ADDRESSING MODES OF 8086

Addressing mode indicate a way of locating data or operands. Depending upon the data types used in the instruction and the memory addressing modes, any instruction may belong to one or more addressing modes, or some instruction may not belong to any of the addressing modes. Thus addressing modes describe the types of operands and the way they are accessed for executing an instruction According to the flow of instruction execution the instruction may be categorized as

1.Sequential control flow instructions

2.Control transfer instructions

Sequential control flow instructions are the instructions which after execution, transfer control to the next instruction appearing immediately after it in the program. For example, the arithmetic, logical, data transfer and processor control instructions are sequential control flow instructions.

The **control transfer instructions**, on the other hand, transfer control to some predefined address or the address somehow specified in the instruction, after their execution. For example, INT, CALL, RET and JUMP instructions fall under this category

Classification of addressing modes

1.Immediate

2.Direct

3.Register

4.Register Indirect

5.Indexed

6.Register Relative

7.Based Indexed

8.Relative Based Indexed

9.Intrasegment Direct

10.Intrasegment Indirect

11.Intersegment Direct

12.Intersegment Indirect

1.Immediate addressing mode

In this type of addressing, immediate data is a part of instruction and appears in the form successive byte or bytes. In this mode, 8- or 16-bit data can be specified as part of the instruction - OP Code Immediate Operand

MOV AX, 0005H

Here 0005H is the immediate data.

MOV DX, 0525 Moves the 16 bit data 0525 H into DX

In the above two examples, the source operand is in immediate mode and the destination operand is in register mode.

A constant such as "VALUE" can be defined by the assembler EQUATE directive such as VALUE EQU 35H

Example: MOV BH, VALUE Used to load 35 H into BH

2.Direct addressing mode

Here 16-bit memory address (offset) is directly specified in the instruction.

Eg: MOV AX, [5000H]

Here, data resides in a memory location in the data segment, whose effective address may be computed using 5000H as the offset address and content of DS as segment address. Thus the effective address is 10H*DS+5000H

3.Register addressing mode

Here the data is stored in a register and it is referred using the particular register. All the registers, except IP, may be used in this mode.

Eg: MOV BX, AX

4.Register Indirect addressing mode

In this mode, the offset address of data is in either BX or SI or DI register. The default segment is either DS or ES.

Eg: MOV AX, [BX]

Here data is present in a memory location in DS whose offset address is in BX. The effective address of the data is given as 10H*DS+[BX]

5.Indexed addressing mode

In this addressing mode, offset of the operand is stored in one of the index registers. DS is the default segment for index registers SI and DI.

Eg: MOV AX, [SI]

Here data is available at an offset address stored in SI in DS. The effective address is computed as 10H*DS+[SI]

6.**Register Relative addressing mode**

In this mode, the data is available at an effective address formed by adding an 8-bit or 16-bit displacement with the content of any one of the registers BX, BP, SI and DI in the default (either DS or ES) segment.

Eg: MOV AX, 50H[BX]

Here the effective address is given as 10H*DS+50H+[BX]

7.Based Indexed addressing mode

In this mode, the effective address is formed by adding content of a base register (any one of BX or BP) to the content of an index register (any one of SI or DI). The default segment register may be ES or DS.

Eg: MOV AX, [BX][SI]

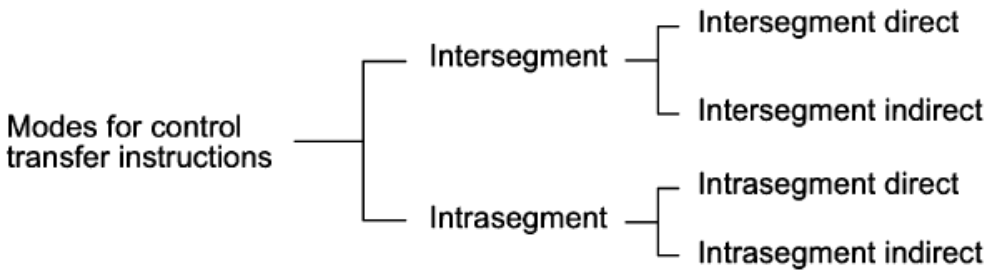
Here, BX is the base register and SI is the index register. The effective address is 10H*DS+[BX]+[SI]

8.**Relative Based Indexed addressing mode**

The effective address is formed by adding an 8 or 16 bit displacement with the sum of contents of any one of the base registers (BX or BP) and any one of the index registers, in a default segment.

Eg: MOV AX, 50H[BX][SI]

Here 50H is an immediate displacement, BX is a base register and SI is an index register. The effective address of data is computed as 10H*DS+[BX]+[SI]+50H



Addressing Modes for Control Transfer Instructions

There are two addressing modes for control transfer instructions

1. **Intrasegment mode**: If the destination location lies in the same segment the mode is called intrasegment
2. **Intersegment mode**: If the location to which the control is to be transferred lies in a different segment other than the current one the mode is called intersegment

9.Intrasegment direct mode

In this mode, the address to which the control is to be transferred lies in the same segment in which the control transfer instruction lies and appears directly in the instruction as an immediate displacement value. In this mode, the displacement is computed relative to the content of the IP.

The effective address to which the control will be transferred is given by the sum of 8 or 16 bit displacement and current content of IP.

In case of jump instruction, if the signed displacement(d) is of 8 bits (i.e, -128<d<+127), then it is **short jump**

if it is of 16 bits(i.e.32768<d<+32767), it is **long jump**.

```
JMP SHORT LABEL
```

Here LABEL lies lies within -128 to +127 from the current IP content

10.Intrasegment Indirect mode

In this mode, the displacement to which the control is to be transferred, is in the same segment in which the control transfer instruction lies, but it is passed to the instruction indirectly. Here the branch address is found as the content of a register or a memory location.

```
JMP [BX]          //Jump to effective address stored in BX
```

11.Intersegment Direct

In this mode, the address to which the control is to be transferred is in a different segment. This addressing mode provides a means of branching from one code segment to another code segment. Here, the CS and IP of the destination address are specified directly in the instruction.

```
JMP 5000H:2000H          //Jump to effective address 2000H in segment 5000H
```

12.Intersegment Indirect

In this mode, the address to which the control is to be transferred lies in a different segment and it is passed to the instruction indirectly. i.e contents of a memory block containing four bytes, i.e. IP(LSB), IP(MSB), CS(LSB) and CS(MSB) sequentially.

```
JMP [2000H]
```

INSTRUCTION SET OF 8086/8088

An instruction is a binary pattern designed inside a microprocessor to perform a specific function. The entire group of instructions that a microprocessor support is called Instruction Set. 8086 has more than 20,000 instructions.

Opcode: - It stands for operational code. It specifies the type of operation to be performed by CPU. It is the first field in the machine language instruction format. E.g. 08 is the opcode for instruction "MOV X,Y".

Operand: - We can also say it as data on which operation should act. operands may be register values or memory values. The CPU executes the instructions using information present in this field. It may be 8-bit data or 16-bit data.

Assembler: - it converts the instruction into sequence of binary bits, so that this bit can be read by the processor.

Mnemonics: - these are the symbolic codes for either instructions or commands to perform a particular function. E.g. MOV, ADD, SUB etc.

The instruction set of 8086 is classified into:

(1). Data Copy/Transfer instructions.

(2). Arithmetic & Logical instructions.

(3). Branch instructions.

(4). Loop instructions.

(5). Machine Control instructions.

(6). Flag Manipulation instructions.

(7). Shift & Rotate instructions.

(8). String instructions.

(1). Data copy/transfer instructions.

MOV Destination, Source

- The MOV instruction copies a word or byte of data from a specified source to a specified destination.
- The destination can be a register or a memory location..
- Destination can be register or memory operand.
- Both Source and Destination cannot be memory location or segment registers at the same time.
- They must both be of the same type (bytes or words)

E.g.

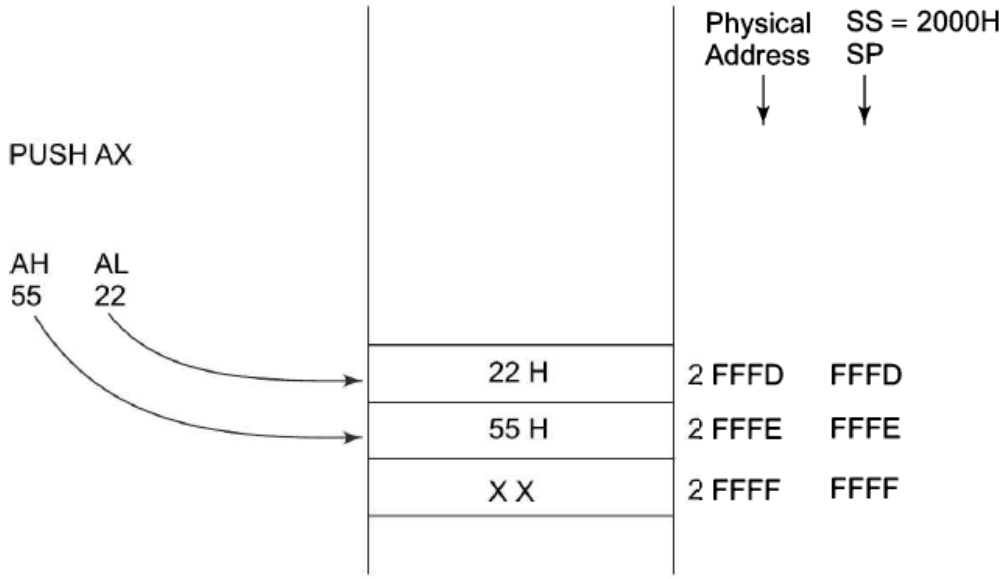
MOV CX, 037A H	Put immediate number 037AH to CX
MOV AL, BL	Copy content of register BL to AL
MOV BX, [0301 H]	Copy byte in DS at offset 031H to BX

PUSH Source

- Source can be register, segment register or memory.
- This instruction pushes the contents of specified source on to the stack.
- In this stack pointer is decremented by 2.
- The higher byte data is pushed first (SP-1).
- Then lower byte data is pushed (SP-2).

E.g.:

```
PUSH AX
PUSH DS
PUSH [5000H]
```

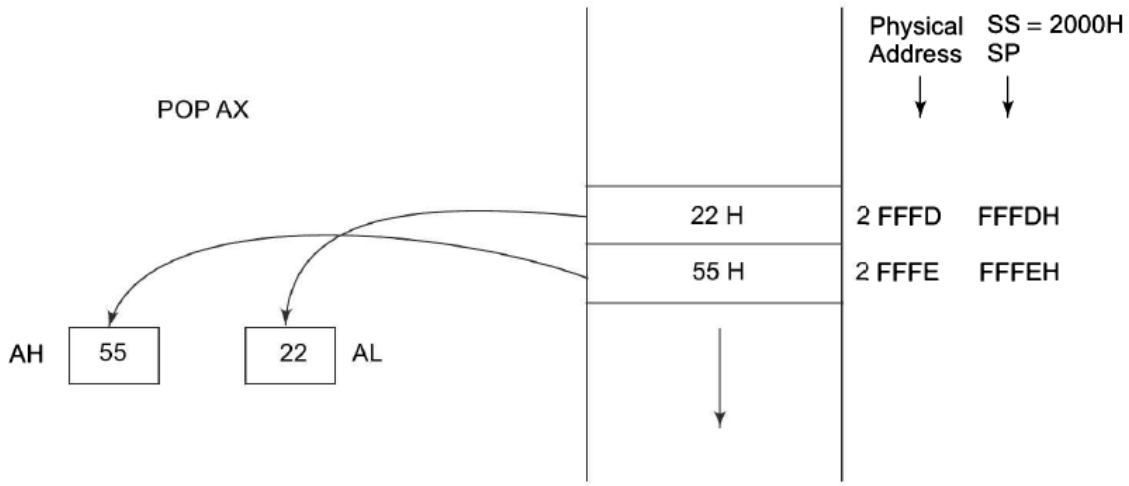


Pushing Data to Stack Memory

- The sequence of operation as below
    1. Current stack top is already occupied so decrement SP by one then store AH into the address pointed to by SP
    2. Further decrement SP by one and store AL into the location pointed to by SP

POP Destination

- Destination can be register, segment register or memory.
- This instruction pops (takes) the contents of specified destination.
- In this stack pointer is incremented by 2.
- The lower byte data is popped first (SP+1).
- Then higher byte data is popped (SP+2).



Popping Register Contents from Stack Memory

- The sequence of operation as below
    1. Content of stack top memory location is stored in AL and SP is incremented by one
    2. Further contents of memory location pointed to by SP are copied to Ah and SP is again incremented by 1

E.g.

POP AX
POP DS
POP [5000H]

XCHG Destination, source

- The XCHG instruction exchanges the content of a register with the content of another register or with the content of memory location(s).
- It cannot exchange two memory locations directly.
- The source and destination must both be of the same type (bytes or words).
- The segment registers cannot be used in this instruction. This instruction does not affect any flag.

E.g.

XCHG BX, AX
XCHG [5000H] ,AX

IN Accumulator, port address

- It reads from the specified port address.
- It copies data to accumulator from a port with 8-bit or 16-bit address.
- AL and AX are the allowed destinations
- DX is the only register is allowed to carry port address.

E.g.

IN AL, 80H
IN AX, DX //DX contains address of 16-bit port.

OUT 8-bit/16-bit port address, Accumulator

- It writes to the specified port address.
- The address of the output port may be specified in the instruction directly or implicitly in DX
- It copies contents of accumulator to the port with 8-bit or 16-bit address.
- The data to an odd addressed port is transferred on D_8-D_5 while that to an even addressed port is transferred on D_0 – D_7
- DX is the only register is allowed to carry port address.

E.g.

```
OUT 80H, AL;
OUT DX, AX;          //DX contains address of 16-bit port
```

XLAT

- Also known as translate instruction.
- It is used to find out codes in case of code conversion. i.e. it translates code of the key pressed to the corresponding 7-segment code.
- After execution this instruction contents of AL register always gets replaced.

E.g. XLAT

LEA source(16-bit register),destination(address)

- **LEA** Also known as **L**oad **E**ffective **A**ddress (**LEA**).
- It loads effective address formed by the destination into the source register.

E.g.

LEA BX, ADR	Effective address of label ADR ie,offset of address od ADR will be transferred to register BX
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 source, dest & **LES source, dest**

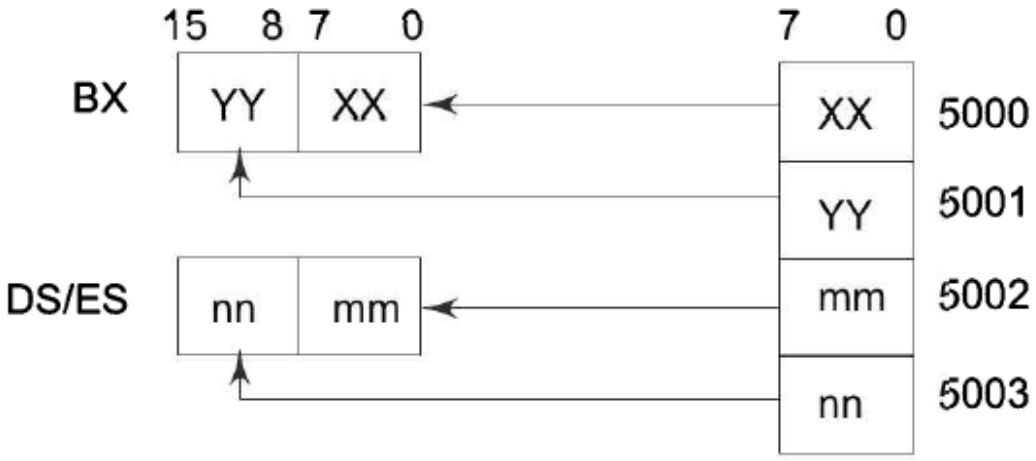
- **LDS** Also known as **L**oad **D**ata **S**egment (**LDS**).
- **LES** Also known as **L**oad **E**xtra **S**egment (**LES**).
- It loads the contents of DS (Data Segment) or ES (Extra Segment) & contents of the destination to the contents of source register.
- LDS does not affect any flag

E.g.

LDS BX, [4326]	Copy content of memory at displacement 4326H in DS to BL, content of 4327H to BH. Copy content at displacement of 4328H and 4329H in DS to DS register.

LDS BX,5000H

LES BX,5000H



LDS/LES Instruction Execution

LAHF(Load AH from Lower Byte of Flag)

- This instruction loads the AH register from the contents of lower byte of the flag register.
- This command is used to observe the status of the all-conditional flags of flag register.

E.g. LAHF

SAHF (Store AH to lower Byte of Flag Register)

- This instruction sets or resets all conditional flags of flag register with respect to the corresponding bit positions.
- If bit position in AH is 1 then related flag is set otherwise flag will be reset.

E.g.SAHF

PUSHF (Push Flags to stack)

- This instruction decrements the stack pointer by 2.
- It copies contents of flag register to the memory location pointed by stack pointer.
- Firs**t** the upper byte and then the lower byte is pushed on to it

E.g. PUSH F

POPF (Pop flags from stack)

- This instruction increments the stack pointer by 2.
- It copies contents of memory location pointed by stack pointer to the flag register.

E.g., POP F;

(2). Arithmetic Instructions

- These instructions perform the operations like:
- Addition,
- Subtraction,
- Increment,
- Decrement.

ADD destination, source

- This instruction adds the contents of source operand with the contents of destination operand.
- The source may be immediate data, memory location or register.
- The destination may be memory location or register.
- The result is stored in destination operand.
- AX is the default destination register.
- The source and the destination in an instruction cannot both be memory locations. The source and the destination must be of the same type (bytes or words)
- Flags affected: AF, CF, OF, SF, ZF

E.g.

```
ADD AX,2020H;     Immediate
ADD AX,BX;        Register
```

ADC destination, source

- Add with carry
- This instruction adds the contents of source operand with the contents of destination operand with carry flag bit.
- The source may be immediate data, memory location or register.
- The destination may be memory location or register.
- The result is stored in destination operand.
- AX is the default destination register.

E.g.

```
ADC AX,2020H;
ADC AX,BX
```

INC source

- This instruction increases the contents of source operand by 1.
- The source may be memory location or register.
- The source can not be immediate data.
- The result is stored in the same place
- All condition code flags ,except the carry flag, are affected depending upon the result

E.g.

```
INC AX;
INC [5000H];
```

DEC source;

- This instruction decreases the contents of source operand by 1.
- The source may be memory location or register.
- The source can not be immediate data.
- The result is stored in the same place.

E.g.

```
DEC AX;          Register
DEC [5000H];     Direct
```

SUB destination, source

- This instruction subtracts the contents of source operand from contents of destination.
- The source may be immediate data, memory location or register.,but source and destination operands both must not be memory operands
- The destination may be memory location or register.
- The result is stored in the destination place. All the condition code flags are affected by this instruction

E.g.

```
SUB AX,1000H;
SUB AX, BX;
```

SBB destination, source;

- Also known as Subtract with Borrow.
- This instruction subtracts the contents of source operand & borrow from contents of destination operand.
- The source may be immediate data, memory location or register.
- The destination may be memory location or register.
- The result is stored in the destination operand.

E.g.

```
SBB AX,1000H;
SBB AX, BX;
```

CMP destination, source

- Also known as Compare.
- This instruction compares the contents of source operand with the contents of destination operands.
- 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 destination operand. 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 source may be immediate data, memory location or register.
- The destination may be memory location or register.
- Then resulting carry & zero flag will be set or reset.

E.g.

```
CMP AX,1000H;
CMP AX, BX;
```

AAA

- Also known as **ASCII Adjust After Addition**.
- This instruction is executed after ADD instruction.
- AAA converts the resulting contents of AL to unpacked decimal digits.(In the case of unpacked BCD numbers, each four-bit BCD group corresponding to a decimal digit is stored in a separate register inside the machine)
- After addition AAA examines the contents of AL

(1). IF lower bits of AL<=09 and AF=0 then,

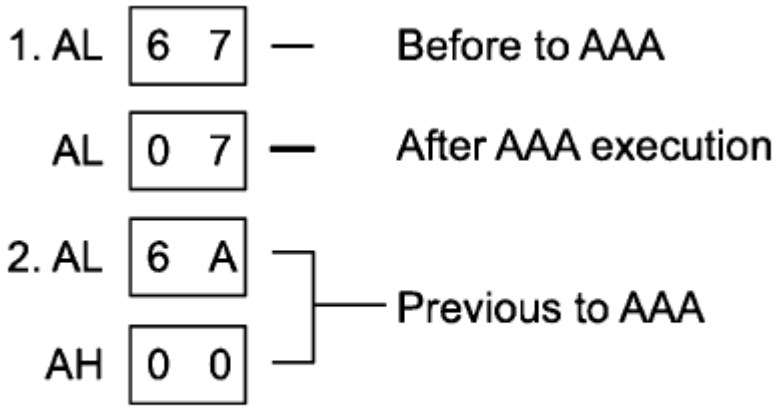
- Higher bits of AL should loaded with zeroes.
- There should be no change in lower bits of AL.
- AH also must be cleared (AH=0000 0000).

(2). IF lower bits of AL<=09 and AF=1 then,

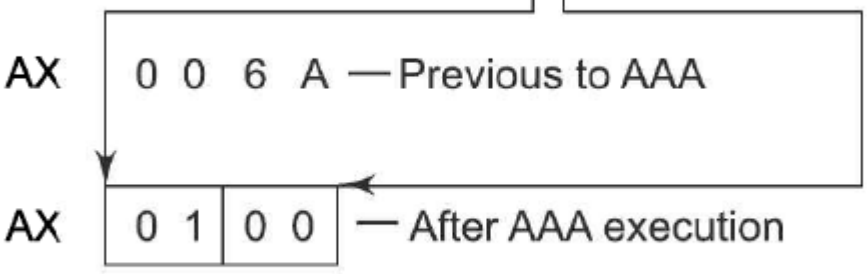
- .Al=AL+06
- Higher bits of AL should loaded with zeroes
- Bits of AH must be incremented by 01 (i.e. AH+0001).

(3)IF lower bits of AL>09 then,

- Bits of AL must be incremented by 06 (i.e. AL+0110).
- Bits of AH must be incremented by 01 (i.e. AH+0001).
- Then higher bits of AL should be loaded with 0000.
- The AAA instruction works only on the AL register. The AAA instruction updates AF and CF; but OF, PF, SF and ZF are left undefined.
- E.g.AAA;



A > 9, hence A + 6 = 1010 + 0110
= 10000 B
= 10 H → AF Set to 1



ASCII Adjust after Addition Instruction

AAS

- Also known as **ASCII Adjust After Subtraction**.
- This instruction is executed after SUB instruction.
- Numerical data coming into a computer from a terminal is usually in an ASCII code. In this code the numbers 0 to 9 are represented by the ASCII codes 30H to 39H. The 8086 allows you to subtract the ASCII codes for two decimal digits without masking

the "3" in the upper nibble of each. The AAS instruction is then used to make sure the result is the correct unpacked BCD

(1). IF lower bits of AL<=09 then,

- Higher bits of AL should loaded with zeroes.
- There should be no change in lower bits of AL.
- AH also must be cleared (AH=0000 0000).

(2). IF lower bits of AL>09 then,

- Bits of AL must be decremented by 06 (i.e. AL-0110).
- Bits of AH must be decremented by 01 (i.e. AH-0001).
- Then higher bits of AL should be loaded with 0000.

E.g. (1). AAS;

AAM

- Also known as **ASCII Adjust After Multiplication**.
- This instruction is executed after MUL instruction.
- Then AH=AL/10 & AL=Remainder.
- E.g.

```
MOV AL,04        // AL=04
MOV BL,09        //  BL=09
MUL BL           //   04*09=36 (i.e. BL*AL)
 AAM             //    AH=03 & AL=06
```

AAD

- It is also known as **ASCII adjust before Division**
- AAD instruction converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL

Then AL=AH*10 +AL & AH=0.

E.g.

```
MOV AX, 0105     // AH=01, AL=05
AAD              //  AL=15 (i.e.0FH) & AH=00
```

DAA

- Decimal Adjust Accumulator.
- Used to convert the result of addition of two unpacked BCD numbers to a valid BCD number.
- IF lower bits of AL>09.
  Then AL=AL+06.
- If now upper nibble 0f AL >9,DAA adds 60 H to AL

E.g.

```
MOV AL,53H       //AL=53H
MOV CL,29H       //CL=29H
ADD AL,CL        // AL=7CH   (i.e. 12=C) & C>9.
 DAA             //  AL=7C+06=82H. (i.e. 0111 1100 + 0000 0110)=1000 0010
```

Example 2.33

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

(ii) AL = 73          CL = 29
     ADD AL, CL       ; AL ← AL + CL
                      ; AL ← 73 + 29
                      ; AL ← 9C
     DAA              ; AL ← 02 and CF = 1
                       AL = 7 3
                            +
                      CL = 2 9
                           9 C
                           + 6
                           A 2
                         + 6 0
                    CF = 1 0 2  in AL
```

DAS

- Decimal Adjust after Subtraction.
- IF lower bits of AL>09.
- Then AL=AL-06.

E.g.

```
MOV AL,30H      // AL=30H
MOV CL,20H      // CL=20H
SUB AL,CL       // AL=0AH   (i.e. A=10) & C=10>9.
DAS             //  AL=0A-06=04H. (i.e. 0000 1010 - 0000 0110)=0000 0100
```

MUL operand

- Unsigned Multiplication.
- Operand contents are positively signed.
- Operand may be general purpose register or memory location.
- If operand is of 8-bit then multiply it with contents of AL.
- If operand is of 16-bit then multiply it with contents of AX. And MSB bit of result is stored in DX and LSB stored in AX
- Result is stored in accumulator (AX).

E.g.

```
MUL BH          // (AX)= AL*BH; // (+3) * (+4) = +12.
MUL CX          //(DX) (AX)=AX*CX;
```

IMUL operand

- Signed Multiplication.
- Operand contents are negatively signed.
- Operand may be general purpose register, memory location or index register.
- If operand is of 8-bit then multiply it with contents of AL.
- If operand is of 16-bit then multiply it with contents of AX.
- Result is stored in accumulator (AX). And LSB in DX

E.g.

```
IMUL BH                 // AX= AL*BH;        // (-3) * (-4) = 12.
IMUL CX                 // AX=AX*CX;
```

DIV operand

- Unsigned Division.
- Operand may be register or memory.
- Operand contents are positively signed.
- Operand may be general purpose register or memory location.
- For 16 bit dividend: AL-Quotient,AH-Reminder
- For 32 bit dividend: higher order bits in DX and lower order bits in AX: Quotient in AX and reminder in DX
- AL=AX/Operand (8-bit/16-bit) & AH=Remainder.

E.g.

```
MOV AX, 0203        // AX=0203
MOV BL, 04          //  BL=04
DIV BL              //   AL=0203/04=50 (i.e. AL=50 & AH=03)
```

IDIV operand

- Signed Division.
- Operand may be register or memory.
- Operand contents are negatively signed.
- Operand may be general purpose register or memory location.
- AL=AX/Operand (8-bit/16-bit) & AH=Remainder.

E.g.

```
MOV AX, -0203       // AX=-0203
MOV BL, 04          //  BL=04
DIV BL              //   AL=-0203/04=-50 (i.e. AL=-50 & AH=03)
```

NEG Src:

- —It creates 2's complement of a given number.
- —That means, it changes the sign of a number.
- For obtaining 2's complement it substracts the contents of destination from zero
- The result is stored back in the destination operand which may be a register or a memory location
- If OF is set it indicates that the operation could not be completed successfully
- This instruction affects all the condition code flags

CBW (Convert Byte to Word):

- This instruction converts byte in AL to word in AX.
- The conversion is done by extending the sign bit of AL throughout AH.
- It does not affect any flag

CWD (Convert Word to Double Word):

- This instruction converts word in AX to double word in DX : AX.
- The conversion is done by extending the sign bit of AX throughout DX.

- It does not affect any flag

(3). Logical Instructions

AND destination,source

- This instruction ANDs each bit in a source byte or word with the same numbered bit in a destination byte or word. The result is put in the specified destination. The content of the specified source is not changed
- Destination operand may be register, memory location.
- Source operand may be register, immediate data or memory location.
- Result is stored in destination operand.

E.g.

```
MOV AX, 3F0FH          //  AX=3F0FH
MOV BX, 0008H          //  BX=0008H
AND AX,BX              //    AX=0008H
```

- Follow the rules as given below:-

(1). 1 AND 1 = 1
(2). 1 AND 0 = 0
(3). 0 AND 1 = 0
(4). 0 AND 0 = 0

3F0FH= 0011 1111 0000 1111
0008H= 0000 0000 0000 100

```
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 destination,source

- Destination operand may be register, memory location.
- Source operand may be register, immediate data or memory location.
- Result is stored in destination operand.

E.g.

```
MOV AX, 3F0FH       //  AX=3F0FH
MOV BX, 0098H       //  BX=0098H
OR AX,BX            //    AX=3F9FH
```

- Follow the rules as given below:-

(1). 1 OR 1 = 1
(2). 1 OR 0 = 1
(3). 0 OR 1 = 1
(4). 0 OR 0 = 0

3F0FH= 0011 1111 0000 1111
0098H=0000 0000 1001 1000

```
0 0 1 1     1 1 1 1     0 0 0 0     1 1 1 1     = 3F0F H
↓ ↓ ↓ ↓     ↓ ↓ ↓ ↓     ↓ ↓ ↓ ↓     ↓ ↓ ↓ ↓       OR
0 0 0 0     0 0 0 0     1 0 0 1     1 0 0 0     = 0098 H
-------------------------------------------------------
0 0 1 1     1 1 1 1     1 0 0 1     1 1 1 1     = 3F9F H
```

Thus the result 3F9FH will be stored in the AX register.

NOT operand;

- Operand may be register, memory location.
- This instruction inverts (complements) the contents of given operand.
- Result is stored in Accumulator (AX).

E.g.

```
MOV AX, 0200FH      //  AX=200FH
NOT AX              //  AX=DFF0H
```

- Follow the rules as given below:-

(1). 1 NOT = 0
(2). 0 NOT = 1

200FH= 0010 0000 0000 1111

```
   AX     = 0 0 1 0     0 0 0 0     0 0 0 0     1 1 1 1
invert       ↓ ↓ ↓ ↓     ↓ ↓ ↓ ↓     ↓ ↓ ↓ ↓     ↓ ↓ ↓ ↓
             1 1 0 1     1 1 1 1     1 1 1 1     0 0 0 0
```

```
Result
in AX =          D           F           F           0
```

The result DFF0H will be stored in the destination register AX.

XOR Des, Src:

- It performs XOR operation of Des and Src.
- The XOR operation gives a high output ,when the 2 input bits are dissimilar
- Src can be immediate number, register or memory location.
- Des can be register or memory location.
- Both operands cannot be memory locations at the same time.
- CF and OF become zero after the operation.
- PF, SF and ZF are updated.

Eg:

```
MOV AX,3F0FH

XOR AX,0098H
```

```
AX = 3F0FH =     0 0 1 1     1 1 1 1     0 0 0 0     1 1 1 1
       XOR       ↓ ↓ ↓ ↓     ↓ ↓ ↓ ↓     ↓ ↓ ↓ ↓     ↓ ↓ ↓ ↓
     0098H =     0 0 0 0     0 0 0 0     1 0 0 1     1 0 0 0
AX = Result =    0 0 1 1     1 1 1 1     1 0 0 1     0 1 1 1
          = 3F97H
```

TEST destination,source

- Both operands may be register, memory location or immediate data.
- This instruction performs bit by bit logical AND operation for flags only (i.e. only flags will be affected).
- If the corresponding 0^{th} bit of result contains '1' then result will be non-zero & zero flag will be cleared/reset (i.e. ZF=0).
- If the corresponding 0^{th} bit of result contains '0' then result will be zero & zero flag will be set (i.e. ZF=1)..

E.g.

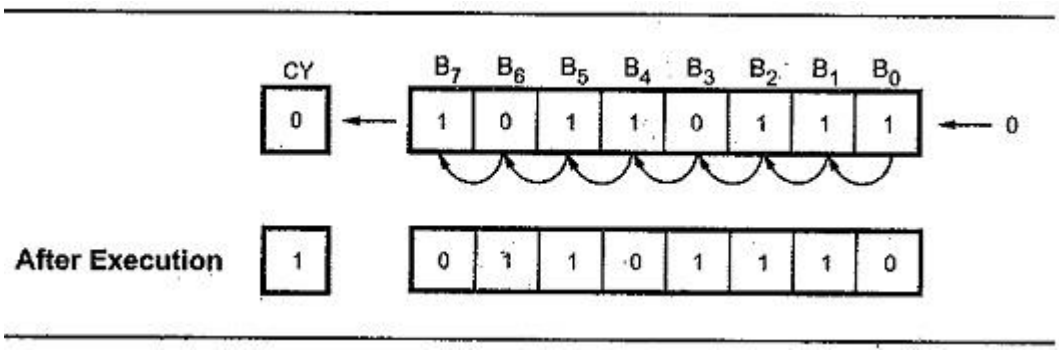
```
TEST AX,BX
TEST [0500],06H
```

SAL/SHL: SAL / SHL destination, count

- Shift logical/Arithmetic left
- SAL and SHL are two mnemonics for the same instruction.
- This instruction shifts each bit in the specified destination to the left and 0 is stored at LSB position. The MSB is shifted into the carry flag.
- The destination can be a byte or a word. It can be in a register or in a memory location. The number of shifts is indicated by count
- The operand may reside in a register or a memory location but cannot be an immediate data

Eg.

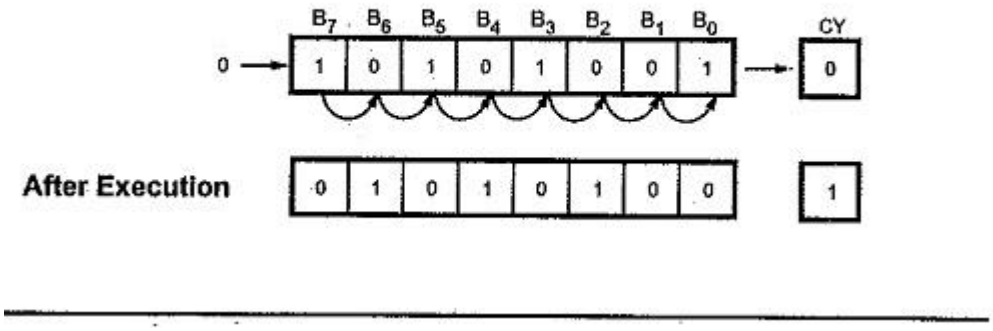
```
SAL CX, 1
SAL AX, CL
```



SHR destination, count

- Shift logical Right
- This instruction shifts each bit in the specified destination to the right and 0 is stored at MSB position. The LSB is shifted into the carry flag. The destination can be a byte or a word
- It can be a register or in a memory location. The number of shifts is indicated by count

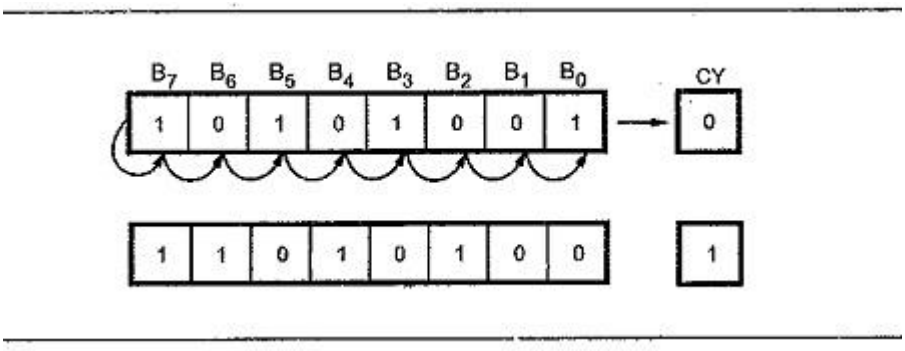
Eg. SHR CX, 1



SAR destination, count

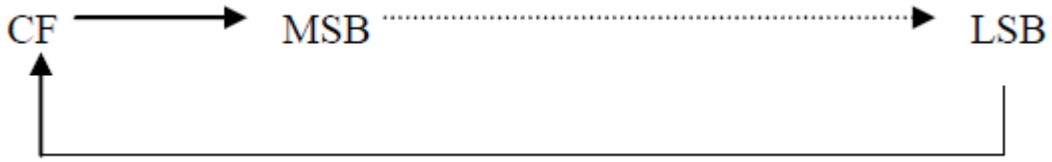
- Shift Arithemetic Right
- This instruction shifts each bit in the specified destination some number of bit positions to the right. As a bit is shifted out of the MSB position, a copy of the old MSB is put in the MSB position. The LSB will be shifted into CF.

Eg. SAR BL, 1



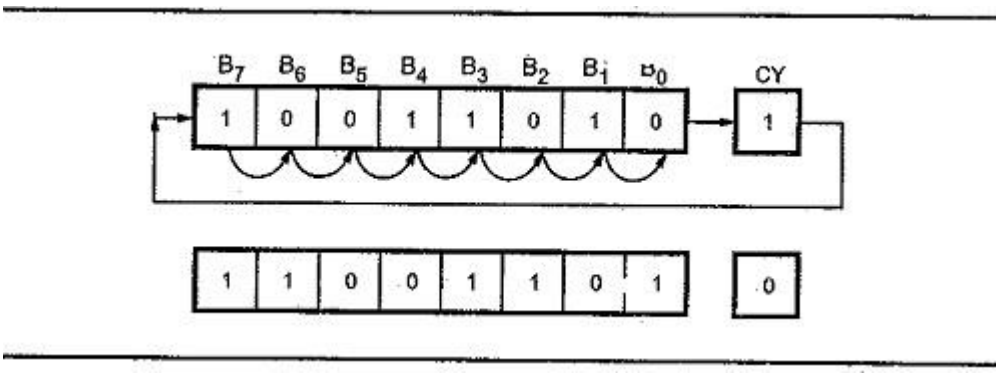
RCR

- Also known as Rotate Right through Carry.
- Each binary bit of the operand is rotated towards right by one position through Carry flag.
- Least Significant Bit (LSB) is placed in the Carry flag.
- Then carry flag bit is placed in the Most Significant Bit (MSB) position



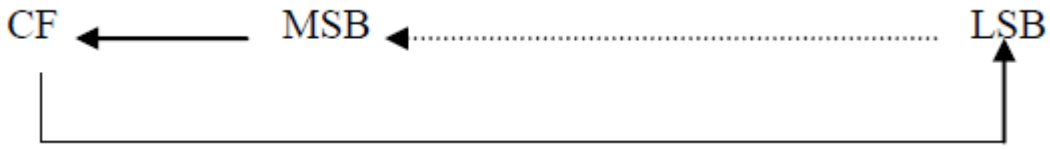
Eg

RCR BX, 1 //Word in BX right 1 bit, CF to MSB, LSB to CF



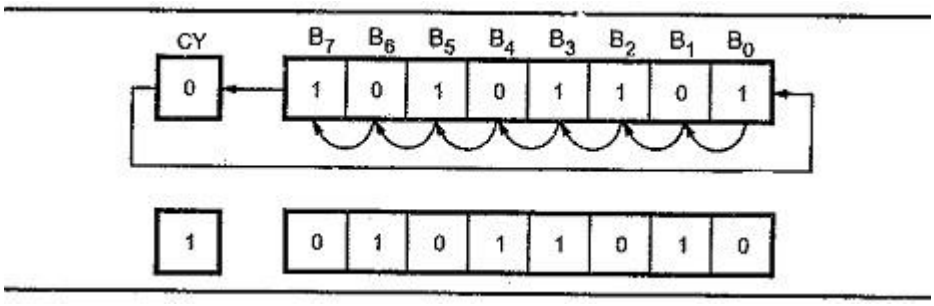
RCL Destination, Count

- Also known as Rotate Left through Carry.
- Each binary bit of the operand is rotated towards left by one position through Carry flag.
- Least Significant Bit (LSB) of operand is placed in next position.
- Then Most Significant Bit (MSB) of operand is placed in carry flag bit.
- If we want to rotate the operand by one bit position, you can specify this by putting a 1 in the count position of the instruction. To rotate by more than one bit position, load the desired number into the CL register and put "CL" in the count position of the instruction



Eg:

RCL DX, 1 // Word in DX 1 bit left, MSB to CF, CF to LSB



ROR destination, count

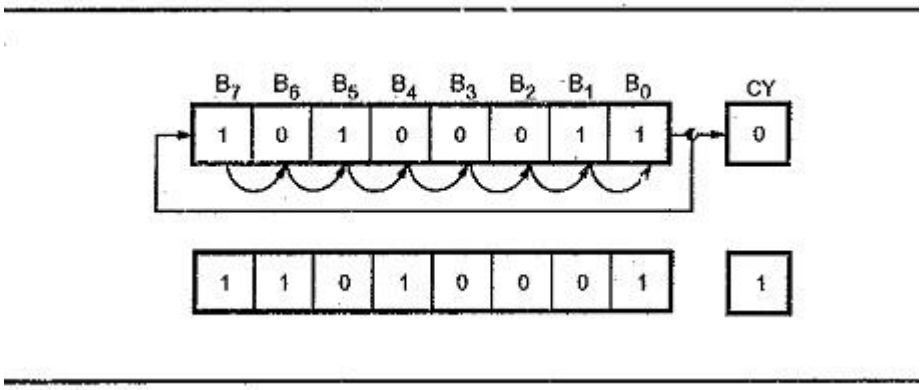
- Rotate Right without carry
- This instruction rotates all bits in a specified byte or word to the right some number of bit positions. LSB is placed as a new MSB and a new CF.
- The LSB bit is pushed into the carry flag and simultaneously it is transferred into the MSB bit position at each operation. The remaining bits are shifted right by the specified position

Eg.

ROR CX, 1

MOV CL, 03H

ROR BL, CL



ROL destination, count

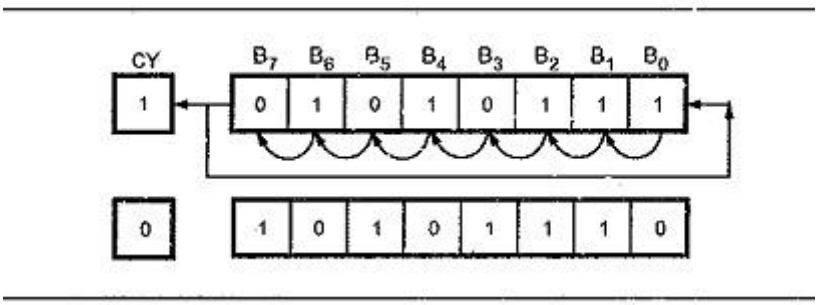
- Rotate Left without carry
- This instruction rotates all bits in a specified byte or word to the left some number of bit positions. MSB is placed as a new LSB and a new CF.

Eg.

ROL CX, 1

MOV CL, 03H

ROL BL, CL



(4). String Manipulation Instructions

REP

- Also known as Repeat instruction prefix.
- REP is a prefix, which is written before one of the string instructions
- This instruction executed repeatedly until the 'CX' register becomes zero.
- When 'CX' becomes zero then program control passes to next instruction.
- There are following sub types of 'REP' instruction,

(i). REPE:- REPeat instruction while Equal.
(ii). REPZ:- REPeat instruction while Zero.
(iii). REPNE:- REPeat instruction while Not Equal.
(iv). REPNZ:- REPeat instruction while Not Zero.

CMPS

- Also known as Compare String Byte or String Word.
- This instruction is used to compare two strings of bytes or words
- The length of the string must be stored in register CX.
- If both the byte or word are equal then zero flag will be set (i.e. ZF=1) otherwise it will be reset (i.e. ZF=0).
- When zero flag will be set then 'CX'=0.
- 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
- There are following sub types,
  (i). CMPSB:- Compare String Byte.
  (ii). CMPSW:- Compare String Word.

Eg:

SCAS String:

- It scans a string. It compares the String with byte in AL or with word in AX.
- The string 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 ,as stated in case of MOVSB instruction
- Whenever a match to the specified operand is found in the string execution stops and the zero flag is set .If no match is found the zero flag is reset
- The REPNE prefix is used with the SCAS instruction

```
  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
```

- The string of instruction finds out if it contains WORD .It WORD is found in the word sring before CX become zero the ZF is set otherwise the ZF is reset.The scanning will continue till a match is found

MOVS / MOVSB / MOVSW:

- Move string ,Byte or string word
- It causes moving of byte or word from one string to another.
- In this instruction, the source string is in Data Segment(DS) and destination string is in Extra Segment(ES).
- SI and DI store the offset values for source and destination index.
- The starting address of the source string is 10H*DS+[SI] while starting address of the destination string is 10H*ES+[DI]
- The MOVSB/MOVSW instruction thus moves a string of bytes/words pointed by DS:SI pair to the memory location pointed to by ES:DI pair

LODS

- Load String Byte or String Word
- The LODS instruction loads the AL/AX register by the contents of a string pointed to by DS:SI register pair
- The SI is modified automatically depending upon DF
- If it is a byte transfer (LODSB) the SI id modified by one and if it is a word transfer(LODSW) the SI modified by two

STOS

- Store String BYTE or Word
- The STOS instruction stores the AL/AX register contents to a location in the string pointed by ES:DI register pair

(5)Control Transfer or Branching Instructions (Control Instructions)

Branch Instructions transfers the flow of execution of the program to a new address specified in the 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 to be transferred.

The Branch Instructions are classified into two types

1. Unconditional Branch Instructions.
2. Conditional Branch Instructions.

Unconditional Branch Instructions:

In 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.

Conditional Branch Instructions

When this instruction is executed, execution control is transferred to the address specified relatively in the instruction, provided the condition implicit in the Opcode is satisfied. Otherwise execution continues sequentially.

CALL

- Also known as unconditional call.
- Under unconditional call, the execution control is transferred to the specified location independent of any status or condition.
- This instruction is used to call subroutine from a main program.
- The address of the procedure may be specified directly or indirectly depending upon the the addressing mode
- There are following sub types,
  (i). **NEAR CALL**:- Procedure is available in the same segment ie, ±32K displacement It pushes only IP into the stack.
- When the 8086 executes a near CALL instruction, it decrements the stack pointer by 2 and copies the offset of the next instruction after the CALL into the stack. This offset saved in the stack is referred to as the return address, because this is the address that execution will return to after the procedure is executed. A near CALL instruction will also load the instruction pointer with the offset of the first instruction in the procedure.

A RET instruction at the end of the procedure will return execution to the offset saved on the stack which is copied back to IP.

(ii). **FAR CALL**:- Procedure is available in the another segment .It pushes IP & CS into the stack.

- When the 8086 executes a far call, it decrements the stack pointer by 2 and copies the content of the CS register to the stack. It then decrements the stack pointer by 2 again and copies the offset of the instruction after the CALL instruction to the stack. Finally, it loads CS with the segment base of the segment that contains the procedure, and loads IP with the offset of the first instruction of the procedure in that segment. A RET instruction at the end of the procedure will return execution to the next instruction after the CALL by restoring the saved values of CS and IP from the stack

RET:

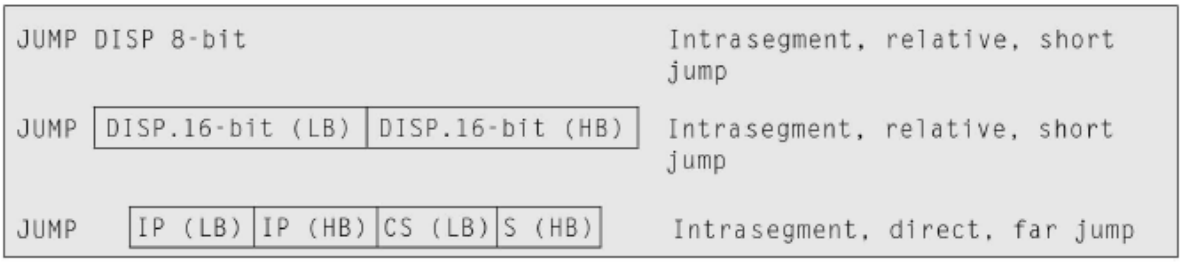
- It returns the control from procedure to calling program.
- At each CALL instruction .the IP and CS of the next instruction is pushed onto stack ,before the control is transferred to the procedure .At the end of the procedure ,the RET instruction must be executed .When it is executed the previously stored contents of IP and CS along with flags are retrieved into to CS,IP and flag registers from the stack and execution of the main program continues
- In case of FAR procedure the current contents of SP points to IP and CS at the time of return
- In case of NEAR ,it points to IP only
- Every CALL instruction should have a RET.

The RET instruction is of four types

- 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

JMP

- Also known as unconditional jump.
- Under unconditional jump, the execution control is transferred to the specified location using 8-bit or 16-bit displacement(intrasegment ,relative ,short or long)
- There are following three formats of jump instruction,



INT N: Interrupt Type N.

- In the interrupt structure of 8086, 256 interrupts are defined corresponding to the types from 00H to FFH. When 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 memory block in 0000 segment.

Thus the instruction INT 20H will find out the address of the interrupt service routine

```
INT      20H
Type* 4 = 20 * 4 = 80H
```

Pointer to IP and CS of the ISR is 0000 : 0080 H

INTO: Interrupt on Overflow

- This instruction is executed, when the overflow flag OF is set. This is equivalent to a Type 4 Interrupt instruction
- The new contents of IP and CS are taken from the address 0000:0010.This is equivalent to a Type 4 interrupt instruction

IRET: Return from ISR

- When an inerrupt sevice routine is to be called before transferring control to it ,the IP ,CS and flagregister are stored on to the stack to indicate the location from where the execution is to be continued ,after the ISR is executed
- So at the end of each ISR ,when IRET is exeuted the values of IP, CS and Flags are retrieved from the stack to continue the execution of the main program.

LOOP

- Jump To Specified Label If Cx not equal to 0 After Auto Decrement
- This instruction is used to repeat a series of instructions some number of times. The number of times the instruction sequence is to be repeated is loaded into CX. Each time the LOOP instruction executes, CX is automatically decremented by 1. If CX is not 0, execution will jump to a destination specified by a label in the instruction. If CX = 0 after the auto decrement, execution will simply go on to the next instruction after LOOP.
- The destination address for the jump must be in the range of –128 bytes to +127 bytes from the address of the instruction after the LOOP instruction.
- This instruction does not affect any flag

```
        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
```

Conditional jump instruction

	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).

- The last four operations are used in case of decisions based on signed binary number operations

```
JCXZ    'Label'   Transfer execution control
                  to address 'Label', if CX=0.
```

Conditional Loop Instructions

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

(6). Flag Manipulation Instructions and processor control instruction

- These instruction control the functioning of the available hardware inside the processor chip
- These are categorized into two types

1. Flag manipulation instruction: It directly modifies some of the flags of 8086
2. Machine control instruction : It controls the bus usage and execution

CMC

- Also known as Complement Carry Flag.
- It inverts contents of carry flag.
- if CF = 1 then CF will be = 0.
- if CF = 0 then CF will be = 1.

E.g. CMC

STC

- Also known as Set Carry Flag.
- It makes carry flag in set condition.
- After execution CF = 1.

E.g. STC

CLI

- Also known as Clear Interrupt Flag.
- It makes interrupt flag in reset condition.
- After execution IF = 0.

E.g. CLI

CLD

- Also known as Clear Direction Flag.
- It makes direction flag in reset condition.
- After execution DF = 0.

E.g. CLD

Flag Manipulation Instructions

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

(7). Machine Control Instructions

HLT

- Also known as Halt
- It makes the processor to be in stable (do nothing) condition.
- The HLT instruction causes the 8086 to stop fetching and executing instructions. The 8086 will enter a halt state. The different ways to get the processor out of the halt state are with an interrupt signal on the INTR pin, an interrupt signal on the NMI pin, or a reset signal on the RESET input

E.g. HLT

NOP (PERFORM NO OPERATION)

- Also known as No Operation.
- It tells about further there will be no operation to be performed.
- This instruction simply uses up three clock cycles and increments the instruction pointer to point to the next instruction. The NOP instruction can be used to increase the delay of a delay loop. When hand coding, a NOP can also be used to hold a place in a program for an instruction that will be added later. NOP does not affect any flag

E.g. NOP

ESC (ESCAPE)

- This instruction is used to pass instructions to a coprocessor
- Instructions for the coprocessor are represented by a 6-bit code embedded in the ESC instruction.
- As the 8086 fetches instruction bytes, the coprocessor also fetches these bytes from the data bus and puts them in its queue. However, the coprocessor treats all the normal 8086 instructions as NOPs.
- When 8086 fetches an ESC instruction, the coprocessor decodes the instruction and carries out the action specified by the 6-bit code specified in the instruction.
- In most cases, the 8086 treats the ESC instruction as a NOP. In some cases, the 8086 will access a data item in memory for the coprocessor.

Eg: ESC

LOCK

- Each microprocessor has its own local buses and memory. The individual microprocessors are connected together by a system bus so that each can access system resources such as disk drive or memory. Each microprocessor takes control of the system bus only when it needs to access some system resources. The LOCK prefix allows a microprocessor to make sure that another processor does not take control of the system bus while it is in the middle of a critical instruction, which uses the system bus
- When it is executed the bus access is not allowed for another master till the lock prefixed instruction is executed completly

Machine Control Instructions		
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.

ASSEMBLER DIRECTIVES AND OPERATORS

The source program is composed of three types of lines: opcode, assemble directive(pseudo opcodes) & the comments. The opcode create the instruction for the CPU. Assembler directives give instruction to the assembler. Comments tells us what the program wants to tell us. Assembler directives are specific for a particular assembler. They are used only during the assembly of program & generate machine executable code.

An assembler is a program used to convert an assembly language program into the equivalent machine code modules which may further be converted to executable codes. The assembler decides the address of each label and substitutes the values for each of the constants and variables. It then forms the machine code for the mnemonics and data in the assembly language program. While doing these things, the assembler may find out syntax errors. The logical errors or other programming errors are not found out by the assembler. For completing all these tasks, an assembler needs some hints from the programmer, i.e. the required storage for a particular constant or a variable, logical names of the segments, types of the different routines and modules, end of file, etc. These, types of hints are given to the assembler using some predefined alphabetical strings called assembler directives. Assembler directives help the assembler to correctly understand the assembly language programs to prepare the codes.

Another type of hint which helps the assembler to assign a particular constant with a label or initialize particular memory locations or labels with constants is called an operator. Rather, the operators perform the arithmetic and logical tasks unlike directives that just direct the assembler to correctly interpret the program to code it appropriately. The following directives are commonly used in the assembly language programming practice using Microsoft Macro Assembler (MASM) or Turbo Assembler (TASM).

ASSUME(Assume Logical Segment Name)
The ASSUME directive is used to inform the assembler, the names of the logical segments to be assumed for different segments used in the program. In the assembly language program, each segment is given a name. For example, the code segment may be given the name CODE, data segment may be given the name DATA etc. The statement ASSUME CS: CODE directs the assembler that the machine codes are available in a segment named CODE, and hence the CS register is to be loaded with the address (segment) allotted by the operating system for the label CODE, while loading. Similarly, ASSUME DS: DATA indicates to the assembler that the data items related to the program, are available in a logical segment named DATA, and the DS register is to be initialized by the segment address value decided by the operating system

for the data segment, while loading. It then considers the segment DATA as a default data segment for each memory operation, related to the data and the segment CODE as a source segment for the machine codes of the program. The ASSUME statement is a must at the starting of each assembly language program,

DB(DEFINE BYTE)

The DB directive is used to reserve byte or bytes of memory locations in the available memory. While preparing the EXE file, this directive directs the assembler to allocate the specified number of memory bytes to the said data type that may be a constant, variable, string, etc. Another option of this directive also initializes the reserved memory bytes with the ASCII codes of the characters specified as a string. The following examples show how the DB directive is used for different purposes.

```
LIST DB 0lH, 02H, 03H, 04H
```

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

```
MESSAGE DB 'GOOD MORNING'
```

This makes the assembler reserve the number of bytes of memory equal to the number of characters in the string named MESSAGE and initialize those locations by the ASCII equivalent of these characters.

DW(DEFINE WORD)

This directive is used to tell the assembler to declare a variable of type word or to reserve storage locations of type word in memory. The DW 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. Some examples are given to explain this directive.

```
WORDS   DW 1234H, 4567H
```

This makes the assembler reserve four words in memory (8 bytes), and initialize the words with the specified values in the statements. During initialisation, the lower bytes are stored at the lower memory addresses, while the upper bytes are stored at the higher addresses. Another option of the DW directive is explained with the DUP operator.

```
WDATA   DW    5   DUP  (6666H
```

This statement reserves five words, i.e. 10-bytes of memory for a word label WDATA and initializes all the word locations with 6666H.

DD(DEFINE DOUBLE WORD)

This directive is used to declare a variable of type double word or to reserve memory locations which can be accessed as type double word (Define Doubleword).

DQ (**DEFINED QUAD WORD**)

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

DT (DEFINE TEN BYTES)

The DT directive directs the assembler to define the specified variable requiring la-bytes for its storage and initialise the 10bytes with the specified values. The directive may be used in case of variables facing heavy numerical calculations, generally processed by numerical processors.

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 clearly.

```
DATA SEGMENT
.
.
.
DATA ENDS
ASSUME CS: CODE, DS:DATA
CODE SEGMENT.
.
.
.
CODE ENDS
END
```

The above structure represents a simple program containing two segments named DATA and CODE. The data related to the program must lie between the DATA SEGMENT and DATA ENDS statements. Similarly, all the executable instructions must lie between CODE SEGMENT and CODE ENDS statements.

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 place of that numerical value, throughout the program. While assembling, whenever the assembler comes across the label, it substitutes the numerical value for that label and finds out the equivalent code.

Using the EQU directive, even an instruction mnemonic can be assigned with a label, and the label can then be used in the program in place of the at mnemonic. Suppose, a numerical constant appears 'in a program ten times. If that constant is to be changed at a later time, one will have to make all these ten corrections. This may lead to human errors, because it is possible that a human programmer may miss one of those corrections. This will result in the generation of wrong codes. If the EQU directive is used to assign the value with a label that can be used in place of each recurrence of that constant, only one change in the

EQU statement will give the correct and modified code. The examples given below show the syntax**.**

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

The first statement assigns the constant 500H with the label LABEL, while the second statement assigns another label ADDITION with mnemonic ADD

END(END of Program)

The END directive marks the end of an assembly language program. When the assembler comes across this END directive, it ignores the source lines available later on. Hence, it should be ensured that the END statement should be the last statement in the file and should not appear in between. No useful program statement should lie in the file, after the END statement

ENDP(END of Procedure)

In assembly language programming, the subroutines are called procedures. Thus, procedures may be independent program modules which return particular results or values to the calling programs. The ENDP directive is used to indicate the end of a procedure. A procedure is usually assigned a name, i.e. label. To mark the end of a particular procedure, the name of the procedure, i.e. label may appear as a prefix with the directive ENDP. The statements, appearing in the same module but after the ENDP directive, are neglected from that procedure. The structure given below explains the use of ENDP.

```
PROCEDURE STAR

STAR ENDP
```

EVEN(Align on Even Memory Address)

The assembler, while starting the assembling procedure of any program, initializes a location counter and goes on updating it, as the assembly proceeds. It goes on assigning the available addresses, i.e. the contents of the location counter, sequentially to the program variables, constants and modules as per their requirements, in the sequence in which they appear in the program. 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. The structure given below explains the directive.constant to that address. The structure given below explains the directive.

```
EVEN
PROCEDURE ROOT
.
.
.
ROOT ENDP
```

The above structure shows a procedure ROOT that is to be aligned 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 ROOT procedure will be assigned with the same address.

GROUP (Group the Related segment)

The directive is used to form logical groups of segments with similar purpose or type. This directive is used to inform the assembler to form a logical group of the following segment names. The assembler passes information to the linker/loader to form the code such that the group declared segments or operands must lie within a 64Kbyte memory segment. Thus all such segments and labels can be addressed using the same segment base

```
PROGRAM GROUP CODE, DATA, STACK
```

The above statement directs the loader/linker to prepare an EXE file such that CODE, DATA and STACK segment must lie within a 64kbyte memory segment that is named as PROGRAM. Now, for the ASSUME statement, one can use the label PROGRAM rather than CODE, DATA and STACK as shown.

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

PROC(Procedure)

The PROC directive marks the start of a named procedure in the statement. Also, the types NEAR or FAR specify the type of the procedure, i.e whether it is to be called by the main program located within 64K of physical memory or not. For example, the statement RESULT PROC NEAR marks the start of a routine RESULT, which is to be called by a program located in the Same segment of memory. The FAR directive is used for the procedures to be called by the programs located in different segments of memory. The example statements are as follows:

```
RESULT PROC NEAR

ROUTINE PROC FAR
```

EXTRN(External and PUBLIC: Public)

The directive EXTRN informs the assembler that the names, procedures and labels declared after this directive have already been defined in some other assembly language modules. While in the other module, where the names, procedures and labels actually appear, they must be declared public, using the PUBLIC directive. If one wants to call a procedure FACTORIAL appearing in MODULE 1 from MODULE 2; in MODULE1, it must be declared PUBLIC using the statement PUBLIC FACTORIAL and in module 2, it must be declared external using the declaration EXTRN FACTORIAL. The statement of declaration EXTRN must be accompanied by the SEGMENT and ENDS directives of the MODULE 1, before it is called in MOBULE 2. Thus the MODULE 1 and MODULE 2 must have the following declarations.

```
MODULEl SEGMENT

PUBLIC FACTORIAL FAR MODULEl ENDS

MODULE2 SEGMENT

EXTRN FACTORIAL FAR

MODULE2 ENDS
```

ORG(Origin)

The ORG directive directs the assembler to start the memory allotment for the particular segment, block or code from the declared address in the ORG statement while starting the assembly process for a module, the assembler initializes a location counter to keep track of the allotted addresses for the module. If the ORG statement is not written in the program, the location counter is initialized to 0000. If an ORG 200H statement is present at the starting of the code segment of that module, then the code will start from 200H address in code segment) In other words, the location counter will get initialized to the address 0200H instead of 0000H. Thus, the code for different modules and segments can be located in the available memory as required by the programmer. The ORG directive can even be used with data segments similarly.

Label

he Label directive is used to assign a name to the current content of the location counter. At the start of the assembly process, the assembler initializes a location counter to keep track of memory locations assigned to the program. As the program assembly proceeds, the contents of the location counter are updated. During the assembly process, whenever the assembler comes across the LABEL directive, it assigns the declared label with the current contents of the location counter. The type of the label must be specified, i.e. whether it is a NEAR or a FAR label, BYTE or WORD label, etc. A LABEL directive may be used to make a FAR jump as shown below. A FAR jump cannot be made at a normal label with a colon. The label CONTINUE can be used for a FAR jump, if the program contains the following statement.

```
CONTINUE LABEL FAR
```

The LABEL directive can be used to refer to the data segment along with the data type, byte or word as shown.

```
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 DATALAST and its type will be byte and far.

OPERATORS

OFFSET(Offset of a Label)

When the assembler comes across the OFFSET operator along with a label, it first computes the 16-bit displacement (also called as offset interchangeably) of the particular label, and replaces the string 'OFFSET LABEL' by the computed displacement. This operator is used with arrays, strings, labels and procedures to decide their offsets in their default segments. The segment may also be decided by another operator of similar type, viz, SEG. Its most common use is in the case of the indirect, indexed, based indexed or other addressing techniques of similar types, used to refer to the memory indirectly. The examples of this operator are as follows:

Example:

```
CODE SEGMENT
MOV SI, OFFSET LIST
CODE ENDS
DATA SEGMENT
LIST DB 10H
DATA ENDS
```

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 or WORD. If the prefix is BYTE, then the particular label, variable or memory operand is treated as an 8-bit quantity, while if WORD is the prefix, then it is treated as a 16- bit quantity. In other words, the PTR operator is used to specify the data type -byte or word. The examples of the PTR operator are as follows:Example:

MOV AL, BYTE PTR [SI]	/Moves content of memory location addressed by SI (8-bit) to AL
INC BYTE PTR [BX]	/Increments byte contents of memory location addressed by BX

MOV BX, WORD PTR [2000H]	//Moves 16-bit content of memory location 2000H to BX, i.e. [2000H] to BL [2001 H] to BH
INC WORD PTR [3000H]	Increments word contents of memory location 3000H considering contents of 3000H (lower byte) and 3001 H (higher byte) as a 16-bit number

In case of JMP instructions, the PTR operator is used to specify the type of the jump, i.e. near or far, as explained in the examples given below.

JMP WORD PTR [BX] -NEAR Jump JMP WORD PTR [BX] -FAR Jump

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 bytes from the address of the byte next to the jump opcode). This method of specifying the jump address saves the memory. Otherwise, the assembler may reserve two bytes for the displacement. The syntax of the statement is as given below.

JMP SHORT LABEL

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 base address in place of 'SEG label'. The example given below explain the use of SEG operator.

MOV AX, SEG ARRAY; This statement moves the segment address

MOV DS, AX; of ARRAY in which it is appearing, to register AX and then to DS.

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. For the word type variable, the data type is 2, for double word type, it is 4, and for byte type, it is 1. Suppose, the STRING is a word array. The instruction

MOV AX, TYPE STRING moves the value 0002H in AX.

'+' & '—' 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.

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

LENGTH(Byte Length of a Label)

This directive is not available in MASM. This is used to refer to the length of a data array string.

```
MOV CX, LENGTH ARRAY
```

This statement, when assembled, will substitute the length of the array ARRAY in bytes, in the instruction.

LOCAL

The labels, variables, constants or procedures declared LOCAL in a module are to be used only by that module. At a later time, some other module may declare a particular data type LOCAL, which is previously declared LOCAL by another module or modules. Thus the samelabel may serve different purposes for different modules of a program. With a single declaration statement, a number of variables can be declared local, as shown.

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

NAME(Logical Name of a Module)

The NAME directive is used to assign a name to an assembly language program module. The module may now be referred to by its declared name. The names, if selected to be suggestive, may point out the functions of the different modules and hence may help in the documentation.

PUBLIC

The PUBLIC directive is used along with the EXTRN directive. This informs the assembler that the labels, variables, constants, or procedures declared PUBLIC may be accessed by other assembly modules to form their codes, but while using the PUBLIC declared labels, variables, constants or procedures the user must declare them externals using the EXTRN directive. On the other hand, the data types declared EXTRN in a module of the program, may be declared PUBLIC in at least anyone of the other modules of the same program.

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 bracket each logical segment of a program. In some cases, the segment may be assigned a type like PUBLIC (i.e. can be used by other modules of the program while linking) or GLOBAL (can be accessed by any other modules). The program structure given below explains the use of the SEGMENT directive.

EXE . CODE SEGMENT GLOBAL; Start of segment named EXE.CODE, that can be accessed by any other module.

EXE . CODE ENDS; END of EXE.CODE logical segment.

GLOBAL

The labels, variables, constants or procedures declared GLOBAL may be used by other modules of the program. Once a variable is declared GLOBAL, it can be used by any module in the program. The following statement declares the procedure ROUTINE as a global label.

ROUTINE PROC GLOBAL

Assembly Language programming

ASSEMBLY PROCESS

Microsoft assembler MASM is an easy to use and popular assembler. The MASM accepts the file names only with extension .ASM. Even if a filename without any extension is given as input, it provides an .ASM extension to it. The command for assembling the program is

C> MASM <filename.asm>

Eg:
C> MASM KMB.ASM
or
C> MASM KMB

After the cross reference file name is entered the assembly process starts. If the program contains syntax errors, they are displayed using error code number and the corresponding line number at which they appear. Once these syntax errors and warnings are taken care of by the programmer, the assembly process is completed successfully. The successful assembly process may generate the .OBJ,.LST and .CRF files, which may further be used by the linker program to link the object modules and generate an executable (.EXE) file from a .OBJ file.

LINKING AND RELOCATION

The DOS linking program LINK.EXE links the different object modules of a source program and function library routines to generate an integrated executable code of the source program. The main input to the linker is the .OBJ file that contains the object modules of the source programs. Other supporting information may be obtained from the files generated by the MASM. The linker program is invoked using the following options.

C> LINK

or

C>LINK <filename.OBJ>

The .OBJ extension is a must for a file to be accepted by the LINK as a valid object file. The first option may generate a display asking for the object file, list file and libraries as inputs and an expected name of the .EXE file to be generated. The output of the LINK program is an executable file with the entered filename and .EXE extension. This executable filename can further be entered at the DOS prompt to execute the file.

After linking, there has to be re-allocation of the sequences of placing the codes before actually placement of the codes in the memory. The loader program performs the task of reallocating the codes after finding the physical RAM addresses available at a given instant.

DEBUGGING

DEBUG.COM is a DOS utility that facilitates the debugging and trouble shooting of assembly language programs. The DEBUG utility enables you to have the control of resources. The DEBUG command at DOS prompt invokes this facility. A '_' (dash) display signals the successful invoke operation of DEBUG.

DEBUG offers a good platform for trouble shooting, executing and observing the results of the assembly language programs. There are different debug commands for executing debug statement.

1. **Write an assembly language program for addition of two numbers.**

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
OPR1 DW 1234H
OPR2 DW 0002H
RESULT DW O1 DUP(?)
DATA ENDS
CODE SEGMENT
START:  MOV AX, DATA
        MOV DS,AX
        MOV AX,OPR1
        MOV BX,OPR2
        CLC
        ADD AX,BX
        MOV DI,OFFSET RESULT
        MOV [DI],AX
        MOV AH,4CH
        INT 21H
```

```
CODE ENDS
        END START
```

2. Write a program to find out the number of even and odd numbers from a given series of 16 bit hexadecimal numbers.

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
LIST DW 2357H, 3456H, 1234H, 6754H
COUNT EQU O4H
DATA ENDS
CODE SEGMENT
START:  XOR BX,BX
        XOR DX,DX
        MOV AX, DATA
        MOV DS,AX
        MOV CL, COUNT
        MOV SI, OFFSET LIST
AGAIN:  MOV AX, [SI]
        ROR AX,01
        JC ODD
        INC BX
        JMP NEXT
ODD :    INC DX
NEXT:   ADD SI, 02
        DEC CL
        JNZ AGAIN
        MOV AH,4CH
        INT 21H
        CODE ENDS
        END START
```