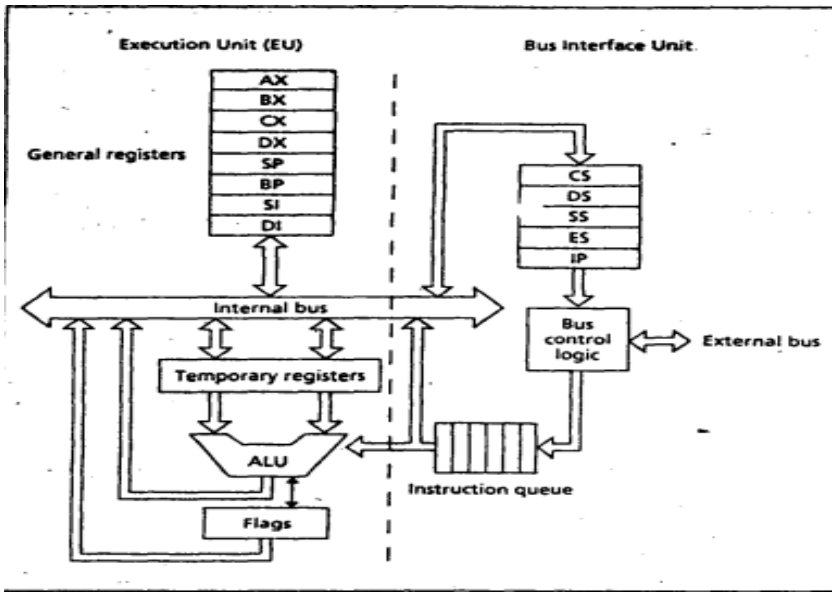
MID PART

CHAPTER-1

Example 1.1 Suppose a processor uses 20 bits for an address. A bit can have two possible values, so In a 20-bit address there can be 2^20 .. 1,048,576 different values or 1 megabyte or 1 MB.



How an Instruction Is executed?

First of all, a machine instruction has two parts: an opcode and operands. The opcode specifies the type of operation, and the operands are often given as memory addresses to the data to be operated on.

The CPU goes through the following steps to execute a machine instruction:

1. Fetch an instruction from memory.
2. Decode the Instruction to determine the operation.
3. Fetch data from memory if necessary.

Execute

4. Perform the operation on the data.
5. Store the result in memory if needed.

To see what this entails, let's trace through the execution of a typical machine language instruction for the 8086. Suppose we look at the instruction that adds the contents of register AX to the contents of the memory word at address 0. The CPU actually adds the two numbers in the ALU and then stores the result back to memory word O. The machine code is 00000001 00000110 00000000 00000000 Before execution, we assume that the first byte of the Instruction is stored at the location indicated by the IP.

1. Fetch the instruction. To start the cycle, the BIU places a memory read request on the control bus and the address of the instruction on the address bus. Memory responds by sending the contents of the location specified-namely, the instruction code just given-over the data bus, Because the instruction code Is four bytes and the 8086 c:an only read a word at a time, this

involves two read operations. The CPU accepts the data and adds four to the JP so that the IP will contain the address of the next instruction.

2. Decode the instruction. On receiving the Instruction, a decoder circuit in the EU decodes the instruction and determines that it is an ADD operation involving the word at address 0.

3. Fetch data from memory. The EU informs the BIU to get the contents of memory word 0. The BIU sends address 0 over the address bus and a memory read request is again sent over the control bus. The contents of memory word 0 arc sent back over the data bus to the EU and arc placed In a holding register.

4. Perform the operation. The contents of the holding register and the AX register arc are sent to the ALU circuit, which performs the required addition and holds the sum.

5. Store the result. The EU directs the BIU to store the sum at address 0. To do so, the llIU sends out a memory write request over the control bus, the address 0 over the address bus, and the sum 10 be stored over the data bus. The previous contents of memory word 0 are overwritten by the sum. The cycle is now repeated for the instruction whose address is cont;iined in the II'

CHAPTER-2

Converting Binary and Hex to Decimal

Consider the hex number 82AD. It can be written as

$$8A2Dh = 8 \times 16^3 + A \times 16^2 + 2 \times 16^1 + D \times 16^0$$
$$= 8 \times 16^3 + 10 \times 16^2 + 2 \times 16^1 + 13 \times 16^0 = 35373d$$

Similarly, the binary number 11101 may be written as

$$11101b = 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 29d$$

Converting Decimal to Binary and Hex

Suppose we want to convert 11172 to hex. The answer 2BA4h may be obtained as follows. First, divide 11172 by 16. We get a quotient of 698 and a remainder of 4. Thus

$$11172 = 698 \times 16 + 4$$

The remainder 4 is the unit's digit in hex representation of 11172. Now divide 698 by 16. The quotient is 43, and the remainder is 10 = Ah. Thus

$$698 = 43 \times 16 + Ah$$

The remainder Ah is the sixteen's digit in the hex representation of 11172. We just continue this process, each time dividing the most recent quotient by 16, until we get a 0 quotient. The remainder each time is a digit in the hex representation of 11172. Here are the calculations:

$$11172 = 698 \times 16 + 4$$
$$698 = 43 \times 16 + 10(Ah)$$
$$43 = 2 \times 16 + 11(Bh)$$
$$2 = 0 \times 16 + 2$$

Now just convert the remainders to hex and put them together in reverse order to get 2BA4h.

This same process may be used to convert decimal to binary. The only difference is that we repeatedly divide by 2.

Example 2.3 Convert 95 to binary.

Solution:

$$95 = 47 \times 2 + 1$$
$$47 = 23 \times 2 + 1$$
$$23 = 11 \times 2 + 1$$
$$11 = 5 \times 2 + 1$$
$$5 = 2 \times 2 + 1$$
$$2 = 1 \times 2 + 0$$
$$1 = 0 \times 2 + 1$$

Taking the remainders in reverse order, we get 95 = 1011111b.

2.4 How Integers Are Represented in the Computer

The hardware of a computer necessarily restricts the size of numbers that can be stored in a register or memory location. In this section, we will see how integers can be stored in an 8-bit byte or a 16-bit word. In Chapter 18 we talk about how real numbers can be stored.

In the following, we'll need to refer to two particular bits in a byte or word: the **most significant bit**, or **msb**, is the leftmost bit. In a word, the msb is bit 15; in a byte, it is bit 7. Similarly, the **least significant bit**, or **lsb**, is the rightmost bit; that is, bit 0.

2.4.1 Unsigned Integers

An **unsigned integer** is an integer that represents a magnitude, so it is never negative. Unsigned integers are appropriate for representing quantities that can never be negative, such as addresses of memory locations, counters, and ASCII character codes (see later). Because unsigned integers are by definition nonnegative, none of the bits are needed to represent the sign, and so all 8 bits in a byte, or 16 bits in a word, are available to represent the number.

The largest unsigned integer that can be stored in a byte is 11111111 = FFh = 255. This is not a very big number, so we usually store integers in words. The biggest unsigned integer a 16-bit word can hold is 1111111111111111 = FFFFh = 65535. This is big enough for most purposes. If not, two or more words may be used.

Note that if the least significant bit of an integer is 1, the number is odd, and it's even if the lsb is 0.

2.4.2 Signed Integers

A **signed integer** can be positive or negative. The most significant bit is reserved for the sign: 1 means negative and 0 means positive. Negative integers are stored in the computer in a special way known as **two's complement**. To explain it, we first define **one's complement**, as follows.

One's Complement

The one's complement of an integer is obtained by complementing each bit; that is, replace each 0 by a 1 and each 1 by a 0. In the following, we assume numbers are 16 bits.

Example 2.6 Find the one's complement of 5 = 0000000000000101.

Solution:

```
                    5 = 0000000000000101
One's complement of 5 = 1111111111111010
```

Note that if we add 5 and its one's complement, we get 1111111111111111.

Two's Complement

To get the two's complement of an integer, just add 1 to its one's complement.

Example 2.7 Find the two's complement of 5.

Solution: From above,

```
one's complement of 5 =  1111111111111010
                                        + 1
-------------------------------------------
two's complement of 5 =  1111111111111011 = FFFBh
```

Now look what happens when we add 5 and its two's complement:

```
                    5 =  0000000000000101
+ two's complement of 5 =  1111111111111011
-------------------------------------------
                         10000000000000000
```

We end up with a 17-bit number. Because a computer word circuit can only hold 16 bits, the 1 carried out from the most significant bit is lost, and the 16-bit result is 0. As 5 and its two's complement add up to 0, the two's complement of 5 must be a correct representation of –5.

It is easy to see why the two's complement of any integer N must represent $-N$: Adding N and its one's complement gives 16 ones; adding 1 to this produces 16 zeros with a 1 carried out and lost. The result stored is always 0000000000000000.

The following example shows what happens when a number is complemented two times.

Example 2.8 Find the two's complement of the two's complement of 5.

Solution: We would guess that after complementing 5 two times, the result should be 5. To verify this, from above,

```
                two's complement of 5 = 1111111111111011
one's complement of 1111111111111011 = 0000000000000100
                                                     + 1
-------------------------------------------------------
two's complement of 1111111111111011 = 0000000000000101 = 5
```

Example 2.9 Show how the decimal integer –97 would be represented (a) in 8 bits, and (b) in 16 bits. Express the answers in hex.

Solution: A decimal-to-hex conversion using repeated division by 16 yields

```
97 = 6 × 16 + 1
 6 = 0 × 16 + 6
```

Thus 97 = 61h. To represent –97, we need to express 61h in binary and take the two's complement.

Subtraction as Two's Complement Addition

The advantage of two's complement representation of negative integers in the computer is that subtraction can be done by bit complementation and addition, and circuits that add and complement bits are easy to design.

Example 2.10 Suppose AX contains 5ABCh and BX contains 21FCh. Find the difference of AX minus BX by using complementation and addition.

Solution:

```
AX contains  5ABCh = 0101 1010 1011 1100
BX contains  21FCh = 0010 0001 1111 1100
             5ABCh = 0101 1010 1011 1100
+ one's complement of 21FCh = 1101 1110 0000 0011
                                                + 1
Difference = 1 0011 1000 1100 0000 = 38C0h
```

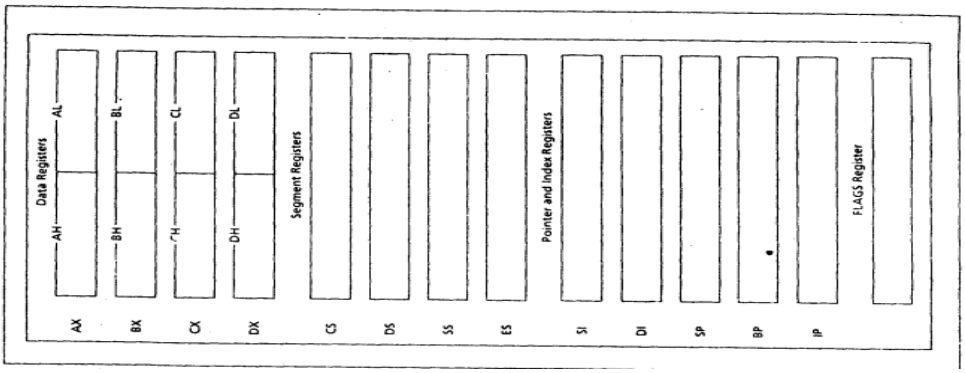
A one is carried out of the most significant bit and is lost. The answer stored, 38C0h, is correct, as may be verified by hex subtraction.

SUMMARY

- Numbers are represented in different ways, according to the basic symbols used. The binary system uses two symbols, 0 and 1. The decimal system uses 0–9. The hexadecimal system uses 0–9, A–F.
- Binary and hex numbers can be converted to decimal by a process of nested multiplication.
- A hex number can be converted to decimal by a process of repeated division by 16; similarly, a binary number can be converted to decimal by a process of repeated division by 2.
- Hex numbers can be converted to binary by converting each hex digit to binary; binary numbers are converted to hex by grouping the bits in fours, starting from the right, and converting each group to a hex digit.
- The process of adding and subtracting hex and binary numbers is the same as for decimal numbers, and can be done with the help of the appropriate addition table.
- Negative numbers are stored in two's complement form. To get the two's complement of a number, complement each bit and add 1 to the result.
- If A and B are stored integers, the processor computes A – B by adding the two's complement of B to A.
- The range of unsigned integers that can be stored in a byte is 0–255; in a 16-bit word, it is 0–65535.
- For signed numbers, the most significant bit is the sign bit; 0 means positive and 1 means negative. The range of signed numbers that can be stored in a byte is –128 to 127; in a word, it is –32768 to 32767.
- The unsigned decimal interpretation of a word is obtained by converting the binary value to decimal. If the sign bit is 0, this is also the signed decimal interpretation. If the sign bit is 1, the signed decimal interpretation may be obtained by subtracting 65536 from the unsigned decimal interpretation.
- The standard encoding scheme for characters is the ASCII code.
- A character requires seven bits to code, so it can be stored in a byte.
- The IBM screen controller can generate a character for each of the 256 possible numbers that can be stored in a byte.

CHAPTER-3

In general, d11ta registers hold data for an operation, address registers hold the address of instruction or data, and a status register keeps the current status of the processor.



As noted in Chapter 1, information inside the microprocessor is stored in registers. The registers are classified according to the functions they perform. In general, *data registers* hold data for an operation, *address registers* hold the address of an instruction or data, and a *status register* keeps the current status of the processor.

The 8086 has four general data registers; the address registers are divided into *segment*, *pointer*, and *index registers*; and the status register is called the *FLAGS register*. In total, there are fourteen 16-bit registers, which we now briefly describe. See Figure 3.1. *Note:* You don't need to memorize the special functions of these registers at this time. They will become familiar with use.

AX (Accumulator Register)

AX is the preferred register to use in arithmetic, logic, and d
transfer instructions because its use generates the shortest machine co

In multiplication and division operations, one of the numbers involved must be in AX or AL. Input and output operations also require the use of AL and AX.

BX (Base Register)

BX also serves as an address register; an example is a table look-up instruction called XLAT (translate).

CX (Count Register)

Program loop constructions are facilitated by the use of CX, which serves as a loop counter. Another example of using CX as counter is REP (repeat), which controls a special class of instructions called *string operations*. CL is used as a count in instructions that shift and rotate bits.

DX (Data Register)

DX is used in multiplication and division. It is also used in I/O operations

Memory Segment

A **memory segment** is a block of 2^{16} (or 64 K) consecutive memory bytes. Each segment is identified by a **segment number**, starting with 0. A segment number is 16 bits, so the highest segment number is FFFFh.

Within a segment, a memory location is specified by giving an **offset**. This is the number of bytes from the beginning of the segment. With a 64-KB segment, the offset can be given as a 16-bit number. The first byte in a segment has offset 0. The last offset in a segment is FFFFh.

Segment:Offset Address

A memory location may be specified by providing a segment number and an offset, written in the form *segment:offset*; this is known as a **logical address**. For example, A4FB:4872h means offset 4872h within segment A4FBh. To obtain a 20-bit physical address, the 8086 microprocessor first shifts the segment address 4 bits to the left (this is equivalent to multiplying by 10h), and then adds the offset. Thus the physical address for A4FB:4872 is

```
  A4FB0h
+ 4872h
--------
  A9822h   (20-bit physical address)
```

3.2.4 Pointer and Index Registers: SP, BP, SI, DI

The registers SP, BP, SI, and DI normally point to (contain the offset addresses of) memory locations. Unlike segment registers, the pointer and index registers can be used in arithmetic and other operations.

SP (Stack Pointer)

The SP (stack pointer) register is used in conjunction with SS for accessing the stack segment. Operations of the stack are covered in Chapter 8.

BP (Base Pointer)

The BP (base pointer) register is used primarily to access data on the stack. However, unlike SP, we can also use BP to access data in the other segments.

SI (Source Index)

The SI (source index) register is used to point to memory locations in the data segment addressed by DS. By incrementing the contents of SI, we can easily access consecutive memory locations.

DI (Destination Index)

The DI (destination index) register performs the same functions as SI. There is a class of instructions, called *string operations*, that use DI to access memory locations addressed by ES.

3.2.5 Instruction Pointer: IP

The memory registers covered so far are for data access. To access instructions, the 8086 uses the registers CS and IP. The CS register contains the segment number of the next instruction, and the IP contains the offset. IP is updated each time an instruction is executed so that it will point to the next instruction. Unlike the other registers, the IP cannot be directly manipulated by an instruction; that is, an instruction may not contain IP as its operand.

CHAPTER-4

Table 4.2 Legal Combinations of Operands for MOV and XCHG

MOV

	Destination Operand			
Source Operand	*General register*	*Segment register*	*Memory location*	*Constant*
General register	yes	yes	yes	no
Segment register	yes	no	yes	no
Memory location	yes	yes	no	no
Constant	yes	no	yes	no

XCHG

	Destination Operand	
Source Operand	*General register*	*Memory location*
General register	yes	yes
Memory location	yes	no

Table 4.3 Legal Combinations of Operands for ADD and SUB

	Destination Operand	
Source Operand	*General register*	*Memory location*
General register	yes	yes
Memory location	yes	no
Constant	yes	yes

INC (increment) is used to add 1 to the contents of a register or memory location and **DEC** (decrement) subtracts 1 from a register or memory location. The syntax is

```
INC destination
DEC destination
```

```
.MODEL SMALL
.STACK 100H
.DATA
;data definitions go here
.CODE
MAIN    PROC
;instructions go here
MAIN    ENDP
;other procedures go here
END     MAIN
```

The INT Instruction

To invoke a DOS or BIOS routine, the **INT** (interrupt) instruction is used. It has the format

```
INT interrupt_number
```

MOV AH, 1 ; input key function
INT 21h ; ASCII code in AL

MOV AH,2 ; display character instruction
MOV DL, '?' ; character is '?'
INT 21hv ; display character

Program Listing PGM4_1.ASM

```
TITLE PGM4_1: ECHO PROGRAM
.MODEL SMALL
.STACK 100H
.CODE
MAIN PROC
;display prompt
        MOV AH,2            ;display character function
```

ng and Running a Program

```
        MOV DL,'?'          ;character is '?'
        INT 21H             ;display it
;input a character
        MOV AH,1            ;read character function
        INT 21H             ;character in AL
        MOV BL,AL           ;save it in BL
;go to a new line
        MOV AH,2            ;display character function
        MOV DL,0DH          ;carriage return
        INT 21H             ;execute carriage return
        MOV DL,0AH          ;line feed
        INT 21H             ;execute line feed
;display character
        MOV DL,BL           ;retrieve character
        INT 21H             ;and display it
;return to DOS
        MOV AH,4CH          ;DOS exit function
        INT 21H             ;exit to DOS
MAIN    ENDP
        END  MAIN
```

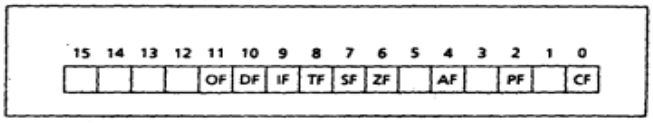
Because no variables were used, the data segment was omitted.

Terminating a Program

The last two lines in the MAIN procedure require some explanation. When a program terminates, it should return control to DOS. This can be accomplished by executing INT 21h, function 4Ch.

<< PROGRAM

CHAPTER-5



The Status Flags

As stated earlier, the processor uses the status flags to reflect the result of an operation. For example, if SUB AX,AX is executed, the zero flag becomes 1, thereby indicating that a zero result was produced. Now let's get to know the status flags.

Carry Flag (CF) detects unsigned overflow

CF = 1 if there is a carry out from the most significant bit (msb) on addition, or there is a borrow into the msb on subtraction; otherwise, it is 0. CF is also affected by shift and rotate instructions (Chapter 7).

Parity Flag (PF)

PF = 1 if the low byte of a result has an even number of one bits (even parity). It is 0 if the low byte has odd parity. For example, if the result of a word addition is FFFEh, then the low byte contains 7 one bits, so PF = 0.

Table 5.1 Flag Names and Symbols

Status Flags		
Bit	***Name***	***Symbol***
0	Carry flag	CF
2	Parity flag	PF
4	Auxiliary carry flag	AF
6	Zero flag	ZF
7	Sign flag	SF
11	Overflow flag	OF
Control Flags		
Bit	***Name***	***Symbol***
8	Trap flag	TF
9	Interrupt flag	IF
10	Direction flag	DF

Auxiliary Carry Flag (AF)

AF = 1 if there is a carry out from bit 3 on addition, or a borrow into bit 3 on subtraction. AF is used in binary-coded decimal (BCD) operations (Chapter 18).

Zero Flag (ZF)

ZF = 1 for a zero result, and ZF = 0 for a nonzero result.

Sign Flag (SF)

SF = 1 if the msb of a result is 1; it means the result is negative if you are giving a signed interpretation. SF = 0 if the msb is 0.

Overflow Flag (OF)

OF = 1 if signed overflow occurred, otherwise it is 0. The meaning of overflow is discussed next.

How the Processor Indicates Overflow

The processor sets OF = 1 for signed overflow and CF = 1 for unsigned overflow. It is then up to the program to take appropriate action, and if nothing is done immediately the result of a subsequent instruction may cause the overflow flag to be turned off.

In determining overflow, the processor does not interpret the result as either signed or unsigned. The action it takes is to use both interpretations for each operation and to turn on CF or OF for unsigned overflow or signed overflow, respectively.

It is the programmer who is interpreting the results. If a signed interpretation is being given, then only OF is of interest and CF can be ignored; conversely, for an unsigned interpretation CF is important but not OF.

How the Processor Determines that Overflow Occurred

Many instructions can cause overflow; for simplicity, we'll limit the discussion to addition and subtraction.

Unsigned Overflow

On addition, unsigned overflow occurs when there is a carry out of the msb. This means that the correct answer is larger than the biggest unsigned number; that is, FFFFh for a word and FFh for a byte. On subtraction, unsigned overflow occurs when there is a borrow into the msb. This means that the correct answer is smaller than 0.

Signed Overflow

On addition of numbers with the same sign, signed overflow occurs when the sum has a different sign. This happened in the preceding example when we were adding 7FFFh and 7FFFh (two positive numbers), but got FFFEh (a negative result).

Subtraction of numbers with different signs is like adding numbers of the same sign. For example, A − (−B) = A + B and −A −(+B) = −A + −B. Signed overflow occurs if the result has a different sign than expected. See example 5.3, in the next section.

In addition of numbers with different signs, overflow is impossible, because a sum like A + (−B) is really A − B, and because A and B are small enough to fit in the destination, so is A − B. For exactly the same reason, subtraction of numbers with the same sign cannot give overflow.

Actually, the processor uses the following method to set the OF: If the carries into and out of the msb don't match—that is, there is a carry into the msb but no carry out, or if there is a carry out but no carry in—then signed overflow has occurred, and OF is set to 1. See example 5.2, in the next section.

Example 5.5 MOV AX, −5

Solution: The result stored in AX is −5 = FFFBh.

None of the flags are affected by MOV.

Example 5.6 NEG AX, where AX contains 8000h.

Solution:

```
                8000h = 1000 0000 0000 0000
one's complement      = 0111 1111 1111 1111
                                       + 1
                        -------------------
                        1000 0000 0000 0000 = 8000h
```

The result stored in AX is 8000h.

SF = 1, PF = 1, ZF = 0.

CF = 1, because for NEG CF is always 1 unless the result is 0.

OF = 1, because the result is 8000h; when a number is negated, we would expect a sign change, but because 8000h is its own two's complement, there is no sign change.

In the next section, we introduce a program that lets us see the actual setting of the flags.

Summary

- The FLAGS register is one of the registers in the 8086 microprocessor. Six of the bits are called status flags, and three are control flags.
- The status flags reflect the result of an operation. They are the carry flag (CF), parity flag (PF), auxiliary carry flag (AF), zero flag (ZF), sign flag (SF), and overflow flag (OF).
- CF is 1 if an add or subtract operation generates a carry out or borrow into the most significant bit-position; otherwise, it is 0.
- PF is 1 if there is an even number of 1 bits in the result; otherwise, it is 0.
- AF is 1 if there is a carry out or borrow into bit 3 in the result; otherwise, it is 0.
- ZF is 1 if the result is 0; otherwise, it is 0.
- SF is 1 if the most significant bit of the result is 1; otherwise, it is 0.
- OF is 1 if the correct signed result is too big to fit in the destination; otherwise, it is 0.

- Overflow occurs when the correct result is outside the range of values represented by the computer. Unsigned overflow occurs if an unsigned interpretation is being given to the result, and signed overflow happens if a signed interpretation is being given.
- The processor uses CF and OF to indicate overflow: CF = 1 means that unsigned overflow occurred, and OF = 1 indicates signed overflow.
- The processor sets CF if there is a carry out of the msb on addition, or a borrow into the msb on subtraction. In the latter case, this means that a larger unsigned number is being subtracted from a smaller one.
- The processor sets OF if there is a carry into the msb but no carry out, or if there is a carry out of the msb but no carry in.
- There is another way to tell whether signed overflow occurred on addition and subtraction. On addition of numbers of like sign, signed overflow occurs if the result has a different sign; subtraction of numbers of different sign is like adding numbers of the same sign, and signed overflow occurs if the result has a different sign.
- On addition of numbers of different sign, or subtraction of numbers of the same sign, signed overflow is impossible.
- Generally the execution of each instruction affects the flags, but some instructions don't affect any of the flags, and some affect only some of the flags.
- The settings of the flags is part of the DEBUG display.
- The DEBUG program may be used to trace a program. Some of its commands are "R", to display registers; "T", to trace an instruction; and "G", to execute a program.

CHAPTER-6

Program Listing PGM6_1.ASM

```
1:  TITLE   PGM6_1: IBM CHARACTER DISPLAY
2:  .MODEL          SMALL
3:  .STACK          100H
4:  .CODE
5:  MAIN            PROC
6:          MOV     AH,2            ;display char function
7:          MOV     CX,256          ;no. of chars to display
8:          MOV     DL,0            ;DL has ASCII code of null cha
9:  PRINT_LOOP:
```



```
10:             INT     21h             ;display a char
11:             INC     DL              ;increment ASCII code
12:             DEC     CX              ;decrement counter
13:             JNZ     PRINT_LOOP      ;keep going if CX not 0
14: ;DOS exit
15:             MOV     AH,4CH          jei line e PRINT_LOOP likha ache
16:             INT     21h             oikhane jaabe until CX = 0
17: MAIN        ENDP
18:             END     MAIN
```

<<< JUMP

Table 6.1 Conditional Jumps

Signed Jumps

Symbol	Description	Condition for Jumps
JG/JNLE	jump if greater than jump if not less than or equal to	ZF = 0 and SF = OF
JGE/JNL	jump if greater than or equal to jump if not less than or equal to	SF = OF
JL/JNGE	jump if less than jump if not greater than or equal	SF <> OF
JLE/JNG	jump if less than or equal jump if not greater than	ZF = 1 or SF <> OF

Unsigned Conditional Jumps

Symbol	Description	Condition for Jumps
JA/JNBE	jump if above	CF = 0 and ZF = 0
	jump if not below or equal	
JAE/JNB	jump if above or equal jump if not below	CF = 0
JB/JNAE	jump if below jump if not above or equal	CF = 1
JBE/JNA	jump if equal jump if not above	CF = 1 or ZF = 1

Single-Flag Jumps

Symbol	Description	Condition for Jumps
JE/JZ	jump if equal jump if equal to zero	ZF = 1
JNE/JNZ	jump if not equal jump if not zero	ZF = 0
JC	jump if carry	CF = 1
JNC	jump if no carry	CF = 0
JO	jump if overflow	OF = 1
JNO	jump if no overflow	OF = 0
JS	jump if sign negative	SF = 1
JNS	jump if nonnegative sign	SF = 0
JP/JPE	jump if parity even	PF = 1
JNP/JPO	jump if parity odd	PF = 0

CMP AX,BX

JG BELOW

>> if AX Is greater than BX (in a signed sense), then JG fiµmp if greater than) transfers to BELOW.

Signed Versus Unsigned Jumps

Each of the signed jumps corresponds to an analogous unsigned jump; for example, the signed jump JG and the unsigned jump JA. Whether to use a signed or unsigned jump depends on the interpretation being given. In fact, Table 6.1 shows that these jumps operate on different flags: the signed jumps operate on ZF, SF, and OF, while the unsigned jumps operate on ZF and CF. Using the wrong kind of jump can lead to incorrect results.

For example, suppose we're giving a signed interpretation. If AX = 7FFFh, BX = 8000h, and we execute

```
CMP   AX,BX
JA    BELOW
```

Example 6.3 Suppose AL and BL contain extended ASCII characters. Display the one that comes first in the character sequence.

Solution:

```
IF AL <= BL
 THEN
  display the character in AL
 ELSE
  display the character in BL
END_IF
```

It can be coded like this:

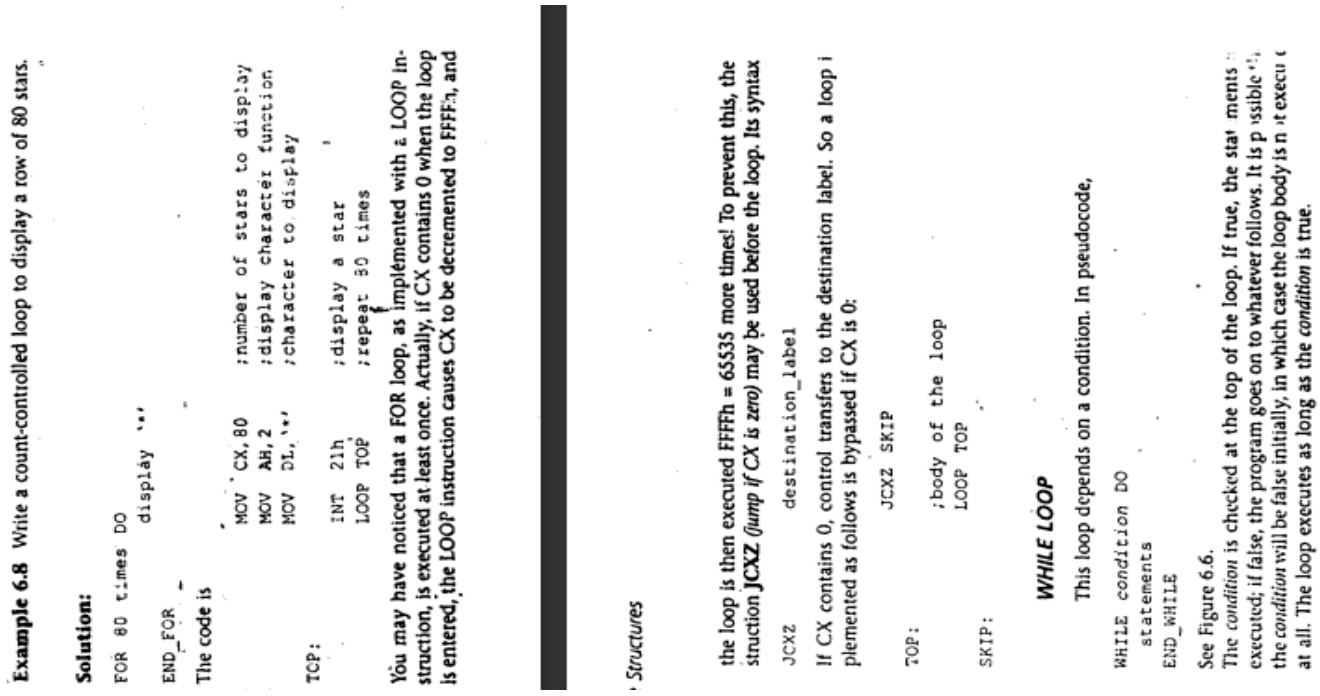
```
                MOV  AH,2       ;prepare to display
;if AL <= BL
                CMP  AL,BL      ;AL <= BL?
                JNBE  ELSE_     ;no, display char in BL
;then                           ;AL <= BL
                MOV DL,AL       ;move char to be displayed
                JMP DISPLAY     ;go to display
ELSE_:                          ;BL < AL
                MOV DL,BL
```

<< IF ELSE

Example 6.6: Read a character, and if it's an uppercase letter, display it.

```
;read a character
                MOV   AH,1        ;prepare to read
                INT   21H         ;char in AL
;if ('A' <= char) and (char <= 'Z')
                CMP   AL,'A'      ;char >= 'A'?
                JNGE  END_IF      ;no, exit
                CMP   AL,'Z'      ;char <= 'Z'?
                JNLE  END_IF      ;no, exit
;then display char
                MOV   DL,AL       ;get char
                MOV   AH,2        ;prepare to display
                INT   21H         ;display char
END_IF:
```

LOOP:



Example 6.8 Write a count-controlled loop to display a row of 80 stars.

Solution:

```
FOR 80 times DO
        display '*'
END_FOR
```

The code is

```
        MOV  CX,80     ;number of stars to display
        MOV  AH,2      ;display character function
        MOV  DL,'*'    ;character to display
TOP:    INT  21h       ;display a star
        LOOP TOP       ;repeat 80 times
```

You may have noticed that a FOR loop, as implemented with a LOOP instruction, is executed at least once. Actually, if CX contains 0 when the loop is entered, the LOOP instruction causes CX to be decremented to FFFFh, and the loop is then executed FFFFh = 65535 more times! To prevent this, the instruction **JCXZ** *(jump if CX is zero)* may be used before the loop. Its syntax

```
JCXZ    destination_label
```

If CX contains 0, control transfers to the destination label. So a loop implemented as follows is bypassed if CX is 0:

```
        JCXZ SKIP
TOP:    ;body of the loop
        LOOP TOP
SKIP:
```

WHILE LOOP

This loop depends on a condition. In pseudocode,

```
WHILE condition DO
  statements
END_WHILE
```

See Figure 6.6.

The *condition* is checked at the top of the loop. If true, the statements are executed; if false, the program goes on to whatever follows. It is possible that the *condition* will be false initially, in which case the loop body is not executed at all. The loop executes as long as the *condition* is true.

Structures

Another one;

```
WHILE_:
        CMP   AL,0DH      ;CR?
        JE    END_WHILE   ;yes, exit
        INC   DX          ;not CR, increment count
        INT   21H         ;read a character
        JMP   WHILE_      ;loop back
END_WHILE:
```

LAB TASKS:

TASK-1

In the first task we had to get the product of the series 1,3,5,7,9,11 .


```
.MODEL SMALL
.STACK 100h

.DATA
  start dw 1d   ; start of sequence
  ans dw 0d ; answer will be stored here

.CODE
 MAIN PROC

   MOV AX,@DATA ; load the data segment
   MOV DS,AX

   MOV CX,6    ; counter == 6
   MOV AX,1d   ; It will be multiplied

   LOOP_START:
    MUL start ; AX = AX*START
    ADD start,2 ; start = start + 2
    MOV ans,AX  ; AX will be stored in ans

    LOOP LOOP_START ; loop until CX == 0

   MOV DX,ans

   MOV AH, 4Ch
   INT 21h

   MAIN ENDP
 END MAIN
```

TASK-2 (ADD TWO HEX NUMBER 4bit)

```
.MODEL SMALL

.STACK 100h

.DATA
    hex1 DW ?
    hex2 DW ?

    Prompt1 DB 'Type a HEX Number, 0 - FFEF: $'    ; input message 1
    Prompt2 DB 'Type a HEX Number, 0 - FFFF: $'    ; input message 2
    Prompt3 DB 10, 13, 'The SUM is: $'             ; output message

    counter db 4 ; number of digits in a hexnumber

.CODE

MAIN PROC
    MOV AX, @DATA  ; loaded the data segment
    MOV DS, AX


    MOV AH,9
    LEA DX, Prompt1  ; first message print
    INT 21h

    CALL INHEX      ; called the inhex procedure
    MOV hex1,BX     ; hex1 a BX rakhtesi

    ;print Carraige return and new line
    MOV AH,2
    MOV DL,0DH
    INT 21H
    MOV AH,2
    MOV DL,0AH
    INT 21H


    MOV AH,9
    LEA DX, Prompt2   ; same as before
    INT 21h


    CALL INHEX
    MOV hex2,BX ; hex2 a BX rakhtesi


    MOV AH,9
    LEA DX, Prompt3
    INT 21h


    ADD BX,hex1

    CALL OUTHEX  ; showign result = hex1 + hex2


MAIN ENDP
```

INHEX:

```
060
061 INHEX PROC
062     XOR BX,BX  ; initialized zero
063     MOV CL,4
064
065
066     MOV AH,1
067     INT 21H ; take the first input digit
068
069 Loop:
070     CMP AL,0DH   ; comparing if it is CR or not
071     JE END1
072
073     CMP AL,'9' ; digit or alphabet?
074     JG Letter
075
076     AND AL,0FH ; getting the hexa decimal value of digit
077     JMP SHIFT
078
079 Letter:
080     SUB AL,37H ; getting the hexa decimal value of letter
081
082 SHIFT:
083     SHL BX,CL  ; shifting BX left by 4 bits
084     OR BL,AL   ; and putting the latest input in the most right section of BL
085
086     INT 21H    ; taking the next input
087
088     JMP Loop
089
090 END1:
091     RET
092
093 INHEX ENDP
094
095
```

OUTHEX:

```
098 OUTHEX PROC
099
100     MOV CL,4  ; 4 digits to show
101
102 PRINT:
103     MOV DL,BH   ; getting the BH(the righmost two digits) to store inside DL
104     SHR DL,CL   ; Then shifting right to display only one digit
105
106     CMP DL,9    ; comparing to see if its a digit or letter
107     JG  ALPHABET
108
109     ADD DL,30H  ; number or digit
110     JMP DIGIT
111
112 ALPHABET:
113     ADD DL,37h  ; letter A,B,C,D,E,F
114
115 DIGIT:
116     MOV AH,2
117     INT 21h
118
119     ROL BX,CL  ; rotating the ans , ans = AE24 ---> E24A ---> 24AE
120     DEC counter
121
122     CMP counter,0
123     JNE PRINT
124
125 RET
126
127 OUTHEX ENDP
128
129
130
131 END MAIN
```

ANOTHER TASK:

Here we are taking six inputs of Fahrenheit value and summing them up, after that converting that to celsius.

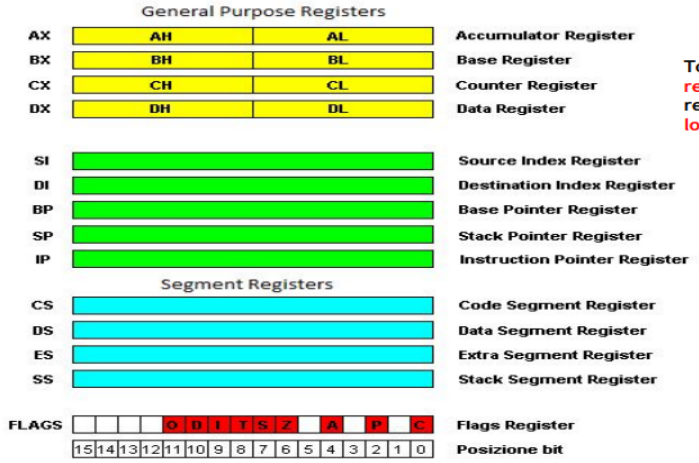
```
.MODEL SMALL
.STACK 100H
.DATA
    MSG1 DB 'ENTER 6 TEMPERATURE IN FARENHEIT (0-255): $'
    MSG2 DB 'TEMPERATURE SUMMATION IN CELSIUS: $'
    NUM1 DW ?
    NUM2 DW 0

.CODE

    MAIN PROC

        MOV AX,@DATA  ; loading data
        MOV DS,AX

        LEA DX,MSG1   ; showing the first message
        MOV AH,9
        INT 21H

        MOV CX,6  ; counter = 6

        ;sum of all inputs

        SUMMATION:
            MOV AH,2
            MOV DL,0DH
            INT 21H
            MOV DL,0AH
            INT 21H

            CALL INDEC  ; calling the input function
            ADD NUM1,AX ;STORE INPUT IN NUM1

            LOOP SUMMATION

        MOV AX,NUM1


        SUB AX,32       ; F to C calculation
        MOV BX,5
        MUL BX
        MOV BX,9 .
        DIV BX
        MOV NUM1, AX

        MOV AH,2
        MOV DL,0DH
        INT 21H         ; new line
        MOV DL,0AH
        INT 21H


        LEA DX,MSG2
        MOV AH,9        ; showing the second message
        INT 21H

        MOV AH,2
        MOV DL,0DH
        INT 21H          ; new line
        MOV DL,0AH
        INT 21H

        MOV AX,NUM1     ; calling the output function|
        CALL OUTDEC


        MOV AH,4CH
        INT 21H

    MAIN ENDP

    INCLUDE E:\Github Repo\University-3-2\CSE-4622-Microprocessor-lab\Lab04\INDEC.ASM
    INCLUDE E:\Github Repo\University-3-2\CSE-4622-Microprocessor-lab\Lab04\OUTDEC.ASM


END MAIN
```

----------MID END----------

Intel 8086 Internal Architecture: Registers

- Information inside microprocessor is stored in register
- Total Fourteen registers: All of these registers are 16-bit long
- Classified based on their functions they perform:
  1. Data Registers: hold data for operation
     - Four (4) general data registers
  2. Address Registers: hold address of data or instruction
     - Segment Register
     - Pointer Register
     - Index Register
  3. A Status Register: keep current status of the processor :FLAGS register
- Temporary register: for holding operands

Intel 8086 Internal Architecture: Registers



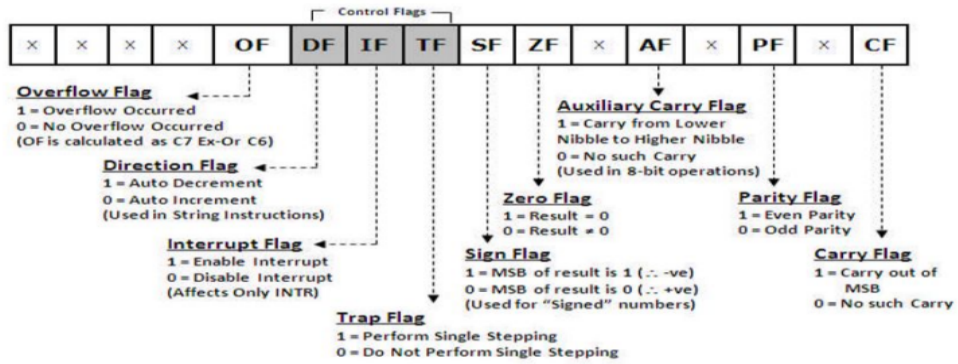
Total Fourteen registers: All of these registers are 16-bit long

Intel 8086 Internal Architecture: Segment Registers

Memory segment	Segment register	Offset register
Code segment	Code segment Register (CSR)	Instruction Pointer (IP)
Data segment	Data segment Register (DSR)	Source index (SI)/ Destination index (DI)
Stack segment	Stack segment Register (SSR)	Stack Pointer (SP)/ Base Pointer (BP)
Extra segment	Extra segment Register (ESR)	Destination Index(DI)

Flag Register

In 16 bit flag: 9 active flag



String Instructions

■ String instructions were designed to operate on large data structures.

■ The SI and DI registers are used as pointers to the data structures being accessed or manipulated.

■ The operation of the dedicated registers stated above are used to simplify code and minimize its size.

■ The registers(DI,SI) are automatically incremented or decremented depending on the value of the direction flag:

- □ DF=0, increment SI, DI.
- □ DF=1, decrement SI, DI.

■ To set or clear the direction flag one should use the following instructions:

- □ CLD to clear the DF.
- □ STD to set the DF.

String Instructions

■ movs (move string)

- □ Copy a string from one location to another

■ cmps (compare string)

- □ Compare the contents of two strings

■ scas (scan string)

- □ Search a string for one particular value

■ stos (store string)

- □ Store a value in some string position

■ lods (load string)

- □ Copies a value out of some string position

String Instructions

■ The REP/REPZ/REPE/REPNZ/REPNE prefixes are used to repeat the operation it precedes.

■ String instructions we will discuss:

- □ LODS
- □ STOS
- □ MOVS
- □ CMPS
- □ SCA

MOVSB/MOVSW

■ Transfers the contents of the memory byte, word or double word pointed to by SI relative to DS to the memory byte, or word pointed to by DI relative to ES. After the transfer is made, the DI register is automatically updated as follows:

- □ DI is incremented if DF=0.
- □ DI is decremented if DF=1

Examples:

- □ MOVSB ES:[DI]=DS:[SI]; DI=DI ± 1;SI=SI ± 1
- □ MOVSW ES:[DI]= DS:[SI]; DI=DI ± 2; SI=SI ± 2

MOVSB/MOVSW

Example
Assume:

Location	Content
Register SI	500H
Register DI	600H
Memory location 500H	'2'
Memory location 600H	'W'

After execution of MOVSB

If DF=0 then:

Location	Content
Register SI	501H
Register DI	601H
Memory location 500H	'2'
Memory location 600H	'2'

Else if DF=1 then:

Location	Content
Register SI	4FFH
Register DI	5FFH
Memory location 500H	'2'
Memory location 600H	'2'

Example
Assume:

Location	Content
Register DI	500H
Memory location 500H	'A'
Register AL	'2'

After execution of STOSB

If DF=0 then:

Location	Content
Register DI	501H
Memory location 500H	'2'
Register AL	'2'

Else if DF=1 then:

Location	Content
Register DI	4FFH
Memory location 500H	'2'
Register AL	'2'

STOSB/STOSW

STOSB/STOSW

■ Transfers the contents of the AL, AX or EAX registers to the memory byte, word or double word pointed to by DI relative to ES. After the transfer is made, the DI register is automatically updated as follows:

- □ DI is incremented if DF=0.
- □ DI is decremented if DF=1.

■ Examples:

- □ STOSB ES:[DI]=AL; DI=DI ± 1
- □ STOSW ES:[DI]=AX; DI=DI ± 2
- □ STOSD ES:[DI]=EAX; DI=DI ± 4

REP/REPZ/REPNZ

- Use REPNE and SCASB to search for the character ‘f’ in the buffer given below.
- BUFFER DB ‘EE3751’
- MOV AL,’f’
- LEA DI,BUFFER
- MOV ECX,6
- CLD
- REPNE SCASB
- JE FOUND

REP/REPZ/REPNZ

- Use REPNE and SCASB to search for the character ‘3’ in the buffer given below.
- BUFFER DB ‘EE3751’
- MOV AL,’f’
- LEA DI,BUFFER
- MOV ECX,6
- CLD
- REPNE SCASB
- JE FOUND

`rep` prefix

- Normally used with `movs` and with `stos`
- Causes this design to be executed:

```
while count in ECX > 0 loop
    perform primitive instruction;
    decrement ECX by 1;
end while;
```

SCASB/SCASW

- Compares the contents of the AL, AX or EAX register with the memory byte, or word pointed to by DI relative to ES and changes the flags accordingly. After the comparison is made, the DI register is automatically updated as follows:
  - DI is incremented if DF=0.
  - DI is decremented if DF=1.

REP/REPZ/REPNZ

- These prefixes cause the string instruction that follows them to be repeated the number of times in the count register ECX or until:
  - ZF=0 in the case of REPZ (repeat while equal).
  - ZF=1 in the case of REPNZ (repeat while not equal).

CMPSB/CMPSW

- Compares the contents of the the memory byte, word or double word pointed to by SI relative to DS to the memory byte, or word pointed to by DI relative to ES and changes the flags accordingly. After the comparison is made, the DI and SI registers are automatically updated as follows:
  - DI and SI are incremented if DF=0.
  - DI and SI are decremented if DF=1.

LODSB/LODSW

- Loads the AL, AX registers with the content of the memory byte, word pointed to by SI relative to DS. After the transfer is made, the SI register is automatically updated as follows:
  - SI is incremented if DF=0.
  - SI is decremented if DF=1.

LODSB/LODSW

- **Examples:**
  - LODSB
    AL=DS:[SI]; SI=SI ± 1
  - LODSW
    AX=DS:[SI]; SI=SI ± 2

LODSB/LODSW

Example

Assume:

Location	Content
Register SI	500H
Memory location 500H	‘A’
Register AL	‘2’

After execution of LODSB

If DF=0 then:

Location	Content
Register SI	501H
Memory location 500H	‘A’
Register AL	‘A’

Else if DF=1 then:

Location	Content
Register SI	4FFH
Memory location 500H	‘A’
Register AL	‘A’

scas

- Used to scan a string for the presence or absence of a particular string element
  - String which is examined is a destination string – the address of the element being examined is in the destination index register EDI
  - Accumulator contains the element being scanned for

Additional Repeat Prefixes

- `repe` (equivalent mnemonic `repz`)
  - "repeat while equal" ("repeat while zero")
- `repne` (same as `repnz`)
  - "repeat while not equal" ("repeat while not zero")
- Each appropriate for use with `cmps` and `scas` which affect the zero flag ZF

stos

- Copies a byte, a word, a doubleword or a quadword from the accumulator to an element of a destination string
- Affects no flag, so only the `rep` prefix is appropriate for use with it
  - When repeated, it copies the same value into consecutive positions of a string

repe and repne Operation

- Each works the same as `rep`, iterating a primitive instruction while ECX is not zero
- Each also examines ZF after the string instruction is executed
  - `repe` and `repz` continue iterating while ZF=1, as it would be following a comparison where two operands were equal
  - `repne` and `repnz` continue iterating while ZF=0

lods

- Copies a source string element to the accumulator
- No repeat prefix is useful with `lods`
- `lods` and `stos` are often used together in a loop
  - `lods` at the beginning of a loop to fetch an element
  - `stos` at the end after the element is manipulated

cmps

- Subtracts two string elements and sets flags based on the difference
- If used in a loop, it is appropriate to follow `cmps` by a conditional jump instruction
- `repe` and `repne` prefixes often used with cmps instructions

CHAPTER-11

Overview

In this chapter we consider a special group of instructions called the *string instructions*. In 8086 assembly language, a **memory string** or **string** is simply a byte or word array. Thus, string instructions are designed for array processing.

Here are examples of operations that can be performed with the string instructions:

- Copy a string into another string.
- Search a string for a particular byte or word.
- Store characters in a string.
- Compare strings of characters alphabetically.

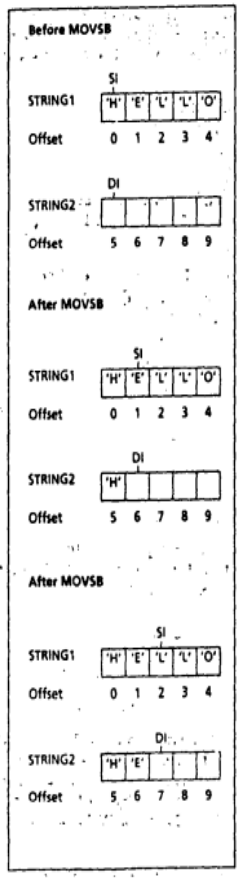
The tasks carried out by the string instructions can be performed by using the register indirect addressing mode we studied in Chapter 10; however, the string instructions have some built-in advantages. For example, they provide automatic updating of pointer registers and allow memory–memory operations.

11.1 The Direction Flag

In Chapter 5, we saw that the FLAGS register contains six status flags and three control flags. We know that the status flags reflect the result of an operation that the processor has done. The control flags are used to control the processor's operations.

One of the control flags is the *direction flag (DF)*. Its purpose is to determine the direction in which string operations will proceed. These operations are implemented by the two index registers SI and DI. Suppose, for example, that the following string has been declared:

Figure 11.1 MOVSB



MOVSB is the first instruction we have seen that permits a memory-memory operation. It is also the first instruction that involves the ES register.

The REP Prefix

MOVSB moves only a single byte from the source string to the destination string. To move the entire string, first initialize CX to the number *N* of bytes in the source string and execute

```
REP MOVSB
```

```
STRING1   DB              'ABCDE'
```

And this string is stored in memory starting at offset 0200h:

Offset address	*Content*	*ASCII character*
0200h	041h	A
0201h	042h	B
0202h	043h	C
0203h	044h	D
0204h	045h	E

If DF = 0, SI and DI proceed in the direction of increasing memory addresses: from left to right across the string. Conversely, if DF = 1, SI and DI proceed in the direction of decreasing memory addresses: from right to left.

In the DEBUG display, DF = 0 is symbolized by UP, and DF = 1 by DN.

CLD and STD

To make DF = 0, use the **CLD** instruction

```
CLD    ;clear directio flag
```

To make DF = 1, use the **STD** instruction:

```
STD    ;set direction flag
```

CLD and STD have no effect on the other flags.

11.2 Moving a String

Suppose we have defined two strings as follows:

```
.DATA
STRING1  DB     'HELLO'
STRING2  DB     5 DUP (?)
```

and we would like to move the contents of STRING1 (the source string) into STRING2 (the destination string). This operation is needed for many string operations, such as duplicating a string or concatenating strings (attaching one string to the end of another string).

The MOVSB instruction

```
MOVSB              ;move string byte
```

copies the contents of the byte addressed by DS:SI, to the byte addressed by ES:DI. The contents of the source byte are unchanged. After the byte has been moved, both SI and DI are automatically incremented if DF = 0, or decremented if DF = 1. For example, to move the first two bytes of STRING1 to STRING2, we execute the following instructions:

```
MOV AX,@DATA
MOV DS,AX          ;initialize DS
MOV ES,AX          ;and ES
LEA SI,STRING1     ;SI points to source string
LEA DI,STRING2     ;DI points to destination string
CLD                ;clear DF
MOVSB              ;move first byte
MOVSB              ;and second byte
```

See Figure 11.1.

```
        remove previous char from string
    ELSE
        store char in string
        chars_read = chars_read + 1
    END_IF
    read a char
END_WHILE
```

```
STD                         ;right to left processing
LEA SI,ARR+8h               ;SI pts to 60
LEA DI,ARR+Ah               ;DI pts to ?
MOV CX,3                    ;3 elts to move
REP MOVSW                   ;move 40,50,60
MOV WORD PTR [DI],30        ;insert 30
```

Note: the PTR operator was introduced in section 10.2.3.

11.3 *Store String*

The STOSB Instruction

```
STOSB       ;store string byte
```

moves the contents of the AL register to the byte addressed by ES:DI. DI is incremented if DF = 0 or decremented if DF = 1. Similarly, the **STOSW** instruction

```
STOSW       ;store string word
```

moves the contents of AX to the word at address ES:DI and updates DI by 2, according to the direction flag setting.

STOSB and STOSW have no effect on the flags.

As an example of STOSB, the following instructions will store two "A"s in STRING1:

```
MOV AX,@DATA
MOV ES,AX           ;initialize ES
LEA DI,STRING1      ;DI points to STRING1
CLD                 ;process to the right
MOV AL,'A'          ;AL has character to store
STOSB               ;store an 'A'
STOSB               ;store another one
```

See Figure 11.2.

Reading and Storing a Character String

INT 21h, function 1 reads a character from the keyboard into AL. By repeatedly executing this interrupt with STOSB, we can read and store a character string. In addition, the characters may be processed before storing them.

The following procedure READ_STR reads and stores characters in a string, until a carriage return is typed. The procedure is entered with the string offset address in DI. It returns the string offset in DI, and number of characters entered in BX. If the user makes a typing mistake and hits the backspace key, the previous character is removed from the string.

This procedure is similar to DOS INT 21h, function 0Ah (see exercise 11.11).

Algorithm for READ_STR

```
chars_read = 0
read a char
WHILE char is not a carriage return DO
 IF char is a backspace
  THEN
   chars_read = chars_read - 1
```

The **REP** prefix causes MOVSB to be executed *N* times. After each MOVSB, CX is decremented until it becomes 0. For example, to copy STRING1 of the preceding section into STRING2, we execute

```
CLD
LEA SI,STRING1
LEA DI,STRING2
MOV CX,5                ;no. of chars in STRING1
REP MOVSB
```

Example 11.1 Write instructions to copy STRING1 of the preceding section into STRING2 in reverse order.

Solution: The idea is to get SI pointing to the end of STRING1, DI to the beginning of STRING2, then move characters as SI travels to the left across STRING1.

```
      LEA  SI,STRING1+4     ;SI pts to end of STRING1
      LEA  DI,STRING2       ;DI pts to beginning of STRING
      STD                   ;right to left processing
      MOV  CX,5
MOVE:
      MOVSB                 ;move a byte
      ADD  DI,2
      LOOP MOVE
```

Here it is necessary to add 2 to DI after each MOVSB. Because we do this when DF = 1, MOVSB automatically decrements both SI and DI, and we want to increment DI.

MOVSW

There is a word form of MOVSB. It is

```
MOVSW    ;move string word
```

MOVSW moves a word from the source string to the destination string. Like MOVSB, it expects DS:SI to point to a source string word, and ES:DI to point to a destination string word. After a string word has been moved, both SI and DI are increased by 2 if DF = 0, or are decreased by 2 if DF = 1.

MOVSB and MOVSW have no effect on the flags.

Example 11.2 For the following array,

```
ARR DW   10,20,40,50,60,?
```

write instructions to insert 30 between 20 and 40. (Assume DS and ES have been initialized to the data segment.)

Solution: The idea is to move 40, 50, and 60 forward one position in the array, then insert 30.

11.4 Load String

The **LODSB** instruction

```
LODSB                   ;load string byte
```

moves the byte addressed by DS:SI into AL. SI is then incremented if DF = 0 or decremented if DF = 1. The word form is

```
LODSW                   ;load string word
```

it moves the word addressed by DS:SI into AX; SI is increased by 2 if DF = 0 or decreased by 2 if DF = 1.

LODSB can be used to examine the characters of a string, as shown later.

LODSB and LODSW have no effect on the flags.

To illustrate LODSB, suppose STRING1 is defined as

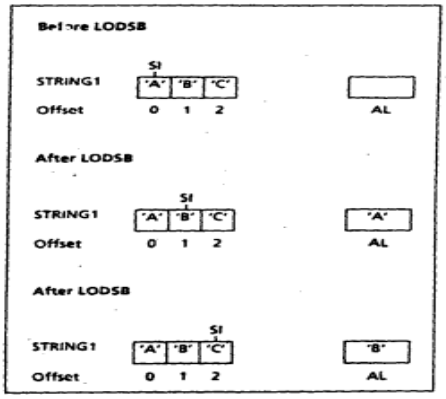
```
STRING1  DB     'ABC'
```

The following code successively loads the first and second bytes of STRING1 into AL

```
MOV AX,@DATA
MOV DS,AX               ;initialize DS
LEA SI,STRING1          ;SI points to STRING1
CLD                     ;process left to right
LODSB                   ;load first byte into AL
LODSB                   ;load second byte into AL
```

See Figure 11.3.

Figure 11.3 LODSB



Displaying a Character String

The following procedure DISP_STR displays the string pointed to by SI, with the number of characters in BX. It can be used to display all or part of a string.

Algorithm for DISP_STR

```
FOR count times DO  /* count = no. of characters to display *
  load a string character into AL
  move it to DL
  output character
END_FOR
```

Program Listing PGM 11_2.ASM

```
;displays a string
;input:  SI = offset of string
;        BX =  no. of chars. to display
;output: none
        PUSH  AX
        PUSH  BX
        PUSH  CX
        PUSH  DX
        PUSH  SI
        MOV   CX,BX         ;no. of chars
       JCXZ   P_EXIT        ;exit if none
       CLD                  ;process left to right
```



```
        MOV    AH,2         ;prepare to print
TOP:
        LODSB               ;char in AL
        MOV    DL,AL        ;move it to DL
        INT    21H          ;print char
        LOOP   TOP          ;loop until done
P_EXIT:
        POP    SI
        POP    DX
        POP    CX
        POP    BX
        POP    AX
        RET
DISP_STR      ENDP
```

11.5 Scan String

The instruction

```
SCASB     ;scan string byte
```

can be used to examine a string for a target byte. The target byte is contained in AL. SCASB subtracts the string byte pointed to by ES:DI from the contents of AL and uses the result to set the flags. The result is not stored. Afterward, DI is incremented if DF = 0 or decremented if DF = 1.

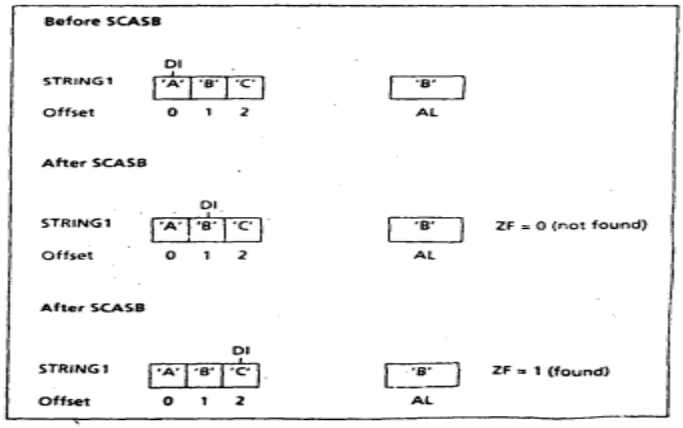
The word form is

```
SCASW     ;scan string word
```

in this case, the target word is in AX. SCASW subtracts the word addressed by ES:DI from AX and sets the flags. DI is increased by 2 if DF = 0 or decreased by 2 if DF = 1.

All the status flags are affected by SCASB and SCASW.

Figure 11.4 SCASB



For example, if the string

```
STRING1   DB     'ABC'
```

is defined, then these instructions examine the first two bytes of STRING1, looking for "B"

```
MOV AX,@DATA
MOV AX,ES              ;initialize ES
CLD                    ;left to right processing
LEA DI,STRING1         ;DI pts to STRING1
MOV AL,'B'             ;target character
SCASB                  ;scan first byte
SCASB                  ;scan second byte
```

See Figure 11.4. Note that when the target "B" was found, ZF = 1 and because SCASB automatically increments DI, DI points to the byte after the target, not the target itself.

In looking for a target byte in a string, the string is traversed until the byte is found or the string ends. If CX is initialized to the number of bytes in the string,

```
REPNE SCASB            ;repeat SCASB while not equal
                       (to target)
```

will repeatedly subtract each string byte from AL, update DI, and decrement CX until there is a zero result (the target is found) or CX = 0 (the string ends). *Note:* **REPNZ** (repeat while not zero) generates the same machine code as **REPNE**.

As an example, let's write a program to count the number of vowels and consonants in a string.

Algorithm for Counting Vowels and Consonants

```
Initialize vowel_count and consonant_count to 0;
Read and store a string
REPEAT
  Load a string character;
  IF it's a vowel
   THEN
    increment vowel_count
   ELSE IF it's a consonant
    THEN increment consonant_count
  END_IF
UNTIL end of string
display no. of vowels
display no. of consonants
```

We'll use procedure READ_STR (section 11.3) to read the string. It returns with DI pointing to the string and BX containing the number of characters read. To display the number of vowels and consonants in the string, we'll use procedure OUTDEC of Chapter 9. It displays the contents of AX as a signed decimal integer. For simplicity, we'll suppose the input is in upper case.

11.7 General Form of the String Instructions

Let us summarize the byte and word forms of the string instructions:

Instruction	*Destination*	*Source*	*Byte form*	*Word form*
Move string	ES:DI	DS:SI	MOVSB	MOVSW
Compare string	ES:DI	DS:SI	CMPSB	CMPSW
Store string	ES:DI	AL or AX	STOSB	STOSW
Load string	AL or AX	DS:SI	LODSB	LODSW
Scan string	ES:DI	AL or AX	SCASB	SCASW

* Result not stored.

The operands of these instructions are implicit; that is, they are not part of the instructions themselves. However, there are forms of the string instructions in which the operands appear explicitly. They are as follows:

Instruction		*Example*
MOVS	destination_string, source_string	MOVSB
CMPS	destination_string, source_string	CMPSB
STOS	destination_string	STOS STRING2
LODS	source_string	LODS STRING1
SCAS	destination_string	SCAS STRING2

When the assembler encounters one of these general forms, it checks to see if (1) the source string is in the segment addressed by DS and the destination string is in the segment addressed by ES, and (2) in the case of MOVS and CMPS, if the strings are of the same type; that is, both byte strings or word strings. If so, then the instruction is coded as either a byte form, such as MOVSB, or a word form, such as MOVSW, to match the data declaration of the string. For example, suppose that DS and ES address the following segment:

```
.DATA
STRING1  DB        'ABCDE'
STRING2  DB        'EFGH'
STRING3  DB        'IJKL'
STRING4  DB        'MNOP'
STRING5  DW        1,2,3,4,5
STRING6  DW        7,8,9
```

Then the following pairs of instructions are equivalent

```
MOVS STRING2,STRING1          MOVSB
MOVS STRING6,STRING5          MOVSW
LODS STRING4                  LODSB
LODS STRING5                  LODSW
SCAS STRING1                  SCASB
STOS STRING6                  STOSW
```

It is important to note that if the general forms are used, it is still necessary to make DS:SI and ES:DI point to the source and destination strings, respectively.

There are advantages and disadvantages in using the general forms of the string instructions. An advantage is that because the operands appear as part of the code, program documentation is improved. A disadvantage is that only by checking the data definitions is it possible to tell whether a general string instruction is a byte form or a word form. In fact, the operands specified in a general string instruction may not be the actual operands used when the instruction is executed! For example, consider the following code:

```
LEA SI,STRING1          ;SI PTS TO STRING1
LEA DI,STRING2          ;DI PTS TO STRING2
MOVS STRING4,STRING3
```

Even though the specified source and destination operands are STRING3 and STRING4, respectively, when MOVS is executed the first byte of STRING1 is moved to the first byte of STRING2. This is because the assembler translates MOVS STRING4, STRING3 into the machine code for MOVSB, and SI and DI are pointing to the first bytes of STRING1 and STRING2, respectively.

11.6 Compare String

The CMPSB Instruction

```
CMPSB                   ;compare string byte
```

subtracts the byte with address ES:DI from the byte with address DS:SI, and sets the flags. The result is not stored. Afterward, both SI and DI are incremented if DF = 0, or decremented if DF = 1.

The word version of CMPSB is

```
CMPSW                   ;compare string word
```

It subtracts the word with address ES:DI from the word whose address is DS:SI, and sets the flags. If DF = 0, SI and DI are increased by 2; if DF = 1, they are decreased by 2. CMPSW is useful in comparing word arrays of numbers.

All the status flags are affected by CMPSB and CMPSW.

For example, suppose

```
.DATA
STRING1 DB      'ACD'
STRING2 DB      'ABC'
```

The following instructions compare the first two bytes of the preceding strings:

```
MOV AX,@DATA            ;initialize DS
MOV DS,AX               ;and ES
MOV ES,AX
CLD                     ;left to right processing
LEA SI,STRING1          ;SI pts to STRING1
```

REPE and REPZ

String comparison may be done by attaching the prefix **REPE** (repeat while equal) or **REPZ** (repeat while zero) to CMPSB or CMPSW. CX is initialized to the number of bytes in the shorter string, then

```
REPE CMPSB              ;compare string bytes while equal
```

or

```
REPE CMPSW              ;compare string words while equal
```

CHAPTER-12

Overview

One of the most interesting and useful applications of assembly language is in controlling the monitor display. In this chapter, we program such operations as moving the cursor, scrolling windows on the screen, and displaying characters with various attributes. We also show how to program the keyboard, so that if the user presses a key, a screen control function is performed; for example, we'll show how to make the arrow keys operate.

The display on the screen is determined by data stored in memory. The chapter begins with a discussion of how the display is generated and how it can be controlled by altering the display memory directly. Next, we'll show how to do screen operations by using BIOS function calls. These functions can also be used to detect keys being pressed; as a demonstration, we'll write a simple screen editor.

CHAPTER-13

Overview

In previous chapters we have shown how programming may be simplified by using procedures. In this chapter, we discuss a program structure called a *macro*, which is similar to a procedure.

As with procedures, a macro name represents a group of instructions. Whenever the instructions are needed in the program, the name is used. However, the way procedures and macros operate is different. A procedure is called at execution time; control transfers to the procedure and returns after executing its statements. A macro is invoked at assembly time. The assembler copies the macro's statements into the program at the position of the invocation. When the program executes, there is no transfer of control.

Macros are especially useful for carrying out tasks that occur frequently. For example, we can write macros to initialize the DS and ES registers, print a character string, terminate a program, and so on. We can also write macros to eliminate restrictions in existing instructions; for example, the operand of MUL can't be a constant, but we can write a multiplication macro that doesn't have this restriction.

13.1 Macro Definition and Invocation

A **macro** is a block of text that has been given a name. When MASM encounters the name during assembly, it inserts the block into the program. The text may consist of instructions, pseudo-ops, comments, or references to other macros.

The syntax of macro definition is

```
macro_name      MACRO    d1,d2,...dn
                statements
                ENDM
```

Here macro_name is the user-supplied name for the macro. The pseudo-ops MACRO and ENDM indicate the beginning and end of the macro definition; d1, d2,...dn is an optional list of dummy arguments used by the macro.

One use of macros is to create new instructions. For example, we know that the operands of MOV can't both be word variables, but we can get around this restriction by defining a macro to move a word into a word.

Example 13.1 Define a macro to move a word into a word.

Solution:

```
MOVW    MACRO   WORD1,WORD2
        PUSH    WORD2
        POP     WORD1
        ENDM
```

Here the name of the macro is MOVW. WORD1 and WORD2 are the dummy arguments.

To use a macro in a program, we **invoke** it. The syntax is

```
macro_name   a1,a2, . . . an
```

where a1, a2,... an is a list of actual arguments. When MASM encounters the macro name, it **expands** the macro; that is, it copies the macro statements into the program at the position of the invocation, just as if the user had typed them in. As it copies the statements, MASM replaces each dummy argument di by the corresponding actual argument ai and creates the machine code for any instructions.

A macro definition must come before its invocation in a program listing. To ensure this sequence, macro definitions are usually placed at the beginning of a program. It is also possible to create a library of macros to be used by any program, and we do this later in the chapter.

CHAPTER-14

Overview

Until now, all our programs have consisted of a code segment, a stack segment, and perhaps a data segment. If there were other procedures besides the main procedure, they were placed in the code segment after the main procedure. In this chapter, you will see that programs can be constructed in other ways.

In section 14.1, we discuss the .COM program format in which code, data, and stack fit into a single segment. .COM programs have a simple structure and don't take up as much disk space as .EXE programs, so system programs are often written in this format.

Section 14.2 shows how procedures can be placed in different modules, assembled separately, and linked into a single program. In this way they can be written and tested separately. The modules containing these procedures may have their own code and data segments; when the modules are linked, the code segments can be combined, as can the data segments.

Section 14.3 covers the full segment definitions. They provide complete control over the ordering, combination, and placement of program segments.

Section 14.4 provides more information about the simplified segment definitions that we have been using throughout the book.

The procedures we've written so far have generally passed data values through registers. Section 14.5 shows other ways for procedures to communicate.

CHAPTER-15

Overview

In previous chapters, we used the INT (interrupt) instruction to call system routines. In this chapter, we discuss different kinds of interrupts and take a closer look at the operation of the INT instruction. In sections 15.2 and 15.3, we discuss the services provided by various BIOS (basic input/output systems) and DOS interrupt routines.

To demonstrate the use of interrupts, we will write a program that displays the current time on the screen. There are three versions: the first version simply displays the time and then terminates, the second version shows the time updated every second, and the third version is a memory resident program that can be called up when other programs are running.

TRI-STATE LOGIC

•In digital electronics three-state, tri-state, or 3-state logic

•allows an output port to assume a high impedance state in addition to the 0 and 1 logic levels,

•effectively removing the output from the circuit.

•To understand the concept and need for tri-state devices one must understand the concept of a 'bus'.

•Bus is typically a set of parallel connections on which several devices are connected together.

•Imagine a bus where many devices are connected in parallel.

•If one of the device's outputs is connected to the bus and the rest have their inputs on it, then there is no problem.

•But what happens when multiple devices have their outputs connected to it?

•For example two or more logic gates' output is connected together. One of the devices might output a logic '1' and simultaneously the other might output a '0'. Leading to short circuit currents and pulling the potential of the line to some indeterminate state.

Tri-state Logic

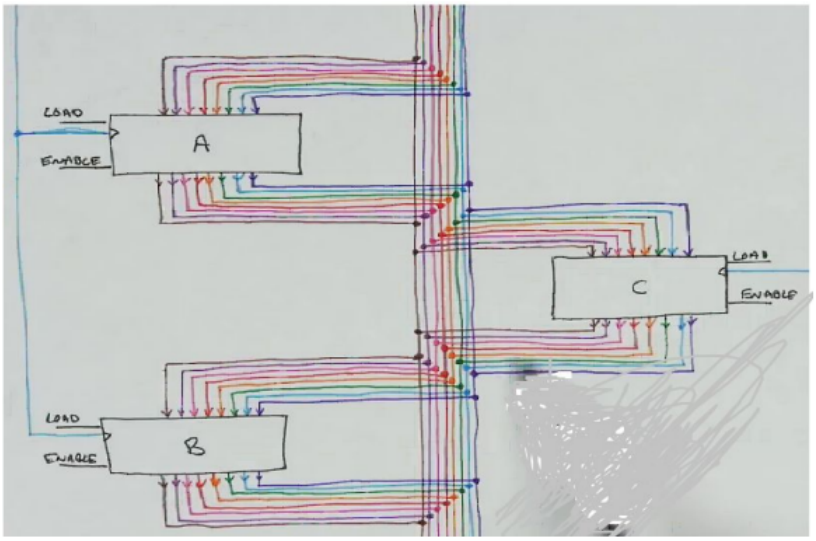
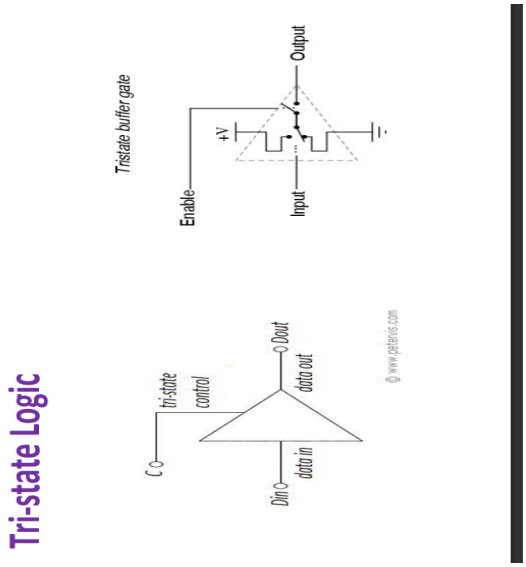
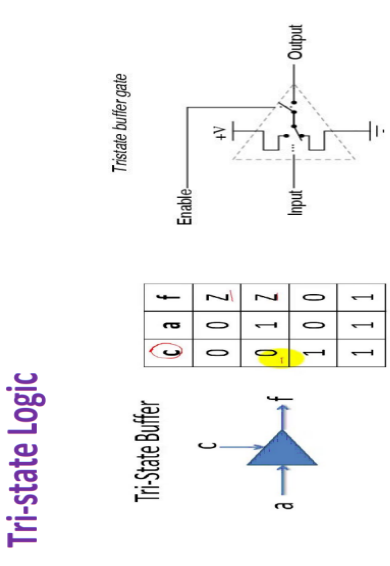


Figure: Typical Bus System, on which several devices are connected together.

•Under such a scenario how do we connect multiple devices (eg. multiple RAMs etc) irrespective of the inputs, the output will either be in '1' or '0'.
•That is where the tri-state comes into picture.

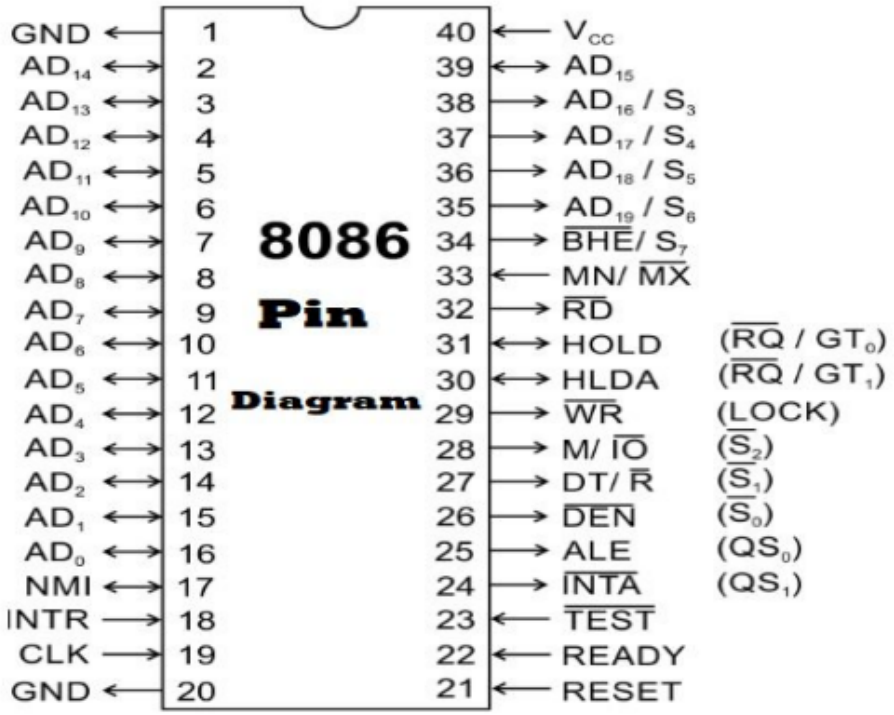
•All the devices will have tri-state drives at their output.
•Now when enabled, the outputs can either be '1' or '0' but when disabled they go into a third state •i.e. tri-state, where they can neither source or sink.
•Such outputs can be connected in parallel and they will not affect the existing condition of the bus.





8086 BUS TIMING DIAGRAM

8086 Pin Diagram

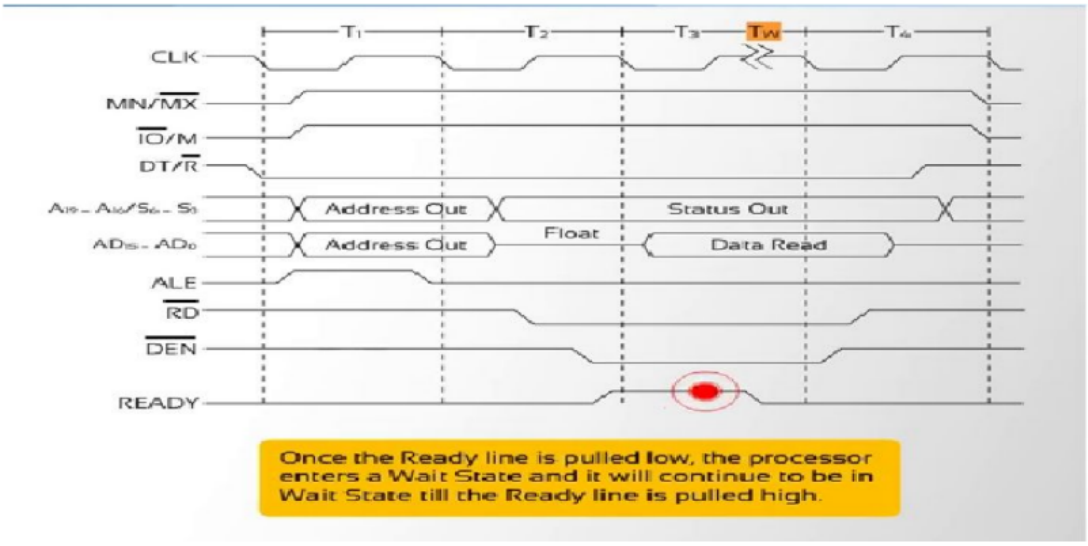


8086 Status Signals

Status Inputs			CPU Cycles	8288 Command
$\overline{S}_2$	$\overline{S}_1$	$\overline{S}_0$		
0	0	0	Interrupt Acknowledge	$\overline{INTA}$
0	0	1	Read I/O Port	$\overline{IORC}$
0	1	0	Write I/O Port	$\overline{IOWC}$, $\overline{AIOWC}$
0	1	1	Halt	None
1	0	0	Instruction Fetch	$\overline{MRDC}$
1	0	1	Read Memory	$\overline{MRDC}$
1	1	0	Write Memory	$\overline{MWTC}$, $\overline{AMWC}$
1	1	1	Passive	None

Bus Status Codes

8086 Bus Timing Diagram



Use of the Ready line:

Ready line is pulled high when the external peripheral device is ready.

The processor reads information of the data bus.

Enables the signal to read data from the memory device at that point of time.

8086: Mode of Operation

Minimum Mode

- Single Processor mode: No additional Co-processor can be connected.
- 8086 responsible for generating all control signals for Memory and I/O.
- Used to design systems used in simple applications.

Versus

Maximum Mode

- Multiprocessor mode: Additional Co-processors can be connected.
- External Bus controller is responsible for generating all control signals for Memory and I/O.
- Used for complex and large applications.

8086 Microprocessor Interrupts

Introduction

- The meaning of interrupts is to break the sequence of operation.
- While the Microprocessor is executing a program, an 'interrupt' breaks the normal sequence of execution of instructions, and diverts its execution to some other program called Interrupt Service Routine (ISR).
    - After executing , control returns the back again to the main program

Interrupt

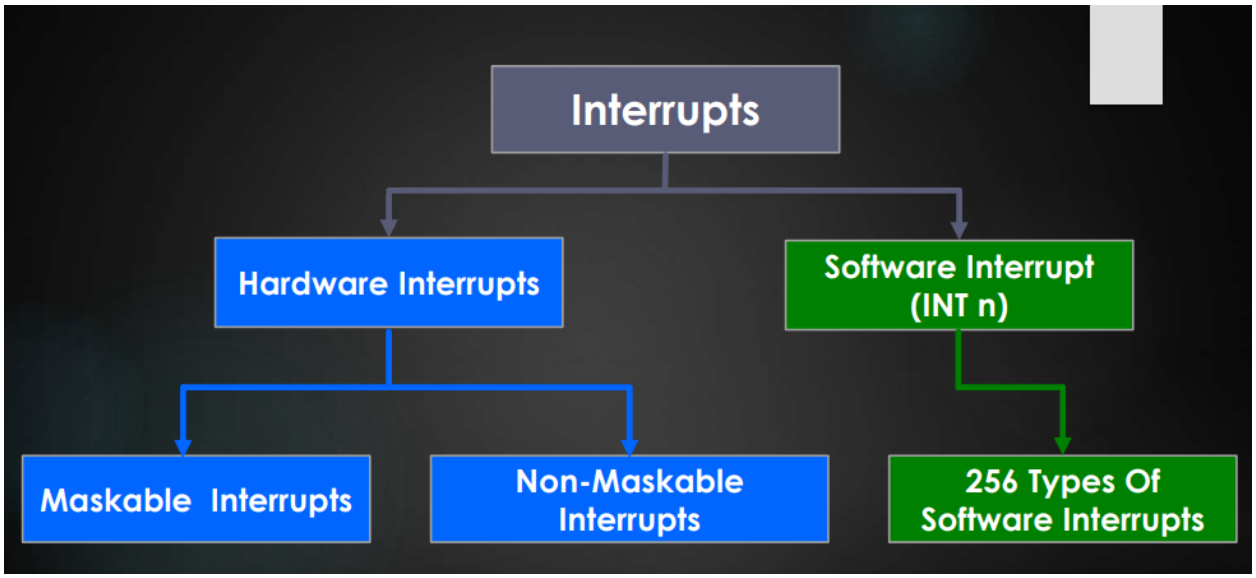
- Keep moving until interrupted by the sensor .
- Interrupt received then does pre-defined operation.
- After finishing the interrupt service return to normal operation i.e keep moving forward again

The processor can be interrupted in the following ways

i) by an external signal generated by a peripheral,

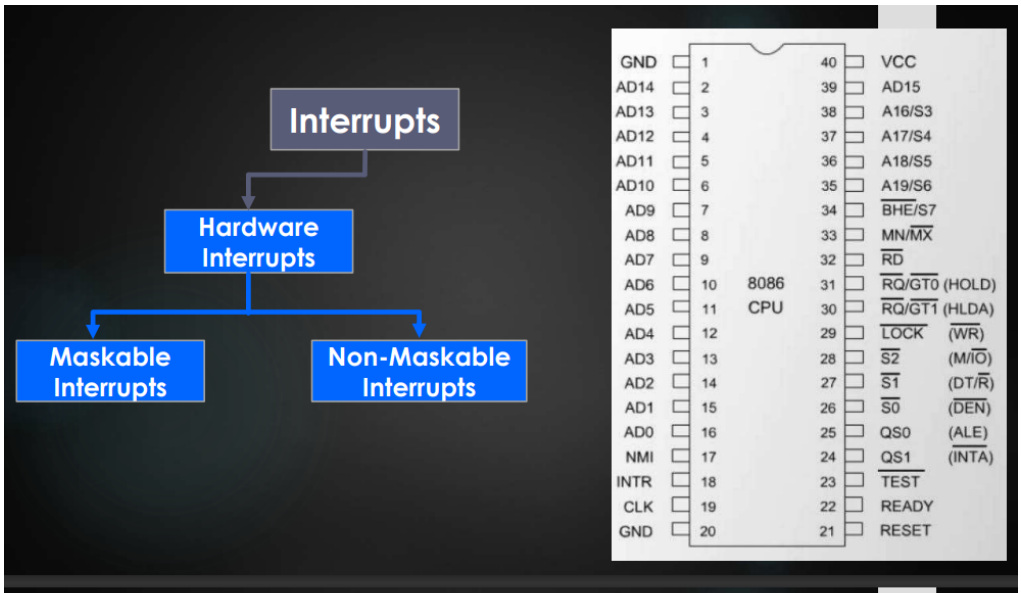
ii) by an internal signal generated by a special instruction in the program,

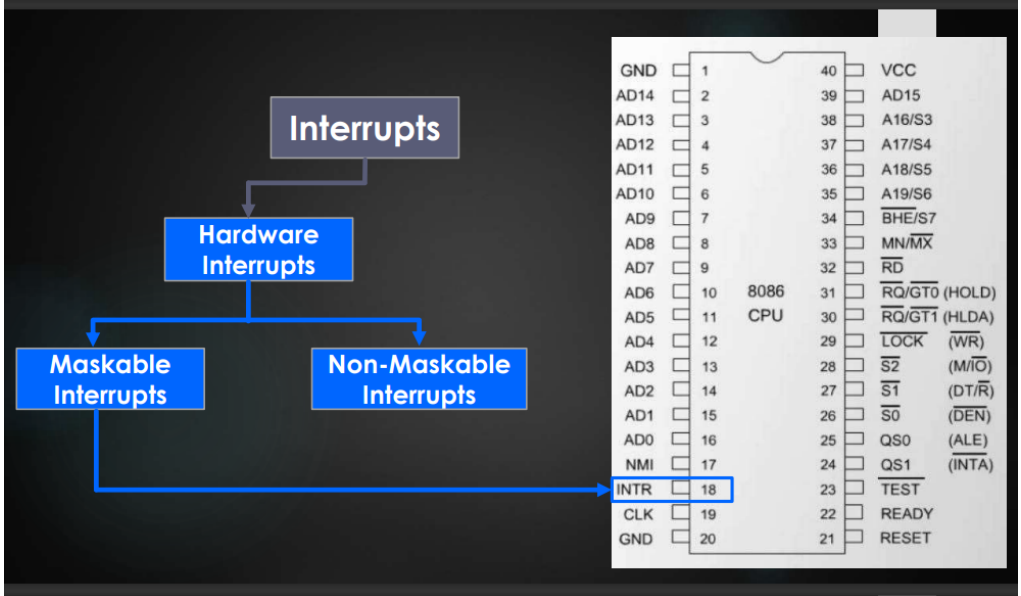
iii) by an internal signal generated due to an exceptional condition which occurs while executing an instruction.

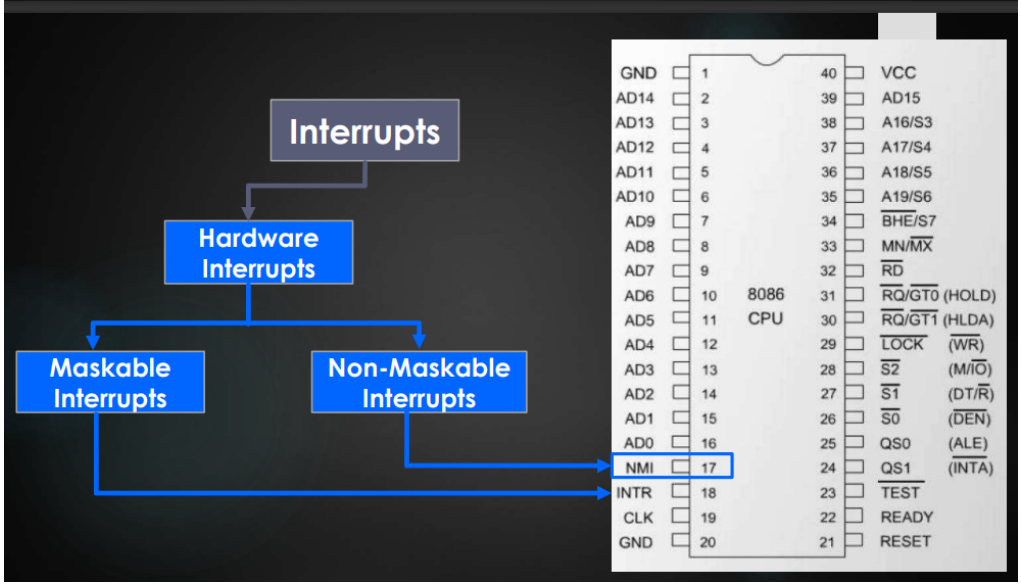


Hardware Interrupts

The interrupts initiated by external hardware by sending an appropriate signal to the interrupt pin of the processor is called hardware interrupt. The 8086 processor has two interrupt pins INTR and NMI. The interrupts initiated by applying appropriate signal to these pins are called hardware interrupts of 8086

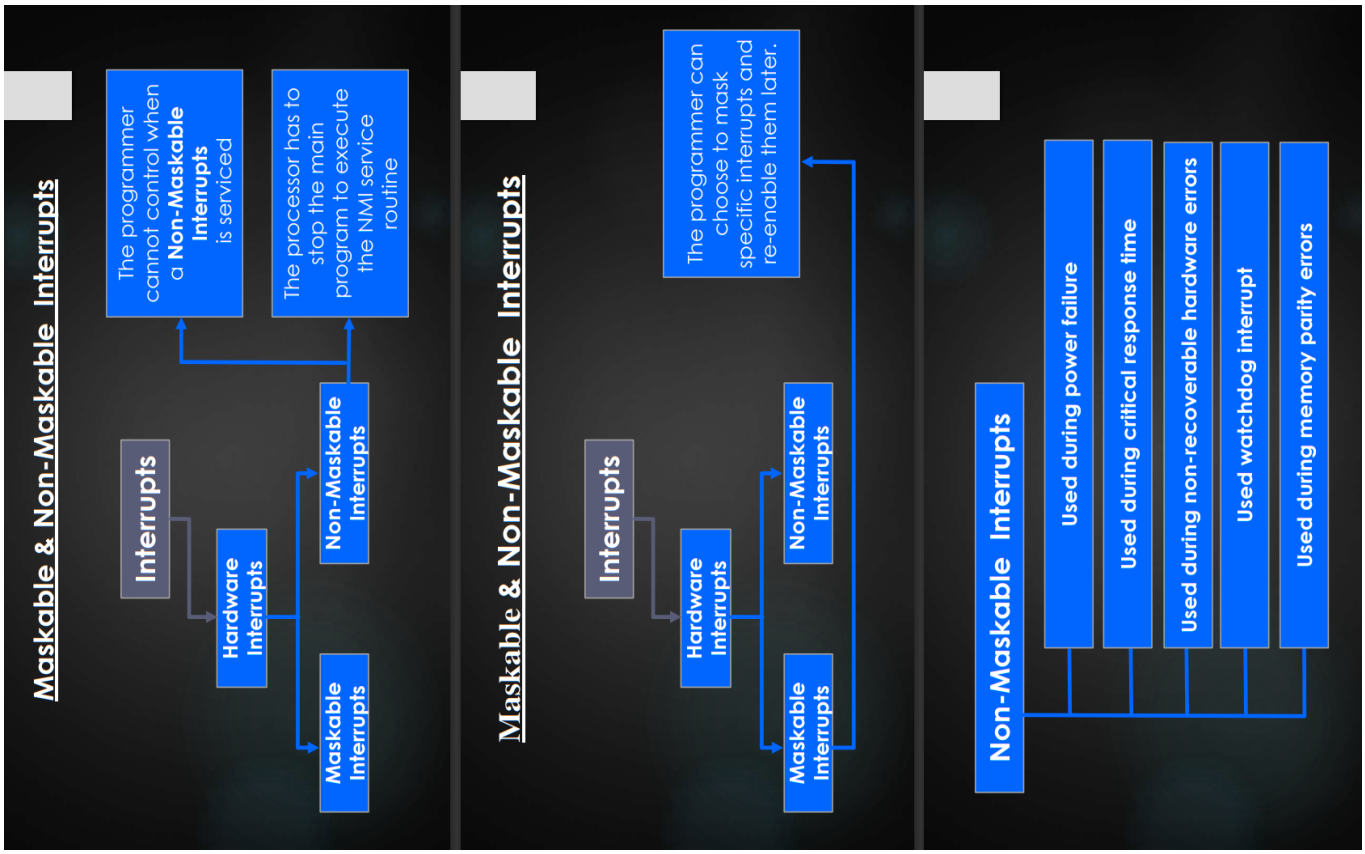

Interrupts
Hardware Interrupts
Maskable Interrupts
Non-Maskable Interrupts
GND 1
AD14 2
AD13 3
AD12 4
AD11 5
AD10 6
AD9 7
AD8 8
AD7 9
AD6 10
AD5 11
AD4 12
AD3 13
AD2 14
AD1 15
AD0 16
NMI 17
INTR 18
CLK 19
GND 20
8086 CPU
40 VCC
39 AD15
38 A16/S3
37 A17/S4
36 A18/S5
35 A19/S6
34 BHE/S7
33 MN/MX
32 RD
31 RQ/GT0 (HOLD)
30 RQ/GT1 (HLDA)
29 LOCK (WR)
28 S2 (M/IO)
27 S1 (DT/R)
26 S0 (DEN)
25 QS0 (ALE)
24 QS1 (INTA)
23 TEST
22 READY
21 RESET


Interrupts
Hardware Interrupts
Maskable Interrupts
Non-Maskable Interrupts
GND 1
AD14 2
AD13 3
AD12 4
AD11 5
AD10 6
AD9 7
AD8 8
AD7 9
AD6 10
AD5 11
AD4 12
AD3 13
AD2 14
AD1 15
AD0 16
NMI 17
INTR 18
CLK 19
GND 20
8086 CPU
40 VCC
39 AD15
38 A16/S3
37 A17/S4
36 A18/S5
35 A19/S6
34 BHE/S7
33 MN/MX
32 RD
31 RQ/GT0 (HOLD)
30 RQ/GT1 (HLDA)
29 LOCK (WR)
28 S2 (M/IO)
27 S1 (DT/R)
26 S0 (DEN)
25 QS0 (ALE)
24 QS1 (INTA)
23 TEST
22 READY
21 RESET


Interrupts
Hardware Interrupts
Maskable Interrupts
Non-Maskable Interrupts
GND 1
AD14 2
AD13 3
AD12 4
AD11 5
AD10 6
AD9 7
AD8 8
AD7 9
AD6 10
AD5 11
AD4 12
AD3 13
AD2 14
AD1 15
AD0 16
NMI 17
INTR 18
CLK 19
GND 20
8086 CPU
40 VCC
39 AD15
38 A16/S3
37 A17/S4
36 A18/S5
35 A19/S6
34 BHE/S7
33 MN/MX
32 RD
31 RQ/GT0 (HOLD)
30 RQ/GT1 (HLDA)
29 LOCK (WR)
28 S2 (M/IO)
27 S1 (DT/R)
26 S0 (DEN)
25 QS0 (ALE)
24 QS1 (INTA)
23 TEST
22 READY
21 RESET

Maskable & Non-Maskable Interrupts

The processor has the facility for accepting or rejecting hardware interrupts. Programming the processor to reject an interrupt is referred to as masking or disabling and programming the processor to accept an interrupt is referred to as unmasking or enabling. In 8086 the interrupt flag (IF) can be set to one to unmask or enable all hardware interrupts and IF is cleared to zero to mask or disable hardware interrupts except NMI. The interrupts whose request can be either accepted or rejected by the processor are called maskable interrupts. Maskable & Non-Maskable Interrupts The interrupts whose request has to be definitely accepted (or cannot be rejected) by the processor are called non-maskable interrupts. Whenever a request is made by non-maskable interrupt, the processor has to definitely accept that request and service that interrupt by suspending its current program and executing an ISR. In the 8086 processor all the hardware interrupts initiated through the INTR pin are maskable by clearing the interrupt flag (IF). The interrupt initiated through NMI pin and all software interrupts are non-maskable



Software Interrupts

The software interrupts are program instructions. These instructions are inserted at desired locations in a program. While running a program, if software interrupt instruction is encountered then the processor initiates an interrupt. The 8086 processor has 256 types of software interrupts. The software interrupt instruction is INT n, where n is the type number in the range 0 to 255

Software Interrupt (INT n)

- Used by operating systems to provide hooks into various function
- Used as a communication mechanism between different parts of the program

8086 INTERRUPT TYPES 256 INTERRUPTS OF 8086 ARE DIVIDED INTO 3 GROUPS

1. TYPE 0 TO TYPE 4 INTERRUPTS
   - These Are Used For Fixed Operations And Hence Are Called Dedicated Interrupt
2. TYPE 5 TO TYPE 31 INTERRUPTS
   - Not Used By 8086,reserved For Higher Processors Like 80286 80386 Etc
3. TYPE 32 TO 255 INTERRUPTS
   - Available For User, called User Defined Interrupts These Can Be H/W Interrupts And Activated Through Intr Line Or Can Be S/W Interrupts

- Type – 0 Divide Error Interrupt
  - Quotient Is Large Can't Be Fit In Al/Ax Or Divide By Zero
- Type –1 Single Step Interrupt
  - Used For Executing The Program In Single Step Mode By Setting Trap Flag
- Type – 2 Non Maskable Interrupt
  - This Interrupt Is Used For Execution Of NMI Pin.
- Type – 3 BreakPoint Interrupt
  - Used For Providing Break Points In The Program
- Type – 4 Overflow Interrupt
  - Used To Handle Any Overflow Error.

Conclusion

The CPU executes the program, as soon as a key is pressed, the Keyboard generates an interrupt. The CPU will respond to the interrupt – read the data. After that, it returns to the original program. So by proper use of interrupt, the CPU can serve many devices at the "same time".

Difficulty Level : Medium • Last Updated : 17 Aug, 2018

An interrupt is a condition that halts the microprocessor temporarily to work on a different task and then return to its previous task. Interrupt is an event or signal that request to attention of CPU. This halt allows peripheral devices to access the microprocessor.

Whenever an interrupt occurs the processor completes the execution of the current instruction and starts the execution of an Interrupt Service Routine (ISR) or Interrupt Handler. ISR is a program that tells the processor what to do when the interrupt occurs. After the execution of ISR, control returns back to the main routine where it was interrupted.

In 8086 microprocessor following tasks are performed when microprocessor encounters an interrupt:

1. The value of flag register is pushed into the stack. It means that first the value of SP (Stack Pointer) is decremented by 2 then the value of flag register is pushed to the memory address of stack segment.
2. The value of starting memory address of CS (Code Segment) is pushed into the stack.
3. The value of IP (Instruction Pointer) is pushed into the stack.
4. IP is loaded from word location (Interrupt type) * 04.
5. CS is loaded from the next word location.
6. Interrupt and Trap flag are reset to 0.

The different types of interrupts present in 8086 microprocessor are given by:

1. **Hardware Interrupts –**
   Hardware interrupts are those interrupts which are caused by any peripheral device by sending a signal through a specified pin to the microprocessor. There are two hardware interrupts in 8086 microprocessor. They are:
   - *(A) NMI (Non Maskable Interrupt)* – It is a single pin non maskable hardware interrupt which cannot be disabled. It is the highest priority interrupt in 8086 microprocessor. After its execution, this interrupt generates a TYPE 2 interrupt. IP is loaded from word location 00008 H and CS is loaded from the word location 0000A H.
   - *(B) INTR (Interrupt Request)* – It provides a single interrupt request and is activated by I/O port. This interrupt can be masked or delayed. It is a level triggered interrupt. It can receive any interrupt type, so the value of IP and CS will change on the interrupt type received.

2. **Software Interrupts –** These are instructions that are inserted within the program to generate interrupts. There are 256 software interrupts in 8086 microprocessor. The instructions are of the format INT type where type ranges from 00 to FF. The starting address ranges from 00000 H to 003FF H. These are 2 byte instructions. IP is loaded from type * 04 H and CS is loaded from the next address give by (type * 04) + 02 H. Some important software interrupts are:
   - (A) *TYPE 0* corresponds to division by zero(0).
   - (B) *TYPE 1* is used for single step execution for debugging of program.
   - (C) *TYPE 2* represents NMI and is used in power failure conditions.
   - (D) *TYPE 3* represents a break-point interrupt.
   - (E) *TYPE 4* is the overflow interrupt.

   Refer for – Interrupts in 8085 microprocessor

CHAPTER 1

Computer Number Systems, Codes, and Digital Devices

Before starting our discussion of microprocessors and microcomputers, we need to make sure that some key concepts of the number systems, codes, and digital devices used in microcomputers are fresh in your mind. If the short summaries of these concepts in this chapter are not enough to refresh your memory, then you may want to consult some of the chapters in *Digital Circuits and Systems*, McGraw-Hill, 1989, before going on in this book.

OBJECTIVES

At the conclusion of this chapter you should be able to:

1. Convert numbers between the following codes: binary, hexadecimal, and BCD.
2. Define the terms *bit*, *nibble*, *byte*, *word*, *most significant bit*, and *least significant bit*.
3. Use a table to find the ASCII or EBCDIC code for a given alphanumeric character.
4. Perform addition and subtraction of binary, hexadecimal, and BCD numbers.
5. Describe the operation of gates, flip-flops, latches, registers, ROMs, PALs, dynamic RAMs, static RAMs, and buses.
6. Describe how an arithmetic logic unit can be instructed to perform arithmetic or logical operations on binary words.

COMPUTER NUMBER SYSTEMS AND CODES

Review of Decimal System

To understand the structure of the binary number system, the first step is to review the familiar decimal or base-10 number system. Here is a decimal number with the value of each place holder or digit expressed as a power of 10.

5	3	4	6	.	7	2
10^3	10^2	10^1	10^0		10^{-1}	10^{-2}

The digits in the decimal number 5346.72 thus tell you that you have 5 thousands, 3 hundreds, 4 tens, 6 ones, 7 tenths, and 2 hundredths. The number of symbols needed in any number system is equal to the base number. In the decimal number system, then, there are 10 symbols, 0 through 9. When the count in any digit position passes that of the highest-value symbol, the digit rolls back to 0 and the next higher digit is incremented by 1. A car odometer is a good example of this.

A number system can be built using powers of any number as place holders or digits, but some bases are more useful than others. It is difficult to build electronic circuits which can store and manipulate 10 different voltage levels but relatively easy to build circuits which can handle two levels. Therefore, a *binary*, or *base-2*, number system is used to represent numbers in digital systems.

The Binary Number System

Figure 1-1*a*, p. 2, shows the value of each digit in a binary number. Each binary digit represents a power of 2. A binary digit is often called a *bit*. Note that digits to the right of the *binary point* represent fractions used for numbers less than 1. The binary system uses only two symbols, zero (0) and one (1), so in binary you count as follows: 0, 1, 10, 11, 100, 101, 110, 111, 1000, etc. For reference, Figure 1-1*b* shows the powers of 2 from 2^1 to 2^{32}.

Binary numbers are often called *binary words* or just *words*. Binary words with certain numbers of bits have also acquired special names. A 4-bit binary word is called a *nibble*, and an 8-bit binary word is called a *byte*. A 16-bit binary word is often referred to just as a *word*, and a 32-bit binary word is referred to as a *doubleword*. The rightmost or *least significant bit* of a binary word is usually referred to as the LSB. The leftmost or *most significant bit* of a binary word is usually called the MSB.

To convert a binary number to its equivalent decimal number, multiply each digit times the decimal value of the digit and just add these up. The binary number 101, for example, represents: $(1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$,

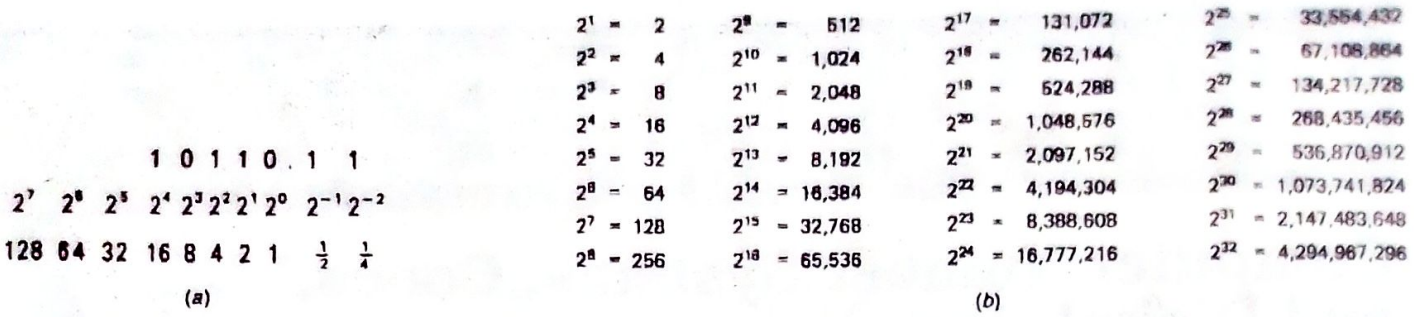


FIGURE 1-1 (*a*) Digit values in binary. (*b*) Powers of 2.

or 4 + 0 + 1 = decimal 5. For the binary number 10110.11, you have:

$$(1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) + (1 \times 2^{-1}) + (1 \times 2^{-2}) =$$
$$16 + 0 + 4 + 2 + 0 + 0.5 + 0.25 = \text{decimal } 22.75$$

To convert a decimal number to binary, there are two common methods. The first (Figure 1-2*a*) is simply a reverse of the binary-to-decimal method. For example, to convert the decimal number 21 (sometimes written as 21_{10}) to binary, first subtract the largest power of 2 that will fit in the number. For 21_{10} the largest power of 2 that will fit is 16 or 2^4. Subtracting 16 from 21 gives a remainder of 5. Put a 1 in the 2^4 digit position and see if the next lower power of 2 will fit in the remainder. Since 2^3 is 8 and 8 will not fit in the remainder of 5, put a 0 in the 2^3 digit position. Then try the next lower power of 2. In this case the next is 2^2 or 4, which will fit in the remainder of 5. A 1 is therefore put in the 2^2 digit position. When 2^2 or 4 is subtracted from the old remainder of 5, a new remainder of 1 is left. Since 2^1 or 2 will not fit into this remainder, a 0 is put in that position. A 1 is put in the 2^0 position because 2^0 is equal to 1 and this fits exactly into the remainder of 1. The result shows that 21_{10} is equal to 10101 in binary. This conversion process is somewhat messy to describe but easy to do. Try converting 46_{10} to binary. You should get 101110.

Another method of converting a decimal number to binary is shown in Figure 1-2*b*. Divide the decimal number by 2 and write the quotient and remainder as shown. Divide this quotient and following quotients by 2 until the quotient reaches 0. The column of remainders will be the binary equivalent of the given decimal number. Note that the MSD is on the bottom of the column and the LSD is on the top of the column if you perform the divisions in order from the top to the bottom of the page. You can demonstrate that the binary number is correct by reconverting from binary to decimal, as shown in the right-hand side of Figure 1-2*b*.

You can convert decimal numbers less than 1 to binary by successive multiplication by 2, recording carries until the quantity to the right of the decimal point becomes zero, as shown in Figure 1-2*c*. The carries represent the binary equivalent of the decimal number, with the *most significant bit* at the top of the column. Decimal 0.625 equals 0.101 in binary. For decimal values that do not convert exactly the way this one did (the quantity to the right of the decimal never becomes zero), you can continue the conversion process until you get the number of binary digits desired.

At this point it is interesting to compare the number of digits required to express numbers in decimal with the number required to express them in binary. In

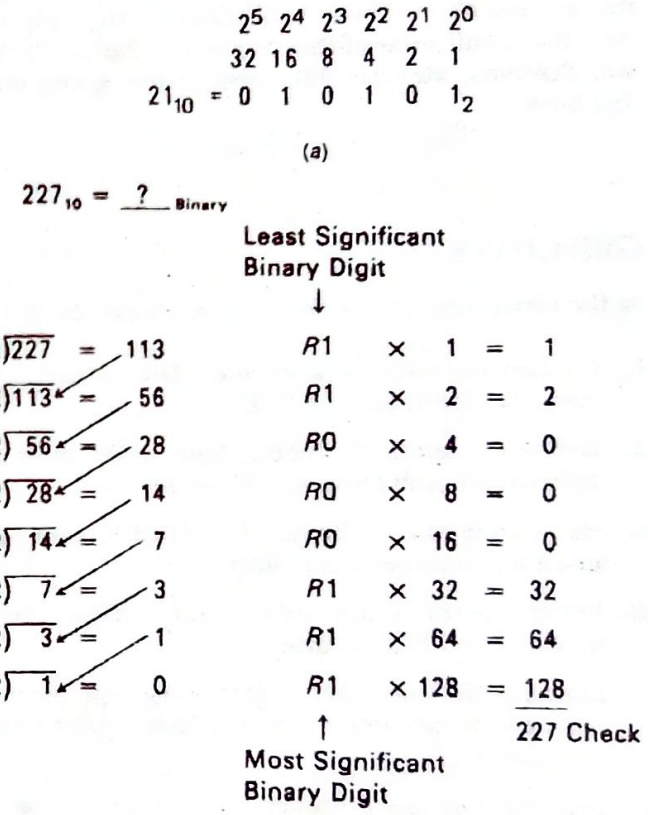


FIGURE 1-2 Converting decimal to binary. (*a*) Digit value method. (*b*) Divide by 2 method. (*c*) Decimal fraction conversion.

decimal, one digit can represent 10^1 numbers, 0 through 9; two digits can represent 10^2 or 100 numbers, 0 through 99; and three digits can represent 10^3 or 1000 numbers, 0 through 999. In binary, a similar pattern exists. One binary digit can represent 2 numbers, 0 and 1; two binary digits can represent 2^2 or 4 numbers, 0 through 11; and three binary digits can represent 2^3 or 8 numbers, 0 through 111. The pattern, then, is that N decimal digits can represent 10^N numbers and N binary digits can represent 2^N numbers. Eight binary digits can represent 2^8 or 256 numbers, 0 through 255 in decimal.

Hexadecimal

Binary is not a very compact code. This means that it requires many more digits to express a number than does, for example, decimal. Twelve binary digits can only describe a number up to 4095_{10}. Computers require binary data, but people working with computers have trouble remembering long binary words. One solution to the problem is to use the *hexadecimal* or base-16 number system.

Figure 1-3*a* shows the digit values for hexadecimal, which is often just called *hex*. Since hex is base 16, you have to have 16 possible symbols, one for each digit. The table of Figure 1-3*b* shows the symbols for hex code.

$$16^3 \quad 16^2 \quad 16^1 \quad 16^0 . \quad 16^{-1} \quad 16^{-2} \quad 16^{-3}$$
$$4096 \quad 256 \quad 16 \quad 1 \quad \frac{1}{16} \quad \frac{1}{256} \quad \frac{1}{4096}$$

(*a*)

Dec	Hex	Dec	Hex
0	0	8	8
1	1	9	9
2	2	10	A
3	3	11	B
4	4	12	C
5	5	13	D
6	6	14	E
7	7	15	F

(*b*)

$227_D = \underline{\quad ? \quad}_{Hex}$ LSD

$16\overline{)227} = 14$ *R*3 × 1 = 3

$16\overline{)14} = 0$ *R*E × 16 = 224

MSD 227

$227_{10} = E3_{16}$

(*c*)

FIGURE 1-3 Hexadecimal numbers. (*a*) Value of place holders. (*b*) Symbols. (*c*) Decimal-to-hexadecimal conversion.

After the decimal symbols 0 through 9 are used up, you use the letters A through F for values 10 through 15.

As mentioned above, each hex digit is equal to four binary digits. To convert the binary number 11010110 to hex, mark off the binary bits in groups of 4, moving to the left from the binary point. Then write the hex symbol for the value of each group of 4.

Binary	1101	0110
Hex	D	6

The 0110 group is equal to 6 and the 1101 group is equal to 13. Since 13 is D in hex, 11010110 binary is equal to D6 in hex. "H" is usually used after a number to indicate that it is a hexadecimal number. For example, D6 hex is usually written D6H. As you can see, 8 bits can be represented with only 2 hex digits.

If you want to convert a number from decimal to hexadecimal, Figure 1-3*c* shows a familiar trick for doing this. The result shows that 227_{10} is equal to E3H. As you can see, hex is an even more compact code than decimal. Two hexadecimal digits can represent a decimal number up to 255. Four hex digits can represent a decimal number up to 65,535.

To illustrate how hexadecimal numbers are used in digital logic, a service manual tells you that the 8-bit-wide data bus of an 8088A microprocessor should contain 3FH during a certain operation. Converting 3FH to binary gives the pattern of 1's and 0's (0011 1111) you would expect to find with your oscilloscope or logic analyzer on the parallel lines. The 3FH is simply a shorthand which is easier to remember and less prone to errors than the binary equivalent.

BCD Codes

STANDARD BCD

In applications such as frequency counters, digital voltmeters, or calculators, where the output is a decimal display, a *binary-coded decimal* or *BCD* code is often used. BCD uses a 4-bit binary code to individually represent each decimal digit in a number. As you can see in Table 1-1, p. 4, the simplest BCD code uses the first 10 numbers of standard binary code for the BCD numbers 0 through 9. The hex codes A through F are invalid BCD codes. To convert a decimal number to its BCD equivalent, just represent each decimal digit by its 4-bit binary equivalent, as shown here.

Decimal	5	2	9
BCD	0101	0010	1001

To convert a BCD number to its decimal equivalent, reverse the process.

GRAY CODE

Gray code is another important binary code; it is often used for encoding shaft position data from machines such as computer-controlled lathes. This code has the same possible combinations as standard binary, but as you can see in the 4-bit example in Table 1-1, they are

TABLE 1-1
COMMON NUMBER CODES

Decimal	Binary	Octal	Hex	Binary-Coded Decimal		Reflected Gray Code	7-Segment Display (1 = on)	
				8421 BCD	EXCESS-3		a b c d e f g	Display
0	0000	0	0	0000	0011 0011	0000	1 1 1 1 1 1 0	0
1	0001	1	1	0001	0011 0100	0001	0 1 1 0 0 0 0	1
2	0010	2	2	0010	0011 0101	0011	1 1 0 1 1 0 1	2
3	0011	3	3	0011	0011 0110	0010	1 1 1 1 0 0 1	3
4	0100	4	4	0100	0011 0111	0110	0 1 1 0 0 1 1	4
5	0101	5	5	0101	0011 1000	0111	1 0 1 1 0 1 1	5
6	0110	6	6	0110	0011 1001	0101	1 0 1 1 1 1 1	6
7	0111	7	7	0111	0011 1010	0100	1 1 1 0 0 0 0	7
8	1000	10	8	1000	0011 1011	1100	1 1 1 1 1 1 1	8
9	1001	11	9	1001	0011 1100	1101	1 1 1 0 0 1 1	9
10	1010	12	A	0001 0000	0100 0011	1111	1 1 1 1 1 0 1	A
11	1011	13	B	0001 0001	0100 0100	1110	0 0 1 1 1 1 1	B
12	1100	14	C	0001 0010	0100 0101	1010	0 0 0 1 1 0 1	C
13	1101	15	D	0001 0011	0100 0110	1011	0 1 1 1 1 0 1	D
14	1110	16	E	0001 0100	0100 0111	1001	1 1 0 1 1 1 1	E
15	1111	17	F	0001 0101	0100 1000	1000	1 0 0 0 1 1 1	F

arranged in a different order. Notice that only one binary digit changes at a time as you count up in this code.

If you need to construct a Gray-code table larger than that in Table 1-1, a handy way to do so is to observe the pattern of 1's and 0's and just extend it. The least significant digit column starts with one 0 and then has alternating groups of two 1's and two 0's as you go down the column. The second most significant digit column starts with two 0's and then has alternating groups of four 1's and four 0's. The third column starts with four 0's, then has alternating groups of eight 1's and eight 0's. By now you should see the pattern. Try to figure out the Gray code for the decimal number 16. You should get 11000.

7-Segment Display Code

Figure 1-4*a* shows the segment identifiers for a 7-segment display such as those commonly used in digital instruments. Table 1-1 shows the logic levels required to display 0 to 9 and A to F on a common-cathode LED display such as that shown in Figure 1-4*b*. For a common-anode LED display such as that in Figure 1-4*c*, simply invert the segment codes shown in Table 1-1.

Alphanumeric Codes

When communicating with or between computers, you need a binary-based code which can represent letters of the alphabet as well as numbers. Common codes used for this have 7 or 8 bits per word and are referred to as *alphanumeric codes*. To detect possible errors in these codes, an additional bit, called a *parity bit*, is often added as the most significant bit.

Parity is a term used to identify whether a data word has an odd or even number of 1's. If a data word contains an odd number of 1's, the word is said to have *odd parity*. The binary word 0110111 with five 1's has odd parity. The binary word 0110000 has an even number of 1's (two), so it has *even parity*.

In practice the parity bit is used as follows. The system that is sending a data word checks the parity of the word. If the parity of the data word is odd, the system will set the parity bit to a 1. This makes the parity of the data word plus parity bit even. If the parity of the data word is even, the sending system will reset the parity bit to a 0. This again makes the parity of the data word plus parity even. The receiving system checks the

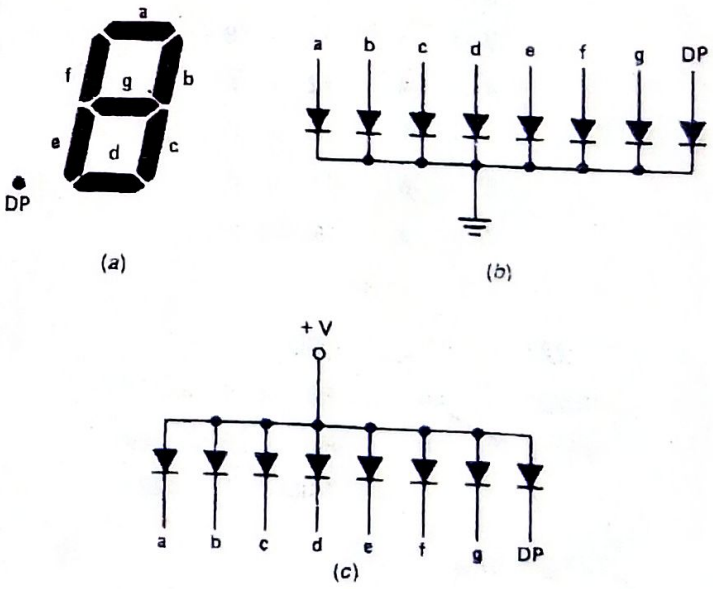


FIGURE 1-4 7-segment LED display. (*a*) Segment labels. (*b*) Schematic of common-cathode type. (*c*) Schematic of common-anode type.

parity of the data word plus parity bit that it receives. If the receiving system detects odd parity in the received data word plus parity, it assumes an error has occurred and tells the sending system to send the data again. The system is then said to be using even parity. The system could have been set up to use (maintain) odd parity in a similar manner.

ASCII

Table 1-2 shows several alphanumeric codes. The first of these is *ASCII*, or American Standard Code for Information Interchange. This is shown in the table as a 7-bit code. With 7 bits you can code up to 128 characters, which is enough for the full upper- and lowercase

TABLE 1-2
COMMON ALPHANUMERIC CODES

ASCII Symbol	*HEX Code for 7-Bit ASCII*	*EBCDIC Symbol*	*HEX Code for EBCDIC*	*ASCII Symbol*	*HEX Code for 7-Bit ASCII*	*EBCDIC Symbol*	*HEX Code for EBCDIC*	*ASCII Symbol*	*HEX Code for 7-Bit ASCII*	*EBCDIC Symbol*	*HEX Code for EBCDIC*
NUL	00	NUL	00	*	2A	*	5C	T	54	T	E3
SOH	01	SOH	01	+	2B	+	4E	U	55	U	E4
STX	02	STX	02	,	2C	,	6B	V	56	V	E5
ETX	03	ETX	03	-	2D	-	60	W	57	W	E6
EOT	04	EOT	37	.	2E	.	4B	X	58	X	E7
ENQ	05	ENQ	2D	/	2F	/	61	Y	59	Y	E8
ACK	06	ACK	2E	0	30	0	F0	Z	5A	Z	E9
BEL	07	BEL	2F	1	31	1	F1	[	5B	[	AD
BS	08	BS	16	2	32	2	F2	\	5C	NL	15
HT	09	HT	05	3	33	3	F3	]	5D	]	DD
LF	0A	LF	25	4	34	4	F4	^	5E	¬	5F
VT	0B	VT	0B	5	35	5	F5	_	5F	—	6D
FF	0C	FF	0C	6	36	6	F6	`	60	RES	14
CR	0D	CR	0D	7	37	7	F7	a	61	a	81
SO	0E	SO	0E	8	38	8	F8	b	62	b	82
SI	0F	SI	0F	9	39	9	F9	c	63	c	83
DLE	10	DLE	10	:	3A	:	7A	d	64	d	84
DC1	11	DC1	11	;	3B	;	5E	e	65	e	85
DC2	12	DC2	12	<	3C	<	4C	f	66	f	86
DC3	13	DC3	13	=	3D	=	7E	g	67	g	87
DC4	14	DC4	35	>	3E	>	6E	h	68	h	88
NAK	15	NAK	3D	?	3F	?	6F	i	69	i	89
SYN	16	SYN	32	@	40	@	7C	j	6A	j	91
ETB	17	EOB	26	A	41	A	C1	k	6B	k	92
CAN	18	CAN	18	B	42	B	C2	l	6C	l	93
EM	19	EM	19	C	43	C	C3	m	6D	m	94
SUB	1A	SUB	3F	D	44	D	C4	n	6E	n	95
ESC	1B	BYP	24	E	45	E	C5	o	6F	o	96
FS	1C	FLS	1C	F	46	F	C6	p	70	p	97
GS	1D	GS	1D	G	47	G	C7	q	71	q	98
RS	1E	RDS	1E	H	48	H	C8	r	72	r	99
US	1F	US	1F	I	49	I	C9	s	73	s	A2
SP	20	SP	40	J	4A	J	D1	t	74	t	A3
!	21	!	5A	K	4B	K	D2	u	75	u	A4
"	22	"	7F	L	4C	L	D3	v	76	v	A5
#	23	#	7B	M	4D	M	D4	w	77	w	A6
$	24	$	5B	N	4E	N	D5	x	78	x	A7
%	25	%	6C	O	4F	O	D6	y	79	y	A8
&	26	&	50	P	50	P	D7	z	7A	z	A9
'	27	'	7D	Q	51	Q	D8	{	7B	{	8B
(	28	(	4D	R	52	R	D9	|	7C	|	4F
)	29	)	5D	S	53	S	E2	}	7D	}	9B
								~	7E	¢	4A
								DEL	7F	DEL	07

TABLE 1-3
DEFINITIONS OF CONTROL CHARACTERS

NULL	Null	DC1	Direct control 1
SOH	Start of heading	DC2	Direct control 2
STX	Start text	DC3	Direct control 3
ETX	End text	DC4	Direct control 4
EOT	End of transmission	NAK	Negative acknowledge
ENQ	Enquiry	SYN	Synchronous idle
ACK	Acknowledge	ETB	End transmission block
BEL	BS	CAN	Cancel
BS	Backspace	EM	End of medium
HT	Horizontal tab	SUB	Substitute
LF	Line feed	ESC	Escape
VT	Vertical tab	FS	Form separator
FF	Form feed	GS	Group separator
CR	Carriage return	RS	Record separator
SO	Shift out	US	Unit separator
SI	Shift in		
DLE	Data link escape		

alphabet, numbers, punctuation marks, and control characters. The code is arranged so that if only uppercase letters, numbers, and a few control characters are needed, the lower 6 bits are all that are required. If a parity check is wanted, a parity bit is added to the basic 7-bit code in the MSB position. The binary word 1100 0100, for example, is the ASCII code for uppercase D with odd parity. Table 1-3 gives the meanings of the control character symbols used in the ASCII code table.

EBCDIC

Another alphanumeric code commonly encountered in IBM equipment is the Extended Binary-Coded Decimal Interchange Code or *EBCDIC*. This is an 8-bit code without parity. A ninth bit can be added for parity. To save space in Table 1-2, the eight binary digits of EBCDIC are represented by their 2-digit hex equivalent.

ARITHMETIC OPERATIONS ON BINARY, HEX, AND BCD NUMBERS

Binary Arithmetic

ADDITION

Figure 1-5*a* shows the truth table for addition of two binary digits and a carry in (C_{IN}) from addition of previous digits. Figure 1-5*b* shows the result of adding two 8-bit binary numbers together using these rules. Assuming that C_{IN} = 1, 1 + 0 + C_{IN} = a sum of 0 and a carry into the next digit, and 1 + 1 + C_{IN} = a sum of 1 and a carry into the next digit because the result in any digit position can only be a 1 or a 0.

2'S-COMPLEMENT SIGNS-AND-MAGNITUDE BINARY

When you handwrite a number that represents some physical quantity such as temperature, you can simply put a + sign in front of the number to indicate that the number is positive, or you can write a − sign to indicate that the number is negative. However, if you want to store values such as temperatures, which can be positive or negative, in a computer memory, there is a problem. Since the computer memory can store only 1's and 0's, some way must be established to represent the sign of the number with a 1 or a 0.

A common way to represent signed numbers is to reserve the most significant bit of the data word as a *sign bit* and to use the rest of the bits of the data word to represent the size (magnitude) of the quantity. A computer that works with 8-bit words will use the MSB (bit 7) as the sign bit and the lower 7 bits to represent the magnitude of the numbers. The usual convention is to represent a positive number with a 0 sign bit and a negative number with a 1 sign bit.

To make computations with signed numbers easier, the magnitude of negative numbers is represented in a special form called *2's complement*. The 2's complement of a binary number is formed by inverting each bit of the data word and adding 1 to the result. Some examples should help clarify all of this.

The number $+7_{10}$ is represented in 8-bit sign-and-magnitude form as 00000111. The sign bit is 0, which indicates a positive number. The magnitude of positive numbers is represented in straight binary, so 00000111 in the least significant bits represents 7_{10}.

To represent -7_{10} in 8-bit 2's-complement sign-and-magnitude form, start with the 8-bit code for +7, 0000 0111. Invert each bit, including the MSB, to get 1111 1000. Then add 1 to get 11111001. This result is the correct representation of -7_{10}. Figure 1-6 shows some more examples of positive and negative numbers expressed in 8-bit sign-and-magnitude form. For practice, try generating each of these yourself to see if you get the same result.

To reverse this procedure and find the magnitude of a number expressed in sign-and-magnitude form, proceed as follows. If the number is positive, as indicated

INPUTS			OUTPUTS	
A	B	C_{IN}	S	C_{OUT}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(*a*)

$S = A \oplus B \oplus C_{IN}$

$C_{OUT} = A \cdot B + C_{IN}(A \oplus B)$

```
  10011010
+ 11011100
-----------
 [1] 01110110
  ↑
  └ Carry
```

(*b*)

FIGURE 1-5 Binary addition. (*a*) Truth table for 2 bits plus carry. (*b*) Addition of two 8-bit words.

	Sign bit ↓	
+ 7	0	0000111
+ 46	0	0101110
+105	0	1101001
− 12	1	1110100
− 54	1	1001010
−117	1	0001011
− 46	1	1010010

Sign and two's complement of magnitude

FIGURE 1-6 Positive and negative numbers represented with a sign bit and 2's complement.

by the sign bit being a 0, then the least significant 7 bits represent the magnitude directly in binary. If the number is negative, as indicated by the sign bit being a 1, then the magnitude is expressed in 2's complement. To get the magnitude of this negative number expressed in standard binary, invert each bit of the data word, including the sign bit, and add 1 to the result. For example, given the word 11101011, invert each bit to get 00010100. Then add 1 to get 00010101. This equals 21_{10}, so you know that the original numbers represent -21_{10}. Again, try reconverting a few of the numbers in Figure 1-6 for practice.

Figure 1-7 shows some examples of addition of signed binary numbers of this type. Sign bits are added together just as the other bits are. Figure 1-7*a* shows the results of adding two positive numbers. The sign bit of the result is zero, so the result is positive. The second example, in Figure 1-7*b*, adds a −9 to a +13 or, in effect, subtracts 9 from 13. As indicated by the zero sign bit, the result of 4 is positive and in true binary form.

Figure 1-7*c* shows the result of adding a −13 to a smaller positive number, +9. The sign bit of the result is a 1. This indicates that the result is negative and the magnitude is in 2's-complement form. To reconvert a 2's complement result to a signed number in true binary form:

1. Invert each bit to produce the 1's complement.
2. Add 1.
3. Put a minus sign in front to indicate that the result is negative.

The final example, in Figure 1-7*d*, shows the result of adding two negative numbers. The sign bit of the result is a 1, so the result is negative and in 2's-complement form. Again, inverting each bit, adding 1, and prefixing a minus sign will put the result in a more recognizable form.

Now let's consider the range of numbers that can be represented with 8 bits in sign-and-magnitude form. Eight bits can represent a maximum of 2^8 or 256 numbers. Since we are representing both positive and negative numbers, half of this range will be positive and half negative. Therefore, the range is −128 to +127. Here are the sign-and-magnitude binary representations for these values:

01111111	+127
⋮	
00000001	+1
00000000	zero
11111111	−1
⋮	
10000001	−127
10000000	−128

If you like number patterns, you might notice that this scheme shifts the normal codes for 128 to 255 downward to represent −128 to −1.

If a computer is storing signed numbers as 16-bit words, then a much larger range of numbers can be represented. Since 16 bits gives 2^{16} or 65,536 possible values, the range for 16-bit sign-and-magnitude numbers is −32,768 to +32,767. Operations with 16-bit sign-and-magnitude numbers are done the same way as operations with 8-bit sign-and-magnitude numbers.

```
+13     00001101
+ 9     00001001
+22     00010110
          └─Sign bit is 0
            so result is positive
```
(a)

```
+13     00001101
− 9     11110111  2's complement for −9 with sign bit
+ 4   1 00000100
        └─Sign bit is 0
          so result is positive
      └─Ignore carry
```
(b)

```
+ 9     00001001
−13     11110011  2's complement for −13 with sign bit
− 4     11111100  Sign bit is 1
        00000011  So invert each bit
      +         1 Add 1
equals ─00000100  Prefix with minus sign
```
(c)

```
− 9     11110111 } 2's complement,
−13     11110011 } sign-and-magnitude form
−22     11101010  Sign bit is 1
        00010101  So invert each bit
      +         1 Add 1
equals ─00010110  Prefix with minus sign
```
(d)

FIGURE 1-7 Addition of signed binary numbers. (*a*) +9 and +13. (*b*) −9 and +13. (*c*) +9 and −13. (*d*) −9 and −13.

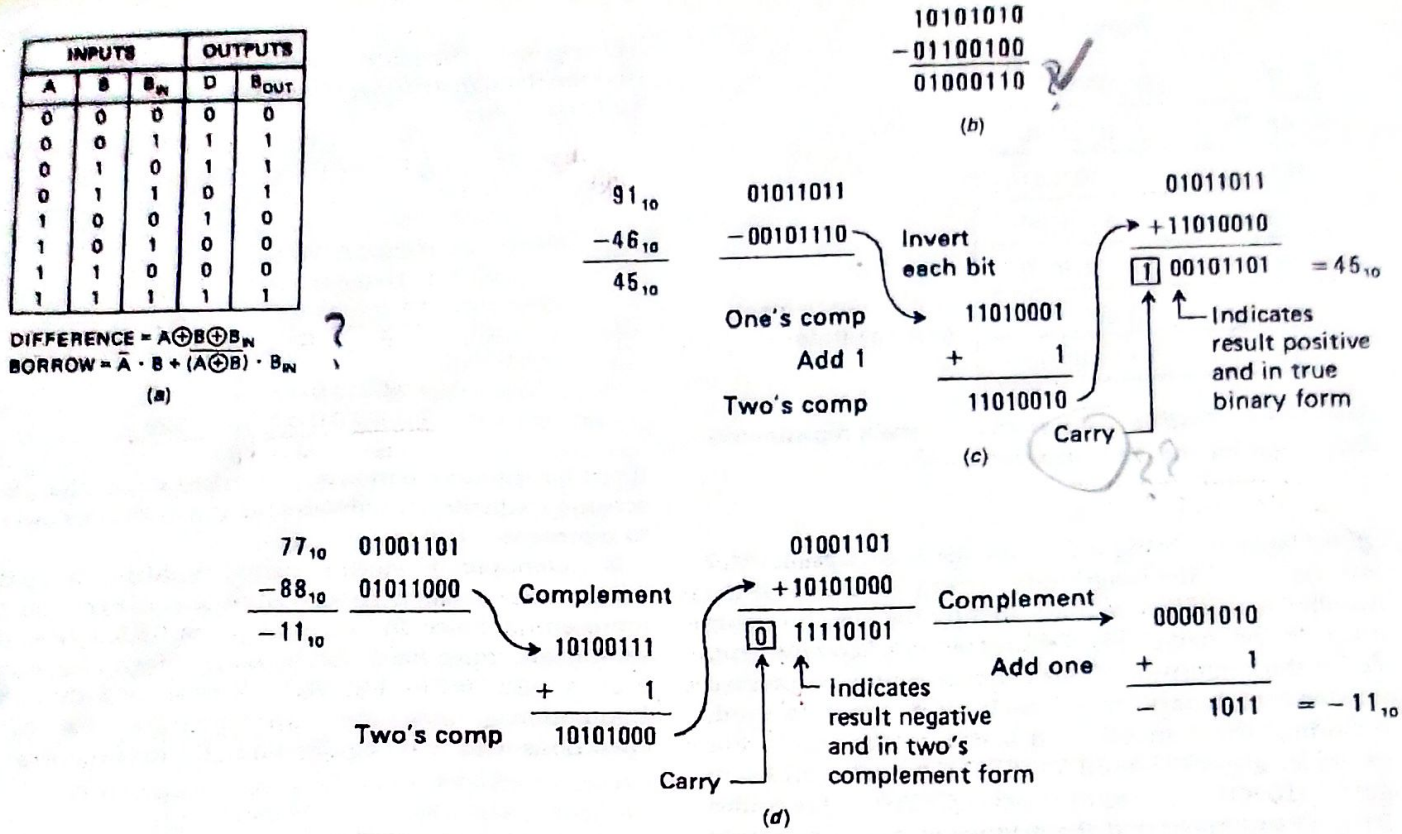


FIGURE 1-8 Binary subtraction. (a) Truth table for 2 bits and borrow. (b) Pencil method. (c) 2's-complement positive result. (d) 2's-complement negative result.

SUBTRACTION

There are two common methods for doing binary subtraction. These are the pencil method and the 2's-complement add method. Figure 1-8*a* shows the truth table for binary subtraction of two binary digits A and B. Also included in the truth table is the effect of a borrow-in, B_{IN}, from subtracting previous digits. Figure 1-8*b* shows an example of the "pencil" method of subtracting two 8-bit numbers. Using the truth table, this method is done the same way that you do decimal subtraction.

A second method of performing binary subtraction is by adding the 2's-complement representation of the bottom number (subtrahend) to the top number (minuend). Figure 1-8*c* shows how this is done. First represent the top number in sign-and-magnitude form. Then form the 2's-complement sign-and-magnitude representation for the negative of the bottom number. Finally, add the two parts formed. For the example in Figure 1-8*c*, the sign of the result is a 0, which indicates that the result is positive and in true form. The final carry produced by the addition can be ignored. Figure 1-8*d* shows another example of this method of subtraction. In this case the bottom number is larger than the top number. Again, represent the top number in sign-and-magnitude form, produce the 2's-complement sign-and-magnitude form for the negative of the bottom number, and add the two together. The sign bit of the result is a 1 for this example. This indicates that the result is negative and its magnitude is represented in 2's-complement form. To get the result into a form that is more recognizable to you, invert each bit of the result, add 1 to it, and put a minus sign in front of it as shown in Figure 1-8*d*.

Problems that may occur when doing signed addition or subtraction are *overflow* and *underflow*. If the magnitude of the number produced by adding two signed numbers is larger than the number of bits available to represent the magnitude, the result will "overflow" into the sign bit position and give an incorrect result. For example, if the signed positive number 01001001 is added to the signed positive number 01101101, the result is 10110110. The 1 in the MSB of this result indicates that it is negative, which is obviously incorrect for the sum of two positive numbers. In a similar manner, doing an 8-bit signed subtraction that produces a magnitude greater than −128 will cause an "underflow" into the sign bit and produce an incorrect result.

For simplicity the examples shown use 8 bits, but the method works for any number of bits. This method may seem awkward, but it is easy to do in a computer or microprocessor because it requires only the simple operations of inverting and adding.

MULTIPLICATION

There are several methods of doing binary multiplication. Figure 1-9 shows what is called the *pencil method* because it is the same way you learned to multiply decimal numbers. The top number, or multiplicand, is multiplied by the least significant digit of the bottom number, or multiplier. The partial product is written

```
  11         1011     MULTIPLICAND
× 9        × 1001     MULTIPLIER
           ------
             1011 ⎫
            0000  ⎬   PARTIAL PRODUCTS
           0000   ⎪
          1011    ⎭
          -------
          1100011     PRODUCT
```

FIGURE 1-9 Binary multiplication.

down. The top number is then multiplied by the next digit of the multiplier. The resultant partial product is written down under the last, but shifted one place to the left. Adding all the partial products gives the total product. This method works well when doing multiplication by hand, but it is not practical for a computer because the type of shifts required makes it awkward to implement.

One of the multiplication methods used by computers is repeated addition. To multiply 7 × 55, for example, the computer can just add up seven 55's. For large numbers, however, this method is slow. To multiply 786 × 253, for example, requires 252 add operations.

Most computers use an add-and-shift-right method. This method takes advantage of the fact that for binary multiplication, the partial product can only be either the top number exactly if the multiplier digit is a 1 or a 0 if the multiplier digit is a 0. The method does the same thing as the pencil method, except that the partial products are added as they are produced and the sum of the partial products is shifted right rather than each partial product being shifted left.

A point to note about multiplying numbers is the number of bits the product requires. For example, multiplying two 4-bit numbers can give a product with as many as 8 bits, and two 8-bit numbers can give a 16-bit product.

DIVISION

Binary division can also be performed in several ways. Figure 1-10 shows two examples of the pencil method. This is the same process as decimal long division. However, it is much simpler than decimal long division because the digits of the result (quotient) can only be 0 or 1. A division is attempted on part of the dividend. If this is not possible because the divisor is larger than that part of the dividend, a 0 is entered in the quotient. Another attempt is then made to divide using one more digit of the dividend. When a division is possible, a 1 is entered in the quotient. The divisor is then subtracted from the portion of the dividend used. As with standard long division, the process is continued until all the dividend is used. As shown in Figure 1-10*b*, 0's can be added to the right of the binary point and division continued to convert a remainder to a binary equivalent.

Another method of division that is easier for computers and microprocessors to perform uses successive subtractions. The divisor is subtracted from the dividend and from each successive remainder until a borrow is produced. The desired quotient is 1 less than the number of subtractions needed to produce a borrow. This method is simple, but for large numbers it is slow.

For faster division of large numbers, computers use a subtract-and-shift-left method that is essentially the same process you go through with a pencil long division.

```
                 01100    QUOTIENT
DIVISOR    110 ) 1001000  DIVIDEND          12
                -110                    6 ) 72
                 ----
                   110
                  -110
                  ----
                     0
                  (a)

               110.01                     6.25
        100 ) 11001.00                4 ) 25
             -100
              ----
                100
               -100
               ----
                 0100
                  (b)
```

FIGURE 1-10 Binary division.

Hexadecimal Addition and Subtraction

People working with computers or microprocessors often use hexadecimal as a shorthand way of representing long binary numbers such as memory addresses. It is therefore useful to be able to add and subtract hexadecimal numbers.

ADDITION

As shown in Figure 1-11*a*, one way to add two hexadecimal numbers is to convert each hexadecimal number to its binary equivalent, add the two binary numbers, and convert the binary result back to its hex equivalent. For converting to binary, remember that each hex digit represents 4 binary digits.

A second method, shown in Figure 1-11*b*, works directly with the hex numbers. When adding hex digits, a carry is produced whenever the sum is 16 decimal or greater. Another way of saying this is that the value of a carry in hex is 16 decimal. For the least significant digits in Figure 1-11*b*, an A in hex is 10 in decimal and an F is 15 in decimal. These add to give 25 decimal. This is greater than 16, so mentally subtract 16 from the 25 to give a carry and a remainder of 9. The 9 is written down and the carry is added to the next digit column. In this column 7 plus 3 plus a carry gives a decimal 11, or B in hex.

```
                                          Carry
                                            ↓
 7A      0111  1010                     7  1  A16
+3F     +0011  1111                   + 3     F16
---     -----------                   -----------
 B9      1011  1001                     1110  2510
            B     9                     B16   916
         (a)                                (b)
```

FIGURE 1-11 Hexadecimal addition.

$$\begin{array}{rcl} 77_{16} & = & 119_{10} \\ -3B_{16} & = & -59_{10} \\ \hline 3C_{16} & & 60_{10} \end{array}$$

FIGURE 1-12 Hexadecimal subtraction.

You may use whichever method seems easier to you and gives you consistently right answers. If you are doing a great deal of hexadecimal arithmetic, you might buy an electronic calculator specifically designed to do decimal, binary, and hexadecimal arithmetic.

SUBTRACTION

Hexadecimal subtraction is similar to decimal subtraction except that when a borrow is needed, 16 is borrowed from the next most significant digit. Figure 1-12 shows an example of this. It may help you to follow the example if you do partial conversions to decimal in your head. For example, 7 plus a borrowed 16 is 23. Subtracting B or 11 leaves 12 or C in hexadecimal. Then 3 from the 6 left after a borrow leaves 3, so the result is 3CH.

BCD Addition and Subtraction

In systems where the final result of a calculation is to be displayed, such as a calculator, it may be easier to work with numbers in a BCD format. These codes, as shown in Table 1-1, represent each decimal digit, 0 through 9, by its 4-bit binary equivalent.

ADDITION

BCD can have no digit-word with a value greater than 9. Therefore, a carry must be generated if the result of a BCD addition is greater than 1001 or 9. Figure 1-13

```
                BCD
 35        0011 0101
+23       +0010 0011
----      ----------
 58        0101 1000
```

(a)

```
                 BCD
  7               0111
+ 5             + 0101
----            -------
 12              1100    INCORRECT BCD
                 + 0110  ADD 6
                -------
                0001 0010  CORRECT BCD 12
```

(b)

```
                 BCD
  9              1001
+ 8            + 1000
----           -------
 17             0001 0001  INCORRECT BCD
                0000 0110  ADD 6
                -------
                0001 0111  CORRECT BCD 17
```

(c)

FIGURE 1-13 BCD addition. (a) No correction needed. (b) Correction needed because of illegal BCD result. (c) Correction needed because of carry-out of BCD digit.

```
 17        0001 0111
 - 9       0000 1001
----       ---------
  8        0000 1110  ILLEGAL BCD
           -0110      SUBTRACT 6
           ---------
           0000 1000  CORRECT BCD
```

FIGURE 1-14 BCD subtraction.

shows three examples of BCD addition. The first, in Figure 1-13*a*, is very straightforward because the sum for each BCD digit is less than 9. The result is the same as it would be for adding standard binary.

For the second example, in Figure 1-13*b*, adding BCD 7 to BCD 5 produces 1100. This is a correct binary result of 12, but it is an illegal BCD code. To convert the result to BCD format, a correction factor of 6 is added. The result of adding 6 is 0001 0010, which is the legal BCD code for 12.

Figure 1-13*c* shows another case where a correction factor must be added. The initial addition of 9 and 8 produces 0001 0001. Even though the lower four digits are less than 9, this is an incorrect BCD result because a carry out of bit 3 of the BCD digit-word was produced. This carry out of bit 3 is often called an *auxiliary carry*. Adding the correction factor of 6 gives the correct BCD result of 0001 0111 or 17.

To summarize, a correction factor of 6 must be added if the result in the lower 4 bits is greater than 9 or if the initial addition produces a carry out of bit 3 of any BCD digit-word. This correction is sometimes called a *decimal adjust operation*.

The reason for the correction factor of 6 is that in BCD we want a carry into the next digit after 1001 or 9, but in binary a carry out of the lower 4 bits does not occur until after 1111 or 15. The difference between the two carry points is 6, so you have to add 6 to produce the desired carry if the result of an addition in any BCD digit is more than 1001.

SUBTRACTION

Figure 1-14 shows a subtraction, BCD 17 (0001 0111) minus BCD 9 (0000 1001). The initial result, 0000 1110, is not a legal BCD number. Whenever this occurs in BCD subtraction, 6 must be subtracted from the initial result to produce the correct BCD result. For the example shown in Figure 1-14, subtracting 6 gives a correct BCD result of 0000 1000 or 8.

The correction factor of 6 must be subtracted from any BCD digit-word if that digit-word is greater than 1001, or if a borrow from the next higher digit was required to do the subtraction.

BASIC DIGITAL DEVICES

Microcomputers such as those we discuss throughout this book often contain basic logic gates as "glue" between LSI (large-scale integration) devices. For troubleshooting these systems, it is important to be able to predict logic levels at any point directly from the schematic rather than having to work your way through a

truth table for each gate. This section should help refresh your memory of basic logic functions and help you remember how to quickly analyze logic gate circuits.

Inverting and Noninverting Buffers

Figure 1-15 shows the schematic symbols and truth tables for simple buffers and logic gates. The first thing to remember about these symbols is that the shape of the symbol indicates the logic function performed by the device. The second thing to remember about these symbols is that a bubble or no bubble indicates the *assertion* level for an input or output signal. Let's review how modern logic designers use these symbols.

The first symbol for a *buffer* in Figure 1-15*a* has no bubbles on the input or output. Therefore, the input is active high and the output is active high. We read this symbol as follows: If the input A is asserted high, then the output Y will be asserted high. The rest of the truth table is covered by the assumption that if the A input is not asserted high, then the Y output will not be asserted high.

The next two symbols for a buffer each contain a bubble. The bubble on the output of the first of these indicates that the output is active low. The input has no bubble, so it is active high. You can read the function of the device directly from the schematic symbol as follows. If the A input is asserted high, then the Y output will be asserted low. This device simply changes the assertion level of a signal. The output Y will always have a logic state which is the complement or inverse of that on the input, so the device is usually referred to as an *inverter*.

The second schematic symbol for an inverter in Figure 1-15*a* has the bubble on the input. We draw the symbol this way when we want to indicate that we are using the device to change an asserted-low signal to an asserted-high signal. For example, if we pass the signal $\overline{CS}$ through this device, it becomes CS. The symbol tells you directly that if the input is asserted low, then the output will be asserted high. Now let's review how you express the functions of logic gates using this approach.

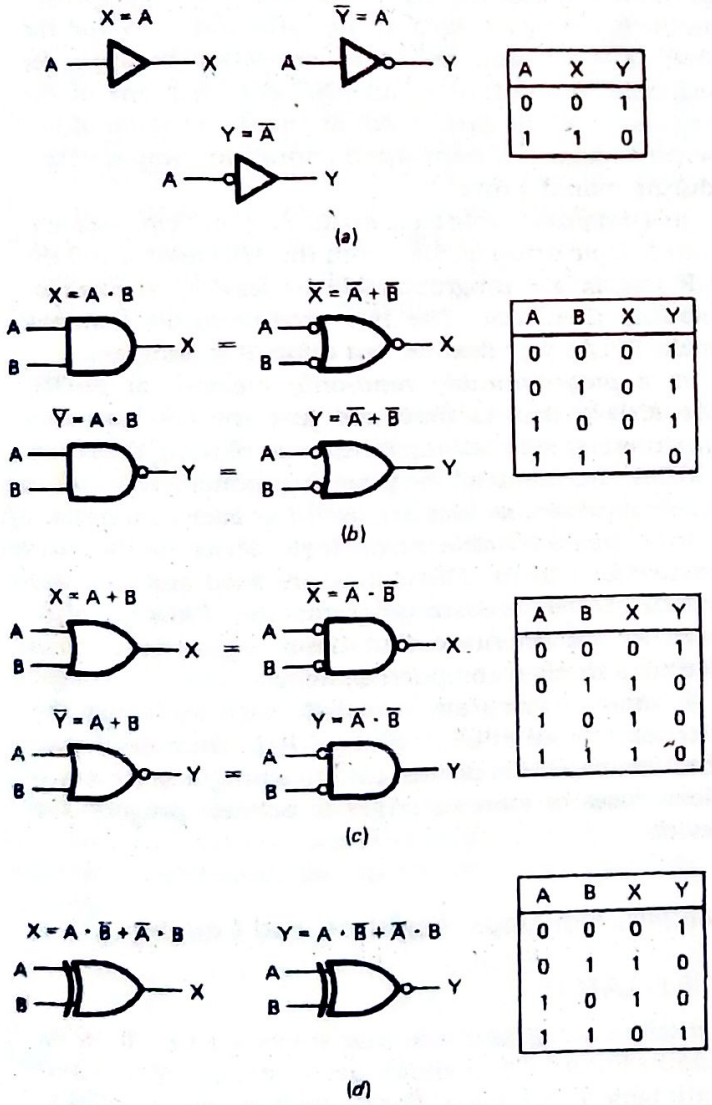


FIGURE 1-15 Buffers and logic gates. (*a*) Buffers. (*b*) AND-NAND. (*c*) OR-NOR. (*d*) Exclusive OR.

Logic Gates

Figure 1-15*b* shows the symbols and truth tables for simple logic gates. A symbol with a flat back and a round front indicates that the device performs the logical *AND* function. This means that the output will be asserted if the A input is asserted *and* the B input is asserted. Again, bubbles or no bubbles are used to indicate the assertion level of each input and output. The first AND symbol in Figure 1-15*b* has no bubbles, so the inputs and the output are active high. The output then will be asserted high if the A input is asserted high *and* the B input is asserted high. The bubble on the output of the second AND symbol in Figure 1-15*b* indicates that this device, commonly called a *NAND* gate, has an active low output. If the A input is asserted high *and* the B input is asserted high, then the Y output will be asserted low. Look at the truth table in Figure 1-15*b* to see if you agree with this.

Figure 1-15*c* shows the other two possible cases for the AND symbol. The first of these has bubbles on the inputs and on the output. If you see this symbol in a schematic, you should immediately see that the output will be asserted low if the A input is asserted low *and* the B input is asserted low. The second AND symbol in Figure 1-15*c* has no bubble on the output, so the output will be asserted high if the A *and* B inputs are both asserted low.

A logic symbol with a curved back indicates that the output of the device will be asserted if the A input is asserted *or* the B input of the device is asserted. Again, bubbles or no bubbles are used to indicate the assertion level for inputs and outputs. Note in Figure 1-15*b* and *c* that each of the AND symbol forms has an equivalent OR symbol form. An AND symbol with active high inputs and an active high output, for example, represents the same device (a 74LS08 perhaps) as an OR symbol with active low inputs and an active low output. Use the truth table in Figure 1-15*b* to convince yourself of this. The bubbled-OR representation tells you that if one input is asserted low, the output will be low, regardless of the state of the other input. As we will show later in this chapter, this is often a useful way to think of the operation of an AND gate.

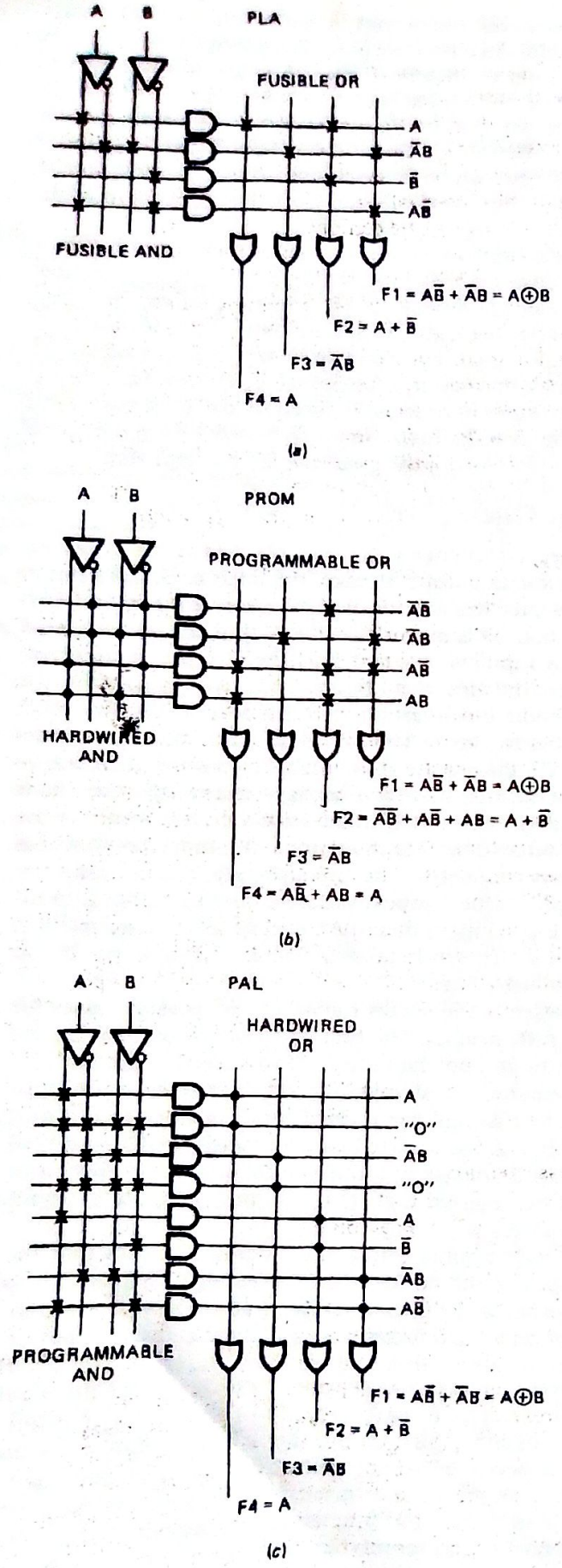


FIGURE 1-16 FPLA, PROM, and PAL programmed to implement some simple logic functions. (*a*) FPLA. (*b*) PROM. (*c*) PAL.

Figure 1-15*d* shows the symbol and truth table for an *exclusive* OR gate and for an *exclusive* NOR gate. The output of an exclusive OR gate will be high if the logic levels on the two inputs are different. The output of an exclusive NOR gate will be high if the logic levels on the two inputs are the same.

You need to be familiar with all these symbols, because most logic designers will use the symbol that best describes the function they want a device to perform in a particular circuit.

Programmable Logic Devices

Instead of using discrete gates, modern microcomputer systems usually use *programmable logic devices* such as PLAs, PROMs, or PALs to implement the "glue" logic between LSI devices. To refresh your memory, Figure 1-16 shows the internal structure of each of these devices. As you can see, they all consist of a programmable AND-OR matrix, so they can easily implement any sum-of-products logic expression. Each AND gate in these figures has up to four inputs, but to simplify the drawing only a single input line is shown. Likewise, the OR gates have several inputs, but are shown with a single input line to simplify the drawing. These devices are programmed by blowing out fuses, which are represented in the figure by Xs. An X in the figure indicates that the fuse is intact and makes a connection between, for example, the output of an AND gate and one of the inputs of an OR gate. A dot at the intersection of two wires indicates a hard-wired connection implemented during manufacture.

In a *programmable logic array* (*PLA*) or *field programmable logic array* (*FPLA*), both the AND matrix and the OR matrix are programmable by leaving in fuses or blowing them out. The two programmable matrixes make FPLAs very flexible, but difficult to program.

In a *programmable read-only memory* or *PROM*, the AND matrix is fixed and just the OR matrix is programmable by leaving in fuses or blowing them out. PROMs implement all the possible product terms for the input variables, so they are useful as code converters.

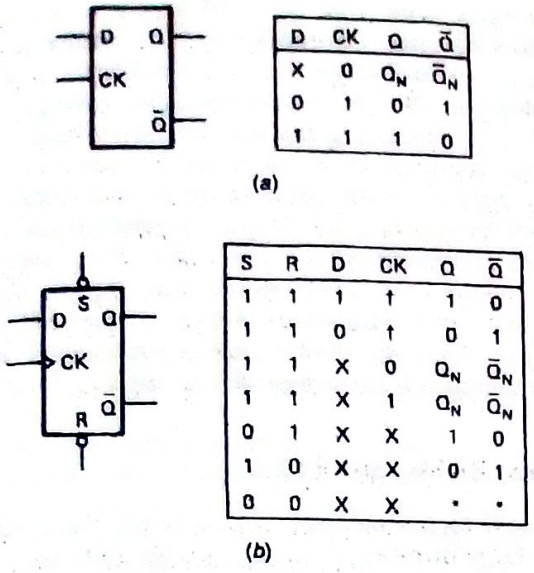
In a *programmable array logic* device or *PAL*, the connections in the OR matrix are fixed and the AND matrix connections are programmable. PALs are often used to implement combinational logic and address decoders in microcomputer systems.

A computer program is usually used to develop the fuse map for an FPLA, PROM, or PAL. Once developed, the fuse-map file is downloaded to a programmer which blows fuses or stores charges to actually program the device.

Latches, Flip-Flops, Registers, and Counters

THE D LATCH

A *latch* is a digital device that stores a 1 or a 0 on its output. Figure 1-17*a* shows the schematic symbol and truth table for a D latch. The device functions as follows. If the *enable* input CK is low, the logic level present on the D input will have no effect on the Q and $\bar{Q}$ outputs.



D	CK	Q	$\bar{Q}$
X	0	Q_N	$\bar{Q}_N$
0	1	0	1
1	1	1	0

S	R	D	CK	Q	$\bar{Q}$
1	1	1	↑	1	0
1	1	0	↑	0	1
1	1	X	0	Q_N	$\bar{Q}_N$
1	1	X	1	Q_N	$\bar{Q}_N$
0	1	X	X	1	0
1	0	X	X	0	1
0	0	X	X	*	*

FIGURE 1-17 Latches and flip-flops. (a) D latch. (*b*) D flip-flop.

This is indicated in the truth table by an X in the D column. If the enable input is high, a high or a low on the D input will be passed to the Q output. In other words, the Q output will follow the D input as long as the enable input is high. The $\bar{Q}$ output will contain the complement of the logic state on Q. When the enable input is made low again, the state on Q at that time will be latched there. Any changes on D will have no effect on Q until the enable input is made high again. When the enable input goes low, then, the state present on D just before the enable goes low will be stored on the Q output. Keep this operation in mind as you read about the D flip-flop in the next section.

THE D FLIP-FLOP

Figure 1-17*b* shows the schematic symbol and the truth table for a typical D flip-flop. The small triangle next to the CK input of this device tells you that the Q and $\bar{Q}$ outputs are updated when a rising signal edge is applied to the CK input. The up arrows in the clock column of the truth table also indicate that a 1 or 0 on the D input will be copied to the Q output when the clock input goes from low to high. In other words, the D flip-flop takes a snapshot of whatever state is on the D input when the clock goes high, and displays the "photo" on the Q output. If the clock input is low, a change on D will have no effect on the output. Likewise, if the clock input is high, a change on D will have no effect on the Q output. Contrast this operation with that of the D latch to make sure you understand the difference between the two devices.

The D flip-flop in Figure 1-17*b* also has direct *set* (S) and *reset* (R) inputs. A flip-flop is considered *set* if its Q output is a 1. It is *reset* if its Q output is a 0. The bubbles on the set and reset inputs tell you that these inputs are active low. The truth table for the D flip-flop in Figure 1-17*b* indicates that the set and reset inputs are *asynchronous*. This means that if the set input is asserted low, the output will be set, regardless of the states on the D and the clock inputs. Likewise, if the reset input is asserted low, the Q output will be reset, regardless of the state of the D and clock inputs. The Xs in the D and CK columns of the truth table remind you that these inputs are "don't cares" if set or reset is asserted. The condition indicated by the asterisks (*) is a nonstable condition; that is, it will not persist when reset or clear inputs return to their inactive (high) level.

REGISTERS

Flip-flops can be used individually or in groups to store binary data. A *register* is a group of D flip-flops connected in parallel, as shown in Figure 1-18*a*. A binary word applied to the data inputs of this register will be transferred to the Q outputs when the clock input is made high. The binary word will remain stored on the Q outputs until a new binary word is applied to the D inputs and a low-to-high signal is applied to the clock input. Other circuitry can read the stored binary word from the Q outputs at any time without changing its value.

If the Q output of each flip-flop in the register is connected to the D input of the next as shown in Figure 1-18*b*, then the register will function as a *shift register*. A 1 applied to the first D input will be shifted to the first Q output by a clock pulse. The next clock pulse will shift this 1 to the output of the second flip-flop. Each additional clock pulse will shift the 1 to the next flip-flop in the register. Some shift registers allow you to load a binary word into the register and shift the loaded word left or right when the register is clocked. As we will show later, the ability to shift binary numbers is very useful.

COUNTERS

Flip-flops can also be connected to make devices whose outputs step through a binary or other count sequence

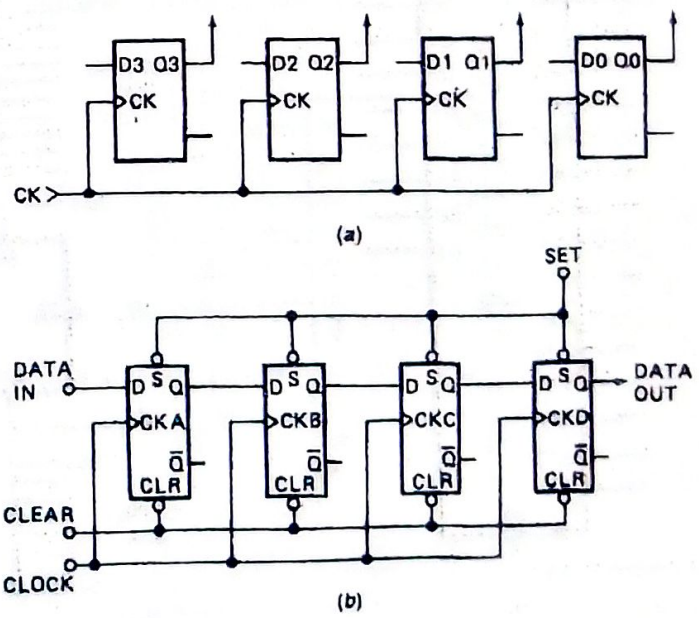
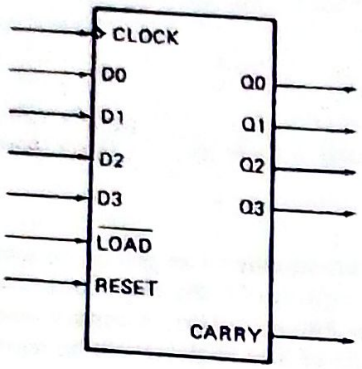


FIGURE 1-18 Registers. (a) Simple data storage. (*b*) Shift register.



(a)

Q3	Q2	Q1	Q0
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

(b)

FIGURE 1-19 Four-bit, presettable binary counter. (a) Schematic symbol. (b) Count sequence.

when they are clocked. Figure 1-19*a* shows a schematic symbol and count sequence for a presettable 4-bit binary counter. The main point we want to review here is how a presettable counter functions, so there is no need to go into the internal circuitry of the device. If the reset input is asserted, the Q outputs will all be made 0's. After the reset signal is unasserted, each clock pulse will cause the binary count on the outputs to be incremented by 1. As shown in Figure 1-19*b*, the count sequence will go from 0000 to 1111. If the outputs are at 1111, then the next clock pulse will cause the outputs to "roll over" to 0000 and a carry pulse to be sent out the carry output.

This carry pulse can be used as the clock input for another counter. Counters can be cascaded to produce as large a count sequence as is needed for a particular application. The maximum count for a binary counter is $2^N - 1$, where N is the number of flip-flops.

Now, suppose that we want the counter to start counting from some number other than 0000. We can do this by applying the desired number to the four data inputs and asserting the load input. For example, if we apply a binary 6, 0110, to the data inputs and assert the load input, this value will be transferred to the Q outputs. After the load signal is unasserted, the next clock signal will increment the Q outputs to 0111 or 7.

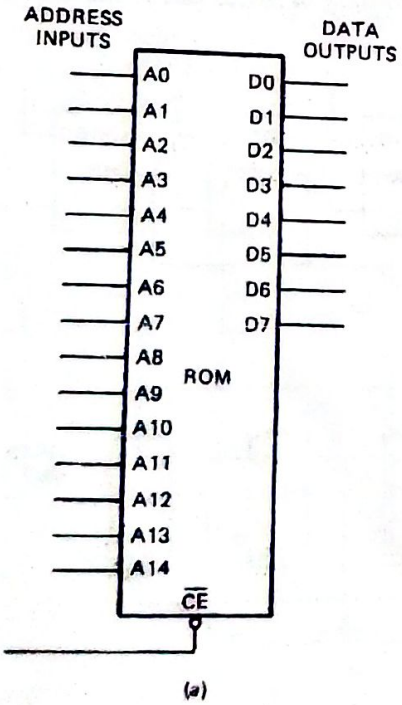
ROMs, RAMs, and Buses

The next topics we need to review are the devices that store large numbers of binary words and how several of these devices can be connected on common data lines.

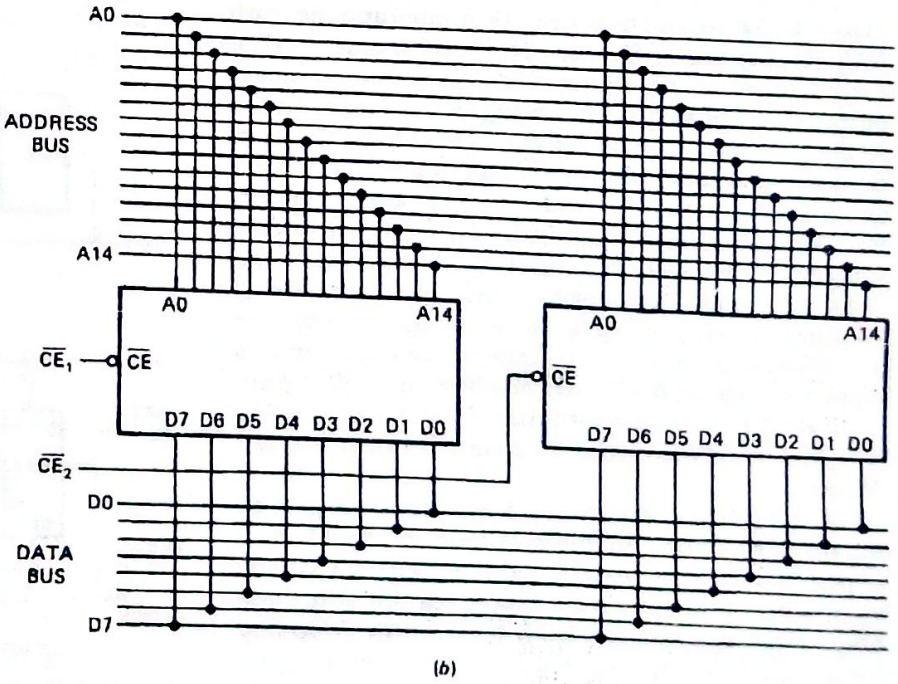
ROMS

The term *ROM* stands for *read-only memory*. There are several types of ROM that can be written to, read, erased, and written to with new data, but the main feature of ROMs is that they are *nonvolatile*. This means that the information stored in them is not lost when the power is removed from them.

Figure 1-20*a* shows the schematic symbol of a common ROM. As indicated by the eight *data* outputs, D0 to D7, this ROM stores 8-bit data words. The data outputs are *three-state* outputs. This means that each output can be at a logic low state, a logic high state, or a high-impedance floating state. In the high-impedance state an output is essentially disconnected from anything connected to it. If the $\overline{CE}$ input of the ROM is not asserted, then all the outputs will be in the high-



(a)



(b)

FIGURE 1-20 ROMs. (a) Schematic symbol. (b) Connection in parallel.

impedance state. Most ROMs also switch to a lower-power-consumption standby mode if $\overline{CE}$ is not asserted. If the $\overline{CE}$ input is asserted, the device will be powered up, and the output buffers will be enabled. Therefore, the outputs will be at a normal logic low or logic high state. If you don't happen to remember, you will soon see why this is important.

You can think of the binary words stored in the ROM as being in a long, numbered list. The number that identifies the location of each stored word in the list is called its *address*. You can tell the number of binary words stored in the ROM by the number of address inputs. The number of words is equal to 2^N, where N is the number of address lines. The device in Figure 1-20*a* has 15 address lines, A0 to A14, so the number of words is 2^{15} or 32,768. In a data sheet this device would be referred to as a 32K × 8 ROM. This means it has 32K addresses with 8 bits per address.

In order to get a particular word onto the outputs of the ROM, you have to do two things. You have to apply the address of that word to the address inputs, A0 to A14, and you have to assert the $\overline{CE}$ input to power up the device and to enable the three-state outputs.

Now, let's see why we want three-state outputs on this ROM. Suppose that we want to store more than 32K data words. We can do this by connecting two or more ROMs in parallel, as shown in Figure 1-20*b*. The address lines connect to each device in parallel, so we can address one of the 32,768 words in each. A set of parallel lines used to send addresses or data to several devices in this way is called a *bus*. The data outputs of the ROMs are likewise connected in parallel so that any one of the ROMs can output data on the common data bus. If these ROMs had standard two-state outputs, a serious problem would occur when both ROMs tried to output data words on the bus. The resulting argument between data outputs would probably destroy some of the outputs and give meaningless information on the data bus. Since the ROMs have three-state outputs, however, we can use external circuitry to make sure that only one ROM at a time has its outputs enabled. The very important principle here is that whenever several outputs are connected on a bus, the outputs should all be three-state, and only one set of outputs should be enabled at a time.

At the beginning of this section we mentioned that some ROMs can be erased and rewritten or reprogrammed with new data. Here's a summary of the different types of ROMs.

Mask-programmed ROM—Programmed during manufacture; cannot be altered.

PROM—User programs by blowing fuses; cannot be altered except to blow additional fuses.

EPROM—Electrically programmable by user; erased by shining ultraviolet light on quartz window in package.

EEPROM—Electrically programmable by user; erased with electrical signals, so it can be reprogrammed in circuit.

Flash EPROM—Electrically programmable by user; erased electrically, so it can be reprogrammed in circuit.

STATIC AND DYNAMIC RAMS

The name RAM stands for *random-access memory*, but since ROMs are also random access, the name probably should be *read-write memory*. RAMs are also used to store binary words. A *static RAM* is essentially a matrix of flip-flops. Therefore, we can write a new data word in a RAM location at any time by applying the word to the flip-flop data inputs and clocking the flip-flops. The stored data word will remain on the flip-flop outputs as long as the power is left on. This type of memory is *volatile* because data is lost when the power is turned off.

Figure 1-21 shows the schematic symbol for a common RAM. This RAM has 12 address lines, A0 to A11, so it stores 2^{12} (4096) binary words. The eight data lines tell you that the RAM stores 8-bit words. When we are reading a word from the RAM, these lines function as outputs. When we are writing a word to the RAM, these lines function as inputs. The *chip enable* input, $\overline{CE}$, is used to enable the device for a read or for a write. The R/$\overline{W}$ input will be asserted high if we want to read from the RAM or asserted low if we want to write a word to the RAM. Here's how all these lines work for reading from and writing to the device.

To write to the RAM, we apply the desired address to the address inputs, assert the $\overline{CE}$ input low to turn on the device, and assert the R/$\overline{W}$ input low to tell the RAM we want to write to it. We then apply the data word we want to store to the data lines of the RAM for a specified time. To read a word from the RAM, we address the desired word, assert $\overline{CE}$ low to turn on the device, and assert R/$\overline{W}$ high to tell the RAM we want to read from it. For a read operation the output buffers on the data lines will be enabled and the addressed data word will be present on the outputs.

The static RAMs we have just reviewed store binary words in a matrix of flip-flops. In *dynamic RAMs* (*DRAMs*), binary 1's and 0's are stored as an electric charge or no charge on a tiny capacitor. Since these tiny capacitors take up less space on a chip than a flip-flop

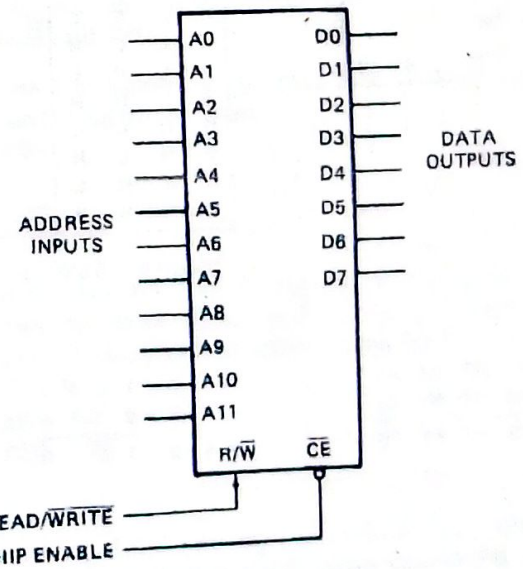


FIGURE 1-21 RAM schematic symbol.

would, a dynamic RAM chip can store many more bits than the same size static RAM chip. The disadvantage of dynamic RAMs is that the charge leaks off the tiny capacitors. The logic state stored in each capacitor must be *refreshed* every 2 milliseconds (ms) or so. A device called a *dynamic RAM refresh controller* can be used to refresh a large number of dynamic RAMs in a system. Some newer dynamic RAM devices contain built-in refresh circuitry, so they appear static to external circuitry.

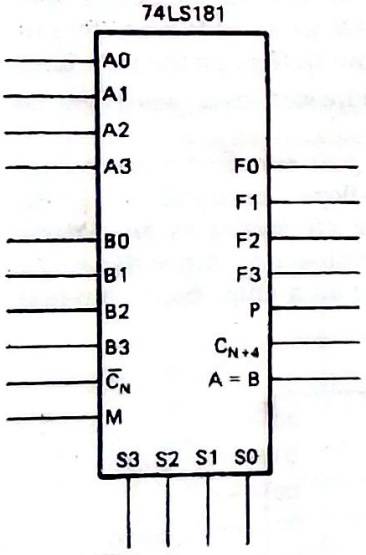
Arithmetic Logic Units

An *arithmetic logic unit*, or *ALU*, is a device that can AND, OR, add, subtract, and perform a variety of other operations on binary words. Figure 1-22*a* shows a block diagram for the 74LS181, which is a 4-bit ALU. This device can perform any one of 16 logic functions or any one of 16 arithmetic functions on two 4-bit binary words. The function performed on the two words is determined by the logic level applied to the mode input M and by the 4-bit binary code applied to the select inputs S0 to S3.

Figure 1-22*b* shows the truth table for the 74LS181. In this truth table, A represents the 4-bit binary word applied to the A0 to A3 inputs, and B represents the 4-bit binary word applied to the B0 to B3 inputs. F represents the 4-bit binary word that will be produced on the F0 to F3 outputs. If the mode input M is high, the device will perform one of 16 logic functions on the two words applied to the A and B inputs. For example, if M is high and we make S3 high, S2 low, S1 high, and S0 high, the 4-bit word on the A inputs will be ANDed with the 4-bit word on the B inputs. The result of this ANDing will appear on the F outputs. Each bit of the A word is ANDed with the corresponding bit of the B word to produce the result on F. Figure 1-22*c* shows an example of ANDing two words with this device. As you can see in this example, an output bit is high only if the corresponding bit is high in both the A word and the B word.

For another example of the operation of the 74LS181, suppose that the M input is high, S3 is high, S2 is high, S1 is high, and S0 is low. According to the truth table, the device will now OR each bit in the A word with the corresponding bit in the B word and give the result on the corresponding F output. Figure 1-22*c* shows the result that will be produced by ORing two 4-bit words. Figure 1-22*c* also shows for your reference the result that would be produced by exclusive ORing these two 4-bit words together.

If the M input of the 74LS181 is low, then the device will perform one of 16 arithmetic functions on the A and B words. Again, the result of the operation will be put on the F outputs. Several 74LS181s can be cascaded to operate on words longer than 4 bits. The ripple-carry input, $\overline{C}_N$, allows a carry from an operation on previous words to be included in the current operation. If the $\overline{C}_N$



(*a*)

SELECTION				ACTIVE–HIGH DATA		
				M = H	M = L; ARITHMETIC OPERATIONS	
S3	S2	S1	S0	LOGIC FUNCTIONS	$\overline{C}_N$ = H (NO CARRY)	$\overline{C}_N$ = L (WITH CARRY)
L	L	L	L	$F = \overline{A}$	F = A	F = A PLUS 1
L	L	L	H	$F = \overline{A + B}$	F = A + B	F = (A + B) PLUS 1
L	L	H	L	$F = \overline{A}B$	$F = A + \overline{B}$	F = (A + $\overline{B}$) PLUS 1
L	L	H	H	F = 0	F = MINUS 1 (2's COMPL)	F = 0
L	H	L	L	$F = \overline{AB}$	F = A PLUS A$\overline{B}$	F = A PLUS A$\overline{B}$ PLUS 1
L	H	L	H	$F = \overline{B}$	F = (A + B) PLUS A$\overline{B}$	F = (A + B) PLUS A$\overline{B}$ PLUS 1
L	H	H	L	F = A⊕B	F = A MINUS B MINUS 1	F = A MINUS B
L	H	H	H	F = A$\overline{B}$	F = A$\overline{B}$ MINUS 1	F = A$\overline{B}$
H	L	L	L	$F = \overline{A} + B$	F = A PLUS AB	F = A PLUS AB PLUS 1
H	L	L	H	F = A⊕B	F = A PLUS B	F = A PLUS B PLUS 1
H	L	H	L	F = B	F = (A + $\overline{B}$) PLUS AB	F = (A + $\overline{B}$) PLUS AB PLUS 1
H	L	H	H	F = AB	F = AB MINUS 1	F = AB
H	H	L	L	F = 1	F = A PLUS A*	F = A PLUS A PLUS 1
H	H	L	H	$F = A + \overline{B}$	F = (A + B) PLUS A	F = (A + B) PLUS A PLUS 1
H	H	H	L	F = A + B	F = (A + $\overline{B}$) PLUS A	F = (A + $\overline{B}$) PLUS A PLUS 1
H	H	H	H	F = A	F = A MINUS 1	F = A

*EACH BIT IS SHIFTED TO THE NEXT MORE SIGNIFICANT BIT POSITION

(*b*)

A =	A3	A2	A1	A0
B =	B3	B2	B1	B0
F =	F3	F2	F1	F0

A =	1	0	1	0
B =	0	1	1	0
F = A + B =	1	1	1	0

A =	1	0	1	0
B =	0	1	1	0
F = A · B =	0	0	1	0

A =	1	0	1	0
B =	0	1	1	0
F = A⊕B =	1	1	0	0

(*c*)

FIGURE 1-22 Arithmetic logic unit (ALU). (*a*) Schematic symbol. (*b*) Truth table. (*c*) Sample AND, OR, and XOR operations.

CHAPTER 2

Computers, Microcomputers, and Microprocessors—An Introduction

We live in a computer-oriented society, and we are constantly bombarded with a multitude of terms relating to computers. Before getting started with the main flow of the book, we will try to clarify some of these terms and to give an overview of computers and computer systems.

OBJECTIVES

At the conclusion of this chapter, you should be able to:

1. Define the terms *microcomputer, microprocessor, hardware, software, firmware, timesharing, multitasking, distributed processing,* and *multiprocessing.*
2. Describe how a microcomputer fetches and executes an instruction.
3. List the registers and other parts in the 8086/8088 execution unit and bus interface unit.
4. Describe the function of the 8086/8088 queue.
5. Demonstrate how the 8086/8088 calculates memory addresses.

TYPES OF COMPUTERS

Mainframes

Computers come in a wide variety of sizes and capabilities. The largest and most powerful are often called *mainframes*. Mainframe computers may fill an entire room. They are designed to work at very high speeds with large data words, typically 64 bits or greater, and they have massive amounts of memory. Computers of this type are used for military defense control, for business data processing (in an insurance company, for example), and for creating computer graphics displays for science fiction movies. Examples of this type of computer are the IBM 4381, the Honeywell DPS8, and the Cray Y-MP/832. The fastest and most powerful mainframes are called *supercomputers*. Figure 2-1*a*, p. 20, shows a photograph of a Cray Y-MP/832 supercomputer, which contains eight central processors and 32 million 64-bit words of memory.

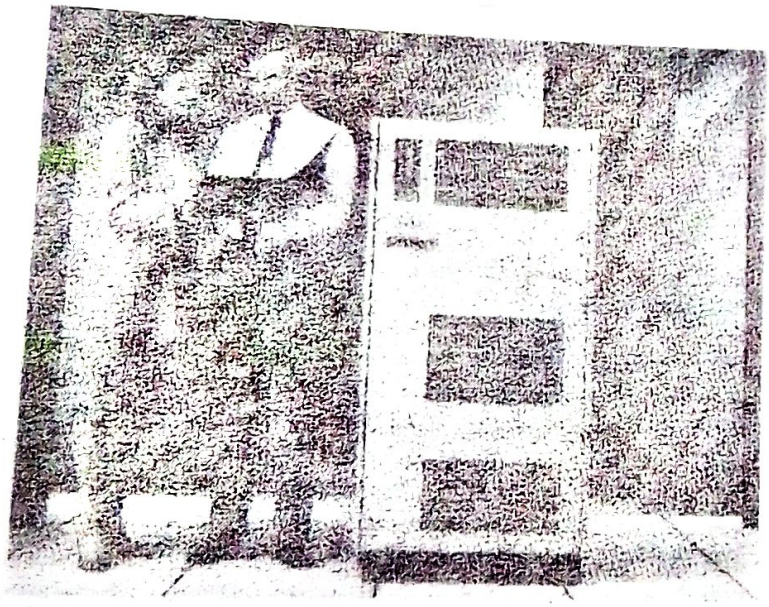
Minicomputers

Scaled-down versions of mainframe computers are often called *minicomputers*. The main unit of a minicomputer usually fits in a single rack or box. A minicomputer runs more slowly, works directly with smaller data words (often 32-bit words), and does not have as much memory as a mainframe. Computers of this type are used for business data processing, industrial control (for an oil refinery, for example), and scientific research. Examples of this type of computer are the Digital Equipment Corporation VAX 6360 and the Data General MV/8000II. Figure 2-1*b* shows a photograph of a Digital Equipment Corporation's VAX 6360 minicomputer.

Microcomputers

As the name implies, *microcomputers* are small computers. They range from small controllers that work directly with 4-bit words and can address a few thousand bytes of memory to larger units that work directly with 32-bit words and can address billions of bytes of memory. Some of the more powerful microcomputers have all or most of the features of earlier minicomputers. Therefore, it has become very hard to draw a sharp line between these two types. One distinguishing feature of a microcomputer is that the CPU is usually a single integrated circuit called a *microprocessor*. Older books often used the terms *microprocessor* and *microcomputer* interchangeably, but actually the microprocessor is the CPU to which you add ROM, RAM, and ports to make a microcomputer. A later section in this chapter discusses the evolution of different types of microprocessors. Microcomputers are used in everything from smart sewing machines to computer-aided design systems. Examples of microcomputers are the Intel 8051 single-chip controller; the SDK-86, a single-board computer design kit; the IBM Personal Computer (PC); and the Apple Macintosh computer. The Intel 8051 microcontroller is contained in a single 40-pin chip. Figure 2-2*a*, p. 21, shows the SDK-86 board, and Figure 2-2*b* shows the Compaq 386/25 system.

(a)



(b)

FIGURE 2-1 (a) Photograph of Cray Y-MP/832 computer. (*Courtesy Cray Research, Inc., and photographer, Paul Shambroom.*) (b) Photograph of VAX 6360 minicomputer. (*Courtesy Digital Equipment Corp.*)

HOW COMPUTERS AND MICROCOMPUTERS ARE USED—AN EXAMPLE

The following sections are intended to give you an overview of how computers are interfaced with users to do useful work. These sections should help you understand many of the features designed into current microprocessors and where this book is heading.

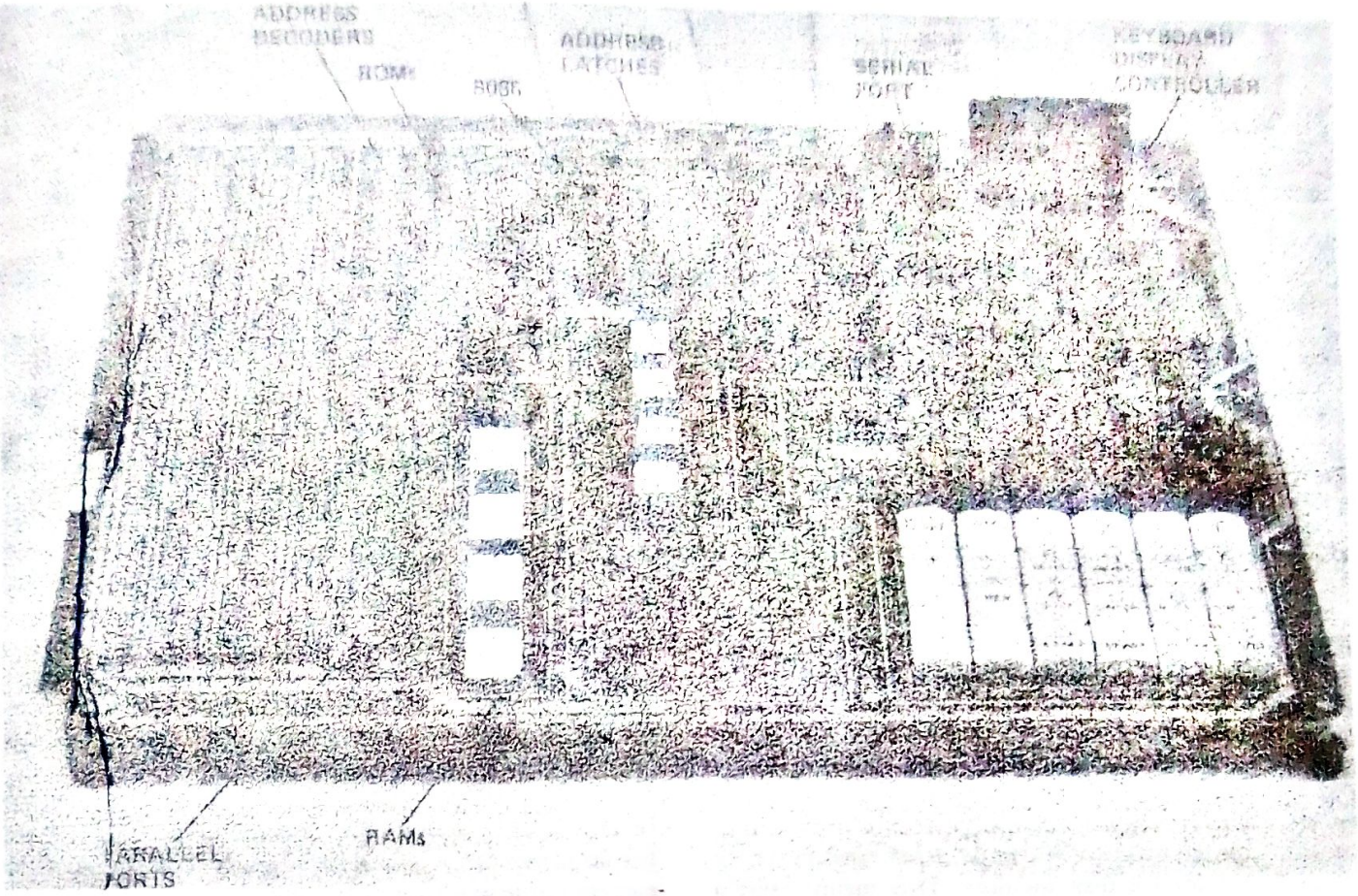
Computerizing an Electronics Factory—Problem

Now, suppose that we want to "computerize" an electronics company. By this we mean that we want to make computer use available to as many people in the company as possible as cheaply as possible. We want the engineers to have access to a computer which can help them design circuits. People in the drafting department should have access to a computer which can be used for computer-aided drafting. The accounting department should have access to a computer for doing all the financial bookkeeping. The warehouse should have access to a computer to help with inventory control. The manufacturing department should have access to a computer for controlling machines and testing finished products. The president, vice presidents, and supervisors should have access to a computer to help them with long-range planning. Secretaries should have access to a computer for word processing. Salespeople should have access to a computer to help them keep track of current pricing, product availability, and commissions. There are several ways to provide all the needed computer power. One solution is to simply give everyone an individual personal computer. The problem with this approach is that it makes it difficult for different people to access commonly needed data. In the next sections we show you two ways to provide computer power and common data to many users.

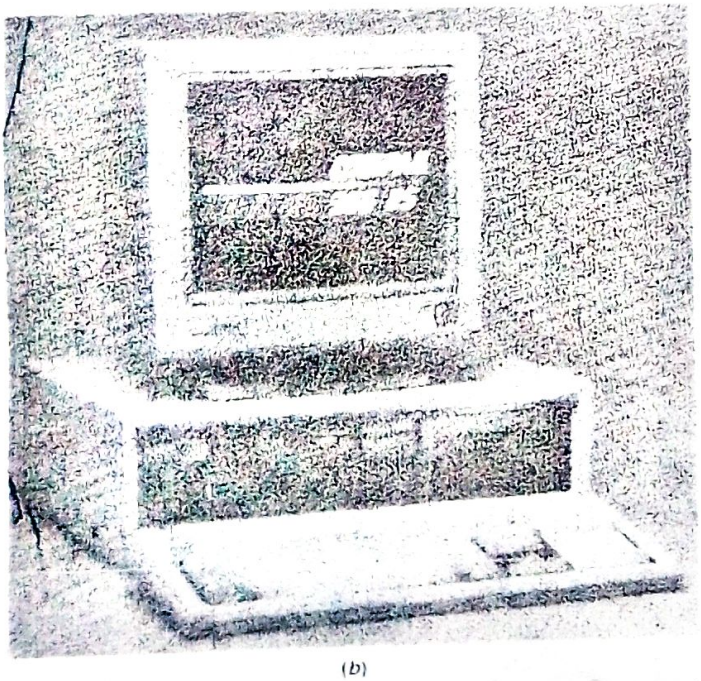
TIMESHARING AND MULTITASKING SYSTEMS

One common method of providing computer access is a *timesharing* system such as shown in Figure 2-3, p. 22. Several video terminals are connected to the computer through direct wires or through telephone lines. The terminal can be on the user's desk or even in the user's home. The rate at which a user usually enters data is very slow compared with the rate at which a computer can process the data. Therefore, the computer can serve many users by dividing its time among them in small increments. In other words, the computer works on user 1's program for perhaps 20 milliseconds (ms), then works on user 2's program for 20 ms, then works on user 3's program for 20 ms, and so on, until all the users have had a turn. In a few milliseconds the computer will get back to user 1 again and repeat the cycle. To each user it will appear as if he or she has exclusive use of the computer because the computer processes data as fast as the user enters it. A timesharing system such as this allows several users to interact with the computer at the same time. Each user can get information from or store information in the large memory attached to the computer. Each user can have an inexpensive printer attached to the terminal or can direct program or data output to a high-speed printer attached directly to the computer.

An airline ticket reservation computer might use a timesharing system such as this to allow users from all over the country to access flight information and make reservations. A time-multiplexed or time-sliced system such as this can also allow a computer to control many machines or processes in a factory. A computer is much faster than the machines or processes. Therefore, it can



(a)



(b)

FIGURE 2-2 (a) Photograph of Intel SDK-86 board. (*Intel Corp.*) (b) Photograph of Compaq 386/25. (*Compaq Corp.*)

check and adjust many pressures, temperatures, motor speeds, etc., before it needs to get back and recheck the first one. A system such as this is often called a *multitasking system* because it appears to be doing many tasks at the same time.

Now let's take another look at our problem of computerizing the electronics company. We could put a powerful computer in some central location and run wires from it to video display terminals on users' desks. Each user could then run the program needed to do a particular task. The accountant could run a ledger program, the secretary could run a word processing program, etc. Each user could access the computer's large data memory. Incidentally, a large collection of data stored in a computer's memory is often referred to as a *data base*. For a small company a system such as this might be adequate. However, there are at least two potential problems.

The first potential problem is, "What happens if the computer is not working?" The answer to this question is that everything grinds to a halt. In a situation where people have become dependent on the computer, not much gets done until the computer is up and running again. The old saying about putting all your eggs in one basket comes to mind here.

The second potential problem of the simple timesharing system is saturation. As the number of users increases, the time it takes the computer to do each user's task increases also. Eventually the computer's response time to each user becomes unreasonably long. People get very upset about the time they have to wait.

DISTRIBUTED PROCESSING OR MULTIPROCESSING

A partial solution for the two potential problems of a simple timesharing system is to use a distributed

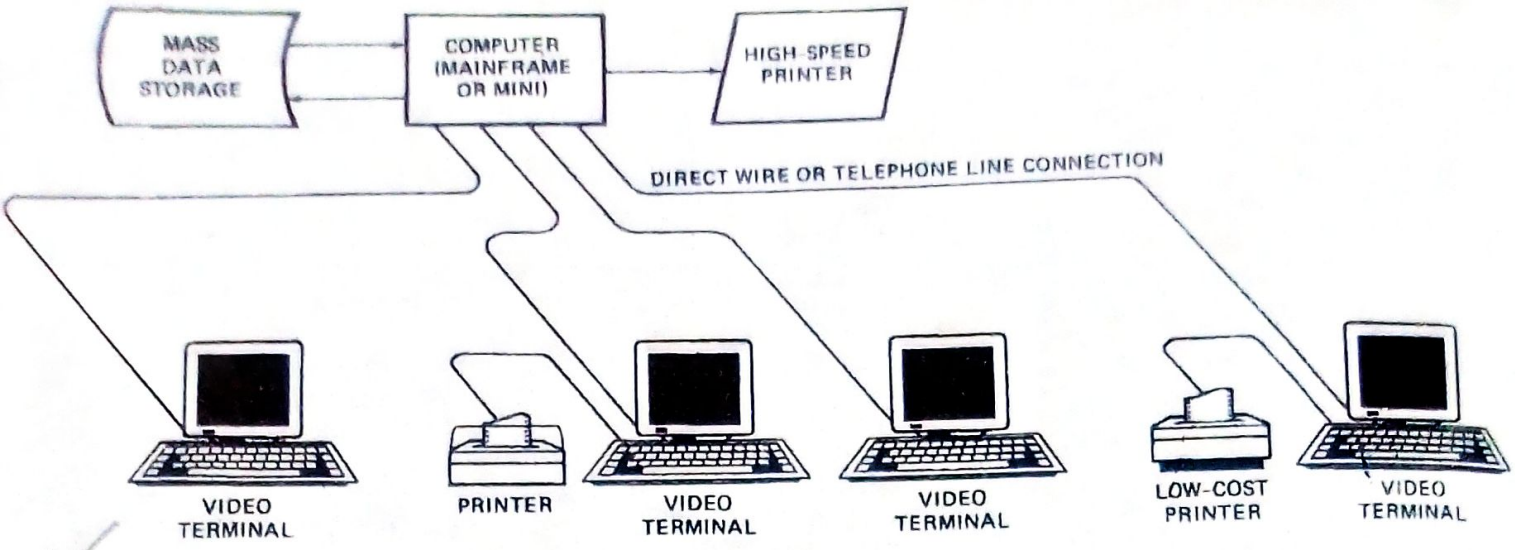


FIGURE 2-3 Block diagram of a computer timesharing system.

processing system. Figure 2-4 shows a block diagram for such a system. The system has a powerful central computer with a large memory and a high-speed printer, as does the simple timesharing system described previously. However, in this system each user has a microcomputer instead of simply a video display terminal. In other words, each user station is an independently functioning microcomputer with a CPU, ROM, RAM, and probably magnetic or optical disk memory. This means that a person can do many tasks locally on the microcomputer without having to use the large computer at all. Since the microcomputers are connected to the large computer through a network, however, a user can access the computing power, memory, or other resources of the large computer when needed.

Distributing the processing to multiple computers or processors in a system has several advantages. First, if the large computer goes down, the local microcomputers can continue working until they need to access the large computer for something. Second, the burden on the

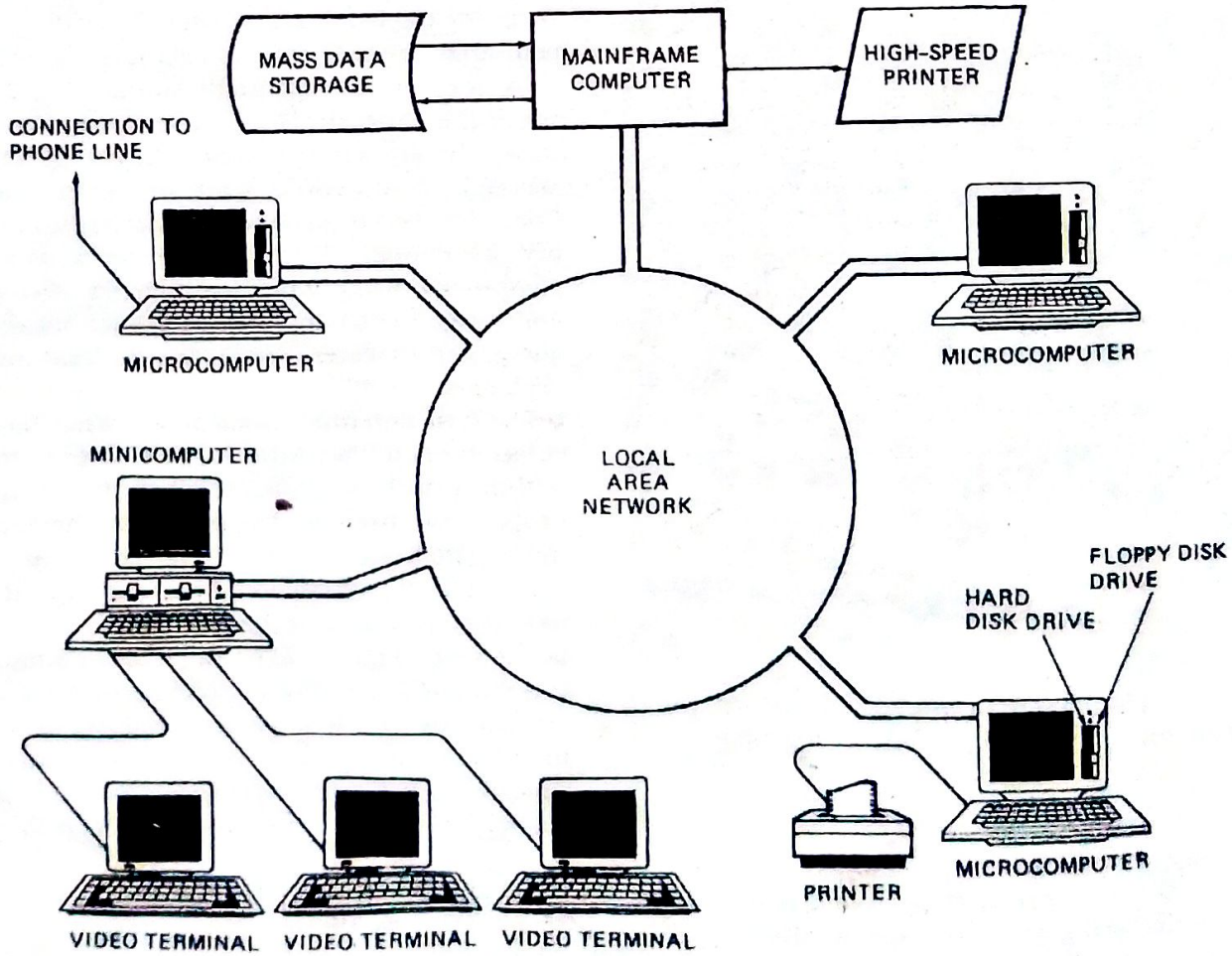


FIGURE 2-4 Block diagram of a distributed processing computer system.

large computer is reduced greatly, because much of the computing is done by the local microcomputers. Finally, the distributed processing approach allows the system designer to use a local microcomputer that is best suited to the task it has to do.

COMPUTERIZED ELECTRONICS COMPANY OVERVIEW

Distributed processing seems to be the best way to go about computerizing our electronics factory. Engineers can have personal computers or engineering workstations on their desks. With these they can use available programs to design and test circuits. They can access the large computer if they need data from its memory. Through the telephone lines, the engineer with a personal computer can access data in the memory of other computers all over the world. The drafting people can have personal computers for simple work, or large computer-aided design systems for more complex work. Completed work can be stored in the memory of the large computer. The production department can have networked computers to keep track of product flow and to control the machines which actually mount components on circuit boards, etc. The accounting department can use personal computers with spreadsheet programs to work with financial data kept in the memory of the large computer. The warehouse supervisor can likewise use a personal computer with an inventory program to keep personal records and those in the large computer's memory updated. Corporate officers can have personal computers tied into the network. They then can interact with any of the other systems on the network. Salespeople can have portable personal computers that they can carry with them in the field. They can communicate with the main computer over the telephone lines using a modem. Secretaries doing word processing can use individual word processing units or personal computers. Users can also send messages to one another over the network. The specifics of a computer system such as this will obviously depend on the needs of the individual company for which the system is designed.

SUMMARY AND DIRECTION FROM HERE

The main concepts that you should take with you from this section are timesharing or multitasking and distributed processing or multiprocessing. As you work your way through the rest of this book, keep an overview of the computerized electronics company in the back of your mind. The goal of this book is to teach you how the microcomputers and other parts of a system such as this work, how the parts are connected together, and how the system is programmed at different levels.

OVERVIEW OF MICROCOMPUTER STRUCTURE AND OPERATION

Figure 2-5 shows a block diagram for a simple microcomputer. The major parts are the *central processing unit* or CPU, *memory*, and the *input and output* circuitry or I/O. Connecting these parts are three sets of parallel lines called *buses*. The three buses are the *address bus*, the *data bus*, and the *control bus*. Let's take a brief look at each of these parts.

Memory

The memory section usually consists of a mixture of RAM and ROM. It may also have magnetic floppy disks, magnetic hard disks, or optical disks. Memory has two purposes. The first purpose is to store the binary codes for the sequences of instructions you want the computer to carry out. When you write a computer program, what you are really doing is writing a sequential list of instructions for the computer. The second purpose of the memory is to store the binary-coded data with which the computer is going to be working. This data might be the inventory records of a supermarket, for example.

Input/Output

The input/output or I/O section allows the computer to take in data from the outside world or send data to the outside world. Peripherals such as keyboards, video display terminals, printers, and modems are connected to the I/O section. These allow the user and the computer to communicate with each other. The actual physical devices used to interface the computer buses to external systems are often called *ports*. Ports in a computer function just as shipping ports do for a country. An *input port* allows data from a keyboard, an A/D converter, or some other source to be read into the computer under control of the CPU. An *output port* is used to send data from the computer to some peripheral, such as a video display terminal, a printer, or a D/A converter. Physically,

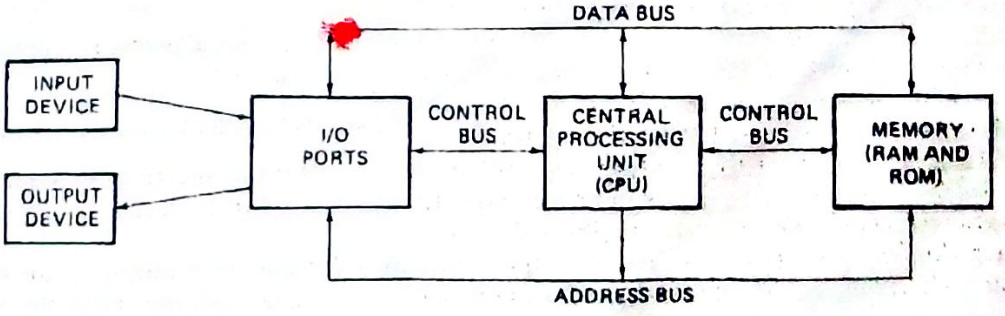


FIGURE 2-5 Block diagram of a simple microcomputer.

the simplest type of input or output port is just a set of parallel D flip-flops. If they are being used as an input port, the D inputs are connected to the external device, and the Q outputs are connected to the data bus which runs to the CPU. Data will then be transferred through the latches when they are enabled by a control signal from the CPU. In a system where they are being used as an output port, the D inputs of the latches are connected to the data bus, and the Q outputs are connected to some external device. Data sent out on the data bus by the CPU will be transferred to the external device when the latches are enabled by a control signal from the CPU.

Central Processing Unit

The central processing unit or CPU controls the operation of the computer. In a microcomputer the CPU is a microprocessor, as we discussed in an earlier section of the chapter. The CPU fetches binary-coded instructions from memory, decodes the instructions into a series of simple actions, and carries out these actions in a sequence of steps.

The CPU also contains an *address counter* or *instruction pointer* register, which holds the address of the next instruction or data item to be fetched from memory; general-purpose registers, which are used for temporary storage of binary data; and circuitry, which generates the control bus signals.

Address Bus

The address bus consists of 16, 20, 24, or 32 parallel signal lines. On these lines the CPU sends out the address of the memory location that is to be written to or read from. The number of memory locations that the CPU can address is determined by the number of address lines. If the CPU has N address lines, then it can directly address 2^N memory locations. For example, a CPU with 16 address lines can address 2^{16} or 65,536 memory locations, a CPU with 20 address lines can address 2^{20} or 1,048,576 locations, and a CPU with 24 address lines can address 2^{24} or 16,777,216 locations. When the CPU reads data from or writes data to a port, it sends the port address out on the address bus.

Data Bus

The data bus consists of 8, 16, or 32 parallel signal lines. As indicated by the double-ended arrows on the data bus line in Figure 2-5, the data bus lines are *bidirectional*. This means that the CPU can read data in from memory or from a port on these lines, or it can send data out to memory or to a port on these lines. Many devices in a system will have their outputs connected to the data bus, but only one device at a time will have its outputs enabled. Any device connected on the data bus must have *three-state outputs* so that its outputs can be disabled when it is not being used to put data on the bus.

Control Bus

The control bus consists of 4 to 10 parallel signal lines. The CPU sends out signals on the control bus to enable the outputs of addressed memory devices or port devices. Typical control bus signals are *Memory Read*, *Memory Write*, *I/O Read*, and *I/O Write*. To read a byte of data from a memory location, for example, the CPU sends out the memory address of the desired byte on the address bus and then sends out a Memory Read signal on the control bus. The Memory Read signal enables the addressed memory device to output a data word onto the data bus. The data word from memory travels along the data bus to the CPU.

Hardware, Software, and Firmware

When working around computers, you hear the terms hardware, software, and firmware almost constantly. *Hardware* is the name given to the physical devices and circuitry of the computer. *Software* refers to the programs written for the computer. *Firmware* is the term given to programs stored in ROMs or in other devices which permanently keep their stored information.

Summary of Important Points So Far

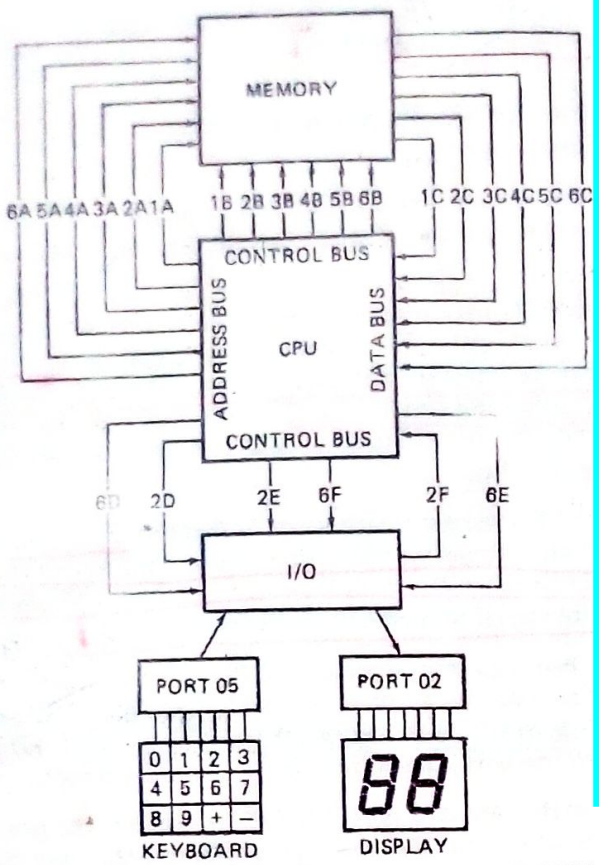
- A computer or microcomputer consists of memory, a CPU, and some input/output circuitry.
- These three parts are connected by the address bus, the data bus, and the control bus.
- The sequence of instructions or program for a computer is stored as binary numbers in successive memory locations.
- The CPU fetches an instruction from memory, decodes the instruction to determine what actions must be done for the instruction, and carries out these actions.

EXECUTION OF A THREE-INSTRUCTION PROGRAM

To give you a better idea of how the parts of a microcomputer function together, we will now describe the actions a simple microcomputer might go through to carry out (execute) a simple program. The three instructions of the program are

1. Input a value from a keyboard connected to the port at address 05H.
2. Add 7 to the value read in.
3. Output the result to a display connected to the port at address 02H.

Figure 2-6 shows in diagram form and sequential list form the actions that the computer will perform to execute these three instructions.



PROGRAM

1. INPUT A VALUE FROM PORT 05.
2. ADD 7 TO THIS VALUE.
3. OUTPUT THE RESULT TO PORT 02.

SEQUENCE

1A CPU SENDS OUT ADDRESS OF FIRST INSTRUCTION TO MEMORY.
1B CPU SENDS OUT MEMORY READ CONTROL SIGNAL TO ENABLE MEMORY.
1C INSTRUCTION BYTE SENT FROM MEMORY TO CPU ON DATA BUS.
2A ADDRESS NEXT MEMORY LOCATION TO GET REST OF INSTRUCTION.
2B SEND MEMORY READ CONTROL SIGNAL TO ENABLE MEMORY.
2C PORT ADDRESS BYTE SENT FROM MEMORY TO CPU ON DATA BUS.
2D CPU SENDS OUT PORT ADDRESS ON ADDRESS BUS.
2E CPU SENDS OUT INPUT READ CONTROL SIGNAL TO ENABLE PORT.
2F DATA FROM PORT SENT TO CPU ON DATA BUS.
3A CPU SENDS ADDRESS OF NEXT INSTRUCTION TO MEMORY.
3B CPU SENDS MEMORY READ CONTROL SIGNAL TO ENABLE MEMORY.
3C INSTRUCTION BYTE FROM MEMORY SENT TO CPU ON DATA BUS.
4A CPU SENDS NEXT ADDRESS TO MEMORY TO GET REST OF INSTRUCTION.
4B CPU SENDS MEMORY READ CONTROL SIGNAL TO ENABLE MEMORY.
4C NUMBER 07H SENT FROM MEMORY TO CPU ON DATA BUS.
5A CPU SENDS ADDRESS OF NEXT INSTRUCTION TO MEMORY.
5B CPU SENDS MEMORY READ CONTROL SIGNAL TO ENABLE MEMORY.
5C INSTRUCTION BYTE FROM MEMORY SENT TO CPU ON DATA BUS.
6A CPU SENDS OUT NEXT ADDRESS TO GET REST OF INSTRUCTION.
6B CPU SENDS OUT MEMORY READ CONTROL SIGNAL TO ENABLE MEMORY.
6C PORT ADDRESS BYTE SENT FROM MEMORY TO CPU ON DATA BUS.
6D CPU SENDS OUT PORT ADDRESS ON ADDRESS BUS.
6E CPU SENDS OUT DATA TO PORT ON DATA BUS.
6F CPU SENDS OUT OUTPUT WRITE SIGNAL TO ENABLE PORT.

(a)

(b)

MEMORY ADDRESS	CONTENTS (BINARY)	CONTENTS (HEX)	OPERATION
00100H	11100100	E4	INPUT FROM
00101H	00000101	05	PORT 05H
00102H	00000100	04	ADD
00103H	00000111	07	07H
00104H	11100110	E6	OUTPUT TO
00105H	00000010	02	PORT 02

FIGURE 2-6 (a) Execution of a three-step computer program. (b) Memory addresses and memory contents for a three-step program.

For this example, assume that the CPU fetches instructions and data from memory 1 byte at a time, as is done in the original IBM PC and its clones. Also assume that the binary codes for the instructions are in sequential memory locations starting at address 00100H. Figure 2-6*b* shows the actual binary codes that would be required in successive memory locations to execute this program on an IBM PC-type microcomputer.

The CPU needs an instruction before it can do anything, so its first action is to fetch an instruction byte from memory. To do this, the CPU sends out the address of the first instruction byte, in this case 00100H, to memory on the address bus. This action is represented by line 1A in Figure 2-6*a*. The CPU then sends out a Memory Read signal on the control bus (line 1B in the figure). The Memory Read signal enables the memory to output the addressed byte on the data bus. This action is represented by line 1C in the figure. The CPU reads in this first instruction byte (E4H) from the data bus and *decodes* it. By *decode* we mean that the CPU determines from the binary code read in what actions it is supposed to take. If the CPU is a microprocessor, it selects the sequence of microinstructions needed to carry out the instruction read from memory. For the example instruction here, the CPU determines that the code read in represents an Input instruction. From decoding this instruction byte, the CPU also determines that it needs more information before it can carry out the instruction. The additional information the CPU needs is the address of the port that the data is to be input from. This port address part of the instruction is stored in the next memory location after the code for the Input instruction.

To fetch this second byte of the instruction, the CPU sends out the next sequential address (00101H) to memory, as shown by line 2A in the figure. To enable the addressed memory device, the CPU also sends out another Memory Read signal on the control bus (line 2B). The memory then outputs the addressed byte on the data bus (line 2C). When the CPU has read in this second byte, 05H in this case, it has all the information it needs to execute the instruction.

To execute the Input instruction, the CPU sends out the port address (05H) on the address bus (line 2D) and sends out an I/O Read signal on the control bus (line 2E). The I/O Read signal enables the addressed port

device to put a byte of data on the data bus (line 2F). The CPU reads in the byte of data and stores it in an internal register. This completes the fetching and execution of the first instruction.

Having completed the first instruction, the CPU must now fetch its next instruction from memory. To do this, it sends out the next sequential address (00102H) on the address bus (line 3A) and sends out a Memory Read signal on the control bus (line 3B). The Memory Read signal enables the memory device to put the addressed byte (04H) on the data bus (line 3C). The CPU reads in this instruction byte from the data bus and decodes it. From this instruction byte the CPU determines that it is supposed to add some number to the number stored in the internal register. The CPU also determines from decoding this instruction byte that it must go to memory again to get the next byte of the instruction, which contains the number that it is supposed to add. To get the required byte, the CPU will send out the next sequential address (00103H) on the address bus (line 4A) and another Memory Read signal on the control bus (line 4B). The memory will then output the contents of the addressed byte (the number 07H) on the data bus (line 4C). When the CPU receives this number, it will add it to the contents of the internal register. The result of the addition will be left in the internal register. This completes the fetching and executing of the second instruction.

The CPU must now fetch the third instruction. To do this, it sends out the next sequential address (00104H) on the address bus (line 5A) and sends out a Memory Read signal on the control bus (line 5B). The memory then outputs the addressed byte (E6H) on the data bus (line 5C). From decoding this byte, the CPU determines that it is now supposed to do an Output operation to a port. The CPU also determines from decoding this byte that it must go to memory again to get the address of the output port. To do this, it sends out the next sequential address (00105H) on the address bus (line 6A), sends out a Memory Read signal on the control bus (line 6B), and reads in the byte (02H) put on the data bus by the memory (line 6C). The CPU now has all the information that it needs to execute the Output instruction.

To output a data byte to a port, the CPU first sends out the address of the desired port on the address bus (line 6D). Next it outputs the data byte from the internal register on the data bus (line 6E). The CPU then sends out an I/O Write signal on the control bus (line 6F). This signal enables the addressed output port device so that the data from the data bus lines can pass through it to the LED displays. When the CPU removes the I/O Write signal to proceed with the next instruction, the data will remain latched on the output pins of the port device. The data will remain latched on the port until the power is turned off or until a new data word is output to the port. This is important because it means that the computer does not have to keep outputting a value over and over in order for it to remain on the output.

All the steps described above may seem like a great deal of work just to input a value from a keyboard, add 7 to it, and output the result to a display. Even a simple microcomputer, however, can run through all these steps in a few microseconds.

Summary of Simple Microcomputer Bus Operation

1. A microcomputer fetches each program instruction in sequence, decodes the instruction, and executes it.
2. The CPU in a microcomputer fetches instructions or reads data from memory by sending out an address on the address bus and a Memory Read signal on the control bus. The memory outputs the addressed instruction or data word to the CPU on the data bus.
3. The CPU writes a data word to memory by sending out an address on the address bus, sending out the data word on the data bus, and sending a Memory Write signal to memory on the control bus.
4. To read data from a port, the CPU sends out the port address on the address bus and sends an I/O Read signal to the port device on the control bus. Data from the port comes into the CPU on the data bus.
5. To write data to a port, the CPU sends out the port address on the address bus, sends out the data to be written to the port on the data bus, and sends an I/O Write signal to the port device on the control bus.

MICROPROCESSOR EVOLUTION AND TYPES

As we told you in the preceding section, a microprocessor is used as the CPU in a microcomputer. There are now many different microprocessors available, so before we dig into the details of a specific device, we will give you a short microprocessor history lesson and an overview of the different types.

Microprocessor Evolution

A common way of categorizing microprocessors is by the number of bits that their ALU can work with at a time. In other words, a microprocessor with a 4-bit ALU will be referred to as a 4-bit microprocessor, regardless of the number of address lines or the number of data bus lines that it has. The first commercially available microprocessor was the Intel 4004, produced in 1971. It contained 2300 PMOS transistors. The 4004 was a 4-bit device intended to be used with some other devices in making a calculator. Some logic designers, however, saw that this device could be used to replace PC boards full of combinational and sequential logic devices. Also, the ability to change the function of a system by just changing the programming, rather than redesigning the hardware, is very appealing. It was these factors that pushed the evolution of microprocessors.

In 1972 Intel came out with the 8008, which was capable of working with 8-bit words. The 8008, however,

required 20 or more additional devices to form a functional CPU. In 1974 Intel announced the 8080, which had a much larger instruction set than the 8008 and required only two additional devices to form a functional CPU. Also, the 8080 used NMOS transistors, so it operated much faster than the 8008. The 8080 is referred to as a *second-generation microprocessor*.

Soon after Intel produced the 8080, Motorola came out with the MC6800, another 8-bit general-purpose CPU. The 6800 had the advantage that it required only a +5-V supply rather than the −5-V, +5-V, and +12-V supplies required by the 8080. For several years the 8080 and the 6800 were the top-selling 8-bit microprocessors. Some of their competitors were the MOS Technology 6502, used as the CPU in the Apple II microcomputer, and the Zilog Z80, used as the CPU in the Radio Shack TRS-80 microcomputer.

As designers found more and more applications for microprocessors, they pressured microprocessor manufacturers to develop devices with architectures and features optimized for doing certain types of tasks. In response to the expressed needs, microprocessors have evolved in three major directions during the last 15 years.

Dedicated or Embedded Controllers

One direction has been *dedicated or embedded controllers*. These devices are used to control "smart" machines, such as microwave ovens, clothes washers, sewing machines, auto ignition systems, and metal lathes. Texas Instruments has produced millions of their TMS-1000 family of 4-bit microprocessors for this type of application. In 1976 Intel introduced the 8048, which contains an 8-bit CPU, RAM, ROM, and some I/O ports all in one 40-pin package. Other manufacturers have followed with similar products. These devices are often referred to as *microcontrollers*. Some currently available devices in this category—the Intel 8051 and the Motorola MC6801, for example—contain programmable counters and a serial port (UART) as well as a CPU, ROM, RAM, and parallel I/O ports. A more recently introduced single-chip microcontroller, the Intel 8096, contains a 16-bit CPU, ROM, RAM, a UART, ports, timers, and a 10-bit analog-to-digital converter.

Bit-Slice Processors

A second direction of microprocessor evolution has been *bit-slice processors*. For some applications, general-purpose CPUs such as the 8080 and 6800 are not fast enough or do not have suitable instruction sets. For these applications, several manufacturers produce devices which can be used to build a custom CPU. An example is the Advanced Micro Devices 2900 family of devices. This family includes 4-bit ALUs, multiplexers, sequencers, and other parts needed for custom-building a CPU. The term *slice* comes from the fact that these parts can be connected in parallel to work with 8-bit words, 16-bit words, or 32-bit words. In other words, a designer can add as many slices as needed for a particular application. The designer not only custom-designs the hardware of the CPU, but also custom-makes the instruction set for it using "microcode."

General-Purpose CPUs

The third major direction of microprocessor evolution has been toward general-purpose CPUs which give a microcomputer most or all of the computing power of earlier minicomputers. After Motorola came out with the MC6800, Intel produced the 8085, an upgrade of the 8080 that required only a +5-V supply. Motorola then produced the MC6809, which has a few 16-bit instructions, but is still basically an 8-bit processor. In 1978 Intel came out with the 8086, which is a full 16-bit processor. Some 16-bit microprocessors, such as the National PACE and the Texas Instruments 9900 family of devices, had been available previously, but the market apparently wasn't ready. Soon after Intel came out with the 8086, Motorola came out with the 16-bit MC68000, and the 16-bit race was off and running. The 8086 and the 68000 work directly with 16-bit words instead of with 8-bit words, they can address a million or more bytes of memory instead of the 64 Kbytes addressable by the 8-bit processors, and they execute instructions much faster than the 8-bit processors. Also, these 16-bit processors have single instructions for functions such as *multiply* and *divide*, which required a lengthy sequence of instructions on the 8-bit processors.

The evolution along this last path has continued on to 32-bit processors that work with gigabytes (10^9 bytes) or terabytes (10^{12} bytes) of memory. Examples of these devices are the Intel 80386, the Motorola MC68020, and the National 32032.

Since we could not possibly describe in this book the operation and programming of even a few of the available processors, we confine our discussions primarily to one group of related microprocessors. The family we have chosen is the Intel 8086, 8088, 80186, 80188, 80286, 80386, 80486 family. Members of this family are very widely used in personal computers, business computer systems, and industrial control systems. Our experience has shown that learning the programming and operation of one family of microcomputers very thoroughly is much more useful than looking at many processors superficially. If you learn one processor family well, you will most likely find it quite easy to learn another when you have to.

THE 8086 MICROPROCESSOR FAMILY—OVERVIEW

The Intel 8086 is a 16-bit microprocessor that is intended to be used as the CPU in a microcomputer. The term *16-bit* means that its arithmetic logic unit, its internal registers, and most of its instructions are designed to work with 16-bit binary words. The 8086 has a 16-bit data bus, so it can read data from or write data to memory and ports either 16 bits or 8 bits at a time. The 8086 has a 20-bit address bus, so it can address any one of 2^{20}, or 1,048,576, memory locations.

Each of the 1,048,576 memory addresses of the 8086 represents a byte-wide location. Sixteen-bit words will be stored in two consecutive memory locations. If the first byte of a word is at an even address, the 8086 can read the entire word in one operation. If the first byte of the word is at an odd address, the 8086 will read the first byte with one bus operation and the second byte with another bus operation. Later we will discuss this in detail. The main point here is that if the first byte of a 16-bit word is at an even address, the 8086 can read the entire word in one operation.

The Intel 8088 has the same arithmetic logic unit, the same registers, and the same instruction set as the 8086. The 8088 also has a 20-bit address bus, so it can address any one of 1,048,576 bytes in memory. The 8088, however, has an 8-bit data bus, so it can only read data from or write data to memory and ports 8 bits at a time. The 8086, remember, can read or write either 8 or 16 bits at a time. To read a 16-bit word from two successive memory locations, the 8088 will always have to do two read operations. Since the 8086 and the 8088 are almost identical, any reference we make to the 8086 in the rest of the book will also pertain to the 8088 unless we specifically indicate otherwise. This is done to make reading easier. The Intel 8088, incidentally, is used as the CPU in the original IBM Personal Computer, the IBM PC/XT, and several compatible personal computers.

The Intel 80186 is an improved version of the 8086, and the 80188 is an improved version of the 8088. In addition to a 16-bit CPU, the 80186 and 80188 each have programmable peripheral devices integrated in the same package. In a later chapter we will discuss these integrated peripherals. The instruction set of the 80186 and 80188 is a *superset* of the instruction set of the 8086. The term *superset* means that all the 8086 and 8088 instructions will execute properly on an 80186 or an 80188, but the 80186 and the 80188 have a few additional instructions. In other words, a program written for an 8086 or an 8088 is *upward-compatible* to an 80186 or an 80188, but a program written for an 80186 or an 80188 may not execute correctly on an 8086 or an 8088. In the instruction set descriptions in Chapter 6, we specifically indicate which instructions work only with the 80186 or 80188.

The Intel 80286 is a 16-bit, advanced version of the 8086 which was specifically designed for use as the CPU in a multiuser or multitasking microcomputer. When operating in its *real address mode*, the 80286 functions mostly as a fast 8086. Most programs written for an 8086 can be run on an 80286 operating in its real address mode. When operating in its *virtual address mode*, an 80286 has features which make it easy to keep users' programs separate from one another and to protect the system program from destruction by users' programs. In Chapter 15 we discuss the operation and use of the 80286. The 80286 is the CPU used in the IBM PC/AT personal computer.

The Intel 80386 is a 32-bit microprocessor which can directly address up to 4 gigabytes of memory. The 80386 contains more sophisticated features than the 80286 for use in multiuser and multitasking microcomputer systems. In Chapter 15 we discuss the features of the 80386 and the 80486, which is an evolutionary step up from the 80386.

8086 INTERNAL ARCHITECTURE

Before we can talk about how to write programs for the 8086, we need to discuss its specific internal features, such as its ALU, flags, registers, instruction byte queue, and segment registers.

As shown by the block diagram in Figure 2-7, the 8086 CPU is divided into two independent functional parts, the *bus interface unit* or BIU, and the *execution unit* or EU. Dividing the work between these two units speeds up processing.

The BIU sends out addresses, fetches instructions from memory, reads data from ports and memory, and writes data to ports and memory. In other words, the BIU handles all transfers of data and addresses on the buses for the execution unit.

The execution unit of the 8086 tells the BIU where to fetch instructions or data from, decodes instructions, and executes instructions. Let's take a look at some of the parts of the execution unit.

The Execution Unit

CONTROL CIRCUITRY, INSTRUCTION DECODER, AND ALU

As shown in Figure 2-7, the EU contains *control circuitry* which directs internal operations. A *decoder* in the EU translates instructions fetched from memory into a series of actions which the EU carries out. The EU has a 16-bit *arithmetic logic unit* which can add, subtract, AND, OR, XOR, increment, decrement, complement, or shift binary numbers.

FLAG REGISTER

A *flag* is a flip-flop which indicates some condition produced by the execution of an instruction or controls certain operations of the EU. A 16-bit *flag register* in the EU contains nine active flags. Figure 2-8 shows the location of the nine flags in the flag register. Six of the nine flags are used to indicate some *condition* produced by an instruction. For example, a flip-flop called the *carry flag* will be set to a 1 if the addition of two 16-bit binary numbers produces a carry out of the most significant bit position. If no carry out of the MSB is produced by the addition, then the carry flag will be a 0. The EU thus effectively runs up a "flag" to tell you that a carry was produced.

The six conditional flags in this group are the *carry flag* (CF), the *parity flag* (PF), the *auxiliary carry flag* (AF), the *zero flag* (ZF), the *sign flag* (SF), and the *overflow flag* (OF). The names of these flags should give you hints as to what conditions affect them. Certain 8086 instructions check these flags to determine which of two alternative actions should be done in executing the instruction.

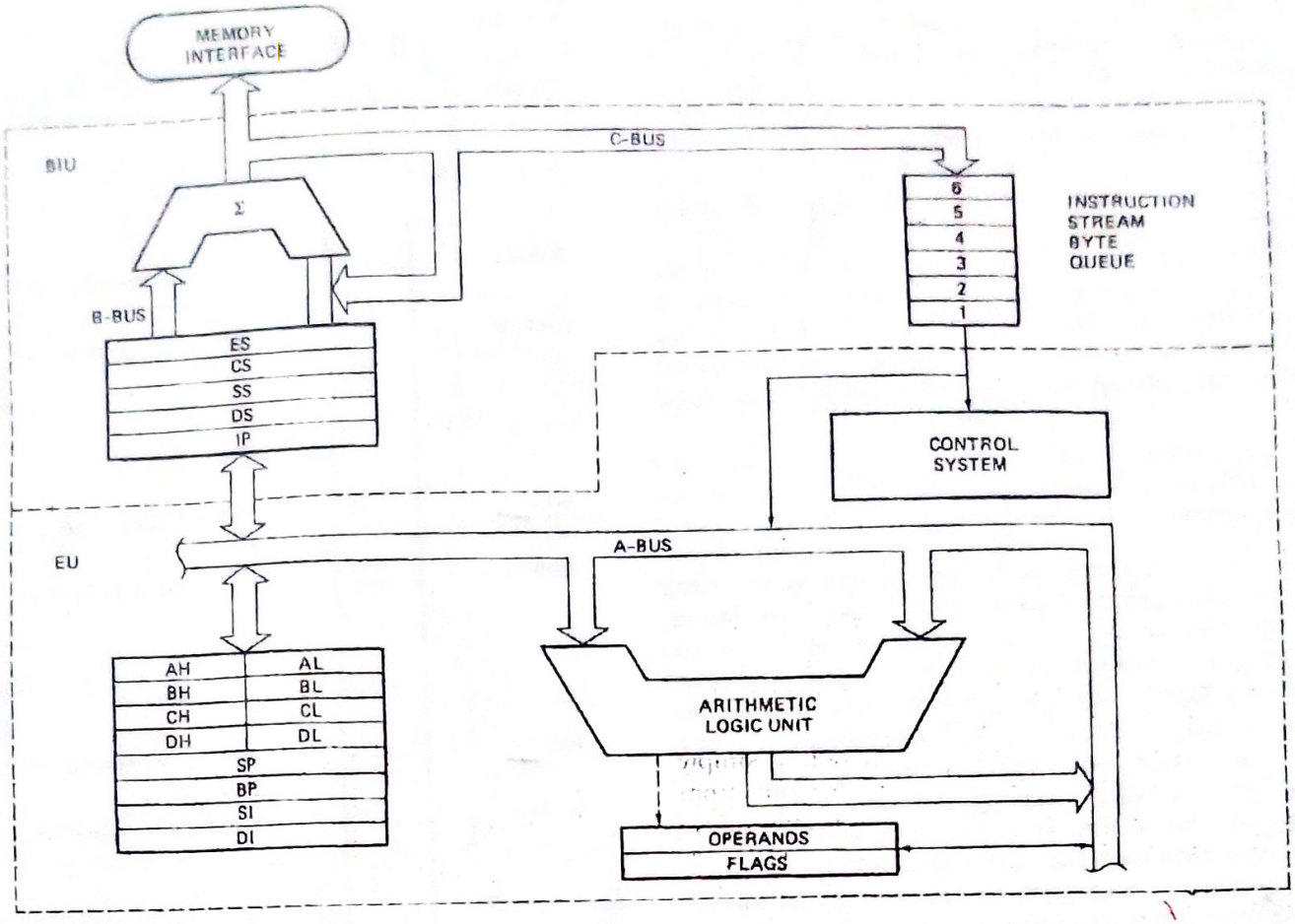


FIGURE 2-7 8086 internal block diagram. (*Intel Corp.*)

The three remaining flags in the flag register are used to *control* certain operations of the processor. These flags are different from the six conditional flags described above in the way they are set or reset. The six conditional flags are set or reset by the EU on the basis of the results of some arithmetic or logic operation. The *control flags* are deliberately set or reset with specific instructions you put in your program. The three control flags are the *trap flag* (TF), which is used for single stepping through a program; the *interrupt flag* (IF), which is used to allow or prohibit the interruption of a program; and the *direction flag* (DF), which is used with string instructions.

Later we will discuss in detail the operation and use of the nine flags.

GENERAL-PURPOSE REGISTERS

Observe in Figure 2-7 that the EU has eight *general-purpose registers*, labeled AH, AL, BH, BL, CH, CL, DH, and DL. These registers can be used individually for temporary storage of 8-bit data. The AL register is also called the *accumulator*. It has some features that the other general-purpose registers do not have.

Certain pairs of these general-purpose registers can be used together to store 16-bit data words. The acceptable

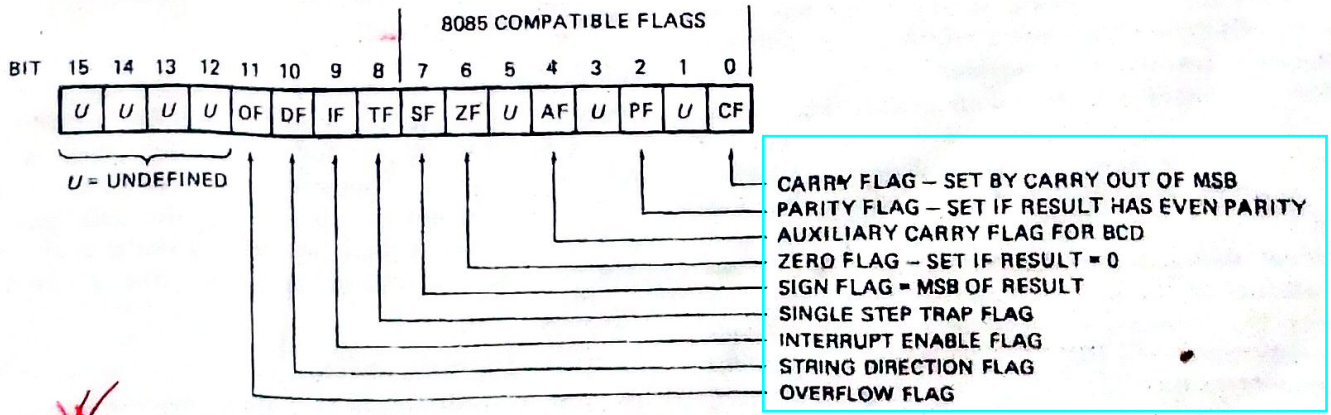


FIGURE 2-8 8086 flag register format. (*Intel Corp.*)

register pairs are AH and AL, BH and BL, CH and CL, and DH and DL. The AH–AL pair is referred to as the *AX register*, the BH–BL pair is referred to as the *BX register*, the CH–CL pair is referred to as the *CX register*, and the DH–DL pair is referred to as the *DX register*.

The 8086 general-purpose register set is very similar to those of the earlier-generation 8080 and 8085 microprocessors. It was designed this way so that the many programs written for the 8080 and 8085 could easily be translated to run on the 8086 or the 8088. The advantage of using internal registers for the temporary storage of data is that, since the data is already in the EU, it can be accessed much more quickly than it could be accessed in external memory. Now let's look at the features of the BIU.

The BIU

THE QUEUE

While the EU is decoding an instruction or executing an instruction which does not require use of the buses, the BIU fetches up to six instruction bytes for the following instructions. The BIU stores these prefetched bytes in a first-in–first-out register set called a *queue*. When the EU is ready for its next instruction, it simply reads the instruction byte(s) for the instruction from the queue in the BIU. This is much faster than sending out an address to the system memory and waiting for memory to send back the next instruction byte or bytes. The process is analogous to the way a bricklayer's assistant fetches bricks ahead of time and keeps a queue of bricks lined up so that the bricklayer can just reach out and grab a brick when necessary. Except in the cases of JMP and CALL instructions, where the queue must be dumped and then reloaded starting from a new address, this prefetch-and-queue scheme greatly speeds up processing. Fetching the next instruction while the current instruction executes is called *pipelining*.

SEGMENT REGISTERS

The 8086 BIU sends out 20-bit addresses, so it can address any of 2^{20} or 1,048,576 bytes in memory. However, at any given time the 8086 works with only four 65,536-byte (64-Kbyte) segments within this 1,048,576-byte (1-Mbyte) range. Four *segment registers* in the BIU are used to hold the upper 16 bits of the starting addresses of four memory segments that the 8086 is working with at a particular time. The four segment registers are the *code segment* (CS) register, the *stack segment* (SS) register, the *extra segment* (ES) register, and the *data segment* (DS) register.

Figure 2-9 shows how these four segments might be positioned in memory at a given time. The four segments can be separated as shown, or, for small programs which do not need all 64 Kbytes in each segment, they can overlap.

To repeat, then, a segment register is used to hold the upper 16 bits of the starting address for each of the segments. The code segment register, for example, holds the upper 16 bits of the starting address for the segment from which the BIU is currently fetching instruction code bytes. The BIU always inserts zeros for the lowest

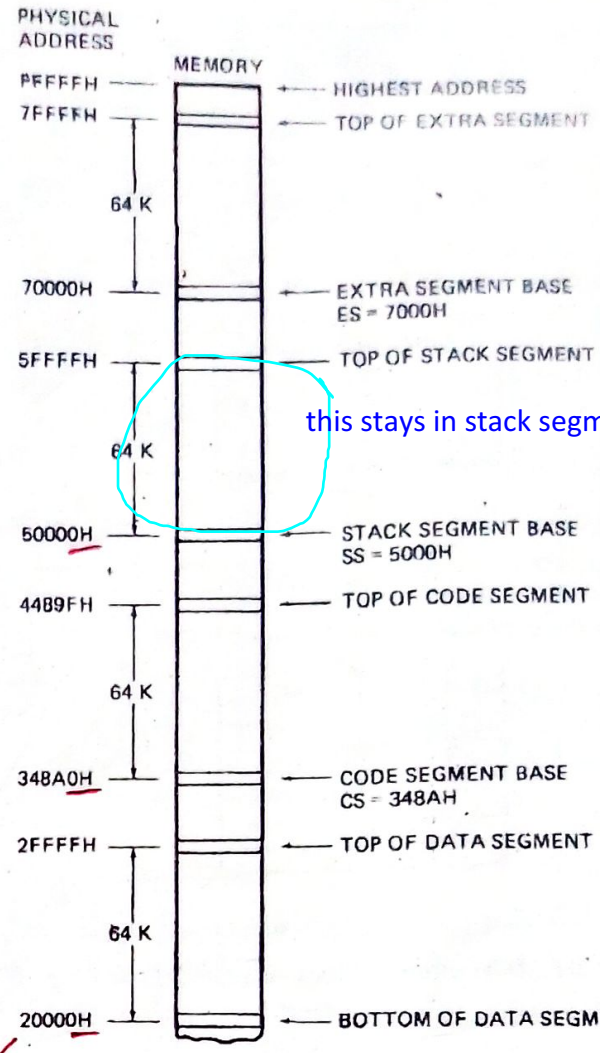


FIGURE 2-9 One way four 64-Kbyte segments might be positioned within the 1-Mbyte address space of an 8086.

4 bits (nibble) of the 20-bit starting address for a segment. If the code segment register contains 348AH, for example, then the code segment will start at address 348A0H. In other words, a 64-Kbyte segment can be located anywhere within the 1-Mbyte address space, but the segment will always start at an address with zeros in the lowest 4 bits. This constraint was put on the location of segments so that it is only necessary to store and manipulate 16-bit numbers when working with the starting address of a segment. The part of a segment starting address stored in a segment register is often called the *segment base*.

A *stack* is a section of memory set aside to store addresses and data while a *subprogram* executes. The stack segment register is used to hold the upper 16 bits of the starting address for the program stack. We will discuss the use and operation of a stack in detail later.

The extra segment register and the data segment register are used to hold the upper 16 bits of the starting addresses of two memory segments that are used for data.

INSTRUCTION POINTER

The next feature to look at in the BIU is the *instruction pointer* (IP) register. As discussed previously, the code

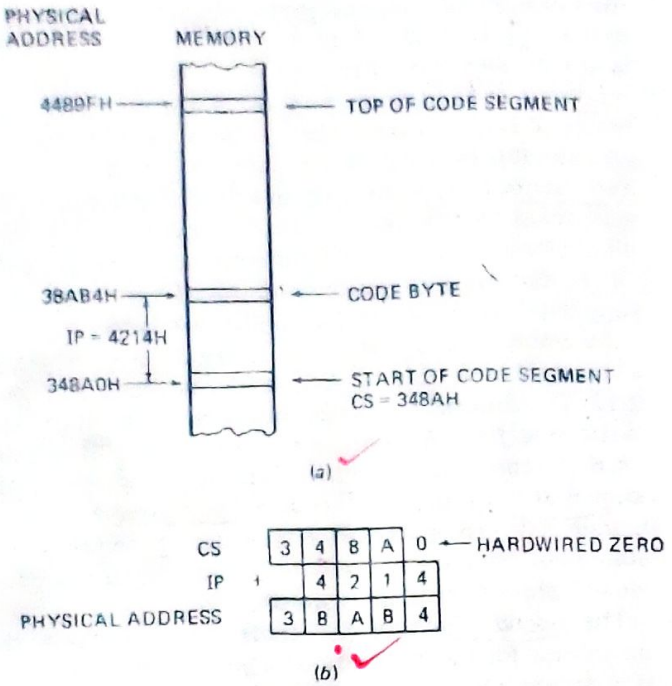


FIGURE 2-10 Addition of IP to CS to produce the physical address of the code byte. (a) Diagram. (b) Computation.

segment register holds the upper 16 bits of the starting address of the segment from which the BIU is currently fetching instruction code bytes. The instruction pointer register holds the 16-bit address, or *offset*, of the next code byte *within* this code segment. The value contained in the IP is referred to as an *offset* because this value must be offset from (added to) the segment base address in CS to produce the required 20-bit physical address sent out by the BIU. Figure 2-10*a* shows in diagram form how this works. The CS register points to the *base* or start of the current code segment. The IP contains the distance or offset from this base address to the next instruction byte to be fetched. Figure 2-10*b* shows how the 16-bit offset in IP is added to the 16-bit segment base address in CS to produce the 20-bit *physical* address. Notice that the two 16-bit numbers are not added directly in line, because the CS register contains only the upper 16 bits of the base address for the code segment. As we said before, the BIU automatically inserts zeros for the lowest 4 bits of the segment base address.

If the CS register, for example, contains 348AH, you know that the starting address for the code segment is 348A0H. When the BIU adds the offset of 4214H in the IP to this segment base address, the result is a 20-bit physical address of 38AB4H.

An alternative way of representing a 20-bit physical address is the *segment base:offset form*. For the address of a code byte, the format for this alternative form will be CS:IP. As an example of this, the address constructed in the preceding paragraph, 38AB4H, can also be represented as 348A:4214.

To summarize, then, the CS register contains the upper 16 bits of the starting address of the code segment in the 1-Mbyte address range of the 8086. The instruction pointer register contains a 16-bit offset which tells where in that 64-Kbyte code segment the next instruction byte is to be fetched from. The actual physical address sent to memory is produced by adding the offset contained in the IP register to the segment base represented by the upper 16 bits in the CS register.

Any time the 8086 accesses memory, the BIU produces the required 20-bit physical address by adding an offset to a segment base value represented by the contents of one of the segment registers. As another example of this, let's look at how the 8086 uses the contents of the stack segment register and the contents of the stack pointer register to produce a physical address.

STACK SEGMENT REGISTER AND STACK POINTER REGISTER

A stack, remember, is a section of memory set aside to store addresses and data while a subprogram is executing. The 8086 allows you to set aside an entire 64-Kbyte segment as a stack. The upper 16 bits of the starting address for this segment are kept in the stack segment register. The *stack pointer* (SP) register in the execution unit holds the 16-bit offset from the start of the segment to the memory location where a word was most recently stored on the stack. The memory location where a word was most recently stored is called the *top of stack*. Figure 2-11*a* shows this in diagram form.

The physical address for a stack read or a stack write is produced by adding the contents of the stack pointer register to the segment base address represented by the upper 16 bits of the base address in SS. Figure 2-11*b* shows an example. The 5000H in SS represents a segment base address of 50000H. When the FFE0H in the SP is added to this, the resultant physical address for the top of the stack will be 5FFE0H. The physical address can be represented either as a single number, 5FFE0H, or in SS:SP form as 5000:FFE0H.

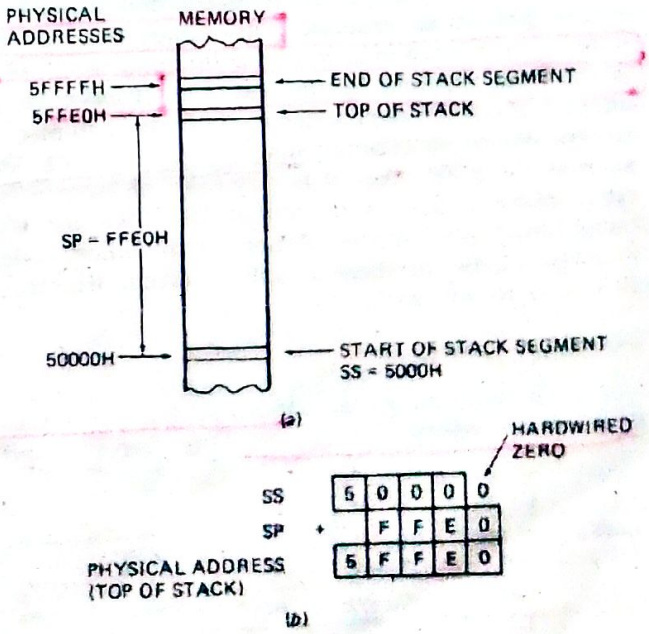


FIGURE 2-11 Addition of SS and SP to produce the physical address of the top of the stack. (a) Diagram. (b) Computation.

The operation and use of the stack will be discussed in detail later as need arises.

POINTER AND INDEX REGISTERS IN THE EXECUTION UNIT

In addition to the stack pointer register (SP), the EU contains a 16-bit *base pointer* (BP) register. It also contains a 16-bit *source index* (SI) register and a 16-bit *destination index* (DI) register. These three registers can be used for temporary storage of data just as the general-purpose registers described above. However, their main use is to hold the 16-bit offset of a data word in one of the segments. SI, for example, can be used to hold the offset of a data word in the data segment. The physical address of the data in memory will be generated in this case by adding the contents of SI to the segment base address represented by the 16-bit number in the DS register. After we give you an overview of the different levels of languages used to program a microcomputer, we will show you some examples of how we tell the 8086 to read data from or write data to a desired memory location.

INTRODUCTION TO PROGRAMMING THE 8086

Programming Languages

Now that you have an overview of the 8086 CPU, it is time to start thinking about how it is programmed. To run a program, a microcomputer must have the program stored in binary form in successive memory locations, as shown in Figure 2-12. There are three language levels that can be used to write a program for a microcomputer.

MACHINE LANGUAGE

You can write programs as simply a sequence of the binary codes for the instructions you want the microcomputer to execute. The three-instruction program in Figure 2-6*b* is an example. This binary form of the program is referred to as *machine language* because it is the form required by the machine. However, it is difficult, if not impossible, for a programmer to memorize the thousands of binary instruction codes for a CPU such as the 8086. Also, it is very easy for an error to occur when working with long series of 1's and 0's. Using hexadecimal representation for the binary codes might help some, but there are still thousands of instruction codes to cope with.

ASSEMBLY LANGUAGE

To make programming easier, many programmers write programs in *assembly language*. They then translate the assembly language program to machine language so that it can be loaded into memory and run. Assembly language uses two-, three-, or four-letter *mnemonics* to represent each instruction type. A mnemonic is just a device to help you remember something. The letters in an assembly language mnemonic are usually initials or a shortened form of the English word(s) for the operation performed by the instruction. For example, the mnemonic for subtract is SUB, the mnemonic for Exclusive OR is XOR, and the mnemonic for the instruction to copy data from one location to another is MOV.

LABEL FIELD	OP CODE FIELD	OPERAND FIELD	COMMENT FIELD
NEXT:	ADD	AL, 07H	;ADD CORRECTION FACTOR

FIGURE 2-12 Assembly language program statement format.

Assembly language statements are usually written in a standard form that has four *fields*, as shown in Figure 2-12. The first field in an assembly language statement is the *label field*. A *label* is a symbol or group of symbols used to represent an address which is not specifically known at the time the statement is written. Labels are usually followed by a colon. Labels are not required in a statement, they are just inserted where they are needed. We will show later many uses of labels.

The *opcode field* of the instruction contains the mnemonic for the instruction to be performed. Instruction mnemonics are sometimes called *operation codes*, or *opcodes*. The ADD mnemonic in the example statement in Figure 2-12 indicates that we want the instruction to do an addition.

The *operand field* of the statement contains the data, the memory address, the port address, or the name of the register on which the instruction is to be performed. *Operand* is just another name for the data item(s) acted on by an instruction. In the example instruction in Figure 2-12, there are two operands, AL and 07H, specified in the operand field. AL represents the AL register, and 07H represents the number 07H. This assembly language statement thus says, "Add the number 07H to the contents of the AL register." By Intel convention, the result of the addition will be put in the register or the memory location specified *before* the comma in the operand field. For the example statement in Figure 2-12, then, the result will be left in the AL register. As another example, the assembly language statement ADD BH, AL, when converted to machine language and run, will add the contents of the AL register to the contents of the BH register. The results will be left in the BH register.

The final field in an assembly language statement such as that in Figure 2-12 is the *comment field*, which starts with a semicolon. Comments do not become part of the machine language program, but they are very important. You write *comments* in a program to remind you of the function that an instruction or group of instructions performs in the program.

To summarize why assembly language is easier to use than machine language, let's look a little more closely at the assembly language ADD statement. The general format of the 8086 ADD instruction is

ADD destination, source

The *source* can be a number written in the instruction, the contents of a specified register, or the contents of a memory location. The *destination* can be a specified register or a specified memory location. However, the

source and the destination in an instruction cannot both be memory locations.

A later section on 8086 addressing modes will show all the ways in which the source of an operand and the destination of the result can be specified. The point here is that the single mnemonic ADD, together with a specified source and a specified destination, can represent a great many 8086 instructions in an easily understandable form.

The question that may occur to you at this point is, "If I write a program in assembly language, how do I get it translated into machine language which can be loaded into the microcomputer and executed?" There are two answers to this question. The first method of doing the translation is to work out the binary code for each instruction a bit at a time using the templates given in the manufacturer's data books. We will show you how to do this in the next chapter, but it is a tedious and error-prone task. The second method of doing the translation is with an *assembler*. An assembler is a program which can be run on a personal computer or *microcomputer development system*. It reads the file of assembly language instructions you write and generates the correct binary code for each. For developing all but the simplest assembly language programs, an assembler and other program development tools are essential. We will introduce you to these program development tools in the next chapter and describe their use throughout the rest of this book.

HIGH-LEVEL LANGUAGES

Another way of writing a program for a microcomputer is with a *high-level language*, such as BASIC, Pascal, or C. These languages use program statements which are even more English-like than those of assembly language. Each high-level statement may represent many machine code instructions. An *interpreter program* or a *compiler program* is used to translate higher-level language statements to machine codes which can be loaded into memory and executed. Programs can usually be written faster in high-level languages than in assembly language because the high-level language works with bigger building blocks. However, programs written in a high-level language and interpreted or compiled almost always execute more slowly and require more memory than the same programs written in assembly language. Programs that involve a lot of hardware control, such as robots and factory control systems, or programs that must run as quickly as possible are usually best written in assembly language. Complex data processing programs that manipulate massive amounts of data, such as insurance company records, are usually best written in a high-level language. The decision concerning which language to use has recently been made more difficult by the fact that current assemblers allow the use of many high-level language features, and the fact that some current high-level languages provide assembly language features.

OUR CHOICE

For most of this book we work very closely with hardware, so assembly language is the best choice. In later chapters, however, we do show you how to write programs which contain modules written in assembly language and modules written in the high-level language C. In the next chapter we introduce you to assembly language programming techniques. Before we go on to that, however, we will use a few simple 8086 instructions to show you more about accessing data in registers and memory locations.

How the 8086 Accesses Immediate and Register Data

In a previous discussion of the 8086 BIU, we described how the 8086 accesses code bytes using the contents of the CS and IP registers. We also described how the 8086 accesses the stack using the contents of the SS and SP registers. Before we can teach you assembly language programming techniques, we need to discuss some of the different ways in which an 8086 can access the data that it operates on. The different ways in which a processor can access data are referred to as its *addressing modes*. In assembly language statements, the addressing mode is indicated in the instruction. We will use the 8086 MOV instruction to illustrate some of the 8086 addressing modes.

The MOV instruction has the format

MOV destination, source

When executed, this instruction copies a word or a byte from the specified source location to the specified destination location. The source can be a number written directly in the instruction, a specified register, or a memory location specified in 1 of 24 different ways. The destination can be a specified register or a memory location specified in any 1 of 24 different ways. The source and the destination cannot both be memory locations in an instruction.

IMMEDIATE ADDRESSING MODE

Suppose that in a program you need to put the number 437BH in the CX register. The MOV CX, 437BH instruction can be used to do this. When it executes, this instruction will put the *immediate* hexadecimal number 437BH in the 16-bit CX register. This is referred to as *immediate addressing mode* because the number to be loaded into the CX register will be put in the two memory locations immediately following the code for the MOV instruction. This is similar to the way the port address was put in memory immediately after the code for the input instruction in the three-instruction program in Figure 2-6*b*.

A similar instruction, MOV CL, 48H, could be used to load the 8-bit immediate number 48H into the 8-bit CL register. You can also write instructions to load an 8-bit immediate number into an 8-bit memory location or to load a 16-bit number into two consecutive memory locations, but we are not yet ready to show you how to specify these.

REGISTER ADDRESSING MODE

Register addressing mode means that a register is the source of an operand for an instruction. The instruction

MOV CX, AX, for example, copies the contents of the 16-bit AX register into the 16-bit CX register. Remember that the destination location is specified in the instruction before the comma, and the source is specified after the comma. Also note that the contents of AX are just copied to CX, not actually moved. In other words, the previous contents of CX are written over, but the contents of AX are not changed. For example, if CX contains 2A84H and AX contains 4971H before the MOV CX, AX instruction executes, then after the instruction executes, CX will contain 4971H and AX will still contain 4971H. You can MOV any 16-bit register to any 16-bit register, or you can MOV any 8-bit register to any 8-bit register. However, you cannot use an instruction such as MOV CX, AL because this is an attempt to copy a *byte-type* operand (AL) into a *word-type* destination (CX). The byte in AL would fit in CX, but the 8086 would not know which half of CX to put it in. If you try to write an instruction like this and you are using a good assembler, the assembler will tell you that the instruction contains a *type error*. To copy the byte from AL to the high byte of CX, you can use the instruction MOV CH, AL. To copy the byte from AL to the low byte of CX, you can use the instruction MOV CL, AL.

Accessing Data in Memory

OVERVIEW OF MEMORY ADDRESSING MODES

The addressing modes described in the following sections are used to specify the location of an operand in memory. To access data in memory, the 8086 must also produce a 20-bit physical address. It does this by adding a 16-bit value called the *effective address* to a segment base address represented by the 16-bit number in one of the four segment registers. The effective address (EA) represents the *displacement* or *offset* of the desired operand from the segment base. In most cases, any of the segment bases can be specified, but the data segment is the one most often used. Figure 2-13*a* shows in graphic form how the EA is added to the data segment base to point to an operand in memory. Figure 2-13*b* shows how the 20-bit physical address is generated by the BIU. The starting address for the data segment in Figure 2-13*b* is 20000H, so the data segment register will contain 2000H. The BIU adds the effective address, 437AH, to the data segment base address of 20000H to produce the physical address sent out to memory. The 20-bit physical address sent out to memory by the BIU will then be 2437AH. The physical address can be represented either as a single number 2437AH or in the segment base:offset form as 2000:437AH.

The execution unit calculates the effective address for an operand using information you specify in the instruction. You can tell the EU to use a number in the instruction as the effective address, to use the contents of a specified register as the effective address, or to compute the effective address by adding a number in the instruction to the contents of one or two specified registers. The following section describes one way you can tell the execution unit to calculate an effective address. In later chapters we show other ways of specifying the effective address. Later we also show how the addressing modes this provides are used to solve some common programming problems.

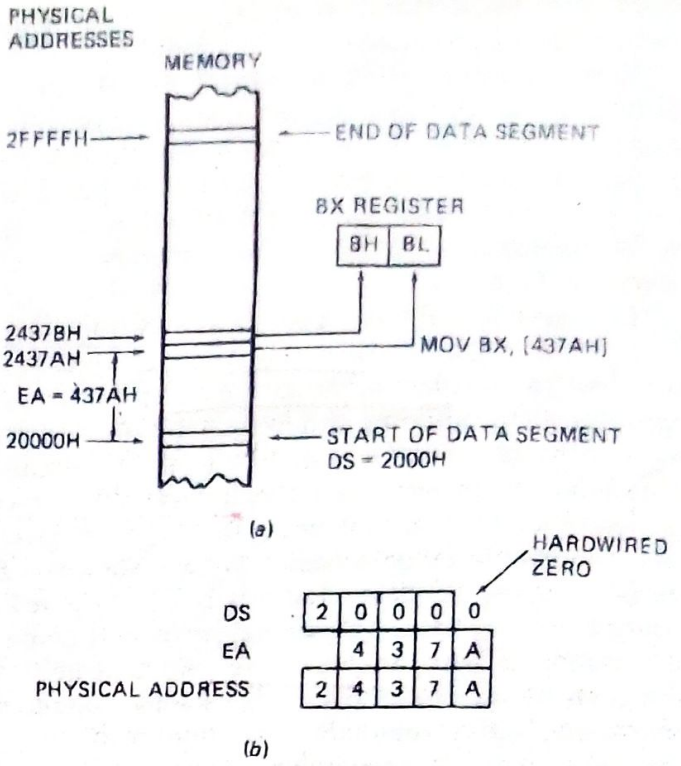


FIGURE 2-13 Addition of data segment register and effective address to produce the physical address of the data byte. (*a*) Diagram. (*b*) Computation.

DIRECT ADDRESSING MODE

For the simplest memory addressing mode, the effective address is just a 16-bit number written directly in the instruction. The instruction MOV BL, [437AH] is an example. The square brackets around the 437AH are shorthand for "the contents of the memory location(s) at a displacement from the segment base of." When executed, this instruction will copy "the contents of the memory location at a displacement from the data segment base of " 437AH into the BL register, as shown by the rightmost arrow in Figure 2-13*a*. The BIU calculates the 20-bit physical memory address by adding the effective address 437AH to the data segment base, as shown in Figure 2-13*b*. This addressing mode is called *direct* because the displacement of the operand from the segment base is specified directly in the instruction. The displacement in the instruction will be added to the data segment base in DS unless you tell the BIU to add it to some other segment base. Later we will show you how to do this.

Another example of the direct addressing mode is the instruction MOV BX, [437AH]. When executed, this instruction copies a 16-bit word from memory into the BX register. Since each memory address of the 8086 represents a byte of storage, the word must come from two memory locations. The byte at a displacement of 437AH from the data segment base will be copied into BL, as shown by the right arrow in Figure 2-13*a*. The contents of the next higher address, displacement 437BH, will be copied into the BH register, as shown by the left arrow in Figure 2-13*a*. From the instruction

coding, the 8086 will automatically determine the number of bytes that it must access in memory.

An important point here is that an 8086 always stores the low byte of a word in the lower of the two addresses and stores the high byte of a word in the higher address. To stick this in your mind, remember:

Low byte–low address, high byte–high address

The previous two examples showed how the direct addressing mode can be used to specify the source of an operand. Direct addressing can also be used to specify the destination of an operand in memory. The instruction MOV [437AH], BX, for example, will copy the contents of the BX register to two memory locations in the data segment. The contents of BL will be copied to the memory location at a displacement of 437AH. The contents of BH will be copied to the memory location at a displacement of 437BH. This operation is represented by simply reversing the direction of the arrows in Figure 2-13a.

NOTE: When you are *hand-coding* programs using direct addressing of the form shown above, make sure to put in the square brackets to remind you how to code the instruction. If you leave the brackets out of an instruction such as MOV BX, [437AH], you will code it as if it were the instruction MOV BX, 437AH. This second instruction will load the immediate number 437AH into BX, rather than loading a word from memory at a displacement of 437AH into BX. Also note that if you are writing an instruction using direct addressing such as this for an assembler, you must write the instruction in the form MOV BL, DS:BYTE PTR [437AH] to give the assembler all the information it needs. As we will show you in the next chapter, when you are using an assembler, you usually use a name to represent the direct address rather than the actual numerical value.

A FEW WORDS ABOUT SEGMENTATION

At this point you may be wondering why Intel designed the 8086 family devices to access memory using the segment:offset approach rather than accessing memory directly with 20-bit addresses. The segment:offset scheme requires only a 16-bit number to represent the base address for a segment, and only a 16-bit offset to access any location in a segment. This means that the 8086 has to manipulate and store only 16-bit quantities instead of 20-bit quantities. This makes for an easier interface with 8- and 16-bit-wide memory boards and with the 16-bit registers in the 8086.

The second reason for segmentation has to do with the type of microcomputer in which an 8086-family CPU is likely to be used. A previous section of this chapter described briefly the operation of a timesharing microcomputer system. In a timesharing system, several users share a CPU. The CPU works on one user's program for perhaps 20 ms, then works on the next user's program for 20 ms. After working 20 ms for each of the other users, the CPU comes back to the first user's program again. Each time the CPU switches from one user's program to the next, it must access a new section of code and new sections of data. Segmentation makes this switching quite easy. Each user's program can be assigned a separate set of logical segments for its code and data. The user's program will contain offsets or displacements from these segment bases. To change from one user's program to a second user's program, all that the CPU has to do is to reload the four segment registers with the segment base addresses assigned to the second user's program. In other words, segmentation makes it easy to keep users' programs and data separate from one another, and segmentation makes it easy to switch from one user's program to another user's program. In Chapter 15 we tell you much more about the use of segmentation in multiuser systems.

CHECKLIST OF IMPORTANT TERMS AND CONCEPTS IN THIS CHAPTER

If you do not remember any of the terms or concepts in the following list, use the index to find them in the chapter.

Microcomputer, microprocessor

Hardware, software, firmware

Timesharing computer system

Multitasking computer system

Distributed processing system

Multiprocessing

CPU

Memory, RAM, ROM

I/O ports

Address, data, and control buses

Control bus signals

ALU

Segmentation

Bus interface unit (BIU)
Instruction byte queue, pipelining,
ES, CS, SS, DS registers, IP register

Execution unit (EU)
AX, BX, CX, DX registers, flag register,
ALU, SP, BP, SI, DI registers

Machine language, assembly language, high-level language

Mnemonic, opcode, operand, label, comment

Assembler, compiler

Immediate address mode, register address mode, direct address mode

Effective address

```
                        ; 8086 PROGRAM F4-05.ASM
;ABSTRACT   : Program produces a packed BCD byte from 2 ASCII-encoded digits
                        ; The first ASCII digit (5) is loaded in BL.
                        ; The second ASCII digit (9) is loaded in AL.
;REGISTERS  ; Uses CS, AL, BL, CL
;PORTS      : None used

 0000
CODE    SEGMENT
        ASSUME CS:CODE
 0000  B3 35    START:  MOV  BL, '5'    ; Load first ASCII digit into BL
 0002  B0 39            MOV  AL, '9'    ; Load second ASCII digit into AL
 0004  80 E3 0F         AND  BL, 0FH    ; Mask upper 4 bits of first digit
 0007  24 0F            AND  AL, 0FH    ; Mask upper 4 bits of second digit
 0009  B1 04            MOV  CL, 04H    ; Load CL for 4 rotates required
 000B  D2 C3            ROL  BL, CL     ; Rotate BL 4 bit positions
 000D  0A C3            OR   AL, BL     ; Combine nibbles, result in AL
 000F           CODE    ENDS
                        END START
```

FIGURE 4-5 List file of 8086 assembly language program to produce packed BCD from two ASCII characters.

For the example program here, we use the instruction OR AL,BL to pack the two BCD nibbles. Bits ORed with 0's will not be changed. Bits ORed with 1's will become or stay 1's. Again look at Figure 4-2 to help you visualize this operation.

SUMMARY OF BCD PACKING PROGRAM

If you compare the algorithm for this program with the finished program in Figure 4-5, you should see that each step in the algorithm translates to one or two assembly language instructions. As we told you before, developing the assembly language program from a good algorithm is really quite easy because you are simply translating one step at a time to its equivalent assembly language instructions. Also, debugging a program developed in this way is quite easy because you simply single-step or breakpoint your way through it and check the results after each step. In the next section we discuss the 8086 JMP instructions and flags so we can show you how you implement some of the other programming structures in assembly language.

JUMPS, FLAGS, AND CONDITIONAL JUMPS

Introduction

The real power of a computer comes from its ability to choose between two or more sequences of actions based on some condition, repeat a sequence of instructions *as long as* some condition exists, or repeat a sequence of instructions *until* some condition exists. *Flags* indicate whether some condition is present or not. *Jump* instructions are used to tell the computer the address to fetch its next instruction from. Figure 4-6 shows in diagram form the different ways a Jump instruction can direct the 8086 to fetch its next instruction from some place in memory other than the next sequential location.

The 8086 has two types of Jump instructions, conditional and unconditional. When the 8086 fetches and decodes an Unconditional Jump instruction, it always goes to the specified jump destination. You might use this type of Jump instruction at the end of a program so that the entire program runs over and over, as shown in Figure 4-6.

When the 8086 fetches and decodes a Conditional Jump instruction, it evaluates the state of a specified flag to determine whether to fetch its next instruction from the jump destination location or to fetch its next instruction from the next sequential memory location.

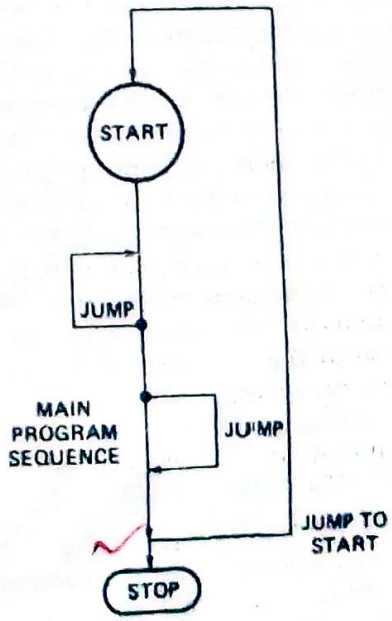


FIGURE 4-6 Change in program flow that can be caused by jump instructions.

```
                        ; 8086 PROGRAM
                        ;ABSTRACT  : This program illustrates a
                        ;REGISTERS : Uses CS, AX
                        ;PORTS     : None used

1
2
3                       CODE    SEGMENT
4                               ASSUME CS:CODE
5                               JMP THERE       ; Skip over a series of instructions
6 0000                          NOP             ; Dummy instructions to represent those
7                               NOP             ; instructions skipped over
8 0000  EB 03 90                MOV AX, 0000H   ; Zero accumulator before addition instructions
9 0003  90              THERE:  NOP             ; Dummy instruction to represent continuation of execution
10 0004 90
11 0005 B8 0000         CODE    ENDS
12 0008 90                      END
13 0009
14
```

FIGURE 4-9 List file of program demonstrating "forward" JMP.

JMP. The displacement is calculated by counting the number of bytes from the next address after the JMP instruction to the destination. If the displacement is negative (backward in the program), then it must be expressed in 2's complement form before it can be written in the instruction code template.

Now let's look at another simple example program, in Figure 4-9, to see how you can jump ahead over a group of instructions in a program. Here again we use a label to give a name to the address that we want to JMP to. We also use NOP instructions to represent the instructions that we want to skip over and the instructions that continue after the JMP. Let's see how this JMP instruction is coded.

When the assembler reads through the source file for this program, it will find the label "THERE" after the JMP mnemonic. At this point the assembler has no way of knowing whether it will need 1 or 2 bytes to represent the displacement to the destination address. The assembler plays it safe by reserving 2 bytes for the displacement. Then the assembler reads on through the rest of the program. When the assembler finds the specified label, it calculates the displacement from the instruction after the JMP instruction to the label. If the assembler finds the displacement to be outside the range of −128 bytes to +127 bytes, then it will code the instruction as a direct within-segment near JMP with 2 bytes of displacement. If the assembler finds the displacement to be within the −128- to +127-byte range, then it will code the instruction as a direct within-segment short-type JMP with a 1-byte displacement. In the latter case, the assembler will put the code for a NOP instruction, 90H, in the third byte it had reserved for the JMP instruction. The instruction codes for the JMP THERE instruction on line 8 of Figure 4-9 demonstrate this. As shown in the instruction template in Figure 4-7, EBH is the basic opcode for the direct within-segment short JMP. The 03H represents the displacement to the JMP destination. Since we are jumping forward in this case, the displacement is a positive number. The 90H in the next memory byte is the code for a NOP instruction. The displacement is calculated from the offset of this NOP instruction, 0002H, to the offset of the destination label, 0005H. The difference of 03H between these two is the displacement you see coded in the instruction.

If you are hand coding a program such as this, you will probably know how far it is to the label, and you can leave just 1 byte for the displacement if that is enough. If you are using an assembler and you don't want to waste the byte of memory or the time it takes to fetch the extra NOP instruction, you can write the instruction as JMP SHORT label. The SHORT operator is a promise to the assembler that the destination will not be outside the range of −128 to +127 bytes. Trusting your promise, the assembler then reserves only 1 byte for the displacement.

Note that if you are making a JMP from an address near the start of a 64-Kbyte segment to an address near the end of the segment, you may not be able to get there with a jump of +32,767. The way you get there is to JMP backward around to the desired destination address. An assembler will automatically do this for you.

One advantage of the direct near- and short-type JMPs is that the destination address is specified *relative* to the address of the instruction after the JMP instruction. Since the JMP instruction in this case does not contain an absolute address or offset, the program can be loaded anywhere in memory and still run correctly. A program which can be loaded anywhere in memory to be run is said to be *relocatable*. You should try to write your programs so that they are relocatable.

Now that you know about unconditional JMP instructions, we will discuss the 8086 flags, so that we can show how the 8086 Conditional Jump instructions are used to implement the rest of the standard programming structures.

The 8086 Conditional Flags

The 8086 has six *conditional flags*. They are the *carry* flag (CF), the *parity* flag (PF), the *auxiliary carry* flag (AF), the *zero* flag (ZF), the *sign* flag (SF), and the *overflow* flag (OF). Chapter 1 shows numerical examples of some of the conditions indicated by these flags. Here we review these conditions and show how some of the important 8086 instructions affect these flags.

THE CARRY FLAG WITH ADD, SUBTRACT, AND COMPARE INSTRUCTIONS

If the addition of two 8-bit numbers produces a sum greater than 8 bits, the carry flag will be set to a 1 to indicate a carry into the next bit position. Likewise, if

the addition of two 16-bit numbers produces a sum greater than 16 bits, then the carry flag will be set to a 1 to indicate that a final carry was produced by the addition.

During subtraction, the carry flag functions as a borrow flag. If the bottom number in a subtraction is larger than the top number, then the carry/borrow flag will be set to indicate that a borrow was needed to perform the subtraction.

The 8086 compare instruction has the format CMP destination,source. The source can be an immediate number, a register, or a memory location. The destination can be a register or a memory location. The comparison is done by subtracting the contents of the specified source from the contents of the specified destination. Flags are updated to reflect the result of the comparison, but neither the source nor the destination is changed. If the source operand is greater than the specified destination operand, then the carry/borrow flag will be set to indicate that a borrow was needed to do the comparison (subtraction). If the source operand is the same size as or smaller than the specified destination operand, then the carry/borrow flag will not be set after the compare. If the two operands are equal, the zero flag will be set to a 1 to indicate that the result of the compare (subtraction) was all 0's. Here's an example and summary of this for your reference.

CMP BX, CX

condition	CF	ZF
CX > BX	1	0
CX < BX	0	0
CX = BX	0	1

The compare instruction is very important because it allows you to easily determine whether one operand is greater than, less than, or the same size as another operand.

THE PARITY FLAG

Parity is a term used to indicate whether a binary word has an even number of 1's or an odd number of 1's. A binary number with an even number of 1's is said to have *even parity*. The 8086 parity flag will be set to a 1 after an instruction if the lower 8 bits of the destination operand has an even number of 1's. Probably the most common use of the parity flag is to determine whether ASCII data sent to a computer over phone lines or some other communications link contains any errors. In Chapter 14 we describe this use of parity.

THE AUXILIARY CARRY FLAG

This flag has significance in BCD addition or BCD subtraction. If a carry is produced when the least significant nibbles of 2 bytes are added, the auxiliary carry flag will be set. In other words, a carry out of bit 3 sets the auxiliary carry flag. Likewise, if the subtraction of the least significant nibbles requires a borrow, the auxiliary carry/borrow flag will be set. The auxiliary carry/borrow flag is used only by the DAA and DAS instructions. Consult the DAA and DAS instruction descriptions in Chapter 6 and the BCD operation examples section of Chapter 1 for further discussion of addition and subtraction of BCD numbers.

THE ZERO FLAG WITH INCREMENT, DECREMENT, AND COMPARE INSTRUCTIONS

As the name implies, this flag will be set to a 1 if the result of an arithmetic or logic operation is zero. For example, if you subtract two numbers which are equal, the zero flag will be set to indicate that the result of the subtraction is zero. If you AND two words together and the result contains no 1's, the zero flag will be set to indicate that the result is all 0's.

Besides the more obvious arithmetic and logic instructions, there are a few other very useful instructions which also affect the zero flag. One of these is the compare instruction CMP, which we discussed previously with the carry flag. As shown there, the zero flag will be set to a 1 if the two operands compared are equal.

Another important instruction which affects the zero flag is the decrement instruction, DEC. This instruction will decrement (or, in other words, subtract 1 from) a number in a specified register or memory location. If, after decrementing, the contents of the register or memory location are zero, the zero flag will be set. Here's a preview of how this is used. Suppose that we want to repeat a sequence of actions nine times. To do this, we first load a register with the number 09H and execute the sequence of actions. We then decrement the register and look at the zero flag to see if the register is down to zero yet. If the zero flag is not set, then we know that the register is not yet down to zero, so we tell the 8086, with a Jump instruction, to go back and execute the sequence of instructions again. The following sections will show many specific examples of how this is done.

The increment instruction, INC destination, also affects the zero flag. If an 8-bit destination containing FFH or a 16-bit destination containing FFFFH is incremented, the result in the destination will be all 0's. The zero flag will be set to indicate this.

THE SIGN FLAG—POSITIVE AND NEGATIVE NUMBERS

When you need to represent both positive and negative numbers for an 8086, you use 2's complement sign-and-magnitude form as described in Chapter 1. In this form, the most significant bit of the byte or word is used as a sign bit. A 0 in this bit indicates that the number is positive. A 1 in this bit indicates that the number is negative. The remaining 7 bits of a byte or the remaining 15 bits of a word are used to represent the magnitude of the number. For a positive number, the magnitude will be in standard binary form. For a negative number, the magnitude will be in 2's complement form. After an arithmetic or logic instruction executes, the sign flag will be a copy of the most significant bit of the destination byte or the destination word. In addition to its use with signed arithmetic operations, the sign flag can be used to determine whether an operand has been decremented beyond zero. Decrementing 00H, for example, will give FFH. Since the MSB of FFH is a 1, the sign flag will be set.

THE OVERFLOW FLAG

This flag will be set if the result of a signed operation is too large to fit in the number of bits available to represent it. To remind you of what *overflow* means, here is an example. Suppose you add the 8-bit signed number 01110101 (+117 decimal) and the 8-bit signed number 00110111 (+55 decimal). The result will be 10101100 (+172 decimal), which is the correct binary result in this case, but is too large to fit in the 7 bits allowed for the magnitude in an 8-bit signed number. For an 8-bit signed number, a 1 in the most significant bit indicates a negative number. The overflow flag will be set after this operation to indicate that the result of the addition has overflowed into the sign bit.

The 8086 Conditional Jump Instructions

As we stated previously, much of the real power of a computer comes from its ability to choose between two courses of action depending on whether some condition is present or not. In the 8086 the six conditional flags indicate the conditions that are present after an instruction. The 8086 Conditional Jump instructions look at the state of a specified flag(s) to determine whether the jump should be made or not.

Figure 4-10 shows the mnemonics for the 8086 Conditional Jump instructions. Next to each mnemonic is a brief explanation of the mnemonic. Note that the terms *above* and *below* are used when you are working with unsigned binary numbers. The 8-bit unsigned number 11000110 is above the 8-bit unsigned number 00111001, for example. The terms *greater* and *less* are used when you are working with signed binary numbers. The 8-bit signed number 00111001 is greater (more positive) than the 8-bit signed number 11000110, which represents a negative number. Also shown in Figure 4-10 is an indication of the flag conditions that will cause the 8086 to do the jump. If the specified flag conditions are not present, the 8086 will just continue on to the next instruction in sequence. In other words, if the jump condition is not met, the Conditional Jump instruction will effectively function as a NOP. Suppose, for example, we have the instruction JC SAVE, where SAVE is the label at the destination address. If the carry flag is set, this instruction will cause the 8086 to jump to the instruction at the SAVE: label. If the carry flag is not set, the instruction will have no effect other than taking up a little processor time.

All conditional jumps are *short-type* jumps. This means that the destination label must be in the same code segment as the jump instruction. Also, the destination address must be in the range of −128 bytes to +127 bytes from the address of the instruction after the Jump instruction. As we show in later examples, it is important to be aware of this limit on the range of conditional jumps as you write your programs.

The Conditional Jump instructions are usually used after arithmetic or logic instructions. They are very commonly used after Compare instructions. For this case, the Compare instruction syntax and the Conditional Jump instruction syntax are such that a little trick makes it very easy to see what will cause a jump to occur. Here's the trick. Suppose that you see the instruction sequence

```
CMP BL, DH
JAE HEATER_OFF
```

in a program, and you want to determine what these instructions do. The CMP instruction compares the byte

MNEMONIC	CONDITION TESTED	"JUMP IF . . ."
JA/JNBE	(CF or ZF) = 0	above/not below nor equal
JAE/JNB	CF = 0	above or equal/not below
JB/JNAE	CF = 1	below/not above nor equal
JBE/JNA	(CF or ZF) = 1	below or equal/not above
JC	CF = 1	carry
JE/JZ	ZF = 1	equal/zero
JG/JNLE	((SF xor OF) or ZF) = 0	greater/not less nor equal
JGE/JNL	(SF xor OF) = 0	greater or equal/not less
JL/JNGE	(SF xor OF) = 1	less/not greater nor equal
JLE/JNG	((SF xor OF) or ZF) = 1	less or equal/not greater
JNC	CF = 0	not carry
JNE/JNZ	ZF = 0	not equal/not zero
JNO	OF = 0	not overflow
JNP/JPO	PF = 0	not parity/parity odd
JNS	SF = 0	not sign
JO	OF = 1	overflow
JP/JPE	PF = 1	parity/parity equal
JS	SF = 1	sign

Note: "above" and "below" refer to the relationship of two unsigned values;
"greater" and "less" refer to the relationship of two signed values.

FIGURE 4-10 8086 Conditional Jump instructions.

2

INTEL 8085

This chapter describes hardware, software, and interfacing aspects of the Intel 8085. Topics include 8085 register architecture, addressing modes, instruction set, input/output, and system design.

2.1 Introduction

The Intel 8085 is an 8-bit microprocessor. The 8085 is designed using NMOS in 40-in DIP (Dual In-line Package). The 8085 can be operated from either 3.03 MHz maximum (8085A) or 5 MHz maximum (8085A-2) internal clock frequency.

The 8085 has three enhanced versions, namely, the 8085AH, 8085AH-2, and 8085AH-1. These enhanced processors are designed using the HMOS (High-density MOS) technology. Each is packaged in a 40-pin DIP like the 8085. These enhanced microprocessors consume 20% lower power than the 8085A. The internal clock frequencies of the 8085AH, 8085AH-2, and 8085AH-1 are 3, 5, 6 MHz, respectively. These HMOS 8-bit microprocessors are expensive compared to the NMOS 8-bit 8085A.

Figure 2.1 shows a simplified block diagram of the 8085 microprocessor. The accumulator connects to the data bus and the Arithmetic and Logic Unit (ALU). The ALU performs all data manipulation, such as incrementing a number or adding two numbers.

The temporary register feeds the ALU's other input. This register is invisible to the programmer and is controlled automatically by the microprocessor's control circuitry.

The flags are a collection of flip-flops that indicate certain characteristics of the result of the most recent operation performed by the ALU. For example, the zero flag is set if the result of an operation is zero. The zero flag is tested by the JZ instruction.

The instruction register, instruction decoder, program counter, and control and timing logic are used for fetching instructions from memory and directing their execution.

2.2 Register Architecture

The 8085 registers and status flags are shown in Figure 2.2.

The accumulator (A) is an 8-bit register. Most arithmetic and logic operations are performed using the accumulator. All I/O data transfers between the 8085 and the I/O devices are performed via the accumulator. Also, there are a number of instructions that move data between the accumulator and memory.

The B, C, D, E, H, and L are each 8 bits long. Registers H and L are the memory address register or data counter. This means that these two registers are used to store the 16-bit address

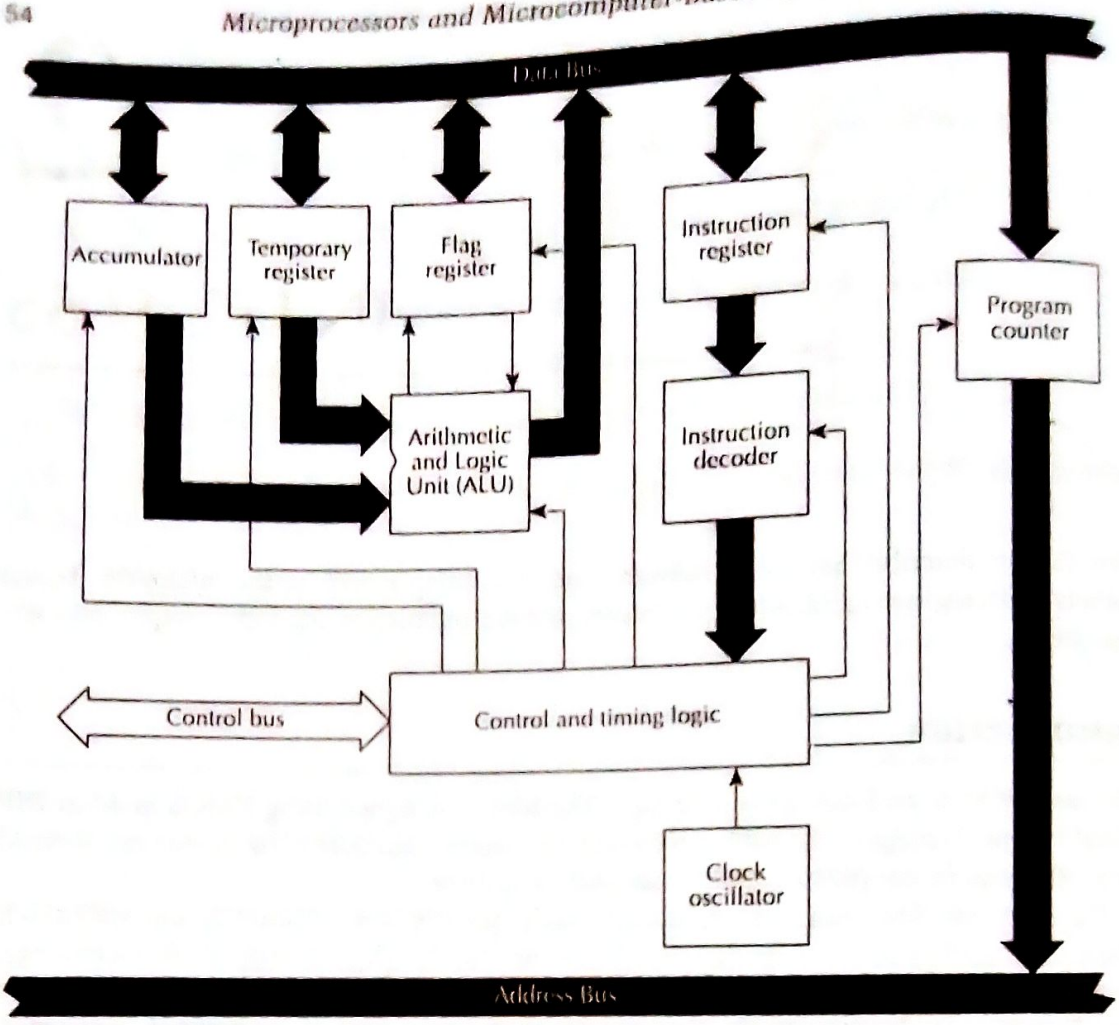


FIGURE 2.1 Simplified 8085 block diagram.

of 8-bit data being accessed from memory. This is the implied or register indirect addressing mode. There are a number of instructions, such as MOV reg, M, and MOV M, reg, which move data between any register and memory location addressed by H and L. However, using any other memory reference instruction, data transfer takes place between a memory location and the only 8085 register, the accumulator. The instruction LDAX B is a typical example.

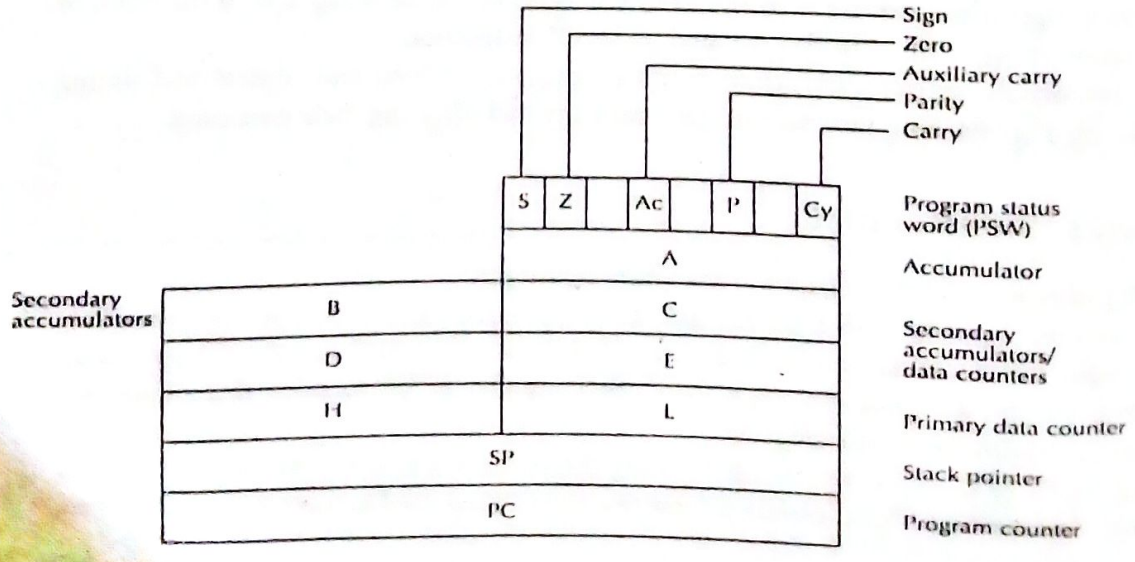


FIGURE 2.2 8085 microprocessor registers and status flags.

B, C, D, and E are secondary accumulators or data counters. There are a number of instructions to move data between any two registers. There are also a few instructions that combine registers B and C or D and E, as a 16-bit data counter with high byte of a pair contained in the first register and low byte in the second. These instructions typically include LDAX B, LDAX D, STAX B, and STAX D, which transfer data between memory and the accumulator.

Each of these 8-bit registers can be incremented and decremented by a single byte instruction. There are a number of instructions which combine two of these 8-bit registers to form 16-bit register pairs as follows:

A	and	PSW
B	and	C
D	and	E
H	and	L
high-order byte		low-order byte

The 16-bit register pair obtained by combining the accumulator and the program status word (PSW) is used only for stack operations. Sixteen bit arithmetic operations use B and C, D and E, or H and L as 16-bit data registers.

The **program status word** consists of five status flags. These are described below.

The **carry flag** (Cy) reflects the final carry out of the most significant bit of any arithmetic operation. Any logic instruction resets or clears the carry flag. This flag is also used by the shift and rotate instructions. The 8085 does not have any CLEAR CARRY instruction. One way of clearing the carry will be by ORing or ANDing the accumulator with itself.

The **parity status flag** (P) is set to 1 if an arithmetic or logic instruction generates an answer with even parity, that is, containing an even number of 1 bits. This flag is 0 if the arithmetic or logic instruction generates an answer with odd parity, that is, containing an odd number of 1s.

The **auxiliary carry flag** (Ac) reflects any carry from bit 3 to bit 4 (assuming 8-bit data with bit 0 as the least significant bit and bit 7 as the most significant bit) due to an arithmetic operation. This flag is useful for BCD operations.

The **zero flag** (Z) is set to 1 whenever an arithmetic or logic operation produces a result of 0. The zero flag is cleared to zero for a nonzero result due to arithmetic or logic operation.

The **sign status flag** (S) is set to the value of the most significant bit of the result in the accumulator after an arithmetic or logic operation. This provides a range of -128_{10} to $+127_{10}$ (with 0 being considered positive) as the 8085's data-handling capacity.

The 8085 does not have an overflow flag. Note that execution of arithmetic or logic instructions in the 8085 affects the flags. All conditional instructions in the 8085 instruction set use one of the status flags as the required condition.

The **stack pointer** (SP) is 16 bits long. All stack operations with the 8085 use 16-bit register pairs. The stack pointer contains the address of the last data byte written into the stack. It is decremented by 2 each time 2 bytes of data are written or pushed onto the stack and is incremented by 2 each time 2 bytes of data are read from or pulled (popped) off the stack, that is, the top of the stack has the lowest address in the stack that grows downward.

The **program counter** (PC) is 16 bits long to address up to 64K of memory. It usually addresses the next instruction to be executed.

2.3 Memory Addressing

When addressing a memory location, the 8085 uses either register indirect or direct memory addressing. With register indirect addressing, the H and L registers perform the function of the memory address register or data counter; that is, the H, L pair holds the address of the data.

With this mode, data transfer may occur between the addressed memory location and any one of the registers A, B, C, D, E, H, or L.

Also, some instructions, such as LDAX B, LDAX D, STAX B, and STAX D, use registers B and C or D and E to hold the address of data. These instructions transfer data between the accumulator and the memory location addressed by registers B and C or D and E using the register indirect mode.

There are also a few instructions, such as the STA ppqq, which use the direct-memory addressing mode to move data between the accumulator and the memory. These instructions use 3 bytes, with the first byte as the OP code followed by 2 bytes of address.

The stack is basically a part of the RAM. Therefore, PUSH and POP instructions are memory reference instructions.

All 8085 JUMP instructions use direct or absolute addressing and are 3 bytes long. The first byte of this instruction is the OP code followed by a 2-byte address. This address specifies the memory location to which the program would branch.

2.4 8085 Addressing Modes

The 8085 has five addressing modes:

1. **Direct** — Instructions using this mode specify the effective address as a part of the instruction. These instructions contain 3 bytes, with the first byte as the OP code followed by 2 bytes of address of data (the low-order byte of the address in byte 2, the high-order byte of the address in byte 3). Consider LDA 2035H. This instruction loads accumulator with the contents of memory location 2035_{16}. This mode is also called the absolute mode.
2. **Register** — This mode specifies the register or register pair that contains data. For example, MOV B, C moves the contents of register C to register B.
3. **Register Indirect** — This mode contains a register pair which stores the address of data (the high-order byte of the address in the first register of the pair, and the low-order byte in the second). As an example, LDAX B loads the accumulator with the contents of a memory location addressed by B, C register pair.
4. **Implied or Inherent** — The instructions using this mode have no operands. Examples include STC (Set the Carry Flag).
5. **Immediate** — For an 8-bit datum, this mode uses 2 bytes, with the first byte as the OP code, followed by 1 byte of data. On the other hand, for 16-bit data, this instruction contains 3 bytes, with the first byte as the OP code followed by 2 bytes of data. For example, MVI B, 05 loads register B with the value 5, and LXI H, 2050H loads H with 20H and L with 50H.

A JUMP instruction interprets the address that it would branch to in the following ways:

1. **Direct** — The JUMP instructions, such as JZ ppqq, use direct addressing and contain 3 bytes. The first byte is the OP code, followed by 2 bytes of the 16-bit address where it would branch to unconditionally or based on a condition if satisfied. For example, JMP 2020 unconditionally branches to location 2020H.
2. **Implied or Inherent Addressing** — This JUMP instruction using this mode is 1 byte long. A 16-bit register pair contains the address of the next instruction to be executed. The instruction PCHL unconditionally branches to a location addressed by the H, L pair.

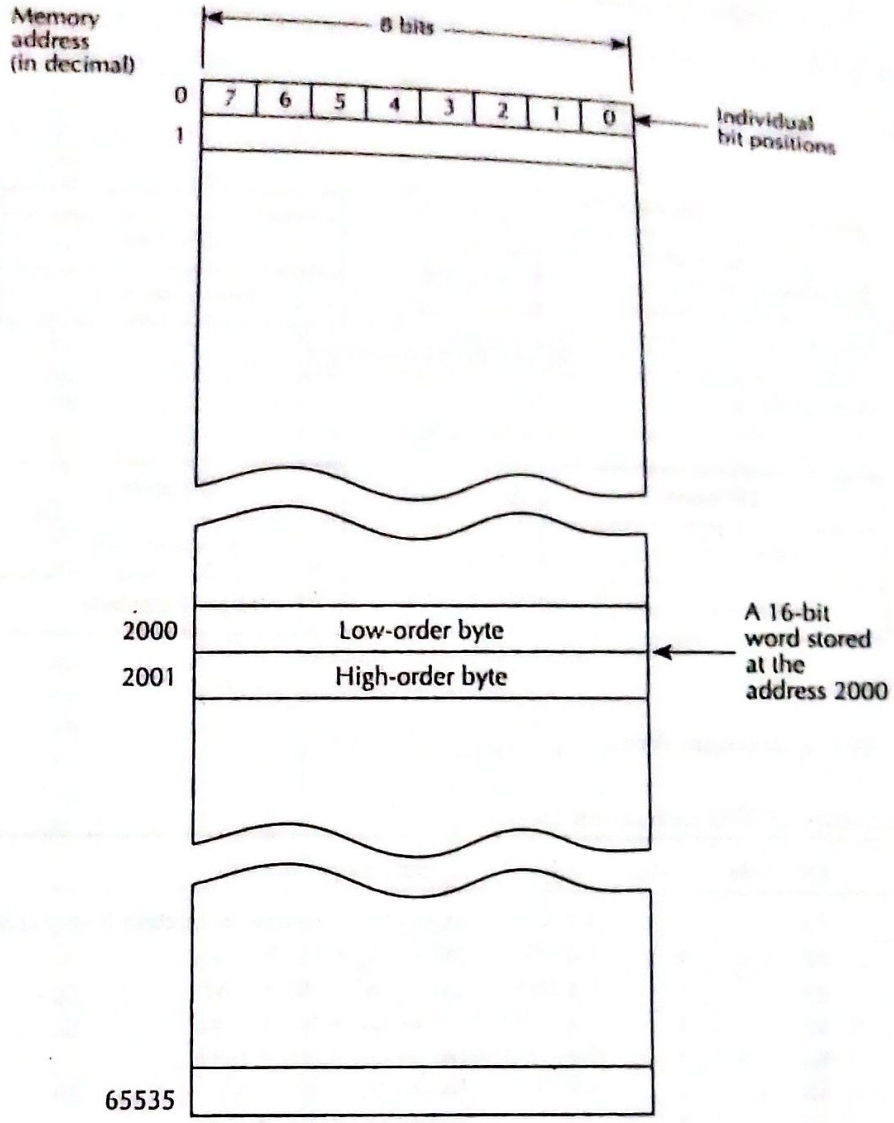


FIGURE 2.3 8085 addressing structures.

2.5 8085 Instruction Set

As mentioned before, the 8085 uses a 16-bit address. Since the 8085 is a byte-addressable machine, it follows that it can directly address 65,536 (2^{16}) distinct memory locations. The addressing structure of the 8085 processor is shown in Figure 2.3.

From this figure, we notice that two consecutive memory locations may be used to represent a 16-bit data item. However, according to the Intel convention, the high-order byte of a 16-bit quantity is always assigned to the high memory address.

The 8085 instructions are 1 to 3 bytes long and these formats are shown in Figure 2.4. The 8085 instruction set contains 74 basic instructions and supports conventional addressing modes such as immediate, register, absolute, and register indirect addressing modes.

Table 2.1 lists the 8085 instructions in alphabetical order; the object codes and instruction cycles are also included. When two instruction cycles are shown, the first is for "condition not met", while the second is for "condition met". Table 2.2 provides the 8085 instructions affecting the status flags. Note that not all 8085 instructions affect the status flags. The 8085 arithmetic and logic instructions normally affect the status flags.

In describing the 8085 instruction set, we will use the symbols in Table 2.3.

The 8085 move instruction transfers 8-bit data from one register to another, register to memory, and vice versa. A complete summary of these instructions is presented in Table 2.4.

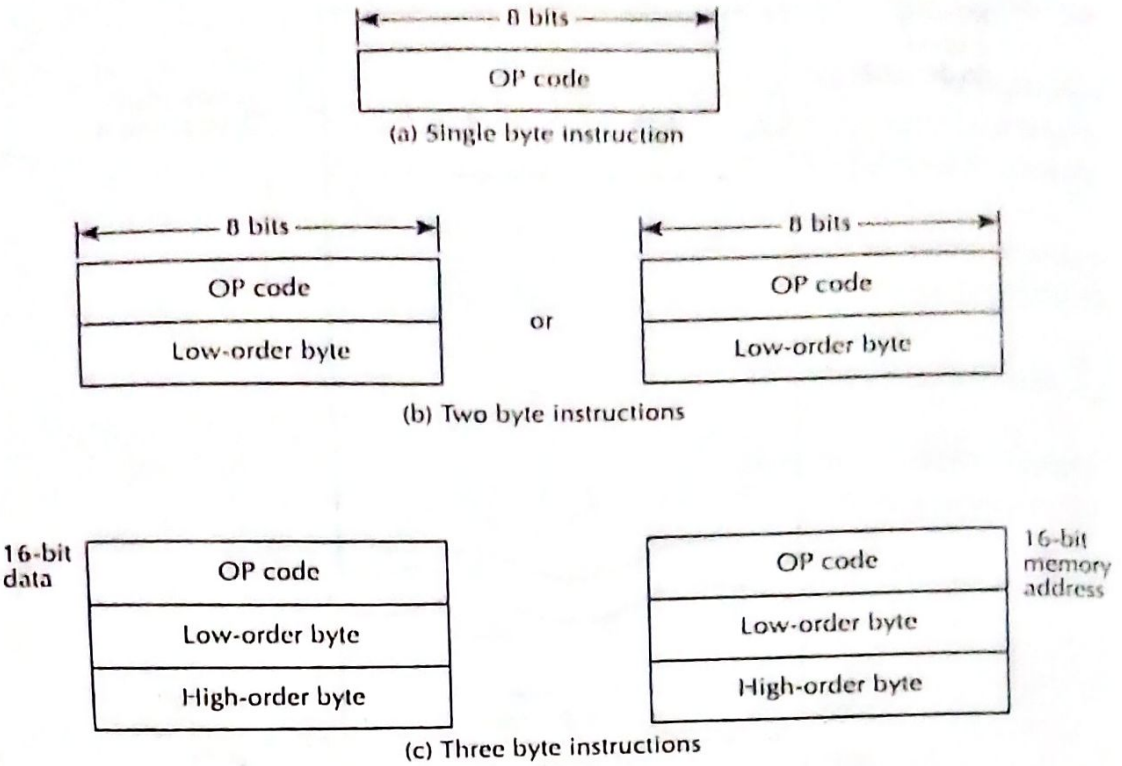


FIGURE 2.4 8085 instructions format.

Table 2.1 Summary of 8085 Instruction Set

Instruction	OP Code	Bytes	Cycles	Operations performed
ACI DATA	CE	2	7	[A] ← [A] + second instruction byte + [Cy]
ADC A	8F	1	4	[A] ←[A] + [A] + [Cy]
ADC B	88	1	4	[A] ←[A] + [B] + [Cy]
ADC C	89	1	4	[A] ← [A] + [C] + [Cy]
ADC D	8A	1	4	[A] ← [A] + [D] + [Cy]
ADC E	8B	1	4	[A] ← [A] + [E] + [Cy]
ADC H	8C	1	4	[A] ← [A] + [H] + [Cy]
ADC L	8D	1	4	[A] ← [A] + [L] + [Cy]
ADC M	8E	1	7	[A] ← [A] + [[H L]] + [Cy]
ADD A	87	1	4	[A] ← [A] + [A]
ADD B	80	1	4	[A] ← [A] + [B]
ADD C	81	1	4	[A] ← [A] + [C]
ADD D	82	1	4	[A] ← [A] + [D]
ADD E	83	1	4	[A] ← [A] + [E]
ADD H	84	1	4	[A] ← [A] + [H]
ADD L	85	1	4	[A] ← [A] + [L]
ADD M	86	1	7	[A] ← [A] + [[H L]]
ADI DATA	C6	2	7	[A] ← [A] + second instruction byte
ANA A	A7	1	4	[A] ← [A] ∧ [A]
ANA B	A0	1	4	[A] ← [A] ∧ [B]
ANA C	A1	1	4	[A] ← [A] ∧ [C]
ANA D	A2	1	4	[A] ← [A] ∧ [D]
ANA E	A3	1	4	[A] ← [A] ∧ [E]
ANA H	A4	1	4	[A] ← [A] ∧ [H]
ANA L	A5	1	4	[A] ← [A] ∧ [L]
ANA M	A6	1	4	[A] ← [A] ∧ [[H L]]
ANI DATA	E6	2	7	[A] ← [A] ∧ second instruction byte
CALL ppqq	CD	3	18	Call A subroutine addressed by ppqq
CC ppqq	DC	3	9/18	Call a subroutine addressed by ppqq if Cy = 1
CM ppqq	FC	3	9/18	Call a subroutine addressed by ppqq if S = 1
CMA	2F	1	4	[A] ← 1's complement of [A]
CMC	3F	1	4	[Cy] ← 1's complement of [Cy]
CMP A	BF	1	4	[A] – [A] and affects flags
CMP B	B8	1	4	[A] – [B] and affects flags

Table 2.1 Summary of 8085 Instruction Set (*continued*)

Instruction	OP Code	Bytes	Cycles	Operations performed
CMP C	B9	1	4	[A] – [C] and affects flags
CMP D	BA	1	4	[A] – [D] and affects flags
CMP E	BB	1	4	[A] – [E] and affects flags
CMP H	BC	1	4	[A] – [H] and affects flags
CMP L	BD	1	4	[A] – [L] and affects flags
CMP M	BE	1	7	[A] – [[H L]] and affects flags
CNC ppqq	D4	3	9/18	Call a subroutine addressed by ppqq if Cy = 0
CNZ ppqq	C4	3	9/18	Call a subroutine addressed by ppqq if Z = 0
CP ppqq	F4	3	9/18	Call a subroutine addressed by ppqq if S = 0
CPE ppqq	EC	3	9/18	Call a subroutine addressed by ppqq if P = 1
CPI DATA	FE	2	7	[A] – second instruction byte and affects flags
CPO ppqq	E4	3	9/18	Call a subroutine addressed by ppqq if P = 0
CZ ppqq	CC	3	9/18	Call a subroutine addressed by ppqq if Z = 1
DAA	27	1	4	Decimal adjust accumulator
DAD B	09	1	10	[HL] ← [HL] + [BC]
DAD D	19	1	10	[HL] ← [HL] + [DE]
DAD H	29	1	10	[HL] ← [HL] + [HL]
DAD SP	39	1	10	[HL] ← [HL] + [SP]
DCR A	3D	1	4	[A] ← [A] – 1
DCR B	05	1	4	[B] ← [B] – 1
DCR C	0D	1	4	[C] ← [C] – 1
DCR D	15	1	4	[D] ← [D] – 1
DCR E	1D	1	4	[E] ← [E] – 1
DCR H	25	1	4	[H] ← [H] – 1
DCR L	2D	1	4	[L] ← [L] – 1
DCR M	35	1	4	[[HL]] ← [[HL]] – 1
DCX B	0B	1	6	[BC] ← [BC] – 1
DCX D	1B	1	6	[DE] ← [DE] – 1
DCX H	2B	1	6	[HL] ← [HL] – 1
DCX SP	3B	1	6	[SP] ← [SP] – 1
DI	F3	1	4	Disable interrupts
EI	FB	1	4	Enable interrupts
HLT	76	1	5	Halt
IN PORT	DB	2	10	[A] ← [specified port]
INR A	3C	1	4	[A] ← [A] + 1
INR B	04	1	4	[B] ← [B] + 1
INR C	0C	1	4	[C] ← [C] + 1
INR D	14	1	4	[D] ← [D] + 1
INR E	1C	1	4	[E] ← [E] + 1
INR H	24	1	4	[H] ← [H] + 1
INR L	2C	1	4	[L] ← [L] + 1
INR M	34	1	4	[[HL]] ← [[HL]] +1
INX B	03	1	6	[BC] ← [BC] +1
INX D	13	1	6	[DE] ← [DE] + 1
INX H	23	1	6	[HL] ← [HL] + 1
INX SP	33	1	6	[SP] ← [SP] + 1
JC ppqq	DA	3	7/10	Jump to ppqq if Cy = 1
JM ppqq	FA	3	7/10	Jump to ppqq if S = 1
JMP ppqq	C3	3	10	Jump to ppqq
JNC ppqq	D2	3	7/10	Jump to ppqq if Cy = 0
JNZ ppqq	C2	3	7/10	Jump to ppqq if Z = 0
JP ppqq	F2	3	7/10	Jump to ppqq if S = 0
JPE ppqq	EA	3	7/10	Jump to ppqq if P = 1
JPO ppqq	E2	3	7/10	Jump to ppqq if P = 0
JZ ppqq	CA	3	7/10	Jump to ppqq if Z = 1
LDA ppqq	3A	3	13	[A] ← [ppqq]
LDAX B	0A	1	7	[A] ← [[BC]]
LDAX D	1A	1	7	[A] ← [[DE]]
	2A	3	16	[L] ← [ppqq],[H] ← [ppqq + 1]

Instruction	OP Code	Bytes	Cycles	Operations perfor
LXI B	01	3	10	[BC] ← second and third instruction bytes
LXI D	11	3	10	[DE] ← second and third instruction bytes
LXI H	21	3	10	[HL] ← second and third instruction bytes
LXI SP	31	3	10	[SP] ← second and third instruction bytes
MOV A,A	7F	1	4	[A] ← [A]
MOV A,B	78	1	4	[A] ← [B]
MOV A,C	79	1	4	[A] ← [C]
MOV A,D	7A	1	4	[A] ← [D]
MOV A,E	7B	1	4	[A] ← [E]
MOV A,H	7C	1	4	[A] ← [H]
MOV A,L	7D	1	4	[A] ← [L]
MOV A,M	7E	1	7	[A] ← [[HL]]
MOV B,A	47	1	4	[B] ← [A]
MOV B,B	40	1	4	[B] ← [B]
MOV B,C	41	1	4	[B] ← [C]
MOV B,D	42	1	4	[B] ← [D]
MOV B,E	43	1	4	[B] ← [E]
MOV B,H	44	1	4	[B] ← [H]
MOV B,L	45	1	4	[B] ← [L]
MOV B,M	46	1	7	[B] ← [[HL]]
MOV C,A	4F	1	4	[C] ← [A]
MOV C,B	48	1	4	[C] ← [B]
MOV C,C	49	1	4	[C] ← [C]
MOV C,D	4A	1	4	[C] ← [D]
MOV C,E	4B	1	4	[C] ← [E]
MOV C,H	4C	1	4	[C] ← [H]
MOV C,L	4D	1	4	[C] ← [L]
MOV C,M	4E	1	7	[C] ← [[HL]]
MOV D,A	57	1	4	[D] ← [A]
MOV D,B	50	1	4	[D] ← [B]
MOV D,C	51	1	4	[D] ← [C]
MOV D,D	52	1	4	[D] ← [D]
MOV D,E	53	1	4	[D] ← [E]
MOV D,H	54	1	4	[D] ← [H]
MOV D,L	55	1	4	[D] ← [L]
MOV D,M	56	1	7	[D] ← [[HL]]
MOV E,A	5F	1	4	[E] ← [A]
MOV E,B	58	1	5	[E] ← [B]
MOV E,C	59	1	4	[E] ← [C]
MOV E,D	5A	1	4	[E] ← [D]
MOV E,E	5B	1	4	[E] ← [E]
MOV E,H	5C	1	4	[E] ← [H]
MOV E,L	5D	1	4	[E] ← [L]
MOV E,M	5E	1	7	[E] ← [[HL]]
MOV H,A	67	1	4	[H] ← [B]
MOV H,B	60	1	4	[H] ← [A]
MOV H,C	61	1	4	[H] ← [C]
MOV H,D	62	1	4	[H] ← [D]
MOV H,E	63	1	4	[H] ← [E]
MOV H,H	64	1	4	[H] ← [H]
MOV H,L	65	1	4	[H] ← [L]
MOV H,M	66	1	7	[H] ← [[HL]]
MOV L,A	6F	1	4	[L] ← [A]
MOV L,B	68	1	4	[L] ← [B]
MOV L,C	69	1	4	[L] ← [C]
MOV L,D	6A	1	4	[L] ← [D]
MOV L,E	6B	1	4	[L] ← [E]
MOV L,H	6C	1	4	[L] ← [H]
MOV L,L	6D	1	4	[L] ← [L]

Table 2.1 Summary of 8085 Instruction Set (*continued*)

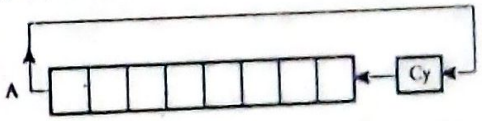
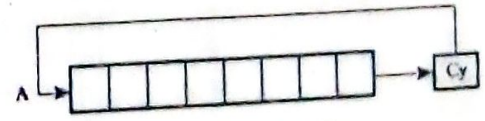
Instruction	OP Code	Bytes	Cycles	Operations performed
MOV L,M	6E	1	7	[L] ← [[HL]]
MOV M,A	77	1	7	[[HL]] ← [A]
MOV M,B	70	1	7	[[HL]] ← [B]
MOV M,C	71	1	7	[[HL]] ← [C]
MOV M,D	72	1	7	[[HL]] ← [D]
MOV M,E	73	1	7	[[HL]] ← [E]
MOV M,H	74	1	7	[[HL]] ← [H]
MOV M,L	75	1	7	[[HL]] ← [L]
MVI A, DATA	3E	2	7	[A] ← second instruction byte
MVI B, DATA	06	2	7	[B] ← second instruction byte
MVI C, DATA	0E	2	7	[C] ← second instruction byte
MVI D, DATA	16	2	7	[D] ← second instruction byte
MVI E, DATA	1E	2	7	[E] ← second instruction byte
MVI H, DATA	26	2	7	[H] ← second instruction byte
MVI L, DATA	2E	2	7	[L] ← second instruction byte
MVI M, DATA	36	2	10	[[HL]] ← second instruction byte
NOP	00	1	4	No operation
ORA A	B7	1	4	[A] ← [A] ∨ [A]
ORA B	B0	1	4	[A] ← [A] ∨ [B]
ORA C	B1	1	4	[A] ← [A] ∨ [C]
ORA D	B2	1	4	[A] ← [A] ∨ [D]
ORA E	B3	1	4	[A] ← [A] ∨ [E]
ORA H	B4	1	4	[A] ← [A] ∨ [H]
ORA L	B5	1	4	[A] ← [A] ∨ [L]
ORA M	B6	1	7	[A] ← [A] ∨ [[HL]]
ORI DATA	F6	2	7	[A] ← [A] ∨ second instruction byte
OUT PORT	D3	2	10	[specified port] ← [A]
PCHL	E9	1	6	[PCH]* ← [H], [PCL]* ← [L]
POP B	C1	1	10	[C] ← [[SP]], [SP] ← [SP] + 2 [B] ← [[SP] + 1]
POP D	D1	1	10	[E] ← [[SP]], [SP] ← [SP] + 2 [D] ← [[SP] + 1]
POP H	E1	1	10	[L] ← [[SP]], [SP] ← [SP] + 2 [H] ← [[SP] + 1]
POP PSW	F1	1	10	[A] ← [[SP] + 1], [PSW] ←[[SP]], [SP] ← [SP] + 2
PUSH B	C5	1	12	[[SP]– 1] ← [B], [SP] ← [SP] – 2 [[SP] – 2] ← [C]
PUSH D	D5	1	12	[[SP] – 1] ← [D], [[SP] – 2] ← [E] [SP] ← [SP] – 2
PUSH H	E5	1	12	[[SP] – 1] ← [H], [SP] ← [SP] – 2 [[SP] – 2] ← [L]
PUSH PSW	F5	1	12	[[SP] – 1] ← [A], [SP] ← [SP] – 2 [[SP] – 2] ← [PSW]
RAL	17	1	4	
RAR	1F	1	4	
RC	D8	1	6/12	Return if carry; [PC] ← [SP]]
RET	C9	1	10	[PCL]* ← [[SP]], [SP] ← [SP] + 2 [PCH]* ← [[SP] + 1]
RIM	20	1	4	Read interrupt mask
RLC	07	1	4	
RM	F8	1	6/12	Return if minus; [PC] ← [[SP]]

Table 2.1 Summary of 8085 Instruction Set (continued)

Instruction	OP Code	Bytes	Cycles	Operations performed
RNC	D0	1	6/12	Return if no carry; [PC] ← [[SP]]
RNZ	C0	1	6/12	Return if result not zero; [PC] ← [[SP]]
RP	F0	1	6/12	Return if positive; [PC] ← [[SP]], [SP] ← [SP] + 2
RPE	E8	1	6/12	Return if parity even; [PC] ← [[SP]], [SP] ← [SP] + 2
RPO	E0	1	6/12	Return if parity odd; [PC] ← [[SP]], [SP] ← [SP] + 2
RRC	0F	1	4	
RST0	C7	1	12	Restart
RST1	CF	1	12	Restart
RST2	D7	1	12	Restart
RST3	DF	1	12	Restart
RST4	E7	1	12	Restart
RST5	EF	1	12	Restart
RST6	F7	1	12	Restart
RST7	FF	1	12	Restart
RZ	C8	1	6/12	Return if zero; [PC] ← [[SP]]
SBB A	9F	1	4	[A] ← [A] − [A] − [Cy]
SBB B	98	1	4	[A] ← [A] − [B] − [Cy]
SBB C	99	1	4	[A] ← [A] − [C] − [Cy]
SBB D	9A	1	4	[A] ← [A] − [D] − [Cy]
SBB E	9B	1	4	[A] ← [A] − [E] − [Cy]
SBB H	9C	1	4	[A] ← [A] − [H] − [Cy]
SBB L	9D	1	4	[A] ← [A] − [L] − [Cy]
SBB M	9E	1	7	[A] ← [A] − [[HL]] − [Cy]
SBI DATA	DE	2	7	[A] ← [A] − second instruction byte − [Cy]
SHLD ppqq	22	3	16	[ppqq] ← [L], [ppqq + 1] ← [H]
SIM	30	1	4	Set interrupt mask
SPHL	F9	1	6	[SP] ← [HL]
STA ppqq	32	3	13	[ppqq] ← [A]
STAX B	02	1	7	[[BC]] ← [A]
STAX D	12	1	7	[[DE]] ← [A]
STC	37	1	4	[Cy] ← 1
SUB A	97	1	4	[A] ← [A] − [A]
SUB B	90	1	4	[A] ← [A] − [B]
SUB C	91	1	4	[A] ← [A] − [C]
SUB D	92	1	4	[A] ← [A] − [D]
SUB E	93	1	4	[A] ← [A] − [E]
SUB H	94	1	4	[A] ← [A] − [H]
SUB L	95	1	4	[A] ← [A] − [L]
SUB M	96	1	7	[A] ← [A] − [[HL]]
SUI DATA	D6	2	7	[A] ← [A] − second instruction byte
XCHG	EB	1	4	[D] ↔ [H], [E] ↔ [L]
XRA A	AF	1	4	[A] ← [A] ⊕ [A]
XRA B	A8	1	4	[A] ← [A] ⊕ [B]
XRA C	A9	1	4	[A] ← [A] ⊕ [C]
XRA D	AA	1	4	[A] ← [A] ⊕ [D]
XRA E	AB	1	4	[A] ← [A] ⊕ [E]
XRA H	AC	1	4	[A] ← [A] ⊕ [H]
XRA L	AD	1	4	[A] ← [A] ⊕ [L]
XRA M	AE	1	7	[A] ← [A] ⊕ [[HL]]
XRI DATA	EE	2	7	[A] ← [A] ⊕ second instruction byte
XTHL	E3	1	16	[[SP]] ↔ [L], [[SP] + 1]] ↔ [H]

*PCL — program counter low byte; PCH — program counter high byte.

Table 2.2 8085 Instructions Affecting the Status Flags

Instructions[a]	Status flags[b]				
	CY	AC	Z	S	P
ACI DATA	+	+	+	+	+
ADC reg	+	+	+	+	+
ADC M	+	+	+	+	+
ADD reg	+	+	+	+	+
ADD M	+	+	+	+	+
ADI DATA	+	+	+	+	+
ANA reg	0	1	+	+	+
ANA M	0	1	+	+	+
ANI DATA	0	1	+	+	+
CMC	+				
CMP reg	+	+	+	+	+
CMP M	+	+	+	+	+
CPI DATA	+	+	+	+	+
DAA	+	+	+	+	+
DAD rp	+				
DCR reg		+	+	+	+
DCR M		+	+	+	+
INR reg		+	+	+	+
INR M		+	+	+	+
ORA reg	0	0	+	+	+
ORA M	0	0	+	+	+
ORI DATA	0	0	+	+	+
RAL	+				
RAR	+				
RLC	+				
RRC	+				
SBB reg	+	+	+	+	+
SBB M	+	+	+	+	+
SBI DATA	+	+	+	+	+
STC	+				
SUB reg	+	+	+	+	+
SUB M	+	+	+	+	+
SUI DATA	+	+	+	+	+
XRA reg	0	0	+	+	+
XRA M	0	0	+	+	+
XRI DATA	0	0	+	+	+

[a]reg — 8-bit register; M — memory; rp — 16-bit register pair.

[b]Note that instructions which are not shown in the table do not affect the flags; + indicates that the particular flag is affected; 0 or 1 indicates that these flags are always 0 or 1 after the corresponding instructions are executed.

Table 2.3 Symbols to be Used in 8085 Instruction Set

Symbol	Interpretation
r1, r2	8-bit register
rp	Register pair
data8	8-bit data
data16	16-bit data
M	Memory location indirectly addressed through the register pair H,L
addr16	16-bit memory address

Table 2.4 8085 MOVE Instructions

Instruction	Symbolic description	Addressing mode		Illustration	
		Source	Destination	Example	Comments
MOV r1, r2	(r1) ← (r2)	Register	Register	MOV A, B	Copy the contents of the register B into the register A
MOV r, M	(r) ← M((HL))	Register indirect	Register	MOV A, M	Copy the contents of the memory location whose address is specified in the register pair H,L into the A register
MVI r, data8	(r) ← data8	Immediate	Register	MVI A, 08	Initialize the A register with the value 08
MOV M, r	M((HL)) ← (r)	Register	Register indirect	MOV M, B	Copy the contents of the B register into the memory location addressed by H,L pair
MVI M, data8	M((HL)) ← data8	Immediate	Register indirect	MVI M, 07	Initialize the memory location whose address is specified in the register pair H,L with the value 07

...quired. Now the program for the 5-s delay can be

```
        LXI  SP,  5000H   ;  Set stack pointer
        MVI  C,   32H     ;  Do DELAY loop 50(10) times by loading C
START   LXI  D,   30D3H   ;  with count 32(16)
        CALL DELAY        ;  Load initial count
        DCR  C            ;  Call DELAY loop
                          ;  Decrement C and check if zero: if
        JNZ  START        ;  not, do another delay
        HLT               ;  Loop back
                          ;  STOP
```

In the above, since execution times of DCR C and JNZ START are very small compared to 5 s, they are not considered in computing the delay.

2.7 8085 Pins and Signals

The 8085 is housed in a 40-pin dual in-line package (DIP). Figure 2.8 shows the 8085 pins and signals.

The low-order address byte and data lines AD0 to AD7 are multiplexed. These lines are bidirectional. The beginning of an instruction is indicated by the rising edge of the ALE signal. At the falling edge of ALE, the low byte of the address is automatically latched by some of the 8085 support chips such as 8155 and 8355: AD0 to AD7 lines can then be used as data lines. Note that ALE is an input to these support chips. However, if the support chips do not latch AD0 to AD7, then external latches are required to generate eight separate address lines A7 to A0 at the falling edge of ALE.

Pins A8 to A15 are unidirectional and contain the high byte of the address.

Table 2.11 lists the 8085 pins along with a brief description of each.

The $\overline{RD}$ pin signal is output LOW by the 8085 during a memory or I/O READ operation. Similarly, the $\overline{WR}$ pin signal is output LOW during a memory or I/O WRITE.

Next, we explain the purpose of IO/$\overline{M}$, S0, and S1 signals. The IO/$\overline{M}$ signal is output HIGH by the 8085 to indicate execution of an I/O instruction such as IN or OUT. This pin is output LOW during execution of a memory instruction such as LDA 2050H.

The IO/$\overline{M}$, S0, and S1 are output by the 8085 during its internal operations, which can be interpreted as follows:

IO/$\overline{M}$	S1	S0	Operation performed by the 8085
0	0	1	Memory WRITE
0	1	0	Memory READ
1	0	1	I/O WRITE
1	1	0	I/O READ
0	1	1	OP code fetch
1	1	1	Interrupt acknowledge

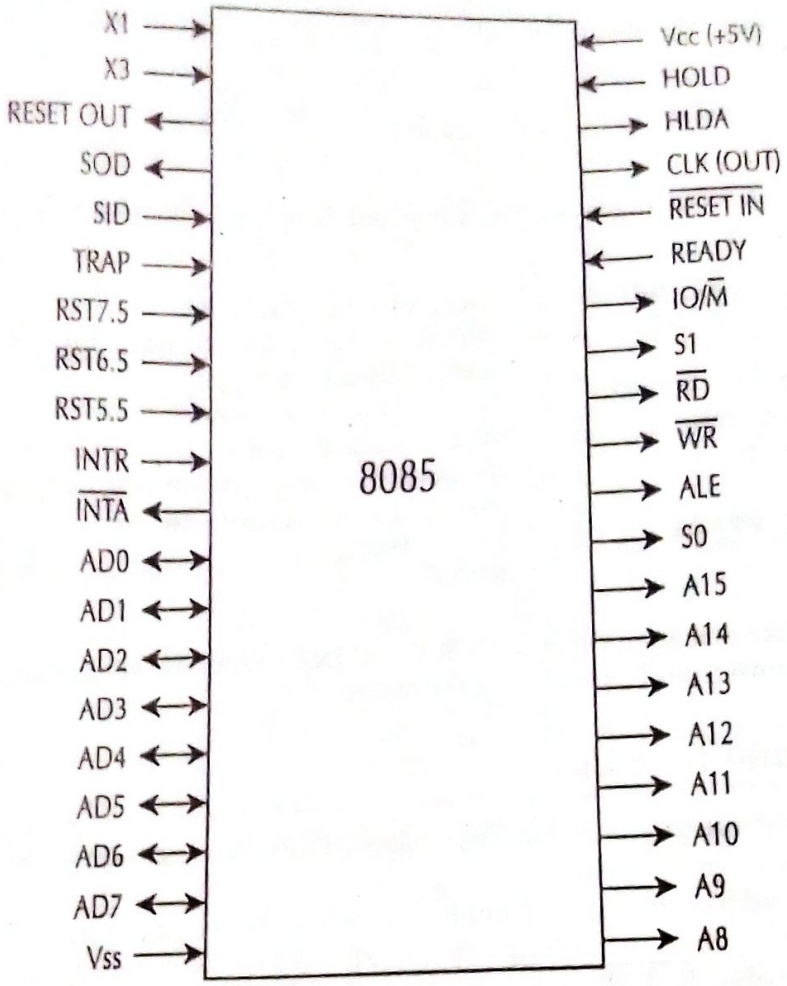
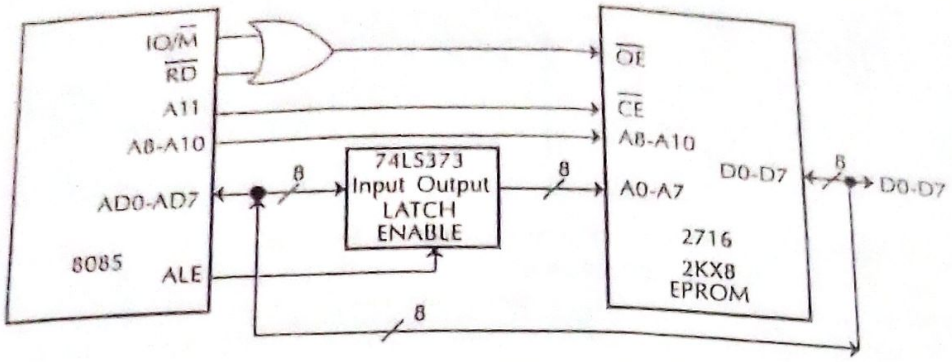


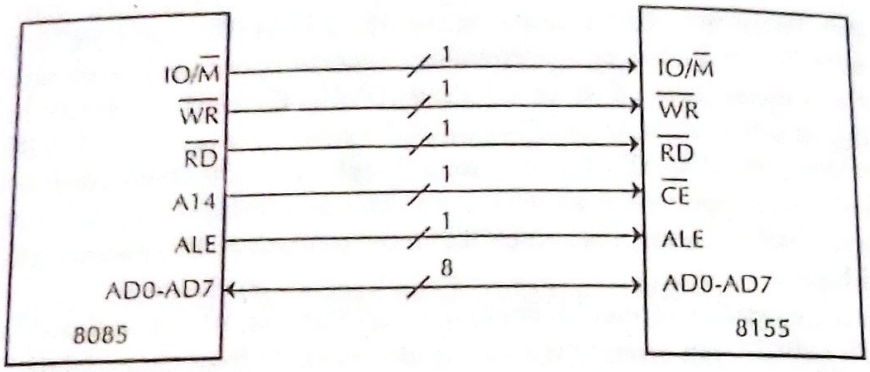
FIGURE 2.8 8085 microprocessor signals and pin assignments.

Table 2.11 8085 Signal Description Summary

Pin name	Description	Type
AD0-AD7	Address/data bus	Bidirectional, tristate
A8-A15	Address bus	Output, tristate
ALE	Address latch enable	Output, tristate
RD	Read control	Output, tristate
WR	Write control	Output, tristate
IO / M	I/O or memory indicator	Output, tristate
S0, S1	Bus state indicators	Output
READY	Wait state request	Input
SID	Serial data input	Input
SOD	Serial data output	Output
HOLD	Hold request	Input
HLDA	Hold acknowledge	Output
INTR	Interrupt request	Input
TRAP	Nonmaskable interrupt request	Input
RST5.5	Hardware vectored	Input
RST6.5	Hardware vectored interrupt request	Input
RST7.5	Hardware vectored	Input
INTA	Interrupt acknowledge	Output
RESET IN	System reset	Input
RESET OUT	Peripherals reset	Output
X1, X2	Crystal or RC connection	Input
CLK (OUT)	Clock signal	Output
Vcc, Vss	Power, ground	



(a) 8155 - 2716 interface using internal latches.



(b) 8085 - 8155 interface using ALE and AD0-AD7

FIGURE 2.9 8085's interface to external device using ALE and the multiplexed AD0 to AD7 pins.

Figure 2.9 illustrates the utilization of ALE and AD0 to AD7 signals for interfacing an EPROM and a RAM.

The 2716 is a 2K × 8 EPROM with separate address and data lines without any built-in latches. This means that a separate latch such as the 74LS373 must be used to isolate the 8085 low byte address and D0-D7 data lines at the falling edge of ALE (Figure 2.9a).

The 8155 contains 256-byte static RAM, three user ports, and a 14-bit timer. The 8155 is designed for 8085 in the sense that it has built-in latches with ALE as input along with multiplexed address (low byte) and data lines, AD0 to AD7. Therefore, as shown in Figure 2.9b, external latches are not required.

The READY input can be used by the slower external devices for obtaining extra time in order to communicate with the 8085. The READY signal (when LOW) can be utilized to provide wait-state clock periods in the 8085 machine. If READY is HIGH during a read or write cycle, it indicates that the memory or peripheral is ready to send or receive data. If not used, it must be tied high.

The Serial Input Data (SID) and Serial Output Data (SOD) lines are associated with the 8085 serial I/O transfer. The SOD line can be used to output the most significant bit of the accumulator. The SID signal can be input into the most significant bit of the accumulator.

The HOLD and HLDA signals are used for the Direct Memory Access (DMA) type of data transfer. The external devices place a HIGH on HOLD line in order to take control of the system bus. The HOLD function is acknowledged by the 8085 by placing a HIGH output on the HLDA pin and driving the tristate outputs high impedance.

The signals on the TRAP, RST7.5, RST6.5, RST5.5, INTR, and $\overline{\text{INTA}}$ are related to the 8085 interrupt signals. TRAP is a nonmaskable interrupt; that is, it cannot be enabled or disabled

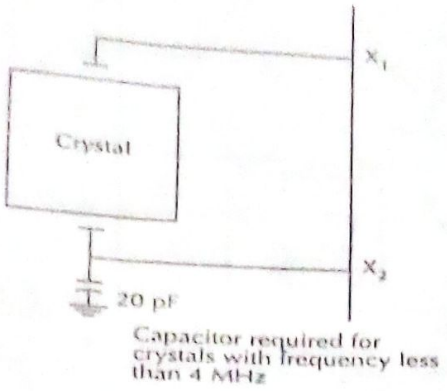


FIGURE 2.10 Crystal connection to X1 and X2 pins.

by an instruction. The TRAP has the highest priority. RST7.5, RST6.5, and RST5.5 are maskable interrupts used by the devices whose vector addresses are generated automatically. $\overline{\text{INTA}}$ is an interrupt acknowledge signal which is pulsed LOW by the 8085 in response to the interrupt INTR request. In order to service INTR, one of the eight OP codes (RST0 to RST7) has to be provided on the 8085 AD0-AD7 bus by external logic. The 8085 then executes this instruction and vectors to the appropriate address to service the interrupt.

All unused control pins such as interrupts and HOLD must be disabled by grounding them. (READY must be tied high).

The 8085 has the clock generation circuit on the chip and, therefore, no external oscillators need to be designed. The 8085A can operate with a maximum clock frequency of 3.03 MHz and the 8085A-2 can be driven with a maximum of 5 MHz clock. The 8085 clock frequency can be generated by a crystal, an LC tuned circuit, or an external clock circuit. The frequency at X1X2 is divided by 2 internally. This means that in order to obtain 3.03 MHz, a clock source of 6.06 MHz must be connected to X1X2. For crystals of less than 4 MHz, a capacitor of 20 pF should be connected between X2 and a ground to ensure the starting up of the crystal at the right frequency (Figure 2.10).

There is a TTL signal which is output on pin 37, called the CLK (OUT) signal. This signal can be used by other external microprocessors or support chips.

The $\overline{\text{RESET IN}}$ signal, when pulsed LOW then high, causes the 8085 to execute the first instruction at the 0000_{16} location. In addition, the 8085 resets instruction register, interrupt mask (RST5.5, RST6.5, and RST7.5) bits, and other registers. The $\overline{\text{RESET IN}}$ must be held LOW for at least three clock periods. A typical 8085 reset circuit is shown in Figure 2.11. In this circuit, when the switch is activated, $\overline{\text{RESET IN}}$ is driven to LOW with a large time constant providing adequate time to reset the system.

The 8085 requires a minimum operating voltage of 4.75 V. Upon applying power, the 8085A attains this voltage after 500 μs. The reset circuit of Figure 2.11 resets the 8085 upon activation

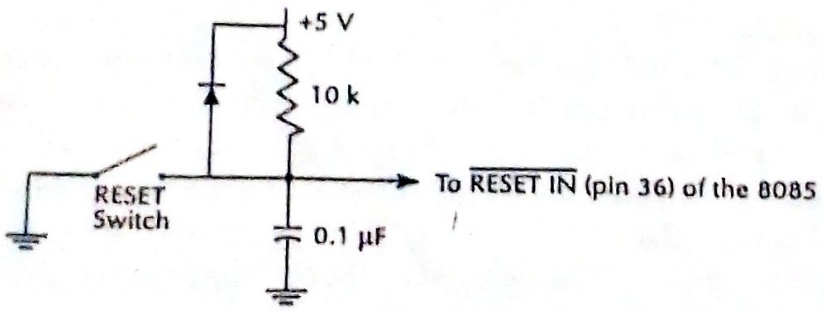


FIGURE 2.11 8085 reset circuit.

FIGURE 2.12 8085 machine cycles.

of the switch. The voltage across the 0.1-μF capacitor is zero on power-up. The capacitor then charges Vcc after a definite time determined by the time constant RC. The chosen values of RC in the figure will drive the $\overline{\text{RESET IN}}$ pin to low for at least three clock periods. In this case, after activating the switch, $\overline{\text{RESET IN}}$ will be low (assuming capacitance charge time is equal to the discharge time) for 10K*0.1 μF = 1 ms, which is greater than three clock periods (3*1/3 μs = 1 μs) of the 3-MHz 8085A. During normal operation of the 8085, activation of the switch will short the capacitor to ground and will discharge it. When the switch is opened, the capacitor charges and the $\overline{\text{RESET IN}}$ pin becomes HIGH. Upon hardware reset, the 8085 clears PC, IR, HALT flip-flop, and some other registers; the 8085 registers PSW, A, B, C, D, E, H, and L are unaffected. Upon activation of the $\overline{\text{RESET IN}}$ to low, the 8085 outputs HIGH at the RESET OUT pin which can be used to reset the memory and I/O chips connected to the 8085. Note that since hardware reset initializes PC to 0, the 8085 fetches the first instruction for address 0000_{16} after reset.

2.8 8085 Instruction Timing and Execution

An 8085's instruction execution consists of a number of machine cycles (MCs). These cycles vary from one to five (M1 to M5) depending on the instruction. Each machine cycle contains a number of 320-ns clock periods. The first machine cycle will be executed in either four or six clock periods, and the machine cycles that follow will have three clock periods. This is shown in Figure 2.12.

The shaded MCs indicate that these machine cycles are required by certain instructions. Similarly, the shaded clock periods (T5 and T6) mean that they are needed in M1 by some instructions.

The clock periods within a machine cycle can be illustrated as shown in Figure 2.13. Note that the beginning of a new machine cycle is indicated on the 8085 by outputting the Address Latch Enable (ALE) signal HIGH. During this time, lines AD0 to AD7 are used for placing the low byte of the address.

When the ALE signal goes LOW, the low byte of the address is latched so that the AD0 to AD7 lines can be used for transferring data.

We now discuss the timing diagrams for instruction fetch, READ, and WRITE.

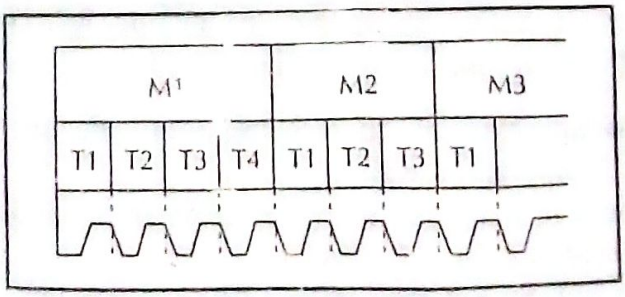


FIGURE 2.13 Clock period within a machine cycle.

Figure 2.14 shows the 8085 basic system timing. An instruction execution includes two operations: OP code fetch and execution.

The OP code fetch cycle requires either four (for one-byte instructions such as MOV A,B) or six cycles (for 3 byte instructions such as LDA 2030H). The machine cycles that follow will need three clock periods.

The purpose of an instruction fetch is to read the contents of a memory location containing an instruction addressed by the program counter and to place it in the instruction register. The 8085 instruction fetch timing diagram shown in Figure 2.14 can be explained in the following way:

1. The 8085 puts a LOW on the $IO/\overline{M}$ line of the system bus, indicating a memory operation.
2. The 8085 sets S0 = 1 and S1 = 1 on the system bus, indicating the memory fetch operation.
3. The 8085 places the program counter high byte on the A8 to A15 lines and the program counter low byte on the AD0 to AD7 lines of the system bus. The 8085 also sets the ALE signal to HIGH. As soon as the ALE signal goes to LOW, the program counter low byte on the AD0 to AD7 is latched automatically by some 8085 support chips such as 8155 (if 8085 support chips are not used, these lines must be latched using external latches), since these lines will be used as data lines for reading the OP code.
4. At the beginning of T2 in M1, the 8085 puts the $\overline{RD}$ line to LOW indicating a READ operation. After some time, the 8085 loads the OP code (the contents of the memory location addressed by the program counter) into the instruction register.
5. During the T4 clock period in M1, the 8085 decodes the instruction.

The Machine Cycle M2 of Figure 2.14 shows a memory (or I/O) READ operation as seen by the external logic, and the status of the S0 and S1 signals indicates whether the operation is instruction fetch or memory READ; for example, S1 = 1, S0 = 1 during instruction fetch and S1 = 1, S0 = 0 during memory READ provided $IO/\overline{M} = 0$.

The purpose of the memory READ is to read the contents of a memory location addressed by a register pair, such as the H,L pair, or a memory location specified with the instruction and the data placed in a microprocessor register such as the accumulator. In contrast, the purpose of the memory fetch is to read the contents of a memory location addressed by PC into IR. The machine cycle M3 of Figure 2.14 indicates a memory (or I/O) write operation. In this case, S1 = 0 and S0 = 1 indicate a memory write operation when $IO/\overline{M} = 0$ and an I/O write operation when $IO/\overline{M} = 1$.

2.8.2 8085 Memory READ ($IO/\overline{M} = 0$, $\overline{RD} = 0$) and I/O READ ($IO/\overline{M} = 1$, $\overline{RD} = 0$)

Figure 2.15a shows an 8085A clock timing diagram. The machine cycle of M2 of Figure 2.14 shows a memory READ timing diagram.

The purpose of the memory READ is to read the contents of a memory location addressed by a register pair, such as HL. Let us explain the 8085 memory READ timing diagram of Figure 2.15b along with the READ timing signals of Figure 2.14:

1. The 8085 uses machine cycle M1 to fetch and decode the instruction. It then performs the memory READ operation in M2.
2. The 8085 continues to maintain $IO/\overline{M}$ at LOW in M2 indicating a memory READ operation (or $IO/\overline{M} = 1$ for I/O READ).
3. The 8085 puts S1 = 1, S0 = 0, indicating a READ operation.
4. The 8085 places the contents of the high byte of the memory address register, such as the contents of the H register, on lines A8 to A15.

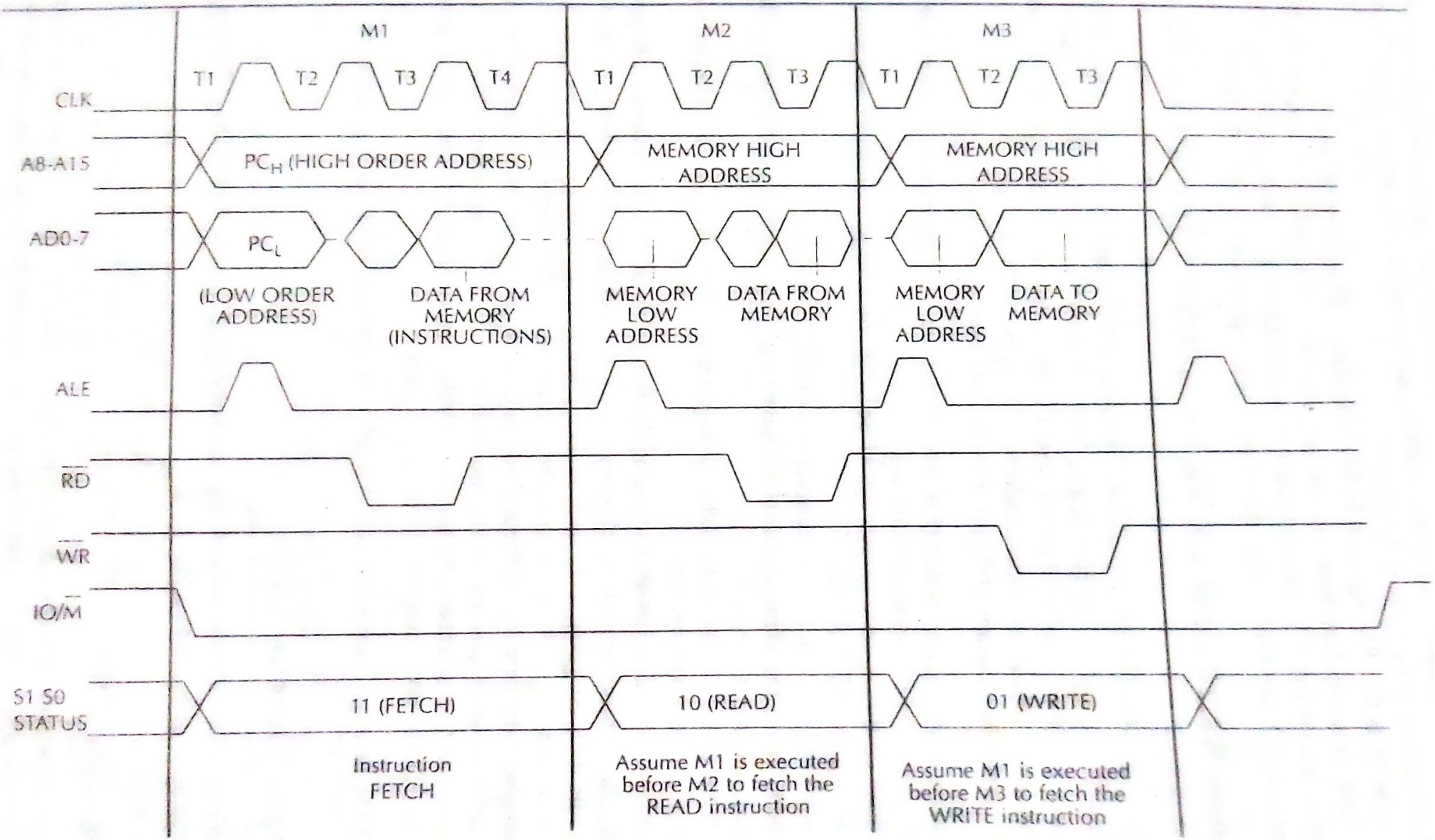


FIGURE 2.14 8085 Basic system timing.

5. The 8085 places the contents of the low byte of the memory address register, such as the contents of the L register, in lines AD0 to AD7.
6. The 8085 sets ALE to high, indicating the beginning of M2. As soon as ALE goes to low, the memory or support chip must latch the low byte of the address lines, since the same lines are going to be used as data lines.
7. The 8085 puts the $\overline{RD}$ signal to LOW, indicating a READ operation.
8. The 8085 gets the data from the memory location addressed by the memory address register, such as the H,L pair, and places the data into a register such as the accumulator. In case of I/O, the 8085 inputs data from the I/O port into the accumulator.

2.8.3 8085 Memory WRITE ($IO/\overline{M} = 0$, $\overline{WR} = 0$) and I/O WRITE ($IO/\overline{M} = 1$, $\overline{WR} = 0$)

The machine cycle M3 of Figure 2.14 shows a memory WRITE timing diagram. As seen by the external logic, the signals S1 = 0, S0 = 1, and $\overline{WR} = 0$ indicate a memory WRITE operation.

The purpose of a memory WRITE is to store the contents of the 8085 register, such as the accumulator, into a memory location addressed by a pair, such as H,L.

The WRITE timing diagram of Figure 2.14 can be explained as follows:

1. The 8085 uses machine cycle M1 to fetch and decode the instruction. It then executes the memory WRITE instruction in M3.
2. The 8085 continues to maintain $IO/\overline{M}$ at LOW, indicating a memory operation (or $IO/\overline{M} = 1$ for I/O WRITE).
3. The 8085 puts S1 = 0, S0 = 1, indicating a WRITE operation.
4. The 8085 places the Memory Address Register high byte, such as the contents of the H register, on lines A8 to A15.
5. The 8085 places the Memory Address Register low byte, such as the contents of L register, on lines AD0 to AD7.
6. The 8085 sets ALE to HIGH, indicating the beginning of M3. As soon as ALE goes to LOW, the memory or support chip must latch the low byte of the address lines, since the same lines are going to be used as data lines.
7. The 8085 puts the $\overline{WR}$ signal to LOW, indicating a WRITE operation.
8. It also places the contents of the register, say, accumulator, on data lines AD0 to AD7.
9. The external logic gets data from the lines AD0 to AD7 and stores the data in the memory location addressed by the Memory Address Register, such as the H,L pair. In case of I/O, the 8085 outputs [A] to an I/O port.

Figures 2.15a through c show the 8085A clock and read and write timing diagrams.

2.9 8085 Input/Output (I/O)

The 8085 I/O transfer techniques are discussed. The 8355/8755 and 8155/8156 I/O ports and 8085 SID and SOD lines are also included.

2.9.1 8085 Programmed I/O

There are two I/O instructions in the 8085, namely, IN and OUT. These instructions are 2 bytes long. The first byte defines the OP code of the instruction and the second byte specifies the I/O port number. Execution of the IN PORT instruction causes the 8085 to receive one byte of data into the accumulator from a specified I/O port. On the other hand, the OUT PORT instruction, when executed, causes the 8085 to send one byte of data from the accumulator into a specified I/O port.

FIGURE 2.15a 8085A clock.

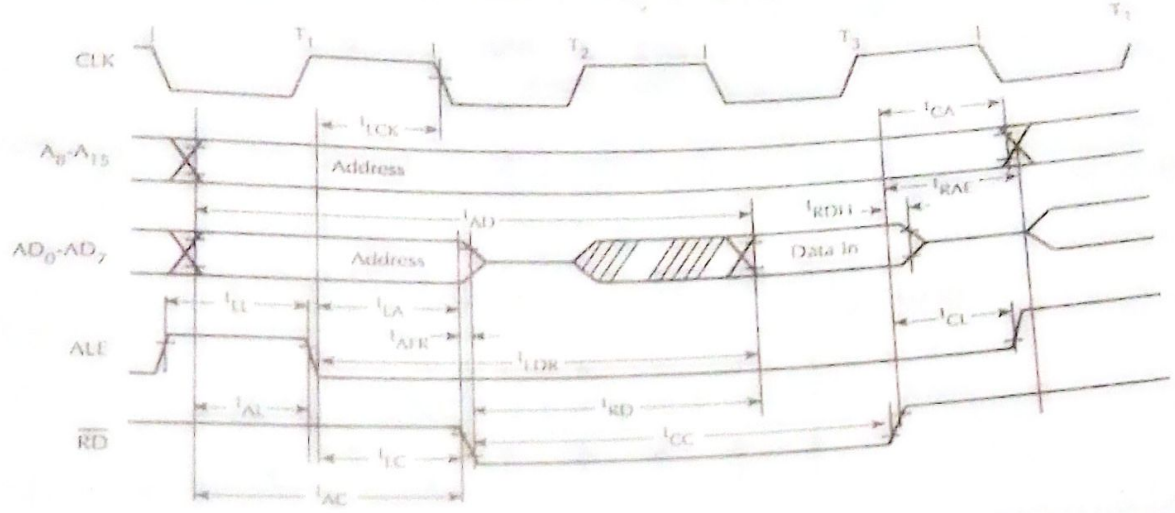


FIGURE 2.15b 8085 Read Timing diagram.

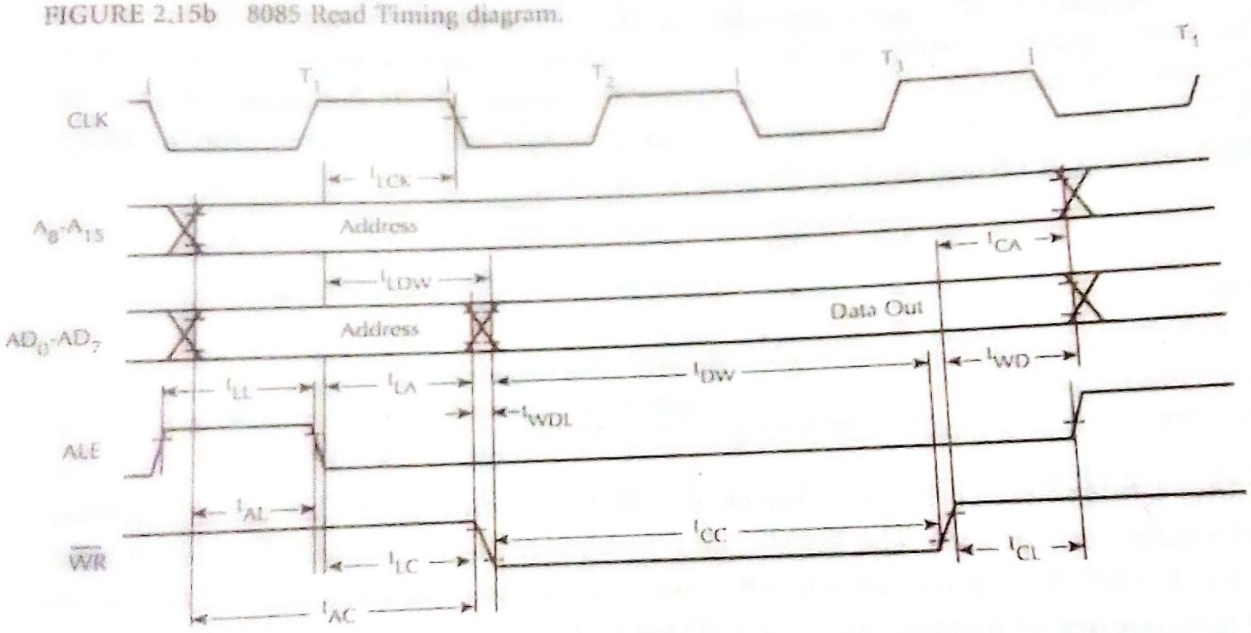


FIGURE 2.15c 8085 Write Timing diagram.

The 8085 can access I/O ports using either standard I/O or memory-mapped I/O.

In standard I/O, the 8085 inputs or outputs data using IN or OUT instructions.

In memory-mapped I/O, the 8085 maps I/O ports as memory addresses. Hence, LDA addr or STA addr instructions are used to input or output data to or from the 8085. The 8085's programmed I/O capabilities are obtained via the support chips, namely, 8355/8755 and 8155/8156. The 8355/8755 contains a 2K-byte ROM/EPROM and two 8-bit I/O ports (ports A and B).

The 8155/8156 contains 256-byte RAM, two 8-bit and one 6-bit I/O ports, and a 14-bit programmable timer. The only difference between the 8155 and 8156 is that chip enable is LOW on the 8155 and HIGH on the 8156.