
Module II

8086 Addressing Modes, 8086 Instruction set and Assembler Directives - Assembly Language Programming with Subroutines, Macros, Passing Parameters, Use of stack.

Instruction Format in 8086

The instruction format of 8086 has one or more number of fields associated with it.

The first filled is called operation code field or opcode field, which indicates the type of operation.

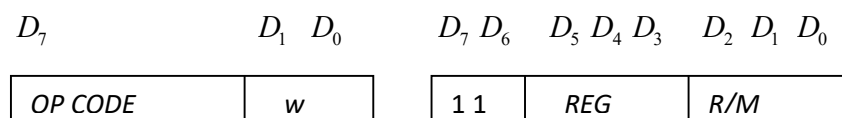
The instruction format also contains other fields known as operand fields.

There are six general formats of instructions in 8086 instruction set. The length of an instruction may vary from one byte to six bytes.

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.



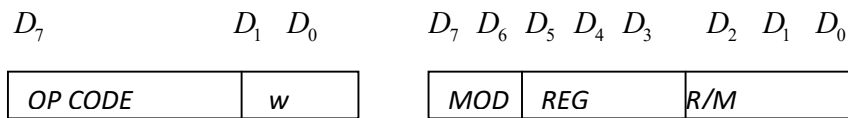
The register represented by the *REG* field is one of the operands. The *R/M* field specifies another register or memory location, ie., the other operand. The register specified by *REG* is a source operand if *D* = 0, else it is a destination operand.

For example:

MOV: data transfer operation from Register to Register.

MOV CL, AL, i.e. $CL \leftarrow AL$

C) Register to/from memory with no displacement: This format is also 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: Data transfer Register/memory to/from register.

This format is similar to register to register transfer. The difference is in mod field.

For register to register, mod = 11

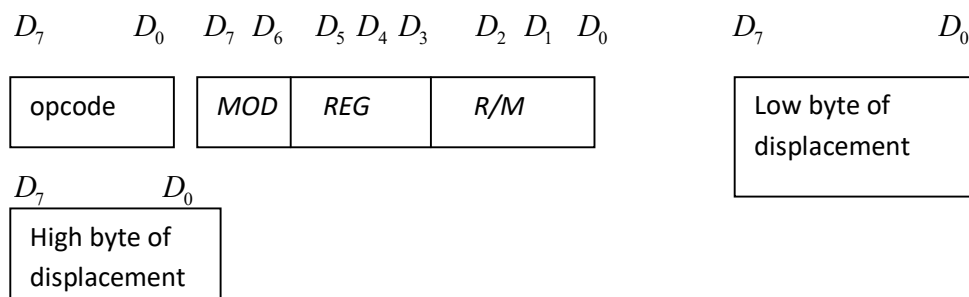
For register to/from memory with no displacement, *mod* = 00.

When *mod* = 00, the r/m fields indicates the address to memory location.

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.



MOD = 01 indicates displacement of 8 bytes (instruction is of size 3 bytes)

MOD = 10 indicates displacement of 16 bytes. (instruction is of size 4 bytes)

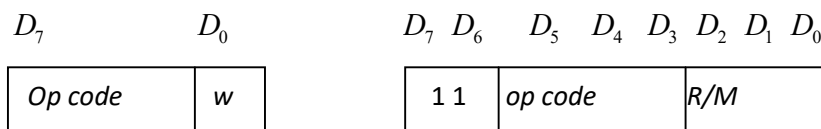
Already we have seen the other two options of *MOD*

MOD = 11 indicates register to register transfer

MOD = 00 indicates memory without displacement

e) Immediate operand to register

In this format, the first byte as well as the 3 bites 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.

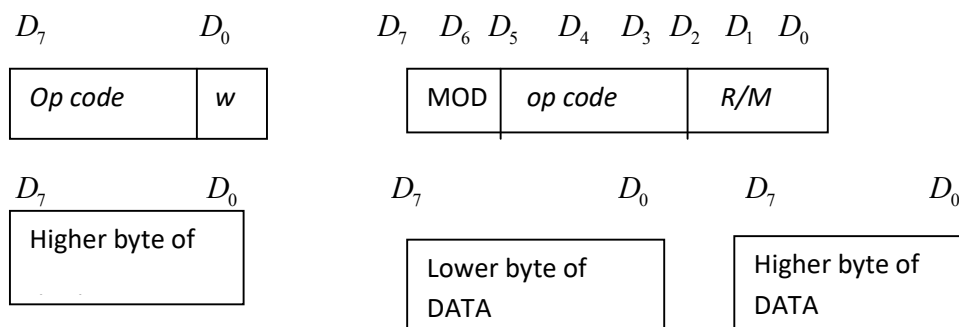




When $w = 0$, the size of immediate data is 8 bits and the size of instruction is 3 bytes.

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 type of instruction format 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.



Microprocessor - 8086 Instruction Sets

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

The 8086 microprocessor supports 8 types of instructions –

- Data Transfer Instructions
- Arithmetic Instructions
- Logical Instructions
- String Instructions
- Program Execution Transfer Instructions (Branch & Loop Instructions)
- Processor Control Instructions
- Iteration Control Instructions
- Interrupt Instructions

Data Transfer Instructions

These instructions are used to transfer the data from the source operand to the destination operand. Following are the list of instructions under this group –

Instruction to transfer a word

- **MOV** – Used to copy the byte or word from the provided source to the provided destination.
Eg:- MOV AX,BX (Move content of BX reg into AX reg)
MOV CX, 5000H (Move 00 into CL reg and 50 into CH reg)
MOV BX,[5000]- (Move content of memory location 5000 into BX reg)
- **PUSH**:-Push content of register/ memory to stack
Eg:- PUSH AX
Here first decrement the stack pointer by one and push the content of AH into stack and again decrement the stack pointer by one, then push AL into stack.
- **POP** – Used to get a word from the top of the stack to the provided location.
Eg:- POP AX
Here first pop the content of stack into AL and increment the stack pointer by one, then pop content of stack into AH and increment SP by one.
- **XCHG** – Used to exchange the data from two locations.
Eg:- XCHG [5000 H]- exchange data between AX and [5000H] memory location
XCHG AX,BX - exchange data between AX and BX
- **PUSHA** – Used to put all the registers into the stack.
- **POPA** – Used to get words from the stack to all registers.
- **XLAT** – Used to translate a byte in AL using a table in the memory.

Instructions for input and output port transfer

- **IN** – Used to read a byte or word from the provided port to the accumulator.
IN AX, 0028 H
Read content of port from 0028 port into AX reg
- **OUT** – Used to send out a byte or word from the accumulator to the provided port.
OUT 0028 H, AX
Write content of 0028 port into AX reg

Instructions to transfer the address

- **LEA** – Used to load the address of operand into the provided register.

eg:- LEA AX, 4000H ---Load the address 4000 into AX register

LEA AX,ADR- load address of ADR into AX reg

LEA BX, [DI]

Load content of DI into BX reg

- **LDS** – Used to load DS register and other provided register from the memory

LDS BX, 5000H

Load the content of 5000 &5001 into BX reg and 5002 &5003 into DS reg

- **LES** – Used to load ES register and other provided register from the memory.

LES BX,5000H

Load the content of 5000 &5001 into BX reg and 5002 &5003 into ES reg

Instructions to transfer flag registers

- **LAHF** – Used to load AH with the lower byte of the flag register.
- **SAHF** – Used to store AH register to lower byte of the flag register.
- **PUSHF** – Used to copy the flag register at the top of the stack.

First decrement SP by one and push lower byte of flag register into stack

Again decrement SP by one and push higher byte of flag register into stack

- **POPF** – Used to copy a word at the top of the stack to the flag register.

First pop the content of stack into lower byte of flag register and increment SP by one

Again pop the content of stack into higher byte of flag register and increment SP by one

Arithmetic Instructions

These instructions are used to perform arithmetic operations like addition, subtraction, multiplication, division, etc.

Following is the list of instructions under this group –

Instructions to perform addition

- **ADD** – Used to add the provided byte to byte/word to word.

ADD AX,BX – add content of AX &BX and store result into AX reg

ADD AX, 0100 - add content of AX &0100 and store result into AX reg

ADD AX, [SI] - add content of AX &content of SI and store result into AX reg

- **ADC** – Used to add with carry.

ADC AX,BX - Add AX+BX+ carry one

ADC AX, 0100H

ADC AX, [5000H]

- **INC** – Used to increment the provided byte by 1.

INC AX - (Increment AX by one)

INC [BX] Content of BX incremented by one

- **DEC** – Used to decrement the provided byte by 1.

DEC AX - (Decrement AX by one)

DEC [BX]

- **SUB** – Used to subtract the byte from byte/word from word.

SUB AX, 0100H

SUB AX, BX

- **SBB** – It subtracts the two operands and also the borrow from the result.

SBB AX, BX (AX-BX-borrow flag ie carry flag)

SBB AX, 0100H

- **AAA** – Used to adjust ASCII after addition.

The AAA instruction works as follows:

If the least significant four bits in AL are > 9 or if AF Auxiliary carry flag) =1, it adds 6 to AL and 1 to AH.

– Both CF and AF are set

In all cases, the most significant four bits in AL are cleared

Example

MOV AH ,0 ;

MOV AL ,6 ; AL=06

ADD AL ,5 ; AL= B

AAA ; , (Output is B ie >9 so add 6 with B ie 11 then clear higher bits ie 1 , now the AL contains 01 and set AH as 1)

O/p is AH=1, AL=1 representing BCD 11

- **AAS** – Used to adjust ASCII codes after subtraction.

The AAS instruction works as follows:

If the least significant four bits in AL are > 9 or if AF =1, it subtracts 6 from AL and 1 from AH.

– Both CF and AF are set

In all cases, the most significant four bits in AL are cleared

- **AAM** – Used to adjust ASCII codes after multiplication

The AAM instruction works as follows

* AL is divided by 10

* Quotient is stored in AH

* Remainder in AL

Example

MOV AL, 5

MOV BL, 7

MUL BL ; Multiply AL by BL , result in AX

AAM ;After AAM, AX =0305h

- **AAD**- Used to adjust ASCII codes after division

The AAD instruction works as follows

* Multiplies AH by 10 and adds it to AL and sets AH to 0

If AX is 0207H before AAD

» AX is changed to 001BH after AAD

- **MUL**- this instruction multiplies unsigned byte or word by the contents of AL or AX

MUL BH (AX) <- (AL)* (BH)

MUL CX (AX) ← (AX)* (CX)

- **IMUL** – Used to multiply signed byte by byte/word by word.

One operand is AL or AX

IMUL BH

IMUL [SI]

- **DIV** – Used to divide the unsigned word by byte or unsigned double word by word.

Dividend must be in AX register

After division AL contains quotient and AH contains remainder

- **IDIV** – Used to divide the signed word by byte or signed double word by word.

- **DAA** – Used to adjust the decimal after the addition/subtraction operation.

The DAA instruction works as follows:

- * If the least significant four bits in AL are > 9 or if AF =1, it adds 6 to AL and sets AF
- * If the most significant four bits in AL are > 9 or if CF =1, it adds 60H to AL and sets CF

Example: mov AL,71H

ADD AL,43H ; AL := B4H

DAA ; AL := 14H and CF := 1

The result including the carry (i.e., 114H) is the correct answer

- **DAS** – Used to adjust decimal after subtraction.
 If AL is >9 (ie) A to F or AuxiliaryCarry=1 then
 Subtract 6 from AL
 Set Carry=1
 Otherwise AuxiliaryCarry=0 ,carry=0
- **NEG** – it forms 2's complement of specified destination in the instruction.
- **CMP** – Used to compare 2 provided byte/word.
 If both operands are equal , zero flag is set. If the source operand is > destination operand, carry flag is set else carry flag is reset.
 CMP BX,0100H
 CMP BX, [SI]
- **CBW** – (Convert signed byte to word)-Used to fill the upper byte of the word with the copies of sign bit of the lower byte.

Example

AX = 00000000 10011011 = - 155 decimal

CBW

Convert signed byte in AL to signed word in AX.

Result in AX = 11111111 10011011
 = - 155 decimal

- **CWD** – (Convert signed word to double word)-Used to fill the upper word of the double word with the sign bit of the lower word.

String Instructions

String is a group of bytes/words and their memory is always allocated in a sequential order.

Following is the list of instructions under this group –

-
- **REP** – Used to repeat the given instruction till CX register $\neq 0$.
REPE/REPZ – Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.
REPNE/REPZ – Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.
 - **MOVS/MOVSW** – Used to move the byte/word from one string to another.
a string of bytes stored in a set of consecutive memory locations is to be moved to another set of destination locations.

Example

To move a string of length 4 bytes from SRC to DST

```
MOV SI, SRC
MOV DI, DST
MOV CX, 04H
CLD; Clear the direction flag
REP MOVS
```

- **CMPS/CMPSW** – Used to compare two string bytes/words. If both strings are equal zero flag is set.

Compare, till a difference is found, two data items of 10 bytes each at STR1 and STR2

```
MOV SI, STR1
MOV DI, STR2
MOV CX, 10
CLD; Clear the direction flag
REPE CMPS
```

- **SCAS/SCASW** – Used to scan a string and compare its byte with a byte in AL or string word with a word in AX.

It compares a byte in AL or a word in AX with a byte or word pointed to by DI in ES. The direction flag determines the direction of scan.

Used with the REP prefix to find the first occurrence of a specified byte (word) in a string.

Example

Scan a string STR of 100 characters for the 'space' character, 20H

```
MOV CX, 100
MOV DI, offset STR
MOV AL, 20H
```

REPNE SCASB

- **LODS** – Used to store the string byte into AL or string word into AX.

This instruction copies a byte (word) of string from the location pointed to by SI. The SI value is automatically incremented (depending on the DF value) after each movement by 1 byte (2 bytes) for a byte string (word string).

CLD

MOV SI, Offset STR

LODSB

- **STOS(Store string or string word)**

The STOS instruction copies a byte (word) from AL (AX) to a memory location stored in [ES:DI]. DI is automatically incremented after the copy; the DF determines the increment/decrement.

MOV DI offset STR

STOSW

Control transfer or Branching Instructions

These instructions are used to transfer/branch the instructions during an execution. It includes the following instructions –

Instructions to transfer the instruction during an execution without any condition(**Unconditional Branch Instructions**)

- **CALL** – Used to call a procedure and save their return address to the stack.

CALL pushes the return address onto the stack and transfers control to a procedure. Two types of CALL are NEAR call and FAR call.

In case of NEAR call, this instruction pushes the content IP(address of next instruction) into stack.

In case of FAR call, this instruction pushes the content IP(address of next instruction)and CS register (where the code segment starts)into the stack.

- **RET** – Used to return from the procedure to the main program.

At the end of procedure, the RET instruction is executed.

RET instruction, pops the content of IP and CS from stack and control is returned to the calling program

-
- **JMP** –it unconditionally transfers the control of execution to the specified address.
 - **INT N**- In 8086 256 interrupts are defined. When INT N instruction is executed , the TYPE byte N is multiplied by 4 and contents of IP of interrupt service routine will be taken as this number and CS register set as 0000

Example

INT 20H

Type*4=20*4= 80 H

Pointer of IP and CS of the ISR (Interrupt Service Routine) is 0000:0080H

- **INTO(Interrupt on Overflow)**
This command is executed, when the overflow flag is set
- **IRET(Return from ISR)**
– Used to return from the Interrupt Service Routine to the calling program.
- **LOOP- Loop Unconditionally**
This instruction executes the part of the program from the label specified in the instruction up to the CX = 0
Label: MOV AX, 0050H
.....
.....
Loop Label

Conditional Branch Instructions

Instructions to transfer the instruction during an execution with some conditions –

- **JE/JZ** – Used to jump if equal/zero flag ZF = 1
- **JNE/JNZ** – Used to jump if not equal/zero flag ZF = 0
- **JS** – Used to jump if sign flag SF = 1
- **JNS** – Used to jump if not sign SF = 0
- **JO** – Used to jump if overflow flag OF = 1
- **JNO** – Used to jump if no overflow flag OF = 0
- **JP/JPE** – Used to jump if parity/parity even PF = 1
- **JNP** – Used to jump if not parity PF = 0
- **JB/JNAE/JC** - Used to jump if CF=1
- **JNB/JAE/JNC** - Used to jump if CF=0

-
- **JBE/JNA** – Used to jump if CF=1 or ZF=1
 - **JNBE/JA** – Used to jump if CF=0 or ZF=0
 - **JL/JNGE** – Used to jump if less than/not greater than/equal instruction satisfies.
 - **JNL/JGE** – Used to jump if not less than/if greater than instruction satisfies.
 - **JLE/JNC**- Used to jump if less than / no carry instruction satisfies
 - **JNLE/JE**- Used to jump if not less than / equal instruction satisfies

Flag Manipulation and Processor Control Instructions

These instructions are used to control the processor action by setting/resetting the flag values.

Following are the instructions under this group –

- **STC** – Used to set carry flag CF to 1
- **CLC** – Used to clear/reset carry flag CF to 0
- **CMC** – Used to put complement at the state of carry flag CF.
- **STD** – Used to set the direction flag DF to 1
- **CLD** – Used to clear/reset the direction flag DF to 0
- **STI** – Used to set the interrupt enable flag to 1, i.e., enable INTR input.
- **CLI** – Used to clear the interrupt enable flag to 0, i.e., disable INTR input.

Machine Control Instructions

WAIT- Wait for Test input pin to low

HLT- Halt the processor

NOP- No operation

ESC- Escape to external device

LOCK- Bus lock instruction

Assembler Directives

Assembler is a program used to convert an assembly language program into machine equivalent code.

There are some instructions in the assembly language program which are not a part of instruction set. These instructions are instructions to the assembler, linker and loader. These are referred to as pseudo-operations or as **assembler directives**.

There are many specialized assembler directives. Let us see the commonly used assembler directive in 8086 assembly language programming.

1. ASSUME:

It is used to tell the name of the logical segment the assembler to use for a specified segment.

E.g.: ASSUME CS: CODE tells that the instructions for a program are in a segment named CODE.

2. DB -Define Byte:

The DB directive is used to reserve byte or bytes of memory locations in the available memory. The following examples show how the DB directive is used for different purposes.

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

This statement directs the assembler to reserve four memory locations for a list named RANKS 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 initializes those locations by the ASCII equivalent of these characters.

VALUE DB 50H

This statement directs the assembler to reserve 50H memory bytes and leave them uninitialized for the variable named VALUE.

3. DW -Define Word:

The DW directives 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, 78ABH, 045CH

This makes the assembler reserve four words in memory (8 bytes), and initialize the words with the specified values in the statements. During initialization, the lower bytes are stored at the lower memory addresses, while the upper bytes are stored at the higher addresses.

NUMBER1 DW 1245H

This makes the assembler reserve one word in memory.

3. DQ -Define Quad word

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

4. DT -Define Ten Bytes:

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

7. END-End of Program:

The END directive marks the end of an assembly language program.

8. ENDP-End Procedure - Used along with the name of the procedure to indicate the end of a procedure.

E.g.:

```
PROCEDURE    ROOT    : start of procedure
.....
ROOT    ENDP                : End of procedure
```

9. 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.

```
DATA SEGMENT
-----
-----
DATA ENDS
ASSUME CS: CODE, DS: DATA
CODE SEGMENT
-----
-----
CODE ENDS
ENDS
```

10. EQU-Equate - Used to give a name to some value or symbol. Each time the assembler finds the given name in the program, it will replace the name with the value.

```
E.g : CORRECTION EQU 03H
MOV AL, CORRECTION
```

11. EVEN - Tells the assembler to increment the location counter to the next even address if it is not already at an even address.

Used because the processor can read even addressed data in one clock cycle

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

12. 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 appears, they must be declared public, using PUBLIC Directive.

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

```
MODULE 2  SEGMENT
EXTRN    FACTORIAL  FAR
MODULE2   ENDS
```

13. GLOBAL - Can be used in place of a PUBLIC directive or in place of an EXTRN directive.

It is used to make a symbol defined in one module available to other modules.

E.g.: GLOBAL DIVISOR makes the variable DIVISOR public so that it can be accessed from other modules.

14. GROUP-Used to tell the assembler to group the logical segments named after the directive into one logical group segment, allowing the contents of all the segments to be accessed from the same group segment base.

E.g.: SMALL GROUP CODE, DATA, and STACK

15. INCLUDE - Used to tell the assembler to insert a block of source code from the named file into the current source module.

This will shorten the source code.

16. LABEL- Used to give a name to the current value in the location counter.

This directive is followed by a term that specifies the type you want associated with that name.

E.g: ENTRY LABEL FAR

Here ENTRY is a label

17. LENGTH: Byte Length of a Label

- Used to refer length of data array or string

18. NAME- Used to give a specific name to each assembly module when programs consisting of several modules are written. E.g.: NAME PC_BOARD

19. LOCAL: - The labels, variables, constants or procedures declared LOCAL in a module are to be used only by that particular module.

LOCAL a,b , DATA

20. OFFSET- Used to determine the offset or displacement of a named data item from the start of the segment which contains it.

E.g.: MOV BX, OFFSET PRICES (address of PRICES set as OFFSET)

21. ORG- The location counter is set to 0000 when the assembler starts reading a segment. The ORG directive allows setting a desired value at any point in the program.

E.g.: ORG 2000H (In this case location counter initially set as 2000H)

22. PROC- Used to identify the start of a procedure.

E.g.: SMART_DIVIDE PROC FAR identifies the start of a procedure named SMART_DIVIDE and tells the assembler that the procedure is far

23. PTR- Used to assign a specific type to a variable or to a label.

E.g.: INC BYTE PTR[BX] tells the assembler that we want to increment the byte pointed to by BX

24. SEGMENT- Used to indicate the start of a logical segment.

E.g.: CODE SEGMENT indicates to the assembler the start of a logical segment called CODE

CODE SEGMENT

.....

.....

CODE ENDS

24. SHORT- Used to tell the assembler that only a 1 byte displacement is needed to code a jump instruction.

E.g.: JMP SHORT LABEL

25. TYPE - Used to tell the assembler to determine the type of a specified variable.

E.g.: ADD BX, TYPE WORD_ARRAY is used where we want to increment BX to point to the next word in an array of words.

26. **+ & - Operators-** These operators represent arithmetic addition and subtraction.

27. FAR PTR-

Example- JMP FAR PTR LABEL

The LABEL following the FAR PTR is not available within the same segment

28. NEAR PTR

JMP NEAR PTR LABEL

The LABEL following the NEAR PTR is available within the same segment

29. **SEG: -** It is used to decide the segment address of the label, variables or procedures

Microprocessor - 8086 Addressing Modes

The different ways in which a source operand is denoted in an instruction is known as **addressing modes**. There are 8 different addressing modes in 8086 programming –

Immediate addressing mode

The addressing mode in which the data operand is a part of the instruction itself is known as immediate addressing mode.

Example

```
MOV CX, 4929 H
ADD AX, 2387 H,
MOV AL, FFH
```

Register addressing mode

It means that the register is the source of an operand for an instruction.

Example

```
MOV CX, AX ; copies the contents of the 16-bit AX register into
           ; the 16-bit CX register),
ADD BX, AX
```

Direct addressing mode

The addressing mode in which the effective address of the memory location is written directly in the instruction.

Example

```
MOV AX, [1592H]
MOV AL, [0300H]
```

Register indirect addressing mode

This addressing mode allows data to be addressed at any memory location through an offset address held in any of the following registers: BP, BX, DI & SI.

Example

```
MOV AX, [BX] ; Suppose the register BX contains 4895H, then the contents
              ; 4895H are moved to AX
ADD CX, {BX}
```

Based addressing mode

In this addressing mode, the offset address of the operand is given by the sum of contents of the BX/BP registers and 8-bit/16-bit displacement.

Example

```
MOV DX, [BX+04]
ADD CL, [BX+08]
```

Indexed addressing mode

In this addressing mode, the operands offset address is found by adding the contents of SI or DI register and 8-bit/16-bit displacements.

Example

```
MOV AX,[SI]
```

```
MOV BX, [SI+16],
ADD AL, [DI+16]
```

Based-index addressing mode

In this addressing mode, the offset address of the operand is computed by summing the base register to the contents of an Index register.

Example

```
MOV AX, [BX][SI] ----(Move [BX]+[SI]) to AX
```

Based indexed with displacement mode(Relative Based Indexed)

In this addressing mode, the operands offset is computed by adding the base register contents. An Index registers contents and 8 or 16-bit displacement.

Example

```
MOV AX, [BX+DI+08],  
ADD CX, [BX+SI+16]
```

Intra segment Direct mode

In this mode the address to which the control is to be transferred lies in the same segment and it is passed to the instruction directly

Eg: JMP 4000H

Intra segment Indirect mode

In this mode the address to which the control is to be transferred lies in the same segment and it is passed to the instruction indirectly

Eg: JMP [BX]

Inter segment Direct

In this mode the address to which the control is to be transferred is in a different segment

The address is passed to the instruction directly

This mode provides a means of branching from one code segment to another code segment.

Intra segment indirect

In this mode the address to which the control is to be transferred is in a different segment

The address is passed to the instruction indirectly

Example 2.13

The contents 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.

Shifting a number four times is equivalent to multiplying it by 16_D or 10_H .

(i) Direct addressing mode

```
MOV AX, [5000H]
DS:OFFSET ⇔ 1000H: 5000H
10H*DS ⇔ 10000
Offset ⇔ + 5000
15000H - Effective address
```

(ii) Register indirect

```
MOV AX, [BX]
DS:BX ⇔ 1000H:2000H
10H*DS ⇔ 10000
[BX] ⇔ + 2000
12000H - Effective address
```

(iii) Register relative

```
MOV AX, 5000 [BX]
DS: [5000 + BX]
10H*DS ⇔ 10000
Offset ⇔ + 5000
[BX] ⇔ + 2000
17000H - Effective address
```

(iv) Based indexed

```
MOV AX, [BX] [SI]
DS:[BX + SI]
10H*DS ⇔ 10000
[BX] ⇔ + 2000
[SI] ⇔ + 3000
15000H - Effective address
```

(v) Relative based indexed

```
MOV AX, 5000 [BX] [SI]
DS: [BX + SI + 5000]
10H*DS ⇔ 10000
[BX] ⇔ + 2000
[SI] ⇔ + 3000
Offset ⇔ + 5000
1A000 - effective address
```

Use of Stack

The stack is a block of memory that may be used for temporarily storing the contents of the registers inside the CPU. It is a top-down data structure whose elements are accessed using the stack pointer (SP)

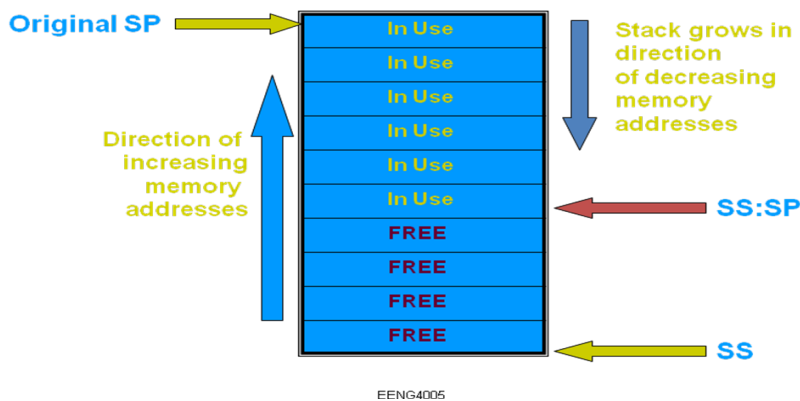
- Used to store temporary data during program execution
- One point of access - the top of the stack
- A stack is always operated as Last-In-First-Out (LIFO) storage, i.e., data are retrieved in the reverse order to which they were stored
- Instructions that directly manipulate the stack

- **PUSH** - place element on top of stack
- **POP** - remove element from top of stack
- SS - Stack Segment
- SP (stack pointer) always points to the top of the stack
 - SP initially points to top of the stack (high memory address).
 - SP decreases as data is PUSHed

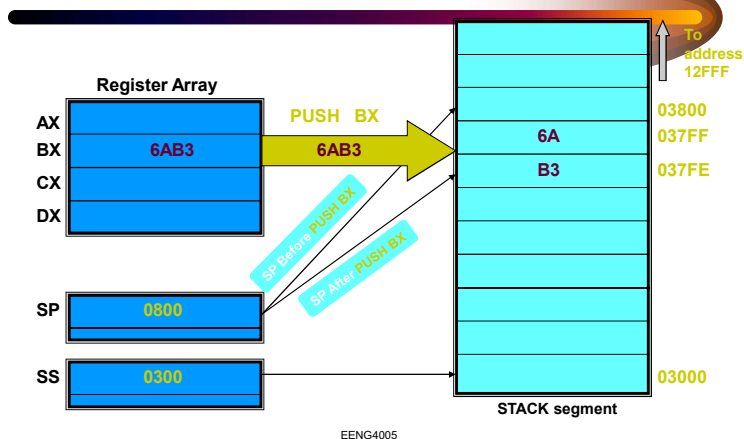
PUSH AX \Rightarrow SUB SP, 2 ; MOV [SS:SP], AX

- SP increases as data is POPed

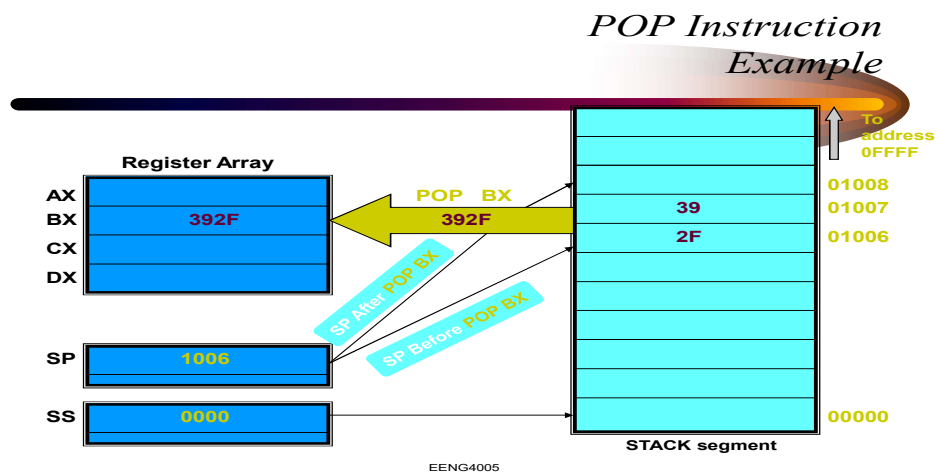
POP AX \Rightarrow MOV AX, [SS:SP] ; ADD SP, 2



PUSH Instruction Example



Push 6A into that location. Decrement top of stack by one and push B3 to that location(037FE)



Pop 39 into that location. increment top of stack by one

Purpose of Stack

Then subroutine is executed. At the end of subroutine (RET), control is returned to the calling program

Programming for Stack

```
        ASSUME CS  : CODE, DS: DATA, SS:STACK
        STACK SEGMENT
        STAKDATA DB 100H DUP(?)
        STACK ENDS
        CODE SEGMENT
START:MOV AX,DATA
        MOV DS,AX
        MOV AX, STACK
        MOV SS ,AX
        MOV SP,OFFSET STAKDATA
        MOV SI,OFFSET SQUARES
        MOV AL,00
NEXTNUM: CALL SQUARE
        MOV BYTE PTR [SI], AH
```

MOV AH,4CH

INT 21H

PROCEDURE SQUARRE NEAR

MOV BH,AL

MOV CH,AL

XOR AL,AL

AGAN:

ADD AL, CH

DAA

DCR CH

JNZ AGAIN

MOV AH,AL

MOV AL,BH

RET

SQUARE ENDP

CODE END S

END START