UNIT-II
INSTRUCTION SET OF 8086

2.1 Explain Addressing modes of 8086?

Addressing mode: Addressing mode indicates a way of locating the data or operands. Addressing modes describe the types of operands and the way they are accessed by the microprocessor for executing an instruction. According to the flow of instruction execution, the instructions may be categorized as

- Sequential control flow instructions
- Control transfer instructions

Sequential control flow instructions: Sequential control flow instructions are the instructions which after execution, transfer control to the next instruction appearing immediately after it (in the sequence) in the program.

Ex: Arithmetic, logical, data transfer and processor control instructions.

Control transfer instructions: The control transfer instructions, transfer control to the some predefined address or the address somehow specified in the instruction, after their execution.

Ex: INT, CALL, RET (return), JMP (jump) instructions.

➢ **The addressing modes for sequential control flow instructions are:**

1. Immediate: In this type of addressing mode, immediate data is a part of the instruction.

Ex: MOV AX,0005H (0005H immediate data is copied to the AX register)
MOV BL,07H (07H immediate data is copied into BL register)

2. Direct: In direct addressing mode, a 16-bit offset address of memory or I/O is directly specified in the instruction.

Ex: MOV AX, [5000H] (The contents present in the 5000H location are copied into AX)
IN AL,80H (The i/p data is read from port address 80H and copied to AL)

3. Register: In this mode, the source operand and destination operand both are the registers. The data stored in a register and it is referred using the particular register. All registers, except IP , may be used in this mode.

Ex: MOV AX,BX (The BX register contents are copied to AX register)
ADD AX,BX (The BX register contents are added with AX register contents)

4. Register Indirect: Sometimes the address of the memory location which contains data or operand is determined in an indirect way, using the offset registers. This mode of addressing is known as register indirect addressing mode. The offset address of data is in either BX or SI or DI register. The default segment is either DS or ES.

Ex: MOV AX,[BX] (The offset address is indirectly specified by the register BX)

The physical address of the data is given as $(DS\times10_H)+[BX]$

5. Indexed: In this mode the offset address of the operand is stored in the one of the index registers. This mode is the special case of the register indirect addressing mode. The default segment is either DS or ES.

Ex: MOV AX,[SI] (the contents of the offset address specified by SI register are copied to AX)

The physical address of the data is given as $(DS\times10_H)+[SI]$

MOV CX,[DI] (the contents of the offset address specified by DI register are copied to CX)

The physical address of the data is given as $(DS\times10_H)+[SI]$

6. Register relative: In this mode, the offset address of the operand is calculated by adding the offset register contents to the 8-bit or 16-bit displacement with the content any one of the registers BX,BP,SI and DI. The default segment is either DS or ES.

Ex: MOV AX,[BX]50H
MOV 10H[SI],DX

The physical address is calculated as **$(DS\times10_H)+[BX]+50H$** and **$(DS\times10_H)+[SI]+10H$** respectively

7.Based indexed:

In this addressing mode, the physical address is calculated by using adding the contents of base register to the content of a base register(any one of BX or BP) to the contents of an index register (SI or DI). The default segment is either DS or ES.

Ex: MOV AX,[BX][SI]
MOV [BX][DI],AX

Here BX is the base register, SI is the index register. The physical address is calculated as

$(DS\times10_H)+[BX]+[SI]$

8.Relative based indexed:

In this addressing mode, the physical address is calculated by using adding the 16-bit or 8-bit displacement with the sum of contents any one of base register BX or BP to the any one of the index register (SI or DI).

Ex: MOV AX,[BX][SI]50H;

The physical address is calculated as **$(DS\times10_H)+[BX]+[SI]+50H$**

- **The addressing modes for control transfer instructions are:**

Intra segment mode : If the location to which the control is to be transferred lies within the segment other , the mode is called intra segment mode.

Inter segment mode: If the location to which the control is to be transferred lies in a different segment other than the current segment , the mode is called inter segment mode.

1. Intra segment Direct:

In this mode the address to which the control is to be transferred lies in the same segment, in which the control transfer instruction appears directly in the instruction as an immediate displacement value (address).

EX: JMP SHORT LABEL ; Label lies within -128 to +127 from the current IP content .
Thus SHORT LABEL is 8-bit signed displacement

2. Intra segment indirect:

In this mode the address to which the control is to be transferred lies in the same segment, in which the control transfer instruction appears indirectly in the instruction.

EX: JMP [BX] ; Jump to effective address stored in [BX].

3. Inter segment direct:

In this mode the address to which the control is to be transferred lies in the different segment, in which the control transfer instruction appears directly in the instruction. Here, the CS and IP of the destination address are specified directly in the instruction..

Ex: JMP 5000H:2000H ; Jump to effective address 2000H in segment 5000H

4. Inter segment Indirect:

In this mode the address to which the control is to be transferred lies in the different segment, in which the control transfer instruction appears indirectly in the instruction as an immediate displacement value (address).i.e. contents of a memory block containing four bytes, IP (LSB) , IP (MSB) ,CS (LSB) , CS (MSB) sequentially.

Ex: JMP [2000H] ; Jump to an address in the other segment specified at effective address [2000]

2.2 Classify the instruction set of 8086?

The 8086 instructions are categorized into following types

- **Data copy/transfer instructions:** These types of instructions are used to transfer data from source operand to destination operand. All MOV, LOAD, STORE, XCHG, IN and OUT instructions are belonging to this category.
- **Arithmetic and logical instructions:** All the instructions performing arithmetic, logical, increment, decrement and compare instructions are belonging to this category.
- **Branch Instructions:** These instructions transfer control of execution to the specified address. All the CALL, JMP, INT and RET instructions are belonging to this category.
- **Loop instructions:** These are used to implement unconditional and conditional loops. The LOOP, LOOPNZ, LOOPZ are come under this category.
- **Machine control instructions:** These instructions control the machine status. NOP, HLT, WAIT and LOCK instructions are belonging to this class.
- **Flag manipulation Instructions:** All the instructions which directly affect the flag register. Instructions like CLC, STC, CMC, STD, CLD belong to this category.
- **Shift and rotate instructions:** These instructions involve bitwise shifting or rotation in either direction with or without a count in CX.
- **String instructions:** These instructions involve various string manipulation operations like load , move, scan, compare, store, etc. These instructions are only to be operated upon strings.

2.3 Use Data Transfer instructions of 8086?

These types of instructions are used to transfer data from source operand to destination operand. Source and destination operand should be of same size, either both the operand size should be byte (or) word.

1. **MOV: Move**
2. **PUSH: Push to Stack**
3. **POP: Pop from Stack**
4. **XCHG: exchange**
5. **IN: Input from port**
6. **OUT: Output to port**
7. **XLAT: Translate**
8. **LEA: Load effective address**
9. **LDS/LES: Loads pointer to DS/ES**
10. **LAHF: Load AH from lower byte of flag**
11. **SAHF:Store AH to lower byte of flag reg**
12. **PUSHF: Push flag register on to the Stack**
13. **POPF: Pop Flags from stack**

MOV: Move

This instruction is used to transfer the data from source operand to destination operand. The source operand may be a register or a memory location, and the destination operand may be a register or a memory location.

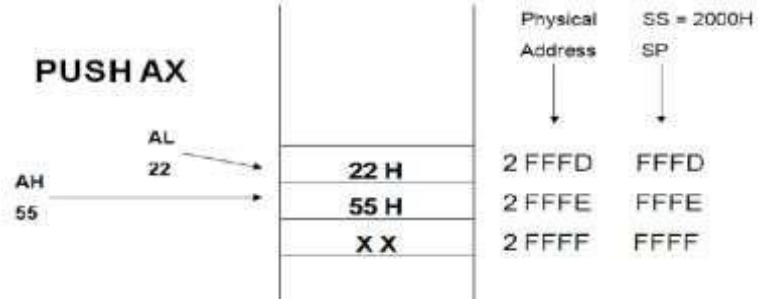
EX:- MOV AX, 5000H (the immediate data 5000h is copied to AX register)
MOV AX, BX (BX register contents are copied to AX)
MOV AX, [5001H] (The contents stored in 5001H memory location are copied into AX)
MOV AX, [SI]
MOV AX, 50H[BX]

PUSH: Push to Stack

This instruction pushes the contents of the specified register/memory location on to the stack. For every PUSH operation the stack pointer is **decremented** by 2. The actual current stack-top is always occupied by previously pushed data. Hence, the PUSH operation decrements SP by two and stores the two bytes of data onto the stack. Higher byte is pushed first on to the stack, then lower byte.

EX: PUSH [5000] ; (The contents of location 5000H and 5001H in DS are pushed onto the stack)

PUSH AX; AX 5522H ⟶

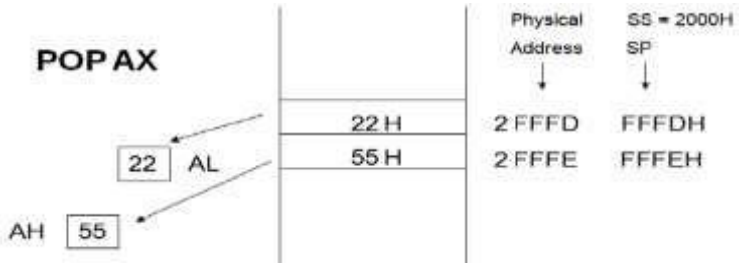


POP: Pop from Stack

This instruction pop's the data which is present in the Stack memory into the specified register or memory location. After the execution POP instruction, the stack pointer will be **incremented** by 2. The lower byte will be taken out first and later higher byte. The POP instruction serves exactly opposite to the PUSH instruction.

EX: POP AX; EX: POP [5001H]

NOTE: The stack pointer always holds the current top of the stack memory.



XCHG: exchange

This instruction exchanges the contents of the specified source to destination operands. The source operand may be a register or a memory location, and the destination operand may be a register or a memory location. But the exchange of contents of two memory locations is not permitted.

EX: - XCHG [5000H], AX (This instruction exchanges data between AX and a memory location [5000H] in the data segment)

XCHG BX, AX (This instruction exchanges data between AX and BX)

IN: Input from port

This instruction is used for reading an input port. The address of the input port maybe specified in the instruction directly (or) indirectly.

EX: - IN AL, 03H (the contents present in the port whose address is 03H is send to the AL)
IN AX, DX(the contents present in the port whose address is implicitly specified in DX is send to the AX)

OUT: Output to port

This instruction is used for writing to an output port. The address of output port may be specified in the instruction directly (or) indirectly.

EX: - OUT 03H, AL(the contents present in AL send to the port whose address is 03H)
OUT DX, AX (the contents present in AX send to the port whose address is implicitly specified in DX)

LEA: Load effective address

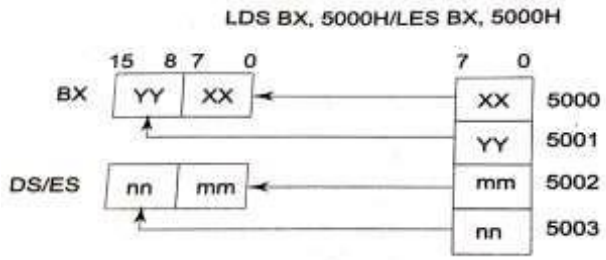
The load effective address instruction loads the effective address formed by destination operand into the specified source register.

EX: - LEA BX, ADR ; (effective address of label ADR is transferred to BX register.)
LEA SI , ADR [BX] ; (offset of label ADR will be added to content of BX to form effective address and it will be loaded in SI)

LDS/LES: Loads pointer to DS/ES

This instruction loads the DS(or) ES reg and the specified destination reg in the instruction within the content of memory location specified as source in the instruction.

EX: - LES BX, 5000H
LDS BX, 5000H



LAHF: Load AH from lower byte of flag

This instruction loads the AH register with the lower byte of the flag register.This command may be used to observe the status of all the condition code flags (except overflow) at a time.

Ex: LAHF

SAHF: Store AH to lower byte of flag reg

This instruction set(or) reset the condition code flags (except overflow) in the lower byte of the flag register depending upon the corresponding bit position in AH. If a bit in the AH is 1, the corresponding flag bit position is set, else it is reset.

Ex: SAHF

PUSHF: PUSH flag register on to the Stack

The push flag instruction pushes the flag register on to stack. First upper byte is pushed and then lower byte is pushes onto the stack. SP is decremented by 2.

EX: PUSHF

POPF:POP Flags from stack

The Pop flag instructions loads the flag register contents from the stack memory addressed by SP, and the SP incremented by 2.

EX: POPF

XLAT: Translate

This instruction is used for finding out the codes in case of code conversion problems.

EX: - XLAT

2.4 Use Arithmetic Instructions of 8086 micro processor?

These instructions perform the arithmetic operations, like addition, subtraction, multiplication and division along with the respective ASCII and decimal adjust instructions. The increment and decrement operations also belong to this type of instructions.

1. **ADD: Add**
2. **ADC: Add with carry**
3. **INC: Increment**
4. **DEC: Decrement:**
5. **SUB: Subtract**
6. **SBB: Subtract with Borrow**
7. **CMP: Compare**
8. **NEG: Negate**
9. **MUL: Unsigned multiplication Byte or Word**
10. **IMUL: Signed multiplication**
11. **CBW: Convert signed byte to word**
12. **CWD: Convert signed word to double word**
13. **DIV: Unsigned division**
14. **IDIV: Signed division**
15. **AAA : ASCII Adjust After Addition**
16. **AAS : ASCII Adjust AL after Subtraction**
17. **AAM : ASCII Adjust after Multiplication**
18. **AAD : ASCII Adjust before Division**
19. **DAA : Decimal Adjust Accumulator**
20. **DAS : Decimal Adjust after Subtraction**

ADD: Add

This instruction adds an immediate data or contents of a memory location specified in the instruction or a register (source) to the contents of another register(destination) or memory location. The result is in the destination operand. However, both the source and destination operands cannot be memory operands. That means memory to memory addition is not possible.

EXAMPLE

1. ADD AX, 0100H; IMMEDIATE
2. ADD AX, BX ; REGISTER
3. ADD AX, [SI] ; REGISTER INDIRECT
4. ADD AX, [5000H]; DIRECT
5. ADD [5000H], 0100H ; IMMEDIATE
6. ADD 0100H; DESTINATION AX (IMPLICIT)

ADC: **Add with carry**

This instruction perform the same operation as ADD instruction, but adds the carry flag bit (with may be set as a result of the previous calculations) to the result. All the condition code flags are affected by this instruction.

EX:

1. ADC AX, BX REGISTER
2. ADC AX, [SI] REGISTER INDIRECT
3. ADC AX, [5000H] DIRECT
4. ADC [5000H], 0100H IMMEDIATE

INC: Increment

This instruction increases the contents of the specified register or memory location by 1. All the condition code flags are affected the except the carry flag CF. This instruction adds 1 to the contents of the operand. Immediate data cannot be operand of this instruction.

EX:

1. INC AX REGISTER
2. INC [BX] REGISTER INDIRECT
3. INC [5000H] DIRECT

DEC: Decrement: The decrement instruction subtracts 1 from the contents of the specified register or memory location. All the condition code flags, except the carry flag, are affected depending upon the result immediate data cannot be operand of the instruction. The examples of this instruction are as follows:

Ex:

1. DEC AX REGISTER
2. DEC [5000H] DIRECT
3. DEC [BX] REGISTER INDIRECT

SUB: Subtract

The subtract instruction subtracts the source operand from the destination operand and the result is left in the destination operand. Source operand may be a register, memory location or immediate data and the destination operand may be a register or a memory location, but source and destination operands both must not be memory operands. Destination operand cannot be an immediate data. All the condition code flags are affected by this instruction.

Ex:

1. SUB AX, 0100H IMMEDIATE
2. SUB AX, BX REGISTER
3. SUB AX, [5000H] DIRECT
4. SUB [5000H],0100H IMMEDIATE

SBB: Subtract with Borrow

The subtract with borrow instruction means, subtracting 1 from the subtraction obtained by SUB, if carry (borrow) flag is set.

Ex:

1. SBB AX,0100H IMMEDIATE [DESTINATION AX]
2. SBB AX,BX REGISTER
3. SBB AX,[5000H] DIRECT
4. SBB [5000H],0100H IMMEDIATE

CMP: Compare

This instruction compares the source operand, which may be a register or an immediate data or a memory location, with a destination operand that may be a register or a memory location.

For comparison, it subtracts the source operand from the destination operand but does not store the result anywhere. The flags are affected depending upon the result of the subtraction. If both of the operands are equal, zero flag is set. If the source operand is greater than the destination operand, carry flag is set or else, carry flag is reset. The Example of this instruction is as follows.

Example

1. CMP BX, 0100H IMMEDIATE
2. CMP AX, 0100H IMMEDIATE
3. CMP [5000H], 0100H DIRECT
4. CMP BX, [SI] REGISTER INDRECT
5. CMP BX, CX REGISTER

NEG: Negate

The negate instruction performs 2's complement of the specified destination in the instruction. For obtaining 2's complement it performs the subtraction of the source operand from zero. And the result is stored in the destination operand.

EX: NEG AX AX ← 0 – AX

NEG AL

If AL = 0011 0101 = 35H (Replace number in AL with its 2's complement)

AL = 1100 1011 = CBH

MUL: Unsigned multiplication Byte or Word

This instruction multiplies an unsigned byte or word by the contents of AL/AX. The unsigned byte or word may be in any one of the general purpose registers or memory locations. The most significant word of the result is stored in DX, while the least significant word of the result is stored in AX.

Ex:

1. MUL BH ; (AX)←(AL)×(BH)
2. MUL CX; (DX) (AX)←(AX)×(CX)
3. MUL WORD PTR [SI] (DX)(AX) (AX) x ([SI])

IMUL: **Signed multiplication**

This instruction multiplies a signed byte in source operand by a signed byte in AL or a signed word in source operand by a signed word in AX. The source can be a general purpose register, memory operand, index register or base register, but it cannot be an immediate data. In case of 32-bit result, the higher order word (MSW) is stored in dx and the lower order word is stored in AX.

Ex:

1. IMUL BH
2. IMUL CX
3. IMUL [SI]

CBW: Convert signed byte to word

This instruction converts a signed byte to a signed word. In other words, it **copies the sign bit** of a byte to be converted to all the bits in the higher byte of the result word. The byte to be converted must be in AL. The result will be in AX. It does not affect any flag.

EX: CBW
AX= 0000 0000 1001 1000 Convert signed byte in AL signed word in AX.
Result in AX = 1111 1111 1001 1000

CWD: Convert signed word to double word

This instruction **copies the sign bit** of AX to all the bits of the DX register. This operation is to be done before signed division. It does not affect any flag.

EX: CWD
AX = 1111 0000 1100 0001 Convert signed word in AX to signed double word in DX : AX
DX= 1111 1111 1111 1111 Result in AX = 1111 0000 1100 0001

DIV: Unsigned division

This instruction performs unsigned division. It divides an unsigned word or double word by a 16-bit operand. The dividend must be in AX for 16-bit operation and divisor may be specified using any one of the addressing modes except immediate. The result will be in AL (quotient) while AH will contain the remainder.

EX: DIV CL ; Word in AX / byte in CL ; Quotient in AL, remainder in AH
DIV CX ; Double word in DX and AX / word ; in CX, and Quotient in AX, ; remainder in DX

IDIV: Signed division

This instruction performs the same operation as the DIV instruction, but with signed operands. The results are stored similarly as in case of DIV instruction in both cases of word and double word division.

EX: IDIV BH

AAA : ASCII Adjust After Addition

The AAA instruction is executed after an ADD instruction that adds two ASCII coded operand to give a byte of result in AL. The AAA instruction converts the resulting contents of Al to a unpacked decimal digits.

```
Eg.  ADD CL, DL ; [CL] = 32H = ASCII for 2
; [DL] = 35H = ASCII for 5
                ; Result [CL] = 67H
     MOV AL, CL ; Move ASCII result into AL since
                ; AAA adjust only [AL]
      AAA       ; [AL]=07, unpacked BCD for 7
```

AAS : ASCII Adjust AL after Subtraction

This instruction corrects the result in AL register after subtracting two unpacked ASCII operands. The result is in unpacked decimal format. The procedure is similar to AAA instruction except for the subtraction of 06 from AL.

AAM : ASCII Adjust after Multiplication

This instruction, after execution, converts the product available In AL into unpacked BCD format.

Eg.

```
MOV AL, 04 ; AL = 04
MOV BL ,09 ; BL = 09
MUL BL     ; AX = AL*BL ; AX=24H
AAM        ; AH = 03, AL=06
```

AAD : ASCII Adjust before Division

This instruction converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL. This adjustment must be made before dividing the two unpacked BCD digits in AX by an unpacked BCD byte. In the instruction sequence, this instruction appears Before DIV instruction.

Eg.
AX 05 08
AAD result in AL 00 3A ,since 58D = 3AH in AL
The result of AAD execution will give the hexadecimal number 3A in AL and 00 in AH. Where 3A is the hexadecimal Equivalent of 58 (decimal).

DAA : Decimal Adjust Accumulator

This instruction is used to convert the result of the addition of two packed BCD numbers to a valid BCD number. The result has to be only in AL.

Eg.

```
AL = 53 CL = 29
ADD AL, CL ; AL <– (AL) + (CL)
           ; AL 53 + 29
           ; AL 7C
DAA ; AL 7C + 06 (as C>9)
    ; AL 82
```

DAS : Decimal Adjust after Subtraction

This instruction converts the result of the subtraction of two packed BCD numbers to a valid BCD number. The subtraction has to be in AL only.

Eg.

```
AL = 75, BH = 46
SUB AL, BH ; AL   2F = (AL) – (BH)
 ; AF = 1
DAS ; AL 29 (as F>9, F – 6 = 9)
```

2.5 Use Logical Instrutions of 8086 microprocessor?

These types of instructions are used for performing the bit by bit shift, rotate, or basic logical operation. Basic logical operations available with 8086 instruction set are AND, OR, NOT, and XOR.

1. **AND: logical AND**
2. **OR: Logical OR**
3. **NOT: logical invert**
4. **XOR: (logical exclusive OR)**
5. **TEST**
6. **SHL/SAL: shift logical/Arithmetic Left**
7. **SHR: Shift Logical Right**
8. **SAR: Shift Arithmetic Right**
9. **ROR: Rotate Right without Carry**
10. **ROL: Rotate Left without Carry**
11. **RCR: Rotate Right through Carry**
12. **RCL: Rotate Left through Carry**

AND: logical AND

This instruction performs the bit by bit AND operation between the source operand(may be a reg or memory location) to the destination operand(may be a reg or memory location) contents. The result is stored in the destination operand. Both operations con't be memory locations and immediate data.

Example 2.37

```
1. AND AX, 0008H
2. AND AX, BX
3. AND AX, [5000H]
4. AND [5000H], DX
```

If the content of AX is 3F0FH, the first example instruction will carry out the operation as given below. The result 3F9FH will be stored in the AX register.

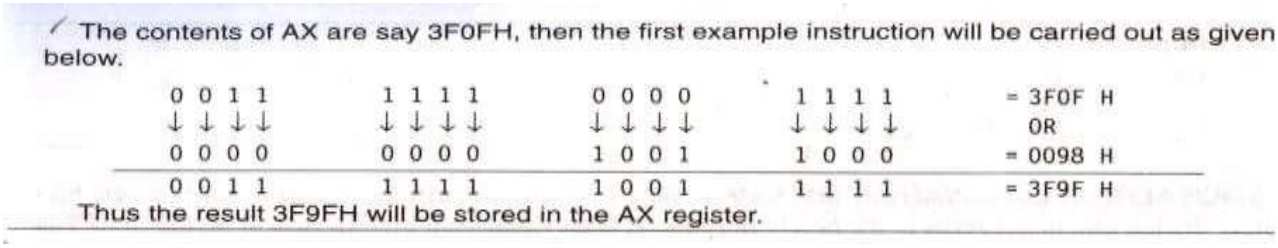
```
0 0 1 1     1 1 1 1     0 0 0 0     1 1 1 1     = 3F0F H [AX]
↓ ↓ ↓ ↓     ↓ ↓ ↓ ↓     ↓ ↓ ↓ ↓     ↓ ↓ ↓ ↓       AND
0 0 0 0     0 0 0 0     0 0 0 0     1 0 0 0     = 0008 H
-------------------------------------------------------
0 0 0 0     0 0 0 0     0 0 0 0     1 0 0 0     = 0008 H [AX]
```

The result 0008H will be in AX.

OR: Logical OR

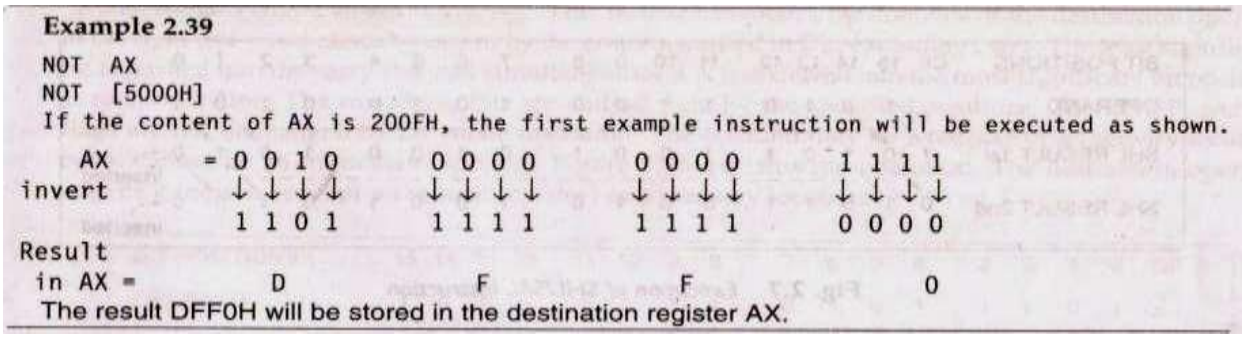
The OR instruction performs the OR operation between the source operand (may be a reg or memory location) to the destination operand(may be a reg or memory location) contents. The result is stored in the destination operand. Both operations con't be memory locations and immediate data.

EX : OR AX, 0098H



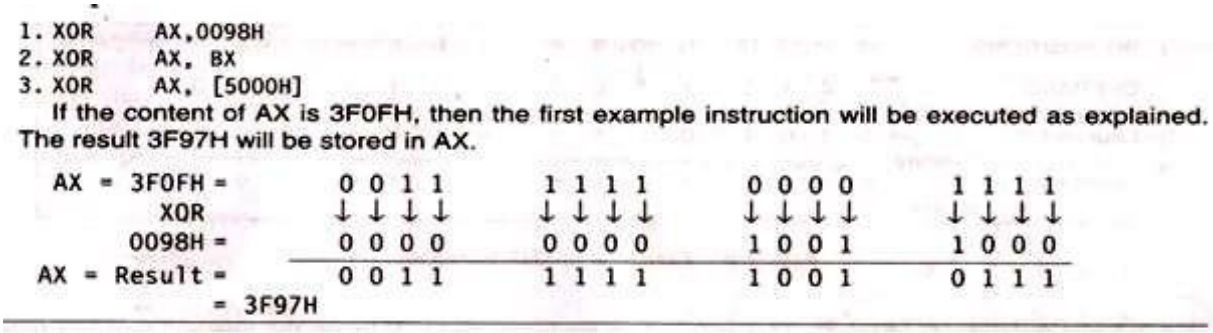
NOT: logical invert

The NOT instruction complements the contents of an operand register or a memory location, bit by bit. The examples are as follows:



XOR: (logical exclusive OR)

The XOR instruction performs XOR operation between the two operands. It gives a high output, when the 2 input bits are dissimilar otherwise, the output is zero. The example instructions are as follows:



TEST:

The TEST instruction performs a bit by bit logical AND operation on the operands. The result of this operation is not stored anywhere, but flags are affected. The affected flags are OF, CF, SF, ZF and PF. The operands may be register, memory or immediate data. The example of this instruction is as follows:

EX: TEST AX, BX
TEST [0500], 06H

SHL/SAL: shift logical/Arithmetic Left

These instructions shift the operand word or byte bit by bit to the left and insert zeros in the newly introduced least significant bits. In case of all the SHIFT and ROTATE instructions, the count is either 1

or specified by register CL. Figure explains the execution of this instruction. It is to be noted here that the shift operation is through carry flag.

Fig. 2.7 *Execution of SHL/SAL Instruction*

SHR: Shift Logical Right

This instruction perform bit-wise right shifts on the operand word or byte that may reside in a register or a memory location, by the specified count in the instruction and inserts zeros in the shifted positions. The result is stored in the destination operation. Figure explains execution of this instruction shifts the operand through the carry flag.

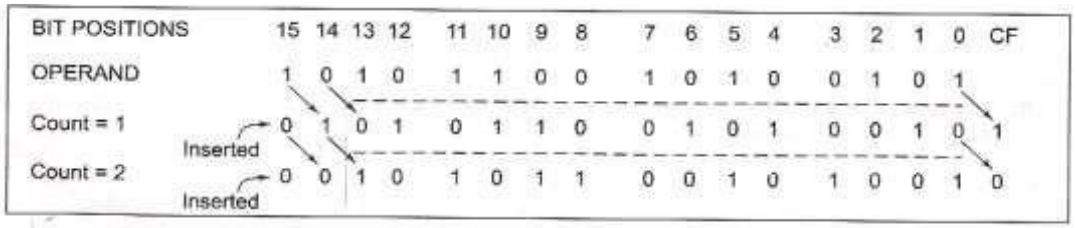


Fig. 2.8 *Execution of SHR Instruction*

SAR: Shift Arithmetic Right

This instruction performs right shifts on the operand word or byte that may be a register or memory location by the specified count in the instruction. In inserts the most significant bit of the operand in the newly inserted positions. The result s stored in the destination operand. Figure explains execution of the instruction. All the condition code flags are affected. This shift operation shifts the operand through the carry flag.

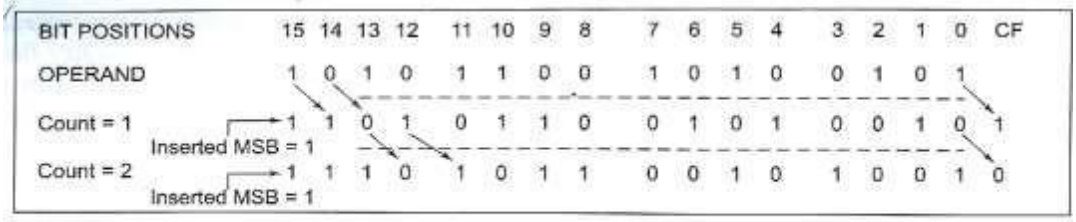


Fig. 2.9 *Execution of SAR Instruction*

ROR: Rotate Right without Carry

This instruction rotates the contents of destination operand to the right (bit-wise) either by the count specified in CL, excluding carry. The least significant bit is pushed into the carry flag and simultaneously it is transferred into the most significant bit position at each operation. The remaining bits are shifted right by the specified positions. The PF, SF, and ZF flags are left unchanged by the rotate operation

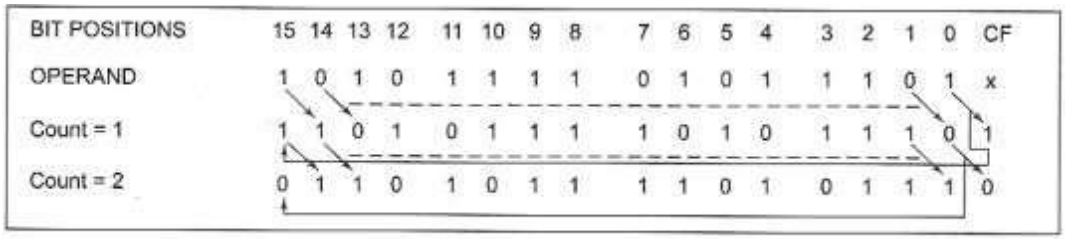


Fig. 2.10 *Execution of ROR Instruction*

ROL: Rotate Left without Carry

This instruction rotates the content of the destination operand to the left by the specified count (bit-wise) excluding carry. The most significant bit is pushed into the carry flag as well as the least significant bit position at each operation. The remaining bits are shifted left subsequently by the specified count positions.

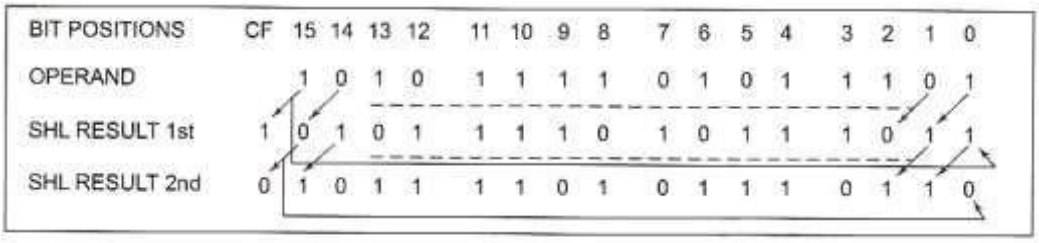


Fig. 2.11 *Execution of ROL Instruction*

RCL: Rotate Left through Carry

This instruction rotates (bit-wise) the contents of the destination operand left by the specified count through the carry flag (CF). For each operation, the carry flag is pushed into LSB, and the MSB of the operand is pushed into carry flag. The remaining bits are shifted left by the specified positions. The SF, PF, ZF are left unchanged. The operand may be a register or a memory location. Figure 2.13 explains the operation.

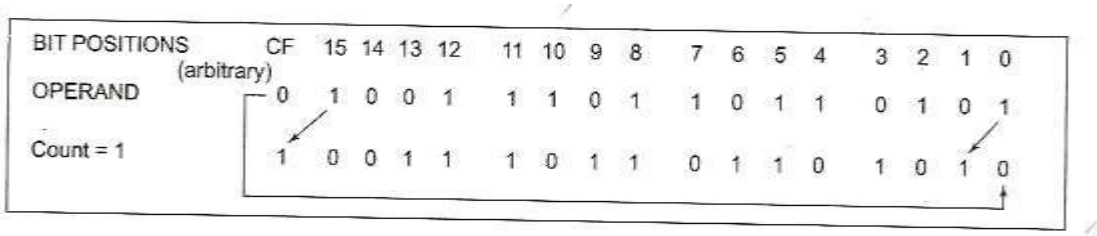


Fig. 2.13 *Execution of RCL Instruction*

RCR: Rotate Right through Carry

This instruction rotates (bit-wise) the contents of the destination operand right by the specified count through the carry flag (CF). For each operation, the carry flag is pushed into MSB, and the LSB of the operand is pushed into carry flag. The remaining bits are shifted left by the specified positions. The SF, PF, ZF are left unchanged. The operand may be a register or a memory location. Eg:

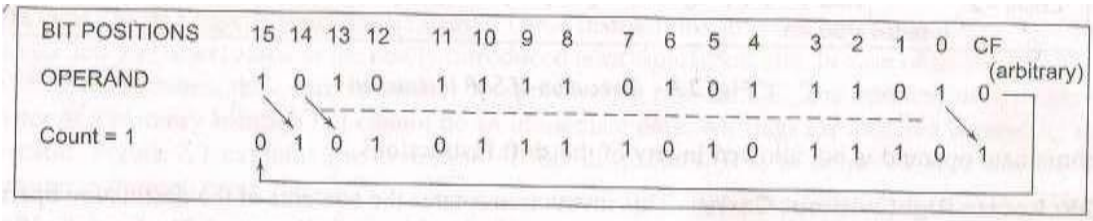


Fig. 2.12 *Execution of RCR Instruction*

2.6 Use Processor/ Machine Control Instructions of 8086 microprocessor?

These instructions control the functioning of the available hardware inside the processor chip. These are categorized into two types.

1. Processor/ machine controlled instructions.
2. Flag manipulated instructions.

Machine control instructions:

The machine control instructions supported by 8086 and 8088 are listed in table along with their functions .They do not require any operand. The Machine control instructions control the bus usage and execution.

WAIT	-	Wait for Test input pin to go low
HLT	-	Halt the processor
NOP	-	No operation
ESC	-	Escape to external device like NDP (numeric co-processor)
LOCK	-	Bus lock instruction prefix.

1. WAIT – Wait for Test input pin to go low.
This instruction causes the processor to enter into an idle state or a wait state and continue to remain in that state until a signal is asserted on TST input pin.
2. HLT – Halt the process.
This instruction is used to terminate the program. On execution of this instruction the processor enters into an idle state.
3. NOP – No operation.
No operation is performed on execution of this instruction.
4. ESC – Escape to external device
This instruction is used to pass instructions to a coprocessor which shares the address and data bus with the 8086
5. LOCK prefix – Bus lock instruction
This instruction prevents the other processors from accessing the system bus.

2.7 Instructions affecting flags of 8086 microprocessor?(or) Flag manipulated instructions?

All the instructions which directly affect the flag register come under this group of instructions. Instructions like CLD, STD, CLI, STI, etc. belong to this category of instructions. 8086 microprocessor has 9 flags. Six flags are altered by arithmetic and logical instructions and three flags are used to control the processor operation.

Flag manipulation instructions:

The flag manipulation instructions directly modify some of the flags of 8086. These instructions modify the carry (CF), Direction (DF) and Interrupt (IF) flags directly.
The Flag manipulation instructions directly modify some of the Flags of 8086.

i. **CLC** – Clear Carry Flag: The carry flag is reset to zero CF=0
ii. **CMC** – Complement Carry Flag: The carry flag is complemented CF= $\overline{CF}$
iii. **STC** – Set Carry Flag: The carry flag is set to one CF=1
iv. **CLD** – Clear Direction Flag: The direction flag is reset to zero DF=0
v. **STD** – Set Direction Flag: The direction flag is set to one DF=1
vi. **CLI** – Clear Interrupt Flag: The interrupt flag is reset to zero IF=0
vii. **STI** – Set Interrupt Flag: The interrupt flag is set to one IF=1

CLC	-	Clear carry flag
CMC	-	Complement carry flag
STC	-	Set carry flag
CLD	-	Clear direction flag
STD	-	Set direction flag
CLI	-	Clear interrupt flag
STI	-	Set interrupt flag

→ CF,AF,PF,ZF,SF,OF are altered by arithmetic logic instructions and TF,IF,DF are altered by process CTRL operations

→ CMP,CMPS,DEC,ROL,ROR, INC,MUL instructions effects the overflow flag

→ STD instruction sets the direction flag, CLD clears the direction flag.

→ STI instruction sets the interrupt flag, CLI clear the interrupt flag.

→ AAD, AAM, ADC, ADD, CMP, CMPS, DAA, DAS instructions alter and define the sign flag.

→ AAD,AAM,ADC,ADD,CMP,CMPS,DAA,DAS,DEC,SBB,SCAS instructions are alter and define the AF,ZF,PF,CF flags.

2.8 Use Control Transfer or Branching Instructions of 8086?

The control transfer instructions transfer the flow of execution of the program to a new address is specified in instruction directly or indirectly. When this type of instruction is executed, the CS and IP registers get loaded with new values of CS and IP corresponding to the location where the flow of execution is going to be transferred. This type of instructions is classified in two types:

1. Unconditional Control Transfer (Branch) Instruction:-

In case of unconditional control transfer instructions, the execution control is transferred to the specified location independent of any status or condition. The CS and IP are unconditionally modified to the new CS and IP.

1. **CALL: Unconditional Call**
2. **RET: Return from the procedure**
3. **INT N: Interrupt Type N**
4. **INT O: Interrupt on Overflow**
5. **JMP: Unconditional Jump**
6. **IRET: Return from ISR**
7. **LOOP: Loop Unconditionally**

CALL: Unconditional Call

This instruction is used to call a **subroutine** from a **main program**. In assembly language programming the term **procedure** is used in the place of subroutine. There are again two types of procedures depending upon whether it is available in the same segment

1. NEAR CALL (with in the segment)
2. FAR CALL (anywhere outside the segment)

On execution, this instruction stores the incremented IP (i.e. address of the next instruction) and CS onto the stack and loads the CS and IP register, respectively, with the segment and offset address of the procedure to be called. In case of **NEAR CALL** it pushes only IP register contents onto the stack and in case of **FAR CALL** it pushes IP and CS both register contents onto the stack. The NEAR and FAR CALLS are discriminated using opcode.

RET: Return from the procedure

At each CALL instruction, the IP and CS of the next instruction are pushed onto stack, before the control is transferred to the procedure. At the end of the procedure, the RET instruction must be executed. When it is executed, the previously stored content of IP and CS along with flag is retrieved into the CS, IP and flag registers from the stack and the execution of the main program continues further.

- In case of a **FAR procedure**, the current contents of SP points to IP and CS at the time of return. While in case of a **NEAR procedure**, it points to only IP, depending upon the type of procedure and the SP contents.
  - Return within segment
  - Return within segment adding 16-bit immediate displacement to the SP contents.
  - Return intersegment.
  - Return intersegment adding 16-bit immediate displacement to the SP contents.

INT N: Interrupt Type N

In the interrupt structure of 8086,256 interrupts are defined corresponding to the types from 00H to FFH. When an INT N instruction is executed, the TYPE byte N is multiplied by 4 and the contents of IP and CS of the interrupt service routine will be taken from the hexadecimal multiplication (N'4) as offset address and 0000 as segment address. For the execution of this instruction, the IF must be enabled.

INT O: **Interrupt on Overflow**

This instruction is executed, when the overflow flag OF is set. The new contents of IP and CS are taken from the address 0000:0010 as explained in INT type instruction. This is equivalent to Type 4 interrupt instruction.

JMP: Unconditional Jump

This instruction unconditionally transfers the control of execution to the specified address using an 8-bit or 16-bit displacement (intra segment relative, short or long) or CS: IP (intra segment direct far). No flags are affected by this instruction. Corresponding to the methods of specifying jump address, the JUMP instruction may have the following three formats.

```
Ex:     JMP [2000]
        JMP CS:IP
```

IRET: Return from ISR

This instruction is used to transfer the program execution control from the **ISR** to the **mainprogram.** At the end of each ISR, when IRET is executed, the values of IP, CS and flags are retrieved from the stack to continue the execution of the main program.

LOOP: **Loop Unconditionally**

This instruction executes the part of the program from the label or address specified in the instruction up to the loop instruction, CX number of times. After each iteration, CX is decremented automatically. In other words, this instruction implements DECREMENT COUNTER and JUMP IF NOT ZERO structure.MOV

```
Ex:     MOV CX , 0005           ; Number of times in CX
        MOV BX , 0FF7H          ; Data to BX
LABEL: MOV AX , CODE1
        OR BX , AX
        AND DX , AX
        LOOP LABEL
```

2.Conditional Control Transfer (Branch) Instruction:-

In the conditional control transfer instructions, the control is transferred to the specified location provided the result of the previous operation satisfies a particular condition, otherwise, the execution continues is normal flow sequence.

(or)

In other words, using this type of instruction the control will be transferred to a particular specified location, if a particular flag satisfies the condition.

In conditional branch instructions, only short jump is possible. That means , the control is transferred with in the displacement of -128 to +127.

The **Conditional JUMP instructions** are used to transfer or jump the execution flow control to a specified address when a particular flag satisfies the condition.

Some of the examples for conditional jump instructions are given below:

	Mnemonic	Displacement	Operation
1.	JZ/JE	Label	Transfer execution control to address 'Label', if ZF=1.
2.	JNZ/JNE	Label	Transfer execution control to address 'Label', if ZF=0.
3.	JS	Label	Transfer execution control to address 'Label' , if SF=1.
4.	JNS	Label	Transfer execution control to address 'Label', if SF=0.
5.	JO	Label	Transfer execution control to address 'Label', if OF=1.
6.	JNO	Label	Transfer execution control to address 'Label', if OF=0.
7.	JP/JPE	Label	Transfer execution control to address 'Label', if PF=1.
8.	JNP	Label	Transfer execution control to address 'Label', if PF=0.
9.	JB/JNAE/JC	Label	Transfer execution control to address 'Label', if CF=1.
10.	JNB/JAE/JNC	Label	Transfer execution control to address 'Label', if CF=0.
11.	JBE/JNA	Label	Transfer execution control to address 'Label', if CF=1 or ZF=1.
12.	JNBE/JA	Label	Transfer execution control to address 'Label', if CF=0 or ZF=0.
13.	JL/JNGE	Label	Transfer execution control to address 'Label', if neither SF=1 nor OF=1.
14.	JNL/JGE	Label	Transfer execution control to address 'Label', if neither SF=0 nor OF=0.
15.	JLE/JNC	Label	Transfer execution control to address 'Label', if ZF=1 or neither SF nor OF is 1.
16.	JNLE/JE	Label	Transfer execution control to address 'Label', if ZF=0 or at least any one of SF and OF is 1(Both SF and OF are not 0).

Conditional loop instructions:

The **Conditional LOOP instructions** are given below: with their meanings. These instructions may be used for implementing structures like DO_WHILE, REPEAT_UNTIL

Mnemonic	Displacement	Operation
LOOPZ/LOOPE(Loop while ZF=1;equal)	Label	Loop through a sequence of instructions from 'label' while ZF=1 and CX≠0
LOOPNZ/LOOPNE(Loop while ZF=0; not equal)	Label	Loop through a sequence of instructions from 'label' while ZF=0 and CX≠0

2.9 Use String Manipulation Instructions of 8086 microprocessor?

String:

A series of data bytes or words available in memory at consecutive locations, to be referred to collectively or individually, are called as byte strings or word strings.

For example, a string of characters may be located in consecutive memory locations, where each character may be represented by its ASCII equivalent.

For referring to a string, two parameters are required,

(a) Starting or end address of the string and

(b) Length of the string.

The length of a string is usually stored as count in the CX register. The incrementing or decrementing of the pointer, in case of 8086 string instructions, depends upon the Direction Flag (DF) status. If it is a byte string operation, the index registers are updated by one. On the other hand, if it is a word string operation, the index registers are updated by two. The counter in both the cases is decremented by one.

1. **REP: Repeat instruction prefix**
2. **MOVSB/MOVSW: Move string byte or string word**
3. **CMPS: Compare string byte or string word**
4. **SCAS: Scan String Byte or String Word**
5. **LODS: Load String Byte or String Word**
6. **STOS: Store String Byte or String Word**

REP: Repeat instruction prefix

This instruction is used as a prefix to other instructions. The instruction to which the REP prefix is provided is executed repeatedly until the CX register becomes zero. When CX becomes zero, the execution proceeds to the next instructions in sequence. There are two more options of the REP instruction.

- *REPE/REPZ* i.e. Repeat operation while equal/zero. (used as a prefix for CMPS)
- *REPNE/REPNZ* allows for repeating the operation while not equal/not zero.(used as a prefix for SCAS)

These options are used for CMPS, SCAS instructions only, as instruction prefixes.

MOVSB/MOVSW: Move string byte or string word

A string of bytes stored in a set of consecutive memory locations is to be moved to another set of destination locations. The MOVSB/MOVSW instruction thus, moves a string of bytes/words pointed to by DS: SI pair to the memory location pointed to by ES: DI pair.

The REP instruction prefix is used with MOVS instruction to repeat it by a value given in the counter. The length of the byte string or word string must be stored in CX register; no flags are affected by this instruction.

Example 2.42

```
MOV AX,5000H        ; Source segment address is 5000h
MOV DS,AX           ; Load it to DS
MOV AX,6000H        ; Destination segment address is 6000h
MOV ES,AX           ; Load it to ES
MOV CX,0FFH         ; Move length of the string to counter register CX
MOV SI,1000H        ; Source index address 1000H is moved to SI
MOV DI,2000H        ; Destination index address 2000H is moved to DI
CLD                 ; Clear DF, i.e. set autoincrement mode
REP MOVSB           ; Move 0FFH string bytes from source address to destination
```

CMPS: Compare string byte or string word

The CMPS instruction can be used to compare two strings of bytes or words, the length of the string must be stored in the register CX. If both the byte or word strings are equal, zero flag is set. The flags are affected in the same way as CMP instruction. The DS: SI and ES: DI point to the two strings, the REP instruction prefix is used to repeat the operation till CX becomes zero or the condition specified by the REP prefix is false.

Example 2.43

```
MOV AX,SEG1              ; Segment address of STRING1, i.e. SEG1 is moved to AX
MOV DS,AX                ; Load it to DS
MOV AX,SEG2              ; Segment address of STRING2, i.e. SEG2 is moved to AX
MOV ES,AX                ; Load it to ES
MOV SI,OFFSET STRING1    ; Offset of STRING1 is moved to SI
MOV DI,OFFSET STRING2    ; Offset of STRING2 is moved to DI
MOV CX,010H              ; Length of the string is moved to CX
CLD                      ; Clear DF, i.e. set autoincrement mode
REPE CMPSW               ; Compare 010H words of STRING1 and
                         ; STRING2, while they are equal, If a mismatch is found,
                         ; modify the flags and proceed with further execution
```

If both strings are completely equal, i.e. CX becomes zero, the ZF is set, otherwise, ZF is reset.

SCAS: Scan String Byte or String Word

This instruction scans a string of bytes or words for an operand byte or word specified in the register AL or AX. The string is pointed to by ES:DI register pair.

The length of the string is stored in CX .The DF controls the mode for scanning of the string. Whenever a match to the operand is found in the string, execution stops and zero flag is set. If no match is found, the zero flag is reset.

```
MOV   AX, SEG;        segment address of the string, i.e., SEG is moved to AX.
MOV   ES, AX;         Load it to ES
MOV   DI, OFFSET;     string offset, i.e., OFFSET is moved to DI.
MOV   CX, 010H;       Length of the string is moved to CX.
MOV   AX, WORD;       The word to be scanned for i.e., WORD is in AL.
CLD;                  clear DF
 REPNE    SCASW;     scan the 010h bytes of the string, till a match to WORD is found
```

LODS: Load String Byte or String Word

The LODS instruction loads the AL/AX register by the content of a string pointed to by DS: SI register pair. The SI is modified automatically depending upon DF. The DF plays exactly the same role as in case of MOVSB/MOVSW instruction. If it is a byte transfer (LODSB), the SI is modified by one and if it is a word transfer (LODSW), the SI is modified by two. No other flags are affected by this instruction.

STOS: Store String Byte or String Word

The STOS instruction stores the AL/AX register contents to a location in the string pointed by **ES: DI** register pair. The DI is modified accordingly. No flags are affected by this instruction.

The direction flag controls the string instruction execution. The source index SI and destination index DI are modified after each instruction automatically. If DF=1, then the execution follows auto decrement mode. In this mode SI and DI are decremented automatically after each iteration (by 1 or 2 depending upon byte or word operations). Hence, in auto decrement mode, the string is referred to by their ending address.

If DF=0, then the execution follows auto increment mode. In this mode SI and CI are incremented automatically (by 1 or 2 depending upon byte or word operation) after each iteration, hence the strings, in the strings, in this case, are referred to by their starting address. Chapter 3 on assembly language programming explains the use of sum of this instruction is assembly language programs.

2.10 Describe Assembler Directives of 8086 microprocessor?

The assembler directives are the instructions to the assembler regarding the program being assembled. The assembler directives specify.

- Start and End of the program
- Attach value to the variable
- Allocate storage location to input/output data
- To define start and End of Segment, Procedures
- The assembler may find out syntax errors. The logical errors or other programming errors are not found out by the assembler.

For completing all tasks, an assembler needs some hints from the programmer i.e.

1. The required storage for a particular constant or a variable
2. Logical names of the segments.

These types of hints are given to assembler using some predefined strings called **assembler directives** which help the assembler to correctly understand the assembly language programs.

The following directives are commonly used in the assembly language programming practice using Microsoft Macro Assembler or Turbo Assembler.

1. DB: Define Byte

This is used to reserve byte or bytes of memory locations in the available memory. This directive directs the assembler to allocate the specified number of memory bytes to the said data type.

EXAMPLE:

```
RANKS DB 01H, 02H, 03H, 04H
```

This statement directs the assembler to reserve 4 memory locations for a list named RANKS and initialize them with the above specified 4 values.

2. DW: Define Word

This directive serves the same purposes as the DB directive, but it now makes the assembler reserve the number of memory words (16-bit) instead of bytes.

EXAMPLE: WORDS DW 1234H, 4567H, 78ABH, 045CH

This makes the assembler reserve 4 words in memory, and initialize the words with the specified values in the statements.

3. DQ: Define Quad word

This directive is used to direct the assembler to reserve 4 words (8 bytes) of memory for the specified variable and may initialize it with the specified values.

4. DT: Define Ten Bytes

The DT directive directs the assembler to define the specified variable requiring 10-bytes for its storage and initialize the 10-bytes with the specified values.

5. ASSUME: Assume Logical Segment Name

The ASSUME directive is used to give the names for logical segments to be assumed for different segments used in the program.

```
ASSUME CS: CODE
ASSUME DS: DATA
```

6. END: END of Program

The END directive indicates the end of an assembly language program. The END statement should be the last statement in the file and should not appear in between.

7. ENDP: END of procedure

In assembly language programming the subroutines are called procedures. The ENDP directive is used to indicate the end of procedure .A procedure is usually assigned a name, i.e. label. To mark the end of particular procedure, the name of the procedure, i.e. label may appear as a prefix with the directive ENDP.

```
PROCEDURE STAR (Procedure name)
        :
        :
STAR ENDP //indicates the end of procedure STAR
```

8. ENDS: END of Segment

This directive marks the end of a logical segment. The logical segments are assigned with the names using the ASSUME directive. The names appear with the ENDS directive as prefixes to mark the end of those particular segments. Whatever are the contents of the segments, they should appear in the program before ENDS. Any statement appearing after ENDS will be neglected from the segment. The structure shown below explains the fact more.

```
DATA    SEGMENT
  :       :
  :       :
DATA    ENDS    //indicates the end of segment DATA

CODE    SEGMENT
  :       :
CODE    ENDS    //indicates the end of segment CODE
```

9. EVEN: Align On Even Memory Address:

The EVEN directive updates the location counter to the next even address, if the current location counter contents are not even, and assigns the following routine or variable or constant to that address.

```
        EVEN
PROCEDURE          ROOT
    :                :
  ROOT             ENDP
```

The above structure shows a procedure ROOT that is to be align at an even address The assembler will start assembling the main program calling ROOT. When the assembler comes across the directive even, It checks the contents of the location counter. If it is odd, It is updated to the next even value and then the ROOT procedure is assigned to that address, i.e the updated contents of the location counter. If the content of the location counter is already even, then the procedure will be assigned with the same address

10. EQU: Equate

The directive EQU is used to assign a label with a value or a symbol. The use of this directive is just to reduce the recurrence of the numerical values or constants in a program code. The recurring value is assigned with a label, and that label is used in the place of that numerical value, throughout the program.

EXAMPLE:

```
LABEL      EQU   0500H
ADDITION   EQU   ADD
```

This first statement assigns the constant 0500h with the label 'LABEL', while the second statement assigns another label ADDITION with mnemonic ADD.

11. OFFSET: **Offset of a Label**: When the assembler comes across the OFFSET operator along with a label, it first computes the 16-bit displacement of the particular label, and replaces the string 'OFFSET LABEL' by the computed displacement.

```
CODE      SEGMENT
MOV SI,   OFFSET LIST      // LIST is a label with displacement 10H
CODE      ENDS
DATA      SEGMENT
LIST DB   10H
DATA      ENDS
```

12. ORG: Origin

The ORG directive directs the assembler to start the memory allotment for a particular segment, block or code from the declared address in the ORG statement.

Ex: ORG 1100H

13. PROC: Procedure

The PROC directive marks the start of a named procedure in the statement. Also thee types NEAR or FAR specify the type of the procedure, i.e. whether the procedure is within the same segment (NEAR) or different segment (FAR).

```
EX: RESULT    PROC    NEAR
    ROUTINE   PROC    FAR
```

14. SEGMENT: Logical segment

The SEGMENT directive marks the starting of a logical segment. The started segment is also assigned a name i.e., label , by this statement. The SEGMENT and ENDS directive must specifies each logical segment of a

program. In some cases, The segment may be assign a type like PUBLIC (can be used by other modules of the program while linking) or GLOBAL (can be accessed by any other modules).

```
Ex:  EXE.CODE  SEGMENT GLOBAL     ( Start of segment named EXE.CODE, that can be accessed by
          :          :                any other module)
          :          :
       EXE.CODE    ENDS
```

15. SHORT:

The SHORT operator indicates to the assembler that only one byte is required to code the displacement for a jump (i.e displacement is within -128 to +127).

```
Ex: JMP SHORT LABEL
```

16. EXTERN: External and PUBLIC: Public: The directive EXTERN informs the assembler that the names, procedures and labels declared after this directive have already been defined in some other assembly language module1, If one wants to call a procedure FACTORIAL appearing in MODULE 1 from MODULE 2,in MODULE 1 it must be declared PUBLIC using the statement PUBLIC FACTORIAL and In MODULE 2,It must be declared as EXTRN FACTORIAL.

```
MODULE 1   SEGMENT
PUBLIC     FACTORIAL FAR
MODULE 1   ENDS
MODULE 2   SEGMENT
EXTRN      FACTORIAL FAR
MODULE 2   ENDS
```

17. GROUP: Group the Related Segments: This directive is used to form logical groups of segments with similar purpose or type.

EX: PROGRAM GROUP CODE, DATA, STACK //this statement directs the loader/linker to prepare an EXE file such that CODE, DATA, STACK segment must lie within a 64byte memory segment that is named as PROGRAM

```
ASSUME CS: PROGRAM , DS: PROGRAM , SS: PROGRAM
```

18. LABEL: The LABEL directive is used to assign a name to the current content of the location counter. A **LABEL** directive can be used to make a **FAR** jump. The label directive can be used to refer to the data segment along with the data type, byte or word.

```
DATA      SEGMENT
DATAS DB 50H DUP (?)
DATA-LAST  LABEL  BYTE  FAR
DATA ENDS
```

After reserving 50H locations for DATAS, the next location will be assigned a label DATA-LAST and its type will be byte and far.

19. LOCAL: The label, variables, constants or procedures declared LOCAL in a module are to be used only by that particular module. With a single declaration statement , a number of variables can be declared local, as shown

```
LOCAL a , b , DATA , ARRAY ,  ROUTINE
```

20. NAME: Logical Name Of A Module: The NAME directive is used to assign a name to an assembly language program module.

21. PTR: **Pointer:** The POINTER operator is used to declare the type of a label, variable or memory operand. The operator PTR is prefixed by either **BYTE** (8-bit quantity) or **WORD** (16-bit quantity).

```
Ex:   MOV AL, BYTE PTR [SI]
      INC WORD PTR [3000H]
```

22. TYPE: The TYPE operator directs the assembler to decide the data type of the specified label and replaces the TYPE label by the decided data type.

```
Ex:   MOV AX , TYPE STRING
```

23. GLOBAL: The labels, variables, constants or procedures declared **GLOBAL** may be used by other modules of the program.

```
Ex :  ROUTINE   PROC GLOBAL
```

24. '+ & -'Operators: These operators represent arithmetic addition and subtraction respectively. And are typically used to add or subtract displacements (8 or 16 bit) to base or index registers or stack or base pointers.

```
Ex:   MOV AL , [ SI +2 ]
      MOV DX , [ BX – 5 ]
      MOV BX , [ OFFSET LABEL + 10H ]
```

25. FAR PTR: This directive indicates the assembler that the label following **FAR PTR** is not available within the same segment and the address of the bit is of 32 bits i.e. 2 bytes offset followed by 2 bytes.

```
Ex:   JMP    FAR  PTR    LABEL
      CALL   FAR  PTR    ROUTINE
```

26. NEAR PTR: This directive indicates that the label following **NEAR PTR** is in the same segment and need only 16 bit i.e. 2 byte offset to address it. A **NEAR PTR** label is considered as default if a label is not preceded by NEAR PTR or FAR PTR.

```
Ex:   JMP    NEAR  PTR    LABEL
      CALL   NEAR  PTR    ROUTINE
```

27. SEG: Segment of a label: The SEG operator is used to decide the segment address of the label, variable, or procedure and substitutes the segment address in place of " SEG" label.

```
Ex:   MOV AX , SEG ARRAY ; ( This statement moves the segment address of array in which it is
      MOV DS , AX                  appearing , to register AX and then to DS
```

28. LENGTH: Byte Length of A Label: This directive is used to refer to the length of a data array or a string. It is Not available in MASM.

```
Ex: MOV  CX  ,  LENGTH ARRAY      ( This statement , when assembled , will substitute the length of
                                    the array in bytes , in the instruction )
```

2.11 ASSEMBLY LANGUAGE PROGRAM DEVELOPMENT TOOLS ?

The microprocessor development system consists of a set of hardware and software tools. The hardware development system usually contains a standard PC, Printer etc.

The software tools are also called program development tools. They are

- Assembler
- editor
- library builder
- linker
- debugger
- Simulator.
- Locator

EDITOR (TEXT EDITOR) :

- The editor is a software tool which allows the user to type /enter and modify the assembly language program.
- The editor provides a set of commands for insertion, deletion, modification of letters, characters, statements, etc.
- The main function of an editor is to help the user to construct the Assembly language program in the right format.
- The program created using editor is known as source program and usually it is saved with file extension ".ASM".

ASSEMBLER:

- The assembler is a software tool which converts the assembly language program to machine language program.
- The input for the assembler is the source program which is saved with file extension ".ASM". The assembler usually generates two output files called object file and list file.
- The object file consists of relocatable machine codes of the program and it is saved with file extension ".OBJ".
- The list file contains the assembly language statements& contains the binary code for each instruction and address of each instruction. The list file is saved with file extension ".LST".
- Some examples of assembler are TASM (Borland's turbo assembler), MASM (Microsoft's macro assembler), ASM86 (INTEL'S 8086 assembler), etc.

LIBRARY BUILDER :

- The library builder is used to create library files which are collection of procedures of frequently used function. Actually a library file is a collection of assembled objects files.
- While developing software for particular application, the programmers can link the library files in their programs.
- The input to library builder is set of assembled object file of program modules/procedures.
- The library builder combines the program modules/procedures into a single file known as library file and it is saved with file extension ".LIB".

LINKER:

- The linker is a software tool which is used to combine object files of program and library functions into a single executable file.
- The entire task of the program can be divided into the smaller task procedures. Each task can be developed individually. These procedures are called program modules

- Each module can be individually assembled, tested and debugged. Then the object files of program modules and the library files can be linked to get executable file
- The linker also generates a link map file which contains the address information about the linked files.
- Some examples of linkers are Microsoft's linker LINK, BORLAND'S turbo linker TLINK, etc.,

DEBUGGER:

- The debugger is a software tool that allows the execution of a program in single step or break-point mode under the control of user.
- The debugger allows the designer to look at the contents of registers and memory locations after running the program.
- A debugger also allows the user to set a breakpoint at any point in user program. When the user run the program, the pc will execute instructions up to this breakpoint and stop. The user can then examine register and memory contents to see whether the results are correct up to that point. If the results are correct, the user can move the breakpoint to a later point in the program.
- The process of identifying and correcting the errors on a program using a debugger is known as debugging.

SIMULATOR:

- The simulator is a program which can be run on the development system (pc) to simulate the operations of the newly designed system.
- Some of the operations of simulator are:
  - Execute a program and display the result.
  - Single step execution of a program.
  - Break-point execution of a program.
  - Display the contents of register memory.

EMULATOR:

- An emulator is a mixture of hardware and software.
- It is used to test and debug the hardware and software of a newly designed microprocessor based system.
- The emulator has a multi-core cable which connects the PC of development system and newly designed hardware of microprocessor system.
- Another powerful feature of an emulator is the ability to use either development system or the memory on the hardware under test for the program that is being debugged.

LOCATOR:

- A locator is a program used to assign the specific address of where the segments of object code are to be loaded into memory.
- A locator program called EXE2BIN comes with the IBM PC Disk Operating System (DOS). EXE2BIN converts a *.exe* file to a *.bin* file which has physical addresses.

Assignment Questions: ***Last date for submission: 20 – 12- 2019***

1. **Explain Addressing modes?**
2. **Explain the following instructions with 5 examples**
   a) data transfer b)arithmetic c) Logical d)string e) control transfer
3. **Define Assembly Language Program Development Tools ?**