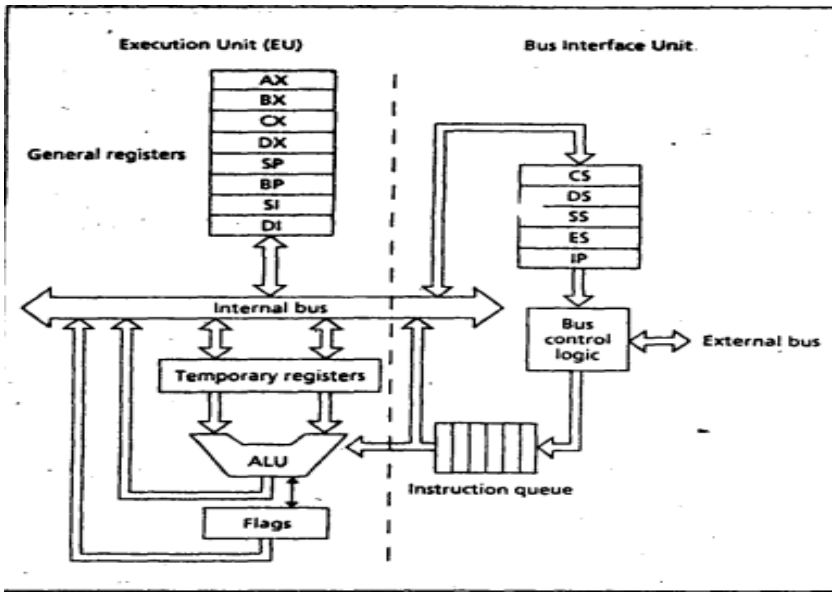
MID PART

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
```

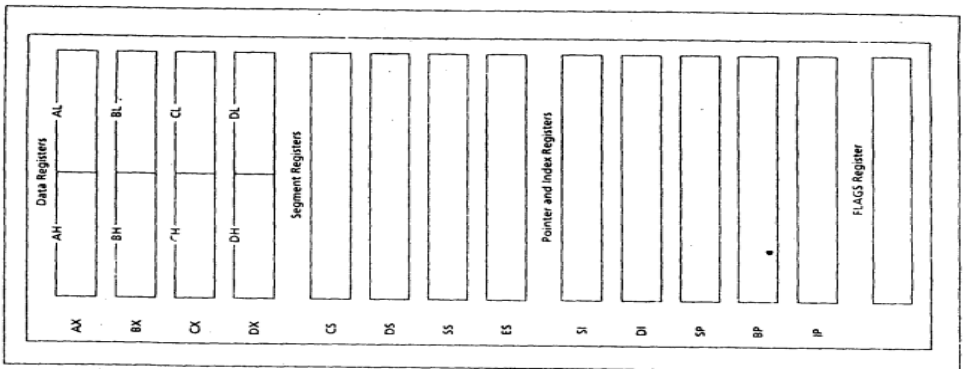
A one is carried out of the most significant bit and is lost. The answer stored, 38C0h, is correct, as may be verified by hex subtraction.

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
```

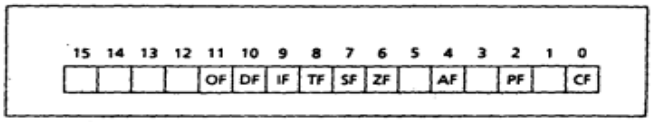
Because no variables were used, the data segment was omitted.

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:

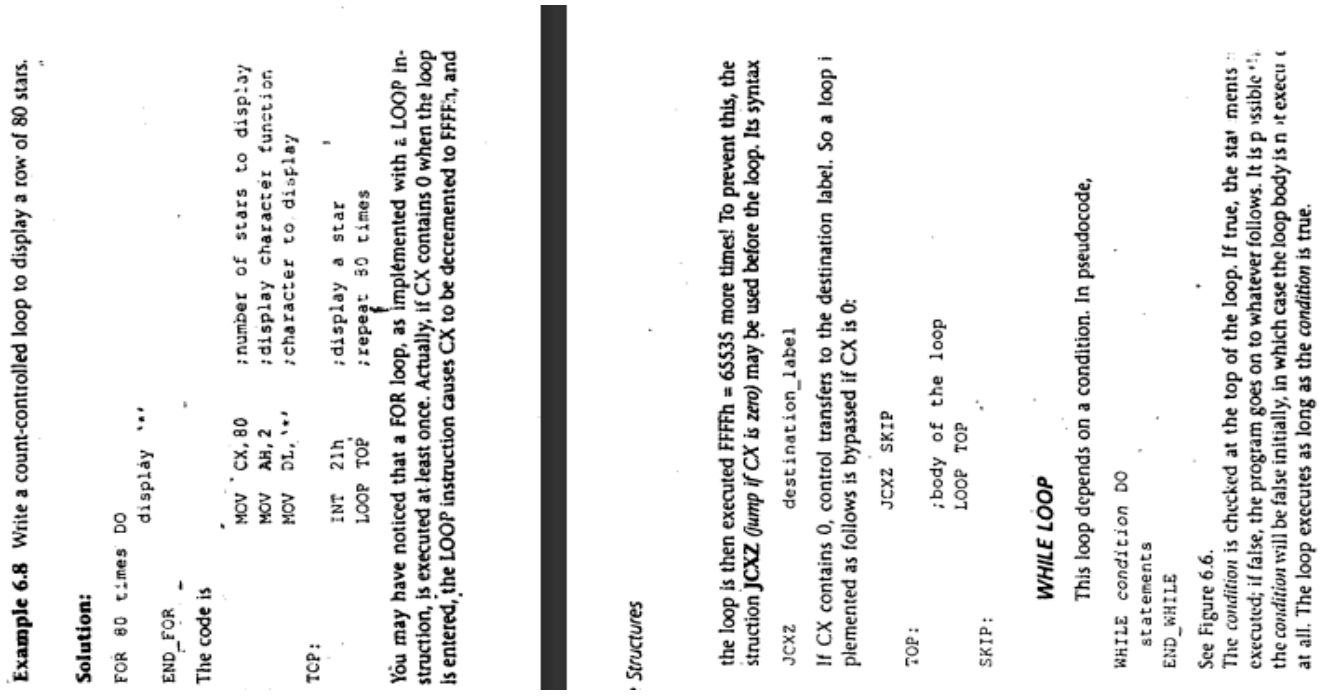
```
                MOV  AH,2       ;prepare to display
;if AL <= BL
                CMP  AL,BL      ;AL <= BL?
                JNBE  ELSE_     ;no, display char in BL
;then                           ;AL <= BL
                MOV DL,AL       ;move char to be displayed
                JMP DISPLAY    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

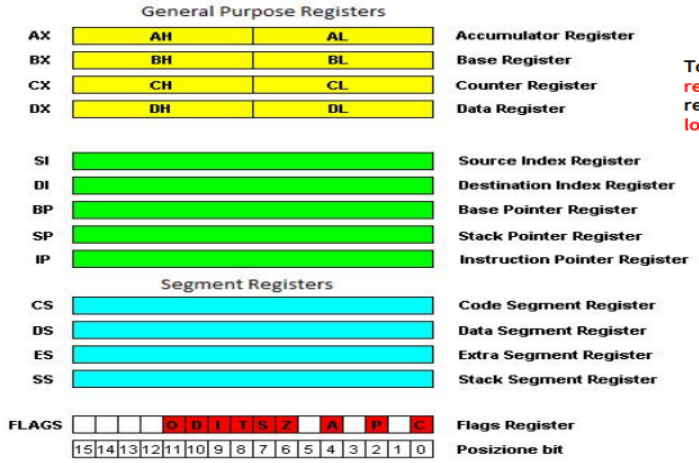
```
.MODEL SMALL
.STACK 100H
.DATA
    MSG1 DB 'ENTER 6 TEMPERATURE IN FARENHEIT (0-255): $'
    MSG2 DB 'TEMPERATURE SUMMATION IN CELSIUS: $'
    NUM1 DW ?
    NUM2 DW 0

.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



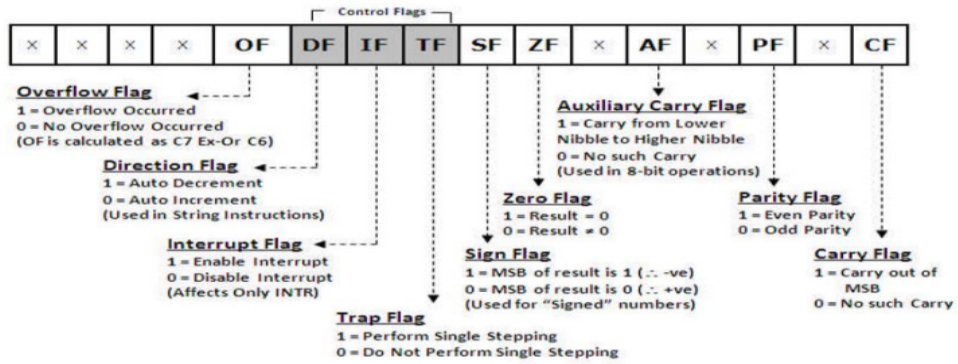
Total Fourteen registers: All of these registers are 16-bit long

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.

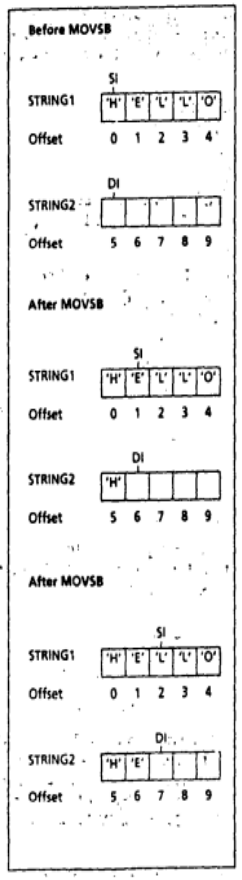
The tasks carried out by the string instructions can be performed by using the register indirect addressing mode we studied in Chapter 10; however, the string instructions have some built-in advantages. For example, they provide automatic updating of pointer registers and allow memory–memory operations.

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

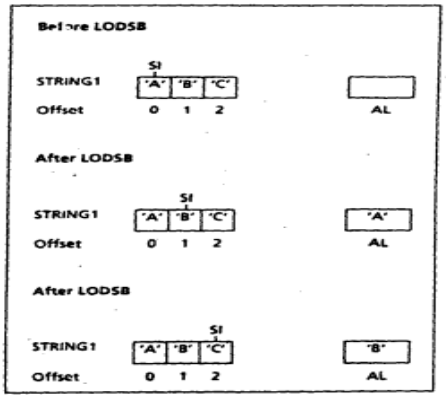
```
STRING1  DB     'ABC'
```

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.

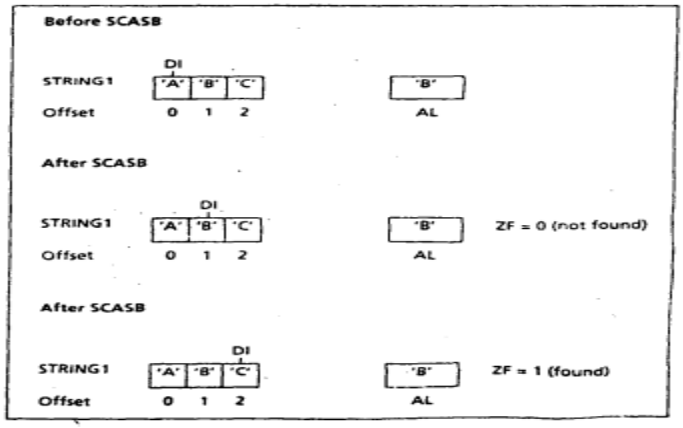
The word form is

```
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

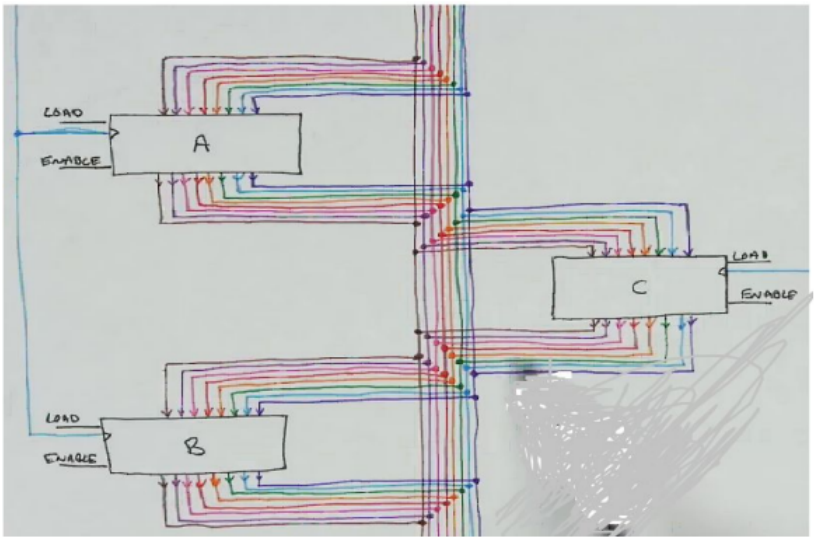
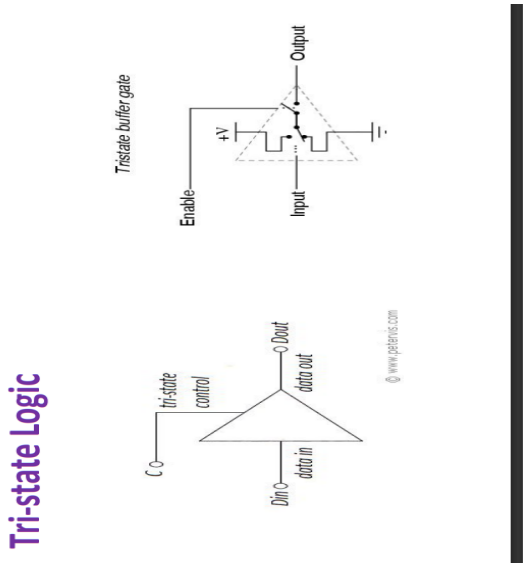
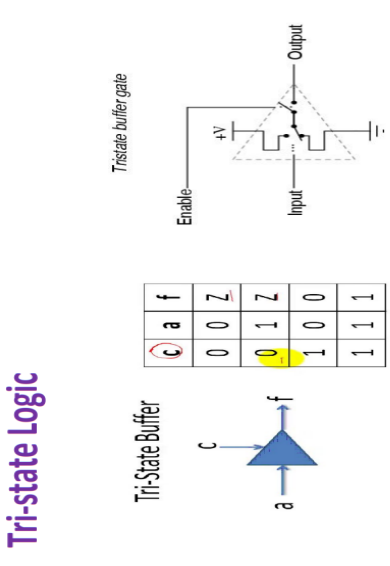


Figure: Typical Bus System, on which several devices are connected together.

•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.

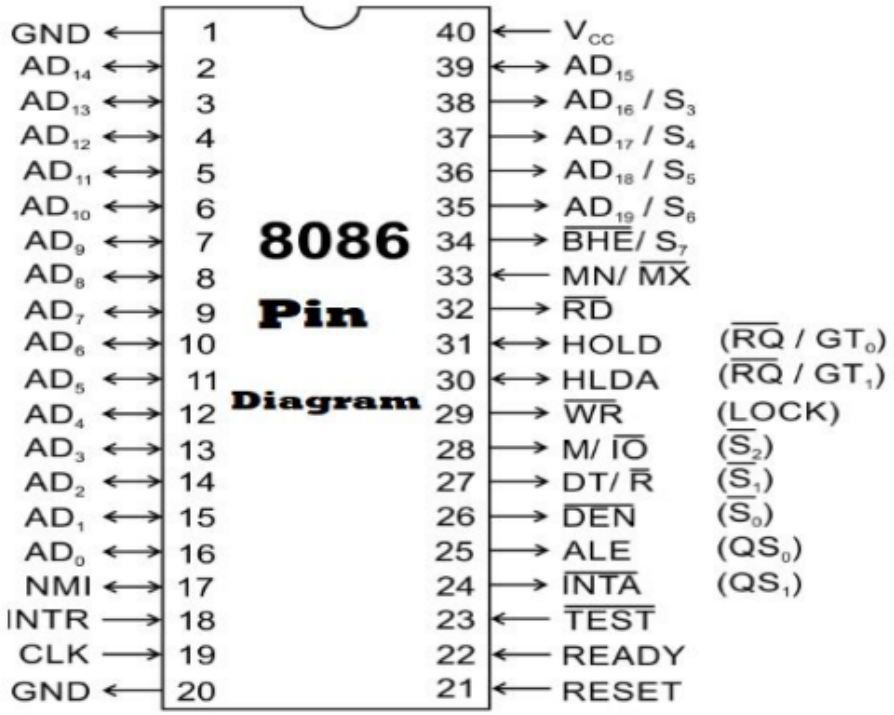
•All the devices will have tri-state drives at their output.
•Now when enabled, the outputs can either be '1' or '0' but when disabled they go into a third state •i.e. tri-state, where they can neither source or sink.
•Such outputs can be connected in parallel and they will not affect the existing condition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





8086 BUS TIMING DIAGRAM

8086 Pin Diagram

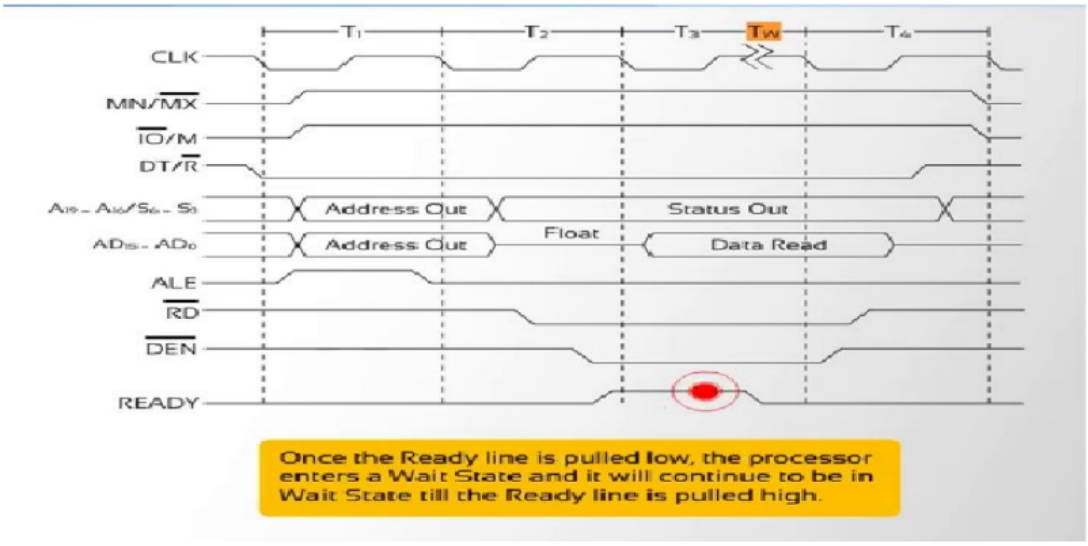


8086 Status Signals

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

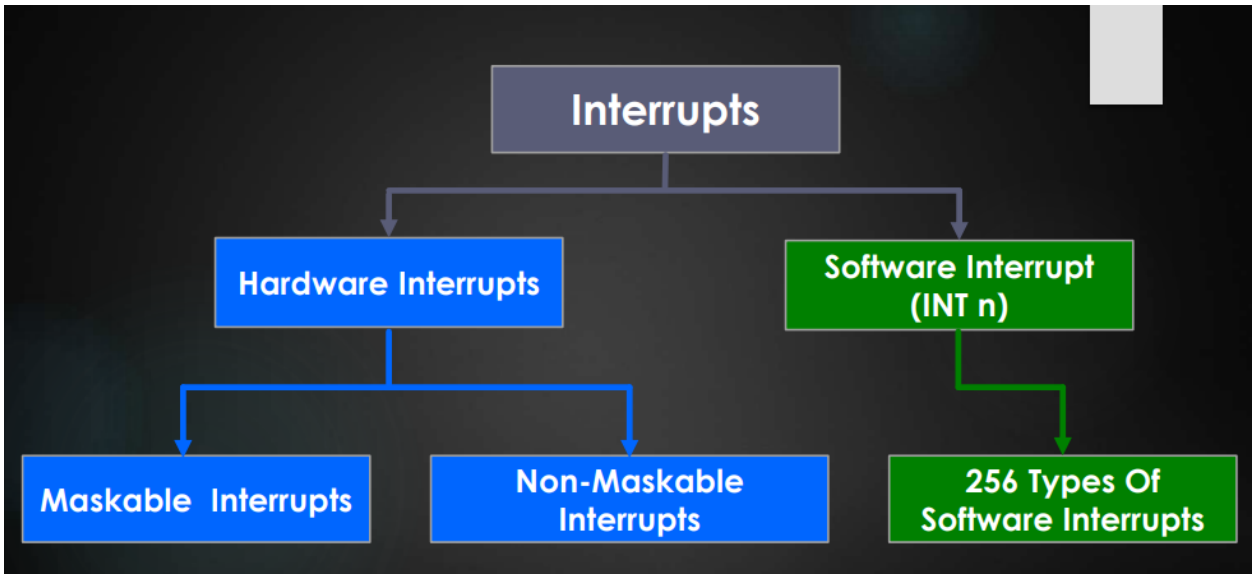
- Keep moving until interrupted by the sensor .
- Interrupt received then does pre-defined operation.
- After finishing the interrupt service return to normal operation i.e keep moving forward again

The processor can be interrupted in the following ways

i) by an external signal generated by a peripheral,

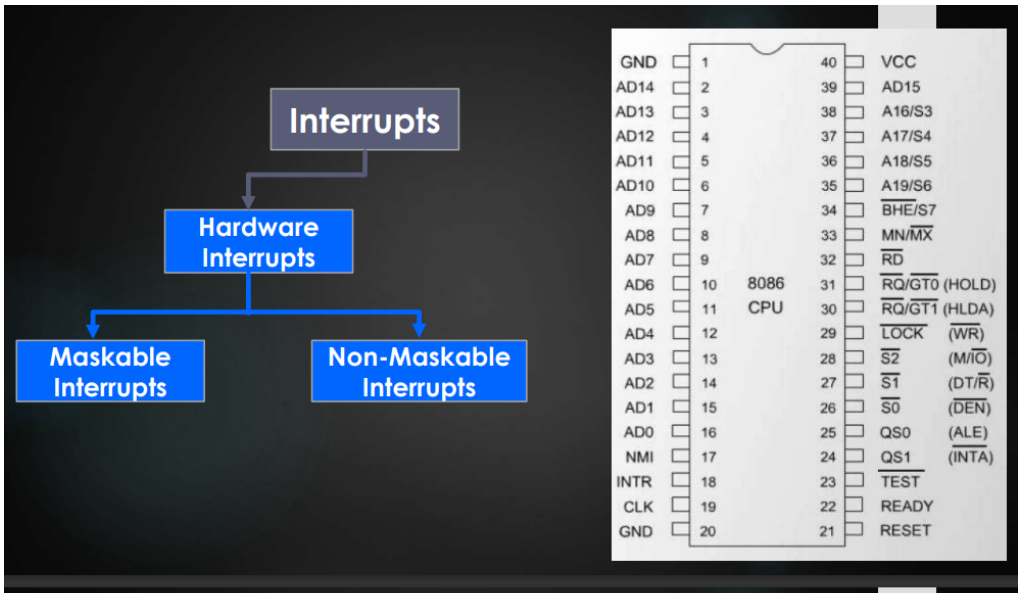
ii) by an internal signal generated by a special instruction in the program,

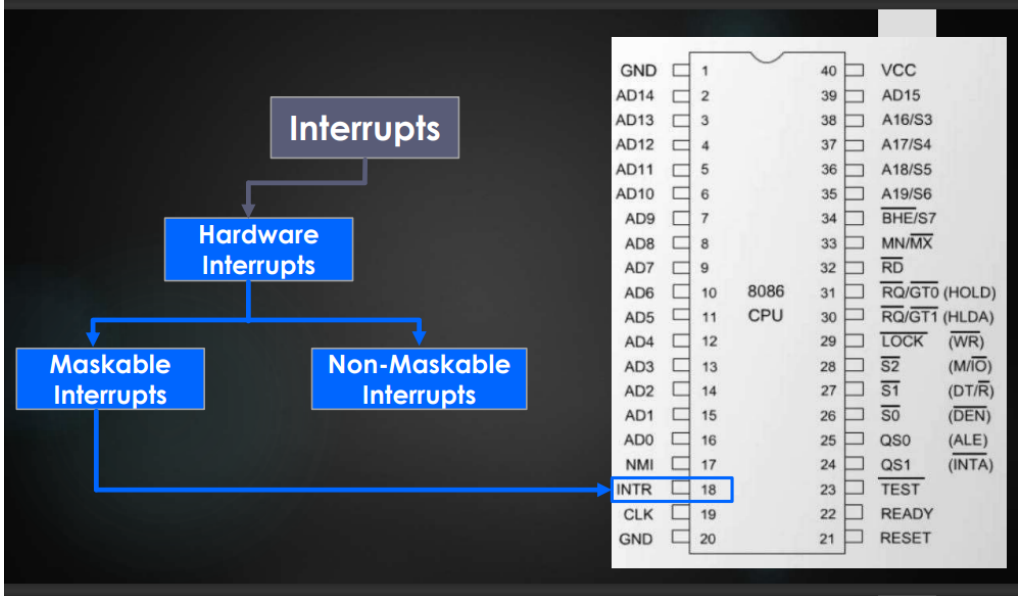
iii) by an internal signal generated due to an exceptional condition which occurs while executing an instruction.

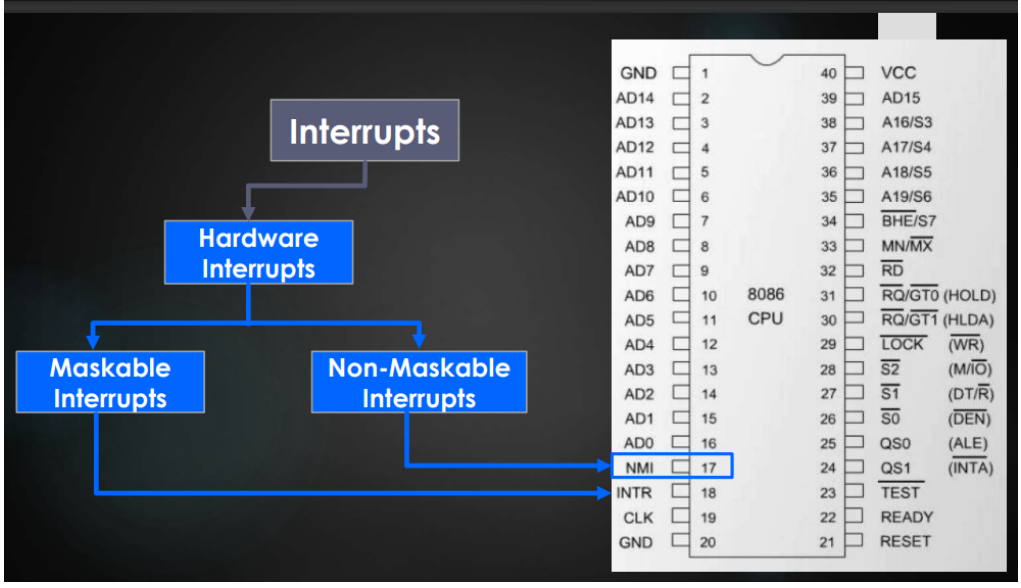


Hardware Interrupts

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

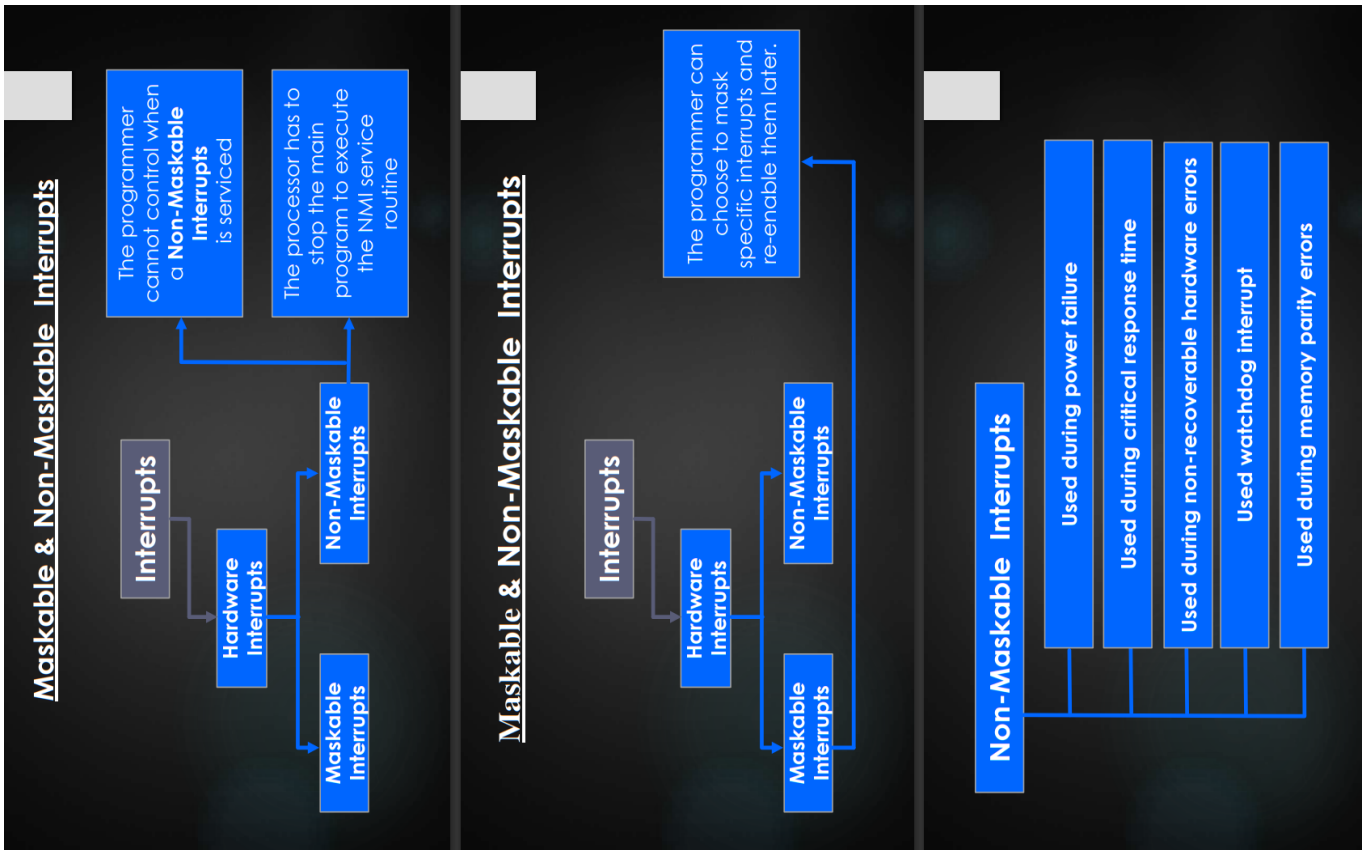

Interrupts
Hardware Interrupts
Maskable Interrupts
Non-Maskable Interrupts
GND 1
AD14 2
AD13 3
AD12 4
AD11 5
AD10 6
AD9 7
AD8 8
AD7 9
AD6 10
AD5 11
AD4 12
AD3 13
AD2 14
AD1 15
AD0 16
NMI 17
INTR 18
CLK 19
GND 20
8086 CPU
40 VCC
39 AD15
38 A16/S3
37 A17/S4
36 A18/S5
35 A19/S6
34 BHE/S7
33 MN/MX
32 RD
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

Attention reader! Don't stop learning now. Get hold of all the important CS Theory concepts for SDE interviews with the **CS Theory Course** at a student-friendly price and become industry ready.