads the input port specified by the contents of register C, and the reading is stored in the memory location specified by the contents of register HL. The contents of register B are decremented and those of register HL are incremented by one. Register B is used as a byte counter, and the instruction is repeated until B = 0.

	S	Z	H	P/V	N	C
Flags	?	1	?	?	1	

**Example** The contents of the registers are HL = 2070<sub>H</sub> and BC = 0407<sub>H</sub>. Show the contents of memory locations and registers after the execution of the instruction INIR.

Instruction: INIR      Hex Code: ED B2

The instruction reads the data at the input port 07<sub>H</sub> four times until register B = 0 and stores the data in memory starting from 2070<sub>H</sub>. The contents of registers and the memory locations are as follows:

Register Contents Before Instruction		Memory Contents		Register Contents After Instruction	
B	04	07	C	2070	BYTE1
H	20	70	L	2071	BYTE2
				2072	BYTE3
				2073	BYTE4
				2074	

#### JP 16-BIT: JUMP TO 16-BIT ADDRESS

Opcode	Operand	Bytes	MC /T-States	Hex Code
JP	16-Bit	3	3 /10(4,3,3)	C3 16-bit

**Description** The program execution is transferred to the memory location specified by the 16-bit address. This is a 3-byte instruction; the second byte specifies the low-order byte and the third byte specifies the high-order byte of the 16-bit address.

**Flags** No flags are affected.

**Example** Write the instruction at location 2010<sub>H</sub> to transfer the program sequence to location 2050<sub>H</sub>.

Instruction: JP 2050H Hex Code: C3 50 20

This machine code can be stored as follows:

Memory Address	Hex Code	Mnemonics
2010	C3	JP 2050H
2011	50	
2012	20	

Make a note of the difference between writing a 16-bit address as mnemonics and machine code. In the code, the low-order byte (50) is entered first, followed by the high-order byte (20). However, in mnemonics the bytes are shown in the proper sequence. If an assembler is used to obtain the codes, it will automatically reverse the sequence of the machine codes.

#### JP cc, 16-BIT: JUMP TO 16-BIT ADDRESS IF CONDITION IS TRUE

Opcode	Operand	Bytes	MC /T-States	Hex Codes
JP	cc, 16-Bit	3	3 /10	NZ Z NC C P0 PE P M C2 CA D2 DA E2 EA F2 FA

**Description** The program execution is transferred to the address specified by the 16-bit of the operand if the flag condition is true. If the condition is false, the program continues to the next memory location.

**Flags** No flags are affected.

**Example** Write two conditional Jump instructions: one with Carry set (C) and the other when the Zero flag is not set (NZ).

Instructions: 1) JP C, 2050H Hex Codes: DA 50 20  
2) JP NZ, 2070H Hex Codes: C2 70 20

#### JP (HL): JUMP TO MEMORY LOCATION SPECIFIED BY HL OR INDEX REGISTERS

JP (IX):  
JP (IY):

Opcode	Operand	Bytes	MC /T-States	Hex Codes
JP	(HL)	1	1 /4	E9
JP	(IX)	2	2 /8(4,4)	DD E9
JP	(IY)			FD E9

**Description** The program execution is transferred to the memory location specified by the contents of the HL register or the index register.

**Flags** No flags are affected.

#### JR: JUMP RELATIVE EQUAL TO DISPLACEMENT

Opcode	Operand	Bytes	MC /T-States	Hex Codes
JR	d	2	3 /12(4,3,5)	18 d

**Description** The program execution is transferred to the memory location specified by the sum of the present program counter and the displacement byte. The value of a displacement byte can be positive for a forward jump or in 2's complement for a backward jump. The total range of the jump is  $-126$  to  $+129$ ; this accounts for the additional two memory locations due to the instruction Jump Relative.

**Flags** No flags are affected.

**Example** Write the instruction at location  $2010_{\text{H}}$  to transfer the program sequence to location  $2050_{\text{H}}$ .

Instruction: JR 3EH      Hex Code: 18 3E

This machine code can be stored as follows:

Memory Address	Hex Code	Mnemonics
2010	18	JR 3EH
2011	3E	
2012		

When the instruction Jump Relative located at  $2010_{\text{H}}$  is executed, the program counter contains  $2012_{\text{H}}$ . By adding  $3E_{\text{H}}$  to  $2012_{\text{H}}$ , the program counter contains  $2050_{\text{H}}$ ; thus, the program is transferred to the location  $2050_{\text{H}}$ . Therefore, to calculate the displacement byte for a forward jump, the value obtained by subtracting the instruction location ( $2010_{\text{H}}$ ) from the jump location ( $2050_{\text{H}}$ ) should be reduced by two. On the other hand, two should be added to the displacement byte in 2's complement for a backward jump as shown in the following example.

**Example** Write the instruction at location  $2010_{\text{H}}$  to transfer the program sequence to location  $2000_{\text{H}}$ .

Instruction: JR EEH      Hex Code: 18 EE

This machine code can be stored as follows:

Memory Address	Hex Code	Mnemonics
2010	18	JR EEH
2011	EE	
2012		

The displacement byte is calculated as follows:

$$\begin{array}{rcl}
 (\text{PC}) & = & 2\ 0\ 1\ 2_{\text{H}} \\
 \text{Jump} & = & 2\ 0\ 0\ 0_{\text{H}} \\
 \text{Location} & & \hline
 & & 1\ 2_{\text{H}} = 0\ 0\ 0\ 1\ 0\ 0\ 1\ 0 \\
 & & \text{2's Complement of } 12_{\text{H}} = 1\ 1\ 1\ 0\ 1\ 1\ 1\ 0 \quad (\text{EE}_{\text{H}})
 \end{array}$$


---

**JR cc, d: JUMP RELATIVE EQUAL TO DISPLACEMENT IF FLAG CONDITION IS TRUE**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
JR	cc, d	2	3 /12(4,3,5) If condition is true 2 /7(4,3) If condition is false	NZ Z NC C 20 28 30 38

**Description** The program execution is transferred to the memory location specified by the sum of the present program counter and the displacement byte if the condition is true. The value of a displacement byte can be positive for a forward jump or in 2's complement for a backward jump. The total range of the jump is -126 to +129; this accounts for the additional two memory locations due to the instruction Jump Relative.

Note: There are no conditional relative jump instructions based on other flags.

**Flags** No flags are affected.

**Example** Write the instruction at location  $2010_{\text{H}}$  to transfer the program sequence to location  $2000_{\text{H}}$  if Carry is set.

Instruction: JR C, EEH Hex Code: 38 EE

This machine code can be stored as follows:

Memory Address	Hex Code	Mnemonics
2010	38	JR C, EEH
2011	EE	
2012		

For the calculation of the displacement byte, see the example in the previous instruction.

#### **LD r<sub>d</sub>, r<sub>s</sub>: COPY SOURCE REGISTER INTO DESTINATION REGISTER**

Opcode	Operand	Bytes	MC /T-States	Hex Codes							
LD	r <sub>d</sub> , r <sub>s</sub>	1	1 /4								
				Source Register							
				A	B	C	D	E	H	L	
				7F	78	79	7A	7B	7C	7D	
				B	47	40	41	42	43	44	45
				C	4F	48	49	4A	4B	4C	4D
				D	57	50	51	52	53	54	55
				E	5F	58	59	5A	5B	5C	5D
				H	67	60	61	62	63	64	65
				L	6F	68	69	6A	6B	6C	6D

**Description** The contents of the source register r<sub>s</sub> are copied into the destination register r<sub>d</sub>. The letters r<sub>s</sub> and r<sub>d</sub> represent any of the registers A, B, C, D, E, H, and L.

**Flags** No flags are affected.

**Example** Register B contains 72<sub>H</sub> and register C contains 9F<sub>H</sub>. Transfer the contents of register B to register C.

Mnemonics: LD C, B      Hex Code: 48

Note that the first operand C specifies the destination register and the second operand B specifies the source register.



#### **LD r, 8-BIT: LOAD REGISTER r WITH 8-BIT DATA**

Opcode	Operand	Bytes	MC /T-States	Hex Codes							
LD	r, 8-Bit	2	2 /7(4,3)	A	B	C	D	E	H	L	
				[3E	06	0E	16	1E	26	2E]	8-bit

**Description** The second byte (8-bit data) is loaded into the specified register r. Register r can be any of the registers A, B, C, D, E, H, or L.

**Flags** No flags are affected.

**Example** Load  $92_H$  into register B.

Mnemonics: LD B, 92H Hex Code: 06 92

**LD r, (HL):** COPY CONTENTS OF MEMORY INTO REGISTER

**LD r, (IX + d):**

**LD r, (IY + d):**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
LD	r, (HL)	1	2 /7(4,3)	A B C D E H L 7E 46 4E 56 5E 66 6E
LD	r, (IX + d)	3	5 /19 (4,4,3,5,3)	DD [7E 46 4E 56 5E 66 6E] d FD [7E 46 4E 56 5E 66 6E] d
	r, (IY + d)			

Opcode	Operand	Bytes	MC /T-States	Hex Codes
LD	r, (HL)	1	2 /7(4,3)	A B C D E H L 7E 46 4E 56 5E 66 6E
LD	r, (IX + d)	3	5 /19 (4,4,3,5,3)	DD [7E 46 4E 56 5E 66 6E] d FD [7E 46 4E 56 5E 66 6E] d
	r, (IY + d)			

**Description** The contents of the memory location indicated by the HL register or by one of the index registers (plus displacement) is copied into the specified register r. Register r can be any one of the registers A, B, C, D, E, H, or L.

**Flags** No flags are affected.

**Example** Assume the contents of register HL are  $20_H$  and  $50_H$ , respectively. The byte  $9F_H$  is stored in memory location  $2050_H$ . Copy the contents of the memory location  $2050_H$  into register D.

Mnemonics: LD D, (HL) Hex Code: 56

Register Contents Before Instruction	Memory Contents	Register Contents After Instruction
D XX XX E H 20 50 L	2050 9F	D 9F XX E H 20 50 L

**Example** Assume the index register IX has  $2040_H$ . Copy the contents of memory location  $2050_H$  into register D as in the previous example.

Mnemonics: LD D, (IX + 10H) Hex Code: DD 56 10

This instruction adds the displacement byte  $10_H$  to the contents of the index register ( $2040_H$ ) and points to location  $2050_H$ . Then it copies the contents of  $2050_H$  into register D.

**LD (HL), r :** COPY CONTENTS OF REGISTER INTO MEMORY

**LD (IX + d), r:**

**LD (IY + d), r:**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
LD	(HL), r	1	2 /7(4,3)	A B C D E H L 77 70 71 72 73 74 75
LD	(IX + d), r	3	5 /19	DD [77 70 71 72 73 74 75] d
	(IY + d), r		(4,4,3,5,3)	FD [77 70 71 72 73 74 75] d

**Description** The contents of register r are copied into the memory location specified by either the contents of the HL register pair or one of the index registers (plus displacement). Register r represents any one of the registers A, B, C, D, E, H, or L.

**Flags** No flags are affected.

**Example** Register B contains  $98_H$  and the register pair HL contains  $2065_H$ . Copy the contents of register B into memory location  $2065_H$ .

Mnemonics: LD (HL), B Hex Code: 70

This instruction copies the contents of register B ( $98H$ ) into memory location  $2065H$ .

---

#### LD (HL), 8-BIT : LOAD 8-BIT INTO MEMORY

**LD (IX + d), 8-BIT:**

**LD (IY + d), 8-BIT:**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
LD	(HL), 8-Bit	2	3 /10(4,3,3)	36 8-bit
LD	(IX + d), 8-Bit	4	5 /19(4,4,3,5,3)	DD 36 d 8-bit
	(IY + d), 8-Bit			FD 36 d 8-bit

**Description** The 8-bit data are loaded into the specified memory location. The address of the memory location is specified by the contents of register pair HL or by one of the index registers (plus displacement).

**Flags** No flags are affected.

**Example** Assume the contents of register pair HL are  $20_H$  and  $50_H$ , respectively. Load the byte  $97_H$  into memory location  $2050_H$ .

Mnemonics: LD (HL), 97H Hex Code: 36 97

This instruction loads  $97_H$  into memory location  $2050_H$ .

**Example** Explain the data transfer in the following instruction if the index register IY holds the contents  $2060_H$ .

Instruction: LD (IY + 0FH), 00H Hex Code: FD 36 0F 00

This instruction adds the displacement byte  $0F_H$  to the contents of the index register IY ( $2060_H$ ) and specifies the memory location  $206F_H$ , then clears that location by loading  $00_H$ .

---

**LD A, (16-BIT): COPY MEMORY CONTENTS INTO ACCUMULATOR**

**LD A, (BC) :**

**LD A, (DE) :**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
LD	A, (16-Bit)	3	4 /13(4,3,3,3)	3A 16-bit
LD	A, (BC)	1	2 /7(4,3)	0A
	A, (DE)			1A

**Description** The contents of the specified memory location are copied into the accumulator. The memory location is specified either directly by 16-bit address or by the contents of BC or DE registers.

**Flags** No flags are affected.

**Example** The register BC contains  $2050_H$  and the byte  $F8_H$  is stored in memory location  $2050_H$ . Copy the byte into the accumulator.

Mnemonics: LD A, (BC) Hex Code: 0A

This instruction copies the contents,  $F8_H$ , of the memory location  $2050_H$  into the accumulator.

**Example** Write the instruction to copy the byte from the memory location  $2050_H$  into the accumulator.

Instruction: LD A, (2050H) Hex Code: 3A 50 20

Note that the 16-bit address is entered in the reversed order: the low-order byte ( $50_H$ ) first, followed by the high-order byte ( $20_H$ ).

---

**LD (16-BIT), A: COPY ACCUMULATOR CONTENTS INTO MEMORY**

**LD (BC), A :**

**LD (DE), A :**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
LD	(16-Bit), A	3	4 /13(4,3,3,3)	32 16-bit
LD	(BC), A	1	2 /7(4,3)	02
LD	(DE),A			12

**Description** The contents of the accumulator are copied into the specified memory location. The memory location is specified either directly by a 16-bit address or by the contents of BC or DE registers.

**Flags** No flags are affected.

**Example** Write instructions to copy the contents of the accumulator into memory location 2050<sub>H</sub> by using the direct addressing and the indirect addressing methods.

Direct Addressing: LD (2050H), A Hex Code: 32 50 20

Indirect : To use the DE pair as a memory pointer, the address 2050<sub>H</sub> Addressing : must be loaded into the DE register. Then the following instruction can be used:

LD (DE), A Hex Code: 12

---

**LD A, I: COPY INTERRUPT VECTOR REGISTER INTO ACCUMULATOR**

**LD A, R: COPY MEMORY REFRESH REGISTER INTO ACCUMULATOR**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
LD	A, I	2	2 /9(4,5)	ED 57
LD	A, R	2	2 /9(4,5)	ED 5F

**Description** The first instruction copies the contents of the interrupt vector register and the second instruction copies the contents of the memory refresh register into the accumulator.

Flags	S	Z	H	P/V	N	C
	/	/	0	0/1	0	

P/V = IFF2 Flag

---

**LD I, A: COPY ACCUMULATOR INTO INTERRUPT VECTOR REGISTER**

**LD R, A: COPY ACCUMULATOR INTO MEMORY REFRESH REGISTER**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
LD	I, A	2	2 /9(4,5)	ED 47
LD	R, A	2	2 /9(4,5)	ED 4F

**Description** The instructions copy the contents of the accumulator into the interrupt vector register and the memory refresh register, respectively.

**Flags** No flags are affected.

---

**LD rp, 16-BIT: LOAD 16-BIT INTO REGISTER PAIR**

**LD IX, 16-BIT: LOAD 16-BIT INTO INDEX REGISTER**

**LD IY, 16-BIT:**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
				BC DE HL SP
LD	rp, 16-Bit	3	3 /10(4,3,3)	01 11 21 31 16-bit
LD	IX, 16-bit	4	4 /14(4,4,3,3)	DD 21 16-bit
LD	IY, 16-bit			FD 21 16-bit

**Description** 16-bit data are loaded into the specified register pair or index register. The 16-bit data are entered low-order byte first, followed by the high-order byte.

**Flags** No flags are affected.

**Example** Write instructions to load 2050<sub>H</sub> into register BC and 4000<sub>H</sub> into index register IY.

Instructions: LD BC, 2050H Hex Code: 01 50 20  
                  LD IY, 4000H FD 21 00 40

Note the order of 16-bit data: low-order byte first, followed by the high-order byte.

**LD rp, (16-BIT): LOAD CONTENTS OF TWO MEMORY**  
**LD rx, (16-BIT): LOCATIONS INTO REGISTER PAIR OR INDEX**  
**LD HL, (16-BIT): REGISTER**

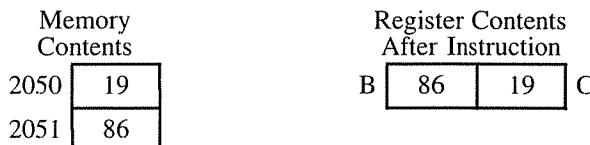
Opcode	Operand	Bytes	MC /T-States	Hex Codes
				BC DE HL SP
LD	rp, (16-Bit)	4	6 /20(4,4,3,3,3,3)	ED [4B 5B 6B 7B] 16-bit
LD	IX, (16-bit)			DD 2A 16-bit
LD	IY, (16-bit)			FD 2A 16-bit
LD	HL, (16-bit)	3	5 /16	2A 16-bit

**Description** The instruction copies the contents of the memory location specified by the 16-bit address into low-order register and then copies the contents of the next memory location into high-order register.

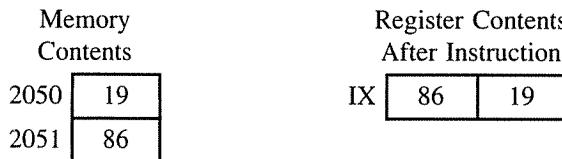
**Flags** No flags are affected.

**Example** The memory locations 2050<sub>H</sub> and 2051<sub>H</sub> contain the data bytes 19<sub>H</sub> and 86<sub>H</sub> respectively. Copy the memory contents into BC and IX registers.

Instruction: LD BC, (2050H) Hex Code: ED 4B 50 20



Instruction: LD IX, (2050H) Hex Code: DD 2A 50 20



**LD (16-BIT), rp:** LOAD CONTENTS OF REGISTER PAIR OR INDEX REGISTER INTO TWO CONSECUTIVE MEMORY LOCATIONS

**LD (16-BIT), XY:**

**LD (16-BIT), HL:**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
LD	(16-Bit), rp	4	6 /20(4,4,3,3,3,3)	BC DE HL SP
LD	(16-bit), IX			ED [43 53 63 73] 16-bit
LD	(16-bit), IY			DD 22 16-bit
LD	(16-bit), HL	3	5 /16	FD 22 16-bit
				22 16-bit

**Description** The instruction copies the contents of the low-order register into memory location specified by the 16-bit address and copies the contents of the high-order register into the next memory location.

**Flags** No flags are affected.

**Example** The BC register contains data 408FH. Copy the register contents into memory locations 2050H and 2051H.

Instruction: LD (2050H), BC Hex Code: ED 43 50 20



**LD SP, HL:** COPY HL OR INDEX REGISTER INTO STACK POINTER

**LD SP, IX:**

**LD SP, IY:**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
LD	SP, HL	1	1 /6	F9
LD	SP, IX	2	2 /10(4,6)	DD F9
LD	SP, IY			FD F9

**Description** The instruction copies the contents of the HL register or an index register into the stack pointer.

**Flags** No flags are affected.

**Example** The HL register contains  $2050_H$ . Specify the contents of the stack pointer after executing the following instruction.

Instruction: LD SP, HL    Hex Code: F9

After the execution of the above instruction, both the stack pointer and the HL register will have  $2050_H$ . The contents of the source are not destroyed.

**LDD: COPY DATA FROM SOURCE MEMORY TO  
DESTINATION MEMORY, AND DECREMENT MEMORY  
POINTERS AND COUNTER**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
LD	D	1	4 /16(4,4,3,5)	ED A8

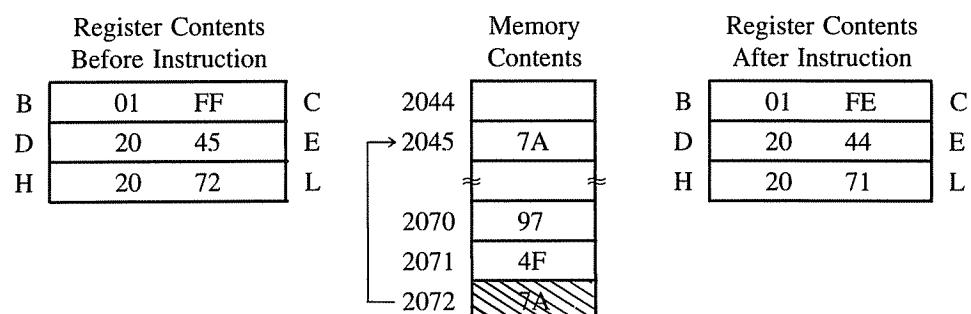
**Description** The contents of the memory location addressed by the HL register are copied into the memory location addressed by the DE register. Then registers BC, DE, and HL are decremented. In this instruction, HL functions as a source memory pointer, DE as a destination memory pointer, and BC as a counter.

Flags	S	Z	H	P/V	N	C
			0	0/1	0	

$$\begin{aligned} P/V &= 0 \text{ if } BC = 0 \\ &= 1 \text{ if } BC \neq 0 \end{aligned}$$

**Example** The memory locations  $2070_H$ ,  $2071_H$ , and  $2072_H$  contain  $97_H$ ,  $4F_H$ , and  $7A_H$ , respectively. The contents of the registers are  $HL = 2072_H$ ,  $DE = 2045_H$ , and  $BC = 01FF_H$ . Show the contents of memory locations and registers after the execution of the instruction LDD.

Instruction: LDD    Hex Code: ED A8



**LDDR: COPY DATA FROM SOURCE MEMORY TO  
DESTINATION MEMORY, AND DECREMENT MEMORY  
POINTERS AND COUNTER UNTIL COUNTER IS ZERO**

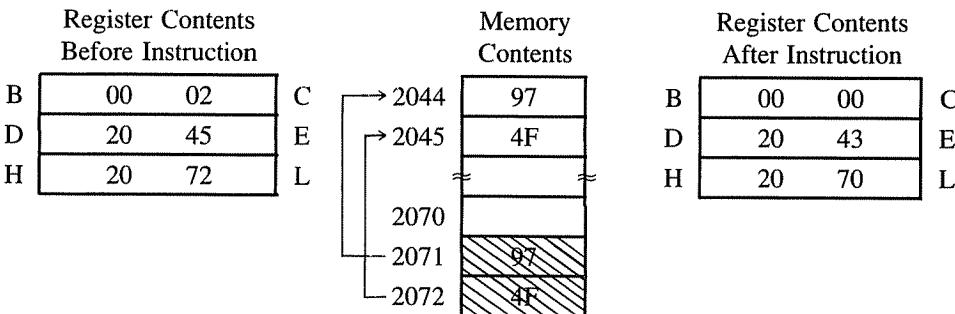
Opcode	Operand	Bytes	MC /T-States	Hex Codes
LDDR		2	5 /21(4,4,3,5,5) if BC ≠ 0 4 /16(4,4,3,5) if BC = 0	ED B8

**Description** The contents of the memory location addressed by the HL register are copied into the memory location addressed by the DE register. Then registers BC, DE, and HL are decremented, and the copying process is continued until BC = 0. In this instruction, HL functions as a source memory pointer, DE as a destination memory pointer, and BC as a counter.

Flags	S	Z	H	P/V	N	C
			0		0	0

**Example** The memory locations  $2071_H$  and  $2072_H$  contain  $97_H$  and  $4F_H$ , respectively. The contents of the registers are HL =  $2072_H$ , DE =  $2045_H$ , and BC =  $0002_H$ . Show the contents of memory locations and registers after the execution of the instruction LDDR.

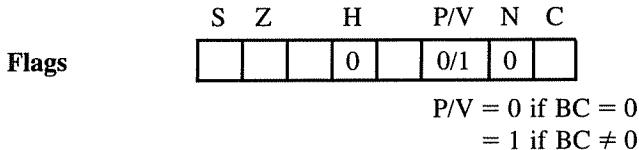
Instruction: LDDR      Hex Code: ED B8



**LDI: COPY DATA FROM SOURCE MEMORY TO DESTINATION  
MEMORY, INCREMENT MEMORY POINTERS, AND  
DECREMENT COUNTER**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
LDI		2	4 /16(4,4,3,5)	ED A0

**Description** The contents of the memory location addressed by the HL register are copied into the memory location addressed by the DE register. Then DE and HL are incremented and BC is decremented. In this instruction, HL functions as a source memory pointer, DE as a destination memory pointer, and BC as a counter.



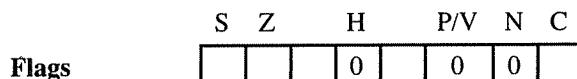
**Example** The memory locations  $2070_H$ ,  $2071_H$ , and  $2072_H$  contain  $97_H$ ,  $4F_H$ , and  $7A_H$ , respectively. The contents of the registers are  $HL = 2070_H$ ,  $DE = 2045_H$ , and  $BC = 01FF_H$ . Show the contents of memory locations and registers after the execution of the instruction LDI.

Instruction: LDI			Hex Code: ED A0	Register Contents After Instruction				
Register Contents Before Instruction			Memory Contents					
B	01 FF	C	2044	<input type="checkbox"/>	<input type="checkbox"/>	B	01 FE	C
D	20 45	E	→ 2045	<input type="checkbox"/> 97	<input type="checkbox"/>	D	20 46	E
H	20 70	L	2070	<input type="checkbox"/> 97	<input checked="" type="checkbox"/>	H	20 71	L
			2071	<input type="checkbox"/> 4F	<input type="checkbox"/>			
			2072	<input type="checkbox"/> 7A	<input type="checkbox"/>			

**LDIR: COPY DATA FROM SOURCE MEMORY TO DESTINATION MEMORY, INCREMENT MEMORY POINTERS, AND DECREMENT COUNTER UNTIL IT IS ZERO**

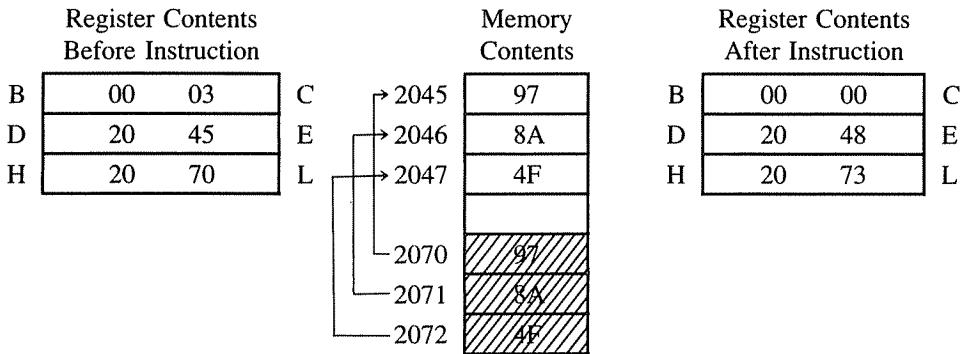
Opcode	Operand	Bytes	MC /T-States	Hex Codes
LDIR		2	5 /21(4,4,3,5,5) if BC ≠ 0 4 /16(4,4,3,5) if BC = 0	ED B0

**Description** The contents of the memory location addressed by the HL register are copied into the memory location addressed by the DE register. Then registers DE and HL are incremented and BC is decremented, and the copying process is continued until BC = 0. In this instruction, HL functions as a source memory pointer, DE as a destination memory pointer, and BC as a counter.



**Example** The memory locations  $2070_H$ ,  $2071_H$ , and  $2072_H$  contain  $97_H$ ,  $8A_H$ ,  $4F_H$ , respectively. The contents of the registers are  $HL = 2070_H$ ,  $DE = 2045_H$ , and  $BC = 0003_H$ . Show the contents of memory locations and registers after the execution of the instruction LDIR.

Instruction: LDIR      Hex Code: ED B0

**NEG: TAKE 2's COMPLEMENT (NEGATE ACCUMULATOR)**

Opcde	Opernd	Bytes	MC /T-States	Hex Code
NEG		2	2 /8 (4, 4)	ED 44

**Description** The instruction converts the accumulator contents into 2's complement by subtracting the contents from 0.

	S	Z	H	P/V	N	C
Flags	✓	✓		✓	1	✓

**Example** The accumulator holds  $97_{\text{H}}$ . Specify the contents of the accumulator and explain the S and CY flags after the execution of NEG instruction.

Mnemonics: NEG      Hex Code: ED 44

Before Instruction: (A) = 1 0 0 1 0 1 1 1 ( $97_{\text{H}}$ )After Instruction : (A) = 0 1 1 0 1 0 0 1 ( $69_{\text{H}}$ )

Flags : S = 0, CY = 1

The Sign flag is reset because  $D_7 = 0$ , and the Carry flag is set because the larger number  $97_{\text{H}}$  is subtracted from 00. However, the result does not mean that  $69_{\text{H}}$  is a negative number; it depends upon the interpretation of  $97_{\text{H}}$ . The appropriate interpretation is that  $97_{\text{H}}$  and  $69_{\text{H}}$  are 2's complements of each other.

**NOP: NO OPERATION**

Opcde	Opernd	Bytes	MC /T-States	Hex Code
NOP		1	1 /4	00

**Description** No operation is performed. The instruction is fetched and decoded; however, no operation is executed.

**Flags** No flags are affected.

**Comments** The instruction is used to increase time delays or delete and insert instructions while troubleshooting.

**OR r** : LOGICALLY OR REGISTER, 8-BIT, OR MEMORY WITH ACCUMULATOR

**OR 8-Bit:**

**OR (m)** :

Opcode	Operand	Bytes	MC /T-States	Hex Codes
OR	r	1	1 /4	A B C D E H L B7 B0 B1 B2 B3 B4 B5
OR	8-bit	2	2 /7(4,3)	F6 d
OR	(HL)	1	2 /7(4,3)	B6
OR	(IX + d) (IY + d)	3	5 /19(4,4,3,5,3)	DD B6 d FD B6 d

**Description** The contents of a register or memory or an 8-bit word are ORed with the contents of the accumulator. The memory address is specified by the contents of the HL register or an index register with a displacement byte d.

Flags	S	Z	H	P/V	N	C
	/	/	0		/	0 0

**Example** The contents of the accumulator and register B are  $54_{16}$  and  $82_{16}$  respectively. Logically OR (B) with (A) and show the flags and the contents of each register after ORing.

Mnemonics: OR B Hex Code: B0

Logical OR			Register Contents After Instruction						
(A) 1 0 0 0 0 0 1 0 (82 <sub>16</sub> )									
(B) 0 1 0 1 0 1 0 0 (54 <sub>16</sub> )									
(A) 1 1 0 1 0 1 1 0 (D6 <sub>16</sub> )			A D6   1 , 0 , 0 , 0 , 0 , 0 F B 54   X C						

**Example** Write mnemonics to OR the contents of memory location  $2070_{16}$  with the contents of the accumulator assuming index register IY contains address  $2000_{16}$ .

Mnemonics: OR (IY + 70H) Hex Code: FD B6 70

**OUT (8-Bit), A: OUTPUT DATA FROM ACCUMULATOR TO PORT WITH 8-BIT ADDRESS**

Opcode	Operand	Bytes	MC /T-States	Hex Code
OUT	8-Bit	2	3 /11(4,3,4)	D3 8-bit

**Description** The contents of the accumulator are copied into the I/O port specified by the 8-bit address.

**Flags** No flags are affected.

---

**OUT (C), r: OUTPUT DATA FROM REGISTER r TO THE PORT WITH ADDRESS IN C**

Opcode	Operand	Bytes	MC /T-States	Hex Codes							
OUT	(C), r	2	3 /12 ED	A	B	C	D	E	H	L	

**Description** The contents of register r are copied into the I/O port specified by the address in register C.

**Flags** No flags are affected.

---

**OUTD: OUTPUT DATA FROM MEMORY AND DECREMENT BYTE COUNTER AND MEMORY POINTER**

Opcode	Operand	Bytes	MC /T-States	Hex Code
OUTD		2	4 /16(4,5,3,4)	ED AB

**Description** This instruction copies data from the memory location specified by the contents of register HL into the output port specified by register C. Register B is used as a byte counter, and both B and HL are decremented by one.

	S	Z	H	P/V	N	C
Flags	[?]	[0/1]	[ ]	[?]	[ ]	[?]

**OTDR: OUTPUT DATA FROM MEMORY AND DECREMENT MEMORY POINTER AND BYTE COUNTER UNTIL BYTE COUNTER IS ZERO**

Opcode	Operand	Bytes	MC /T-States	Hex Code
OTDR		2	5 /21(4,5,3,4,5) if B≠0 4 /16(4,5,3,4) if B=0	ED BB

**Description** This instruction copies data from the memory location specified by the contents of register HL into the output port specified by register C. Register B is used as a byte counter, and both B and HL are decremented by one; the instruction is repeated until B = 0.

	S	Z	H	P/V	N	C
Flags	[?]	[1]	[ ]	[?]	[ ]	[?]

**Example** The contents of the registers are  $HL = 2070_H$  and  $BC = 0401_H$ . Show the contents of memory locations and registers after the execution of the instruction OTDR.

Instruction: OTDR      Hex Code: ED BB

The instruction copies data starting from memory  $2070_H$  to the output port  $01_H$  four times until register B = 0. Assuming the port  $01_H$  is a printer, the four bytes will be printed. The contents of registers at the end of the instruction are as follows:

Register Contents Before Instruction				Register Contents After Instruction			
B    04    01    C				B    00    01    C			
H    20    70    L				H    20    6C    L			

**OUTI: OUTPUT DATA FROM MEMORY, DECREMENT BYTE COUNTER, AND INCREMENT MEMORY POINTER**

Opcode	Operand	Bytes	MC /T-States	Hex Code
OUTI		2	4 /16(4,5,3,4)	ED A3

**Description** This instruction copies data from the memory location specified by the contents of register HL into the output port specified by register C. The contents of register B are decremented and those of register HL are incremented by one.

Flags	S	Z	H	P/V	N	C
	?	0/1		?		1

**OTIR: OUTPUT DATA FROM MEMORY, INCREMENT MEMORY POINTER, AND DECREMENT BYTE COUNTER UNTIL BYTE COUNTER IS ZERO**

Opcode	Operand	Bytes	MC /T-States	Hex Code
OTIR		2	5 /21(4,5,3,4,5) if B≠0 4 /16(4,5,3,4) if B = 0	ED B3

**Description** This instruction copies data from the memory location specified by the contents of register HL into the output port specified by register C. The contents of register B are decremented and those of register HL are incremented by one. The instruction is repeated until B = 0.

	S	Z	H	P/V	N	C
Flags	?	1		?	?	1

**Example** The contents of the registers are  $HL = 2070_H$  and  $BC = 0407_H$ . Show the contents of memory locations and registers after the execution of the instruction OTIR.

Instruction: OTIR      Hex Code: ED B3

The instruction copies data starting from memory location  $2070_H$  into output port  $07_H$ . The instruction is repeated four times until  $B = 0$ , and every time data is taken from the next memory location by incrementing HL. The contents of registers are as follows:

Register Contents Before Instruction			Register Contents After Instruction		
B    04    07    C			B    00    07    C		
H	20	70	L	H	20
					74

#### **POP rp: PLACE STACK CONTENTS INTO REGISTER PAIR OR INDEX REGISTER**

##### **POP ix:**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
POP	rp	1	3 /10(4,3,3)	BC DE HL AF
POP	IX	2	4 /14(4,4,3,3)	C1 D1 E1 F1
POP	IY			DD E1
				FD E1

**Description** The contents at the top of the stack indicated by the address in the stack pointer are copied into the low-order register (C, E, L, or F) or as a low-order byte into an index register and the stack pointer is incremented by one. Again, the contents of the top of the stack (after incrementing the stack pointer) are copied into the high-order register or as a high-order byte into an index register. The stack pointer is incremented by one to indicate the new top of the stack.

**Flags** No flags are affected.

**Example** Assume the stack pointer contains  $2090_H$ , data byte  $50_H$  is stored in location  $2090_H$  and  $80_H$  is stored in location  $2091_H$ . Transfer the contents of the top two stack locations into HL registers.

Mnemonics: POP HL      Hex Code: E1

Register Contents Before Instruction			Stack Contents		Register Contents After Instruction		
H	XX	XX	L	2090	50	H	80
SP	2090			2091	80	SP	2092

---

**PUSH rp: COPY REGISTER PAIR OR INDEX REGISTER ON STACK**

**PUSH rx:**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
PUSH	rp	1	3 /10(4,3,3)	BC DE HL AF C5 D5 E5 F5
PUSH	IX	2	4 /14(4,4,3,3)	DD E5
PUSH	IY			FD E5

**Description** The contents of the specified register pair or index register are copied into the stack locations as follows. First, the stack pointer is decremented by one and the contents of the high-order register (B, D, H, A) or high-order byte of the index register are copied into the memory location indicated by the stack pointer. The stack pointer is decremented again and the contents of the low-order register (C, E, L, F) or the low-order byte of the index register are copied into that location.

**Flags** No flags are affected.

**Example** Assume that the stack pointer contains 2099<sub>H</sub>, register H contains 40<sub>H</sub>, and register L contains F8<sub>H</sub>. Save the contents of HL on the stack.

Mnemonics: PUSH HL Hex Code: E5

Register Contents Before Instruction			Stack Contents		Register Contents After Instruction		
H	40	F8	L	2097	F8	H	40
SP	2099			2098	40	SP	2097
				2099	XX		

---

**RES b, r: RESET BIT IN A REGISTER OR IN MEMORY**

**RES b, (m):**

Opcode	Operand	Bytes	MC /T-States								Hex Codes							
			Register								Register							
RES	b, r	2	2	/8(4,4)	(Bit)	A	B	C	D	E	H	L						
					CB (0)	87	80	81	82	83	84	85						
					CB (1)	8F	88	89	8A	8B	8C	8D						
					CB (2)	97	90	91	92	93	94	95						

				CB (3)	9F	98	99	9A	9B	9C	9D						
				CB (4)	A7	A0	A1	A2	A3	A4	A5						
				CB (5)	AF	A8	A9	AA	AB	AC	AD						
				CB (6)	B7	B0	B1	B2	B3	B4	B5						
				CB (7)	BF	B8	B9	BA	BB	BC	BD						
				Bit													
				0 1 2 3 4 5 6 7													
RES	b, (HL)	2	4 /15 (4,4,4,3)	CB	[86	8E	96	9E	A6	AE	B6	BE]					
RES	b, (IX + d)	4	6/23 (4,4,3,5,4,3)	DD	CB	d	[86	8E	96	9E	A6	AE	B6	BE]			
RES	b, (IY + d)			FD	CB	d	[86	8E	96	9E	A6	AE	B6	BE]			

**Description** The specified bit is reset in a register or in memory. The values of b (0–7) correspond to bits D<sub>0</sub>–D<sub>7</sub>.

**Flags** No flags are affected.

**Example** Write instructions to reset bit 3 in register A and bit 0 in memory 2055<sub>H</sub>. Assume that registers HL are already loaded with the address 2055<sub>H</sub>.

Mnemonics	Hex Code
RES 3, A	CB 9F
RES 0, (HL)	CB 86

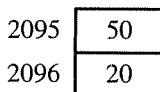
### RET: RETURN FROM SUBROUTINE UNCONDITIONALLY

Opcode	Operand	Bytes	MC /T-States	Hex Code
RET		1	3 /10(4,3,3)	C9

**Description** The program execution is transferred from the subroutine to the calling program. The two bytes from the top of the stack are copied into the program counter and the program execution begins at the new address. The instruction is equivalent to POP Program Counter.

**Flags** No flags are affected.

**Example** Assume that the stack pointer contains 2095<sub>H</sub>. Explain the effect of the RET instruction if the contents of the stack location are as follows:



After instruction RET, the contents of the top two stack locations are copied into the program counter, and the program execution is transferred to location 2050<sub>H</sub>. The stack pointer is incremented to location 2097<sub>H</sub>.

**RET cc: RETURN FROM SUBROUTINE IF CONDITION IS TRUE**

Opcode	Operand	Bytes	MC /T-States							
RET	cc	1	3 /11(5,3,3) ; If condition is true		1 /5 ; If condition is false					
Condition Flags			NZ	Z	NC	C	PO	PE	P	M
Hex Codes			C0	C8	D0	D8	E0	E8	F0	F8

**Description** The program execution is transferred from the subroutine to the calling program if the flag condition is true. If the condition is false, the program continues to the next memory location.

**Flags** No flags are affected.

**RETI: RETURN FROM INTERRUPT**

Opcode	Operand	Bytes	MC /T-States	Hex Code
RETI		2	4 /14(4,4,3,3)	ED 4D

**Description** The instruction copies the contents of the top two stack locations into the program counter, and the program execution is transferred to the address stored on the stack; this execution is similar to that of a RET instruction.

This instruction is used at the end of a maskable interrupt service routine. In addition to transferring the program execution to the interrupted program, it indicates to a Z80 family I/O device that it is the end of the service routine.

**Flags** No flags are affected.

**RETN: RETURN FROM NONMASKABLE INTERRUPT**

Opcode	Operand	Bytes	MC /T-States	Hex Code
RETN		2	4 /14(4,4,3,3)	ED 45

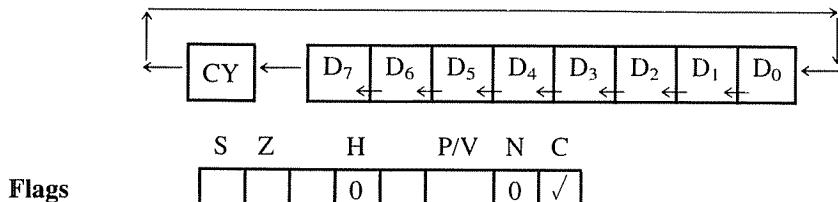
**Description** The instruction copies the contents of the top two stack locations into the program counter and the program execution is transferred to the address stored on the stack; this execution is similar to that of a RET instruction.

This instruction is used at the end of the nonmaskable interrupt service routine. In addition to transferring the program execution to the interrupted program, it restores the status of the maskable interrupts by copying the state of IFF2 (Interrupt Flip-flop 2) into IFF1 (Interrupt Flip-flop 1).

**RLA: ROTATE ACCUMULATOR LEFT THROUGH CARRY**

Opcode	Operand	Bytes	MC /T-States	Hex Code
RLA		1	1 /4	17

**Description** Each bit in the accumulator is rotated left by one position through the Carry flag. Bit D<sub>7</sub> is placed into the CY position and the CY flag is placed into bit D<sub>0</sub>.



The CY flag is modified according to bit D<sub>7</sub>.

**Example** The accumulator contains A7<sub>H</sub> and the CY flag is reset. Show the contents of the accumulator and the CY flag after the execution of the instruction RLA.

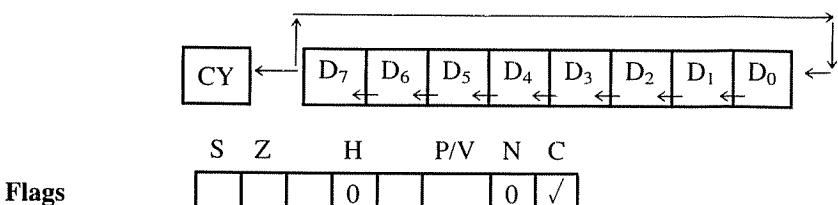
Mnemonics: RLA      Hex Code: 17

Accumulator Contents Before Instruction	CY 0	D <sub>7</sub> D <sub>6</sub> D <sub>5</sub> D <sub>4</sub> D <sub>3</sub> D <sub>2</sub> D <sub>1</sub> D <sub>0</sub>
		1 0 1 0 1 0 1 1
Accumulator Contents After Instruction	1	0 1 0 0 1 1 1 0

### RLCA: ROTATE ACCUMULATOR LEFT

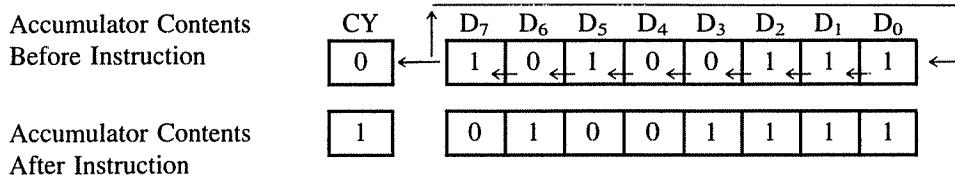
Opcode	Operand	Bytes	MC /T-States	Hex Code
RLCA		1	1 /4	07

**Description** Each bit in the accumulator is rotated left by one position. Bit D<sub>7</sub> is placed into the position of bit D<sub>0</sub> and the CY flag is modified according to bit D<sub>7</sub>.



**Example** The accumulator contains A7<sub>H</sub> and the CY flag is reset. Show the contents of the accumulator and the CY flag after the execution of the instruction RLCA.

Mnemonics: RLCA      Hex Code: 07



**RL r:** ROTATE EACH BIT LEFT IN REGISTER OR MEMORY THROUGH CARRY

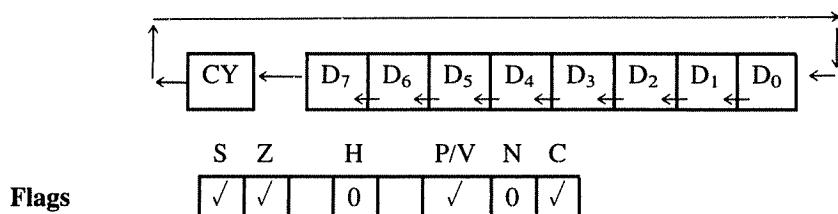
**RL (m):**

**Opcode Operand Bytes MC /T-States**

**Hex Codes**

A	B	C	D	E	H	L
RL	r	2	2 /8(4,4)	CB	17	10 11 12 13 14 15
RL	(HL)	2	4 /15(4,4,4,3)	CB	16	
RL	(IX + d)	4	6 /23(4,4,3,5,4,3)	DD	CB	d 16
RLC	(IY + d)			FD	CB	d 16

**Description** Each bit in the specified register or memory is rotated left by one position through carry. Bit D<sub>7</sub> is placed into the CY flag and the CY flag is placed into bit D<sub>0</sub>. The memory address is specified using either HL or an index register.



**RLC r:** ROTATE EACH BIT LEFT IN REGISTER OR MEMORY

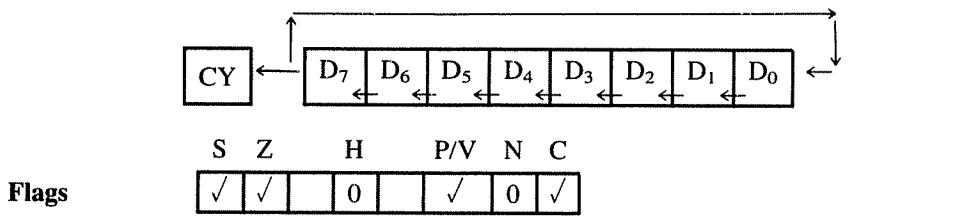
**RLC (m):**

**Opcode Operand Bytes MC /T-States**

**Hex Codes**

RLC	r	2	2 /8(4,4)	A	B	C	D	E	H	L
				CB	07	00	01	02	03	04 05
RLC	(HL)	2	4 /15(4,4,4,3)						CB	06
RLC	(IX + d)	4	6 /23(4,4,3,5,4,3)	DD	CB	d	06			
RLC	(IY + d)			FD	CB	d	06			

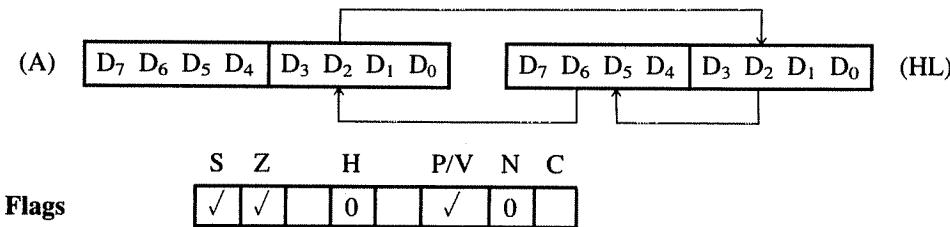
**Description** Each bit in the specified register or memory is rotated left by one position. Bit D<sub>7</sub> is placed into the position of bit D<sub>0</sub> and the CY flag is modified according to bit D<sub>7</sub>. The memory address is specified using either HL or an index register.



### RLD: ROTATE LEFT BCD DIGIT BETWEEN ACCUMULATOR AND MEMORY

Opcode	Operand	Bytes	MC /T-States	Hex Code
RLD		2	5 /18(4,4,3,4,3)	ED 6F

**Description** The instruction shifts 4-bit digits between memory and the accumulator as shown below. The four low-order bits (D<sub>3</sub>-D<sub>0</sub>) in the memory location indicated by HL register are shifted left, bits D<sub>7</sub>-D<sub>4</sub> are copied into the low-order bits of the accumulator, and bits D<sub>3</sub>-D<sub>0</sub> of the accumulator are copied into bits D<sub>3</sub>-D<sub>0</sub> of the memory.



**Example** The accumulator contains 67<sub>H</sub> and the memory location 2050<sub>H</sub> contains 92<sub>H</sub>. If the HL register holds the address 2050<sub>H</sub>, show the contents of the memory location and the accumulator after the execution of the instruction RLD.

Mnemonics: RLD      Hex Code: ED 6F

Contents Before  
Instruction      A 

6	7
---	---

9	2
---	---

 2050

Contents After  
Instruction      A 

6	9
---	---

2	7
---	---

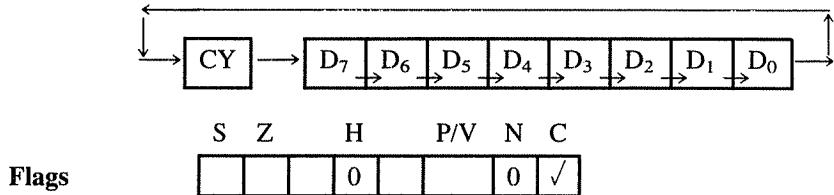
 2050

---

### RRA: ROTATE ACCUMULATOR RIGHT THROUGH CARRY

Opcode	Operand	Bytes	MC /T-States	Hex Code
RRA		1	1 /4	1F

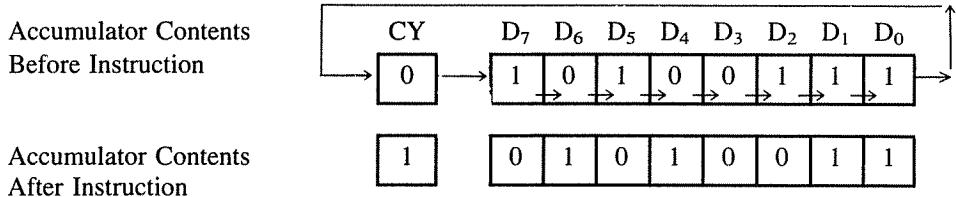
**Description** Each bit in the accumulator is rotated right by one position through the Carry flag. Bit D<sub>0</sub> is placed into the CY position, and the CY flag is placed into bit D<sub>7</sub>.



The CY flag is modified according to bit D<sub>0</sub>.

**Example** The accumulator contains A7<sub>H</sub> and the CY flag is reset. Show the contents of the accumulator and the CY flag after the execution of the instruction RRA.

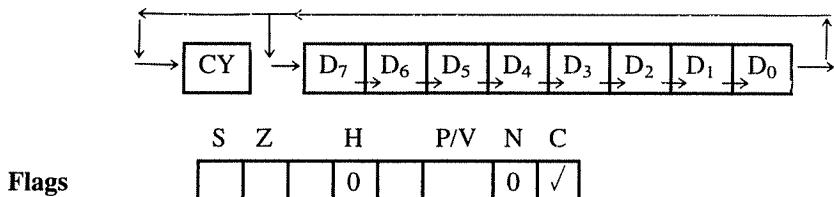
Mnemonics: RRA      Hex Code: 1F



#### RRCA: ROTATE ACCUMULATOR RIGHT

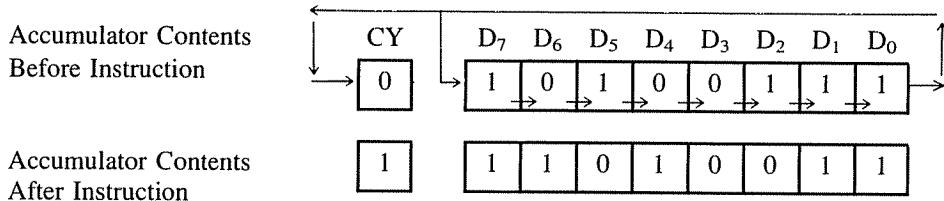
Opcode	Operand	Bytes	MC /T-States	Hex Code
RRCA		1	1 /4	0F

**Description** Each bit in the accumulator is rotated right by one position. Bit D<sub>0</sub> is placed in the position of bit D<sub>7</sub> and the CY flag is also modified according to bit D<sub>0</sub>.



**Example** The accumulator contains A7<sub>H</sub> and the CY flag is reset. Show the contents of the accumulator and the CY flag after the execution of the instruction RRCA.

Mnemonics: RRCA      Hex Code: 0F

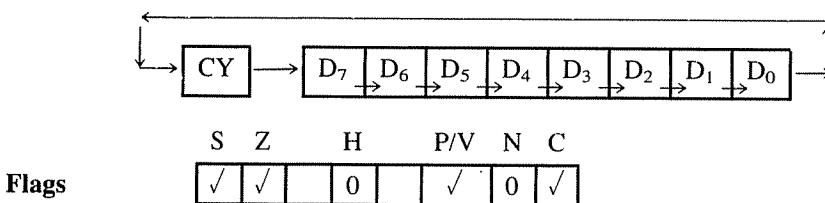


**RR r:** ROTATE EACH BIT RIGHT IN REGISTER OR MEMORY  
THROUGH CARRY

**RR (m):**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
RR	r	2	2 /8(4,4)	A B C D E H L CB 1F 18 19 1A 1B 1C 1D
RR	(HL)	2	4 /15(4,4,4,3)	CB 1E
RR	(IX + d)	4	6 /23 (4,4,3,5,4,3)	DD CB d 1E
RR	(IY + d)			FD CB d 1E

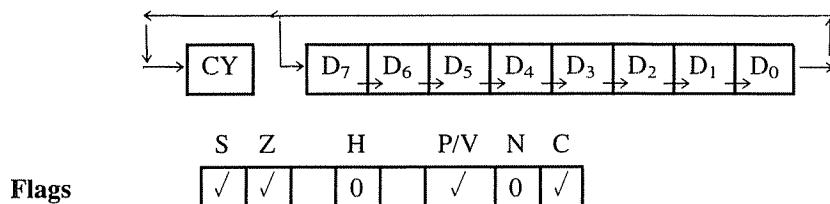
**Description** Each bit in the specified register or memory is rotated right by one position through carry. Bit D<sub>0</sub> is placed into the CY flag and the CY flag is placed into bit D<sub>7</sub>. The memory address is specified using either HL or an index register.



**RRC r:** ROTATE EACH BIT RIGHT IN REGISTER OR MEMORY  
**RRC (m):**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
RRC	r	2	2 /8(4,4)	A B C D E H L CB 0F 08 09 0A 0B 0C 0D
RRC	(HL)	2	4 /15(4,4,4,3)	CB 0E
RRC	(IX + d)	4	6 /23 (4,4,3,5,4,3)	DD CB d 0E
RRC	(IY + d)			FD CB d 0E

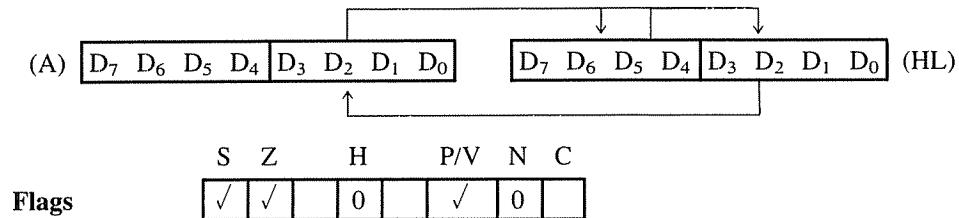
**Description** Each bit in the specified register or memory is rotated right by one position. Bit D<sub>0</sub> is placed into the position of bit D<sub>7</sub>, and the CY flag is modified according to bit D<sub>0</sub>. The memory address is specified using either HL or an index register.



#### RRD: ROTATE RIGHT BCD DIGIT BETWEEN ACCUMULATOR AND MEMORY

Opcode	Operand	Bytes	MC /T-States	Hex Code
RRD		2	5 /18(4,4,3,4,3)	ED 67

**Description** The instruction shifts 4-bit digits between memory and the accumulator as shown below. The four high-order bits (D<sub>7</sub>–D<sub>4</sub>) in the memory location indicated by HL are shifted right, bits D<sub>3</sub>–D<sub>0</sub> are copied into low-order bits of the accumulator, and bits D<sub>3</sub>–D<sub>0</sub> of the accumulator are copied into bits D<sub>7</sub>–D<sub>4</sub> of the memory.



**Example** The accumulator contains 67<sub>H</sub> and the memory location 2050<sub>H</sub> contains 92<sub>H</sub>. If the HL register holds the address 2050<sub>H</sub>, show the contents of the memory location and the accumulator after the execution of the instruction RRD.

Mnemonics: RRD      Hex Code: ED 67

Contents Before  
Instruction      A [6] [7] [9] [2] 2050

Contents After  
Instruction      A [6] [2] [7] [9] 2050

**RST: RESTART**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
RST	0 to 7	1	3 /11(5,3,3)	As shown below
Opcode/Operand		Restart Address (Hex)	Hex Code	
	RST 00H	0000	C7	
	RST 08H	0008	CF	
	RST 10H	0010	D7	
	RST 18H	0018	DF	
	RST 20H	0020	E7	
	RST 28H	0028	EF	
	RST 30H	0030	F7	
	RST 38H	0038	FF	

**Description** The RST instructions are 1-byte call instructions to one of eight memory locations on page 0. The instructions are generally used in conjunction with interrupts and inserted using external hardware. These instructions are also used in a program as software instructions to set up breakpoints or to transfer program execution to one of the eight memory locations.

**Flags** No flags are affected.

**SBC A, r:** SUBTRACT REGISTER AND BORROW FROM ACCUMULATOR

**SBC A, 8-BIT:** SUBTRACT 8-BIT AND BORROW FROM ACCUMULATOR

**SBC A, (m):** SUBTRACT MEMORY AND BORROW FROM ACCUMULATOR

Opcode	Operand	Bytes	MC /T-States	Hex Codes
SBC	A, r	1	1 /4	A B C D E H L 9F 98 99 9A 9B 9C 9D
SBC	A, 8-Bit	2	2 /7(4,3)	DE 8-Bit
SBC	A, (HL)	1	2 /7(4,3)	9E
SBC	A, (IX + d)	3	5 /19(4,4,3,5,3)	DD 9E d
SBC	A, (IY + d)			FD 9E d

**Description** The contents of the operand (register, 8-bit data, memory) and the Carry flag CY (as borrow) are subtracted from the contents of the accumulator, and the result is placed in the accumulator.

	S	Z	H	P/V	N	C
Flags	✓	✓		✓	1	✓

**Example** The register BC contains  $7498_H$  and the register DE contains  $19A1_H$ . Subtract (DE) from (BC) and store the result in BC.

**Step 1:** Copy (C) into A and subtract (E) using SUB instruction. The Z80 performs this subtraction in 2's complement.

$$\begin{array}{r} (B) \rightarrow (A): 98_H \\ - (E): A1_H \\ \hline (A): 1/F7_H \end{array}$$

**Step 2:** Save the result  $F7_H$  in register C.

**Step 3:** Copy (B) into A and subtract (D) using SBC instruction to account for the borrow of the previous result. The Z80 performs this subtraction in 2's complement.

$$\begin{array}{r} (B) \rightarrow (A): 74_H \\ (D): 19_H \\ \hline \text{Borrow: } 1 \\ 5A_H \end{array}$$

**Step 4:** Save the result ( $5A_H$ ) in B. The SBC instruction resets the previous CY flag.

**Comments** The instruction is generally used in 16-bit subtraction or multi-byte subtraction; the borrow generated by  $D_7$  is subtracted from bit  $D_8$  in the next subtraction. This instruction should not be used to account for carries (borrows) generated in subtracting 8-bit numbers.

#### SBC HL, rp: SUBTRACT REGISTER PAIR AND BORROW FROM HL

Opcode	Operand	Bytes	MC /T-States	Hex Codes
--------	---------	-------	--------------	-----------

SBC	HL, rp	2	4 /15(4,4,4,3)	BC DE HL SP ED 42 52 62 72
-----	--------	---	----------------	-------------------------------

**Description** The contents of the operand (BC, DE, HL, or SP) and the Carry flag (as borrow) are subtracted from the contents of the HL register, and the result is stored in the HL register.

	S	Z	H	P/V	N	C
Flags	✓	✓	?	✓	1	✓

**Example** The HL register contains  $F850_H$ , the DE register contains  $8200_H$ , and the Carry flag CY is set. Subtract the contents of DE from HL with borrow.

Mnemonics: SBC HL, DE      Hex Code: ED 52

Subtraction with Borrow					Register Contents After Instruction		
(HL): 1111	1000	0101	0000	(F850) <sub>H</sub>	D	82	00
(DE): 1000	0010	0000	0000	(8200) <sub>H</sub>	H	76	4F
CY :				1			
0/0111	0110	0100	1111	(764F) <sub>H</sub>			

**SCF: SET CARRY FLAG**

Opcode	Operand	Bytes	MC /T-States	Hex Code
SCF		1	1 /4	37

**Description** The Carry flag (C) is set to 1.

Flags	S	Z	H	P/V	N	C
			0		0	1

**SET b, r:** SET BIT IN REGISTER OR MEMORY

**SET b, (m):**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
SET	b, r	2	4 /8(4,4)	<i>Register</i>
				(Bit) A B C D E H L
				CB (0) C7 C0 C1 C2 C3 C4 C5
				(1) CF C8 C9 CA CB CC CD
				(2) D7 D0 D1 D2 D3 D4 D5
				(3) DF D8 D9 DA DB DC DD
				(4) E7 E0 E1 E2 E3 E4 E5
				(5) EF E8 E9 EA EF EC ED
				(6) F7 F0 F1 F2 F3 F4 F5
				(7) FF F8 F9 FA FB FC FD
				<i>Bit</i>
				0 1 2 3 4 5 6 7
SET	b, (HL)	2	4 /15 (4,4,4,3)	[CB C6 CE D6 DE E6 EE F6 FE]
SET	b, (IX + d)	4	6/23 (4,4,3,5,4,3)	DD CB d [C6 CE D6 DE E6 EE F6 FE]
SET	b, (IY + d)	4		FD CB d [C6 CE D6 DE E6 EE F6 FE]

**Description** The specified bit is set in a register or in memory. The values of b (0–7) correspond to bits D<sub>0</sub>–D<sub>7</sub>.

**Flags** No flags are affected.

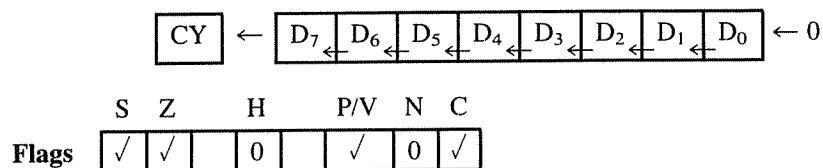
**Example** Write instructions to set bit 6 in register A and bit 0 in memory 2055<sub>H</sub>. Assume that HL is already loaded with the address 2055<sub>H</sub>.

Mnemonics	Hex Code
SET 6, A	CB F7
SET 0, (HL)	CB C6

**SLA r:** ARITHMETIC SHIFT LEFT IN REGISTER OR MEMORY**SLA (m):**

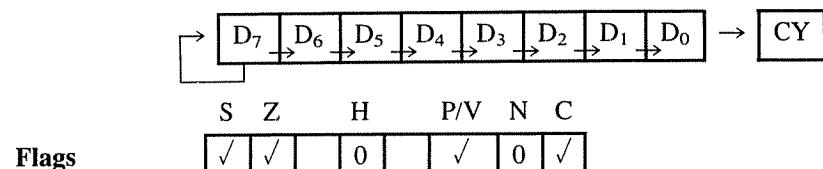
Opcode	Operand	Bytes	MC /T-States	Hex Codes							
				A	B	C	D	E	H	L	
SLA	r	2	2 /8(4,4)	CB	27	20	21	22	23	24	25
SLA	(HL)	2	4 /15 (4,4,4,3)			CB	26				
SLA	(IX + d)	4	6 /23 (4,4,3,5,4,3)		DD	CB	d	16			
SLA	(IY + d)				FD	CB	d	16			

**Description** Each bit in the specified register or memory is arithmetically shifted left by one position. Bit D<sub>7</sub> is placed into the CY flag and 0 is placed into bit D<sub>0</sub>. The memory address is specified using either HL or an index register.

**SRA r:** ARITHMETIC SHIFT RIGHT IN REGISTER OR MEMORY**SRA (m):**

Opcode	Operand	Bytes	MC /T-States	Hex Codes							
				A	B	C	D	E	H	L	
SRA	r	2	2 /8(4,4)	CB	2F	28	29	2A	2B	2C	2D
SRA	(HL)	2	4 /15 (4,4,4,3)			CB	2E				
SRA	(IX + d)	4	6 /23 (4,4,3,5,4,3)		DD	CB	d	2E			
SRA	(IY + d)				FD	CB	d	2E			

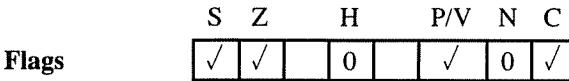
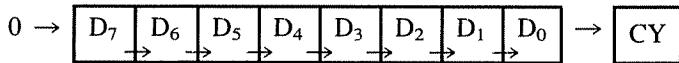
**Description** Each bit in the specified register or memory is arithmetically shifted right by one position. Bit D<sub>0</sub> is placed into the CY flag and bit D<sub>7</sub> remains unchanged. The memory address is specified using either HL or an index register.



**SRL r: LOGICAL SHIFT RIGHT IN REGISTER OR MEMORY****SRL (m):**

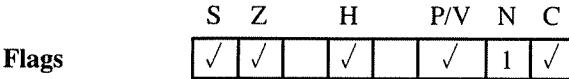
Opcode	Operand	Bytes	MC /T-States	Hex Codes
SRL	r	2	2 /8(4,4)	A B C D E H L CB 3F 38 39 3A 3B 3C 3D
SRL	(HL)	2	4 /15(4,4,4,3)	CB 3E
SRL	(IX + d)	4	6 /23 (4,4,3,5,4,3)	DD CB d 3E
SRL	(IY + d)			FD CB d 3E

**Description** Each bit in the specified register or memory is logically shifted right by one position. Bit D<sub>0</sub> is placed into the CY flag and 0 is placed into bit D<sub>7</sub>. The memory address is specified using either HL or an index register.

**SUB r: SUBTRACT REGISTER FROM ACCUMULATOR****SUB 8-BIT: SUBTRACT 8-BIT FROM ACCUMULATOR****SUB (m): SUBTRACT MEMORY FROM ACCUMULATOR**

Opcode	Operand	Bytes	MC /T-States	Hex Codes
SUB	r	1	1 /4	A B C D E H L 97 90 91 92 93 94 95
SUB	8-Bit	2	2 /7(4,3)	D6 8-Bit
SUB	(HL)	1	2 /7(4,3)	96
SUB	(IX + d)	3	5 /19(4,4,3,5,3)	DD 96 d
	(IY + d)			FD 96 d

**Description** The contents of the operand (register, 8-bit, or memory) are subtracted from the contents of the accumulator, and the result is stored in the accumulator.



**Example** Register B has 47<sub>H</sub> and the accumulator has 61<sub>H</sub>. Subtract (B) from (A).

Mnemonics: SUB B Hex Code: 90

Subtraction					
(A)	0	1	1	0	0 0 0 1 (61 <sub>H</sub> )
		+			
(B)	1	0	1	1	1 0 0 1 (2's comp. of 47 <sub>H</sub> )
(A)	1	0	0	0 1	1 0 1 0 Complement
		CY			CY
	CY	0	0	0 1	1 0 1 0 (1A <sub>H</sub> )

Register Contents After Execution					
	S	Z	H	VNC	
A		1A	0 0 1	0 1 0	F
B		47		X	C

**Example** The accumulator contains the byte 76<sub>H</sub>, and the HL pair contains 2050<sub>H</sub>. Subtract the byte A7<sub>H</sub> which is stored in memory location 2050<sub>H</sub> from the contents of the accumulator.

Mnemonics: SUB (HL)      Hex Code: 96					
Subtraction					
(A)	0	1	1	1	0 (76 <sub>H</sub> )
		+			
2050 (Mem)	0	1	0	1	1 0 0 1 (2's comp. of A7 <sub>H</sub> )
	1	1	0	0	1 1 1 1 (Complement CY)
		CY	1	1	0 0 1 1 CF

Register Contents After Execution					
	S	Z	H	VNC	
A		CF	1 0	1	1 1 1
H		20			50 L

**XOR r:** EXCLUSIVELY OR REGISTER, 8-BIT, OR MEMORY WITH ACCUMULATOR

**XOR 8-bit:**

**XOR (m):**

Opcode	Operand	Bytes	MC /T-States	Hex Codes							
				A	B	C	D	E	H	L	
XOR	r	1	1 /4								
XOR	8-Bit	2	2 /7(4,3)	AF	A8	A9	AA	AB	AC	AD	EE 8-bit
XOR	(HL)	1	2 /7(4,3)								AE
XOR	(IX + d)	3	5 /19(4,4,3,5,3)	DD	AE	d					FD AE d
	(IY + d)										

**Description** The contents of register, memory, or an 8-bit are exclusively ORed with the contents of the accumulator. The memory address is specified by the contents of the HL register or an index register with a displacement byte d.

	S	Z	H	P/V	N	C
Flags	✓	✓	0		✓	0

**Example** The contents of the accumulator and register B are  $54_H$  and  $96_H$  respectively. Exclusively OR (B) with (A) and show the flags and the contents of each register after the operation.

Mnemonics: XOR B      Hex Code: A8

Exclusive OR						
(A)	1	0	0	1	0	$1\ 1\ 1\ 0$ ( $96_H$ )
(B)	0	1	0	1	0	0
(A)	1	1	0	0	0	1

Register Contents After Instruction

	S	Z	H	PNC
A	C2	1	0	0
B	54		X	

**Example** Write mnemonics to exclusive OR the contents of memory location  $2070_H$  with the contents of the accumulator assuming index register IY contains address  $2000_H$ .

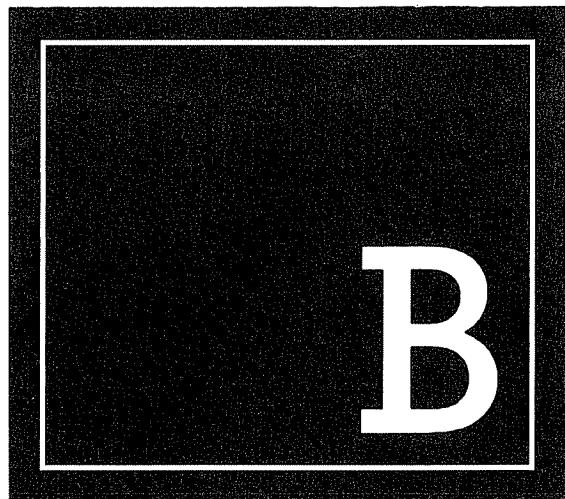
Mnemonics: XOR (IY + 70H)      Hex Code: FD AE 70



# Number Systems

Computers communicate and operate in binary digits 0 and 1; on the other hand, human beings generally use the decimal system with ten digits, from 0–9. Other number systems are also used, such as octal with eight digits, from 0–7, and hexadecimal (Hex) system with digits from 0–15. In the hexadecimal system, digits 10 through 15 are designated as A through F, respectively, to avoid confusion with the decimal numbers, 10 to 15.

A positional scheme is usually used to represent a number in any of the number systems. This means that each digit will have its value according to its position in a number. The number of digits in a position is also referred to as the base. For example, the binary system has base 2, the decimal system has base 10, and the hexadecimal system has base 16.



## B.1 NUMBER CONVERSION

A number in any base system can be represented in a generalized format as follows:

$$N = A_n B^n + A_{n-1} B^{n-1} + \dots + A_1 B^1 + A_0 B^0$$

N = number, B = base, A = any digit in that base

For example, number 154 can be represented in various number systems as follows:

$$\text{Decimal: } 154 = 1 \times 10^2 + 5 \times 10^1 + 4 \times 10^0 = 154$$

$$\begin{aligned} \text{Octal: } 232 &= 2 \times 8^2 + 3 \times 8^1 + 2 \times 8^0 \\ &= 128 + 24 + 2 = 154 \end{aligned}$$

$$\begin{aligned} \text{Hexadecimal: } 9A &= 9 \times 16^1 + A \times 16^0 \\ &= 144 + 10 = 154 \end{aligned}$$

$$\begin{aligned} \text{Binary: } 10011010 &= 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\ &= 128 + 0 + 0 + 16 + 8 + 0 + 2 + 0 = 154 \end{aligned}$$

The above example also shows how to convert a given number in any system into its decimal equivalent.

**CONVERSION TABLE: DECIMAL, HEXADECIMAL, BINARY, AND OCTAL**

Decimal	Hex	Binary	Octal
0	0	0000	00
1	1	0001	01
2	2	0010	02
3	3	0011	03
4	4	0100	04
5	5	0101	05
6	6	0110	06
7	7	0111	07
8	8	1000	10
9	9	1001	11
10	A	1010	12
11	B	1011	13
12	C	1100	14
13	D	1101	15
14	E	1110	16
15	F	1111	17

### HOW TO CONVERT A NUMBER FROM BINARY INTO HEXADECIMAL AND OCTAL

---

**Example** Convert the binary number 1 0 0 1 1 0 1 0 into its Hex and octal equivalents.

#### Hexadecimal

**Step 1:** Starting from the right (LSB) arrange the binary digits in groups of four.

1 0 0 1    1 0 1 0

**Step 2:** Convert each group into its equivalent Hex number.

9 A

#### Octal

**Step 1:** Starting from the right (LSB) arrange the binary digits in groups of three.

1 0 0 1 1 0 1 0

**Step 2:** Convert each group into its equivalent octal number.

2 3 2

---

## 2'S COMPLEMENT AND ARITHMETIC OPERATIONS

B.2

The Z80 microprocessor performs the subtraction of two binary numbers using the 2's complement method. In digital logic circuits, it is easier to design a circuit to add numbers than to design a circuit to subtract numbers. The 2's complement of a binary number is equivalent to its negative number; thus by adding the complement of the subtrahend (the number to be subtracted) to the minuend, a subtraction can be performed. The method of 2's complement is explained below with the examples from the decimal number system.

### DECIMAL SUBTRACTION

Subtract the following two decimal numbers using the borrow method and the 10's complement method: (52 – 23)

Example  
B.1

Minuend:  $52 = 5 \times 10 + 2$

Subtrahend:  $23 = 2 \times 10 + 3$

Borrow  
Method

**Step 1:** To subtract 3 from 2, 10 must be borrowed from the second place of the minuend.

$$52 = 4 \times 10 + 12$$

**Step 2:** The subtraction of the digits in the first place and the second place is as follows.

$$\begin{array}{r} 52 \\ -23 \\ \hline 2 \times 10 + 9 = 29 \end{array}$$

**Step 1:** Find 9's complement of the subtrahend (23), meaning subtract each digit of the subtrahend from 9.

10's  
Complement  
Method

$$\begin{array}{r} 9 \ 9 \\ -2 \ 3 \\ \hline 7 \ 6 \end{array}$$

**Step 2:** Add 1 to the 9's complement to find the 10's complement of the subtrahend.

$$\begin{array}{r} 76 \\ + 1 \\ \hline 77 \end{array}$$

The reason to find the 9's complement is to demonstrate a similar procedure to find the 2's complement of a binary number. However, in reality, the 10's complement of 23 is equivalent to subtracting 23 from 100.

**Step 3:** Add 10's complement of the subtrahend (77) to the minuend (52) to subtract 23 from 52.

$$\begin{array}{r} \text{10's complement of 23: } 77 \\ \text{Minuend: } + 52 \\ \hline 1 \ 29 = 29 \text{ (By dropping the most significant digit)} \end{array}$$

The elimination of the most significant bit is equivalent to subtracting 100 from the sum. This is necessary to compensate for the 100 that was added to find the 10's complement of 23.

**Example  
B.2**

Perform the subtraction of the following two numbers using the borrow method and the 10's complement method:  $23 - 52$ .

**Borrow  
Method**

$$\begin{array}{r} \text{Minuend: } 2 \ 3 \\ \text{Subtrahend: } 5 \ 2 \end{array}$$

**Step 1:** The subtraction of the digits in the first place results in:  $3 - 2 = 1$ .

**Step 2:** To subtract the digits in the second place, a borrow is required from the third place. Assuming the borrow is available from the third place, the digit 5 can be subtracted from 2 as follows:

$$\begin{array}{r} 1 \ 2 \\ - 5 \\ \hline 1 \ 7 \end{array} \text{ (the nonexistent borrow is shown with the bar)}$$

$$\begin{array}{r} \text{Result: } 23 \\ - 52 \\ \hline 1 \ 71 \end{array}$$

The same result is obtained with the 10's complement method, as shown below.

**10's  
Complement  
Method**

**Step 1:** Find the 9's complement of the subtrahend (52).

$$\begin{array}{r} \text{9's complement of 52: } 9 \ 9 \\ - 5 \ 2 \\ \hline 4 \ 7 \end{array}$$

**Step 2:** Add 1 to the 9's complement to find 10's complement:  $47 + 1 = 48$

**Step 3:** Add the 10's complement of the subtrahend to the minuend.

$$\begin{array}{r} \text{10's complement of 52: } 48 \\ \text{Minuend: } 23 \\ \hline 71 \end{array} \text{ (this is negative 29, expressed in 10's complement)}$$

By examining these two examples, the following conclusions can be drawn, and these conclusions can be used for any number system.

1. The complement of a number is its equivalent negative number.
  2. A number can be subtracted by using its complement.
  3. The sum of a number and its complement results in 0 if the most significant digit of the sum is ignored.
  4. When the subtrahend is larger than the minuend, the result of the 10's complement method is negative, and it is expressed in terms of 10's complement. The same result can be obtained by borrowing a digit from the most significant position.

## **PROCEDURE TO FIND 2'S COMPLEMENT OF A BINARY NUMBER**

**Step 1:** Find 1's complement. This amounts to replacing 0 by 1 and 1 by 0.

**Step 2:** To find 2's complement, add 1 to the 1's complement. This is similar to the procedure of 10's complement.

Find the 2's complement of the binary number:

### Example B.3

0 0 0 1 1 1 0 0 (1C<sub>H</sub> or 28<sub>10</sub>)

**Step 1:** Find 1's complement, meaning replace 0 with 1 and 1 with 0.

$$1's \text{ complement} = 1 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1$$

**Step 2:** Add 1                            +

2's complement = 1 1 1 0 0 1 0 0

**2.5. Complement**      1      1      3      3      1      3

By examining the result of the example, the following rule can be stated to find the 2's complement of a binary number, instead of the above procedure of the 1's complement.

**Rule 1:** Start at the LSB of a given number, and check all the bits to the left. Keep all the bits as they are up to and including the least significant 1.

**Rule 2:** After the first 1, replace all 0's with 1's and 1's with 0's.

These rules can be applied to the given binary number ( $1C_H$ ) as illustrated below:

Binary Number:  0 0 0 1 1      1 0 0 ← Start Here

↓                    ↓

Replace 0 with 1      Keep as they are  
and 1 with 0

2's complement: 1 1 1 0 0      1 0 0

The 2's complement of the number can be verified by adding the complement to the original number as follows, and the sum should be 0:

$$\begin{array}{r}
 \text{Binary Number: } 0 \ 0 \ 0 \ 1 \quad 1 \ 1 \ 0 \ 0 \\
 \text{2's Complement: } 1 \ 1 \ 1 \ 0 \quad 0 \ 1 \ 0 \ 0 \\
 \hline
 1 \quad 0 \ 0 \ 0 \ 0 \quad 0 \ 0 \ 0 \ 0 \quad (\text{ignore the MSB})
 \end{array}$$


---

### BINARY SUBTRACTION USING 2'S COMPLEMENT

The binary subtraction can be performed by using 2's complement method, and if the result is negative, it is expressed in terms of 2's complement.

---

#### Example B.4

Subtract  $32_H$  (0011 0010) from  $45_H$  (0100 0101).

$$\begin{array}{r}
 \text{Subtrahend: } 32_H = 0 \ 0 \ 1 \ 1 \quad 0 \ 0 \ 1 \ 0 \\
 \text{2's complement of } 32_H = 1 \ 1 \ 0 \ 0 \quad 1 \ 1 \ 1 \ 0 \\
 + \\
 \text{Minuend: } 45_H = 0 \ 1 \ 0 \ 0 \quad 0 \ 1 \ 0 \ 1 \\
 \text{CY} \quad 1 \quad 1 \\
 \hline
 0 \ 0 \ 0 \ 1 \quad 0 \ 0 \ 1 \ 1 = 13_H
 \end{array}$$


---

#### Example B.5

Subtract  $45_H$  (0100 0100) from  $32_H$  (0011 0010).

$$\begin{array}{r}
 \text{Subtrahend: } 45_H = 0 \ 1 \ 0 \ 0 \quad 0 \ 1 \ 0 \ 1 \\
 \text{2's complement of } 45_H = 1 \ 0 \ 1 \ 1 \quad 1 \ 0 \ 1 \ 1 \\
 + \\
 \text{Minuend: } 32_H = 0 \ 0 \ 1 \ 1 \quad 0 \ 0 \ 1 \ 0 \\
 \text{CY} = 1 \ 1 \\
 \hline
 1 \ 1 \ 1 \ 0 \quad 1 \ 1 \ 0 \ 1 = ED_H
 \end{array}$$


---

The result is negative and it is expressed in 2's complement. This can be verified by taking the 2's complement of the result; the 2's complement of the result should be  $13_H$  as in Example 4.

$$\begin{array}{r}
 \text{Result } ED_H = 1 \ 1 \ 1 \ 0 \quad 1 \ 1 \ 0 \ 1 \\
 \text{Two's complement of } ED_H = 0 \ 0 \ 0 \ 1 \quad 0 \ 0 \ 1 \ 1 = 13_H
 \end{array}$$


---

### SIGNED NUMBERS

To perform the arithmetic operations with signed numbers (positive and negative), the sign must be indicated as well as the magnitude of the number. In 8-bit microprocessors, bit  $D_7$  is used to indicate the sign of a number; 0 in  $D_7$  indicates a positive number and 1 indicates

a negative number. Bit D<sub>7</sub> can be used to indicate the sign of a number because:

1. The Z80 performs the subtraction of two numbers using 2's complement and, if the result is negative, it saves (shows) the result in the form of 2's complement.
2. 2's complements of all the 7-bit numbers have 1 in D<sub>7</sub>.

When a programmer uses bit D<sub>7</sub> to indicate the sign of a number, the magnitude of the number can be represented by seven bits (D<sub>6</sub>-D<sub>0</sub>). For example, number 74<sub>H</sub> is represented with sign as follows:

	D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>
+74 <sub>H</sub> =	0	1	1	1	0	1	0	0
-74 <sub>H</sub> =	1	0	0	0	1	1	0	0

↑                                   ↓

sign                              magnitude

(2's complement of 74<sub>H</sub>)

However, the microprocessor cannot differentiate between a positive number and a negative number. For example, in the above illustration, -74<sub>H</sub> can be interpreted as the unsigned positive number 8C<sub>H</sub> or the bit pattern. It is the responsibility of the programmer to provide the necessary interpretation.

### SUBTRACTION PROCESS IN THE Z80 MICROPROCESSOR

The Z80 performs the following operations when it subtracts (SUB) two binary numbers:

**Step 1:** Finds 1's complement of the subtrahend.

**Step 2:** Finds 2's complement of the subtrahend by adding 1 to the result of Step 1.

**Step 3:** Adds the 2's complement of the subtrahend to the minuend.

**Step 4:** Complements the CY flag.

These steps are internal to the microprocessor and invisible to the user; only the result is available to the user.

Show the internal steps performed by the microprocessor to subtract the following unsigned numbers:

**Example  
B.6**

a. FA<sub>H</sub> - 62<sub>H</sub>

b. 62<sub>H</sub> - FA<sub>H</sub>

a. Minuend: FA<sub>H</sub> = 1 1 1 1     1 0 1 0  
Subtrahend: 62<sub>H</sub> = 0 1 1 0     0 0 1 0

**Step 1:** 1's complement of  $62_H = \begin{array}{ccccccccc} 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \end{array}$

**Step 2:** Add 1       $\begin{array}{r} + \\ \hline 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{array}$

2's complement of  $62_H = \begin{array}{r} +1 \\ \hline 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \end{array}$

**Step 3:** Add minuend ( $FA_H$ )       $\begin{array}{r} +1 \\ \hline 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \end{array}$

**Step 4:** Complement CY       $\begin{array}{r} 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \end{array}$

Result:       $\begin{array}{r} 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \end{array} = 98_H$

Flags:      CY = 0, S = 1, Z = 0, V = 0.

b. Minuend:  $62_H = \begin{array}{ccccccccc} 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \end{array}$   
 Subtrahend:  $FA_H = \begin{array}{ccccccccc} 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \end{array}$

**Step 1:** 1's complement of  $FA_H = \begin{array}{ccccccccc} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{array}$

**Step 2:** Add 1       $\begin{array}{r} + \\ \hline 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{array}$

2's complement of  $FA_H = \begin{array}{r} +0 \\ \hline 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \end{array}$

**Step 3:** Add minuend ( $62_H$ )       $\begin{array}{r} +0 \\ \hline 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \end{array}$

**Step 4:** Complement CY       $\begin{array}{r} 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \end{array}$

Result:       $\begin{array}{r} 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \end{array} = 68_H$  (CY = 1)

Flags:      CY = 1, S = 0, Z = 0, V = 0

This result is negative and expressed in 2's complement of the magnitude.

#### Results

- $FA_H - 62_H = 98_H$  (positive), CY = 0, S = 1
- $62_H - FA_H = 68_H$  (negative), CY = 1, S = 0

These results and associated flags appear to be confusing. In Example B.6a, the result is positive but the sign flag indicates that it is negative. On the other hand, in Example B.6b, the result is negative but the sign flag indicates that it is positive. This confusion can be explained as follows:

1. This subtraction is concerned with the unsigned numbers; therefore, the sign flag is irrelevant. In signed arithmetic, the number  $FA_H$  is invalid because it is an 8-bit number.
2. The programmer can check whether the result indicates the true magnitude by checking the CY flag. If CY is reset, the result is positive, and if CY is set, the result is expressed in 2's complement.

#### Example B.7

In Example B.6a, assume that the numbers are signed numbers, and interpret the result.

*Minuend:  $FA_H$*

This is a negative number because  $D_7 = 1$ ; therefore, this must be represented in 2's complement. The magnitude of the number can be found by taking the 2's complement of

$FA_H$ :

$$\begin{aligned} FA_H &= 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \\ 2\text{'s complement of } FA_H &= 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \\ &= 06_H \text{ (magnitude)} \end{aligned}$$

*Subtrahend:*  $62_H$  (This is a positive number because  $D_7 = 0$ .)

The problem given in 6a can be represented as follows:

$$\begin{aligned} FA_H - 62_H &= (-06_H) - (+62_H) \\ &= -68_H \end{aligned}$$

The final result is  $-68_H$ , which will be in the form of its 2's complement:

$$\begin{aligned} -68_H &= -(0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0) \\ 2\text{'s complement of } 68_H &= 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \\ &= 98_H \end{aligned}$$

The final answer is the same as before; however, it will be interpreted as a negative number with the magnitude of  $68_H$ . When signed numbers are used in arithmetic operations, the sign flag will indicate the proper sign of the result.Add the following two positive numbers and interpret the sign flag:  $+41_H$ ,  $+54_H$ .Example  
B.8

$$\begin{array}{r} 41_H = 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \\ + \quad \quad \quad \quad \quad \quad \quad \quad \quad \\ 54_H = 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \\ \hline 95_H = 1 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \end{array} \quad S = 1, CY = 0, Z = 0, V = 1$$

This is an addition of two positive numbers. The sign flag indicates that the sum is negative. However, the overflow (P/V = 1) flag indicates that there is an overflow; therefore, the sum is inaccurate if these are signed numbers. If this had been the sum of two unsigned numbers, the sign flag would have no significance.

## MODULO-2 ARITHMETIC

B.3

Modulo-2 arithmetic is binary addition (or subtraction) without carries. There are no negative numbers, and the result can be either 0 or 1. An addition or subtraction of two numbers in Modulo-2 is similar to exclusive-ORing two logic functions as shown, and the results are the same.

$$\begin{array}{r} 1 \ 0 \ 1 \ 0 \\ + \ 1 \ 1 \ 1 \ 0 \\ \hline 0 \ 1 \ 0 \ 0 \end{array} \quad \begin{array}{r} 1 \ 0 \ 1 \ 0 \\ - \ 1 \ 1 \ 1 \ 0 \\ \hline 0 \ 1 \ 0 \ 0 \end{array}$$

Similarly, a division of two polynomials can be performed as follows. The example is from the Cyclic Redundancy Check (CRC—Section 15.13).

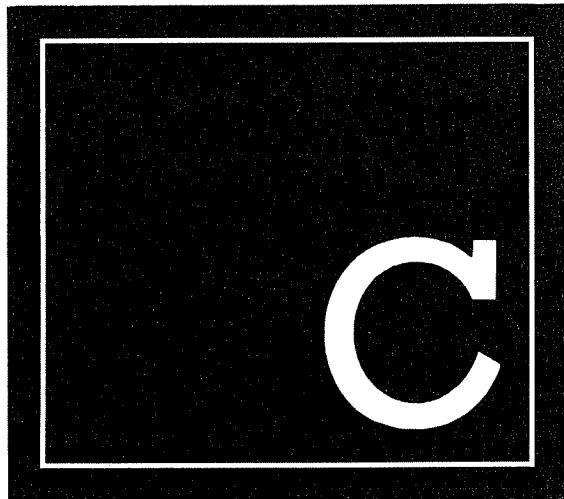
The modified polynomial for data bits 1000 1010:  $x^{10} + x^8 + x^4$

The generator polynomial:  $x^4 + 1$

$$\begin{array}{r}
 & x^6 + x^4 + x^2 \\
 x^4 + 1 \quad | & \overline{x^{10} + x^8 + x^4} \\
 & x^{10} \quad + x^6 \\
 & \hline
 & x^8 + x^6 + x^4 \\
 & x^8 \quad + x^4 \rightarrow \text{By multiplying } (x^4 + 1)(x^6) \\
 & \hline
 & x^6 \\
 & x^6 + x^2 \rightarrow \text{In Modulo-2, subtraction is} \\
 & \text{same as addition} \\
 & \hline
 & x^2 \rightarrow \text{By multiplying } (x^4 + 1)(x^4) \\
 & \hline
 & x^2 \rightarrow \text{By multiplying } (x^4 + 1)(x^2) \\
 & \hline
 & x^2 \rightarrow \text{Remainder}
 \end{array}$$

# American Standard Code for Information Interchange: ASCII Codes

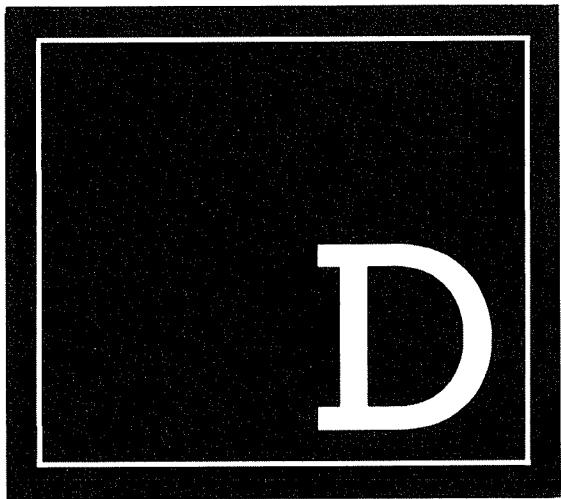
Graphic or Control	ASCII	(Hexadecimal)
NUL Null	00	
SOH Start of Heading	01	
STX Start of Text	02	
ETX End of Text	03	
EOT End of Transmission	04	
ENQ Enquiry	05	
ACK Acknowledge	06	
BEL Bell	07	
BS Backspace	08	
HT Horizontal Tabulation	09	
LF Line Feed	0A	
VT Vertical Tabulation	0B	
FF Form Feed	0C	
CR Carriage Return	0D	
SO Shift Out	0E	
SI Shift In	0F	
DLE Data Link Escape	10	
DC1 Device Control 1	11	
DC2 Device Control 2	12	
DC3 Device Control 3	13	
DC4 Device Control 4	14	
NAK Negative Acknowledge	15	
SYN Synchronous Idle	16	
ETB End of Transmission Block	17	
CAN Cancel	18	
EM End of Medium	19	
SUB Substitute	1A	
ESC Escape	1B	



Graphic or Control	ASCII	(Hexadecimal)
FS File Separator	1C	
GS Group Separator	1D	
RS Record Separator	1E	
US Unit Separator	1F	
SP Space	20	
	!	21
	"	22
	#	23
	\$	24
	%	25

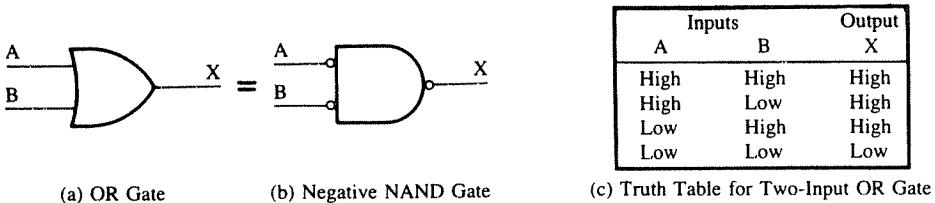
Graphic or Control	ASCII (Hexadecimal)	Graphic or Control	ASCII (Hexadecimal)
&	26	S	53
,	27	T	54
(	28	U	55
)	29	V	56
*	2A	W	57
+	2B	X	58
,	2C	Y	59
-	2D	Z	5A
.	2E	[	5B
/	2F	\	5C
0	30	]	5D
1	31	^	5E
2	32	-	5F
3	33	`	60
4	34	a	61
5	35	b	62
6	36	c	63
7	37	d	64
8	38	e	65
9	39	f	66
:	3A	g	67
;	3B	h	68
<	3C	i	69
=	3D	j	6A
>	3E	k	6B
?	3F	l	6C
@	40	m	6D
A	41	n	6E
B	42	o	6F
C	43	p	70
D	44	q	71
E	45	r	72
F	46	s	73
G	47	t	74
H	48	u	75
I	49	v	76
J	4A	w	77
K	4B	x	78
L	4C	y	79
M	4D	z	7A
N	4E	{	7B
O	4F		7C
P	50	}	7D
Q	51	~	7E
R	52	DEL Delete	7F

# Preferred Logic Symbols



## THE OR GATE AS THE NEGATIVE NAND GATE

The OR gate is normally represented as shown in Figure D.1(a). The truth table in Figure D.1(c) shows that when any one of the inputs is high, the output is high; and when both inputs are low, the output is low. In applications where the output required is active high, the symbol in Figure D.1(a) accurately represents the signal states. However, in some applications, the output required is *active low* when both inputs are low. For example, in Figure 3.9(a), the control signal MEMWR is generated when both inputs MREQ and WR are low. Therefore, it is preferable to represent the active states by the Negative NAND gate as shown in Figure D.1(b). Physically, the gate in Figure D.1(a) is the same as the



**FIGURE D.1**  
OR Gate Logic Symbols and Truth Table

gate in D.1(b); both are OR gates. However, they will be interpreted differently. In Figure D.1(a), the gate function should be read as follows: when input A *or* input B is high, the output goes high. In Figure D.1(b), the gate function should be read as follows: when input A *and* input B are low, the output goes low.

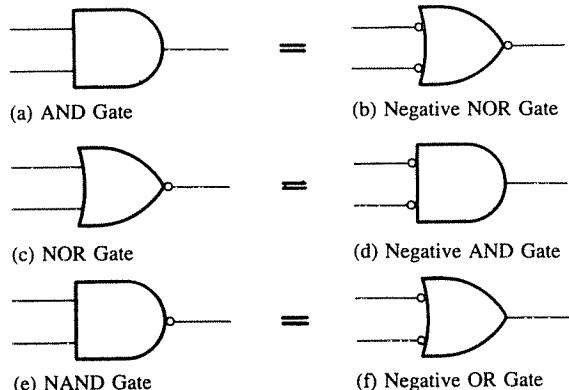
#### THE AND GATE AS THE NEGATIVE NOR GATE

The AND gate shown in Figure D.2(a) should be read as follows: when input A *and* input B are high, the output goes high. However, the equivalent gate shown in Figure D.2(b) should be read as follows: when input A *or* input B is low, the output goes low.

#### THE NOR GATE AS THE NEGATIVE AND GATE

The NOR gate in Figure D.2(c) should read as follows: when input A *or* input B is high, the output goes low. However, its equivalent gate shown in Figure D.2(d) should be read as follows: when input A *and* input B are low, the output goes high. Figure 5.4 shows the NOR gate (74LS02) connected as the Negative AND gate. It suggests that when the signal I/O write (IOWR) and the decoded address pulse are low, the output goes high and enables the flip-flop.

**FIGURE D.2**  
Logic Gates and Their Equivalent Symbols



**THE NAND GATE AS THE NEGATIVE OR GATE**

Figure D.2(e) shows the normal representation of the NAND gate and Figure D.2(f) shows its equivalent as the Negative OR gate.

**INVERTERS AND BUBBLE MATCHING**

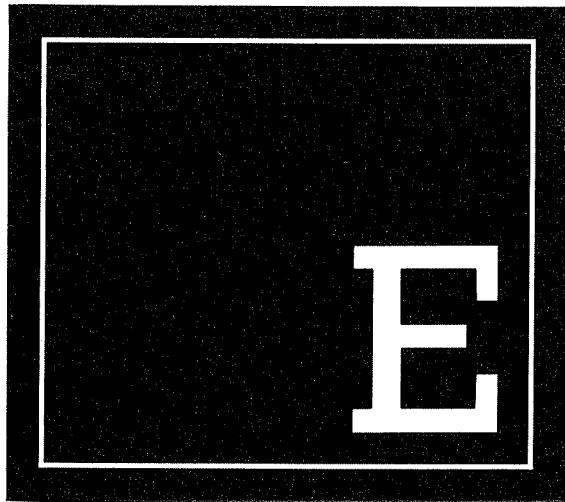
When the inverter is represented by its symbol, the bubble can be shown either in the front or in the back depending upon the active level of the input signal. For example, in Figure 2.10 the inverters to the 8-input NAND gate are shown with bubbles at the back. The bubble suggests that address lines  $A_{15}$ – $A_8$  should be at logic 0 to cause the output of the gate to go active low.

**SUMMARY**

A logic gate can be represented with different symbols. However, the symbol should be selected based on the active level of the signals. If the active level of the signals is represented, it is easy to interpret the gate function and it facilitates troubleshooting.



# The Micro-Trainer\*: Z80-Based Single-Board Microcomputer for Laboratory Use



## THE MICRO-TRAINER SYSTEM

The Micro-Trainer is a single-board microcomputer designed with the Z80 microprocessor. The trainer system is well designed for laboratory work in colleges and universities. It includes a Hex keyboard and seven-segment LED display. The user can enter a Z80 assembly language program (in Hex code) using the keyboard or by downloading from a host microcomputer such as an IBM PC and its compatibles as described in Chapter 7 of

---

\*The Micro-Trainer is available from CAMI Research, Arlington, MA, and information in this appendix is adapted from the Micro-Trainer manual. The main circuit board of the trainer was designed and built by the Multitech Industrial Corporation and is separately available as a training system under the trade name "Micro-Professor."

this book. The user can write Z80 assembly language programs using an editor, assemble it using a cross-assembler, and enter the code in R/W memory of the Micro-Trainer using the program called Downloader through a serial link; such a facility, called the ZAD system, is available from CAMI Research and can be added to the Micro-Trainer.

Figure E.1 shows the functional block diagram of the Micro-Trainer system. It includes the Z80 microprocessor, the 2532 4K-byte EPROM, the 6116 2K-byte R/W memory with the memory map from  $1800_H$  to  $1FFF_H$ , Hex keyboard, and six seven-segment LEDs for display. The keyboard enables the user to enter and store the Z80 assembly language program (Hex machine code) in R/W memory and execute the program. When a program (Hex machine code) is being entered, memory addresses and data are displayed by the LED display.

### KEYBOARD AND DISPLAY

The keyboard has 36 keys, of which four are directly connected to system hardware, and the remaining 32 keys are connected in the matrix format. The display includes six seven-segment LEDs, four for memory addresses and two for data bytes. The keyboard and the LED display are scanned, and they are connected in the format as shown in Figure 17.14 using three ports of the programmable device 8255; Port C is common to the digit driver of the display and the rows of the keyboard.

The keys are divided into two primary groups: 20 function keys and 16 Hex keys which can also be used for register examine functions (Figure E.1). The function group includes features such as decrementing a memory address, inserting or deleting bytes from the stored program, moving blocks of data from one area of memory to another, and setting and clearing breakpoints. We will illustrate some of the commonly used keys in the following example.

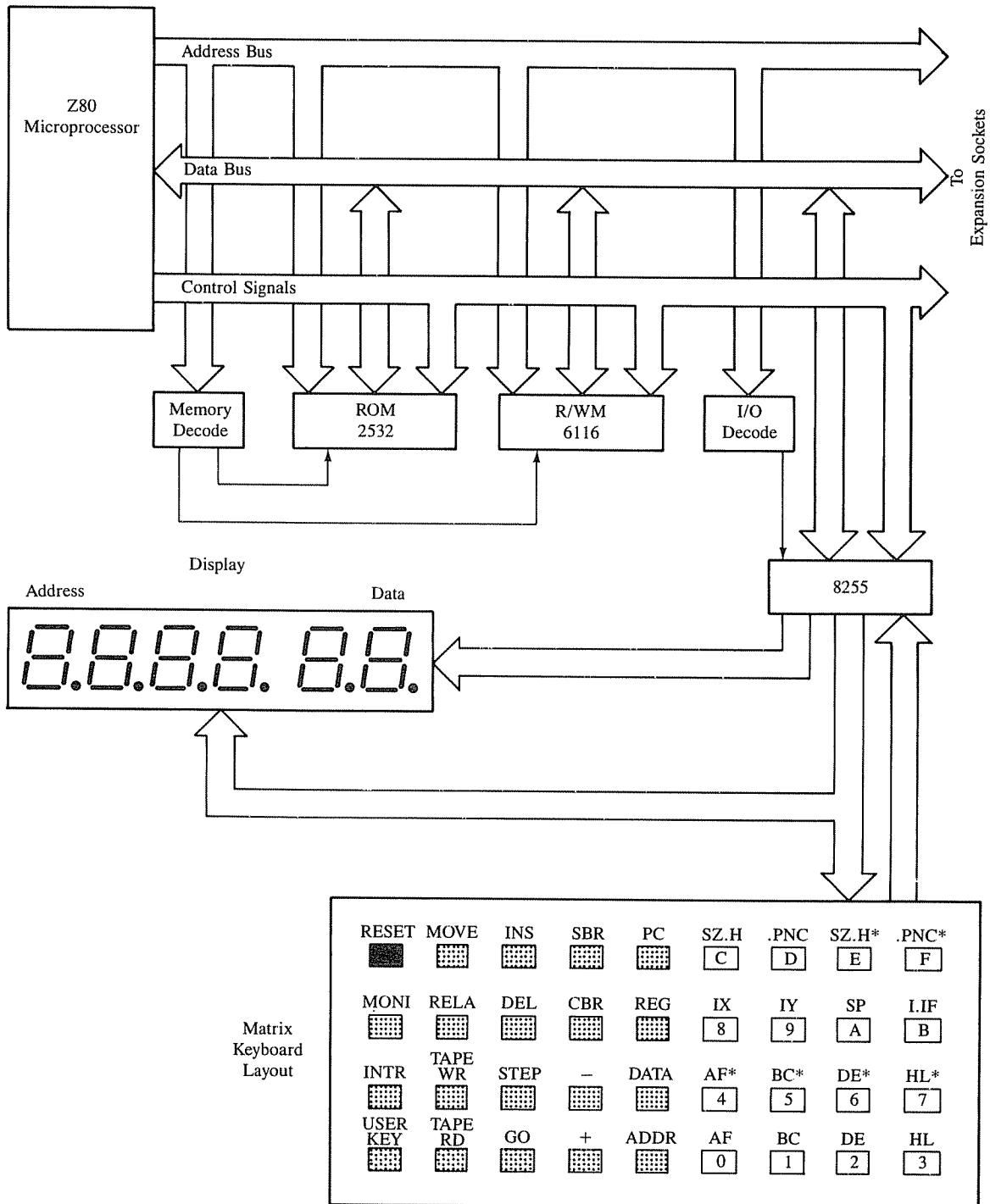
**Example  
E.1**

The following program loads two bytes,  $45_H$  and  $4F_H$ , into registers A and B, respectively, and adds the bytes. The sum is stored in the accumulator. Illustrate how to enter this program in R/W memory of the Micro-Trainer, starting at location  $1820_H$ , how to execute the program, and how to verify the result.

**Solution**

1. The steps in entering the program are as follows:

	<b>Key Sequence</b>	<b>Display at LED</b>	<b>Comments</b>
	<i>Press</i>	<i>Memory Field</i>	<i>Data Field</i>
	<b>RESET</b>	1800	X X ;Reset the system and display the first R/W memory location
	<b>ADDR</b>	....	;Address function—subsequent Hex keys will be interpreted as memory address

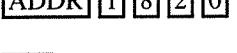


**FIGURE E.1**  
The Micro-Trainer Functional Block Diagram

SOURCE: Courtesy of CAMI Research Inc.

	1820	X X	;Enter address desired
	1820	..	;Data—subsequent Hex keys will be interpreted as data
	1820	3 E	;LD A, 45H—Load accumulator
	1821	X X	;Next memory location
	1821	4 5	;First data byte in A
	1822	X X	;Next location
	1822	0 6	;LD B, 4FH—Load register B
	1823	X X	;Next location
	1823	4 F	;Second byte in B
	1824	X X	;Next location
	1824	8 0	;ADD A, B—Add register B to A
	1825	X X	;Next location
	1825	7 6	;HALT—End of program

2. To execute the program, the steps are as follows:

	1800	X X	;Reset the system
	1820	3 E	;Enter the address where program begins
			;Execute the program

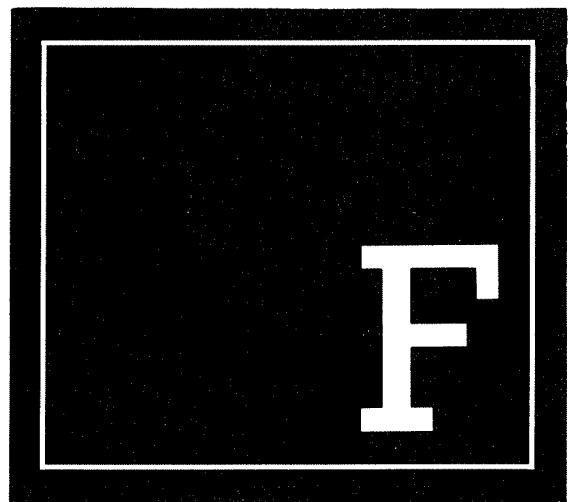
3. To examine the result, press the following keys:

	1800	X X	
	r E G -		;Examine register
	94 84	A F	;Display contents of accumulator and flags

4. The sum of the two bytes is  $94_{16}$ , which is displayed at the left, followed by the flag bits. This result sets Sign and Overflow flags; the flag byte is  $84_{16}$  as shown.

**Comment:** We selected  $1820_{16}$  as the starting location to demonstrate how to enter codes in any memory location. Ordinarily, a program is stored starting at location  $1800_{16}$ , and you need not use the ADDR key to specify the starting address.

# Z80 Instructional Summary



## Key:

n = 8-bit number      ( ) = contents as a pointer to memory or I/O

nn = 16-bit number

d = 7-bit displacement (express in 2's complement for backward displacement.)

e = 7-bit displacement in reference to program counter (express in 2's complement for backward displacement.)

Mnemonic	Hex	Mnemonic	Hex	Mnemonic	Hex
ADC A,A	8F	CALL C,nn	DC 16-bit	HALT	76
ADC A,B	88	CALL M,nn	FC 16-bit	IM 0	ED 46
ADC A,C	89	CALL NC,nn	D4 16-bit	IM 1	ED 56
ADC A,D	8A	CALL NZ,nn	C4 16-bit	IM 2	ED 5E
ADC A,E	8B	CALL P,nn	F4 16-bit	IN A,(n)	DB 8-bit
ADC A,H	8C	CALL PE,nn	EC 16-bit	IN A,(C)	ED 78
ADC A,L	8D	CALL PO,nn	E4 16-bit	IN B,(C)	ED 40
ADC A,n	CE 8-bit	CALL Z,nn	CC 16-bit	IN C,(C)	ED 48
ADC A,(HL)	8E	CCF	3F	IN D,(C)	ED 50
ADC A,(IX+d)	DD 8E d	CP A	BF	IN E,(C)	ED 58
ADC A,(IY+d)	FD 8E d	CP B	B8	IN F,(C)	ED 70
ADC HL,BC	ED 4A	CP C	B9	IN H,(C)	ED 60
ADC HL,DE	ED 5A	CP D	BA	IN L,(C)	ED 68
ADC HL,HL	ED 6A	CP E	BB	INC A	3C
ADC HL,SP	ED 7A	CP H	BC	INC B	04
ADD A,A	87	CP L	BD	INC BC	03
ADD A,B	80	CP n	FE 8-bit	INC C	0C
ADD A,C	81	CP (HL)	BE	INC D	14
ADD A,D	82	CP (IX+d)	DD BE d	INC DE	13,
ADD A,E	83	CP (IY+d)	FD BE d	INC E	1C
ADD A,H	84	CPD	ED A9	INC H	24
ADD A,L	85	CPDR	ED B9	INC HL	23
ADD A,n	C6 8-bit	CPI	ED A1	INC IX	DD 23
ADD A,(HL)	86	CPIR	ED B1	INC IY	FD 23
ADD A,(IX+d)	DD 86 d	CPL	2F	INC L	2C
ADD A,(IY+d)	FD 86 d	DAA	27	INC SP	33
ADD HL,BC	09	DEC A	3D	INC (HL)	34
ADD HL,DE	19	DEC B	05	INC (IX+d)	DD 34 d
ADD HL,HL	29	DEC BC	0B	INC (IY+d)	FD 34 d
ADD HL,SP	39	DEC C	0D	IND	ED AA
ADD IX,BC	DD 09	DEC D	15	INDR	ED BA
ADD IX,DE	DD 19	DEC DE	1B	INI	ED A2
ADD IX,IX	DD 29	DEC E	1D	INIR	ED B2
ADD IX,SP	DD 39	DEC H	25	JP nn	C3 16-bit
ADD IY,BC	FD 09	DEC HL	2B	JP (HL)	E9
ADD IY,DE	FD 19	DEC IX	DD 2B	JP (IX)	DD E9
ADD IY,IY	FD 29	DEC IY	FD 2B	JP (IY)	FD E9
ADD IY,SP	FD 39	DEC L	2D	JP C,nn	DA 16-bit
AND A	A7	DEC SP	3B	JP M,nn	FA 16-bit
AND B	A0	DEC (HL)	35	JP NC,nn	D2 16-bit
AND C	A1	DEC (IX+d)	DD 35 d	JP NZ,nn	C2 16-bit
AND D	A2	DEC (IY+d)	FD 35 d	JP P,nn	F2 16-bit
AND E	A3	DI	F3	JP PE,nn	EA 16-bit
AND H	A4	DJNZ e	10 e	JP PO,nn	E2 16-bit
AND L	A5	EI	FB	JP Z,nn	CA 16-bit
AND n	E6 8-bit	EX (SP),HL	E3	JR C,e	38 e
AND (HL)	A6	EX (SP),IX	DD E3	JR NC,e	30 e
AND (IX+d)	DD A6 d	EX (SP),IY	FD E3	JR NZ,e	20 e
AND (IY+d)	FD A6 d	EX AF,AF'	08	JR Z,e	28 e
BIT b,s	see pp. 568-69	EX DE,HL	EB	JR e	18 e
CALL nn	CD 16-bit	EXX	D9	LD A,A	7F
				LD A,B	78

Mnemonic	Hex	Mnemonic	Hex	Mnemonic	Hex
LD A,C	79	LD E,A	5F	LD (HL),D	72
LD A,D	7A	LD E,B	58	LD (HL),E	73
LD A,E	7B	LD E,C	59	LD (HL),H	74
LD A,H	7C	LD E,D	5A	LD (HL),L	75
LD A,I	ED 57	LD E,E	5B	LD (HL),n	36 8-bit
LD A,L	7D	LD E,H	5C	LD (IX+d),A	DD 77 d
LD A,n	3E 8-bit	LD E,L	5D	LD (IX+d),B	DD 70 d
LD A,R	ED 5F	LD E,n	1E 8-bit	LD (IX+d),C	DD 71 d
LD A,(BC)	0A	LD E,(HL)	5E	LD (IX+d),D	DD 72 d
LD A,(DE)	1A	LD E,(IX+d)	DD 5E d	LD (IX+d),E	DD 73 d
LD A,(HL)	7E	LD E,(IY+d)	FD 5E d	LD (IX+d),H	DD 74 d
LD A,(IX+d)	DD 7E d	LD H,A	67	LD (IX+d),L	DD 75 d
LD A,(IY+d)	FD 7E d	LD H,B	60	LD (IX+d),n	DD 36 d 8-bit
LD A,(nn)	3A 16-bit	LD H,C	61	LD (IY+d),A	FD 77 d
LD B,A	47	LD H,D	62	LD (IY+d),B	FD 70 d
LD B,B	40	LD H,E	63	LD (IY+d),C	FD 71 d
LD B,C	41	LD H,H	64	LD (IY+d),D	FD 72 d
LD B,D	42	LD H,L	65	LD (IY+d),E	FD 73 d
LD B,E	43	LD H,n	26 8-bit	LD (IY+d),H	FD 74 d
LD B,H	44	LD H,(HL)	66	LD (IY+d),L	FD 75 d
LD B,L	45	LD H,(IX+d)	DD 66 d	LD (IY+d),n	FD 36 d 8-bit
LD B,n	06 8-bit	LD H,(IY+d)	FD 66 d	LD (nn),A	32 16-bit
LD B,(HL)	46	LD HL,nn	21 16-bit	LD (nn),BC	ED 43 16-bit
LD B,(IX+d)	DD 46 d	LD HL,(nn)	2A 16-bit	LD (nn),DE	ED 53 16-bit
LD B,(IY+d)	FD 46 d	LD I,A	ED 47	LD (nn),HL	22 16-bit
LD BC,nn	01 16-bit	LD IX,nn	DD 21 16-bit	LD (nn),IX	DD 22 16-bit
LD BC,(nn)	ED 4B 16-bit	LD IX,(nn)	DD 2A 16-bit	LD (nn),IY	FD 22 16-bit
LD C,A	4F	LD IY,nn	FD 21 16-bit	LD (nn),SP	ED 73 16-bit
LD C,B	48	LD IY,(nn)	FD 2A 16-bit	LDDR	ED A8
LD C,C	49	LD L,A	6F	LDI	ED B8
LD C,D	4A	LD L,B	68	LDIR	ED A0
LD C,E	4B	LD L,C	69	NEG	ED B0
LD C,H	4C	LD L,D	6A	NOP	00
LD C,L	4D	LD L,E	6B	OR A	B7
LD C,n	0E 8-bit	LD L,H	6C	OR B	B0
LD C,(HL)	4E	LD L,L	6D	OR C	B1
LD C,(IX+d)	DD 4E d	LD L,n	2E 8-bit	OR D	B2
LD C,(IY+d)	FD 4E d	LD L,(HL)	6E	OR E	B3
LD D,A	57	LD L,(IX+d)	DD 6E d	OR H	B4
LD D,B	50	LD L,(IY+d)	FD 6E d	OR L	B5
LD D,C	51	LD R,A	ED 4F	OR n	F6 8-bit
LD D,D	52	LD SP,HL	F9	OR (HL)	B6
LD D,E	53	LD SP,IX	DD F9	OR (IX+d)	DD B6 d
LD D,H	54	LD SP,IY	FD F9	OR (IY+d)	FD B6 d
LD D,L	55	LD SP,nn	31 16-bit	OTDR	ED BB
LD D,n	16 8-bit	LD SP,(nn)	ED 7B 16-bit	OTIR	ED B3
LD D,(HL)	56	LD (BC),A	02	OUT (C),A	ED 79
LD D,(IX+d)	DD 56 d	LD (DE),A	12	OUT (C),B	ED 41
LD D,(IY+d)	FD 56 d	LD (HL),A	77	OUT (C),C	ED 49
LD DE,nn	11 16-bit	LD (HL),B	70	OUT (C),D	ED 51
LD DE,(nn)	ED 5B 16-bit	LD (HL),C	71		

Mnemonic	Hex	Mnemonic	Hex	Mnemonic	Hex
OUT (C,E)	ED 59	RLCA	07	SLA B	CB 20
OUT (C,H)	ED 61	RLD	ED 6F	SLA C	CB 21
OUT (C,L)	ED 69	RR A	CB 1F	SLA D	CB 22
OUT (n),A	D3 8-bit	RR B	CB 18	SLA E	CB 23
OUTD	ED AB	RR C	CB 19	SLA H	CB 24
OUTI	ED A3	RR D	CB 1A	SLA L	CB 25
POP AF	F1	RR E	CB 1B	SLA (HL)	CB 26
POP BC	C1	RR H	CB 1C	SLA (IX+d)	DD CB d 26
POP DE	D1	RR L	CB 1D	SLA (IY+d)	FD CB d 26
POP HL	E1	RR (HL)	CB 1E	SRA A	CB 2F
POP IX	DD E1	RR (IX+d)	DD CB d 1E	SRA B	CB 28
POP IY	FD E1	RR (IY+d)	FD CB d 1E	SRA C	CB 29
PUSH AF	F5	RRA	1F	SRA D	CB 2A
PUSH BC	C5	RRC A	CB 0F	SRA E	CB 2B
PUSH DE	D5	RRC B	CB 08	SRA H	CB 2C
PUSH HL	E5	RRC C	CB 09	SRA L	CB 2D
PUSH IX	DD E5	RRC D	CB 0A	SRA (HL)	CB 2E
PUSH IY	FD E5	RRC E	CB 0B	SRA (IX+d)	DD CB d 2E
RES b,s	see pp. 604-5	RRC H	CB 0C	SRA (IY+d)	FD CB d 2E
RET	C9	RRC L	CB 0D	SRL A	CB 3F
RET C	D8	RRC (HL)	CB 0E	SRL B	CB 38
RET M	F8	RRC (IX+d)	DD CB d 0E	SRL C	CB 39
RET NC	D0	RRC (IY+d)	FD CB d 0E	SRL D	CB 3A
RET NZ	C0	RRCA	0F	SRL E	CB 3B
RET P	F0	RRD	ED 67	SRL H	CB 3C
RET PE	E8	RST 00H	C7	SRL L	CB 3D
RET PO	E0	RST 08H	CF	SRL (HL)	CB 3E
RET Z	C8	RST 10H	D7	SRL (IX+d)	DD CB d 3E
RETI	ED 4D	RST 18H	DF	SRL (IY+d)	FD CB d 3E
RETN	ED 45	RST 20H	E7	SUB A	97
RL A	CB 17	RST 28H	EF	SUB B	90
RL B	CB 10	RST 30H	F7	SUB C	91
RL C	CB 11	RST 38H	FF	SUB D	92
RL D	CB 12	SBC A,A	9F	SUB E	93
RL E	CB 13	SBC A,B	98	SUB H	94
RL H	CB 14	SBC A,C	99	SUB L	95
RL L	CB 15	SBC A,D	9A	SUB n	D6 8-bit
RL (HL)	CB 16	SBC A,E	9B	SUB (HL)	96
RL (IX+d)	DD CB d 16	SBC A,H	9C	SUB (IX+d)	DD 96 d
RL (IY+d)	FD CB d 16	SBC A,L	9D	SUB (IY+d)	FD 96 d
RLA	17	SBC A,n	DE 8-bit	XOR A	AF
RLC A	CB 07	SBC A,(HL)	9E	XOR B	A8
RLC B	CB 00	SBC A,(IX+d)	DD 9E d	XOR C	A9
RLC C	CB 01	SBC A,(IY+d)	FD 9E d	XOR D	AA
RLC D	CB 02	SBC HL,BC	ED 42	XOR E	AB
RLC E	CB 03	SBC HL,DE	ED 52	XOR H	AC
RLC H	CB 04	SBC HL,HL	ED 62	XOR L	AD
RLC L	CB 05	SBC HL,SP	ED 72	XOR n	EE 8-bit
RLC (HL)	CB 06	SCF	37	XOR (HL)	AE
RLC (IX+d)	DD CB d 06	SET b,s	see p. 615	XOR (IX+d)	DD AE d
RLC (IY+d)	FD CB d 06	SLA A	CB 27	XOR (IY+d)	FD AE d

## SUMMARY OF FLAG OPERATION

Instructions	D <sub>7</sub> S	Z	H	P/V	N	D <sub>0</sub> C	Comments		
ADD A, s; ADC A, s	†	†	X	†	X	V	0	8-bit add or add with carry.	
SUB s; SBC A, s; CP s; NEG	†	†	X	†	X	V	1	8-bit subtract, subtract with carry, compare and negate accumulator.	
AND s	†	†	X	1	X	P	0	Logical operation.	
ORs, XOR s	†	†	X	0	X	P	0	Logical operation.	
INC s	†	†	X	†	X	V	0	8-bit increment.	
DEC s	†	†	X	†	X	V	1	8-bit decrement.	
ADD DD, ss	•	•	X	X	X	•	0	16-bit add.	
ADC HL, ss	†	†	X	X	X	V	0	16-bit add with carry	
SBC HL, ss	†	†	X	X	X	V	1	16-bit subtract with carry.	
RLA; RLCA; RRA; RRCA	•	•	X	0	X	•	0	Rotate accumulator.	
RL m; RLC m; RR m;	†	†	X	0	X	P	0	Rotate and shift locations.	
RRC m; SLA m; SRA m; SRL m									
RLD; RRD	†	†	X	0	X	P	0	•	Rotate digit left and right.
DAA	†	†	X	†	X	P	•	†	Decimal adjust accumulator.
CPL	•	•	X	1	X	•	1	•	Complement accumulator.
SCF	•	•	X	0	X	•	0	1	Set carry.
CCF	•	•	X	X	X	•	0	†	Complement carry.
IN r(C)	†	†	X	0	X	P	0	•	Input register indirect
INI; IND; OUTI; OUTD	X	†	X	X	X	X	1	•	Block input and output. Z = 1 if B ≠ 0, otherwise Z = 0.
INIR; INDR; OTIR; OTDR	X	1	X	X	X	X	1	•	Block input and output. Z = 1 if B ≠ 0, otherwise Z = 0.
LDI; LDD	X	X	X	0	X	†	0	•	Block transfer instructions. P/V = 1 if BC ≠ 0, otherwise P/V = 0.
LDIR; LDDR	X	X	X	0	X	0	0	•	Block transfer instructions. P/V = 1 if BC ≠ 0, otherwise P/V = 0.
CPI; CPIR; CPD; CPDR	X	†	X	X	X	†	1	•	Block search instructions. Z = 1 if A = (HL), otherwise Z = 0. P/V = 1 if BC ≠ 0, otherwise P/V = 0.
LD A; I, LD A, R	†	†	X	0	X	IFF	0	•	IFF, the content of the interrupt enable flip-flop, (IFF <sub>2</sub> ), is copied into the P/V flag
BIT b, s	X	†	X	1	X	X	0	•	The state of bit b of location s is copied into the Z flag.

## SYMBOLIC NOTATION

Symbol	Operation	Symbol	Operation
S	Sign flag. S = 1 if the MSB of the result is 1.	†	The flag is affected according to the result of the operation.
Z	Zero flag. Z = 1 if the result of the operation is 0.	•	The flag is unchanged by the operation.
P/V	Parity or overflow flag. Parity (P) and overflow (V) share the same flag. Logical operations affect this flag with the parity of the result while arithmetic operations affect this flag with the overflow of the result. If P/V holds parity: P/V = 1 if the result of the operation is even; P/V = 0 if result is odd. If P/V holds overflow, P/V = 1 if the result of the operation produced an overflow. If P/V does not hold overflow, P/V = 0.	0	The flag is reset by the operation.
H*	Half-carry flag. H = 1 if the add or subtract operation produced a carry into, or borrow from, bit 4 of the accumulator.	1	The flag is set by the operation.
N*	Add/Subtract flag. N = 1 if the previous operation was a subtract.	X	The flag is indeterminate.
C	Carry/Link flag. C = 1 if the operation produced a carry from the MSB of the operand or result.	V	P/V flag affected according to the overflow result of the operation.
		P	P/V flag affected according to the parity result of the operation.
H*	Half-carry flag. H = 1 if the add or subtract operation produced a carry into, or borrow from, bit 4 of the accumulator.	r	Any one of the CPU registers A, B, C, D, E, H, L.
N*	Add/Subtract flag. N = 1 if the previous operation was a subtract.	s	Any 8-bit location for all the addressing modes allowed for the particular instruction.
C	Carry/Link flag. C = 1 if the operation produced a carry from the MSB of the operand or result.	ss	Any 16-bit location for all the addressing modes allowed for that instruction.
		ii	Any one of the two index registers IX or IY.
		R	Refresh counter.
		n	8-bit value in range <0, 255>.
		nn	16-bit value in range <0, 65535>.

\*H and N flags are used in conjunction with the decimal adjust instruction (DAA) to properly correct the result into packed BCD format following addition or subtraction using operands with packed BCD format.



# Index

- Absolute jump instructions, 182–86  
conditional absolute jump, 183  
unconditional absolute jump, 183
- Accumulator, 52  
data copy instructions, 176–77  
rotate instructions, 213–15
- ADBYTE subroutine, 268, 269
- ADC instructions, 265, 266, 562–63
- A/D converters. *See* Analog-to-digital converters
- ADD instructions, 130–31, 134, 147, 178, 267, 563, 564–66
- Addition, 134  
example program, 193–94  
instructions for, 177–78  
16-bit operations, addition with carry, 265  
two hex numbers, example programs, 139–43, 162–65
- Address bus, 27–28  
in MPU design, 498–99
- Z80, 59
- Address decoding  
memory, design, 480, 482–83  
memory interfacing, 86–87
- Addressing modes, 137–39  
listing of, 138
- Add/subtract flag, 56
- Alphanumeric codes, 19
- Alternate registers, 55
- ALU. *See* Arithmetic logic unit
- Analog signal processor, Intel 2920, 540
- Analog-to-digital converters, 117
- AND gate, 634
- AND instructions, 135, 138, 148, 208, 209, 566–68
- Appliance control, in memory-mapped I/O interfacing, 113–16
- Arithmetic logic unit (ALU), 6, 8, 30
- Arithmetic operations, 133–34, 147, 177–82  
addition, 134  
instructions for, 177–78  
division (binary), example program, 271–74  
flags, 180–82  
increment/decrement, 134  
instructions for, 178
- Modulo-2 arithmetic, 629–30
- multibyte numbers, adding, example program, 268–69
- multiplication (binary), example program, 270–71
- 1's and 2's complement, 134  
instructions for, 179
- signed numbers, 181–82, 626–27
- 16-bit operations, 265–67  
addition with carry, 265  
subtraction with carry, 266–67
- software project, example program, 282–85
- subtraction, 134, 627  
binary subtraction, 626  
decimal subtraction, 623–25  
instructions for, 177–78
- See also* specific operations.
- ASCBIN subroutine, 280–81, 285
- ASCII, 19, 152  
ASCII codes, listing of (Appendix C), 631  
conversion from binary, example program, 281–82  
conversion to binary, example program, 279–81  
software design project, 282–85
- ASCII keyboard, 9, 152, 280
- ASCII subroutine, 282, 285
- ASM extension, 158
- Assembler directives, 161–62  
DB, 161  
DW, 161  
END, 161, 163  
EQU, 161, 163  
ORG, 161, 163
- Assemblers  
advantages in use of, 162  
assembler directives, 161–62  
listing of, 161  
assembler statements, 160
- CP/M, 157  
error messages, 164  
precautions, in program writing, 165  
two-pass assembler, 164  
use of, example program, 162–63
- Assembler statements, 160
- Assembling the code, 139, 140–41
- Assembly language programming, 17, 18, 19–20, 129–49  
addressing modes, 137–39  
listing of, 138
- assemblers, 159–65
- assembling the code, 139, 140–41
- binary division, 271–74
- binary multiplication, 269–71
- example program, 139–42  
assembling the code, 140–41  
execution of program, 142  
program documentation, 142–43  
storing in memory, 141–42  
writing format, 142–43
- flowcharting, 143–46
- instructions, 130–39, 170–89
- programming statements, 160
- repetitive techniques, 189–90
- 16-bit operations, 264–67
- software design project, 282–85
- stack, 233, 234–41
- subroutines, 241–58
- See also* specific topics.
- Assembly process, 159–60  
assembling the code, 139, 140–41  
source program, statements used in, 160
- Asynchronous format, 290, 413, 420–21
- Motorola MC68000, signals, 548
- software controlled, 422–25  
serial data reception, 425  
serial data transmission, 422–23
- Backward jump, 186, 187
- BAK file, 158
- BASE subroutine, 282
- Baud, 411  
baud generator design, example program, 395–97  
baud requirements, 414–15
- BCD  
BCD operations, 53, 55  
flags, 55
- BCD subroutine, 278–79
- codes, table look-up technique, 255
- conversion from binary, example program, 277–79, 622–23
- conversion to binary, example program, 274–77
- counter, example program, 252–57
- packed BCD, 253
- BDOS, 155
- Bidirectional data transfer, example program, 359–66
- Binary code  
assembling the code, 139, 141  
translated from hex, 141

- Binary Code Decimal (BCD)  
operations. *See BCD*
- Binary division, 271–74  
example program, 271–74
- Binary multiplication, 269–71  
example program, 270–71
- Binary subtraction, 626
- BINASC subroutine, 281–82
- BINBCD subroutine, 278
- BIOS, 155
- Bit, addressing mode, 138
- Bit and byte processor, MCS-51, 539–40
- BIT instructions, 135, 149, 216, 568–69
- Bit manipulation, 135, 148, 213  
bit reset, 135  
instructions for, 216
- bit set, 135  
instructions for, 216
- bit test, 135  
instructions for, 216
- 16-bit operations, 267
- Bit mode, Mode 3, PIO, 333, 344–48
- Bit rotation, 135, 148, 149
- Bits, 4–5
- Block transfers, example program, 198–99
- Branching operations, 136, 148, 182–87  
call/return, 136  
instructions for, 182–87
- jump, 136  
absolute jump instructions, 182–86  
relative jump instructions, 186–87
- restart, 136
- Breakpoint technique  
dynamic debugging, 196–97  
restart (RST) instruction, 322–24
- BSR mode, Intel 8255A, 366, 371, 373
- Bubble matching, 635
- Bus, 8, 10  
in MPU design, 504  
*See also* Address bus; Control bus;  
Data bus.
- Business microcomputers, 13
- Bus interface standards, 554–57  
bus signals, groups of, 555  
defining standards, 554
- GPIB interface bus, 557  
bus signals, 557–58
- IBM PC bus, 555–56
- multibus, 556–57
- S-100 bus, 554–56  
interface, 555–56
- STD bus, 555
- Bytes, 16  
adding, example program, 193–94, 195
- instructions, 1-byte through 4-byte, 130–31
- transfer to memory, 112
- CACODE, 354
- CALL instructions, 136, 148, 241–42, 297, 569–70
- Call/return, 136
- CALL subroutine, 241, 242, 243
- Carry flag, 55, 183, 225, 274
- CCF instructions, 267, 570
- CCP, 156
- Central processing unit (CPU),  
explanation of, 6
- CHAR subroutine, 283–84
- Check sum, error checks, data  
transmission, 416
- Clock signals, microprocessor unit  
(MPU), 29–30
- CMOS microprocessor, 551
- Code conversions. *See* ASCII; BCD;  
Number conversion
- COM files, 157, 158, 165
- COMMAND instruction, 159, 428
- Comments, 143, 160
- Compare, 135  
instructions for, 212–13  
special instructions, 223–24
- Comparing numbers, search for  
maximum number, example  
programs, 217–18, 224–25
- Compiler, use of, 20
- Complement Carry flag, 54, 267, 570
- Complement operations. *See* 1's and  
2's complement
- Computers  
types of, 11–15  
business microcomputers, 13  
home computers, 14  
mainframe computers, 12  
microcomputers, 12  
midicomputers, 12  
minicomputers, 12  
single-board microcomputers,  
14–15  
single-chip microcomputers  
(microcontrollers), 15
- Conditional absolute jump, 183
- Conditional relative jump, 186
- Control bus, in MPU design, 501
- Controllers, Z80 DMA controller,  
487–91
- Control signals, 28  
counter/timer circuit (CTC), 384, 388, 402
- serial I/O mode  
Intel 8251A, 428  
Z80 and DART, 441–43
- Z80, 59, 68–69
- Control unit, 9
- Control words  
channel control word, 389–92, 397, 398, 399, 403  
Intel 8255A, 369–70, 371, 373, 374, 378  
PIO, 337–39, 347–48
- Coprocessing, Intel 8086/8088, 545
- COPY, 159
- Counter/timer circuit (CTC), 383–407  
channel control word, 389–92, 397, 398, 399, 403
- control signals, 384, 388, 402
- counter mode, 395, 398–401  
use with interrupt, example  
program, 398–401
- Intel 8253 interval timer/counter,  
401–7  
compared to Z80 CTC, 407
- control signals, 402
- control words, 403
- data bus buffer, 402
- programming of, 403–5
- as square wave generator, 405–7
- interrupts, 392–94, 400  
interrupt priorities, 394
- programming of, 389–92  
programmable modes, 389
- timer mode, 395  
baud generator design, example  
program, 395–97
- Counting, 189
- CP instructions, 135, 148, 212, 571–72
- CPD instructions, 223, 572
- CPDR instructions, 149, 223, 573
- CPI instructions, 223, 224–25, 574
- CPIR instructions, 223, 574–75
- CPL instructions, 134, 179, 575
- CP/M, 155–58  
BDOS, 155
- BIOS, 155
- CCP, 156
- program assembly, 158  
files resulting from, 158
- utility programs, 156–57  
assemblers, 157
- debugger, 157
- editor, 156–57
- linking loading, 157
- CRIR, 223
- Cross-assemblers, and MS-DOS, 159

- Cyclic redundancy check, error checks, data transmission, 416–17
- DAA instructions, 252, 253–54, 575–76
- Daisy chain format, 394  
PIO, 343–44
- DART. *See* Z80 and DART
- Data acquisition, flowcharts, 191, 192
- Data bus, 28  
in MPU design, 499, 501  
Z80, 59
- Data copy operations, 132–33, 146–47, 170–77  
addressing modes, 137  
example program, 190–92  
16-bit operations, 264  
types of  
among registers, 170–72  
between accumulator and I/Os, 176–77  
from and into memory, 172–75
- Data processing, flowcharts, 191
- Data transfer  
bidirectional data transfer, 343, 378  
example program, 359–66
- Direct Memory Access (DMA), 491–92
- microprocessor controlled  
data transfer with handshake signals, 291–92  
data transfer with interrupt, 291  
data transfer with status check (polling), 291  
data transfer with WAIT signal, 291  
unconditional data transfer, 291  
peripheral-controlled, 292
- DB, assembler directive, 161
- DBONCE, 352
- Debugging, 194–97, 221–23  
common errors, 194, 197, 221, 296  
debugger, CP/M, 157  
dynamic debugging, 194, 196–97  
breakpoint, 196–97  
register examine, 196  
single step, 196
- example program for, 221–23
- modular programs, 258
- static debugging, 194, 196
- tools for  
disassembly, 533  
in-line assembly, 533  
real-time trace, 533  
register display, 533
- Decimal subtraction, 623–25
- DEC instructions, 134, 147, 178, 577
- Decision making  
flags, use of, 180–82  
flowcharts, 191
- Decoding  
absolute decoding, 92  
address decoding, 86–87  
decoding instructions, microprocessor, 30  
linear decoding, 92
- Defining standards, 554
- DELAY subroutine, 249, 250, 256
- Delimiters, 160  
types of, 160
- Designing systems  
memory system, 478–84  
single-board microcomputer, 496–534  
*See also* specific systems.
- Destination, 132
- Diagnostic routine, 94–95  
input/output (I/O) interfacing, 120  
instructions in, 94–95  
memory interfacing, 94–95
- Digit driver, 509
- DIGIT subroutine, 284
- DI instructions, 136, 578
- DIR, 159
- Directives, 160
- Direct Memory Access (DMA), 292, 484–91  
data transfer, 491–92  
requirements of, 486  
sequential transfer, 486  
simultaneous transfer, 486  
Z80 DMA controller, 487–91  
interfacing for, 489–90  
programming of, 490–91  
signals for, 488–89
- Disable Interrupt, 296
- Disassembly, debugging tool, 533
- Display buffer, 528
- Display module, software design, 527–28
- DIV8 subroutine, 273
- DIVIDE subroutine, 273, 274, 277
- Division, binary division, 271–74
- DJNZ d, 197, 198
- DJNZ instructions, 149, 578
- Documentation, for subroutines, 250
- Downloading, in-circuit emulator, 533
- DISPLAY subroutine, 255, 257, 354
- Duplex transmission, 413–14  
full duplex, 414  
half duplex, 413
- DW, assembler directive, 161
- Dynamic debugging  
breakpoint, 196–97  
register examine, 196
- single step, 196
- Dynamic memory, 468–78  
circuit design, 471–73  
CAS, generating, 473, 477  
RAS, generating, 472, 476
- interfacing, 2118 with Z80, example of, 475–78
- interfacing, 470  
refreshing technique, 474, 478
- structure of, 468
- EBCDIC, 19
- Editor, CP/M, 156–57
- 8-bit microprocessors  
Hitachi HD64180, 551  
Intel 8008, 550  
Intel 8085, 71–74  
Motorola MC6800, 74–75, 550  
National Semiconductor NSC800, 74
- Z280, 551, 554
- Z8001, 550
- Z80, 550
- 18-bit microprocessor, 7, 9
- 80386 microprocessor, 549
8253. *See* Intel 8253 interval timer/counter
- 8255A. *See* Intel 8255A
- EI instructions, 578
- Electrically Erasable PROM (EE-PROM), 43, 44
- Emulation process, 532
- Enable Interrupt, 296
- Encoder, keyboard, 519–20
- END, assembler directive, 161, 163
- EQU, assembler directive, 161, 163
- Erasable Programmable Read-Only Memory (EPROM), 43, 506
- Error messages, of assembler, 164
- Exchange, 239  
16-bit operations, 264–65
- Exchange instructions, 239, 264, 265
- Executing instructions, microprocessor, 30
- EX instructions, 239, 264, 578–79
- Extended, addressing mode, 138, 173
- Externally initiated operations, microprocessor unit (MPU), 28–29
- External requests, Z80, 59–60
- EXX instructions, 133, 239, 580
- Fetching instructions, microprocessor, 30
- Fields, 160
- File, 155
- File management utilities, 156

Flag register, 53–55  
  alternate flag register, 55  
Flags, 562  
  add/subtract flag, 56  
  carry flag, 55  
  decision making, use in, 180–81  
  examining/manipulating, example program, 239–41  
  function of, 181  
  half-carry flag, 56  
  and jump instructions, 183  
  parity/overflow flag, 55–56  
  setting flags, 189  
  and signed numbers, 181–82  
  sign flag, 55  
  zero flag, 55  
Flip-flops, 52  
  *See also* Flags.  
Floppy disks, 13, 152–54  
  sectors/tracks of, 153–54  
Flowcharting, 143–46  
  data transfer example, 192, 194  
  steps, designating blocks, 191  
  symbols used in, 144–45  
FORMAT, 159  
Forward jump, 186, 187  
4-byte instruction, 131  
Framing, 413  
Frequency shift keying, modems, 417  
Function generator, computer as,  
  example program, 218–21  
Function module, software design, 528–30  
  
General-purpose registers, 55  
  
Half-carry flag, 56  
HALT instructions, 136, 149, 170,  
  171, 175, 580  
Handshake mode, Intel 8255A, 366  
Handshake signals, 291–92, 330  
  PIO, 339–40, 343  
Hard disk, 154  
Hardware, 4  
Hardware model (Z80), 57–61  
  address bus, 59  
  control signals, 59, 68–69  
  data bus, 59  
  external requests, 59–60  
  hex decoder/driver, 510–11  
  power/frequency signals, 61  
  request acknowledgment, 60  
  special signals, 60  
Hex code, 143  
  translated from mnemonics, 139,  
    140–41  
  translated into binary, 141

HEX files, 157, 158, 165  
  Hex keyboard, 9  
High-level languages, 17, 20–21  
  compiler, 20  
  types of, 20  
Home computers, 14  
  
IBMDOS, 159  
IM instructions, 580–81  
Immediate, addressing mode, 137, 138,  
  170, 172  
Immediate extended addressing mode,  
  137, 138, 171  
Implied addressing, 138  
  *See also* Logic operations.  
INC instructions, 134, 138, 147, 178,  
  582–83  
In-circuit emulator, 532  
  and emulation process, 532  
  features of, 532–33  
  debugging tools, 533  
  downloading, 533  
  resource sharing, 533  
Increment/decrement, 134  
  instructions for, 178  
Indexed, addressing mode, 138  
Index registers (IX and IY), 56,  
  187–89  
  instructions, 188–89  
Indexing, 189  
IND instructions, 582–83  
INDR instructions, 149, 583–84  
INI instructions, 584  
  input devices, interfacing, 108–9  
IN instructions, 133, 176, 581  
INIR instructions, 585  
Initialization  
  flowcharts, 191  
  PIO, 342–43  
  software design, 527  
In-line assembly, debugging tool, 533  
Input devices  
  keyboards  
    ASCII keyboard, 9  
    hexadecimal keyboard, 9  
Input devices (interfacing), 107–11  
  IN instruction, 108–9  
    execution/timing of, 108–9  
    machine cycles, 108–9  
  input switches, interfacing, 110–11  
  steps in, 109  
Input/output (I/O) modes  
  parallel I/O, 101  
  serial I/O, 101  
Input/output (I/O), explanation of, 5  
Input/output (I/O) interfacing, 101–22  
  data transfer (microprocessor controlled), 291–92  
  data transfer with interrupt, 291  
  data transfer with handshake signals, 291–92  
  data transfer with status check (polling), 291  
  data transfer with WAIT signal, 291  
  unconditional data transfer, 291  
data transfer (peripheral-controlled), 292  
of dc motor, 119–20  
diagnostic routine, 120  
memory-mapped I/O, 111–16, 290,  
  291  
  appliance control using, 113–16  
  data transfer instructions,  
    execution, 112–13  
parallel I/O mode, 101, 290–91  
parallel-to-serial conversion, 291  
peripheral-mapped I/O, 44–45, 101,  
  290, 291  
  interfacing LEDs, example of,  
    105–7  
  switches as input device, example of, 110–11  
questions related to, 121–22  
serial I/O mode, 101, 290, 291  
of temperature sensor, 117–19  
  *See also* Input devices, interfacing;  
    Output devices, interfacing.  
Instructions, 130–39, 143, 170–89  
  arithmetic operations, 133–34, 147,  
    177–82  
  bit manipulation, 135, 148  
  branching operations, 136, 148,  
    182–87  
  data copy operations, 132–33,  
    146–47, 170–77  
  4-byte instruction, 131  
  listing of instructions, 562–619  
  logic operations, 135, 148  
  machine control operations, 136, 149  
  mnemonic, 18  
  1-byte instruction, 130–31  
  repetitive instructions, 149  
  special instructions  
    Compare instructions, 223–24  
    for multiple tasks, 197–98  
    repetitive instructions, 149  
stack, 235–38  
subroutines, 241–42  
3-byte instruction, 131  
2-byte instruction, 131  
Z80, 61  
  *See also* specific operations.

- Instruction set, 130  
     Intel 8086/8088, 545  
     Motorola MC68000, 547–48
- Integrated circuits (IC), 7
- Integrated RAM, 44
- Intel 2920 analog signal processor, 540
- Intel 4004, 7
- Intel 8008, 7
- Intel 80186/80286, 548
- Intel 8085, 71–74
- Intel 8086/8088  
     coprocessing, 545  
     instruction set, 545  
     memory segmentation, 542–44  
     simultaneous processing, 544
- Intel 8251A  
     serial I/O  
         control register, 428, 429  
         control signals, 428  
         data register, 428, 429  
         elements of chips, 425–27  
         interfacing RS-232 terminal, example program, 435–39  
         programming of, 431–35  
         receiver section, 431  
         status register, 428  
         transmitter section, 430–31
- Intel 8253 interval timer/counter, 401–7  
     compared to Z80 CTC, 407  
     control signals, 402  
     control words, 403  
     data bus buffer, 402  
     programming of, 403–5  
     as square wave generator, 405–7
- Intel 8255A  
     BSR mode, 366, 371, 373  
     compared to PIO, 378–79  
     control logic, 367, 369  
     control words, 369–70, 371, 373, 374, 378  
     handshake mode, 366  
     interfacing with A/D converter, 371–79  
     Mode 0, 370, 373  
     Mode 1, 374, 376–78  
     Mode 2, 378
- Intel MCS-51 single-chip microcomputer, 539–40
- Interfacing, 290  
*See also* Input/output (I/O)  
     interfacing; Memory interfacing; Programmable interface devices.
- Interpreter, 20
- Interrupt Enable flip-flops, 295, 297, 300
- Interrupt Enable word, PIO, 341–42
- Interrupts, 293–324  
     counter/timer circuit (CTC), 392–94, 400  
     data transmission, Z80 and DART, 453–54, 455–57  
     explanation of, 294–95  
         steps in process, 299–300  
         telephone analogy, 294–95, 299  
     implementation issues, 305–7  
     Interrupt Request and Acknowledge, 300, 303  
     maskable interrupt, 293, 295–96  
     Disable Interrupt, 296  
     Enable Interrupt, 296  
     Interrupt Enable flip-flops, 295, 297, 300  
     interrupt mode, instructions for, 296  
     Mode 0, 296, 298–300  
         illustration of, 302–7  
     Mode 1, 296  
         illustration of, 307–13  
     Mode 2, 296, 313–15  
         in MPU design, 504  
     multiple interrupts, 318–22  
         interrupt vector technique, 320–22  
         polling method, 318–20  
     nonmaskable interrupt, 293, 295, 297, 315–17  
     PIO, 341–42, 343–44  
     Restart (RST) instructions, 297–98, 300, 302, 307  
         breakpoint technique, 322–24  
         and single-board microcomputers, 305
- Interrupt vectors  
     interrupt vector register (I), 56  
     interrupt vector technique, multiple interrupts, 320–22  
     PIO, 341–42
- JP instructions, 136, 139, 148, 183, 264, 585–86
- JR instructions, 138, 186, 587–88
- Jump, 136  
     instructions, 182–87  
         absolute jump, 182–86  
         relative jump, 186–87
- Keyboards  
     ASCII keyboard, 9  
     hexadecimal keyboard, 9  
     interfacing with seven-segment LED, bidirectional data transfer, 348–59
- KYBRD subroutine, 354, 514–16
- KYCHK subroutine, 350, 352
- KYCODE subroutine, 351, 352, 353
- KYPUSH subroutine, 350
- Labels, 143, 160, 163, 192–93
- Languages (computer)  
     alphanumeric codes, 19  
         ASCII, 19  
         EBCDIC, 19  
     assembly language, 17, 18, 19–20  
         writing/execution of, 19–20  
         Z80 assembly language, 18–19  
     high-level languages, 17, 20–21  
         compiler, use of, 20  
         types of, 20  
     machine language, 16, 17–18  
         Z80 machine language, 17–18  
*See also* Assembly language programming.
- Large-Scale Integration (LSI), 7, 8
- LDD instructions, 197, 596
- LDLR instructions, 197, 597
- LDI instructions, 197, 597–98
- LD instructions, 130, 131, 133, 138, 140, 146–47, 170–71, 172–73, 174–75, 234, 235, 236, 264, 589–95
- LDIR instructions, 197, 198–99, 599–600
- LEDs (Light Emitting Diodes), 5, 9, 44  
     interfacing, 105–7  
     seven-segment codes, 254, 255
- LIFO sequence, 238, 239
- Line drivers, RS-232C, 420, 436
- Line receivers, RS-232C, 420, 436–37
- Linking loading, 165  
     CP/M, 157
- Load operations, 132–33, 146–47  
*See also* Data copy operations.
- Logic operations, 135, 148, 208–12  
     AND instructions, 208, 209  
     compare, 135  
         instructions for, 212–13  
     logic functions, 135  
     OR instructions, 208, 209  
     rotate, instructions for, 213–15  
     rotate and shift, instructions for, 215–16  
     shift and rotate, 135  
     XOR instructions, 208, 209
- Logic state analyzer, 533–34
- LOOKUP subroutine, 255–56, 257
- Looping, 189
- LOOP subroutine, 256–57
- Machine code, static debugging, 194, 196
- Machine control operations, 136, 149
- Machine cycles, 464–65  
     definition of, 62

- Machine cycles (*continued*)  
     memory read machine cycle, 65–67, 68  
     memory write cycle, 67–68  
     opcode fetch machine cycles, 62–64, 68  
     T-states in, 62, 70
- Machine language, 16, 17–18  
     Z80 machine language, 17–18
- Mainframe computers, 12
- Maskable interrupt, 293, 295–96  
     Disable Interrupt, 296  
     Enable Interrupt, 296  
     Interrupt Enable flip-flops, 295, 297, 300
- Masked ROM, 43
- Matrix keyboard, 355–57  
     interfacing, 356–57  
         hardware approach, 519–20  
         interfacing circuit, 513  
         problem areas, 514  
         program for, 513–17
- Medium-Scale Integration (MSI), 7
- Memory, 32–44  
     classification of, 41–42  
     prime memory, 41  
     storage memory, 41  
     data copy, from and into memory, 172–75  
     designing system, 478–84  
         address decoding, 480, 482–83  
     EPROM memory, 506  
     issues related to, 478–80  
     PROM decoding, 484–85  
     Read/Write (R/W) memory, 506–7
- Direct Memory Access (DMA), 484–91  
     dynamic memory, 468–78  
     explanation of, 5, 10, 32–36  
     memory chip, 33, 36  
     memory map, 37–40  
     MPU, writing/reading, 40–41  
     new advancements, 43–44  
         Integrated RAM, 44  
         Non-Volatile RAM, 44  
         Zero Power RAM, 43–44
- Random-Access Memory (RAM), 10
- Read-Only Memory (ROM), 10, 32, 42–43  
     Electrically Erasable PROM (EE-PROM), 43, 44  
     Erasable Programmable Read-Only Memory (EPROM), 43  
     masked ROM, 43  
     Programmable Read-Only Memory (PROM), 43
- Read/Write Memory (R/WM), 10, 32, 41–42  
     rotate instructions, 213–15
- Memory addresses, 143  
     high-order/low-order addresses, 39–40
- Memory interfacing, 83–97  
     address decoding, 86–87  
     design factors, 92–94  
         circuit analysis, 93–94  
         diagnostic routine, 94–95  
         problem analysis, 93  
         problem statement, 93  
     dynamic memory, 470, 475–78  
     function of, 83
- MOSTEK MK4802, 90–92  
     absolute vs. linear decoding, 92  
     interfacing circuit, 90–91  
     memory map, 92  
     questions related to, 95–97  
     requirements of microprocessor, 83–86  
     2732 EPROM, 87–90  
         interfacing circuit, 88–89  
         memory map, 89–90  
     wait states, 464–69  
         generation of, 466–68
- Memory map, memory interfacing, 89–90, 92
- Memory-mapped I/O, 111–16  
     appliance control using, 113–16  
         data transfer instructions, 112–13
- Memory pointers, 55  
     index registers, 187–89
- Memory read machine cycle, 65–67, 68
- Memory refresh register (R), 56
- Memory segmentation, Intel 8086/8088, 542–44
- Memory write cycle, 67–68
- Microcomputers, 12  
     components of  
         input/output (I/O) devices, 44–45  
         memory, 32–44  
         microprocessor unit (MPU), 26–32  
     organization of, 8–10  
         input, 9  
         memory, 10  
         microprocessor, 8  
         output, 9–10  
         system bus, 10  
         program execution, 10–11, 45–46
- Microcontroller, 4  
     *See also* Single-chip microcomputers.
- Microprocessors  
     areas of  
         arithmetic/logic unit (ALU), 8  
         control unit, 9  
         register array, 9
- binary digits, 4–5, 16  
     as a central processing unit (CPU), 6  
     and computer language, 16–21  
         assembly language, 17, 18, 19–20  
         compiler, use of, 20  
         machine language, 16, 17–18  
     explanation of, 4–5  
     and input/output, 5  
     instruction set, 16, 17  
     and memory, 5  
     and microcomputer system, 8–11  
     as programmable devices, 5  
     and semiconductor technology, 7  
     types available, 7, 9
- Microprocessor unit (MPU), 26–32  
     clock signals, 29–30  
     externally initiated operations, 28–29  
     and instructions, 30  
         decoding instructions, 30  
         executing instructions, 30  
         fetching instructions, 30  
     and memory, writing/reading, 40–41  
     program-initiated operations, 27–28  
         address bus, 27–28  
         control signals, 28  
         data bus, 28  
         operations in, 27  
     *See also* Hardware model.
- Microprogramming, 9
- Micro-Trainer, 637–40
- Midicomputers, 12
- Minicomputers, 12
- MLTPLY subroutine, 270, 285
- Mnemonics, 18  
     translated into hex code, 139, 140–41
- Mode 0, 296, 298–300  
     Intel 8255A, 370, 373  
     interrupts, illustration of, 302–7  
     PIO, 333, 340
- Mode 1, 296  
     Intel 8255A, 374, 376–78  
     interrupts, illustration of, 307–13  
     PIO, 333, 339–40
- Mode 2, 296, 313–15  
     Intel 8255A, 378  
     PIO, 333, 343, 359–66
- Mode 3, PIO, 333, 344–48
- Mode instruction, 428
- Mode 0, 296, 298–300  
     Intel 8255A, 370, 373  
     interrupts, illustration of, 302–7

- Mode 0 (*continued*)  
 PIO, 333, 340  
 Modular approach, 234, 257–58  
   advantages of, 258  
   debugging, 258  
 Modulo-2 arithmetic, 629–30  
 MOSTEK MK4802, memory interfacing, 90–92  
 Motorola MC6800, 74–75  
 Motorola MC68000  
   asynchronous signals, 548  
   instruction set, 547–48  
   nonsegmented memory, 547  
   synchronous signals, 548  
 Motors, dc, I/O interfacing of, 119–20  
 MS-DOS, 158–59  
   COMMAND, 159  
   and cross-assemblers, 159  
   IBMDOS, 159  
   ROM-BIOS, 159  
 Multibyte numbers, adding, example program, 268–69  
 Multiple call subroutine, 249, 251  
 Multiple ending subroutine, 252, 281  
 Multiple interrupts, 318–22  
   interrupt vector technique, 320–22  
   polling method, 318–20  
 Multiplexing, 357–59  
 Multiplication, binary multiplication, 269–71  
 Multi-user systems, 32-bit microprocessors, 549  
 NAND gate, 635  
 National Semiconductor NSC800, 74  
 NEG instructions, 134, 179, 599  
 Nested subroutine, 251  
 NEXTKY, 355  
 Nibble, 16  
 NOADD, 271  
 Nonmaskable interrupt, 293, 295, 297, 315–17  
 Nonsegmented memory, Motorola MC68000, 547  
 Non-Volatile RAM, 44  
 NOP instructions, 149, 599  
 NOR gate, 634  
 Number conversion, 621–23  
   convert from binary into hexadecimal/octal, 622–23  
 NXTBIT 270, 273  
 Object code, 20  
 1-byte instruction, 130–31  
 1's and 2's complement, 134  
   instructions for, 179  
 Opcode fetch machine cycle, 62–64, 68, 464–65  
 Opcodes, 61, 70, 160  
   in instruction set, 130  
 Operand, 61, 160  
 Operating systems, 151, 155–56  
   CP/M, 155–58  
   MS-DOS, 158–59  
 ORG, assembler directive, 161, 163  
 OR gate, 633–34  
 OR instructions, 208, 209, 600  
 OTIR instructions, 149  
 OTI instructions, 602–3  
 OUTD instructions, 601  
 OUTI instructions, 602  
 OUT instructions, 133, 176, 600–601  
   output devices, interfacing, 102–4  
 Output  
   flowcharts, 191  
 Output devices, 5, 10  
 Output devices (interfacing)  
   LEDs  
     interfacing, 105–7  
     interfacing circuit, 106–7  
   Out instruction, 102–4  
     execution/timing of, 103–4  
     machine cycles, 103  
     steps in, 104–5  
 OUTR instructions, 601–2  
 Packed BCD, 253  
 PACK subroutine, 284, 285  
 Page zero, addressing mode, 138  
 Parallel Input/Output (PIO) device, 329  
   *See also* PIO  
 Parallel I/O mode, 101, 290–91, 411  
 Parallel-to-serial conversion, 291, 411  
 Parameter passing, 246, 248, 250  
 Parity check, 280  
   error checks, data transmission, 415–16  
 Parity/overflow flag, 56, 181, 182, 183, 225  
 Peripheral-mapped I/O, 101  
   examples of  
     interfacing, LEDs, 105–7  
     switches as input device, 110–11  
 Peripherals, 6, 289  
   asynchronous format, 290  
   interrupts, 293–324  
   programmable interface devices, 329–79  
   synchronous format, 290  
   *See also* Input/output (I/O)  
     interfacing; specific topics.  
 Phase shift keying, modems, 417  
 PIO  
   bidirectional data transfer, 359–66  
   compared to Intel 8255A, 378–79  
   control words, 337–39, 347–48  
   daisy chain priority, 343–44  
   handshake signals, 339–40, 343  
   initialization, 342–43  
   interrupt enable word, 341–42  
   interrupt vector, 341–42  
   keyboard/seven-segment display, 348–59  
   Mode 0, 333, 344–48  
   Mode 1, 333, 339–40  
   Mode 2, 333, 343, 359–66  
   Mode 3, 333, 344–48  
   signals, 335–36  
 Polling, 291  
   multiple interrupts, method for, 318–20  
 POP instructions, 234, 235, 236, 238, 239, 603  
 Positional weighing, 274  
 Power/frequency signals, 61  
 Preferred logic symbols, 633–35  
   AND gate, 634  
   inverters/bubble matching, 635  
   NAND gate, 635  
   NOR GATE, 634  
   OR gate, 633–34  
 PRN file, 157, 158, 165  
 Program, 4  
 Program assembly, CP/M, 158  
 Program coding, software design, 530  
 Program counter (PC), 56  
 Program development utilities, 156  
 Program documentation, 142–43  
 Program-initiated operations,  
   microprocessor unit (MPU), 27–28  
 Programmable interface devices, 329–79  
   bidirectional data transfer, 330, 343  
   handshake signals in, 330  
 Intel 8255A, 367–79  
   BSR mode, 366, 371, 373  
   compared to PIO, 378–79  
   control logic, 367, 369  
   control words, 369–70, 371, 373, 374, 378  
   handshake mode, 366  
   interfacing with A/D converter, 371–79  
   example program, 371–79  
   Mode 0, 370, 373  
   Mode 1, 374, 376–78  
   Mode 2, 378  
 PIO, 333–66  
   bidirectional data transfer, example program, 359–66  
   compared to Intel 8255A, 378–79  
   control words, 337–39, 347–48  
   daisy chain priority, 343–44  
   handshake signals, 339–40, 343

Programmable interface devices  
*(continued)*  
 initialization, 342–43  
 interrupt enable word, 341–42  
 interrupt vector, 341–42  
 keyboard/seven-segment display, example program, 348–59  
 Mode 0, 333, 340  
 Mode 1, 333, 339–40  
 Mode 2, 333, 343, 359–66  
 Mode 3, 333, 344–48  
 74LS234 bidirectional buffer, 331–33  
 signals, categories of, 335–36  
 Programmable Read-Only Memory (PROM), 43  
 decoding, 484–85  
 Programming model (Z80), 52–57  
 components for programming  
   accumulator, 52  
   alternate registers, 55  
   flag register, 53–55  
   general-purpose registers, 55  
   index registers (IX and IY), 56  
   interrupt vector register (I), 56  
   memory refresh register (R), 56  
   program counter (PC), 56  
   16-bit registers, as memory  
   pointers, 55  
   stack pointer (SP), 56  
   illustration of use, 57  
 Programming statements, 160  
 delimiters, 160  
 fields, 160  
 Pseudo operations, 161  
 PUSH instructions, 133, 234, 235, 236, 237, 239, 604  
 Random-Access Memory (RAM), 10  
 RD-423A, serial I/O mode, 418, 420  
 Reading the keyboard, software design, 528  
 Read-Only Memory (ROM), 10, 32, 42–43  
 Electrically Erasable PROM (EE-PROM), 43, 44  
 Erasable Programmable Read-Only Memory (EPROM), 43  
 masked ROM, 43  
 Programmable Read-Only Memory (PROM), 43  
 Read Registers, data transmission, Z80 and DART, 446, 451  
 Read/Write (R/W/M) Memory, 10, 32, 41–42, 82–83, 506–07  
 requirements of, 83

Real-time trace, debugging tool, 533  
 Receiver section  
   serial I/O mode  
     Intel 8251A, 431  
     Z80 and DART, 444, 446  
 Refreshing technique, dynamic  
   memory, 474, 478  
 Registers  
   addressing mode, 137, 138, 170, 172, 173  
   data copy  
     among registers, 170–72  
     between Z80 and memory, 172–75  
   register array, 9  
   register display, debugging tool, 533  
   register examine, dynamic  
     debugging, 196  
   register indirect, 138, 172, 173  
   rotate instructions, 213–15  
     *See also* Programming model (Z80).  
 Relative, addressing mode, 138  
 Relative jump instructions, 186–87, 213  
   conditional relative jump, 186  
   forward/backward jump, 186, 187  
   unconditional relative jump, 186, 187  
 REL files, 157, 158, 165  
 Repetitive techniques, 149, 189–90  
   counting, 189  
   example programs, 190–94  
   indexing, 189  
   looping, 189  
     setting flags, 189  
 Request acknowledgment, Z80, 60  
 RES instructions, 135, 149, 216, 604–5  
 Resource sharing, in-circuit emulator, 533  
 Restart (RST) instructions, 136, 297–98, 300, 302, 307  
   breakpoint technique, 322–24  
 RESULT subroutine, 273  
 RET instructions, 148, 241–42, 307, 605–6  
 RET subroutine, 241, 242, 244–45  
 RETI instructions, 307, 606  
 RETN instructions, 606  
 RL instructions, 149, 215, 608  
 RLA instructions, 148, 606–7  
 RLCA instructions, 148, 607  
 RLC instructions, 135, 149, 215, 608–9  
 RLD instructions, 609  
 ROM-BIOS, 159  
 Rotate, instructions for, 213–15  
 Rotate and shift, 135  
   instructions for, 215–16  
 RR instructions, 215, 611  
 RRA instructions, 135, 148, 609  
 RRC instructions, 215, 611–12  
 RRCA instructions, 148, 610–11  
 RRD instructions, 612  
 RS-232C  
   interfacing with Intel 8251A,  
     example program, 435–39  
   interfacing with Z80 and DART,  
     455–59  
   serial I/O mode, 418, 419–20  
     interface requirements, 420  
     line drivers, 420, 436  
     line receivers, 420, 436–37  
     signals, 419  
 RS-422A, serial I/O mode, 418, 420  
 RST instructions, 136, 138, 297–302, 613  
 S-100 bus, 554–56  
   interface, 555–56  
 SBC instructions, 266, 613–14  
 Scanned display  
   interfacing  
     hardware approach, 510–11  
     interfacing circuit, 507, 509  
     program for, 509–10  
 SCF instructions, 267, 615  
 Segment driver, 509  
 Semiconductor technology,  
   advancement of, 7, 11  
 SEND subroutine, 364  
 Sequential transfer, Direct Memory Access (DMA), 486  
 Serial I/O mode, 101, 290, 291, 411, 412–59, 413–15  
   asynchronous format, 413, 420–21  
   baud requirements, 414–15  
   duplex transmission, 413–14  
   error checks, 415–17  
     check sum, 416  
     cyclic redundancy check, 416–17  
     parity check, 415–16  
   framing, 413  
   hardware approach, 421  
   hardware approach (Intel 8251A), 425–39  
     control register, 428, 429  
     control signals, 428  
     data register, 428, 429  
     elements of chip, 425–27  
     interfacing RS-232 terminal,  
       example program, 435–39  
       programming of, 431–35  
     receiver section, 431  
     status register, 428  
     transmitter section, 430–31

Serial I/O mode (*continued*)  
hardware approach (Z80 SIO and DART), 439–57  
control signals, 441–43  
interfacing RS-232 terminal, example program, 455–57  
interrupts, 453–54, 455–57  
programming of, 446–52  
Read Registers, 446, 451  
receiver section, 444, 446  
transmitter section, 444  
versions of SIO, 439  
Write Registers, 446, 449–50  
interfacing requirements, 413  
modem, 417  
simple transmission, 413  
software approach, 420, 421, 422–25  
serial data reception, 425  
serial data transmission, 422–23  
standards used, 418–20  
  RS-232c, 418, 419–20  
  RS-422A, 418, 420  
  RS-423A, 418, 420  
synchronous format, 413  
Serial-to-parallel conversion, 411  
Service subroutine, 300, 400–401  
Set carry flag, 54, 267  
SET instructions, 135, 138, 216, 615  
Setting flags, 189  
Seven-segment LED  
  components of, 352–53  
  interfacing with keyboard, bidirectional data transfer, 348–59  
LED codes, 254, 255  
74LS245 bidirectional buffer, 331–33  
Signature analyzer, 534  
Signed numbers, 626–27  
Sign flag, 55, 181, 182, 183, 225  
Simple transmission, 413  
Simultaneous processing, Intel 8086/8088, 544  
Simultaneous transfer, Direct Memory Access (DMA), 486  
Single-board microcomputer, 14–15, 496–534  
  design alternatives related to, 520–23  
  display in, 497  
  execute in, 497  
instruments used  
  debugging tools, 533  
  in-circuit emulator, 532  
  logic state analyzer, 533–34  
  signature analyzer, 534  
keyboard in, 497

matrix keyboard, 512–20  
  hardware approach, encoder, use of, 519–20  
interfacing circuit, 513  
problem areas, 514  
program for, 513–17  
memory design, 504–7  
  design considerations, 504–6  
  EPROM memory, 506  
  Read/Write (R/W) Memory, 506–7  
Micro-Trainer, 637–40  
MPU design, 497–504  
  address bus, 498–99  
  bus request, 504  
  control bus, 501  
  data bus, 499, 501  
  interrupts, 504  
  power requirements, 502  
  reset circuits, 502  
  WAIT signal, 504  
prototype building/testing, 530–31  
scanned display, 507–11  
  hardware approach, hex decoder/driver, use of, 510–11  
  interfacing circuit, 507, 509  
  program for, 509–10  
software design modules, 526–31  
  display module, 527–28  
  function module, 528–30  
  initialization, 527  
  program coding, 530  
  reading the keyboard, 528  
specifications for, 496  
Single-chip microcomputers, 15, 538–40  
  Intel 2920 analog signal processor, 540  
  Intel MCS-51 single-chip microcomputer, 539–40  
  Z80 microcomputer, 538  
Single step, dynamic debugging, 196  
Single-user systems, 32-bit microprocessors, 549  
16-bit microprocessors, 7, 9, 540, 542–48  
  Intel 80186/80286, 548  
  Intel 8086/8088, 542–45  
    coprocessing, 545  
    instruction set, 545  
    memory segmentation, 542–44  
    simultaneous processing, 544  
Motorola MC68000, 546–47  
asynchronous signals, 548  
instruction set, 547–48  
nonsegmented memory, 547  
synchronous signals, 548  
objectives of, 542  
Z8000, 545–47  
16-bit operations, 264–67  
instructions  
  arithmetic instructions, 265–67  
  bit manipulation, 267  
  data copy instructions, 264  
  exchange instructions, 264–65  
16-bit registers, 55  
SLA instructions, 149, 215, 216, 616  
Small-Scale Integration (SSI), 7  
Software, 4  
Software design modules  
  display module, 527–28  
  function module, 528–30  
  initialization, 527  
  program coding, 530  
  reading the keyboard, 528  
Software design project, 282–85  
Software development system, 151, 152  
  components of, 152–53  
floppy disks, 152–54  
hard disk, 154  
hardware, 152  
operating systems, 155–56  
  CP/M, 155–58  
  MS-DOS, 158–59  
Source, 132  
Source code, 20  
Source file, 158  
Source program error messages, 164  
Special signals, Z80, 60  
SRA instructions, 616  
SRL instructions, 215, 617  
Stack, 233, 234–41  
  Exchange, instructions for, 239  
  flags, examining/manipulating, example program, 239–41  
  instructions for, 235–38  
  stack pointer, 234–35  
Stack pointer (SP), 56, 398  
Static debugging, 194, 196  
Static memory, interfacing, 470  
STORE subroutine, 279  
SUB instructions, 134, 147, 178, 617–18  
Subroutines, 241–58  
  BCD counter, example program, 252–57  
  documentation of, 250  
  instructions for, 241–42

- Subroutines (*continued*)**
- conditional instruction, 241–42, 243
  - unconditional instruction, 241–42, 244
  - modular approach, 234, 257–58
  - multiple call subroutine, 249, 251
  - multiple ending subroutine, 252
  - nested subroutine, 251
  - parameter passing, 246, 248, 250
  - traffic signal controller, example program, 245–49
- Subtraction**, 134, 627
- binary subtraction, 626
  - decimal subtraction, 623–25
  - instructions for, 177–78
  - 16-bit operations, subtraction with carry, 266–67
- SUM subroutine**, 276
- Switches**, 210–12
- Synchronous format**, 290, 413
- synchronous signals, Motorola MC68000, 548
- System display**, 497
- Table look-up technique**, 254
- Temperature sensor, I/O interfacing of**, 117–19
- Temporary storage, flowcharts**, 191
- Terminal error messages**, 164
- 32-bit microprocessors**, 7, 9, 548–50
- 80386 microprocessor, 549
  - single-user systems, 549
  - Zilog Z80000 microprocessor, 549
- 3-byte instruction**, 131
- Time delays**, 218–20
- waveform generating, example program, 218–21
- Timer mode**, 395
- baud generator design, example program, 395–97
- See also* Counter/timer circuit (CTC).
- Traffic signal controller, example program**, 245–49
- Transmitter section**
- serial I/O mode
  - Intel 8251A, 430–31
  - Z80 and DART, 444
- Tri-state buffer**, 298
- 74LS245 bidirectional buffer, 331–33
- T-state**, 62
- 2732 EPROM, memory interfacing, 87–90
  - 2-byte instruction, 131
- Unconditional absolute jump**, 183
- Unconditional data transfer**, 291
- Unconditional relative jump**, 186, 187
- Universal Asynchronous Receiver/Transmitter (UART)**, 551, 554
- Universal Synchronous/Asynchronous Receiver/Transmitter (USART)**, 425
- Unpacking**, 253, 255
- UNPACK subroutine**, 255, 257, 275–76, 282
- UPDATE subroutine**, 256, 257
- Upward software compatible**, 7
- User display**, 497
- Utility programs, categories of**, 156
- Very-large-scale integration (VLSI)**, 7, 8
- WAIT signal**
- data transfer, 291
  - in MPU design, 504
- Wait states**
- memory interfacing, 464–68
  - generation of, 466–68
- WAIT subroutine**, 256
- Waveforms, generating, example program**, 218–21
- Words**, 5, 130
- definition of, 5, 16
- Write Registers, data transmission**, Z80 and DART, 446, 449–50
- XOR instructions**, 135, 208, 209, 618–19
- Z80**
- assembly language programming, 129–49
  - hardware model, 57–61
  - input/output (I/O) interfacing, 101–22
  - instructions, 61
  - opcode, 61, 70
  - operand, 61
  - machine cycles, 61–67, 70–71
  - definition of, 62
  - memory read machine cycle, 65–67, 68
  - memory write cycle, 67–68
  - opcode fetch machine cycle, 62–64, 68
  - T-states in, 62, 70
  - memory interfacing, 83–97
  - reading from/writing to memory, 40–41, 83
- programming model**, 52–57
- questions related to**, 69–71
- See also* specific topics.
- Z80 assembly language**, 18–19
- Z80 machine language**, 17–18
- Z80 SIO and DART**
- serial I/O
  - control signals, 441–43
  - interfacing RS-232 terminal, 455–57
  - interrupts, 453–54, 455–57
  - programming of, 446–52
  - Read Registers, 446, 451
  - receiver section, 444, 446
  - transmitter section, 444
  - versions of SIO, 439
  - Write Registers, 446, 449–50
- Z80000 microprocessor**, 549
- Zero flag**, 55, 183, 225
- Zero Power RAM**, 43–44

ISBN 0-675-20540-9

A standard linear barcode representing the ISBN number 0-675-20540-9.

9 0000 >



9 780675 205405