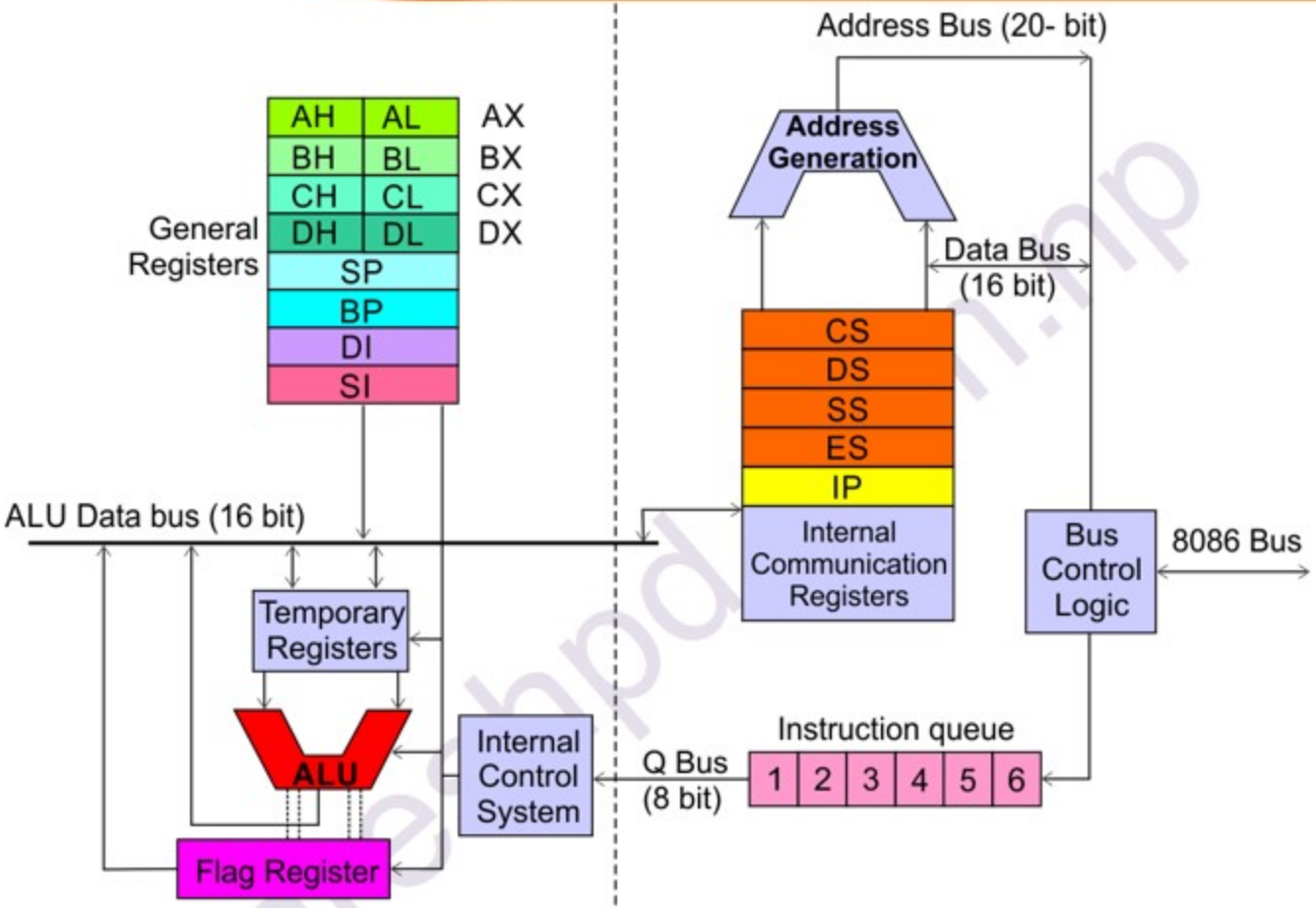
Architecture



Execution Unit (EU)

EU executes instructions that have already been fetched by the BIU.

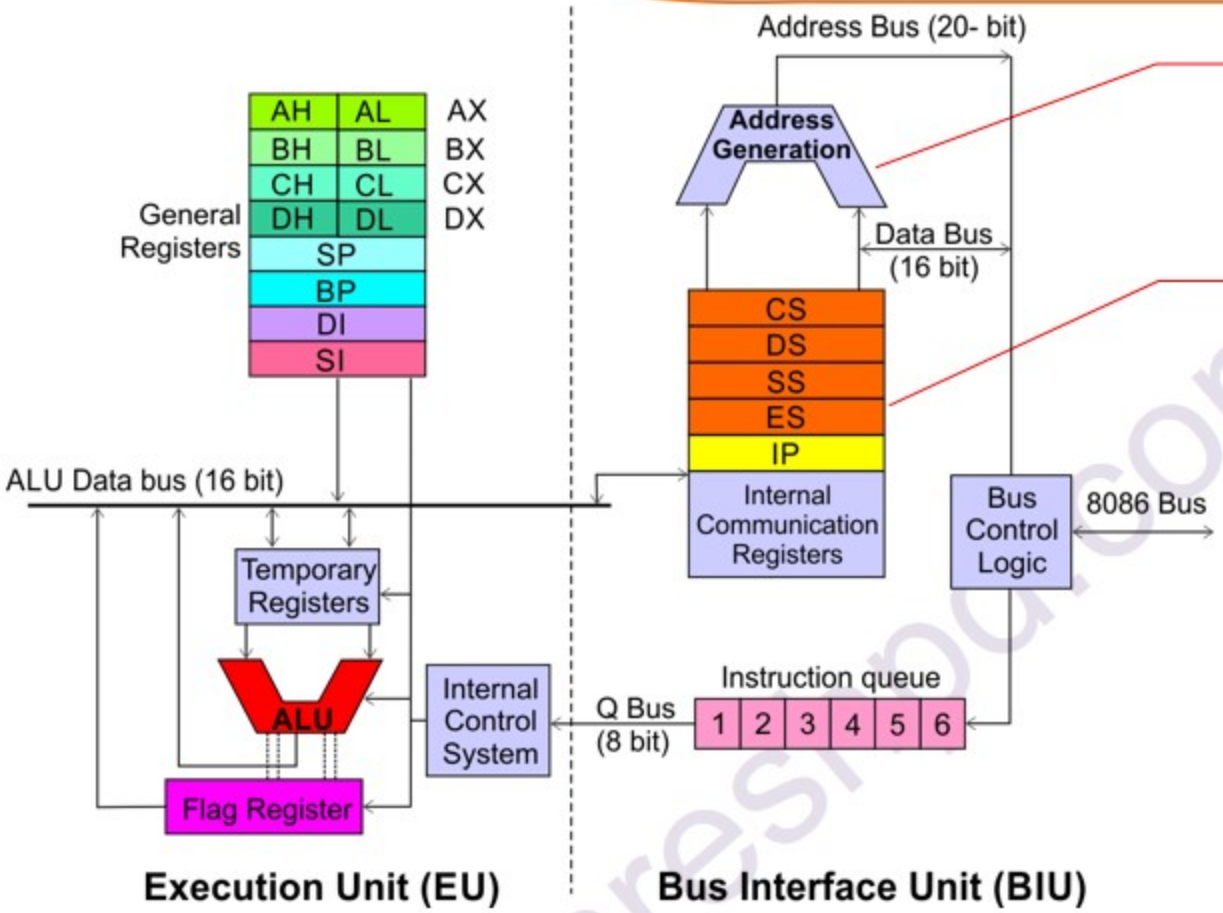
BIU and EU functions separately.

Bus Interface Unit (BIU)

BIU fetches instructions, reads data from memory and I/O ports, writes data to memory and I/ O ports.

Architecture

Bus Interface Unit (BIU)



Dedicated Adder to generate 20 bit address

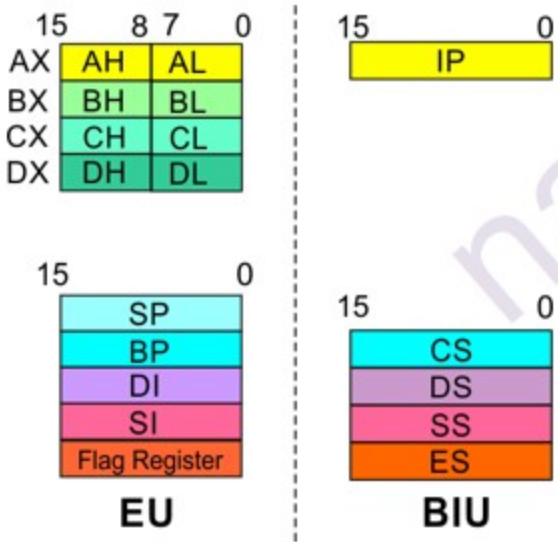
Four 16-bit segment registers

Code Segment (CS)
Data Segment (DS)
Stack Segment (SS)
Extra Segment (ES)

Segment Registers

Code Segment Register

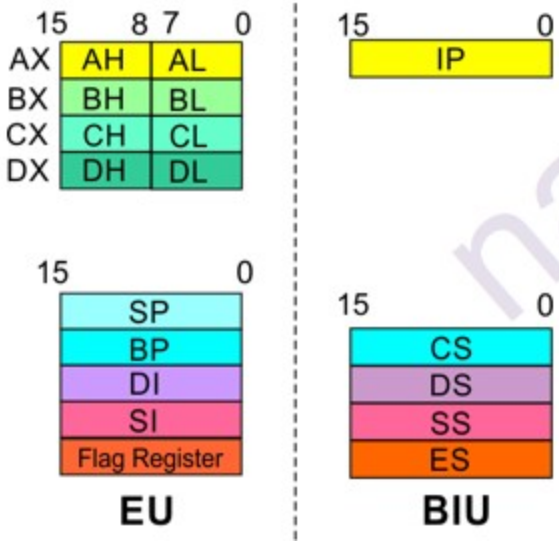
- 16-bit
- CS contains the base or start of the current code segment; IP contains the distance or offset from this address to the next instruction byte to be fetched.
- BIU computes the 20-bit physical address by logically shifting the contents of CS 4-bits to the left and then adding the 16-bit contents of IP.
- That is, all instructions of a program are relative to the contents of the CS register multiplied by 16 and then offset is added provided by the IP.



Segment Registers

Data Segment Register

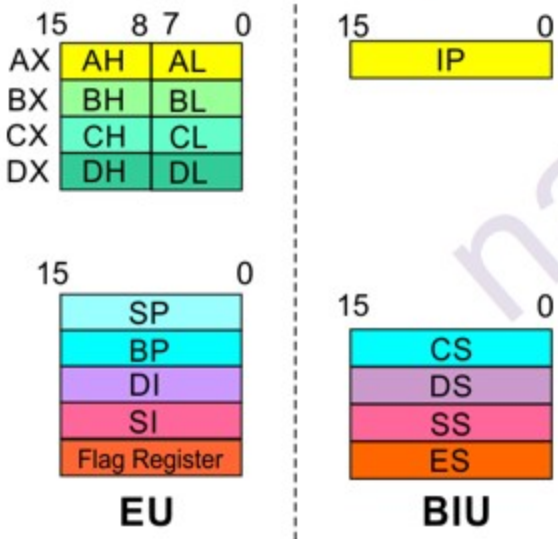
- 16-bit
- Points to the current data segment; operands for most instructions are fetched from this segment.
- The 16-bit contents of the Source Index (SI) or Destination Index (DI) or a 16-bit displacement are used as offset for computing the 20-bit physical address.



Segment Registers

Stack Segment Register

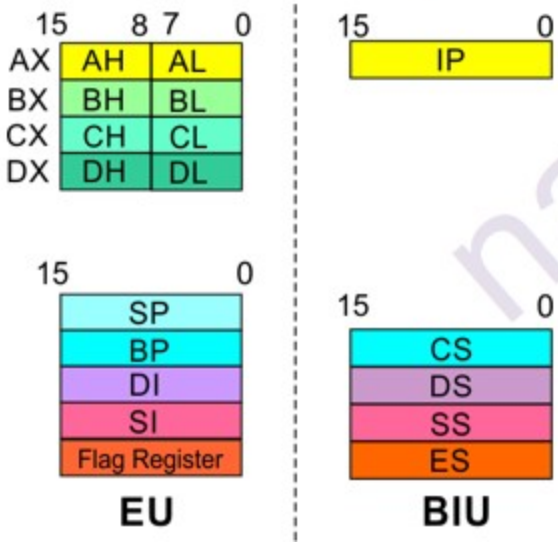
- 16-bit
- Points to the current stack.
- The 20-bit physical stack address is calculated from the Stack Segment (SS) and the Stack Pointer (SP) for stack instructions such as PUSH and POP.
- In based addressing mode, the 20-bit physical stack address is calculated from the Stack segment (SS) and the Base Pointer (BP).



Segment Registers

Extra Segment Register

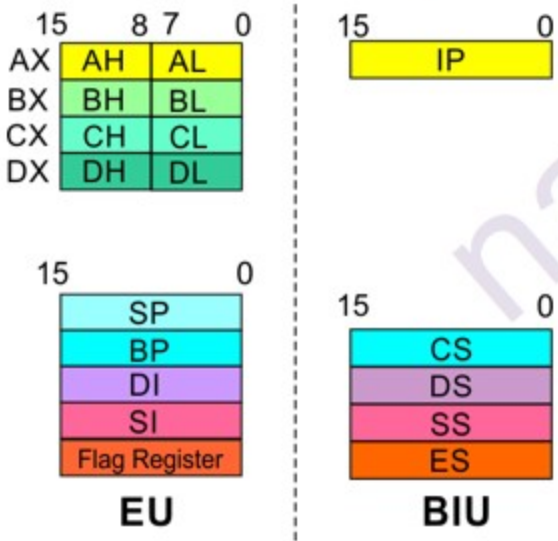
- 16-bit
- Points to the extra segment in which data (in excess of 64K pointed to by the DS) is stored.
- String instructions use the ES and DI to determine the 20-bit physical address for the destination.



Segment Registers

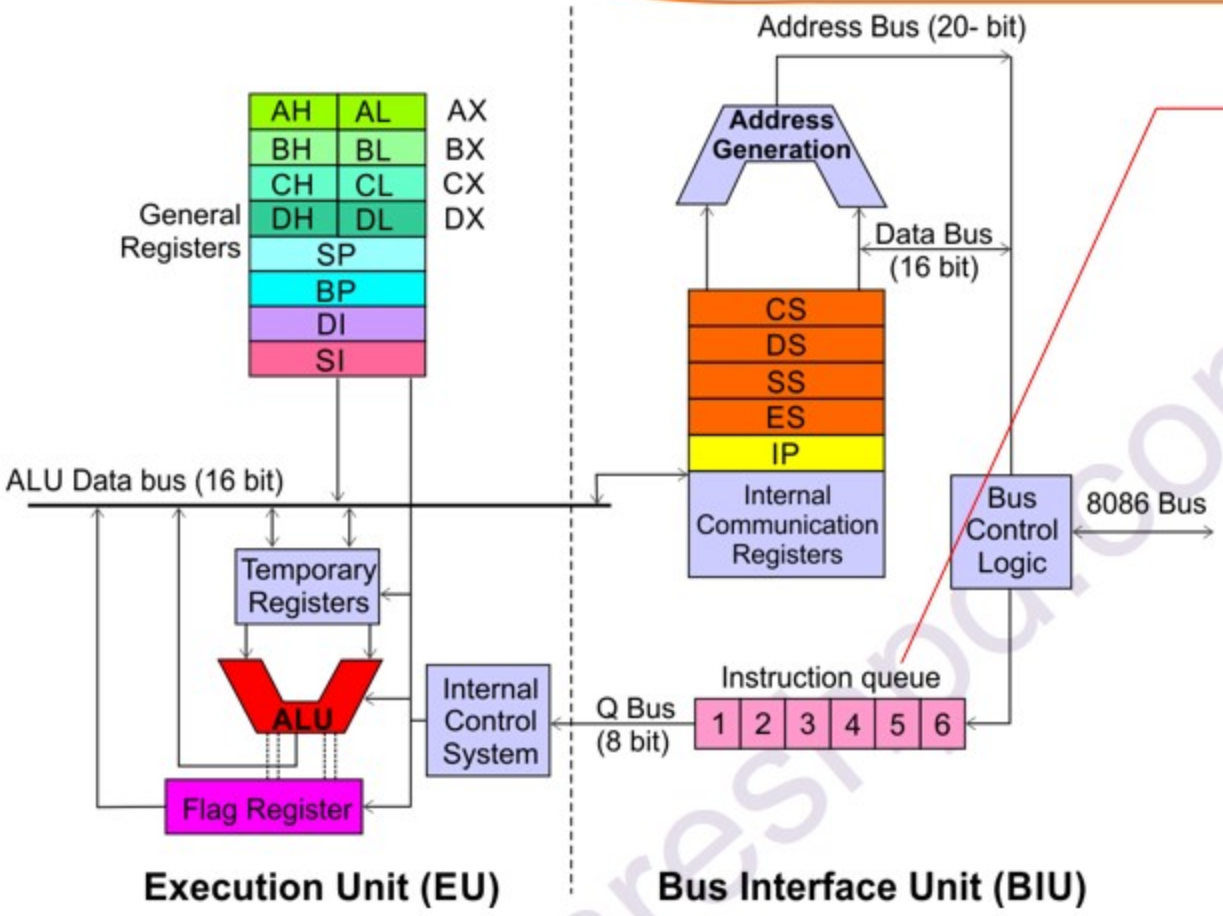
Instruction Pointer

- 16-bit
- Always points to the next instruction to be executed within the currently executing code segment.
- So, this register contains the 16-bit offset address pointing to the next instruction code within the 64Kb of the code segment area.
- Its content is automatically incremented as the execution of the next instruction takes place.



Architecture

Bus Interface Unit (BIU)



Instruction queue

- **A group of First-In-First-Out (FIFO) in which up to 6 bytes of instruction code are pre fetched from the memory ahead of time.**
- **This is done in order to speed up the execution by overlapping instruction fetch with execution.**
- **This mechanism is known as pipelining.**

Architecture

Execution Unit (EU)

EU decodes and executes instructions.

A decoder in the EU control system translates instructions.

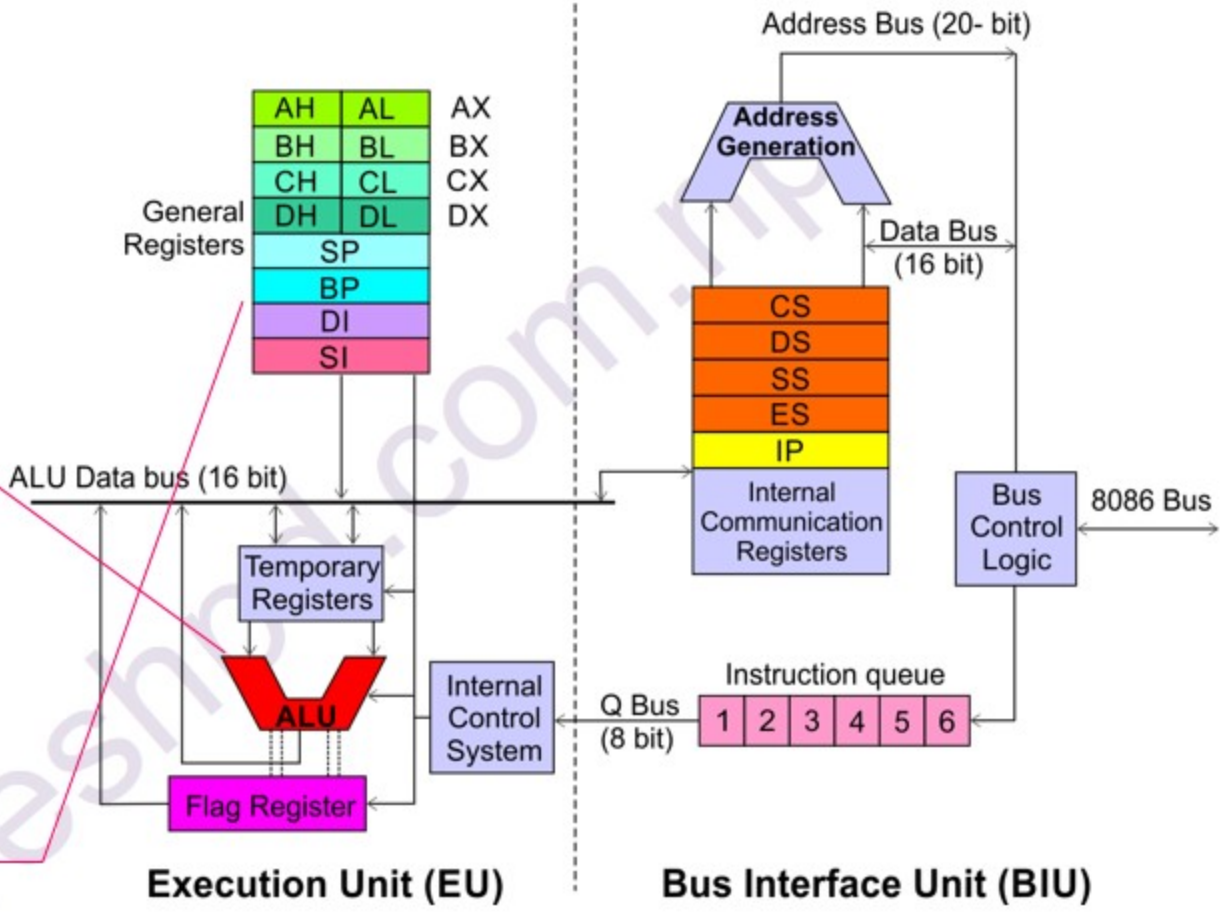
16-bit ALU for performing arithmetic and logic operation

Four general purpose registers(AX, BX, CX, DX);

Pointer registers (Stack Pointer, Base Pointer);

and

Index registers (Source Index, Destination Index) each of 16-bits



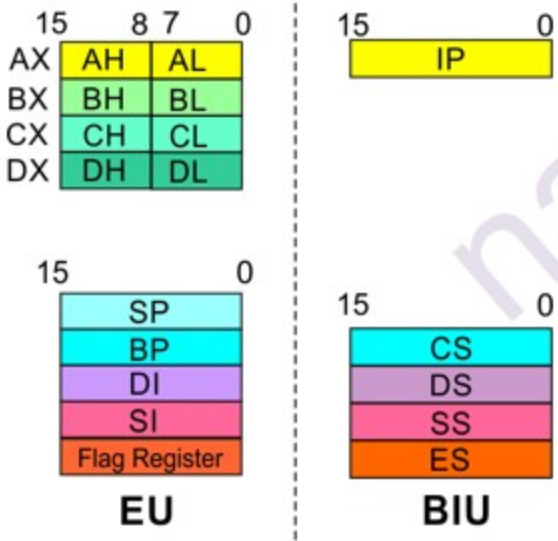
Some of the 16 bit registers can be used as two 8 bit registers as :

AX can be used as AH and AL
BX can be used as BH and BL
CX can be used as CH and CL
DX can be used as DH and DL

EU Registers

Accumulator Register (AX)

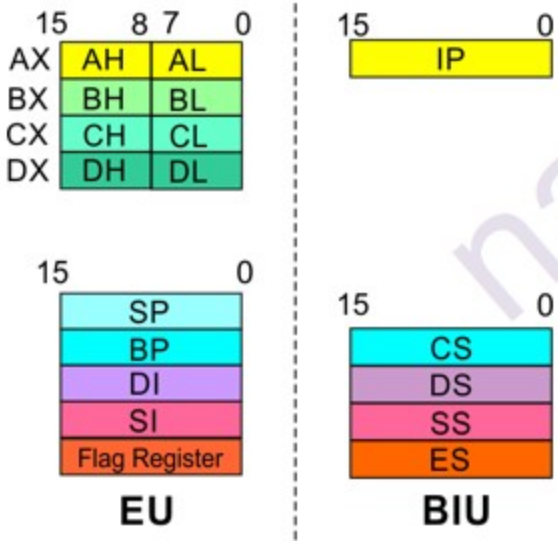
- Consists of two 8-bit registers AL and AH, which can be combined together and used as a 16-bit register AX.
- AL in this case contains the low order byte of the word, and AH contains the high-order byte.
- The I/O instructions use the AX or AL for inputting / outputting 16 or 8 bit data to or from an I/O port.
- Multiplication and Division instructions also use the AX or AL.



EU Registers

Base Register (BX)

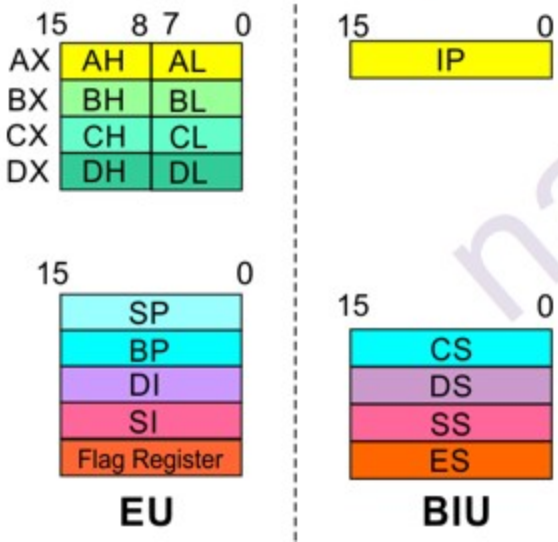
- Consists of two 8-bit registers BL and BH, which can be combined together and used as a 16-bit register BX.
- BL in this case contains the low-order byte of the word, and BH contains the high-order byte.
- This is the only general purpose register whose contents can be used for addressing the 8086 memory.
- All memory references utilizing this register content for addressing use DS as the default segment register.



EU Registers

Counter Register (CX)

- Consists of two 8-bit registers CL and CH, which can be combined together and used as a 16-bit register CX.
- When combined, CL register contains the low order byte of the word, and CH contains the high-order byte.
- Instructions such as **SHIFT**, **ROTATE** and **LOOP** use the contents of CX as a counter.



Example:

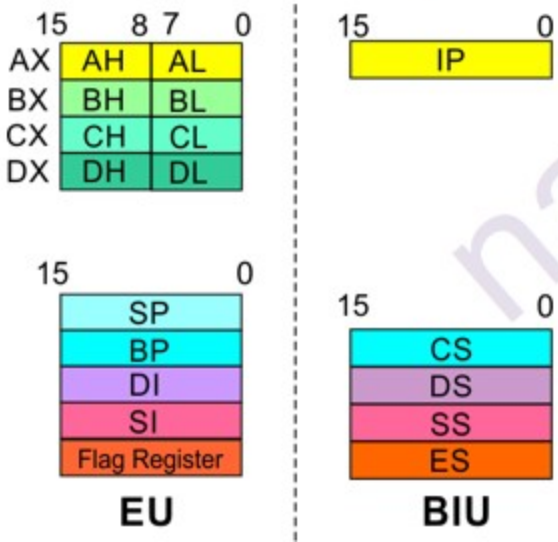
The instruction LOOP START automatically decrements CX by 1 without affecting flags and will check if [CX] = 0.

If it is zero, 8086 executes the next instruction; otherwise the 8086 branches to the label START.

EU Registers

Data Register (DX)

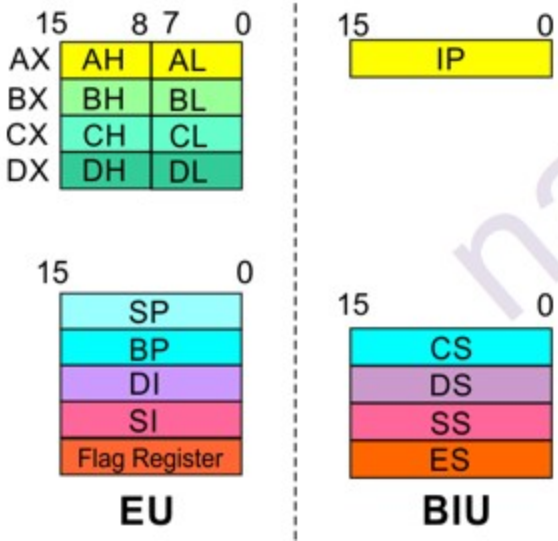
- Consists of two 8-bit registers DL and DH, which can be combined together and used as a 16-bit register DX.
- When combined, DL register contains the low order byte of the word, and DH contains the high-order byte.
- Used to hold the high 16-bit result (data) in 16 X 16 multiplication or the high 16-bit dividend (data) before a 32 ÷ 16 division and the 16-bit reminder after division.



EU Registers

Stack Pointer (SP) and Base Pointer (BP)

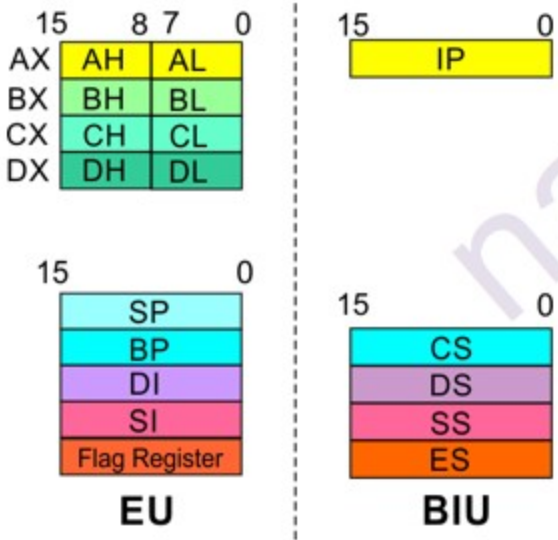
- SP and BP are used to access data in the stack segment.
- SP is used as an offset from the current SS during execution of instructions that involve the stack segment in the external memory.
- SP contents are automatically updated (incremented/ decremented) due to execution of a POP or PUSH instruction.
- BP contains an offset address in the current SS, which is used by instructions utilizing the based addressing mode.



EU Registers

Source Index (SI) and Destination Index (DI)

- Used in indexed addressing.
- Instructions that process data strings use the SI and DI registers together with DS and ES respectively in order to distinguish between the source and destination addresses.



Architecture

Execution Unit (EU)

Flag Register

Auxiliary Carry Flag

This is set, if there is a carry from the lowest nibble, i.e, bit three during addition, or borrow for the lowest nibble, i.e, bit three, during subtraction.

Carry Flag

This flag is set, when there is a carry out of MSB in case of addition or a borrow in case of subtraction.

Sign Flag

This flag is set, when the result of any computation is negative

Zero Flag

This flag is set, if the result of the computation or comparison performed by an instruction is zero

Parity Flag

This flag is set to 1, if the lower byte of the result contains even number of 1's ; for odd number of 1's set to zero.

Over flow Flag

This flag is set, if an overflow occurs, i.e, if the result of a signed operation is large enough to accommodate in a destination register. The result is of more than 7-bits in size in case of 8-bit signed operation and more than 15-bits in size in case of 16-bit sign operations, then the overflow will be set.

Tarp Flag

If this flag is set, the processor enters the single step execution mode by generating internal interrupts after the execution of each instruction

Direction Flag

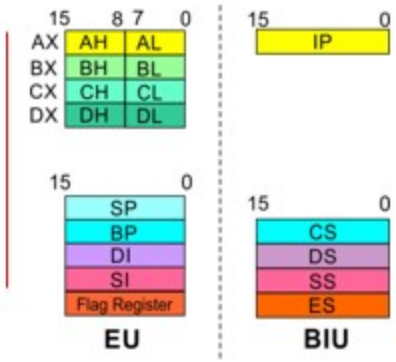
This is used by string manipulation instructions. If this flag bit is '0', the string is processed beginning from the lowest address to the highest address, i.e., auto incrementing mode. Otherwise, the string is processed from the highest address towards the lowest address, i.e., auto incrementing mode.

Interrupt Flag

Causes the 8086 to recognize external mask interrupts; clearing IF disables these interrupts.

Architecture

8086 registers categorized into 4 groups



Sl.No.	Type	Register width	Name of register
1	General purpose register	16 bit	AX, BX, CX, DX
		8 bit	AL, AH, BL, BH, CL, CH, DL, DH
2	Pointer register	16 bit	SP, BP
3	Index register	16 bit	SI, DI
4	Instruction Pointer	16 bit	IP
5	Segment register	16 bit	CS, DS, SS, ES
6	Flag (PSW)	16 bit	Flag register

Architecture

Registers and Special Functions

Register	Name of the Register	Special Function
AX	16-bit Accumulator	Stores the 16-bit results of arithmetic and logic operations
AL	8-bit Accumulator	Stores the 8-bit results of arithmetic and logic operations
BX	Base register	Used to hold base value in base addressing mode to access memory data
CX	Count Register	Used to hold the count value in SHIFT, ROTATE and LOOP instructions
DX	Data Register	Used to hold data for multiplication and division operations
SP	Stack Pointer	Used to hold the offset address of top stack memory
BP	Base Pointer	Used to hold the base value in base addressing using SS register to access data from stack memory
SI	Source Index	Used to hold index value of source operand (data) for string instructions
DI	Data Index	Used to hold the index value of destination operand (data) for string operations

ADDRESSING MODES

&

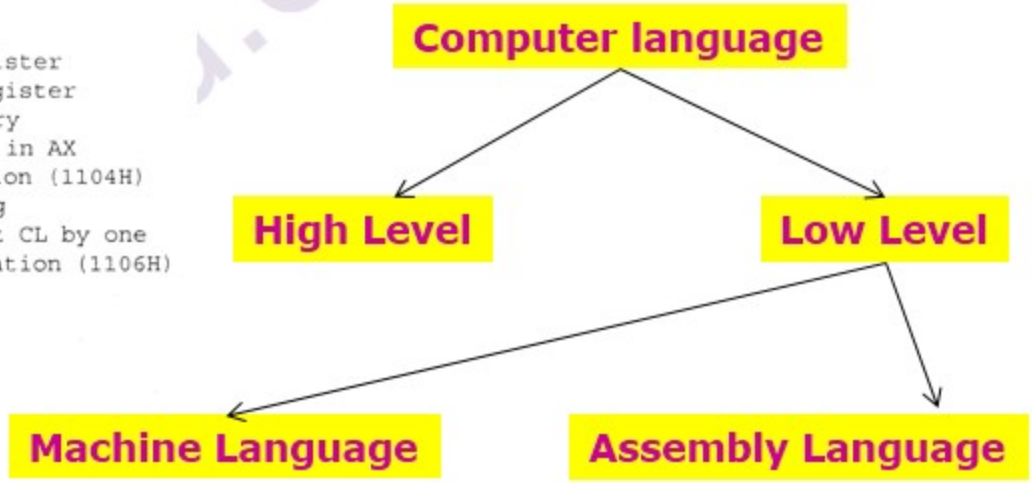
Instruction set

Introduction

```
;PROGRAM TO ADD TWO 16-BIT DATA (METHOD-1)

DATA SEGMENT              ;Assembler directive

        ORG 1104H         ;Assembler directive
        SUM DW 0          ;Assembler directive
        CARRY DB 0        ;Assembler directive

DATA ENDS                 ;Assembler directive

CODE SEGMENT              ;Assembler directive

        ASSUME CS:CODE    ;Assembler directive
        ASSUME DS:DATA    ;Assembler directive
        ORG 1000H         ;Assembler directive

        MOV AX,205AH      ;Load the first data in AX register
        MOV BX,40EDH      ;Load the second data in BX register
        MOV CL,00H        ;Clear the CL register for carry
        ADD AX,BX         ;Add the two data, sum will be in AX
        MOV SUM,AX        ;Store the sum in memory location (1104H)
        JNC AHEAD         ;Check the status of carry flag
        INC CL            ;If carry flag is set,increment CL by one
AHEAD:  MOV CARRY,CL      ;Store the carry in memory location (1106H)
        HLT

CODE ENDS                 ;Assembler directive
END                       ;Assembler directive
```

Program
A set of instructions written to solve a problem.

Instruction
Directions which a microprocessor follows to execute a task or part of a task.



- **Binary bits**

- **English Alphabets**
- **'Mnemonics'**
- **Assembler**
  Mnemonics → Machine Language

Introduction

Program is a set of instructions written to solve a problem. Instructions are the directions which a microprocessor follows to execute a task or part of a task. Broadly, computer language can be divided into two parts as high-level language and low level language. Low level language are machine specific. Low level language can be further divided into machine language and assembly language.

Machine language is the only language which a machine can understand. Instructions in this language are written in binary bits as a specific bit pattern. The computer interprets this bit pattern as an instruction to perform a particular task. The entire program is a sequence of binary numbers. This is a machine-friendly language but not user friendly. Debugging is another problem associated with machine language.

To overcome these problems, programmers develop another way in which instructions are written in English alphabets. This new language is known as Assembly language. The instructions in this language are termed *mnemonics. As* microprocessor can only understand the machine language so mnemonics are translated into machine language either manually or by a program known as *assembler.*

Efficient software development for the microprocessor requires a complete familiarity with the instruction set, their format and addressing modes. Here in this chapter, we will focus on the addressing modes and instructions formats of microprocessor 8086.

ADDRESSING MODES

Addressing Modes

- Every instruction of a program has to operate on a data.
- The different ways in which a source operand is denoted in an instruction are known as addressing modes.

1. Register Addressing 2. Immediate Addressing	Group I : Addressing modes for register and immediate data
3. Direct Addressing 4. Register Indirect Addressing 5. Based Addressing 6. Indexed Addressing 7. Based Index Addressing 8. String Addressing	Group II : Addressing modes for memory data
9. Direct I/O port Addressing 10. Indirect I/O port Addressing	Group III : Addressing modes for I/O ports
11. Relative Addressing	Group IV : Relative Addressing mode
12. Implied Addressing	Group V : Implied Addressing mode

Addressing Modes

Group I : Addressing modes for register and immediate data

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

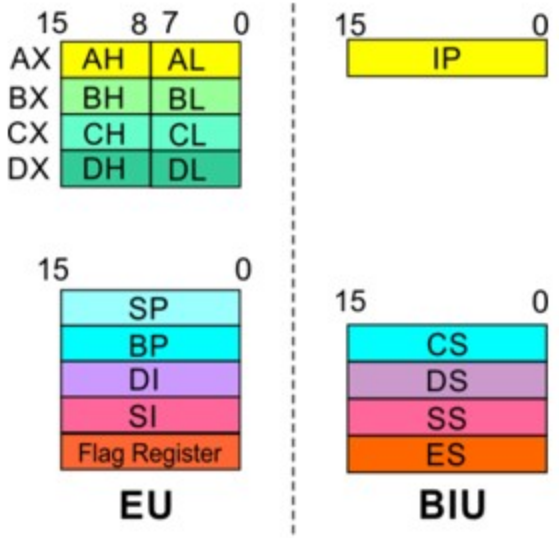
The instruction will specify the name of the register which holds the data to be operated by the instruction.

Example:

MOV CL, DH

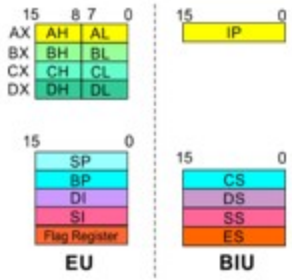
The content of 8-bit register DH is moved to another 8-bit register CL

(CL) ← (DH)



Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In immediate addressing mode, an 8-bit or 16-bit data is specified as part of the instruction

Example:

MOV DL, 08H

The 8-bit data (08_H) given in the instruction is moved to DL

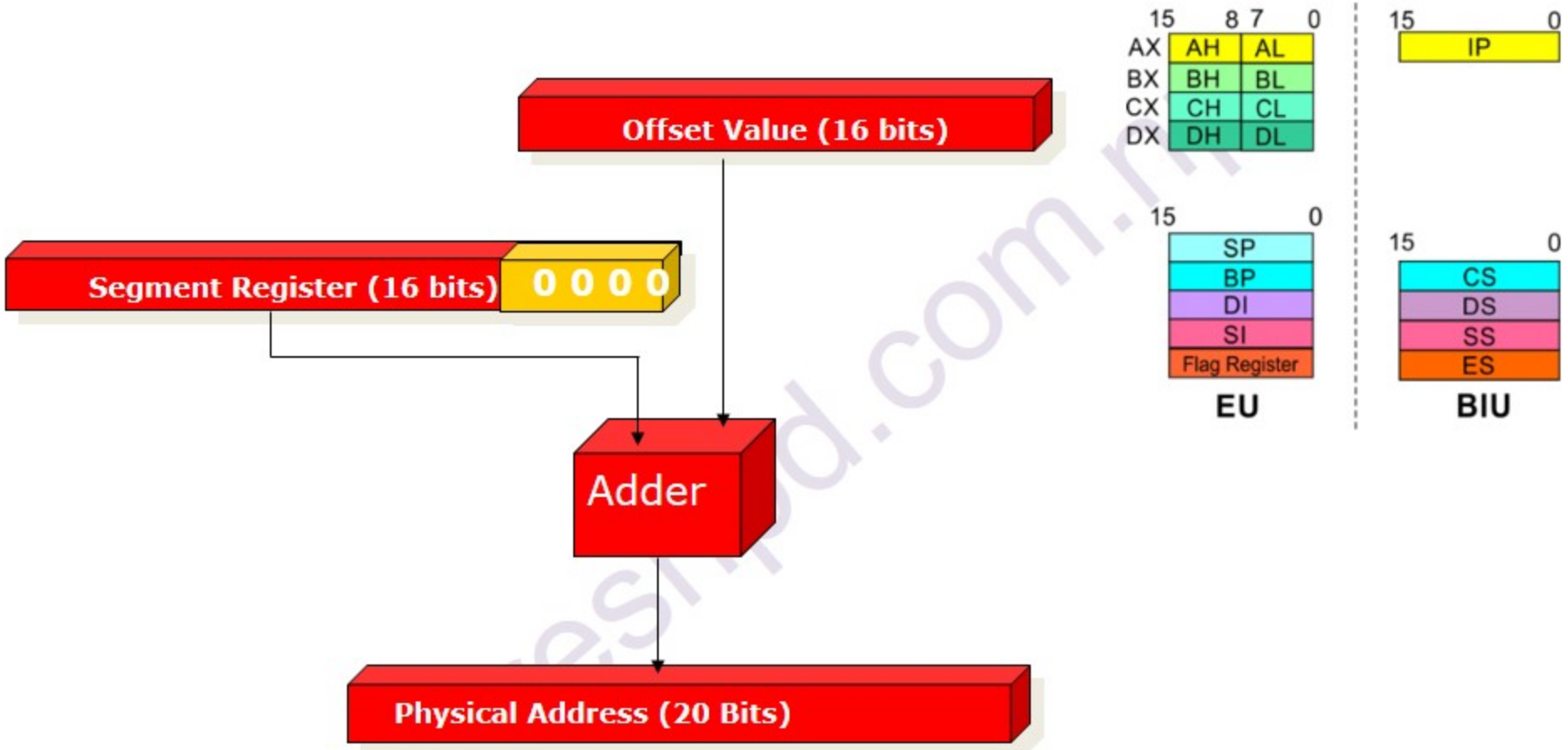
(DL) ← 08_H

MOV AX, 0A9FH

The 16-bit data ($0A9F_H$) given in the instruction is moved to AX register

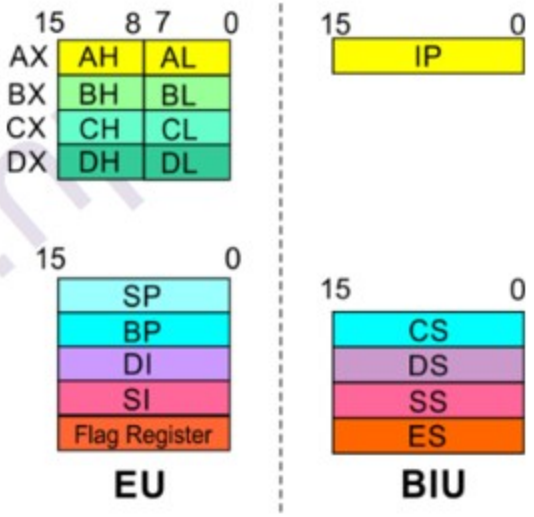
(AX) ← $0A9F_H$

Addressing Modes : Memory Access



Addressing Modes : Memory Access

- **20 Address lines ⇒ 8086 can address up to 2^{20} = 1M bytes of memory**
- **However, the largest register is only 16 bits**
- **Physical Address will have to be calculated**
  Physical Address : Actual address of a byte in memory. i.e. the value which goes out onto the address bus.
- **Memory Address represented in the form – Seg : Offset (Eg - 89AB:F012)**
- **Each time the processor wants to access memory, it takes the contents of a segment register, shifts it one hexadecimal place to the left (same as multiplying by 16_{10}), then add the required offset to form the 20- bit address**

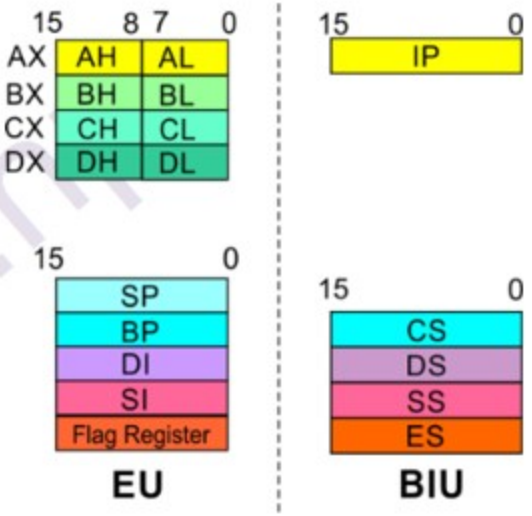


16 bytes of contiguous memory

89AB : F012 → 89AB → 89AB0 (Paragraph to byte → 89AB x 10 = 89AB0)
F012 → 0F012 (Offset is already in byte unit)
+ -------
98AC2 (The absolute address)

Addressing Modes : Memory Access

- **To access memory we use these four registers: BX, SI, DI, BP**
- **Combining these registers inside [ ] symbols, we can get different memory locations (Effective Address, EA)**
- **Supported combinations:**

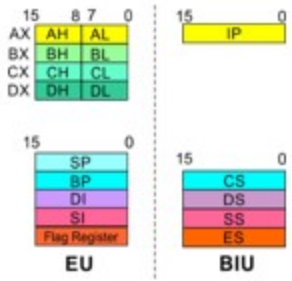


[BX + SI] [BX + DI] [BP + SI] [BP + DI]	[SI] [DI] d16 (variable offset only) [BX]	[BX + SI + d8] [BX + DI + d8] [BP + SI + d8] [BP + DI + d8]
[SI + d8] [DI + d8] [BP + d8] [BX + d8]	[BX + SI + d16] [BX + DI + d16] [BP + SI + d16] [BP + DI + d16]	[SI + d16] [DI + d16] [BP + d16] [BX + d16]

BX **BP**	**SI** **DI**	**+ disp**

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



Here, the effective address of the memory location at which the data operand is stored is given in the instruction.

The effective address is just a 16-bit number written directly in the instruction.

Example:

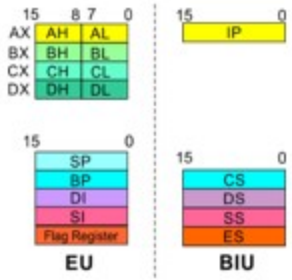
```
MOV  BX, [1354H]
MOV  BL, [0400H]
```

The square brackets around the 1354_H denotes the contents of the memory location. When executed, this instruction will copy the contents of the memory location into BX register.

This addressing mode is called direct because the displacement of the operand from the segment base is specified directly in the instruction.

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In Register indirect addressing, name of the register which holds the effective address (EA) will be specified in the instruction.

Registers used to hold EA are any of the following registers:

BX, BP, DI and SI.

Content of the DS register is used for base address calculation.

Example:

MOV CX, [BX]

Note : Register/ memory enclosed in brackets refer to content of register/ memory

Operations:

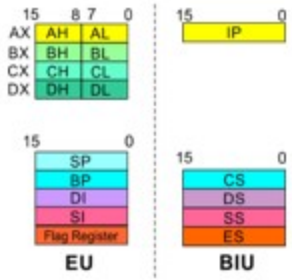
EA = (BX)
BA = (DS) x 16_{10}
MA = BA + EA

(CX) ← (MA) or,

(CL) ← (MA)
(CH) ← (MA +1)

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In Based Addressing, **BX or BP** is used to hold the base value for effective address and a **signed 8-bit** or **unsigned 16-bit** displacement will be specified in the instruction.

In case of 8-bit displacement, it is **sign extended** to 16-bit before adding to the base value.

When **BX** holds the base value of EA, 20-bit physical address is calculated from **BX and DS**.

When **BP** holds the base value of EA, **BP and SS** is used.

Example:

MOV AX, [BX + 08H]

Operations:

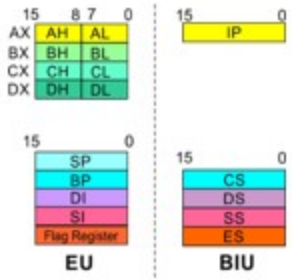
$0008_H \leftarrow 08_H$ (Sign extended)
$EA = (BX) + 0008_H$
$BA = (DS) \times 16_{10}$
$MA = BA + EA$

$(AX) \leftarrow (MA)$ or,

$(AL) \leftarrow (MA)$
$(AH) \leftarrow (MA + 1)$

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



SI or DI register is used to hold an index value for memory data and a signed 8-bit or unsigned 16-bit displacement will be specified in the instruction.

Displacement is added to the index value in **SI** or **DI** register to obtain the **EA**.

In case of 8-bit displacement, it is sign extended to 16-bit before adding to the base value.

Example:

MOV CX, [SI + 0A2H]

Operations:

$FFA2_H \leftarrow A2_H$ (Sign extended)

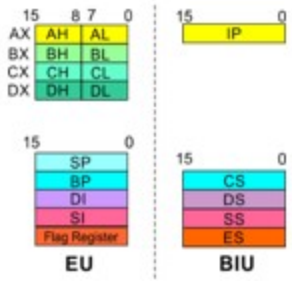
$EA = (SI) + FFA2_H$
$BA = (DS) \times 16_{10}$
$MA = BA + EA$

$(CX) \leftarrow (MA)$ or,

$(CL) \leftarrow (MA)$
$(CH) \leftarrow (MA + 1)$

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In Based Index Addressing, the effective address is computed from the sum of a base register (BX or BP), an index register (SI or DI) and a displacement.

Example:

MOV DX, [BX + SI + 0AH]

Operations:

$000A_H \leftarrow 0A_H$ (Sign extended)

$EA = (BX) + (SI) + 000A_H$
$BA = (DS) \times 16_{10}$
$MA = BA + EA$

$(DX) \leftarrow (MA)$ or,

$(DL) \leftarrow (MA)$
$(DH) \leftarrow (MA + 1)$

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

Note : Effective address of the Extra segment register

Employed in string operations to operate on string data.

The effective address (EA) of source data is stored in SI register and the EA of destination is stored in DI register.

Segment register for calculating base address of source data is DS and that of the destination data is ES

Example: MOVS BYTE

Operations:

Calculation of source memory location:

EA = (SI) BA = (DS) x 16_{10} MA = BA + EA

Calculation of destination memory location:

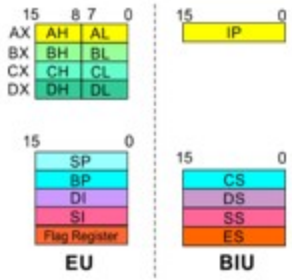
EA_E = (DI) BA_E = (ES) x 16_{10} $MA_E = BA_E + EA_E$

(MAE) ← (MA)

If DF = 1, then (SI) ← (SI) – 1 and (DI) = (DI) - 1
If DF = 0, then (SI) ← (SI) +1 and (DI) = (DI) + 1

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



These addressing modes are used to access data from standard I/O mapped devices or ports.

In direct port addressing mode, an 8-bit port address is directly specified in the instruction.

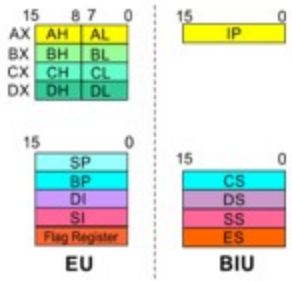
Example: IN AL, [09H]

Operations: $PORT_{addr} = 09_H$
(AL) ← (PORT)

Content of port with address 09_H is moved to AL register

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In this addressing mode, the effective address of a program instruction is specified relative to Instruction Pointer (IP) by an 8-bit signed displacement.

Example: JZ 0AH

Operations:

$000A_H \leftarrow 0A_H$ **(sign extend)**

If ZF = 1, then

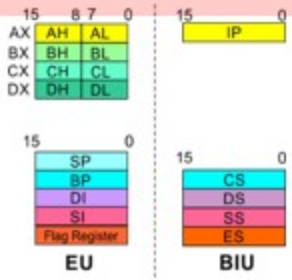
$EA = (IP) + 000A_H$
$BA = (CS) \times 16_{10}$
$MA = BA + EA$

If ZF = 1, then the program control jumps to new address calculated above.

If ZF = 0, then next instruction of the program is executed.

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



Instructions using this mode have no operands. The instruction itself will specify the data to be operated by the instruction.

Example: CLC

This clears the carry flag to zero.

Assembly Language vs Machine Language Programming

- Machine Language Programming
  - Writing a list of numbers representing the bytes of machine instructions to be executed and data constants to be used by the program
- Assembly Language Programming
  - Using symbolic instructions to represent the raw data that will form the machine language program and initial data constants

Assembly Language vs Machine Language Programming

- Mnemonics represent Machine Instructions
  - Each mnemonic used represents a single machine instruction
  - The assembler performs the translation
- Some mnemonics require operands
  - Operands provide additional information
    - register, constant, address, or variable

Addressing mode and instruction set

- The different ways in which a processor can access data are called addressing modes

How 8086 accesses data?

- 8086 assembly language instructions can be used to illustrate the addressing modes

Addressing Modes

- Every instruction of a program has to operate on a data.
- The different ways in which a source operand is denoted in an instruction are known as addressing modes.

1. Register Addressing 2. Immediate Addressing	Group I : Addressing modes for register and immediate data
3. Direct Addressing 4. Register Indirect Addressing 5. Based Addressing 6. Indexed Addressing 7. Based Index Addressing 8. String Addressing	Group II : Addressing modes for memory data
9. Direct I/O port Addressing 10. Indirect I/O port Addressing	Group III : Addressing modes for I/O ports
11. Relative Addressing	Group IV : Relative Addressing mode
12. Implied Addressing	Group V : Implied Addressing mode

Addressing Modes

Register Addressing

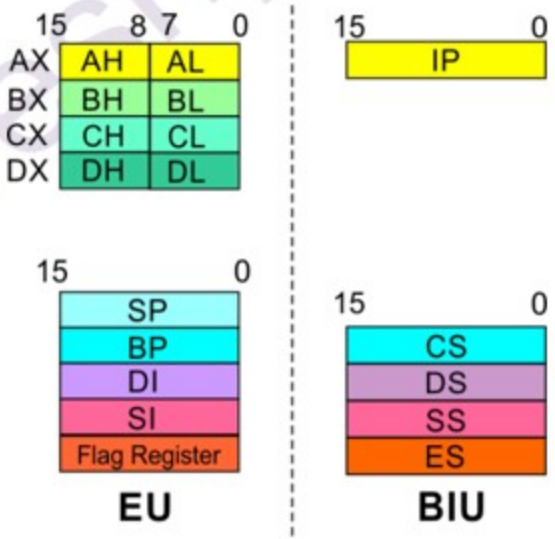
The instruction will specify the name of the register which holds the data to be operated by the instruction.

Example:

MOV CL, DH

The content of 8-bit register DH is moved to another 8-bit register CL

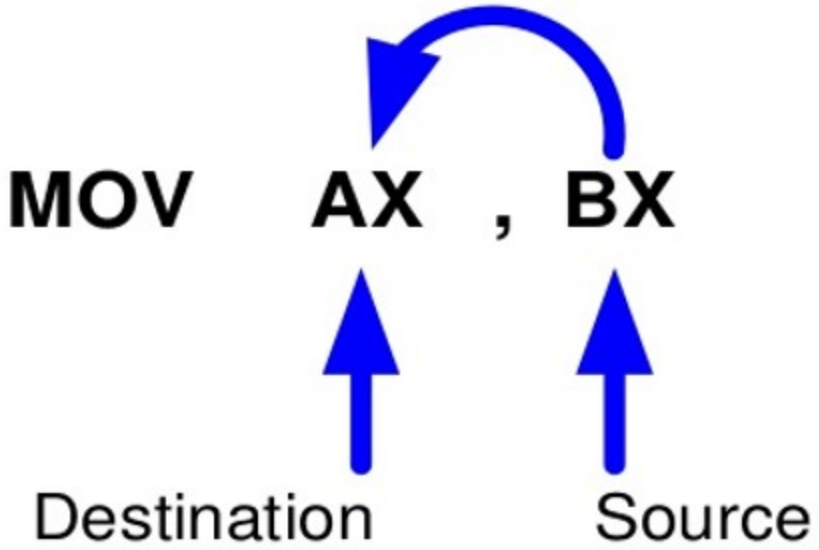
(CL) ← (DH)



Addressing Modes

Register Addressing

The MOV instruction



Function: Copy a word or byte from a register, memory location, or immediate number to a register or memory location. Source and destination must be of the same size and cannot both be memory locations.

Addressing Modes

Register Addressing

Syntax as: Mov Dest, Source

Mnemonic	Meaning	Format	Operation	Flags affected
MOV	Move	Mov D,S	(S) → (D)	None

Destination	Source
Memory	Accumulator
Accumulator	Memory
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Seg reg	Reg 16
Seg reg	Mem 16
Reg 16	Seg reg
Memory	Seg reg

NO MOV

Memory → Memory
Immediate → Segment Register
Segment Register → Segment Register

MOV AX,1

MOV BX, AX

immediate

Register

Addressing Modes

Register Addressing

Rule #1:
moving a value that is too large into a register will cause an error

```
MOV    BL,7F2H          ;Illegal: 7F2H is larger than 8 bits
MOV    AX,2FE456H       ;Illegal
```

Rule #2:
Data can be moved directly into nonsegment registers only
(Values cannot be loaded directly into any segment register.
To load a value into a segment register, first load it to a nonsegment register and then move it to the segment register.)

```
MOV    AX,2345H          MOV    DI,1400H
MOV    DS,AX             MOV    ES,DI
```

Rule #3:
If a value less than FFH is moved into a 16-bit register, the rest of the bits are assumed to be all zeros.

```
MOV BX, 5          BX =0005
                   BH = 00, BL = 05
```

Addressing Modes

Immediate Addressing

In immediate addressing mode, an 8-bit or 16-bit data is specified as part of the instruction

Example:

MOV DL, 08H

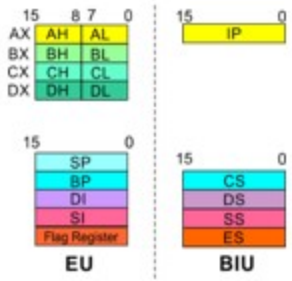
The 8-bit data (08_H) given in the instruction is moved to DL

$(DL) \leftarrow 08_H$

MOV AX, 0A9FH

The 16-bit data ($0A9F_H$) given in the instruction is moved to AX register

$(AX) \leftarrow 0A9F_H$



Addressing Modes

Direct Addressing

Here, the effective address of the memory location at which the data operand is stored is given in the instruction.

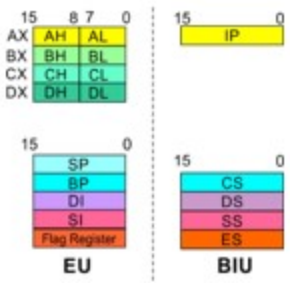
The effective address is just a 16-bit number written directly in the instruction.

Example:

```
MOV  BX, [1354H]
MOV  BL, [0400H]
```

The square brackets around the 1354_H denotes the contents of the memory location. When executed, this instruction will copy the contents of the memory location into BX register.

This addressing mode is called direct because the displacement of the operand from the segment base is specified directly in the instruction.



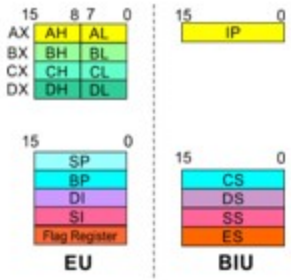
Addressing Modes

Register Indirect Addressing

In Register indirect addressing, name of the register which holds the effective address (EA) will be specified in the instruction.

Registers used to hold EA are any of the following registers:

BX, BP, DI and SI.



Content of the DS register is used for base address calculation.

Example:

MOV CX, [BX]

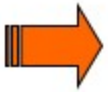
Arithmetic Instructions-ADD, ADC, INC

Mnemonic	Meaning	Format	Operation
ADD	Addition	ADD D,S	(S)+(D) → (D) carry → (CF)
ADC	Add with carry	ADC D,S	(S)+(D)+(CF) → (D) carry → (CF)
INC	Increment by one	INC D	(D)+1 → (D)

ADD instruction

ADD destination,source ;ADD the source operand to the destination

```
MOV   AL,25H
MOV   BL,34H
ADD   AL,BL
```



ADD dest, source ; dest=dest+source

You can do this,

```
MOV   DH,25H
ADD   DH,34H
```

INC Operand

Operand+1 → Operand

Examples:

Ex.1 **ADD AX,2**
ADC AX,2

Ex.2 **INC BX**

Arithmetic Instructions-ADD, ADC, INC

SUB Operand1, Operand2

Operand1-Operand2 ⟶ 0perand1

DEC Operand

Operand-1 ⟶ Operand

Examples:

```
MOV BL, 28H
MOV AL, 83H
SUB AL,BL       ; AL=5BH
```

Arithmetic Instructions-ADD, ADC, INC

Mnemonic	Meaning	Format	Operation
SUB	Subtract	SUB D,S	(D) - (S) → (D) Borrow → (CF)
SBB	Subtract with borrow	SBB D,S	(D) - (S) - (CF) → (D)
DEC	Decrement by one	DEC D	(D) - 1 → (D)
NEG	Negate	NEG D	
DAS	Decimal adjust for subtraction	DAS	Convert the result in AL to packed decimal format

Multiplication and Division

Mnemonic	Meaning	Format	Operation
MUL	Multiply (unsigned)	MUL S	(AL) · (S8) → (AX) (AX) · (S16) → (DX),(AX)
DIV	Division (unsigned)	DIV S	(1) Q((AX)/(S8)) → (AL) R((AX)/(S8)) → (AH) (2) Q((DX,AX)/(S16)) → (AX) R((DX,AX)/(S16)) → (DX) If Q is FF_{16} in case (1) or $FFFF_{16}$ in case (2), then type 0 interrupt occurs
IMUL	Integer multiply (signed)	IMUL S	(AL) · (S8) → (AX) (AX) · (S16) → (DX),(AX)
IDIV	Integer divide (signed)	IDIV S	(1) Q((AX)/(S8)) → (AL) R((AX)/(S8)) → (AH) (2) Q((DX,AX)/(S16)) → (AX) R((DX,AX)/(S16)) → (DX) If Q is positive and exceeds $7FFF_{16}$ or if Q is negative and becomes less than 8001_{16}, then type 0 interupt occurs

Multiplication and Division

Multiplication (MUL or IMUL)	Multiplicant	Operand (Multiplier)	Result
Byte * Byte	AL	Register or memory	AX
Word * Word	AX	Register or memory	DX :AX

Division (DIV or IDIV)	Dividend	Operand (Divisor)	Quotient : Remainder
Word / Byte	AX	Register or memory	AL : AH
Dword / Word	DX:AX	Register or memory	AX : DX

Multiplication and Division

Ex1: Assume that each instruction starts from these values:

AL = 85H, BL = 35H, AH = 0H

1. MUL BL → AL . BL = 85H * 35H = 1B89H → AX = 1B89H

2. IMUL BL → AL . BL = 2'S AL * BL = 2'S (85H) * 35H
   = 7BH * 35H = 1977H → 2's comp → E689H → AX.

- DIV BL → $\frac{AX}{BL} = \frac{0085H}{35H} = 02$ (85-02*35=1B) →

AH	AL
1B	02

4. IDIV BL → $\frac{AX}{BL} = \frac{0085H}{35H} =$

AH	AL
1B	02

Multiplication and Division

Ex2: **AL = F3H, BL = 91H, AH = 00H**

1. MUL BL → AL * BL = F3H * 91H = 89A3H → AX = 89A3H

2. IMUL BL → AL * BL = 2'S AL * 2'S BL = 2'S (F3H) * 2'S(91H) =
   0DH * 6FH = 05A3H → AX.

3. IDIV BL → $\frac{AX}{BL} = \frac{00F3H}{2'S(91H)} = \frac{00F3H}{6FH} = 2$ → (00F3 – 2*6F=15H)

AH	AL
15	02
R	Q

→ $\frac{POS}{NEG} = NEG$ → 2's(02) = FEH →

AH	AL
15	FE

4. DIV BL → $\frac{AX}{BL} = \frac{00F3H}{91H} = 01$ → (F3-1*91=62) →

AH	AL
62	01
R	Q

INSTRUCTION SET

Instruction Set

8086 supports 6 types of instructions.

1. **Data Transfer Instructions**
2. **Arithmetic Instructions**
3. **Logical Instructions**
4. **String manipulation Instructions**
5. **Process Control Instructions**
6. **Control Transfer Instructions**

Instruction Set

1. Data Transfer Instructions

Instructions that are used to transfer data/ address in to registers, memory locations and I/O ports.

Generally involve two operands: Source operand and Destination operand of the same size.

Source: Register or a memory location or an immediate data
Destination : Register or a memory location.

The size should be a either a byte or a word.

A 8-bit data can only be moved to 8-bit register/ memory and a 16-bit data can be moved to 16-bit register/ memory.

Instruction Set

1. Data Transfer Instructions

Mnemonics: **MOV, XCHG, PUSH, POP, IN, OUT ...**

MOV reg2/ mem, reg1/ mem	
MOV reg2, reg1 **MOV mem, reg1** **MOV reg2, mem**	**(reg2) ← (reg1)** **(mem) ← (reg1)** **(reg2) ← (mem)**
MOV reg/ mem, data	
MOV reg, data **MOV mem, data**	**(reg) ← data** **(mem) ← data**

XCHG reg2/ mem, reg1	
XCHG reg2, reg1 **XCHG mem, reg1**	**(reg2) ↔ (reg1)** **(mem) ↔ (reg1)**

Instruction Set

1. Data Transfer Instructions

Mnemonics: **MOV, XCHG, PUSH, POP, IN, OUT ...**

PUSH reg16/ mem	
PUSH reg16	(SP) ← (SP) – 2 MA_S = (SS) x 16_{10} + SP (MA_S ; MA_S + 1) ← (reg16)
PUSH mem	(SP) ← (SP) – 2 MA_S = (SS) x 16_{10} + SP (MA_S ; MA_S + 1) ← (mem)
POP reg16/ mem	
POP reg16	MA_S = (SS) x 16_{10} + SP (reg16) ← (MA_S ; MA_S + 1) (SP) ← (SP) + 2
POP mem	MA_S = (SS) x 16_{10} + SP (mem) ← (MA_S ; MA_S + 1) (SP) ← (SP) + 2

Instruction Set

1. Data Transfer Instructions

Mnemonics: **MOV, XCHG, PUSH, POP, IN, OUT ...**

IN A, [DX]	
IN AL, [DX]	**$PORT_{addr}$ = (DX)** **(AL) ← (PORT)**
IN AX, [DX]	**$PORT_{addr}$ = (DX)** **(AX) ← (PORT)**
IN A, addr8	
IN AL, addr8	**(AL) ← (addr8)**
IN AX, addr8	**(AX) ← (addr8)**

OUT [DX], A	
OUT [DX], AL	**$PORT_{addr}$ = (DX)** **(PORT) ← (AL)**
OUT [DX], AX	**$PORT_{addr}$ = (DX)** **(PORT) ← (AX)**
OUT addr8, A	
OUT addr8, AL	**(addr8) ← (AL)**
OUT addr8, AX	**(addr8) ← (AX)**

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…**

ADD reg2/ mem, reg1/mem	
ADC reg2, reg1	(reg2) ← (reg1) + (reg2)
ADC reg2, mem	(reg2) ← (reg2) + (mem)
ADC mem, reg1	(mem) ← (mem)+(reg1)
ADD reg/mem, data	
ADD reg, data	(reg) ← (reg)+ data
ADD mem, data	(mem) ← (mem)+data
ADD A, data	
ADD AL, data8	(AL) ← (AL) + data8
ADD AX, data16	(AX) ← (AX) +data16

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…**

ADC reg2/ mem, reg1/mem	
ADC reg2, reg1 ADC reg2, mem ADC mem, reg1	(reg2) ← (reg1) + (reg2)+CF (reg2) ← (reg2) + (mem)+CF (mem) ← (mem)+(reg1)+CF
ADC reg/mem, data	
ADC reg, data ADC mem, data	(reg) ← (reg)+ data+CF (mem) ← (mem)+data+CF
ADDC A, data	
ADD AL, data8 ADD AX, data16	(AL) ← (AL) + data8+CF (AX) ← (AX) +data16+CF

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…**

SUB reg2/ mem, reg1/mem	
SUB reg2, reg1 **SUB reg2, mem** **SUB mem, reg1**	**(reg2) ← (reg1) - (reg2)** **(reg2) ← (reg2) - (mem)** **(mem) ← (mem) - (reg1)**
SUB reg/mem, data	
SUB reg, data **SUB mem, data**	**(reg) ← (reg) - data** **(mem) ← (mem) - data**
SUB A, data	
SUB AL, data8 **SUB AX, data16**	**(AL) ← (AL) - data8** **(AX) ← (AX) - data16**

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…**

SBB reg2/ mem, reg1/mem	
SBB reg2, reg1 **SBB reg2, mem** **SBB mem, reg1**	**(reg2) ← (reg1) - (reg2) - CF** **(reg2) ← (reg2) - (mem)- CF** **(mem) ← (mem) - (reg1) –CF**
SBB reg/mem, data	
SBB reg, data **SBB mem, data**	**(reg) ← (reg) – data - CF** **(mem) ← (mem) - data - CF**
SBB A, data	
SBB AL, data8 **SBB AX, data16**	**(AL) ← (AL) - data8 - CF** **(AX) ← (AX) - data16 - CF**

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…**

INC reg/ mem	
INC reg8	**(reg8) ← (reg8) + 1**
INC reg16	**(reg16) ← (reg16) + 1**
INC mem	**(mem) ← (mem) + 1**
DEC reg/ mem	
DEC reg8	**(reg8) ← (reg8) - 1**
DEC reg16	**(reg16) ← (reg16) - 1**
DEC mem	**(mem) ← (mem) - 1**

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

MUL reg/ mem	
MUL reg	**For byte : (AX) ← (AL) x (reg8)** **For word : (DX)(AX) ← (AX) x (reg16)**
MUL mem	**For byte : (AX) ← (AL) x (mem8)** **For word : (DX)(AX) ← (AX) x (mem16)**
IMUL reg/ mem	
IMUL reg	**For byte : (AX) ← (AL) x (reg8)** **For word : (DX)(AX) ← (AX) x (reg16)**
IMUL mem	**For byte : (AX) ← (AX) x (mem8)** **For word : (DX)(AX) ← (AX) x (mem16)**

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

DIV reg/ mem	
DIV reg	**For 16-bit :- 8-bit :** **(AL) ← (AX) :- (reg8) Quotient** **(AH) ← (AX) MOD(reg8) Remainder** **For 32-bit :- 16-bit :** **(AX) ← (DX)(AX) :- (reg16) Quotient** **(DX) ← (DX)(AX) MOD(reg16) Remainder**
DIV mem	**For 16-bit :- 8-bit :** **(AL) ← (AX) :- (mem8) Quotient** **(AH) ← (AX) MOD(mem8) Remainder** **For 32-bit :- 16-bit :** **(AX) ← (DX)(AX) :- (mem16) Quotient** **(DX) ← (DX)(AX) MOD(mem16) Remainder**

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…**

IDIV reg/ mem	
IDIV reg	For 16-bit :- 8-bit : (AL) ← (AX) :- (reg8) Quotient (AH) ← (AX) MOD(reg8) Remainder For 32-bit :- 16-bit : (AX) ← (DX)(AX) :- (reg16) Quotient (DX) ← (DX)(AX) MOD(reg16) Remainder
IDIV mem	For 16-bit :- 8-bit : (AL) ← (AX) :- (mem8) Quotient (AH) ← (AX) MOD(mem8) Remainder For 32-bit :- 16-bit : (AX) ← (DX)(AX) :- (mem16) Quotient (DX) ← (DX)(AX) MOD(mem16) Remainder

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

CMP reg2/mem, reg1/ mem	

CMP reg2, reg1	Modify flags ← (reg2) – (reg1)
	
	If (reg2) > (reg1) then CF=0, ZF=0, SF=0
	If (reg2) < (reg1) then CF=1, ZF=0, SF=1
	If (reg2) = (reg1) then CF=0, ZF=1, SF=0

CMP reg2, mem	Modify flags ← (reg2) – (mem)
	
	If (reg2) > (mem) then CF=0, ZF=0, SF=0
	If (reg2) < (mem) then CF=1, ZF=0, SF=1
	If (reg2) = (mem) then CF=0, ZF=1, SF=0

CMP mem, reg1	Modify flags ← (mem) – (reg1)
	
	If (mem) > (reg1) then CF=0, ZF=0, SF=0
	If (mem) < (reg1) then CF=1, ZF=0, SF=1
	If (mem) = (reg1) then CF=0, ZF=1, SF=0

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

CMP reg/mem, data	

CMP reg, data	Modify flags ← (reg) – (data)
	
	If (reg) > data then CF=0, ZF=0, SF=0
	If (reg) < data then CF=1, ZF=0, SF=1
	If (reg) = data then CF=0, ZF=1, SF=0

CMP mem, data	Modify flags ← (mem) – (mem)
	
	If (mem) > data then CF=0, ZF=0, SF=0
	If (mem) < data then CF=1, ZF=0, SF=1
	If (mem) = data then CF=0, ZF=1, SF=0

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

CMP A, data	
CMP AL, data8	**Modify flags ← (AL) – data8** **If (AL) > data8 then CF=0, ZF=0, SF=0** **If (AL) < data8 then CF=1, ZF=0, SF=1** **If (AL) = data8 then CF=0, ZF=1, SF=0**
CMP AX, data16	**Modify flags ← (AX) – data16** **If (AX) > data16 then CF=0, ZF=0, SF=0** **If (mem) < data16 then CF=1, ZF=0, SF=1** **If (mem) = data16 then CF=0, ZF=1, SF=0**

Instruction Set

3. Logical Instructions

Mnemonics: AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...

AND A, data AND AL, data8	(AL) ← (AL) & data8
AND AX, data16	(AX) ← (AX) & data16

AND reg/mem, data AND reg, data	(reg) ← (reg) & data
AND mem, data	(mem) ← (mem) & data

Instruction Set

3. Logical Instructions

Mnemonics: AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...

OR reg2/mem, reg1/mem OR reg2, reg1	(reg2) ← (reg2) | (reg1)
OR reg2, mem	(reg2) ← (reg2) | (mem)
OR mem, reg1	(mem) ← (mem) | (reg1)

OR reg/mem, data OR reg, data	(reg) ← (reg) | data
OR mem, data	(mem) ← (mem) | data

OR A, data OR AL, data8	(AL) ← (AL) | data8
OR AX, data16	(AX) ← (AX) | data16

Instruction Set

3. Logical Instructions

Mnemonics: AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...

XOR reg2/mem, reg1/mem	
XOR reg2, reg1	(reg2) ← (reg2) ^ (reg1)
XOR reg2, mem	(reg2) ← (reg2) ^ (mem)
XOR mem, reg1	(mem) ← (mem) ^ (reg1)

XOR reg/mem, data	
XOR reg, data	(reg) ← (reg) ^ data
XOR mem, data	(mem) ← (mem) ^ data

XOR A, data	
XOR AL, data8	(AL) ← (AL) ^ data8
XOR AX, data16	(AX) ← (AX) ^ data16

Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

TEST reg2/mem, reg1/mem	
TEST reg2, reg1	Modify flags ← (reg2) & (reg1)
TEST reg2, mem	Modify flags ← (reg2) & (mem)
TEST mem, reg1	Modify flags ← (mem) & (reg1)

TEST reg/mem, data	
TEST reg, data	Modify flags ← (reg) & data
TEST mem, data	Modify flags ← (mem) & data

TEST A, data	
TEST AL, data8	Modify flags ← (AL) & data8
TEST AX, data16	Modify flags ← (AX) & data16

Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

SHR reg/mem

SHR reg

i) SHR reg, 1

ii) SHR reg, CL

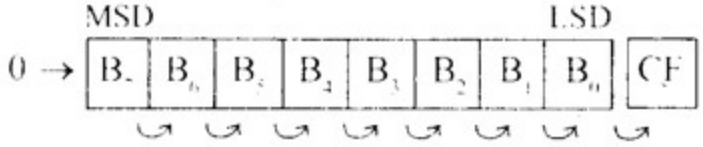
SHR mem

i) SHR mem, 1

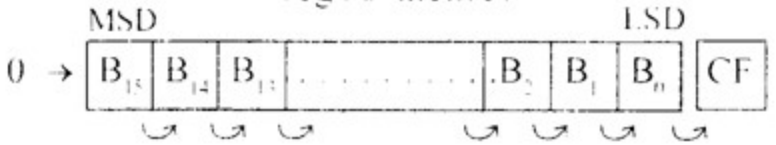
ii) SHR mem, CL

$$CF \leftarrow B_{LSD} : B_n \leftarrow B_{n+1} \; ; \; B_{MSD} \leftarrow 0$$

reg 8 / mem 8



reg 16 / mem 16



Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

SHL reg/mem or SAL reg/mem

SHL reg or SAL reg

i) SHL reg, 1 or SAL reg, 1

ii) SHL reg, CL or SAL reg, CL

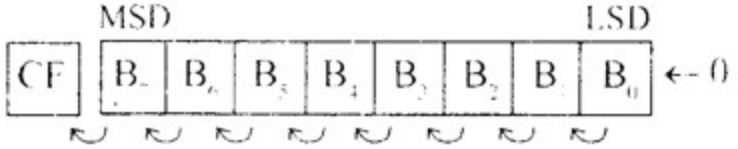
SHL mem or SAL mem

i) SHL mem, 1 or SAL mem, 1

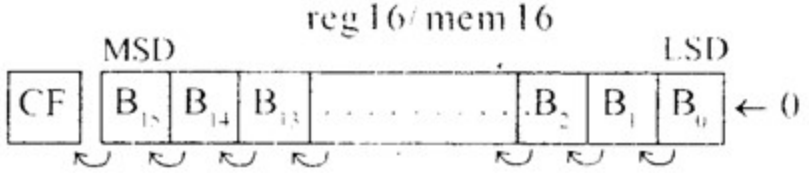
ii) SHL mem, CL or SAL mem, CL

$CF \leftarrow B_{MSD} ; B_{n-1} \leftarrow B_n ; B_{LSD} \leftarrow 0$

reg 8 / mem 8



reg 16 / mem 16



Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

RCR reg/mem

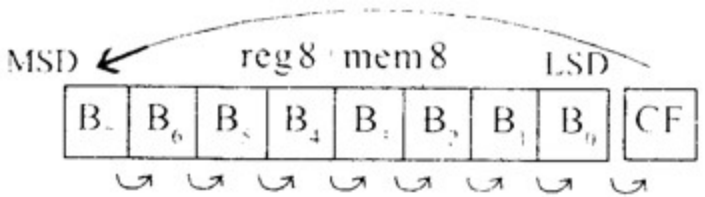
RCR reg

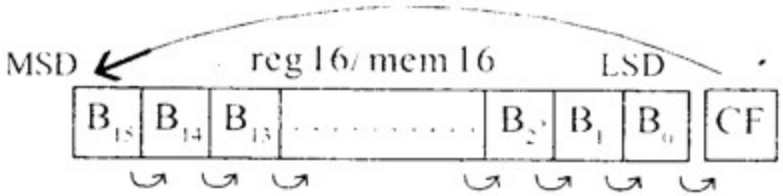
i) RCR reg, 1
ii) RCR reg, CL

RCR mem

i) RCR mem, 1
ii) RCR mem, CL

$B_n \leftarrow B_{n-1}$; $B_{MSD} \leftarrow CF$; $CF \leftarrow B_{LSD}$

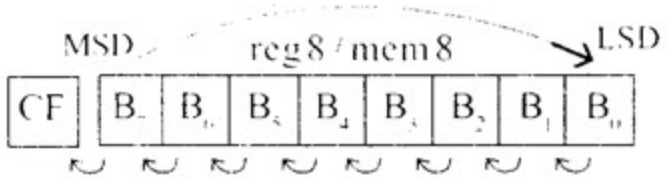
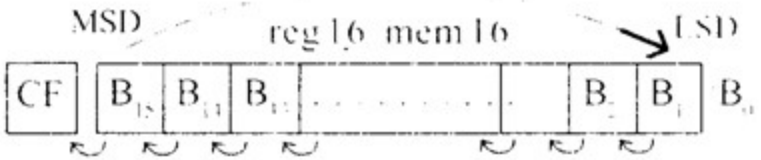




Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

ROL reg/mem ROL reg i) ROL reg, 1 ii) ROL reg, CL ROL mem i) ROL mem, 1 ii) ROL mem, CL	$B_{n-1} \leftarrow B_n : CF \leftarrow B_{MSD} ; B_{LSD} \leftarrow B_{MSD}$  

Instruction Set

4. String Manipulation Instructions

- String : Sequence of bytes or words
- 8086 instruction set includes instruction for string movement, comparison, scan, load and store.
- REP instruction prefix : used to repeat execution of string instructions
- String instructions end with S or SB or SW.
  S represents string, SB string byte and SW string word.
- Offset or effective address of the source operand is stored in SI register and that of the destination operand is stored in DI register.
- Depending on the status of DF, SI and DI registers are automatically updated.
- DF = 0 ⇒ SI and DI are incremented by 1 for byte and 2 for word.
- DF = 1 ⇒ SI and DI are decremented by 1 for byte and 2 for word.

Instruction Set

4. String Manipulation Instructions

Mnemonics: **REP, MOVS, CMPS, SCAS, LODS, STOS**

REP **REPZ/ REPE** **(Repeat CMPS or SCAS until ZF = 0)**	**While CX ≠ 0 and ZF = 1, repeat execution of string instruction and (CX) ← (CX) – 1**
REPNZ/ REPNE **(Repeat CMPS or SCAS until ZF = 1)**	**While CX ≠ 0 and ZF = 0, repeat execution of string instruction and (CX) ← (CX) - 1**

Instruction Set

4. String Manipulation Instructions

Mnemonics: **REP, MOVS, CMPS, SCAS, LODS, STOS**

MOVS	
MOVSB	**MA = (DS) x 16_{10} + (SI)** **MA_E = (ES) x 16_{10} + (DI)** **(MA_E) ← (MA)** **If DF = 0, then (DI) ← (DI) + 1; (SI) ← (SI) + 1** **If DF = 1, then (DI) ← (DI) - 1; (SI) ← (SI) - 1**
MOVSW	**MA = (DS) x 16_{10} + (SI)** **MA_E = (ES) x 16_{10} + (DI)** **(MA_E ; MA_E + 1) ← (MA; MA + 1)** **If DF = 0, then (DI) ← (DI) + 2; (SI) ← (SI) + 2** **If DF = 1, then (DI) ← (DI) - 2; (SI) ← (SI) - 2**

Instruction Set

4. String Manipulation Instructions

Mnemonics: **REP, MOVS, CMPS, SCAS, LODS, STOS**

Compare two string byte or string word

CMPS	
CMPSB	**MA = (DS) x 16_{10} + (SI)** **MA_E = (ES) x 16_{10} + (DI)** **Modify flags ← (MA) - (MA_E)** **If (MA) > (MA_E), then CF = 0; ZF = 0; SF = 0** **If (MA) < (MA_E), then CF = 1; ZF = 0; SF = 1** **If (MA) = (MA_E), then CF = 0; ZF = 1; SF = 0**
CMPSW	**<u>For byte operation</u>** **If DF = 0, then (DI) ← (DI) + 1; (SI) ← (SI) + 1** **If DF = 1, then (DI) ← (DI) - 1; (SI) ← (SI) - 1** **<u>For word operation</u>** **If DF = 0, then (DI) ← (DI) + 2; (SI) ← (SI) + 2** **If DF = 1, then (DI) ← (DI) - 2; (SI) ← (SI) - 2**

Instruction Set

4. String Manipulation Instructions

Mnemonics: **REP, MOVS, CMPS, SCAS, LODS, STOS**

Scan (compare) a string byte or word with accumulator

SCAS	

SCASB	MA_E = (ES) x 16_{10} + (DI)
	Modify flags ← (AL) - (MA_E)
	
	If (AL) > (MA_E), then CF = 0; ZF = 0; SF = 0
	If (AL) < (MA_E), then CF = 1; ZF = 0; SF = 1
	If (AL) = (MA_E), then CF = 0; ZF = 1; SF = 0
	
	If DF = 0, then (DI) ← (DI) + 1
	If DF = 1, then (DI) ← (DI) – 1

SCASW	MA_E = (ES) x 16_{10} + (DI)
	Modify flags ← (AL) - (MA_E)
	
	If (AX) > (MA_E ; MA_E + 1), then CF = 0; ZF = 0; SF = 0
	If (AX) < (MA_E ; MA_E + 1), then CF = 1; ZF = 0; SF = 1
	If (AX) = (MA_E ; MA_E + 1), then CF = 0; ZF = 1; SF = 0
	
	If DF = 0, then (DI) ← (DI) + 2
	If DF = 1, then (DI) ← (DI) – 2

Instruction Set

4. String Manipulation Instructions

Mnemonics: **REP, MOVS, CMPS, SCAS, LODS, STOS**

Load string byte in to AL or string word in to AX

LODS	
LODSB	**MA = (DS) x 16_{10} + (SI)** **(AL) ← (MA)** **If DF = 0, then (SI) ← (SI) + 1** **If DF = 1, then (SI) ← (SI) – 1**
LODSW	**MA = (DS) x 16_{10} + (SI)** **(AX) ← (MA ; MA + 1)** **If DF = 0, then (SI) ← (SI) + 2** **If DF = 1, then (SI) ← (SI) – 2**

Instruction Set

4. String Manipulation Instructions

Mnemonics: **REP, MOVS, CMPS, SCAS, LODS, STOS**

Store byte from AL or word from AX in to string

STOS	
STOSB	$MA_E = (ES) \times 16_{10} + (DI)$ $(MA_E) \leftarrow (AL)$ If DF = 0, then (DI) ← (DI) + 1 If DF = 1, then (DI) ← (DI) – 1
STOSW	$MA_E = (ES) \times 16_{10} + (DI)$ $(MA_E ; MA_E + 1) \leftarrow (AX)$ If DF = 0, then (DI) ← (DI) + 2 If DF = 1, then (DI) ← (DI) – 2

Instruction Set

5. Processor Control Instructions

Mnemonics	Explanation
STC	Set CF ← 1
CLC	Clear CF ← 0
CMC	Complement carry CF ← CF/
STD	Set direction flag DF ← 1
CLD	Clear direction flag DF ← 0
STI	Set interrupt enable flag IF ← 1
CLI	Clear interrupt enable flag IF ← 0
NOP	No operation
HLT	Halt after interrupt is set
WAIT	Wait for TEST pin active
ESC opcode mem/ reg	Used to pass instruction to a coprocessor which shares the address and data bus with the 8086
LOCK	Lock bus during next instruction

Instruction Set

6. Control Transfer Instructions

- Transfer the control to a specific destination or target instruction
- Do not affect flags

❑ 8086 Unconditional transfers

Mnemonics	Explanation
CALL reg/ mem/ disp16	Call subroutine
RET	Return from subroutine
JMP reg/ mem/ disp8/ disp16	Unconditional jump

Instruction Set

6. Control Transfer Instructions

- **8086 signed conditional branch instructions**
- **8086 unsigned conditional branch instructions**

- **Checks flags**
- **If conditions are true, the program control is transferred to the new memory location in the same segment by modifying the content of IP**

Instruction Set

6. Control Transfer Instructions

- **8086 signed conditional branch instructions**

Name	Alternate name
JE disp8 Jump if equal	JZ disp8 Jump if result is 0
JNE disp8 Jump if not equal	JNZ disp8 Jump if not zero
JG disp8 Jump if greater	JNLE disp8 Jump if not less or equal
JGE disp8 Jump if greater than or equal	JNL disp8 Jump if not less
JL disp8 Jump if less than	JNGE disp8 Jump if not greater than or equal
JLE disp8 Jump if less than or equal	JNG disp8 Jump if not greater

- **8086 unsigned conditional branch instructions**

Name	Alternate name
JE disp8 Jump if equal	JZ disp8 Jump if result is 0
JNE disp8 Jump if not equal	JNZ disp8 Jump if not zero
JA disp8 Jump if above	JNBE disp8 Jump if not below or equal
JAE disp8 Jump if above or equal	JNB disp8 Jump if not below
JB disp8 Jump if below	JNAE disp8 Jump if not above or equal
JBE disp8 Jump if below or equal	JNA disp8 Jump if not above

Instruction Set

6. Control Transfer Instructions

- **8086 conditional branch instructions affecting individual flags**

Mnemonics	Explanation
JC disp8	Jump if CF = 1
JNC disp8	Jump if CF = 0
JP disp8	Jump if PF = 1
JNP disp8	Jump if PF = 0
JO disp8	Jump if OF = 1
JNO disp8	Jump if OF = 0
JS disp8	Jump if SF = 1
JNS disp8	Jump if SF = 0
JZ disp8	Jump if result is zero, i.e, Z = 1
JNZ disp8	Jump if result is not zero, i.e, Z = 1

Assembler directives

Assemble Directives

- Instructions to the Assembler regarding the program being executed.

- Control the generation of machine codes and organization of the program; but no machine codes are generated for assembler directives.

- Also called 'pseudo instructions'

- Used to :
  - specify the start and end of a program
  - attach value to variables
  - allocate storage locations to input/ output data
  - define start and end of segments, procedures, macros etc..

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- **Define Byte**
- **Define a byte type (8-bit) variable**
- **Reserves specific amount of memory locations to each variable**
- **Range : 00_H – FF_H for unsigned value;**
  00_H – $7F_H$ for positive value and
  80_H – FF_H for negative value
- **General form : variable DB value/ values**

Example:

```
LIST DB 7FH, 42H, 35H
```

Three consecutive memory locations are reserved for the variable LIST and each data specified in the instruction are stored as initial value in the reserved memory location

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- Define Word
- Define a word type (16-bit) variable
- Reserves two consecutive memory locations to each variable
- Range : 0000_H – $FFFF_H$ for unsigned value;
  0000_H – $7FFF_H$ for positive value and
  8000_H – $FFFF_H$ for negative value
- General form : variable DW value/ values

Example:

```
ALIST DW 6512H, 0F251H, 0CDE2H
```

Six consecutive memory locations are reserved for the variable ALIST and each 16-bit data specified in the instruction is stored in two consecutive memory location.

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- **SEGMENT : Used to indicate the beginning of a code/ data/ stack segment**
- **ENDS : Used to indicate the end of a code/ data/ stack segment**
- **General form:**

```
Segnam SEGMENT
   ...
   ...
   ...
   ...
   ...
   ...
Segnam ENDS
```

Program code
or
Data Defining Statements

User defined name of the segment

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- **Informs the assembler the name of the program/ data segment that should be used for a specific segment.**
- **General form:**

ASSUME segreg : segnam, .. , segreg : segnam

Segment Register	User defined name of the segment

Example:

ASSUME CS: ACODE, DS:ADATA	Tells the compiler that the instructions of the program are stored in the segment ACODE and data are stored in the segment ADATA

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- **ORG** (Origin) is used to assign the starting address (Effective address) for a program/ data segment
- **END** is used to terminate a program; statements after END will be ignored
- **EVEN** : Informs the assembler to store program/ data segment starting from an even address
- **EQU** (Equate) is used to attach a value to a variable

Examples:

ORG 1000H	Informs the assembler that the statements following ORG 1000H should be stored in memory starting with effective address 1000_H
LOOP EQU 10FEH	Value of variable LOOP is $10FE_H$
_SDATA SEGMENT ORG 1200H A DB 4CH EVEN B DW 1052H _SDATA ENDS	In this data segment, effective address of memory location assigned to A will be 1200_H and that of B will be 1202_H and 1203_H.

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

- **PROC** Indicates the beginning of a procedure
- **ENDP** End of procedure
- **FAR** Intersegment call
- **NEAR** Intrasegment call
- **General form**

```
procname PROC[NEAR/ FAR]
        ...
        ...
        ...
        RET
procname ENDP
```

Program statements of the procedure

Last statement of the procedure

User defined name of the procedure

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

Examples:

ADD64 PROC NEAR ... ... RET ADD64 ENDP	The subroutine/ procedure named ADD64 is declared as NEAR and so the assembler will code the CALL and RET instructions involved in this procedure as near call and return
CONVERT PROC FAR ... ... RET CONVERT ENDP	The subroutine/ procedure named CONVERT is declared as FAR and so the assembler will code the CALL and RET instructions involved in this procedure as far call and return

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

- **Reserves one memory location for 8-bit signed displacement in jump instructions**

Example:

JMP SHORT AHEAD	The directive will reserve one memory location for 8-bit displacement named AHEAD

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

- **MACRO** Indicate the beginning of a macro
- **ENDM** End of a macro
- General form:

```
macroname MACRO[Arg1, Arg2 ...]
          ...
          ...
          ...
macroname ENDM
```

Program statements in the macro

User defined name of the macro

Interfacing I/O and peripheral devices

I/O devices

⇒ For communication between microprocessor and outside world

⇒ Keyboards, CRT displays, Printers, Compact Discs etc.

⇒

⇒ Data transfer types

- **Programmed I/ O**
  Data transfer is accomplished through an I/O port controlled by software
  - Memory mapped
  - I/O mapped
- **Interrupt driven I/ O**
  I/O device interrupts the processor and initiate data transfer
- **Direct memory access**
  Data transfer is achieved by bypassing the microprocessor