8086 Microprocessor’s

1. Addressing Modes
2. Instruction Set
3. Directives
4. Sample Programs

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

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.

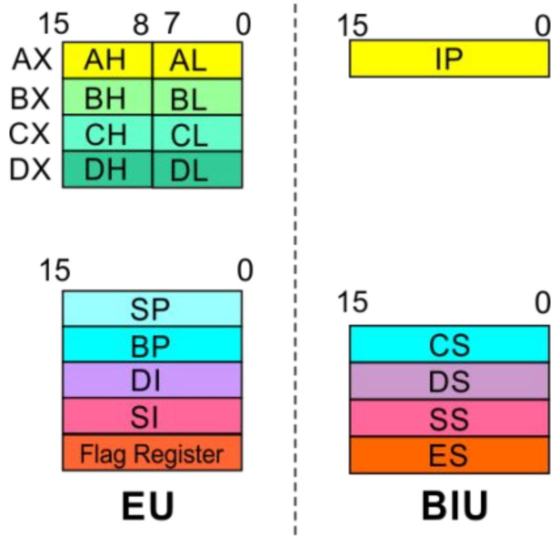
Example:

MOV CL, DH

MOV AX,BX

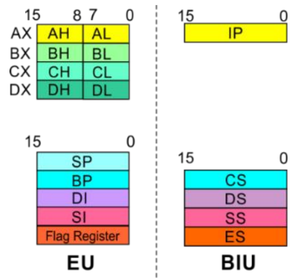
The content of 8-bit register DH is moved to another 8-bit register CL

(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

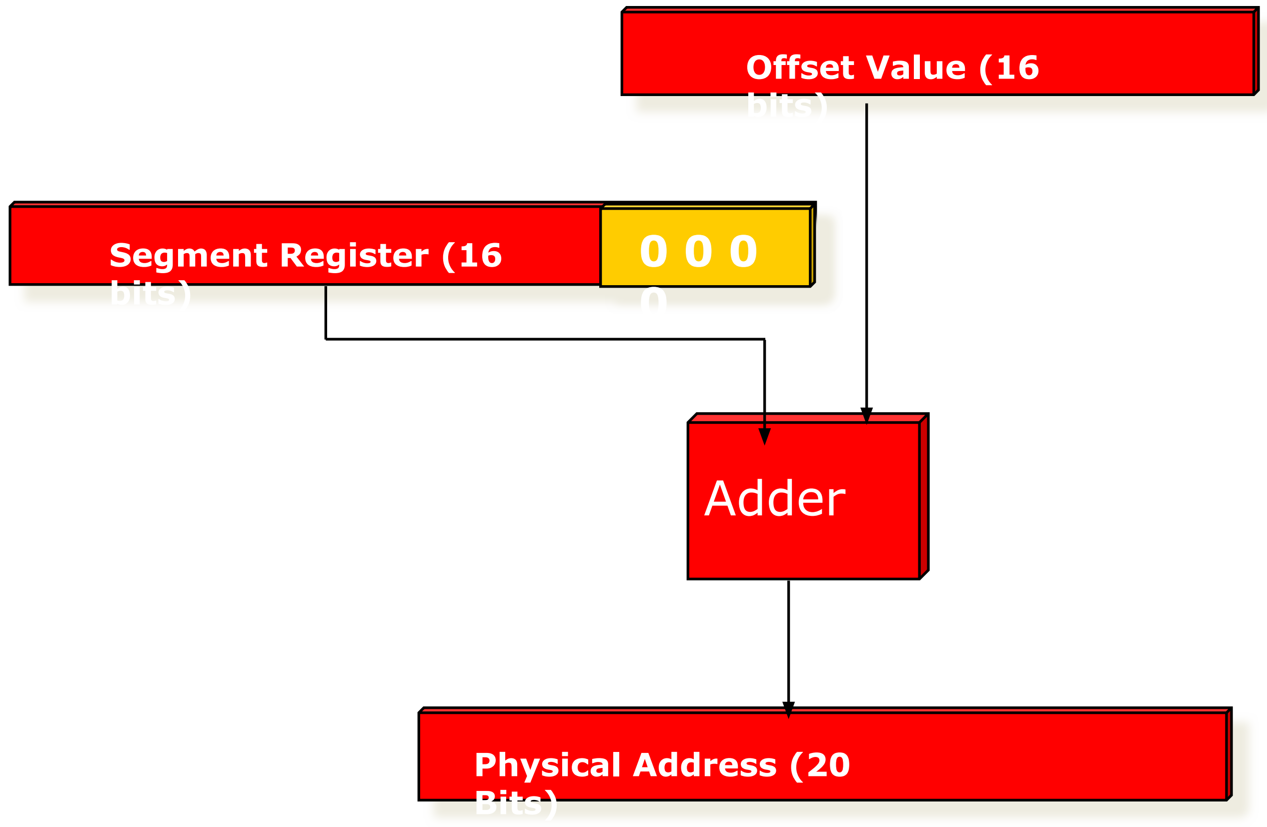
$(DL) \leftarrow 08_H$

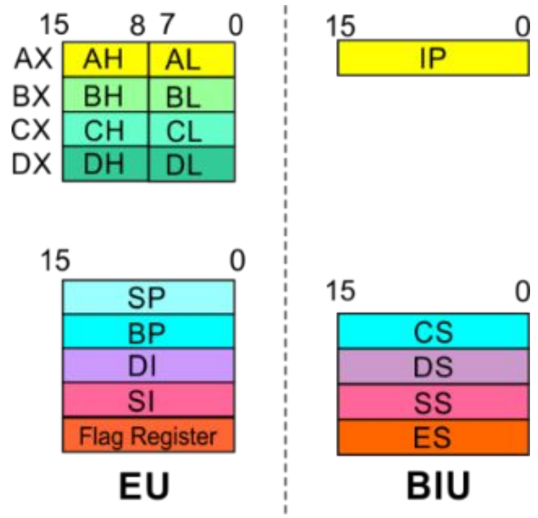
MOV AX, 0A9FH

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

$(AX) \leftarrow 0A9F_H$

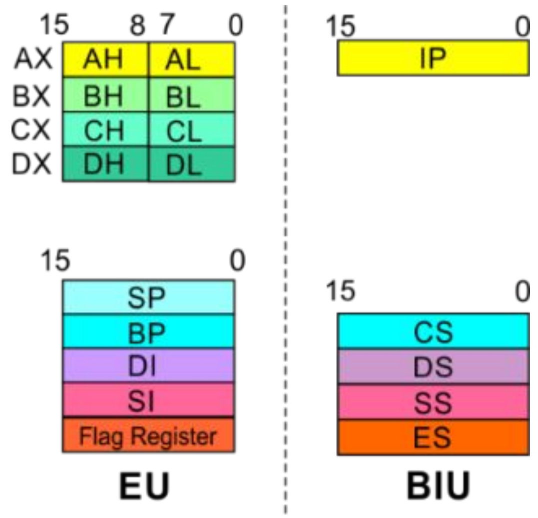
Addressing Modes : Memory Access





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

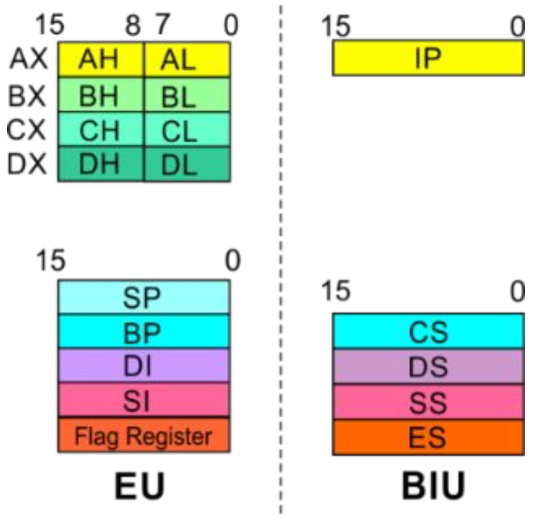
89AB : F012 → 89AB → 89AB0 (Paragraph to byte → 89AB x 10 = 89AB0)
F012 → 0F012 (Offset is already in byte unit)
+ -------
98AC2 (The absolute address)

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:**

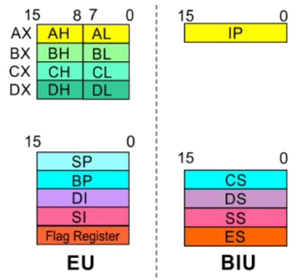
[BX + SI] [BX + DI] [BP + SI] [BP + DI]	[SI] [DI] d16 (variable offset only) [BX]	[BX + SI + d8] [BX + DI + d8] [BP + SI + d8] [BP + DI + d8]
[SI + d8] [DI + d8] [BP + d8] [BX + d8]	[BX + SI + d16] [BX + DI + d16] [BP + SI + d16] [BP + DI + d16]	[SI + d16] [DI + d16] [BP + d16] [BX + d16]

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



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.

DS=5000H BX=4000H

[54000]□BL

1354

[51354]□BL

[51355]□BH

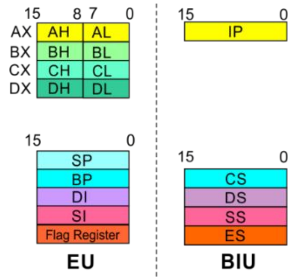
Example:

```
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



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:

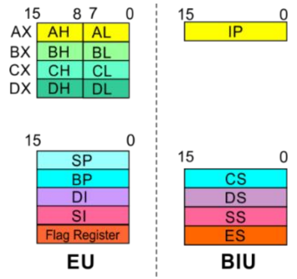
EA = (BX)
BA = (DS) x 16_{10}
MA = BA + EA

(CX) ← (MA) or,

(CL) ← (MA)
(CH) ← (MA +1)

DS=4000 BX=2500
40000
2500
42500

Addressing Modes



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:DS=5000 BX=0100

MOV AX, [BX + 08H] 1111 1111 1000 1000 1111 1111 1000 1000

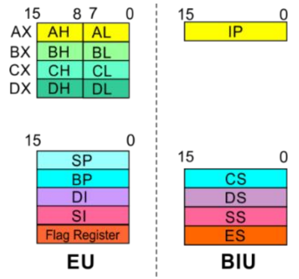
Operations:

$0008_H \leftarrow 08_H$ (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



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:0C5 FFC5 1100 0101

MOV CX, [SI + 52H] **SI=1200 DS=5000**

```
 50000    1200
           0052
```

Operations:

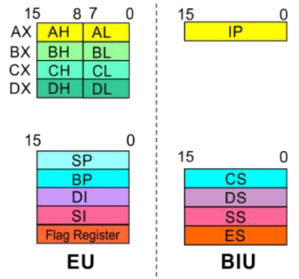
$FFA2_H \leftarrow A2_H$ (Sign extended)

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

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

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

Addressing Modes



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

MOV CX,0005H

Example: REP MOVS BYTE

Operations:

Calculation of source memory location:

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

Calculation of destination memory location:

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

$(MAE) \leftarrow (MA)$

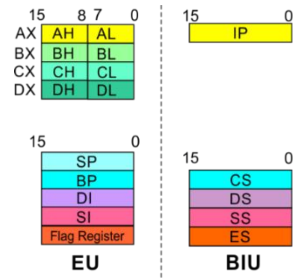
If DF = 1, then $(SI) \leftarrow (SI) - 1$ and $(DI) = (DI) - 1$

If DF = 0, then $(SI) \leftarrow (SI) + 1$ and $(DI) = (DI) + 1$

Addressing Modes

Group III : Addressing modes for I/O ports

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: $PORT_{addr} = 09_H$

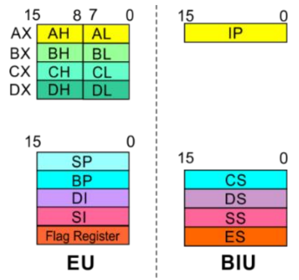
$(AL) \leftarrow (PORT)$

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

[illegible]

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.

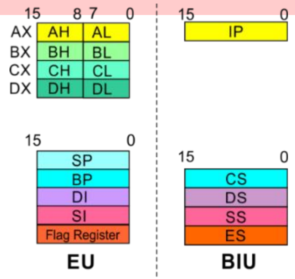
Addressing Modes

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.



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	$(SP) \leftarrow (SP) - 2$ $MA_S = (SS) \times 16_{10} + SP$ $(MA_S ; MA_S + 1) \leftarrow (reg16)$
PUSH mem	$(SP) \leftarrow (SP) - 2$ $MA_S = (SS) \times 16_{10} + SP$ $(MA_S ; MA_S + 1) \leftarrow (mem)$
POP reg16/ mem	
POP reg16	$MA_S = (SS) \times 16_{10} + SP$ $(reg16) \leftarrow (MA_S ; MA_S + 1)$ $(SP) \leftarrow (SP) + 2$
POP mem	$MA_S = (SS) \times 16_{10} + SP$ $(mem) \leftarrow (MA_S ; MA_S + 1)$ $(SP) \leftarrow (SP) + 2$

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)

LEA – LEA Register, Source

This instruction determines the offset of the variable or memory location named as the source and puts this offset in the indicated 16-bit register. LEA does not affect any flag.

 LEA BX, PRICES Load BX with offset of PRICE in DS

LDS – LDS Register, Memory address of the first word

This instruction loads new values into the specified register and into the DS register from four successive memory locations. The word from two memory locations is copied into the specified register and the word from the next two memory locations is copied into the DS registers. LDS does not affect any flag.

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

LES – LES Register, Memory address of the first word

This instruction loads new values into the specified register and into the ES register from four successive memory locations. The word from the first two memory locations is copied into the specified register, and the word from the next two memory locations is copied into the ES register. LES does not affect any flag.

- LES DI, [BX] Copy content of memory at offset [BX] and offset [BX] + 1 in DS to DI register. Copy content of memory at offset [BX] + 2 and [BX] + 3 to ES register.

LAHF : Load AH from Lower Byte of Flag

SAHF : Store AH to Lower Byte of Flag Register

PUSHF : Push Flags to Stack

POPF : Pop Flags from Stack

Instruction Set

2. Arithmetic Instructions

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

ADD reg2/ mem, reg1/mem **ADC reg2, reg1** **ADC reg2, mem** **ADC mem, reg1**	 (reg2) ← (reg1) + (reg2) (reg2) ← (reg2) + (mem) (mem) ← (mem)+(reg1)
ADD reg/mem, data **ADD reg, data** **ADD mem, data**	 (reg) ← (reg)+ data (mem) ← (mem)+data
ADD A, data **ADD AL, data8** **ADD AX, data16**	 (AL) ← (AL) + data8 (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	<u>For 16-bit :- 8-bit</u> : (AL) ← (AX) :- (reg8) Quotient (AH) ← (AX) MOD(reg8) Remainder <u>For 32-bit :- 16-bit</u> : (AX) ← (DX)(AX) :- (reg16) Quotient (DX) ← (DX)(AX) MOD(reg16) Remainder
IDIV mem	<u>For 16-bit :- 8-bit</u> : (AL) ← (AX) :- (mem8) Quotient (AH) ← (AX) MOD(mem8) Remainder <u>For 32-bit :- 16-bit</u> : (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**

ASCII code related Instructions

AAA : ASCII Adjust After Addition

Let AL = 0011 0101 (ASCII 5), and BL = 0011 1001 (ASCII 9)
    ADD AL, BL AL = 0110 1110 (6EH, which is incorrect BCD)
    AAA AL = 0000 0100 (unpacked BCD 4)
    CF = 1 indicates answer is 14 decimal.

The AAA instruction works only on the AL register. The AAA instruction updates AF and CF; but OF, PF, SF and ZF are left undefined.

AAS : ASCII Adjust AL After Subtraction

Let AL = 00110101 (35H or ASCII 5), and BL = 00111001 (39H or ASCII 9)
    SUB AL, BL AL = 11111100 (– 4 in 2’s complement form), and CF = 1
    AAS AL = 00000100 (BCD 06), and CF = 1 (borrow required)

The AAS instruction works only on the AL register. It updates ZF and CF; but OF, PF, SF, AF are left undefined.

ASCII code related Instructions

AAM : ASCII Adjust After Multiplication

Let AL = 00000101 (unpacked BCD 5), and BH = 00001001 (unpacked BCD 9)
MUL BH AL x BH: AX = 00000000 00101101 = 002DH
AAM AX = 00000100 00000101 = 0405H (unpacked BCD for 45)

AAD : ASCII Adjust Before Division

Let AX = 0607 (unpacked BCD for 67 decimal), and CH = 09H
AAD AX = 0043 (43H = 67 decimal)
DIV CH AL = 07; AH = 04; Flags undefined after DIV
If an attempt is made to divide by 0, the 8086 will generate a type 0 interrupt.

DAA : Decimal Adjust Accumulator

DAS : Decimal Adjust after Subtraction

CBW (CONVERT SIGNED BYTE TO SIGNED WORD)

This instruction copies the sign bit of the byte in AL to all the bits in AH. AH is then said to be the sign extension of AL. CBW does not affect any flag.

- Let AX = 00000000 10011011 (–155 decimal)

CBW Convert signed byte in AL to signed word in AX
AX = 11111111 10011011 (–155 decimal)

CWD (CONVERT SIGNED WORD TO SIGNED DOUBLE WORD)

This instruction copies the sign bit of a word in AX to all the bits of the DX register. In other words, it extends the sign of AX into all of DX. CWD affects no flags.

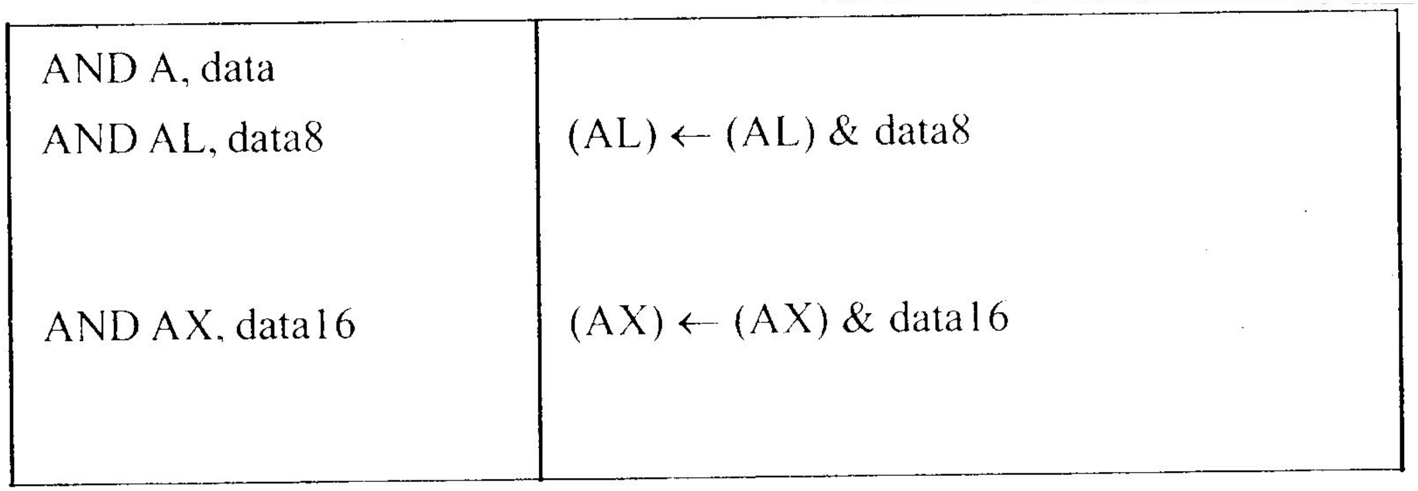
- Let DX = 00000000 00000000, and AX = 11110000 11000111 (–3897 decimal)

CWD Convert signed word in AX to signed double word in DX:AX
DX = 11111111 11111111
AX = 11110000 11000111 (–3897 decimal)

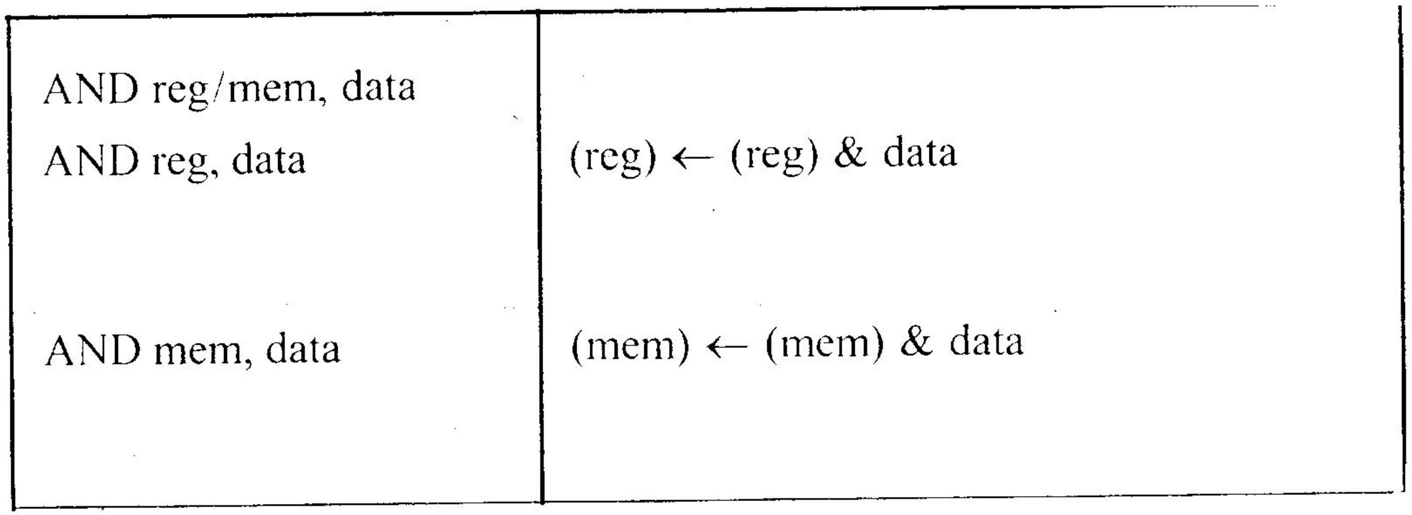
Instruction Set

3. Logical Instructions

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



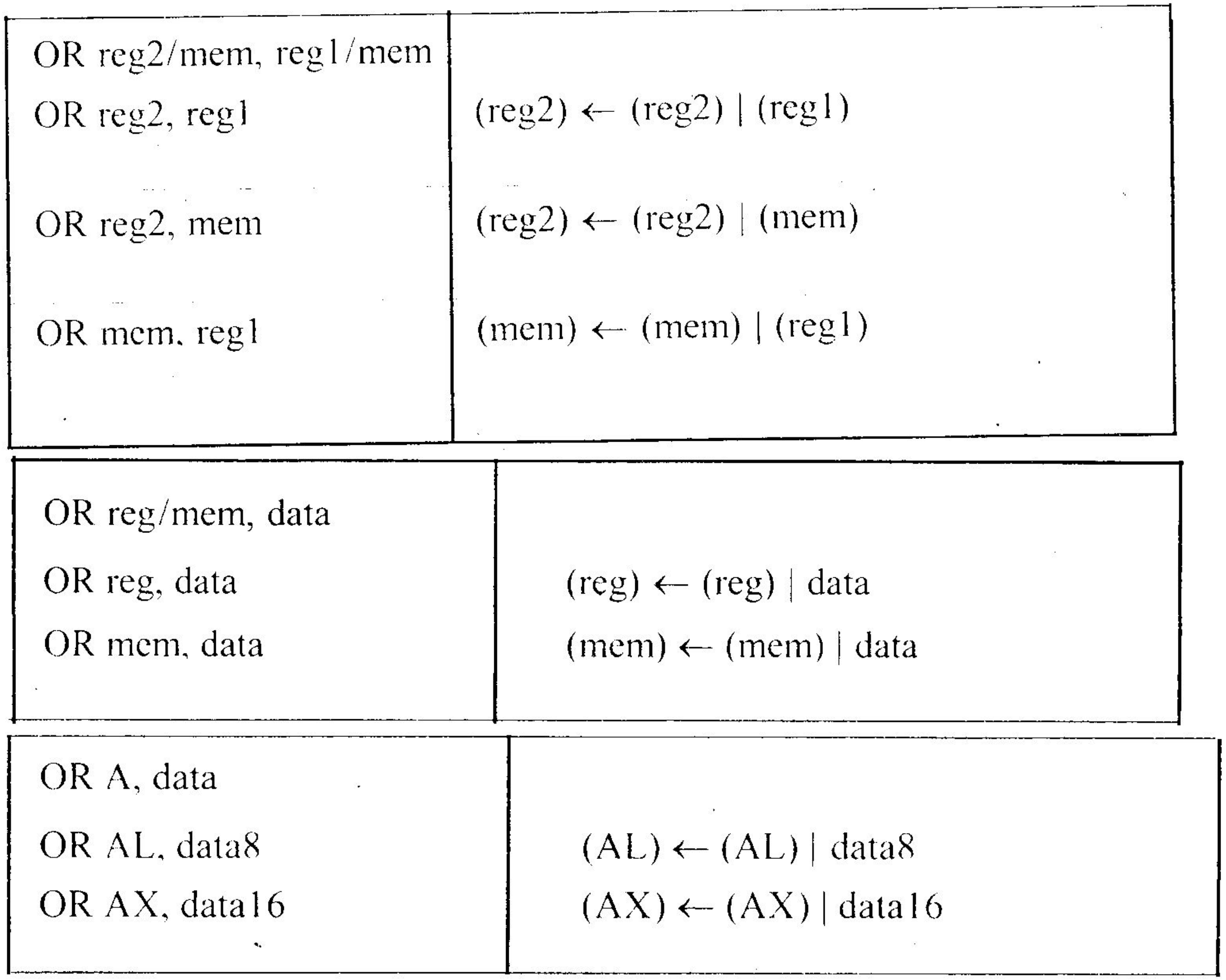
AND A, data AND AL, data8	(AL) ← (AL) & data8
AND AX, data16	(AX) ← (AX) & data16



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

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

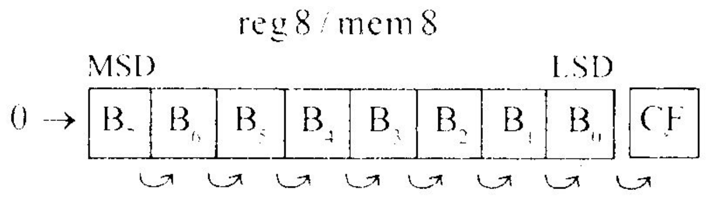
ii) SHR reg, CL

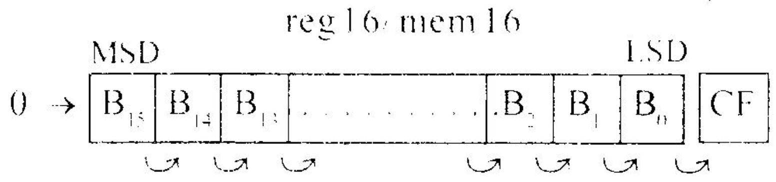
SHR mem

i) SHR mem, 1

ii) SHR mem, CL

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





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

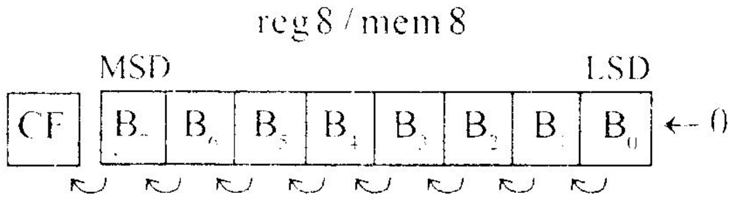
ii) SHL reg, CL or SAL reg, CL

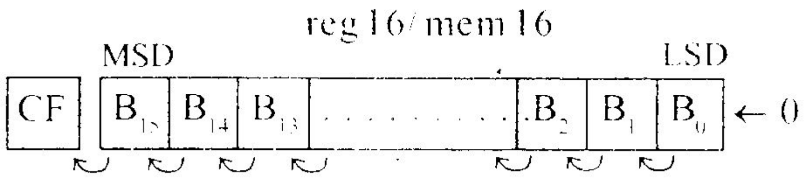
SHL mem or SAL mem

i) SHL mem, 1 or SAL mem, 1

ii) SHL mem, CL or SAL mem, CL

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

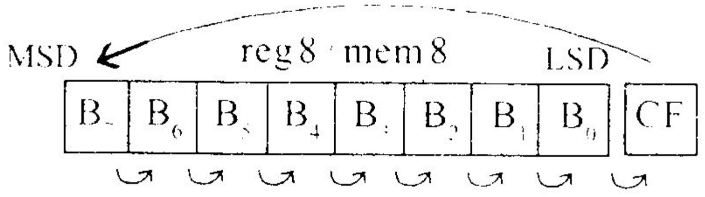
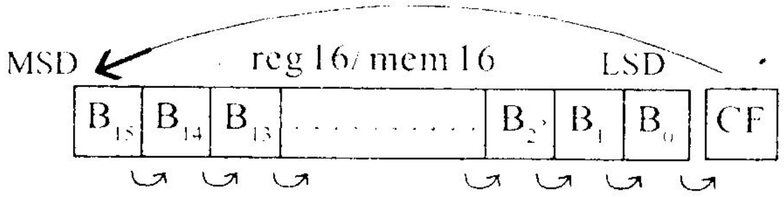




Instruction Set

3. Logical Instructions

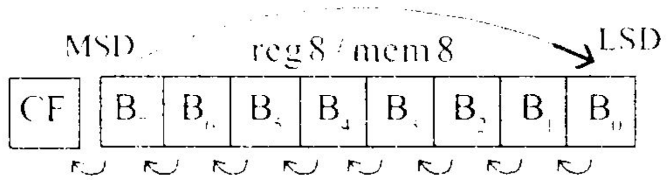
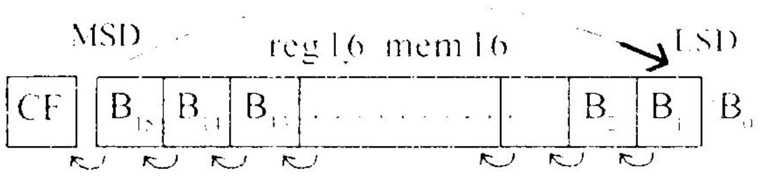
Mnemonics: AND, OR, XOR, TEST, SHR, SHL, RCR, RCL …

RCR reg/mem	$B_n \leftarrow B_{n-1}$; $B_{MSD} \leftarrow CF$; $CF \leftarrow B_{LSD}$
RCR reg i) RCR reg, 1 ii) RCR reg, CL	 MSD reg 8 / mem 8 LSD B7 B6 B5 B4 B3 B2 B1 B0 CF
RCR mem i) RCR mem, 1 ii) RCR mem, CL	 MSD reg 16 / mem 16 LSD B15 B14 B13 ………… B2 B1 B0 CF

Instruction Set

3. Logical Instructions

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

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

MSD
reg8 / mem8
LSD
CF
B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0

MSD
reg16 / mem16
LSD
CF
B_{15} B_{14} B_{13} B_2 B_1 B_0

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) $\leftarrow$ (MA) If DF = 0, then (DI) $\leftarrow$ (DI) + 1; (SI) $\leftarrow$ (SI) + 1 If DF = 1, then (DI) $\leftarrow$ (DI) - 1; (SI) $\leftarrow$ (SI) - 1
MOVSW	MA = (DS) x 16_{10} + (SI) MA_E = (ES) x 16_{10} + (DI) (MA_E ; MA_E + 1) $\leftarrow$ (MA; MA + 1) If DF = 0, then (DI) $\leftarrow$ (DI) + 2; (SI) $\leftarrow$ (SI) + 2 If DF = 1, then (DI) $\leftarrow$ (DI) - 2; (SI) $\leftarrow$ (SI) - 2

Instruction Set

4. String Manipulation Instructions

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

Compare two string byte or string word

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

Instruction Set

4. String Manipulation Instructions

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

Scan (compare) a string byte or word with accumulator

SCAS	
SCASB	$MA_E = (ES) \times 16_{10} + (DI)$ Modify flags $\leftarrow$ $(AL) - (MA_E)$ 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 $(DI) \leftarrow (DI) + 1$ If $DF = 1$, then $(DI) \leftarrow (DI) - 1$
SCASW	$MA_E = (ES) \times 16_{10} + (DI)$ Modify flags $\leftarrow$ $(AL) - (MA_E)$ 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 $(DI) \leftarrow (DI) + 2$ If $DF = 1$, then $(DI) \leftarrow (DI) - 2$

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

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

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

    …
    …
    …          Program code
    …          or
    …          Data Defining Statements
    …

Segnam ENDS
```

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:

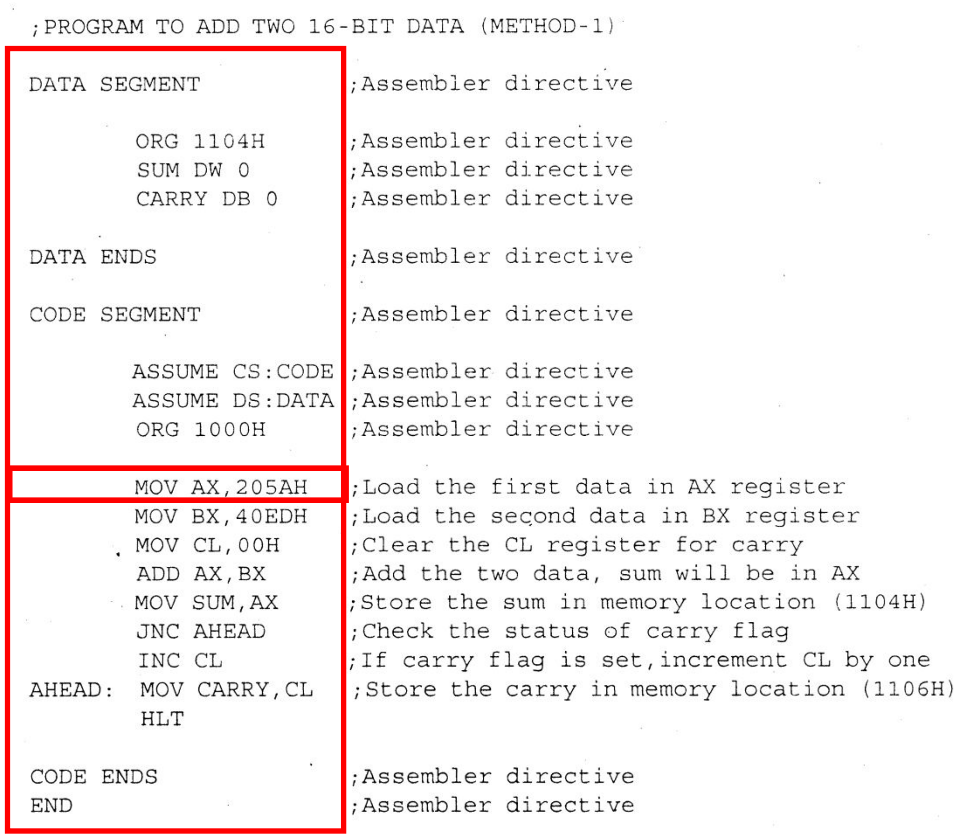
```
macroname MACRO[Arg1, Arg2 ...]

          ...
          ...
          ...

macroname ENDM
```

Program statements in the macro

User defined name of the macro

Sample Programs

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.

$$\begin{tikzcd}
 & \text{Computer language} \arrow[dl] \arrow[dr] & \\
\text{High Level} & & \text{Low Level} \arrow[dll] \arrow[d] \\
\text{Machine Language} & & \text{Assembly Language}
\end{tikzcd}$$

- **Binary bits**

- **English Alphabets**
- **'Mnemonics'**
- **Assembler**

Mnemonics → Machine Language

WAP to find largest no. out of an array of 6 nos OF BYTE Type.

```
DATA SEGMENT
                LIST DB 12H,4H5,28H,25H,F3H,A5H
                COUNT EQU 06
                LARGEST_NO DB 01 DUP(?)
DATA ENDS
CODE SEGMENT
                ASSUME CS: CODE, DS: DATA                                   START:MOV
                START:AX, 2000H
                MOV DS, AX                      ; INITIALIZE THE SEGMENT
                MOV CX, COUNT
                DEC CX
                MOV SI,OFFSET  LIST     ;LOAD OFFSET OFVARIABLE LIST TO SI
                MOV AL, [SI]            ; LOAD THE FIRST NO. IN AL
BACK:           INC SI                  ; POINTER TO NEXT NO.
                CMP AL, [SI]            ; COMPARE TWO NOS.
                JNL NEXT        ; IF 1ST NO. LARGER THAN NEXT NO.JUMP TO
                ;REPEAT LOOP
                MOV AL, [SI]        ;ELSE SWAP THE NOS..
NEXT: LOOP BACK
          MOV DI, OFFSET LARGEST_NO ;STORE RESULT IN VARIABLE LARGEST
           MOV [DI], AL
CODE ENDS
END START
```

WAP to find largest no. out of an array of 6 nos OF WORD Type.

```
DATA SEGMENT
          LIST DW 1250H,4500H,28FFH,2560H,A550H,F3000H
          COUNT EQU 0AH
          LARGEST_NO DW 01 DUP(?)
DATA ENDS

CODE SEGMENT
          ASSUME CS: CODE, DS: DATA
          START:MOV AX, 2000H
          MOV DS, AX                ; INITIALIZE THE SEGMENT
          MOV CX, COUNT
          DEC CX
          MOV SI,OFFSET  LIST   ;LOAD OFFSET OFVARIABLE LIST TO SI
          MOV AX, [SI]          ; LOAD THE FIRST NO. IN AX
BACK:     INC SI
     INC SI                    ; POINTER TO NEXT NO.
          CMP AX, [SI]          ; COMPARE TWO NOS.
          JNL NEXT              ; IF 1ST NO. LARGER THAN NEXT NO.JUMP
               ;TO REPEAT LOOP
          MOV AX, [SI]               ;ELSE SWAP THE NOS..
NEXT: LOOP BACK
     MOV DI, OFFSET LARGEST_NO   ;STORE RESULT IN VARIABLE
     MOV [DI], AX
CODE ENDS
END START
```

WAP to sort the no. in ascending order in an array of 10 nos OF BYTE Type.(METHOD-I)

```
DATA SEGMENT
LIST DB 12H,45,28H,25H,A5H,F3H,14H,F4H,5AH,85H
COUNT EQU 10
DATA ENDS

CODE SEGMENT
           ASSUME CS: CODE, DS: DATA
START:  MOV AX, 2000H
        MOV DS, AX                         ; INITIALIZE THE SEGMENT

       MOV CX, COUNT
       DEC CX
BACK2: MOV DX, CX
       MOV SI,OFFSET  LIST         ;LOAD OFFSET OFVARIABLE LIST TO SI
BACK1: MOV AL, [SI]                          ; LOAD THE FIRST NO. IN AL
       INC SI                                ; POINTER TO NEXT NO.
       CMP AL, [SI]                         ; COMPARE TWO NOS.
       JC NEXT                ; IF 1ST NO. LARGER THAN NEXT NO.JUMP TO REPEAT LOOP
       MOV BH,[SI]
       MOV [SI],AL
       DEC SI
      MOV [SI],BH
      INC SI
 NEXT: DEC DX
      JNZ BACK1
      LOOP BACK2
CODE ENDS
           END START
```

WAP to sort the no. in ascending order in an array of 10 nos OF BYTE Type.(METHOD-II)

```
DATA SEGMENT
LIST DB 12H,45H,28H,25H,A5H,F3H,14H,F4H,5AH,85H
COUNT EQU 10
DATA ENDS

CODE SEGMENT
           ASSUME CS: CODE, DS: DATA
START:  MOV AX, 2000H
        MOV DS, AX                        ; INITIALIZE THE SEGMENT

       MOV CX, COUNT
       DEC CX
BACK2: MOV DX,COUNT
       DEC DX
       MOV SI,OFFSET  LIST         ;LOAD OFFSET OF VARIABLE LIST TO SI
BACK1: MOV AL, [SI]                           ; LOAD THE FIRST NO.
       CMP AL, [SI+1]                ; COMPARE TWO NOS.
       JC NEXT                ; IF 1ST NO. LARGER THAN NEXT NO.JUMP TO REPEAT LOOP
      XCHG AL, [SI+1]
     MOV [SI], AL
     INC SI
 NEXT: DEC DX
      JNZ BACK1
      LOOP BACK2
CODE ENDS
           END START
```