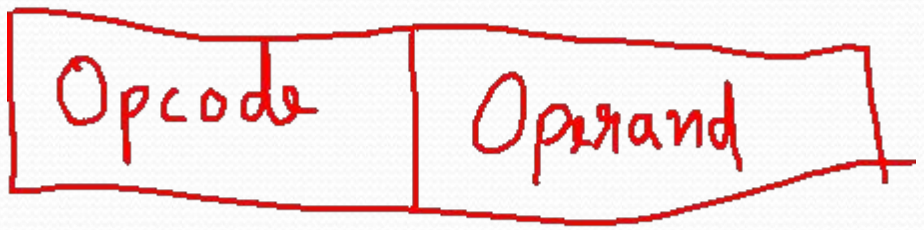
MODULE II

CONTENTS

Machine Language Instruction Format in 8086(Module-1)

- The instruction format of 8086 has one or more number of fields associated with it.
- The first field is called operation code field or opcode field, which indicates the type of operation.
- The instruction format also contains other fields known as operand fields.
- There are six general formats of instructions in 8086 instruction set. The length of an instruction may vary from one byte to six bytes.

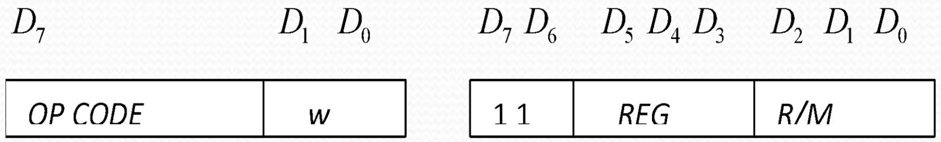
Instn



3

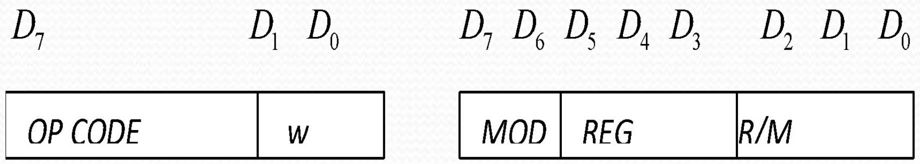
- **a) One byte Instruction**: This format is only one byte long and may have the implied data or register operands. The least significant **3 bits of the opcode** are used for specifying the **register operand**, if any. Otherwise, all the eight bits used to store opcode
- *CLC* : clear carry

- **b) Register to Register:** This format is 2 bytes long. The first byte of the code specifies the operation code and the width of the operand specifies by *w* bit. The second byte of the opcode shows the register operands and *R/M* field.



- For example: *MOV CL, AL*

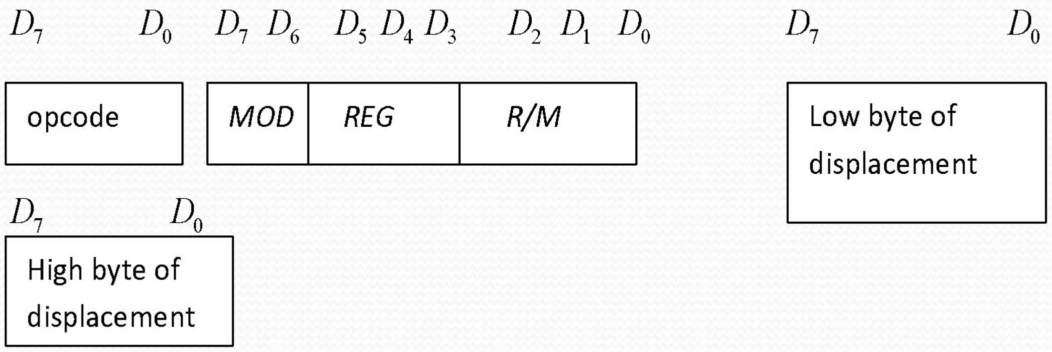
- **C) Register to/from memory with no displacement**:
- 2 bytes long and similar to the register to register format except for the *MOD* field.
- The *MOD* field shows the *MOD* of addressing.



MOV AX , *[BX]*

d) Register to/from Memory with Displacement:

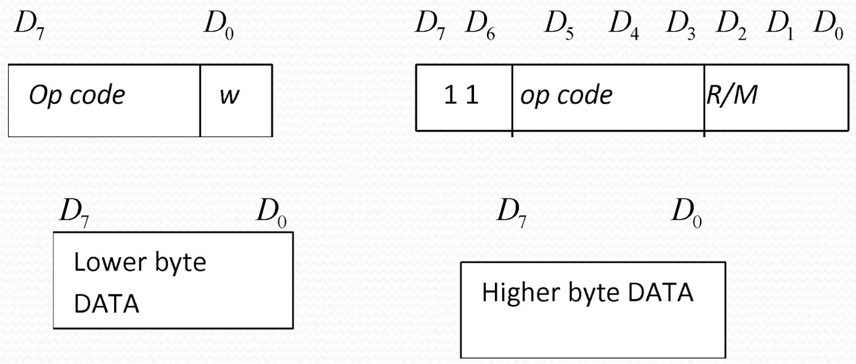
- This type of instruction format contains one or two additional bytes for displacement along with 2-byte the format of the register to/from memory without displacement.


D_7 D_0
opcode
D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0
MOD
REG
R/M
D_7 D_0
Low byte of displacement
D_7 D_0
High byte of displacement

Eg: MOV AX, 2000h[BX]

- *MOD* = 0 1 indicates displacement of 8 bits (instruction is of size 3 bytes)
- *MOD* = 1 0 indicates displacement of 16 bits. (instruction is of size 4 bytes)
- Already we have seen the other two options of *MOD*
- *MOD* = 1 1 indicates register to register transfer
- *MOD* = 0 0 indicates memory without displacement

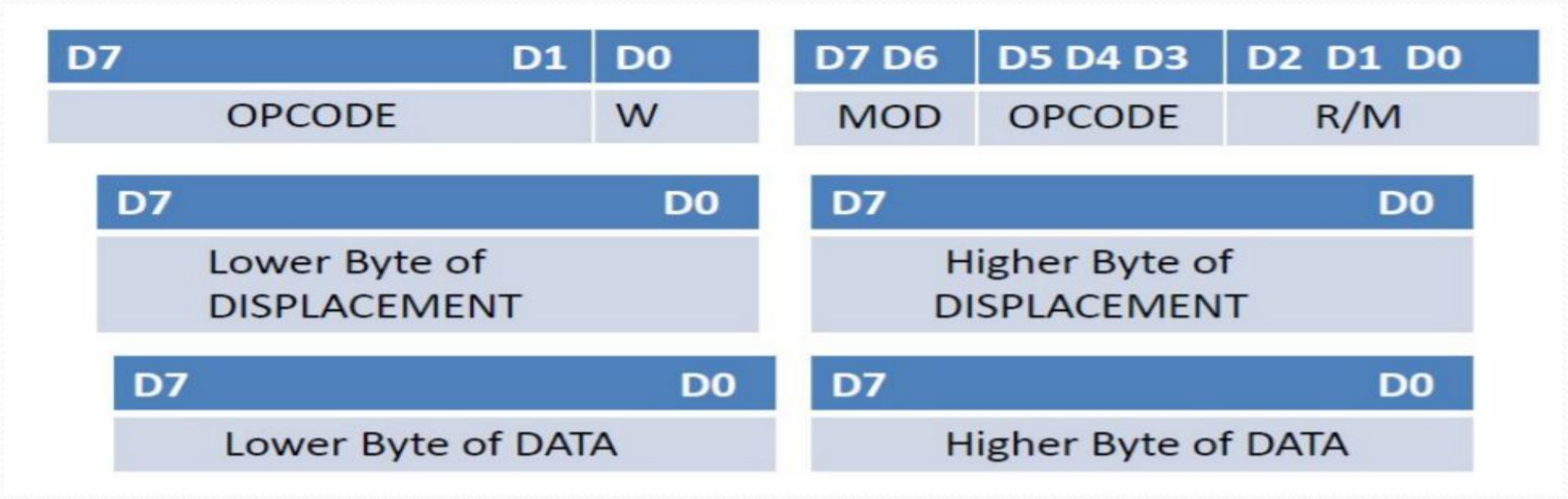
- **e) Immediate operand to register**
- In this format, the first byte as well as the 3 bits from the second byte which are used for *REG* field in case of register to register format are used for opcode. It also contains one or two bytes of immediate data.



When $w = 0$, the size of immediate data is 8 bits and the size of instruction is 3 bytes.

When $w = 1$, the size of immediate data is 16 bits and the size of instruction is 4 bytes.

- **f) immediate operand to memory with 16-bit displacement** : This requires 5 to 6 bytes for coding. The first two bytes contain the information regarding *OPCODE*, *MOD* and *R/M* fields. The remaining 4 bytes contain 2 bytes of displacement and 2 bytes of data.
- Mov 2000h[ax], 2340h



D7 D1	D0
OPCODE	W

D7 D6	D5 D4 D3	D2 D1 D0
MOD	OPCODE	R/M

D7 D0
Lower Byte of DISPLACEMENT

D7 D0
Higher Byte of DISPLACEMENT

D7 D0
Lower Byte of DATA

D7 D0
Higher Byte of DATA

Addressing Modes of 8086

- A way of locating data or operands.
- According to the flow of instruction execution, the instructions may be categorized as

1. Sequential control flow instructions and
2. Control transfer instructions.

- **Sequential control flow instructions** are the instructions which after execution, transfer control to the next instruction appearing immediately after in the program. Examples: Add,Sub,Mov
- **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.

Examples: INT, CALL, RET & JUMP

Addressing modes-sequential control flow

1. Immediate addressing mode:

- In this type of addressing, immediate data is a part of instruction, and appears in the form of successive byte or bytes.

Example: MOV AX, 0005H.

2. Direct addressing mode:

In the direct addressing mode, a 16-bit memory address(offset) directly specified in the instruction as a part of it.

Example: MOV AX, [5000H].

Effective address,EA = 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.

Example: MOV BX, AX

4. Register indirect addressing mode:

- In this addressing mode, the offset address of data is in either BX or SI or DI Register. The default segment is either DS or ES.
- **Example: MOV AX, [BX].**

EA=10H * DS +[BX]

- **5. Indexed addressing mode:**

In this addressing mode, offset of the operand is stored one of the index registers. DS & ES are the default segments for index registers SI & DI respectively.

Example: MOV AX, [SI]

Here, data is available at an offset address stored in SI in DS.

EA=10H * DS +[SI]

6. Register relative addressing mode:

- Here 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 register BX, BP, SI & DI in the default (either in DS & ES) segment.
- **Example: MOV AX, 50H [BX]**
- **EA=10H * DS +50H+[BX]**

7. Based indexed addressing mode:

- The effective address of data is formed in this addressing mode, 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.

Example: MOV AX, [BX][SI]

EA=10H * DS + [BX] + [SI]

8. Relative based indexed:

- The effective address is formed by adding an 8 or 16-bit displacement with the sum of contents of any of the base registers (BX or BP) and any one of the index registers, in a default segment.
- **Example: MOV AX, 50H [BX] [SI]**

EA=10H * DS +50H+[BX]+[SI]

- For the control transfer instructions, the addressing modes depend upon whether the destination location is within the same segment or in a different one
- Basically, there are two addressing modes for the control transfer instructions, viz. Inter segment and intra segment addressing modes.
- 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 mode

- If the destination location lies in the same segment, the m it is called intra segment
- **Addressing Modes for control transfer instructions:**

1. Intersegment
   - Intersegment direct
   - Intersegment indirect
2. Intrasegment
   - Intrasegment direct
   - Intrasegment indirect

- **1. 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.

- **Example: JMP 5000H, 2000H**

jump to effective address 2000H in segment 5000H.

- **2. 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.

The starting address of the memory block may be referred using any of the addressing modes, except immediate mode.

- **Example:** JMP [2000H].
- Jump to an address in the other segment specified at effective address 2000H in DS

3)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 transfers instruction lies and appears directly in the instruction as an immediate displacement value.
- JMP SHORT LABEL.

4) 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 directly. Here, the branch address is found as the content of a register or a memory location.

Example: JMP [BX]

The content of different registers are given below. Form effective addresses for different addressing modes

- Offset(displacement)=5000H [AX]-1000H, [BX]-2000H,[SI]-3000H,[DI]-4000H [BP]-5000H,[SP]-6000H,[CS]-0000H,[DS]-1000H [SS]-2000H,[IP]-7000H,[CS]-0000H,[DS]-1000H
- i) Direct AM MOV AX,[5000H]
- ii) Register Indirect MOV AX,[BX]
- iii) Register Relative MOV AX,5000[BX]
- iv) Based Indexed MOV AX,[BX] [SI]
- V) Relative Based Indexed MOV AX,5000[BX] [SI]

- i) 15000H
- ii) 12000H
- iii) 17000H
- iv)15000H
- v)1A000H

Instruction Set of 8086

Classification of Instruction Set

- Data Copy /Transfer instructions
- Arithmetic& Logical instructions
- Branch instructions
- Loop instructions
- Machine Control Instructions
- Shift / rotate instructions
- Flag manipulation instructions
- String instructions

Data Transfer Instructions

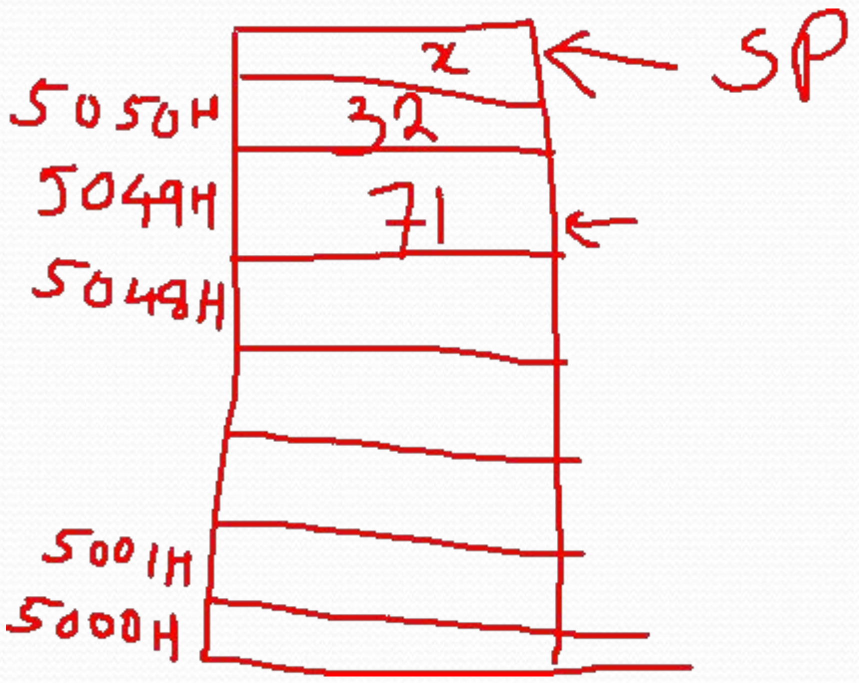
- These instructions are used to transfer data from source to destination.
- The operand can be a constant, memory location, register or I/O port address.

Data Transfer Instructions

- **MOV Des, Src:**
  - Src operand can be register, memory location or immediate operand.
  - Des can be register or memory operand.
  - Both Src and Des cannot be memory location at the same time.
  - Direct loading of Segment reg with immediate data is not permitted
  - E.g.:
    - MOV CX, 037A H
    - MOV AL, BL
    - MOV BX, [0301 H]

Data Transfer Instructions

- **PUSH Operand:**
  - The PUSH instruction decrements the stack pointer by two and copies the word from source to the location where stack pointer now points. E.g.: PUSH BX
  - Source can be a general purpose register, segment register or a memory location.
- The PUSH instruction first pushes the most significant byte to sp-1, then the least significant to the sp-2.



PUSH BX

[BX] = 3271H

POP BX

POP Des:

- The POP instruction copies a word from the stack location pointed by the stack pointer to the destination. The destination can be a General purpose register, a segment register or a memory location.
- Here after the content is copied the stack pointer is automatically incremented by two.

Data Transfer Instructions

- **XCHG Des, Src:**
  - This instruction exchanges Src with Des.
  - It cannot exchange two memory locations directly.
  - E.g.: XCHG DX, AX

Data Transfer Instructions

- **IN Accumulator, Port Address:**
  - It transfers the operand from specified port to accumulator register.
  - E.g.: IN AX, 0028 H
  - IN AX, DX
    - MOV DX, O8OOH
    - IN AX, DX
- **OUT Port Address, Accumulator:**
  - It transfers the operand from accumulator to specified port.
  - E.g.: OUT 0028H, AX MOV DX, O8OOH
  - OUT DX, AX OUT AX, DX
  -

Data Transfer Instructions

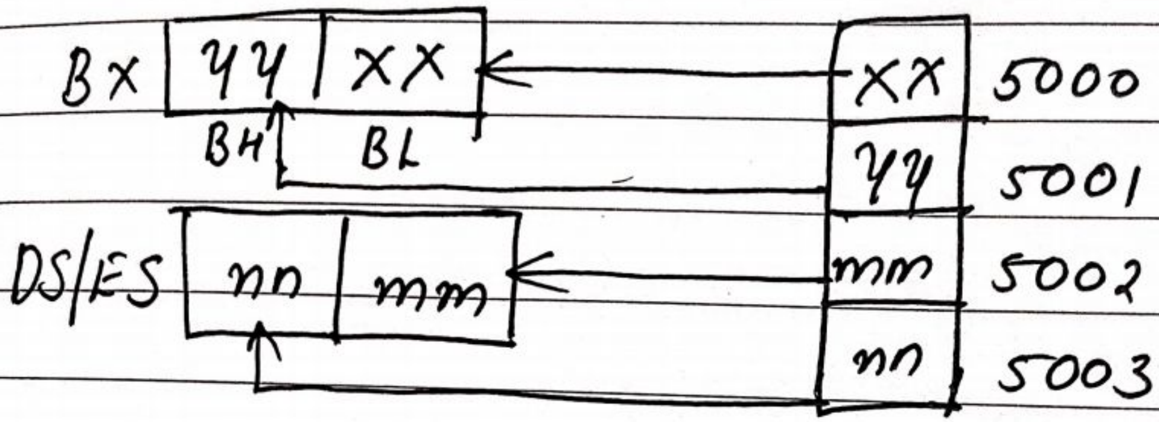
- **LEA Register, Src:**
  - It loads a 16-bit register with the offset address of the data specified by the Src.
  - E.g. LEA BX, ADR
  - LEA SI, ADR[BX]

Data Transfer Instructions

- **LDS/LES Des, Src:**
  - It loads 32-bit pointer from memory source to destination register and DS.
  - The offset is placed in the destination register and the segment is placed in DS.
  - To use this instruction the word at the lower memory address must contain the offset and the word at the higher address must contain the segment.
  - E.g.: LDS BX, 5000H

LDS BX, 5000H

LES BX, 5000H.



LDS/LES Instruction Execution

Data Transfer Instructions

- **LES Des, Src:**
  - It loads 32-bit pointer from memory source to destination register and ES.
  - The offset is placed in the destination register and the segment is placed in ES.
  - This instruction is very similar to LDS except that it initializes ES instead of DS.
  - E.g.: LES BX, 5000H

Data Transfer Instructions

- **LAHF:**
  - It copies the lower byte of flag register to AH.
- **SAHF:**
  - It copies the contents of AH to lower byte of flag register.
- **PUSHF:**
  - Pushes flag register to top of stack.
- **POPF:**
  - Pops the stack top to flag register.

Arithmetic Instructions

- ADD,ADC,SUB,SBB, INC,DEC
- MUL ,DIV
- IMUL,IDIV
- CBW
- CWD
- NEG
- CMP
- AAA,DAA

Arithmetic Instructions

- **ADD Des, Src:**
  - It adds a byte to byte or a word to word.
  - It effects AF, CF, OF, PF, SF, ZF flags.
  - E.g.:
    - ADD AL, 74H   AL $\leftarrow$ AL + 74H
    - ADD DX, AX
    - ADD AX, [BX]   FF +FF = 1FE

Arithmetic Instructions

- **ADC Des, Src:**
  - It adds the two operands with CF.
  - It effects AF, CF, OF, PF, SF, ZF flags.
  - E.g.:ADC AL, 74H AL AL + 74H+CF
    - ADC DX, AX
    - ADC AX, [BX] ←

12 FF H +

00 01 H

=13 00 H

Arithmetic Instructions

- **SUB Des, Src:**
  - It subtracts a byte from byte or a word from word.
  - It effects AF, CF, OF, PF, SF, ZF flags.
  - For subtraction, CF acts as borrow flag.
  - E.g.:
    - SUB AL, 74H
    - SUB DX, AX
    - SUB AX, [BX]

Arithmetic Instructions

- **SBB Des, Src:**
  - It subtracts the two operands and also the borrow from the result.
  - It effects AF, CF, OF, PF, SF, ZF flags.
  - E.g.:
    - SBB AL, 74H
    - SBB DX, AX
    - SBB AX, [BX]

Arithmetic Instructions

- **INC Src:**
  - It increments the byte or word by one.
  - The operand can be a register or memory location.
  - It effects AF, OF, PF, SF, ZF flags.
  - CF is not effected.
  - E.g.: INC AX

Arithmetic Instructions

- **DEC Src:**
  - It decrements the byte or word by one.
  - The operand can be a register or memory location.
  - It effects AF, OF, PF, SF, ZF flags.
  - CF is not effected.
  - E.g.: DEC AX

Arithmetic Instructions

- **DAA (Decimal Adjust after Addition)**
  - It is used to make sure that the result of adding two BCD numbers is adjusted to be a correct BCD number.
  - It only works on AL register.
- **DAS (Decimal Adjust after Subtraction)**
  - It is used to make sure that the result of subtracting two BCD numbers is adjusted to be a correct BCD number.
  - It only works on AL register.

Arithmetic Instructions

- **AAA (ASCII Adjust after Addition):**
  - Executed after an ADD instn that adds two ascii coded operands to give a byte of result in AL
  - AAA instruction converts the resulting content of AL in to unpacked decimal digit
  - In ASCII, 0 – 9 are represented by 30H – 39H.
  - This instruction allows us to add the ASCII codes.
- **Other ASCII Instructions:**
  - **AAS** (ASCII Adjust after Subtraction)
  - **AAM** (ASCII Adjust after Multiplication)
  - **AAD** (ASCII Adjust Before Division)

Arithmetic Instructions

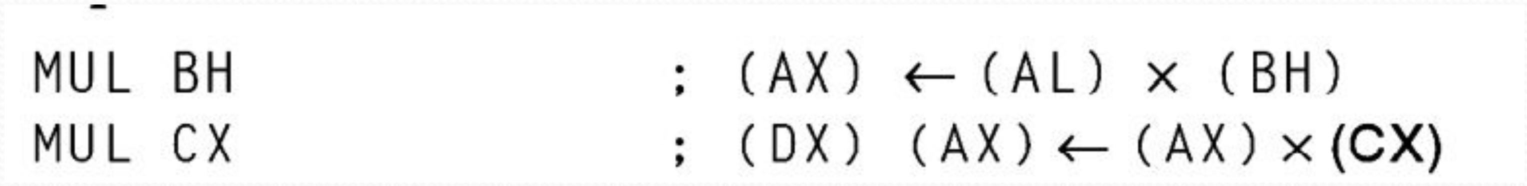
- **NEG Src:**
  - It creates 2's complement of a given number.
  - That means, it changes the sign of a number.

Arithmetic Instructions

- **CMP Des, Src:**
  - It compares two specified bytes or words.
  - The Src and Des can be a constant, register or memory location.
  - Both operands cannot be a memory location at the same time.
  - The comparison is done simply by internally subtracting the source from destination.
  - The value of source and destination does not change, but the flags are modified to indicate the result.

Arithmetic Instructions

- **MUL Src:**
  - It is an unsigned multiplication instruction.
  - It multiplies two bytes to produce a word or two words to produce a double word.



```
MUL  BH                    ;  (AX) ← (AL) × (BH)
MUL  CX                    ;  (DX) (AX) ← (AX) × (CX)
```

  - This instruction assumes one of the operand in AL or AX.
  - Src can be a register or memory location.
- **IMUL Src:**
  - It is a signed multiplication instruction.

Arithmetic Instructions

- **DIV Src:**
  - It is an unsigned division instruction.
  - It divides word by byte or double word by word.
  - The operand is stored in AX, divisor is Src and the result is stored as:
  - DIV BH (AX /BH)
    - AH = remainder AL = quotient
    - DIV BX (DX AX/ BX)
    - AX= Quotient DX= Remainder
- **IDIV Src:**
  - It is a signed division instruction.

Arithmetic Instructions

- **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.
- **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.

Bit Manipulation Instructions

- These instructions are used at the bit level.
- These instructions can be used for:
  - Testing a zero bit
  - Set or reset a bit
  - Shift bits across registers

Bit Manipulation Instructions

- **NOT Src:**
  - It complements each bit of Src to produce 1's complement of the specified operand.
  - The operand can be a register or memory location.

Bit Manipulation Instructions

- **AND Des, Src:**
  - It performs AND operation of Des and Src.
  - 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: AND BL,0FH (for higher nibble zeros)
  - MOV AL, 08H MOV AL, 05H

  AND AL, 01H MOV AL, 01H

Bit Manipulation Instructions

- **OR Des, Src:**
  - It performs OR operation of Des and Src.
  - 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.
  - OR BL,FOH (for higher nibble ones)

Bit Manipulation Instructions

- **XOR Des, Src:**
  - It performs XOR operation of Des and Src.
  - 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.
  - XOR AX,AX // AX=0

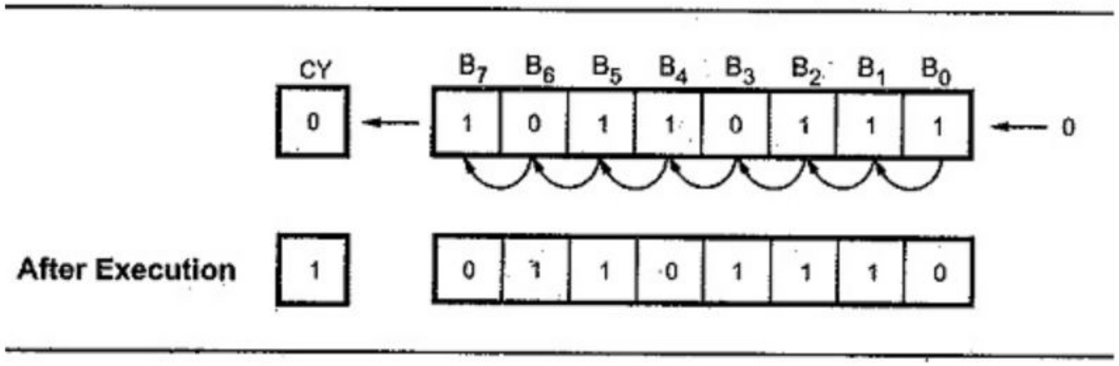
- **TEST: Logical Compare** – Used to AND operands to update flags, without affecting operands.

```
MOV AL, 07H
TEST AL, 01H    // AL= 7H
JZ  EVEN        // affects the flag
```

Shift Instructions

- **SHL/SAL Des, Count:**
  - It shift bits of byte or word left, by count.
  - It puts zero(s) in LSBs.
  - MSB is shifted into carry flag.
  - If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in Count.
  - However, if the number of bits to be shifted is more than 1, then the count is put in CL register.

Diagram shows SAL instruction for byte operation.



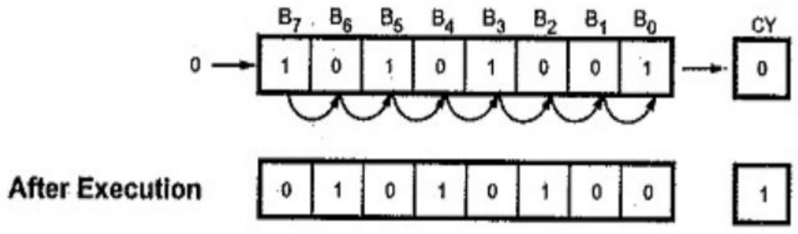
Flags : All flags are affected.

Examples :

```
SAL  CX,  1            ; Shift word in CX 1 bit position
                       ; left, 0 in LSB
MOV  CL,  05H          ; Load desired number of shifts in CL
SAL  AX,  CL           ; Shift word in AX left 5 times
                       ; 0s in 5 least-significant bits.
```

Shift Instructions

- **SHR Des, Count:**
  - It shift bits of byte or word right, by count.
  - It puts zero(s) in MSBs.
  - LSB is shifted into carry flag.
  - If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in Count.
  - However, if the number of bits to be shifted is more than 1, then the count is put in CL register.



Flags : All flag's are affected.

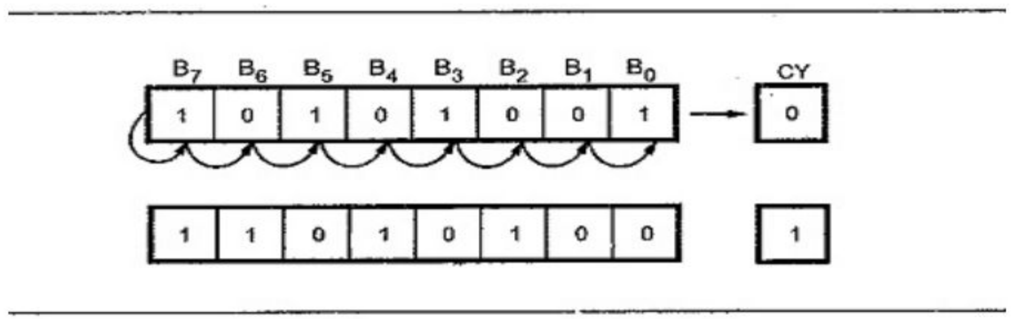
Examples :

```
SHR  CX,  1            ; Shift word in CX 1 bit position
                       ; right, 0 in MSB.
MOV  CL,  05H          ; Load desired number of shifts in CL.
SHR  AX,  CL           ; Shift word in AX right 5 times
                       ; 0's in 5 most significant bits.
```

Shift Instructions

- **SAR** – Used to shift bits of a byte/word towards the right and copy the old MSB into the new MSB.

Diagram shows SAR instruction for byte operation.

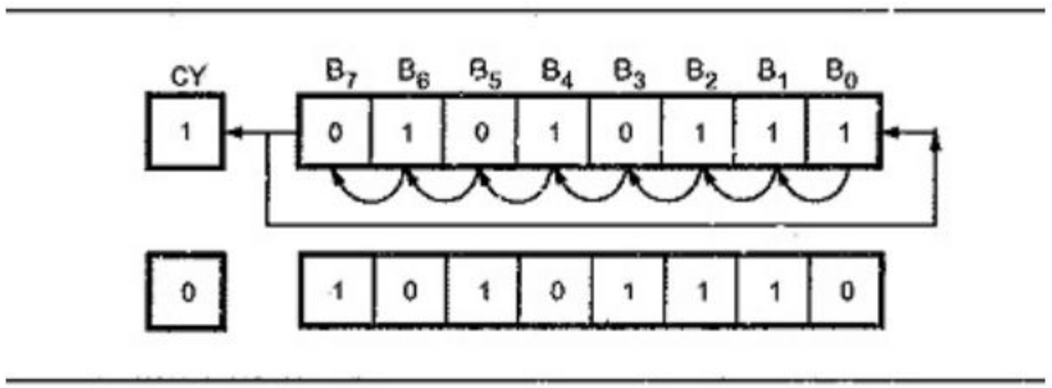


Flags : All flags are affected.

Roatate Instructions

- **ROL Des, Count:**
  - It rotates bits of byte or word left, by count.
  - MSB is transferred to LSB and also to CF.
  - If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in Count.
  - However, if the number of bits to be shifted is more than 1, then the count is put in CL register.

Diagram shows ROL instruction for byte rotation.



Examples :

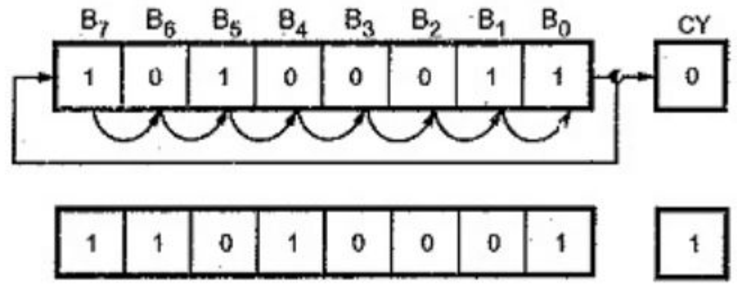
```
ROL  CX,  1         ; Word in CX one bit position left, MSB to
                    ; LSB and CF
MOV  CL,  03H       ; Load desired number of bits to rotate in CL.
ROL  BL,  CL        ; Rotate BL three positions.
```

Rotate Instructions

- **ROR Des, Count:**
  - It rotates bits of byte or word right, by count.
  - LSB is transferred to MSB and also to CF.
  - If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in Count.
  - However, if the number of bits to be shifted is more than 1, then the count is put in CL register.

Rotate Instructions

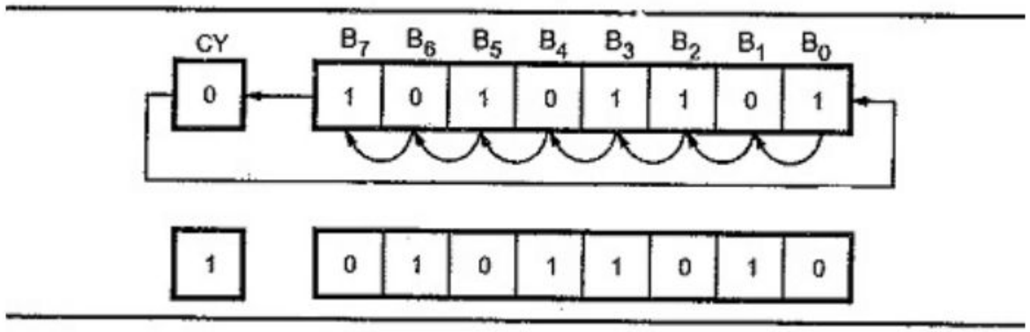
- **ROL** – Used to rotate bits of byte/word towards the left, i.e. MSB to LSB and to Carry Flag [CF].
- **RCR** – Used to rotate bits of byte/word towards the right, i.e. LSB to CF and CF to MSB.
- **RCL** – Used to rotate bits of byte/word towards the left, i.e. MSB to CF and CF to LSB.



Examples :

```
ROR  CX,  1              ; Rotated word in CX one bit position
                         ; left, LSB to MSB and CF.
MOV  CL,  03H            ; Load number of bits to rotate in CL.
ROR  BL,  CL             ; Rotate BL three positions.
```

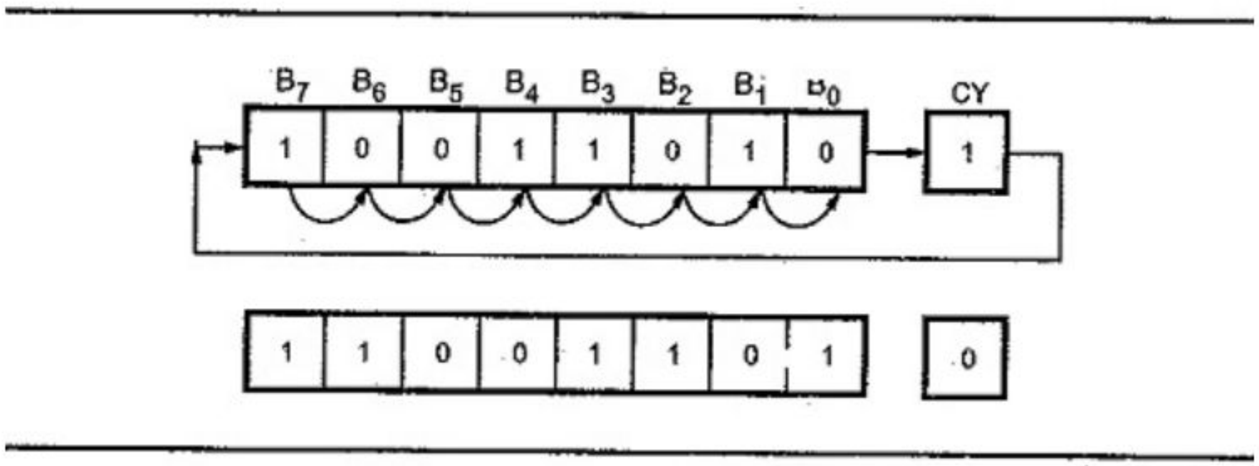
Diagram shows RCL instruction for byte rotation.



Examples :

```
RCL  CX,  1          ;  Rotated word in CX 1 bit left, MSB to
                     ;  CF, CF to LSB.
MOV  CL,  04H        ;  Load number of bit positions to rotate
                     ;  in CL.
RCL  AL,  CL         ;  Rotate AL 4 bits left.
```

Diagram shows RCR instruction for byte rotation.



Control Transfer or Branching Instructions

- These instructions cause change in the sequence of the execution of instruction.
- This change can be through a condition or sometimes unconditional.
- The conditions are represented by flags.

Program Execution Transfer Instructions

- **CALL Des:**
  - This instruction is used to call a subroutine or function or procedure.
  - The address of next instruction after CALL is saved onto stack.
- **RET:**
  - It returns the control from procedure to calling program.
  - Every CALL instruction should have a RET.

Program Execution Transfer Instructions

- **JMP Des:**
  - This instruction is used for unconditional jump from one place to another.
- **Jxx Des (Conditional Jump):**
  - All the conditional jumps follow some conditional statements or any instruction that affects the flag.

Conditional Jump Instructions

- **JA/JNBE** – Used to jump if above/not below/equal instruction satisfies.
- **JAE/JNB** – Used to jump if above/not below instruction satisfies.
- **JBE/JNA** – Used to jump if below/equal/ not above instruction satisfies.
- **JC** – Used to jump if carry flag CF = 1
- **JE/JZ** – Used to jump if equal/zero flag ZF = 1
- **JG/JNLE** – Used to jump if greater/not less than/equal instruction satisfies.
- .

Conditional Jump Instructions

- **JGE/JNL** – Used to jump if greater than/equal/not less than instruction satisfies.
- **JL/JNGE** – Used to jump if less than/not greater than/equal instruction satisfies.
- **JLE/JNG** – Used to jump if less than/equal/if not greater than instruction satisfies.
- **JNC** – Used to jump if no carry flag (CF = 0)
- **JNE/JNZ** – Used to jump if not equal/zero flag ZF = 0
- **JNO** – Used to jump if no overflow flag OF = 0

Conditional Jump Instructions

- **JNP/JPO** – Used to jump if not parity/parity odd, PF = 0
- **JNS** – Used to jump if not sign SF = 0
- **JO** – Used to jump if overflow flag OF = 1
- **JP/JPE** – Used to jump if parity/parity even, PF = 1
- **JS** – Used to jump if sign flag SF = 1

Iteration Control Instructions

- These instructions are used to execute the given instructions for number of times.
- **LOOP** – Used to loop a group of instructions until CX = 0
- **LOOPE/LOOPZ** – Used to loop a group of instructions till it satisfies ZF = 1 & CX = 0
- **LOOPNE/LOOPNZ** – Used to loop a group of instructions till it satisfies ZF = 0 & CX = 0
- **JCXZ** – Used to jump to the provided address if CX = 0

(Conditional jmp instn doesn't check status flag for condition testing)

String Instructions

- String in assembly language is just a sequentially stored bytes or words.
- There are very strong set of string instructions in 8086.
- By using these string instructions, the size of the program is considerably reduced.

String Instructions

- **CMPS Des, Src:**
  - It compares the string bytes or words.

- **SCAS String:**
  - It scans a string.
  - It compares the String with byte in AL or with word in AX.

CMPS

- MOV AX,SEG1
- MOV DS,AX
- MOV AX,SEG2
- MOV ES,AX
- MOV SI,OFFSET STRING1
- MOV DI,OFFSET STRING2
- MOV CX,010H
- CLD
- REPE CMPSW

SCAS

- MOV AX,SEG
- MOV ES,AX
- MOV DI,OFFSET
- MOV CX,010H
- MOV AX,WORD
- CLD
- REPNE SCASW

String Instructions

- **MOVS / MOVSB / MOVSW:**
  - It causes moving of byte or word from one string to another.
  - In this instruction, the source string is in Data Segment and destination string is in Extra Segment.
  - SI and DI store the offset values for source and destination index.

```
MOV AX,5000H
 MOV DS,AX
 MOV AX,6000H
 MOV ES,AX
MOV CX,OFFH
 MOV SI, 1000H
MOV DI,2000H
 CLD
 REP MOVSB
```

String Instructions

- **REP (Repeat):**
  - This is an instruction prefix.
  - It causes the repetition of the instruction until CX becomes zero.
  - E.g.: REP MOVSB STR1, STR2
    - It copies byte by byte contents.
    - REP repeats the operation MOVSB until CX becomes zero.

String Instructions

- **LODS/LODSB/LODSW** – Used to store the string byte pointed by DS:SI pair into AL or string word into AX.
- **STOS:** Used to stores AL/AX register contents to a location in the string pointed by ES:DI register pair.

Flag Manipulation Instructions

- Flag manipulation instructions directly modify some of the flags of 8086.

Flag Manipulation Instructions

- **STC:**
  - It sets the carry flag to 1.
- **CLC:**
  - It clears the carry flag to 0.
- **CMC:**
  - It complements the carry flag.

Flag Manipulation Instructions

- **STD:**
  - It sets the direction flag to 1.
  - If it is set, string bytes are accessed from higher memory address to lower memory address.
- **CLD:**
  - It clears the direction flag to 0.
  - If it is reset, the string bytes are accessed from lower memory address to higher memory address.

Processor Control Instructions

- WAIT – Wait for Test input 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

Interrupt Instructions

- **INT** – Used to interrupt the program during execution and calling service specified.
- **INTO** – Used to interrupt the program during execution if OF = 1
- **IRET** – Used to return from interrupt service to the main program

Assembler Directives

- **Assembler directives** are instructions that direct the **assembler** to do something.
- These are the hints given to assembler using some predefined alphabetical strings.
- Operator is the hint which helps the assembler to assign a particular constant with a label or initialize particular memory locations or labels with constants

Assembler Directives

- ASSUME
- DB: Defined Byte.
- DD:Defined Double Word
- DQ:Defined Quad Word
- DT:Define Ten Bytes
- DW:Define Word

Assembler Directives

- **Assume**: the names of the logical segments to be assumed for different segments used in the program.
- ASUME CS:CODE ; This tells the assembler that the logical segment named CODE contains the instruction statements for the program and should be treated as a code segment.
- ASUME DS:DATA ;This tells the assembler that for any instruction which refers to a data in the data segment, data will found in the logical segment DATA

- **DB** : DB directive is used to declare a byte-type variable or to store a byte in memory location.

Example:

- 1)PRICE DB 49h, 98h, 29h

Declare an array of 3 bytes, named as PRICE and initialize.

- 2) NAME DB 'ABCDEF'

Declare an array of 6 bytes and initialize with ASCII code for letters

3) TEMP DB 100 DUP(?) Set 100 bytes of storage in memory and give it the name as TEMP, but leave the 100 bytes uninitialized. Program instructions will load values into these locations

- END-End Program
- ENDP End Procedure
- ENDS End Segment
- EQU Equate
- EVEN Align on Even Memory Address
- EXTRN

- **END** directive is placed after the last statement of a program . The assembler will ignore any statement after an END directive.
- **ENDP** directive is used along with the name of the procedure to indicate the end of a procedure Example:
- SQUARE_NUM PROC

; It start the procedure

SQUARE_NUM ENDP

- ;Hear it is the End for the procedure

- **ENDS** directive is used with name of the segment to indicate the end of that logic segment.

Example:

CODE SEGMENT

;Hear it Start the logic segment containing code

; Some instructions statements to perform the logical operation

- CODE ENDS
- ;End of segment named as CODE

- **EQU - This EQU directive is used to give a name** to some value or to a symbol. Each time the assembler finds the name in the program, it will replace the name with the value or symbol you given to that name.
- **Example:**
- **FACTOR EQU 03H**

- **EVEN - This EVEN directive instructs the** assembler to increment the location of the counter to the next even address if it is not already in the even address. If the word is at even address 8086 can read a memory in 1 bus cycle.

- **GROUP :** Group Related Segments
- **LABEL:** Assign name to the current content of the location counter
- **NAME:** Assign a name to an assembly language program module
- **OFFSET :** assembler computes the offset of the label
- **ORG :originate**

ORG 2000h

- PROC - Procedure
- PTR - Pointer
- PUBLC
- SEGMENT
- SHORT
- TYPE

- **PROC - The PROC directive is used to identify the** start of a procedure. The term near or far is used to specify the type of the procedure.
- Example: **SMART PROC FAR ; This identifies that** the start of a procedure named as SMART and instructs the assembler that the procedure is far .
- **SMART ENDP**
- This PROC is used with ENDP to indicate the break of
- the procedure.

- **PTR - This PTR operator is used to assign a** specific type to a variable or to a label.

Eg: MOV AL,BYTE PTR [SI]

```
    INC BYTE PTR [BX]
    INC WORD PTR [BX]
```

- **TYPE - TYPE operator instructs the assembler to** determine the type of a variable and determines the number of bytes specified to that variable.
- **Example:**
- Byte type variable – assembler will give a value 1
- Word type variable – assembler will give a value 2
- Double word type variable – assembler will give a value 4

Eg:MOV BX, TYPE WORD_ ARRAY ; moves the value 2 to BX

- **PUBLIC - is used to instruct** the assembler that a specified name or label will be accessed from other modules.
- Example:

PUBLIC DIVISOR, DIVIDEND ;these two variables are public so these are available to all modules.

- If an instruction in a module refers to a variable in another assembly module, we can access that module by declaring as **EXTRN directive.**

DOS Function Calls

INT 21H : Call DOS Function

- **AH 00H : Terminate a Program**
- **AH 01H : Read the Keyboard**
- **AH 02H : Write to a Standard Output Device**
- **AH 08H : Read a Standard Input without Echo**
- **AH 09H : Display a Character String**
- **AH 0AH : Buffered keyboard Input**
- **AH 4CH :Return to DOS prompt**

Thank You