UNIT-II
8086 ASSEMBLY LANGUAGE PROGRAMMING

Contents at a glance:

- ✓ 8086 Instruction Set
- ✓ Assembler directives
- ✓ Procedures and macros.

8086 MEMORY INTERFACING:
- ✓ 8086 addressing and address decoding
- ✓ Interfacing RAM, ROM, EPROM to 8086

INSTRUCTION SET OF 8086

The 8086 instructions are categorized into the following main types

(i) **Data copy /transfer instructions:** These type of instructions are used to transfer data from source operand to destination operand. All the store, load, move, exchange input and output instructions belong to this category.

(ii) **Arithmetic and Logical instructions:** All the instructions performing arithmetic , logical, increment, decrement, compare and ASCII instructions belong to this category.

(iii) **Branch Instructions:** These instructions transfer control of execution to the specified address. All the call, jump, interrupt and return instruction belong to this class.

(iv) **Loop instructions:** These instructions can be used to implement unconditional and conditional loops. The LOOP, LOOPNZ , LOOPZ instructions belong to this category.

(v) **Machine control instructions:** These instructions control the machine status. NOP, HLT, WAIT and LOCK instructions belong to this class.

(vi) **Flag manipulation instructions:** All the instructions which directly effect the flag register come under this group of instructions. Instructions like CLD, STD, CLI, STI etc.., belong to this category of instructions.

(vii) **Shift and Rotate instructions:** These instructions involve the bit wise shifting or rotation in either direction with or without a count in CX.

(viii) **String manipulation instructions:** These instructions involve various string manipulation operations like Load, move, scan, compare, store etc..,

1. Data Copy/ Transfer Instructions:

The following instructions come under data copy / transfer instructions:

MOV	PUSH	POP	IN	OUT	PUSHF	POPF	LEA	LDS/LES	XLAT
XCHG	LAHF	SAHF							

Data Copy/ Transfer Instructions:
MOV: MOVE: This data transfer instruction transfers data from one register / memory location to another register / memory location. The source may be any one of the segment register or other general purpose or special purpose registers or a memory location and another register or memory location may act as destination.

Syntax: 1) MOV mem/reg1, mem/reg2

[mem/reg1] ← [mem/reg2]

Ex: MOV BX, 0210H
MOV AL, BL
MOV [SI], [BX] → is not valid

Memory uses DS as segment register. No memory to memory operation is allowed. It won't affect flag bits in the flag register.

2) MOV mem, data
[mem] ← data

Ex: MOV [BX], 02H
MOV [DI], 1231H

3) MOV reg, data
[reg] ← data

Ex: MOV AL, 11H
MOV CX, 1210H

4) MOV A, mem
[A] ← [mem]

Ex: MOV AL, [SI]
MOV AX, [DI]

5) MOV mem, A
[mem] ← A
A ← : AL/AX

Ex: MOV [SI], AL
MOV [SI], AX

6) MOV segreg,mem/reg
[segreg] ← [mem/reg]

Ex: MOV SS, [SI]

7) MOV mem/reg, segreg
[mem/reg] ← [segreg]

Ex: MOV DX, SS

In the case of immediate addressing mode, a segment register cannot be destination register. In other words, direct loading of the segment registers with immediate data is not permitted. To load the segment registers with immediate data, one will have to load any general-purpose register with the data and then it will have to be moved to that particular segment register.

Ex: Load DS with 5000H
1) MOV DS, 5000H; Not permitted (invalid)

Thus to transfer an immediate data into the segment register, the convert procedure is given below:

2) MOV AX, 5000H
MOV DS, AX

Both the source and destination operands cannot be memory locations (Except for string instructions)

Other MOV instructions examples are given below with the corresponding addressing modes.

3)	MOV AX, 5000H;	Immediate
4)	MOV AX, BX;	Register
5)	MOV AX, [SI];	Indirect
6)	MOV AX, [2000H];	Direct
7)	MOV AX, 50H[BX];	Based relative, 50H displacement

PUSH: **Push to Stack**: This instruction pushes the contents of the specified register/memory location on to the stack. The stack pointer is decremented by 2, after each execution of the instruction. The actual current stack-top is always occupied by the previously pushed data. Hence, the push operation decrements SP by two and this store the two-byte contents of the operand onto the stack. The higher byte is pushed first and then the lower byte. Thus out of the two decremental stack addresses the higher byte occupies the higher address and the lower byte occupies the lower address.

Syntax: PUSH reg
[SP] $\leftarrow$ [SP]-2
[[S]] $\leftarrow$ [reg]

Ex:

1) PUSH AX
2) PUSH DS

3) PUSH [5000H]; content of location 5000H & 5001H in DS are pushed onto the stack.

POP: **Pop from stack**: This instruction when executed, loads the specified register / memory location with the contents of the memory location of which address is formed using the current stack segment and stack pointer as usual. The stack pointer is incremented by 2. The POP instruction serves exactly opposite to the PUSH instruction.

Syntax:

i) POP mem
[SP] $\leftarrow$ [SP] +2
[mem] $\leftarrow$ [[SP]]

ii) POP reg
[SP] $\leftarrow$ [SP] + 2
[reg] $\leftarrow$ [[SP]]

Ex:

1. POP AX
2. POP DS
3. POP [5000H]

XCHG: **Exchange**: This instruction exchanges the contents of the specified source and destination operands, which may be registers or one of them may be a memory location. However, exchange of data contents of two memory locations is not permitted.

Syntax:

i) XCHG AX, reg 16
[AX] $\longleftrightarrow$ [reg 16]
Ex: XCHG AX, DX

ii) XCHG mem, reg
[mem] $\longleftrightarrow$ [reg]
Ex: XCHG [BX], DX

Register and memory can be both 8-bit and 16-bit and memory uses DS as segment register.

iii) XCHG reg, reg
[reg] $\longleftrightarrow$ [reg]

Ex: XCHG AL, CL
XCHG DX, BX

Other examples:

1. XCHG [5000H], AX; This instruction exchanges data between AX and a memory location [5000H] in the data segment.
2. XCHG BX; This instruction exchanges data between AX and BX.

I/O Operations:

IN: Input the port: This instruction is used for reading an input port. The address of the input port may be specified in the instruction directly or indirectly AL and AX are the allowed destinations for 8 and 16-bit input operations. DX is the only register (implicit), which is allowed to carry the port address.

Ex: 1. IN AL, DX
[AL] ← [PORT DX]
Input AL with the 8-bit contents of the port addressed by DX

2. IN AX, DX
[AX] ← [PORT DX]
3. IN AL, PORT
[AL]← [PORT]

4. IN AX, PORT
[AX]←[PORT]

5. IN AL, 0300H; This instruction reads data from an 8-bit port whose address is 0300H and stores it in AL.

6. IN AX ; This instruction reads data from a 16-bit port whose address is in DX (implicit) and stores it in AX.

OUT: Output to the Port: This instruction is used for writing to an output port.The address of the output port may be specified in the instruction directly or implicitly in DX. Contents of AX or AL are transferred to a directly or indirectly addressed port after execution of this instruction. The data to an odd addressed port is transferred on D_8 –D_{15} while that to an even addressed port is transferred on D_0-D_7.The registers AL and AX are the allowed source operands for 8-bit and 16-bit operations respectively.

Ex: 1. OUTDX,AL
[PORT DX] ← [AL]
2. OUT DX,AX
[PORT DX] ← [AX]
3. OUT PORT,AL
[PORT] ← [AL]
4. OUT PORT,AX
[PORT] ← [AX]
Output the 8-bit or 16-bit contents of AL or AX into an I/O port addressed by the contents of DX or local port.
5. OUT 0300H,AL; This sends data available in AL to a port whose address is 0300H
6. OUT AX; This sends data available in AX to a port whose address is specified implicitly in DX.

2. Arithmetic Instructions:

ADD	ADC	SUB	SBB	MUL	IMUL	DIV	IDIV	CMP	NEGATE
INC	**DEC**	**DAA**	**DAS**	**AAA**	**AAS**	**AAM**	**AAD**	**CBW**	**CWD**

These instructions usually perform the arithmetic operations, like addition, subtraction, multiplication and division along with the respective ASCII and decimal adjust instructions. The increment and decrement operations also belong to this type of instructions. The arithmetic instructions affect all the conditional code flags. The operands are either the registers or memory locations immediate data depending upon the addressing mode.

ADD: **Addition:** This instruction adds an immediate data or contents of a memory location specified in the instruction or a register (source) to the contents of another register (destination) or memory location. The result is in the destination operand. However, both the source and destination operands cannot be memory operands. That means memory to memory addition is not possible. Also the contents of the segment registers cannot be added using this instruction. All the condition code flags are affected depending upon the result.

Syntax: i. ADD mem/reg1, mem/reg2

[mem/reg1]← [mem/reg2] + [mem/reg2]

Ex : ADD BL, [ST]

ADD AX, BX

ii. ADD mem, data

[mem]←[mem]+data

Ex: ADD Start, 02H

ADD [SI], 0712H

iii. ADD reg, data

[reg]←[reg]+data

Ex: ADD CL, 05H

ADD DX, 0132H

iv. ADD A, data

[A]←[A]+data

Ex: ADD AL, 02H

ADD AX, 1211H

Examples with addressing modes:

1. ADD AX, 0100H	Immediate
2. ADD AX, BX	Register
3. ADD AX, [SI]	Register Indirect
4. ADD AX, [5000H]	Direct
5. ADD [5000H], 0100H	Immediate
6. ADD 0100H	Destination AX (implicit)

ADC: Add with carry: This instruction performs the same operation as ADD instruction, but adds the carry flag bit (which may be set as a result of the previous calculations) to the result. All the condition code flags are affected by this instruction.

Syntax: i. ADC mem/reg1, mem/reg2

[mem/reg1]←[mem/reg1]+[mem/reg2]+CY

Ex: ADC BL, [SI]

ADC AX, BX

ii. ADC mem,data

[mem]←[mem]+data+CY

Ex: ADC start, 02H

ADC [SI],0712H

iii. ADC reg, data

[reg]←[reg]+data+CY

Ex: ADC AL, 02H
ADC AX, 1211H

Examples with addressing modes:

1. ADC 0100H	Immediate(AX implicit)
2. ADC AX,BX	Register
3. ADC AX,[SI]	Register indirect
4. ADC AX,[5000H]	Direct
5. ADC [5000H],0100H	Immediate

SUB: Subtract: The subtract instruction subtracts the source operand from the destination operand and the result is left in the destination operand. Source operand may be a register or a memory location, but source and destination operands both must not be memory operands. Destination operand cannot be an immediate data. All the condition code flags are affected by this instruction.

Syntax: i. Sub mem/reg1, mem/reg2
[mem/reg1]←[mem/reg2]-[mem/reg2]

Ex: SUB BL,[SI]
SUB AX, BX

ii. SUB mem/data
[mem]←[mem]-data

Ex: SUB start, 02H
SUB [SI],0712H

iii. SUB A,data
[A]←[A]-data

Ex: SUB AL, 02H
SUB AX, 1211H

Examples with addressing modes:

1. SUB 0100H	Immediate [destination AX]
2. SUB AX, BX	Register
3. SUB AX,[5000H]	Direct
4. SUB [5000H], 0100	Immediate

SBB: Subtract with Borrow: The subtract with borrow instruction subtracts the source operand and the borrow flag (CF)which may reflect the result of the previous calculations, from the destination operand .Subtraction with borrow ,here means subtracting 1 from the subtraction obtained by SUB ,if carry (borrow) flag is set.

The result is stored in the destination operand. All the conditional code flags are affected by this instruction.

Syntax: i. SBB mem/reg1,mem/reg2
[mem/reg1] ← [mem/reg1]-[mem/reg2]-CY

Ex: SBB BL,[SI]
SBB AX,BX

ii. SBB mem,data
[mem] ← [mem]-data-CY

Ex: SBB Start,02H
SBB [SI],0712H

iii. SBB reg,data
[reg] ← [reg]-data-CY

Ex: SBB CL,05H

SBB DX,0132H

iv. SBB A,data
[A] ← [A]-data-CY

Ex: SBB AL,02H
SBB AX,1211H

INC: Increment: This instruction increments the contents of the specified register or memory location by 1. All the condition flags are affected except the carry flag CF. This instruction adds a to the content of the operand. Immediate data cannot be operand of this instruction.

Syntax: i. INC reg16
[reg 16]←[reg 16]+1
Ex: INC BX

ii. INC mem/reg 8
[mem]←[mem]+1
[reg 8]←[reg 8]+1

Ex: INC BL
INC SI

Segment register cannot be incremented. This operation does not affect the carry flag.

Examples with addressing modes:

1. INC AX	Register
2. INC [BX]	Register indirect
3. INC [5000H]	Direct

DEC: **Decrement:** The decrement instruction subtracts 1 from the contents of the specified register or memory location. All the condition code flags except carry flag are affected depending upon the result. Immediate data cannot be operand of the instruction.

Syntax: i. DEC reg16
[reg 16]←[reg 16]-1
Ex: DEC BX

ii. DEC mem/reg8
[mem]←[mem-1
[reg 8]←[reg 8]-1
Ex: DEC BL

Segment register cannot be decremented.

Examples with addressing mode:

1. DEC AX	Register
2. DEC [5000H]	Direct

MUL: Unsigned multiplication Byte or Word: This instruction multiplies unsigned byte or word by the content of AL. The unsigned byte or word may be in any one of the general-purpose register or memory locations. The most significant word of result is stored in DX, while the least significant word of the result is stored in AX. All the flags are modified depending upon the result. Immediate operand is not allowed in this instruction. If the most significant byte or word of the result is '0' IF and OF both will be set.

Syntax: MUL mem/reg

For 8X8
[AX]←[AL]*[mem8/reg8]
Ex: MUL BL
[AX]←[AL]*[BL]
For 16X16
[DX][AX]←[AX]*[mem16/reg16]
Ex: MUL BX

$$\underset{\substack{\downarrow\\ \text{higher}\\ \text{16-bit}}}{[DX]}\underset{\substack{\downarrow\\ \text{lower}\\ \text{16-bit}}}{[AX]} \leftarrow [AX]*[BX]$$

Ex: 1. MUL BH ; [AX]←[AL]*[BH]
2. MUL CX ; [DX][AX]←[AX*[CX]
3. MUL WORD PTR[SI];[DX][AX]←[AX]*[SI]

IMUL: Signed Multiplication: This instruction multiplies a signed byte in source operand by a signed byte in AL or signed word in source operand by signed word in AX. The source can be a general purpose register, memory operand, index register or base register, but it cannot be an immediate data. In case of 32-bit results, the higher order word (MSW) is stored in DX and the lower order word is stored in AX. The AF, PF, SF and ZF flags are undefined after IMUL. If AH and DH contain parts of 16 and 32-bit result respectively, CF and OF both will of set. The AL and AX are the implicit operands in case of 8-bit and 16-bit multiplications respectively. The unused higher bits of the result are filled by sign bit and CF, AF are cleared.

Syntax: IMUL mem/reg
For 8X8
[AX]←[AL]*[mem8/reg8]
Ex: IMUL BL
[AX]←[AL]*[BL]

For 16X16
[DX][AX]←[AX*[mem16/reg16]
Ex: IMUL BX
[DX][AX]←[AX]*[BX]
Memory or register can be 8-bit or 16-bit and this instruction will affect carry flag & overflow flag.
Ex: 1. IMUL BH
2. IMUL CX
3. IMUL [SI]

DIV: Unsigned division: This instruction performs unsigned division. It divides an unsigned word or double word by a 16-bit or 8-bit operand. The dividend must be in AX for 16-bit operation and divisor may be specified using any one of the addressing modes except immediate. The result will be in AL (quotient) while AH will contain the remainder. If the result is too big to fit in AL, type 0(divide by zero) interrupt is generated. In case of a double word dividend (32-bit), the higher word should be in DX and lower word should be in AX. The divisor may be specified as already explained. The quotient and the remainder, in this case, will be in AX and DX respectively. This instruction does not affect any flag.

Syntax: DIV mem/reg

Ex: DIV BL (i.e. [AX]/[BX])

$$\text{For } 16 \div 8 \quad \frac{[AX]}{[\text{mem 8/reg 8}]} \rightarrow \begin{matrix}[AH] \leftarrow \text{Remainder}\\ [AL] \leftarrow \text{Quotient}\end{matrix}$$

For 32 ÷ 16 $\dfrac{[DX]\,[AX]}{[\text{mem 16/reg 16}]} \rightarrow \begin{array}{l}[DX] \leftarrow \text{Remainder}\\ [AX] \leftarrow \text{Quotient}\end{array}$

Ex: DIV BX (i.e. $\dfrac{[DX][AX]}{[BX]}$)

IDIV: Signed Division: This instruction performs same operation as the DIV instruction, but it with signed operands the results are stored similarly as in case of DIV instruction in both cases of word and double word divisions the results will also be signed numbers. The operands are also specified in the same way as DIV instruction. Divide by zero interrupt is generated, if the result is too big to fit in AX (16-bit dividend operation) or AX and DX (32-bit dividend operation) all the flags are undefined after IDIV instruction.

AAA: ASCII Adjust after addition: The AAA instruction is executed after an ADD instruction that adds two ASCII coded operands to give a byte of result in AL. The AAA instruction converts the resulting contents of AL to unpacked decimal digits. After the addition, the AAA instruction examines the lower 4-bits of AL to check whether it contains a valid BCD number in the range 0 to 9. If it is between 0 to 9 and AF is zero, AAA sets the 4- higher order bits of AL to 0. The AH must be cleared before addition. If the lower digit of AL is between 0 to 9 and AF is set, 06 is added to AL. The upper 4-bits of AL are cleared and AH is incremented by one. If the value of lower nibble of AL is greater than 9 then the AL is incremented by 06, AH is incremented by 1, the AF and CF flags are set to 1, and the higher4-bits of AL are cleared to 0. The remaining flags are unaffected. The AH is modified as sum of previous contents (usually 00) and the carry from the adjustment, as shown in Fig1.7. This instruction does not give exact ASCII codes of the sum, but they can be obtained by adding 3030H to AX.

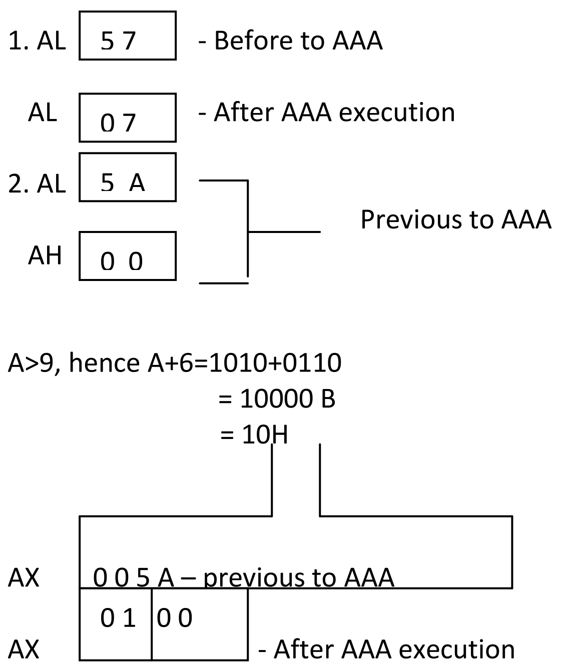


Fig1.7 ASCII Adjust After Addition Instruction

AAS: ASCII Adjust After Subtraction: AAS instruction corrects the result in AL register after subtracting two unpacked ASCII operands. The result is in unpacked decimal format. If the lower 4-bits of AL register are greater than 9 or if the AF flag is one, the AL is decremented by 6 and AH is decremented by 1, the CF and AF are set to 1. Otherwise, the CF and AF are set to 0, the result needs to no correction. As a result, the upper nibble of AL is 00 and the lower nibble may be any number from 0 to 9. The procedure similar to the AAA instruction AH is modified as difference of previous contents (usually 0) of AH and the borrow for adjustment.

AAM: ASCII Adjust after Multiplication: This instruction, after execution, converts the product available in AL into unpacked BCD format. This follows a multiplication instruction. The lower byte of result (unpacked) remains in AL and the higher byte of result remains in AH.

The example given below explains execution of the instruction. Suppose, a product is available in AL, say AL=5D. AAM instruction will form unpacked BCD result in AX. DH is greater than 9, so add of 6(0110) to it D+6=13H. LSD of 13H is the lower unpacked byte for the result. Increment AH by 1, 5+1=6 will be the upper unpacked byte of the result. Thus after the execution, AH=06 and AL=03.

AAD: ASCII Adjust before Division: Though the names of these two instructions (AAM and AAD) appear to be similar, there is a lot of difference between their functions. The AAD instruction converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL. This adjustment must be made before dividing number the two unpacked BCD digits in AX by an unpacked BCD byte. PF, SF, ZF are modified while AF, CF, OF are undefined, after the execution of the instruction AAD. The example explains the execution of the instruction.

Let AX contain 0508 unpacked BCD for 58 decimal and DH contain 02H.
Ex:

AX	5	8

AAD result in AL	0	3A	58D=3AH in AL

The result of AAD execution will give the hexadecimal number 3A in AL and 00 in AH.

DAA: Decimal Adjust Accumulator: This instruction is used to convert the result of the addition of two packed BCD numbers to a valid BCD number. The result has to be only in AL. If the lower nibble is greater than 9, after addition or if AF is set, it will add 06 to the lower nibble in AL. After adding 06 in the lower nibble of AL, if the upper nibble of AL is greater than 9 or if carry flag is set, DAA instruction adds 60H to AL.

The example given below explains the instruction:

```
i. AL=53 CL=29
        ADD AL, CL      ;       AL ←(AL) + (CL)
                        ;       AL←53+29
                        ;       AL←7C
                        ;       AL←7C+06(as C>9)
                        ;       AL←82
ii. AL=73       CL=29
        ADD AL,CL       ;       AL←AL+CL
                        ;       AL←73+29
                        ;       AL←9C
                        ;       AL←9C
        DAA             ;       AL←02 & CF=1
        AL=73
          +
        CL=29
        ________
          9C
          +6
        ________

         A2
         +60
        ________
        CF=1 02 in AL
```

The instruction DAA affects AF, CF, PF and ZF flags. The OF flag is undefined.

DAS: Decimal Adjust After Subtraction: This instruction converts the results of subtraction of two packed BCD numbers to a valid BCD number. The subtraction has to be in AL only. If the lower nibble of AL is greater than 9, this instruction will subtract 06 from lower nibble of AL. If the result of subtractions sets the carry flag or if upper nibble is greater than 9, it subtracts 60H from AL. This instruction modifier the AF, CF, PF and ZF flags. The OF is undefined after DAS instruction.
The examples are as follows:

```
Ex:   i.    AL=75   BH=46
            SUB AL,BH    ;    AL←2F=(AL)-(BH)
                         ;    AF=1
            DAS          ;    AL←29 (as F>9,F-6=9)


      ii.   AL=38   CH=61
            SUB AL, CH   ;    AL←D7 CF=1(borrow)
            DAS          ;    AL←77(as D>9, D-6=7)
                         ;    CF=1(borrow)
```

NEG: Negate: The negate instruction forms 2's complement of the specified destination in the instruction. For obtaining 2's complement, it subtracts the contents of destination from zero. The result is stored back in the destination operand which may be a register or a memory location. If OF is set, it indicates that the operation could not be completed successfully. This instruction affects all the condition code flags.

CBW: Convert signed Byte to Word: This instruction converts a signed byte to a signed word. In other words, it copies the sign bit of a byte to be converted to all the bits in the higher byte of the result word. The byte to be converted must be in AL. The result will be in AX. It does not affect any flag.

CWD: Convert Signed Word to double Word: This instruction copies the sign bit of AX to all the bits of DX register. This operation is to be done before signed division. It does not affect any other flag.

3. Logical Instructions:

AND	OR	NOT	XOR	TEST

These byte of instructions are used for carrying out the bit by bit shift, rotate or basic logical operations. All the conditional code flags are affected depending upon the result. Basic logical operations available with 8086 instruction set an AND, OR, NOT and XOR.

AND: Logical AND: This instruction bit by bit ANDs the source operand that may be an immediate, a register, or a memory location to the destination operand that may be a register or a memory location. The result is stored in the destination operand. At least one of the operand should be a register or a memory operand. Both the operands cannot be memory locations or immediate operand.

The examples of this instruction are as follows:

```
Syntax: i.  AND mem/reg1, mem/reg2
                [mem/reg1]←[mem/reg1]∧[mem/reg2]
            Ex:     AND BL, CH

            ii.     AND mem,data
                    [mem]←[mem] ∧ data
            Ex:     AND start,05H

            iii.    AND reg,data
                    [reg]←[reg] ∧ data
```

Ex: AND AL, FOH

iv. AND A,data
$[A] \leftarrow [A] \wedge$ data
A:AL/AX
Ex: AND AX,1021H

OR: Logical OR: The OR instruction carries out the OR operation in the same way as described in case of the AND operation. The limitations on source and destination operands are also the same as in case of AND operation.

Syntax: i. OR mem/reg1, mem/reg2
$[mem/reg1] \leftarrow [mem/reg1] \vee [mem/reg2]$
Ex: OR BL, CH

ii. OR mem,data
$[mem \leftarrow [mem] \vee$ data
Ex: OR start, 05H

iii. OR Start,05H
$[reg] \leftarrow [reg] \vee$ data
Ex: OR AL, FOH

iv. OR A, data
$[A] \leftarrow [A] \vee$ data
Ex: OR AL, 1021H
A: AL/AX.

NOT: Logical Invert: The NOT instruction complements (invents) the contents of an operand register or a memory location bit by bit.

Syntax: i. NOT reg
$[reg] \leftarrow [reg]'$
Ex: NOT AX

ii. NOT mem
$[mem] \leftarrow [mem]'$
Ex: NOT [SI]

XOR: Logical Exclusive OR: The XOR operation is again carried out in a similar way to the AND and OR operation. The constraints on high output, when the 2 input bits are dissimilar. Otherwise, the output is zero.

Syntax: i. XOR mem/reg1, mem/reg2
$[mem/reg1] \leftarrow [mem/reg1] \oplus [mem/reg2]$
Ex: XOR BL, CH

ii. XOR mem,data
$[mem] \leftarrow [mem] \oplus$ data
Ex: XOR start, 05H

iii. XOR reg, data
$[reg] \leftarrow [reg] \oplus$ data
Ex: XOR AL, FOH

iv. XOR A, data
$[A] \leftarrow [A] \oplus$ data

A: AL/AX
Ex: XOR AX, 1021H

CMP: Compare: This instruction compares the source operand, which may be a register or an immediate data or a memory location, with a destination operand that may be a register or a memory location. For comparison, it subtracts the source operand from the destination operand but does not store the result anywhere. The flags are affected depending on the result of subtraction. If both the operands are equal, zero flag is set. If the source operand is greater than the destination operand, carry flag is set or else, carry flag is reset.

Syntax: i. CMP mem/reg1, mem/reg2
[mem/reg1] – [mem/reg2]
Ex: CMP CX, BX

ii. CMP mem/reg, data
[mem/reg] – data
Ex: CMP CH, 03H

iii. CMP A, data
[A]- data
A: AL/AX
Ex: CMP AX, 1301H

TEST: Logical Compare Instruction: The TEST instruction performs a bit by bit logical AND operation on the two operands. Each bit of the result is then set to 1, if the corresponding bits of both operands are1, else the result bit is rest to 0. The result of this and operation is not available for further use, but flags are affected. The affected flags are OF, CF, ZF and PF. The operands may be register, memory or immediate data.

Syntax: i. TEST mem/reg1, mem/reg2
[mem/reg1] $\wedge$ [mem/reg2]
Ex: Test CX,BX
ii. TEST mem/reg, data
[mem/reg] $\wedge$ data
Ex: TEST CH, 03H

iii. TEST A, data
[A] $\wedge$ data
A: AL/AX
Ex: TEST AX, 1301H

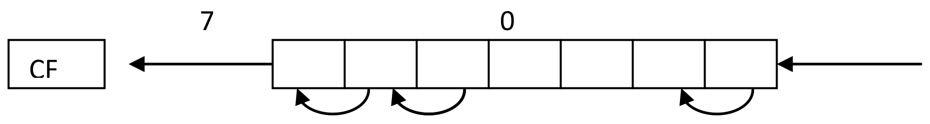
4.Shift Instructions:

SHL/SAL	**SHR**	**SAR**

SHL/SAL: Shift Logical/ Arithmetic Left: These instructions shift the operand word or byte bit by bit to the left and insert zeros in the newly introduced least significant bits. In case of all the SHIFT and ROTATE instructions, the count is either 1 or specified by register CL. The operand may reside in a register or memory location but cannot be immediate data. All flags are affected depending on the result.
Ex:

```
BIT POSITIONS: CF 15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
OPERAND:        1   0   1   0   1   1   0  0  1  0  1  0  0  1  0  1
                ________________________________________________________
SHL          1      0   1   0   1   1   0  0  1  0  1  0  0  1  0  1  0
RESULT1st      ________________________________________________________
SHL          0  1   0   1   1   0   0   1  0  1  0  0  1  0  1  0  0
RESULT 2nd     ________________________________________________________
```

Syntax: i. SAL mem/reg,1
Shift arithmetic left once



ii. SAL mem/reg, CL

Shift arithmetic left a byte or word by shift count in CL register.

iii. SHL mem/reg,1
Shift Logical Left
Ex: SHL BL, 01H

iv. SHL mem/reg, CL
Shift Logical Left once a byte or word in mem/reg.

SHR: Shift Logical Right: This instruction performs bit-wise right shifts on the operand word or byte that may reside in a register or a memory location, by the specified count in the instruction and inserts zeros in the shifted positions. The result is stored in the destination operand. This instruction shifts the operand through carry flag.

Ex:

```
BIT POSITIONS: 15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0  CF
OPERAND       : 1   0   1   0   1   1  0  0  1  0  1  0  0  1  0  1
              ________________________________________________________
Count=1          0  1   0   1   0   1   1  0  0  1  0  1  0 0 1  0  1
              ________________________________________________________
Count=2       0  0   1   0   1   0   1   1  0  0  1  0  1  0  0  1  0
              ________________________________________________________
```

SAR: Shift Arithmetic Right: This instruction performs right shifts on the operand word or byte, that may be a register or a memory location by the specified count in the instruction and inserts the most significant bit of the operand the newly inserted positions. The result is stored in the destination operand. All the condition code flags are affected. This shift operation shifts the operand through carry flag.

Ex:

```
BIT POSITIONS: 15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0 CF
OPERAND:          1   0   1   0   1   1  0  0  1  0  1  0  0  1  0  1
              ________________________________________________________
Count=1          1  1   0   1   0   1   1  0  0  1  0  1  0  0  1  0  1
                 ________________________________________________________
inserted MSB=1
              ________________________________________________________
Count=2       1   1   1  0   1   0   1  1  0  0  1  0  1  0  0  1  0
              ________________________________________________________
              inserted MSB=1
```

Immediate operand is not allowed in any of the shift instructions.

Syntax: i. SAR mem/reg,1
ii. SAR mem/reg, CL

5.Rotate Instructions:

ROR	**ROL**
RCR	**RCL**

ROR: Rotate Right without Carry: This instruction rotates the contents of the destination operand to the right (bit-wise) either by one or by the count specified in CL, excluding carry. The least significant bit is pushed into the carry flag and simultaneously it is transferred into the most significant bit position at each operation. The remaining bits are shifted right by the specified positions. The PF, SF, and ZF flags are left unchanged by the rotate operation. The operand may be a register or a memory location but it can't be an immediate operand. The destination operand may be a register (except a segment register) or a memory location.

Syntax: i. mem/reg, 01
Ex: ROR BL, 01

ii. ROR mem/reg, CL
Ex: ROR BX, CL

```
                                                Ex:
BIT POSITIONS: 15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0     CF
OPERAND:        1   0   1   0   1   1   1  1  0  1  0  1  1  1  0  1
               ____________________________________________________
Count=1           1  1  0   1   0   1   1  1  1  0  1  0  1  1  1  0  1
               ____________________________________________________
Count=2        0  1  1  0   1   0   1   1  1  1  0  1  0  1  1  1  0
               ____________________________________________________
```

Execution of ROR Instruction.

ROL: Rotate Left without Carry: This instruction rotates the content of the destination operand to the left by the specified count (bit-wise) excluding carry. The most significant bit is pushed into the carry flag as well as the least significant bit position at each operation. The remaining bits are shifted left subsequently by the specified count positions. The PF, SF and ZF flags are left unchanged by this rotate operation. The operand may be a register or a memory location.

Syntax: i. ROL mem/reg, 1
Rotate once left

ii. ROL mem/reg, CL
Rotate once left a byte or a word in mem/reg.

```
                                                Ex:
BIT POSITIONS: CF 15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
OPERAND          :  1   0   1   0   1   1   1  1  0  1  0  1  1  1  0  1
                    ____________________________________________________
SHL RESULT 1st:1    0  1   0   1   1   1   1  0  1  0  1  1  1  0  1  1
                  ____________________________________________________
SHL RESULT 2nd: 0   1   0   1   1   1   1  0  1  0  1  1  1  0  1  1  0
                  ____________________________________________________
```

Execution of ROL instruction

RCR: Rotate Right Through Carry: This instruction rotates the contents (bit-wise) of the destination operand right by the specified count through carry flag (CF) For each operation, the carry flag is pushed into the MSB of the operand, and the LSB is pushed into carry flag. The remaining bits are shifted right by the specified count positions. The SF, PF, ZF are left unchanged. The operand may be a register or memory location.

Syntax: i. RCL mem/reg, 1
Ex: RCL BL, 1

ii. mem/reg, CL
Ex: RCL BX, CL

Rotate through carry left once a byte or word in mem/reg.
Ex:

```
BIT POSITIONS: 15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0 CF
OPERAND  :      1   0   1   0   1   1   1  1  0  1  0  1  1  1  0  1  0
               ________________________________________________________
Count=1          0   1   0   1   0   1   1  1  1  0  1  0  1  1  1  0  1
                 ________________________________________________________
```

Execution of RCR Instruction

RCL: Rotate Left through Carry: This instruction rotates (bit-wise) the contents of the destination operand left by the specified count through the carry flag (CF) For each operation, the carry flag is pushed into LSB, and the MSB of the operand is pushed into carry flag. The remaining bits are shifted left by the specified positions. The SF, PF, ZF are left unchanged. The operand may be a register or a memory location.

Ex:

```
BIT POSITIONS :CF  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
OPERAND :       0   1   0   0   1   1   1   0  1 1  0  1  1  0  1  0 1
             ______________________________________________________________
Count=1          1   0   0   1   1   1   0  1  1  0  1  1  0  1  0  1  0
             ______________________________________________________________
```

Execution of RCL Instruction

The count for rotation or shifting is either 1 or is specified using register CL, in case of all the shift and rotate instructions.

6.String Manipulation Instructions:
A series of data bytes or words available in memory at consecutive locations, to be referred to collectively or individually are called as **byte strings** or **word strings**.

REP	**MOVSB/MOVSW**	**CMPSB/CMPSW**	**SCASB/SCASW**
STOSB/STOSW	**LODSB/LODSW**		

REP: Repeat Instruction Prefix: This instruction is used as a prefix to other instructions. The instruction to which the REP prefix is provided, is executed repeatedly until the CX register becomes zero (at each iteration CX is automatically decremented by one) When CX becomes zero, the execution proceeds to the next instruction in sequence. There are two more options of the REP instruction. The first is REPE/REPZ (i.e. repeat operation which equal/zero. The second is REPNE/REPNZ allows for repeating the operation which not equal/not zero. These options are used for CMPS, SCAS instructions only, as instruction prefixes.

MOVSB/MOVSW: Move String Byte or String Word: Suppose a string of bytes, stored in a set of consecutive memory locations is to be moved to another set of destination locations. The starting byte of the source string is located in the memory location whose address may be computed using SI (Source Index) and DS (Data Segment) contents. The starting address of the destination locations where this string has to be relocated is given by DI (Destination Index) and ES (Extra Segment) contents. The starting address of the source string is 10H * DS + [SI] while the starting address of the destination string is 10H * ES + [DI]. The MOVSB/MOVSW instruction thus, moves a string of bytes/words pointed to by DS:SI pair (source) to the memory location pointed to by ES:DI pair (destination)
After the MOVS instruction is executed once, the index registers are automatically updated and CX is decremented. The incrementing or decrementing of the pointers, i.e. SI and DI depend upon the direction flag DF. If DF is 0, the index registers are incremented, otherwise, they are decremented, in case of all string manipulation instructions.
Ex:

```
(a)
 DATA   SEGMENT
        TEST-MESS      DB "IT'S TIME FOR A NEW HOME"        ;string to move
                       DB  100  DUP(?) ;stationary block of text
        NEW-LOC        DB   23   DUP(0) ;string destination.
DATA  ENDS

CODE    SEGMENT
        ASSUME         CS:CODE,DS:DATA,ES:DATA
        MOV     AX,DATA         ;initialize data segment register
        MOV     DS,AX
        MOV     ES,AX           ;initialize extra segment register
        LEA     SI,TEST-MESS    ;point SI at source string
        LEA     DI,NEW-LOC              ;point DI at destination string
        MOV     CS,23                   ;use CX register as counter
        CLD                             ;clear DF, so pointers auto increment
REP     MOVSB                   ;after each string element is moved
                                        ;move string byte until all moved
CODE    ENDS
        END
                       (b)
```

Fig : program for moving a string from one location to another in memory
(a) Memory map (b) AL program.

Here, the REPEAT-UNTIL loop then consists of moving a byte, incrementing the pointers to point to the source and destination for next byte, and decrementing the counter to determine whether all bytes have been moved.

The single 8086 instruction MOVSB will perform all the actions in the REPEAT-UNTIL loop. The MOVSB instruction will copy a byte from the location pointed to by the DI register. It will then automatically increment SI to point to next destination location. The repeat (REP) prefix in front of the MOVSB instruction, the MOVSB instruction will be repeated and CX decremented until CX is counted down to zero. In other words, the REP MOVSB instruction will move the entire string from the source location to the destination location if the pointers are properly initialized.
CMPSB/CMPSW: Compare String Byte or String Word: The CMPS instruction is used to compare two strings of bytes or words. The length of the string must be stored in the register CX. If both the byte or word strings are equal, zero flag is set. The flags are affected in the same way as CMP instruction. The DS:SI and ES:DI point to the two strings. The REP instruction prefix is used to repeat the operation till CX (counter) becomes zero or the condition specified by the REP prefix is false.
The following string of instructions explain the instruction. The comparison of the string starts from initial or word of the string, after each comparison the index registers are updated depending on the direction flag and the counter is decremented. This byte by byte or word by word comparison continues till a mismatch is found. When, a mismatch is found, the carry and zero flags are modified appropriately and the execution proceeds further.

Ex:

```
MOV AX, SEG1          ; Segment address of String1, i.e. SEG1 is moved to AX.
MOV DS, AX                          ; Load it to DS.
MOV AX, SEG2          ; segment address of STRING2, i.e. SEG@ is moved to AX.
MOV ES, AX                           ; Load it to ES.
MOV SI, OFFSET STRING1          ; Offset of STRING1 Is moved to SI.
MOV DI, OFFSET STRING2          ; Offset of string2 is moved to DI.
MOV CX, 0110H              ; Length of string is moved to CX.
CLD                             ; clear DF, i.e. set auto increment mode.
REPE CMPSW                      ; Compare 010H words of STRING1 And STRING2, while     they are equal, IF a
                                mismatch is found, modify the flags and proceed with further execution.
```

If both strings are completely equal, i.e. CX becomes zero, the ZF is set, otherwise ZF is reset.

SCAS: Scan String BYTE or String Word: This instruction scans a string of bytes or words for an operand byte or word specified in the register AL or AX. The string is pointed to by ES:DI register pair. The length of the string is stored in CX. The DF controls the mode for scanning of the string as stated in case of MOVSB instruction. Whenever a match to the specified operand, is found in the string, execution stops and the zero flag is set. If no match is found, the zero flag is reset. The REPNE prefix is used with the SCAS instruction. The pointers and counters are updated automatically, till a match is found.

Ex:

```
MOV AX, SEG         ; Segment address of the string, i.e. SEG is moved to AX.
MOV ES, AX          ; Load it to ES.
MOV DI, OFFSET      ; String offset, i.e. OFFSET is moved to DI.
MOV CX,010H         ; Length of the string is moved to CX.
MOV AX, WORD        ; The word to be scanned for, i.e. WORD is in AL.
CLD                 ; Clear DF
REPNE SCASW         ; Scan the 010H bytes of the string, till a match to WORD is found.
```

This string of instructions finds out, if it contains WORD. IF the WORD is found in the word string, before CX becomes zero, the ZF is set, otherwise the ZF is reset. The scanning will continue till a match is found. Once a match is found the execution of the program proceeds further.

LODS: Load string Byte or String word: The LODS instruction loads the AL/AX register by the content of a string pointed to by DS:SI register pair. The SI is modified automatically depending on DF. If it is a byte transfer (LODSB), the SI is modified bye one and if it is a word transfer (LODSW), the SI is modified by two. No other flags are affected by this instruction.

STOS: Store String Byte or String Word: The STOS instruction stores the AL/AX register contents to a location in the string pointed by ES:DI register pair. The DI is modified Accordingly. No flags are affected by this instruction.

The direction flag controls the string instruction execution. The source index SI and destination index DI are modified after each iteration automatically. If DF=1, then the execution follows auto decrement mode. In this mode, SI and DI are decremented automatically after each iteration (by1 or 2 depending on byte or word operations) Hence, in auto decrementing mode, the string are referred to by their ending addresses. If DF=0, then the execution follows auto increment mode. In this mode, SI and DI are incremented automatically (by 1 or 2 depending on byte or word operation) After each iteration, hence the strings, in this case, are referred to by their starting addresses.

7.Control Transfer or Branching Instruction:

The control transfer instructions transfer the flow of execution of the program to a new address specified in the instruction directly or indirectly. When this type of instruction is executed, the CS and IP registers get loaded with new values of CS and IP corresponding to the location where the flow of execution is going to be transferred.

This type of instructions are classified in two types:

i. **Unconditional control Transfer (Branch) Instructions:**
In case of unconditional control transfer instructions; the execution control is transferred to the specified location independent of any status or condition. The CS and IP are unconditionally modified to the new CS and IP.

ii. **Conditional Control Transfer (Branch) Instructions:**
In the conditional control transfer instructions, the control is transferred to the specified location provided the result of the previous operation satisfies a particular condition, otherwise, the execution continues in normal flow sequence. The results of the previous operations are replicated by condition code flags. In other words, using type of instruction the control will be transferred to a particular specified location, if a particular flag satisfies the condition.

Unconditional Branch Instructions:

CALL	**RET**	**JUMP**	**IRET**
INT N	**INT O**	**LOOP**	

CALL: Unconditional Call: This instruction is used to call a subroutine procedure from a main program. The address of the procedure may specify directly or indirectly depending on the address mode.
There are again two types of procedures depending on whether it is available in the same segment (Near CALL, i.e. + 2K displacement) or in another segment (Far CALL, i.e. anywhere outside the segment). The modes for them are respectively called as intrasegment and intersegment addressing (i.e. address of the next instruction) and CS onto the stack along with the flags and loads the CS and IP registers, respectively, with the segment and offset addresses of the procedure to be called.

RET: Return from the Procedure: At each CALL instruction, the IP and CS of the next instruction is pushed onto stack, before the control is transferred to the procedure. At the end of the procedure, the RET instruction must be executed. When it is executed, the previously stored content of IP and CS along with flags are retrieved into the CS, IP and flag registers from the stack and the execution of the main program continues further. In case of a FAR procedure the current contents of SP points to IP and CS at the time of return. While in case of a NEAR procedure, it points to only IP. Depending on the byte of procedure and the SP contents, the RET instruction is of four types:

i. Return within a segment.
ii. Return within a segment adding 16-bit immediate displacement to the SP contents.
iii. Return intersegment.
iv. Return intersegment adding 16-bit immediate displacement to the SP contents.

INT N: Interrupt Type N: In the interrupt structure of 8086/8088, 256 interrupts are defined corresponding to the types from 00H to FFH. When an INT N instruction is executed, the TYPE byte N is multiplied by 4 and the contents of IP and CS of the interrupt service routine will be taken from the hexadecimal multiplication (N * 4) as offset address and 0000 as segment address. In other words, the multiplication of type N by 4 (offset) points to a memory block in 0000 segment, which contains the IP and CS values of the interrupt service routine. For the execution of this instruction, the IP must be enabled.

Ex: The INT 20H will find out the address of the interrupt service routine follows:

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

Pointer to IP and CS of the ISR is 0000:0080H
The arrangement of CS and IP addresses of the ISR in the interrupt rector table is as follows.

INTO: Interrupt on overflow: This is executed, when the overflow flag OF is set. The new contents of IP an CS are taken from the address 0000:0000 as explained in INT type instruction. This is equivalent to a type 4 instruction.

JMP: Unconditional Jump: This instruction unconditionally transfer the control of execution to the specified address using an 8-bit or 16-bit displacement (intrasegment relative, short or long) or CS:IP (intersegment direct for) No flags are affected by this instruction. Corresponding to the three methods of specifying jump address, the JUMP instruction has the following three formats.

JUMP	DISP 8-bit		Intrasegment, relative, near jump
JUMP	DISP 16-bit	DISP 16-bit	Intrasegment, relative, For jump
JUMP	IP(LB) IP(UB)	CS(LB) CS(UB)	Intersegment, direct, jump

IRET: Return from ISR: When interrupt service routine is to be called, before transferring control to it, the IP, CS and flag register are stored on to the stack to indicate the location from where the execution is to be continued, after the ISR is executed. So, at the end of each ISR, when IRET is executed, the valuesof IP, CS and flags are retrieved from the stack to continue the execution of the main program. The stack is modified accordingly.

LOOP: Loop unconditionally: This instruction executes the part of the program from the label or address specified in the instruction up to the loop instruction, CX number of times. At each iteration, CX is decremented automatically, in other words, this instruction implements DECREMENT counter and JUMP IF NOT ZERO structure.

Ex:

```
         MOV CX,0005H ; Number of times in CX
         MOV BX, 0FF7H ; Data to BX
Label    MOV AX, CODE1
         OR BX,AX
         AND DX,AX
         LOOP Label
```

The execution proceeds in sequence, after the loop is executed, CX number of times. IF CX is already 00H, the execution continues sequentially. No flags are affected by this instruction.

Conditional Branch Instructions:

LOOPE/LOOPZ	LOOPNE/LOOPNZ

When these instructions are executed, they transfer execution control to the address specified relatively in the instruction, provided the condition in the opcode is satisfied, otherwise, the execution continues sequentially. The conditions, here means the status of the condition code flags. These type of instructions don't affect any flags. The address has to be specified in the instruction relatively in terms of displacement, which must lie within – 80H to 7FH (or –128 to 127) bytes from the address of the branch instruction. In other words, only short jumps can be implemented using conditional branch instructions. A label may represent the displacement, if it has within the above-specified range.

The different 8086/8088 conditional branch instructions and their operations are listed in Table1

SL.No	**Mnemonic**	**Displacement**	**Operation**
1	JZ/JE	Label	Transfer execution control to address 'Label', if ZF=1
2	JNZ/JNE	Label	Transfer execution control to address 'Label', if ZF=0
3	JS	Label	Transfer execution control to address 'Label', if SF=1
4	JNS	Label	Transfer execution control to address 'Label', if SF=0
5	JO	Label	Transfer execution control to address 'Label', if OF=1
6	JNO	Label	Transfer execution control to address 'Label', if OF=0
7	JP/JPE	Label	Transfer execution control to address 'Label', if PF=1
8	JNP	Label	Transfer execution control to address 'Label', if PF=0
9	JB/JNAE/JC	Label	Transfer execution control to address 'Label', if CF=1
10	JNB/JNE/JNC	Label	Transfer execution control to address 'Label', if CF=0
11	JBE/JNA	Label	Transfer execution control to address 'Label', if CF=1 or ZF=1
12	JNBE/JA	Label	Transfer execution control to address 'Label', if CF=0 or ZF=0
13	JL/JNGE	Label	Transfer execution control to address 'Label', if neither SF=1 nor OF=1
14	JNL/JGE	Label	Transfer execution control to address 'Label', if neither SF=0 nor OF=0
15	JNE/JNC	Label	Transfer execution control to address

			'Label', if ZF=1or neither SF nor OF is 1
16	JNLE/JE	Label	Transfer execution control to address 'Label', if ZF=0 or at least any are of SF & OF is 1

Table:1 Conditional branch instructions.

8. Flag Manipulation and Processor Control Instructions:

These instructions control the functioning of the available hardware inside the processor chip.

These are categorized into 2 types:

a) flag manipulation instructions

b) Machine control instructions.

The flag manipulation instructions directly modify same of the flags of 8086.

The flag manipulation instructions and their functions are as follows:

CLC – clear carry flag
CMC – Complement carry flag
STC – Set carry flag
CLD – clear direction flag
STD - Set direction flag
CLI – clear interrupt flag
STI – Set interrupt flag

These instructions modify the carry (CF), Direction (DF) and interrupt (IF) flags directly. The DF and IF, which may be the processor operation; like interrupt responses and auto increment or auto-decrement modes. Thus the respective instructions may also be called as machine or processor control instructions. The other flags can be modified using POPF and SAHF instructions, which are termed as data transfer instructions. No direct instructions are available for modifying the status flags except carry flags. The machine control instructions don't require any operational.

The **machine control instructions** supported by 8086/8088 are listed as follows along with their functions:

WAIT	– Wait for Test input pin to go low
HLT	– Halt the processor
NOP	– No operation
ESC	– Escape to external device like NDP
LOCK	– Bus lock instruction prefix.

<u>ASSEMBLER DIRECTIVES</u>

- Assembler directives are the commands to the assembler that direct the assembly process.
- They indicate how an operand is treated by the assembler and how assembler handles the program.
- They also direct the assembler how program and data should arrange in the memory.
- ALP's are composed of two type of statements.

(i) The instructions which are translated to machine codes by assembler.

(ii) The directives that direct the assembler during assembly process, for which no machine code is generated.

1. **ASSUME:** Assume logical segment name.

The ASSUME directive is used to inform the assembler the names of the logical segments to be assumed for different segments used in the program .In the ALP each segment is given name.

Syntax: ASSUME segreg:segname,…segreg:segname

Ex: ASSUME CS:CODE

ASSUME CS:CODE,DS:DATA,SS:STACK

2. **DB:** Define Byte

The DB directive is used to reserve byte or bytes of memory locations in the available memory.

Syntax: Name of variable DB initialization value.

Ex: MARKS DB 35H,30H,35H,40H

NAME DB "VARDHAMAN"

3. **DW:** Define Word

The DW directive serves the same puposes as the DB directive,but it now makes the assembler reserve the number of memory words(16-bit) instead of bytes.

Syntax: variable name DW initialization values.

Ex: WORDS DW 1234H,4567H,2367H

WDATA DW 5 Dup(522h)

(or) Dup(?)

4. **DD:** Define Double:

The directive DD is used to define a double word (4bytes) variable.

Syntax: variablename DD 12345678H

Ex: Data1 DD 12345678H

5. **DQ:** Define Quad Word

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

Syntax: Name of variable DQ initialize values.

Ex: Data1 DQ 123456789ABCDEF2H

6. **DT:** Define Ten Bytes

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

Syntax: Name of variable DT initialize values.

Ex: Data1 DT 123456789ABCDEF34567H

7. **END:** End of Program

The END directive marks the end of an ALP. The statement after the directive END will be ignored by the assembler.

8. **ENDP:** End of Procedure

The ENDP directive is used to indicate the end of procedure. In the AL programming the subroutines are called procedures.

Ex: Procedure Start

:

Start ENDP

9. ENDS: End of segment

The ENDS directive is used to indicate the end of segment.

Ex: DATA SEGMENT

    :

    DATA ENDS

10.EVEN: Align on Even memory address

The EVEN directives updates the location counter to the next even address.

Ex: EVEN

    Procedure Start

        :

    Start ENDP

- The above structure shows a procedure START that is to be aligned at an even address.

11.EQU: Equate

The directive EQU is used to assign a label with a value or symbol.

Ex: LABEL EQU 0500H

    ADDITION EQU ADD

12.EXTRN: External and public

- The directive EXTRN informs the assembler that the names, procedures and labels declared after this directive have been already defined in some other AL modules.
- While in other module, where names, procedures and labels actually appear, they must be declared public using the PUBLIC directive.

Ex: MODULE1 SEGMENT

PUBLIC FACT FAR

MODULE1 ENDS

MODULE2 SEGMENT

EXTRN FACT FAR

MODULE2 END

13.GROUP: Group the related segments

This directive is used to form logical groups of segments with similar purpose or type.

Ex: PROGRAM GROUP CODE, DATA, STACK

*CODE, DATA and STACK segments lie within a 64KB memory segment that is named as PROGRAM.

14.LABEL: label

The label is used to assign name to the current content of the location counter.

Ex: CONTINUE LABEL FAR

The label CONTINUE can be used for a FAR jump, if the program contains the above statement.

15.LENGTH: Byte length of a label

This is used to refer to the length of a data array or a string

Ex : MOV CX, LENGTH ARRAY

16.LOCAL: The labels, variables, constant or procedures are declared LOCAL in a module are to be used only by the particular module.

Ex : LOCAL a, b, Data1, Array, Routine

17.NAME: logical name of a module

The name directive is used to assign a name to an assembly language program module. The module may now be refer to by its declared name.

Ex : Name “addition”

18.OFFSET: offset of a label

When the assembler comes across the OFFSET operator along with a label, it first computing the 16-bit offset address of a particular label and replace the string ‘OFFSET LABEL’ by the computed offset address.

Ex : MOV SI, offset list

19.ORG: origin

The ORG directive directs the assembler to start the memory allotment for the particular segment, block or code from the declared address in the ORG statement.

Ex: ORG 1000H

20.PROC: Procedure

The PROC directive marks the start of a named procedure in the statement.

Ex: RESULT PROC NEAR

ROUTINE PROC FAR

21.PTR: pointer

The PTR operator is used to declare the type of a label, variable or memory operator.

Ex : MOV AL, BYTE PTR [SI]

MOV BX, WORD PTR [2000H]

22.SEG: segment of a label

The SEG operator is used to decide the segment address of the label, variable or procedure.

Ex : MOV AX, SEG ARRAY

MOV DS, AX

23.SEGMENT: logical segment

The segment directive marks the starting of a logical segment

```
Ex: CODE SEGMENT
       :
CODE ENDS
```

24.SHORT: The SHORT operator indicates to the assembler that only one byte is required to code the displacement for jump.

Ex : JMP SHORT LABEL

25.TYPE: The TYPE operator directs the assembler to decide the data type of the specified label and replaces the TYPE label by the decided data type.

For word variable, the data type is 2.

For double word variable, the data type is 4.

For byte variable, the data type is 1.

```
Ex : STRING DW 2345H, 4567H
MOV AX, TYPE STRING
AX=0002H
```

26.GLOBAL: The labels, variables, constants or procedures declared GLOBAL may be used by other modules of the program.

Ex : ROUTINE PROC GLOBAL.

27.FAR PTR: This directive indicates the assembler that the label following FAR PTR is not available within the same segment and the address of the label is of 32-bits i.e 2-bytes of offset followed by 2-bytes of segment address.

Ex : JMP FAR PTR LABEL

28.NEAR PTR: This directive indicates that the label following NEAR PTR is in the same segment and needs only 16-bit i.e 2-byte offset to address it

```
Ex : JMP NEAR PTR LABEL
CALL NEAR PTR ROUTINE
```

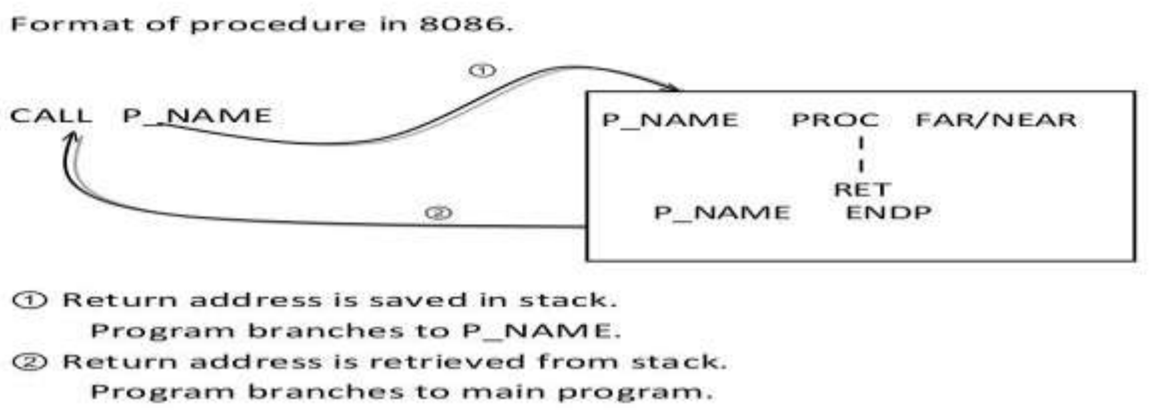
<u>Procedures and Macros:</u>

- When we need to use a group of instructions several times throughout a program there are two ways we can avoid having to write the group of instructions each time we want to use them.
  1. One way is to write the group of instructions as a separate **procedure.**
  2. Another way we can use **macros.**

Procedures:

- The procedure is a group of instructions stored as a separate program in the memory and it is called from the main program whenever required using CALL instruction.
- For calling the procedure we have to store the return address (next instruction address followed by CALL) onto the stack.
- At the end of the procedure RET instruction used to return the execution to the next instruction in the main program by retrieving the address from the top of the stack.

- Machine codes for the procedure instructions put only once in memory.
- The procedure can be defined anywhere in the program using assembly directives PROC and ENDP.


Format of procedure in 8086.
CALL P_NAME
①
②
P_NAME PROC FAR/NEAR
RET
P_NAME ENDP
① Return address is saved in stack.
Program branches to P_NAME.
② Return address is retrieved from stack.
Program branches to main program.

- **The four major ways of passing parameters to and from a procedure are:**
  1. In registers
  2. In dedicated memory location accessed by name
  3 .With pointers passed in registers
  4. With the stack
- The type of procedure depends on where the procedure is stored in the memory.
- If it is in the same code segment where the main program is stored the it is called near procedure otherwise it is referred to as far procedure.
- For near procedure CALL instruction pushes only the IP register contents on the stack, since CS register contents remains unchanged for main program.
- But for Far procedure CALL instruction pushes both IP and CS on the stack.

Syntax:

Procedure name PROC near

instruction 1

instruction 2

RET

Procedure name ENDP

Example:

near procedure:

ADD2 PROC near

ADD AX,BX

RET

ADD2 ENDP

far procedure:

Procedures segment

Assume CS : Procedures

ADD2 PROC far

ADD AX,BX

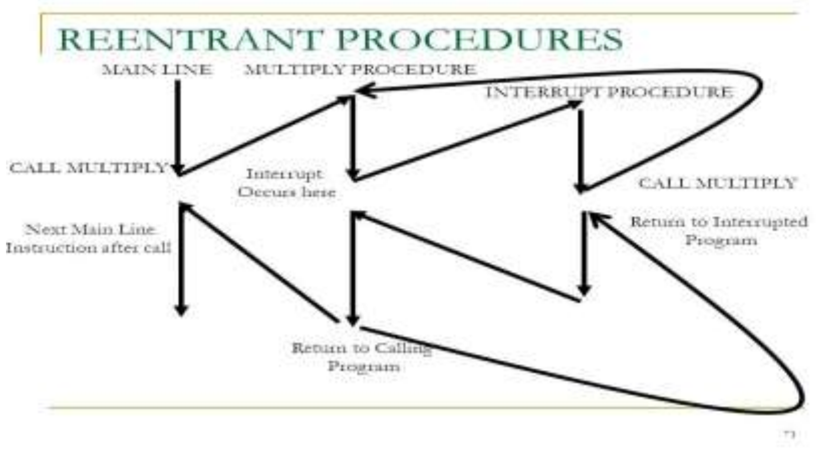
RET ADD2 ENDP

Procedures ends

- Depending on the characteristics the procedures are two types
  1. Re-entrant Procedures
  2. Recursive Procedures

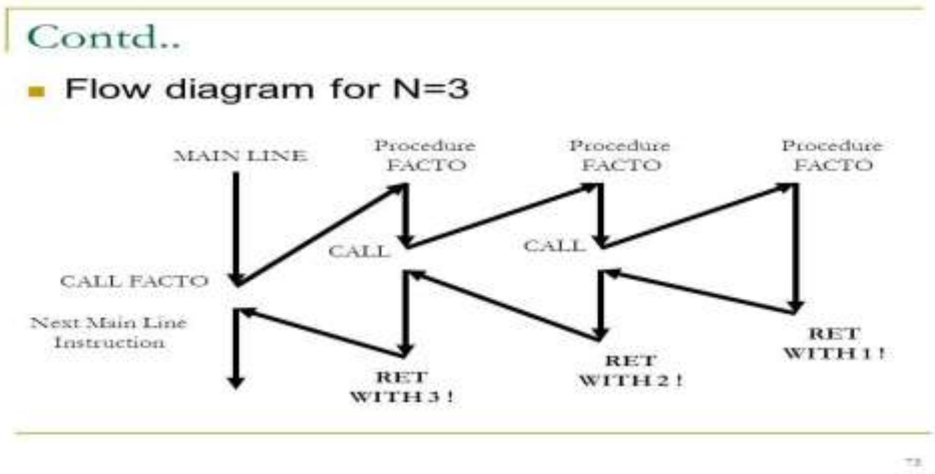
Reentrant Procedures

- The procedure which can be interrupted, used and “reentered” without losing or writing over anything.



Recursive Procedure

- A recursive procedure is procedure which calls itself.



-

ALP for Finding Factorial of number using procedures

```
CODE SEGMENT
ASSUME CS:CODE
START: MOV AX,7
CALL FACT
MOV AH,4CH
INT 21H
FACT PROC NEAR
MOV BX,AX
DEC BX
BACK: MUL BX
DEC BX
JNZ BACK
RET
ENDP
CODE ENDS
END START
```

Macros:

- A **macro** is a group of repetitive instructions in a program which are codified only once and can be used as many times as necessary.
- A macro can be defined anywhere in program using the directives **MACRO** and **ENDM**
- Each time we call the macro in a program, the assembler will insert the defined group of instructions in place of the call.
- The assembler generates machine codes for the group of instructions each time the macro is called.
- Using a macro avoids the overhead time involved in calling and returning from a procedure.

Syntax of macro:

```
macroname MACRO

instruction1

instruction2

.

.

ENDM
```

- **Example:**

```
Read MACRO
 mov ah,01h
int 21h
ENDM
```

```
Display MACRO
mov dl,al
 Mov ah,02h
int 21h
 ENDM
```

ALP for Finding Factorial of number using procedures

```
CODE SEGMENT
ASSUME CS:CODE
      FACT MACRO
       MOV BX,AX
       DEC BX
      BACK: MUL BX
       DEC BX
       JNZ BACK
      ENDM
START: MOV AX,7
       FACT
       MOV AH,4CH
       INT 21H
       CODE ENDS
       END START
```

<u>Advantage of Procedure and Macros:</u>

Procedures:

Advantages

- The machine codes for the group of instructions in the procedure only have to be put once.

Disadvantages

- Need for stack
- Overhead time required to call the procedure and return to the calling program.

Macros:

Advantages

- Macro avoids overhead time involving in calling and returning from a procedure.

Disadvantages

- Generating in line code each time a macro is called is that this will make the program take up more memory than using a procedure.

Differences between Procedures and Macros:

PROCEDURES	**MACROS**
Accessed by CALL and RET mechanism during program execution	Accessed by name given to macro when defined during assembly
Machine code for instructions only put in memory once	Machine code generated for instructions each time called
Parameters are passed in registers, memory locations or stack	Parameters passed as part of statement which calls macro
Procedures uses stack	Macro does not utilize stack
A procedure can be defined anywhere in program using the directives PROC and ENDP	A macro can be defined anywhere in program using the directives MACRO and ENDM
Procedures takes huge memory for CALL (3 bytes each time CALL is used) instruction	Length of code is very huge if macro's are called for more number of times

8086 MEMORY INTERFACING:

- Most the memory ICs are byte oriented i.e., each memory location can store only one byte of data.
- The 8086 is a 16-bit microprocessor, it can transfer 16-bit data.
- So in addition to byte, word (16-bit) has to be stored in the memory.
- To implement this , the entire memory is divided into two memory banks: Bank0 and Bank1.
- Bank0 is selected only when A0 is zero and Bank1 is selected only when BHE' is zero.
- A0 is zero for all even addresses, so Bank0 is usually referred as even addressed memory bank.
- BHE' is used to access higher order memory bank, referred to as odd addressed memory bank.

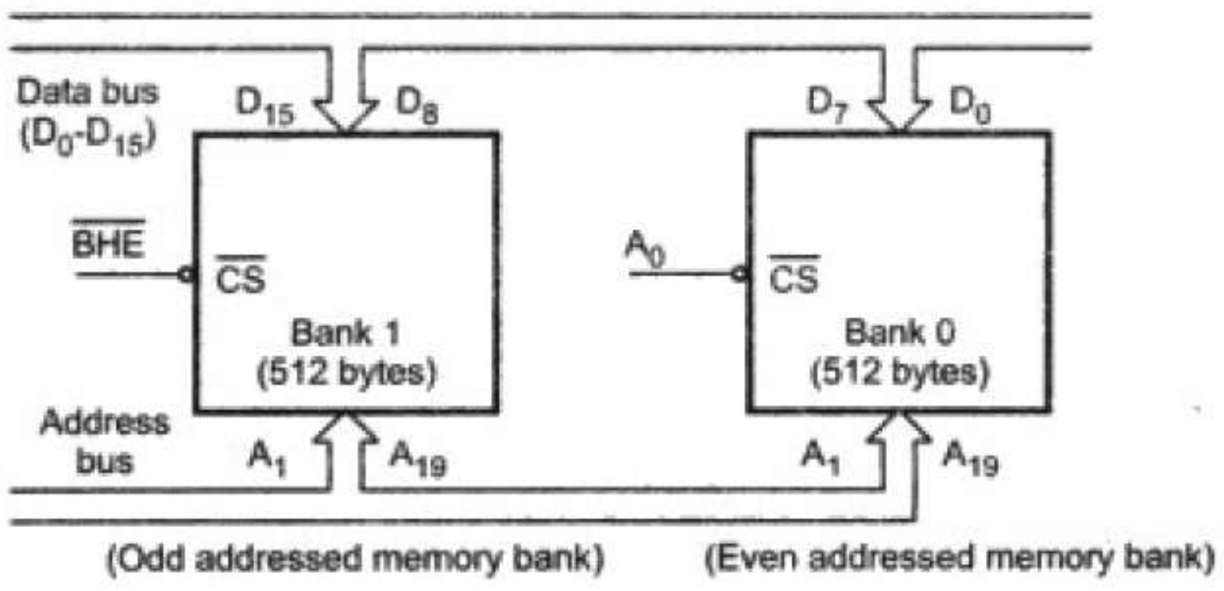


Fig. 5.2 Memory interfacing

- Every microprocessor based system has a memory system.
- Almost all systems contain two basic types of memory, read only memory (ROM) and random access memory (RAM) or read/write memory.
- ROM contains system software and permanent system data such as lookup tables, IVT..etc.
- RAM contains temporary data and application software.
- ROMs/PROMs/EPROMs are mapped to cover the CPU's reset address, since these are non-volatile.
- When the 8086 is reset, the next instruction is fetched from the memory location FFFF0H.
- So in the 8086 system the location FFFF0H must be in ROM location.

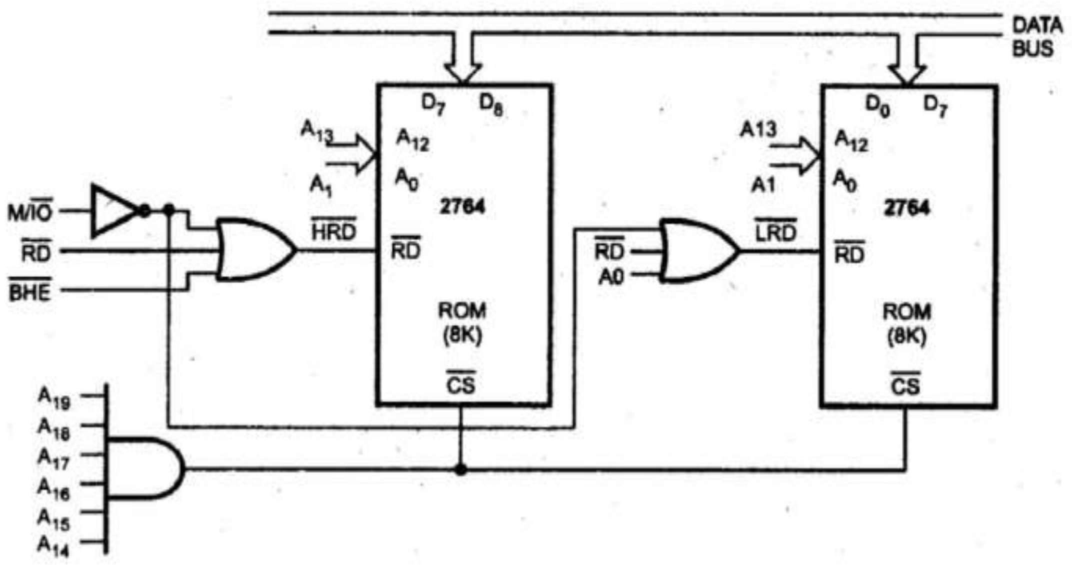
Address Decoding Techniques

1. Absolute decoding
2. Linear decoding
3. Block decoding

1. Absolute Decoding:

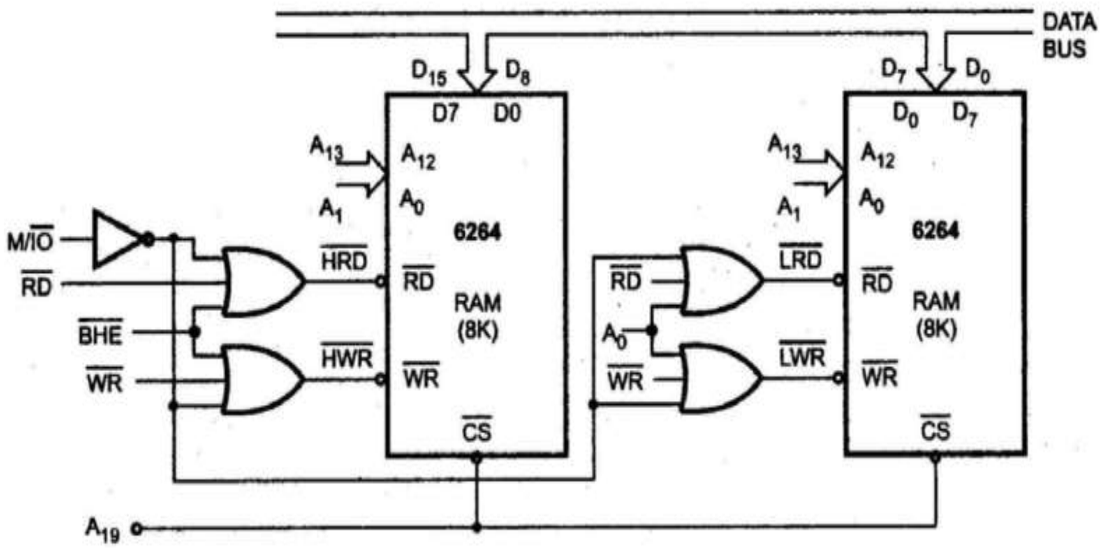
- In the absolute decoding technique the memory chip is selected only for the specified logic level on the address lines: no other logic levels can select the chip.
- Below figure the memory interface with absolute decoding. Two 8K EPROMs (2764) are used to provide even and odd memory banks.
- Control signals BHE and A0 are use to enable output of odd and even memory banks respectively. As each memory chip has 8K memory locations, thirteen address lines are required to address each locations, independently.

- All remaining address lines are used to generate an unique chip select signal. This address technique is normally used in large memory systems.



Linear Decoding:

In small system hardware for the decoding logic can be eliminated by using only required number of addressing lines (not all). Other lines are simple ignored. This technique is referred as linear decoding or partial decoding. Control signals BHE and Ao are used to enable odd and even memory banks, respectively. Figure shows the addressing of 16K RAM (6264) with linear decoding.

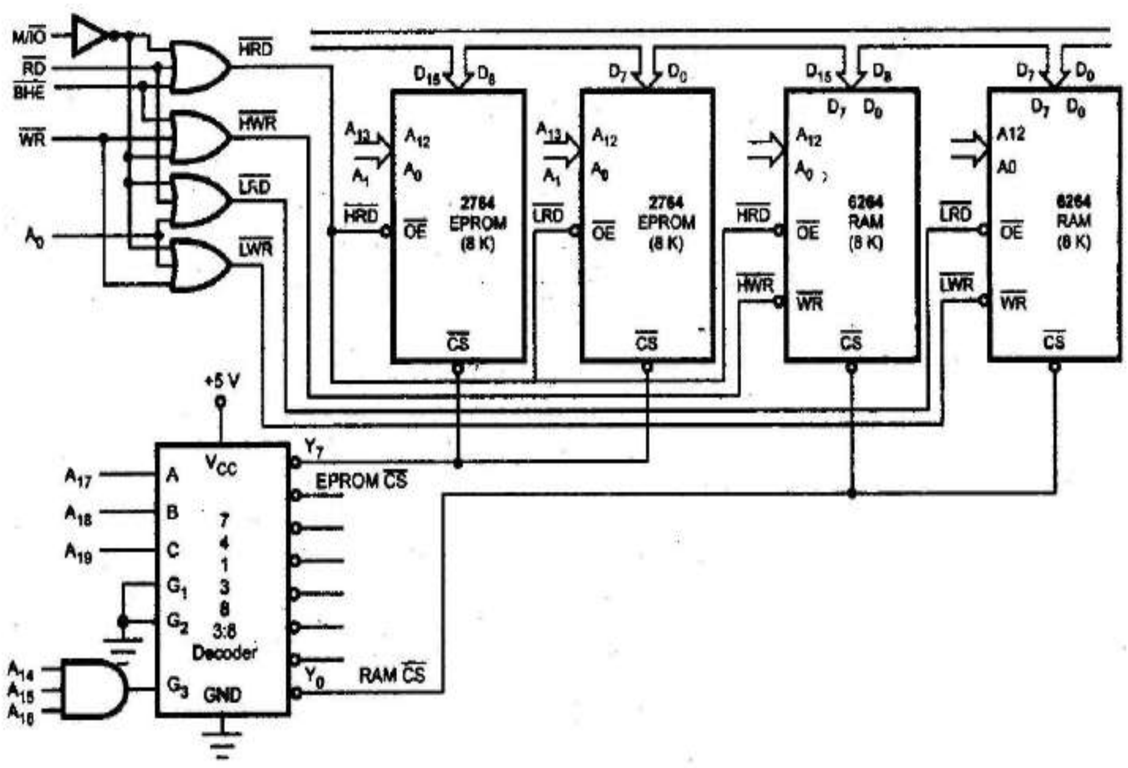


The address line A19 is used to select the RAM chips. When A19 is low, chip is selected, otherwise it is disabled. The status of A14 to A18 does not affect the chip selection logic. This gives you multiple addresses (shadow addresses). This technique reduces the cost of decoding circuit, but it gas drawback of multiple addresses.

Block Decoding:

In a microcomputer system the memory array is often consists of several blocks of memory chips. Each block of memory requires decoding circuit. To avoid separate decoding for each memory block special decoder IC is used to generate chip select signal for each block.

Figure shows the Block decoding technique using 74138, 3:8 decoder



Interfacing RAM, ROM, EPROM to 8086:

- The general procedure of static memory interfacing with 8086

1. Arrange the available memory chips so as to obtain 16-bit data bus width.
    - The upper 8-bit bank is called ‘odd address memory bank’.
    - The lower 8-bit bank is called ‘even address memory bank’.
2. Connect available memory address lines of memory chips with those of the microprocessor and also connect the RD and WR inputs to the corresponding processor control signals.
3. Connect the 16-bit data bus of memory bank with that of the microprocessor 8086.
4. The remaining address lines of the microprocessor, BHE and A_0 are used for decoding the required chip select signals for the odd and even memory banks. The CS of memory is derived from the output of the decoding circuit.

Problem 1:

Interface two 4Kx8 EPROM and two 4Kx8 RAM chips with 8086. Select suitable maps.

Solution:

We know that, after reset, the IP and CS are initialized to form address FFFF0H. Hence, this address must lie in the EPROM. The address of RAM may be selected anywhere in the 1MB address space of 8086, but we will select the RAM address such that the address map of the system is continuous.

Memory Map Table

Address	A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A09	A08	A07	A06	A05	A04	A03	A02	A01	A00
FFFFF H	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
EPROM								8K X 8												
FE000 H	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
FDFFF H	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
RAM								8K X 8												
FC000 H	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Total 8K bytes of EPROM need 13 address lines A0-A12 (since z13 = 8K).

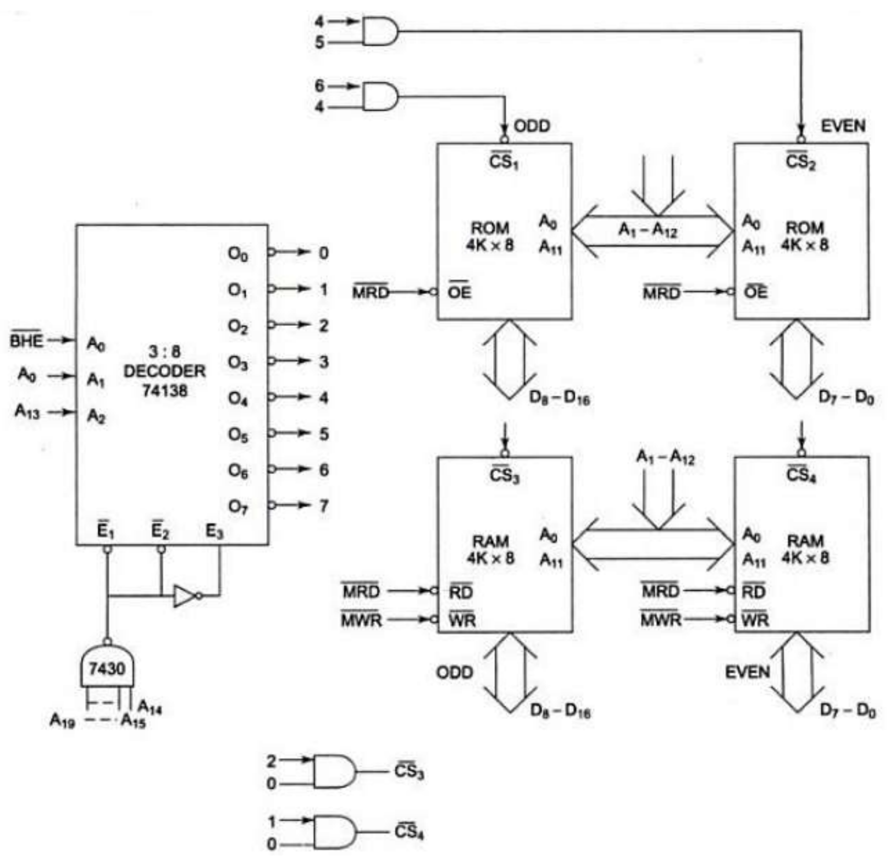
Address lines A13 - A19 are used for decoding to generate the chip select.

The $\overline{BHE}$ signal goes low when a transfer is at odd address or higher byte of data is to be accessed.

Let us assume that the latched address, $\overline{BHE}$ and demultiplexed data lines are readily available for interfacing.

The memory system in this problem contains in total four 4K x 8 memory chips.

The two 4K x 8 chips of RAM and ROM are arranged in parallel to obtain 16-bit data bus width. If A0 is 0, i.e., the address is even and is in RAM, then the lower RAM chip is selected indicating 8-bit transfer at an even address. If A0 is i.e., the address is odd and is in RAM, the $\overline{BHE}$ goes low, the upper RAM chip is selected, further indicating that the 8-bit transfer is at an odd address. If the selected addresses are in ROM, the respective ROM chips are selected. If at a time A0 and $\overline{BHE}$ both are 0, both the RAM or ROM chips are selected, i.e., the data transfer is of 16 bits. The selection of chips here takes place as shown in table below.

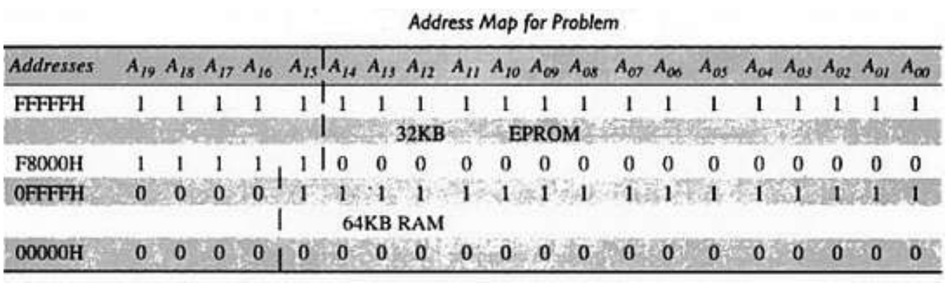


Memory Chip Selection Table:

Decoder I/P --> Address/$\overline{BHE}$ -->	A2 A13	A1 A0	A0 $\overline{BHE}$	Selection/ Comment
Word transfer on D0 - D15	0	0	0	Even and odd address in RAM
Byte transfer on D7 - D0	0	0	1	Only even address in RAM
Byte transfer on D8 - D15	0	1	0	Only odd address in RAM
Word transfer on D0 - D15	1	0	0	Even and odd address in RAM
Byte transfer on D7 - D0	1	0	1	Only even address in RAM
Byte transfer on D8 - D15	1	1	0	Only odd address in ROM

Problem2: Design an interface between 8086 CPU and two chips of 16K×8 EPROM and two chips of 32K×8 RAM. Select the starting address of EPROM suitably. The RAM address must start at 00000 H.

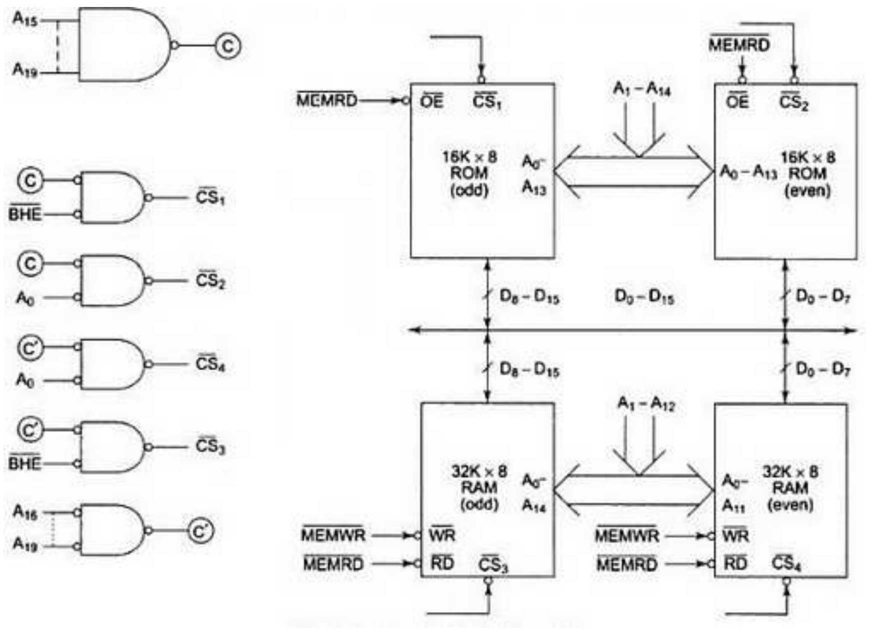
Solution: The last address in the map of 8086 is FFFFF H. after resetting, the processor starts from FFFF0 H. hence this address must lie in the address range of EPROM.



Address Map for Problem

Addresses	A_{19}	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_{09}	A_{08}	A_{07}	A_{06}	A_{05}	A_{04}	A_{03}	A_{02}	A_{01}	A_{00}
FFFFFH	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
							32KB			EPROM										
F8000H	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0FFFFH	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
						64KB RAM														
00000H	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

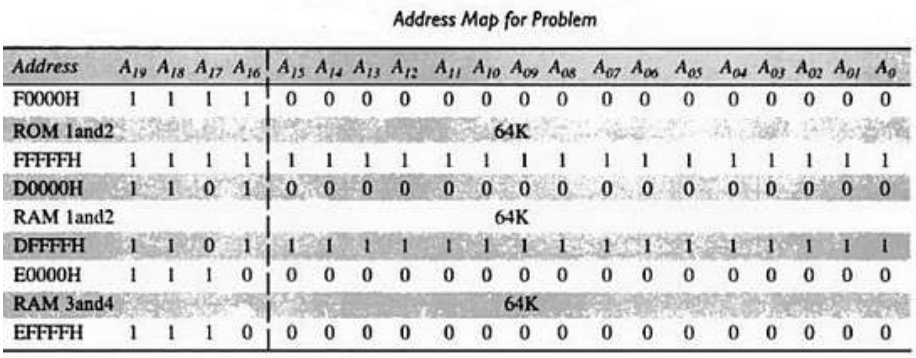
It is better not to use a decoder to implement the above map because it is not continuous, i.e. there is some unused address space between the last RAM address (0FFFF H) and the first EPROM address (F8000 H). Hence the logic is implemented using logic gates.



Problem3: It is required to interface two chips of 32K×8 ROM and four chips of 32K×8 RAM with 8086, according to following map.

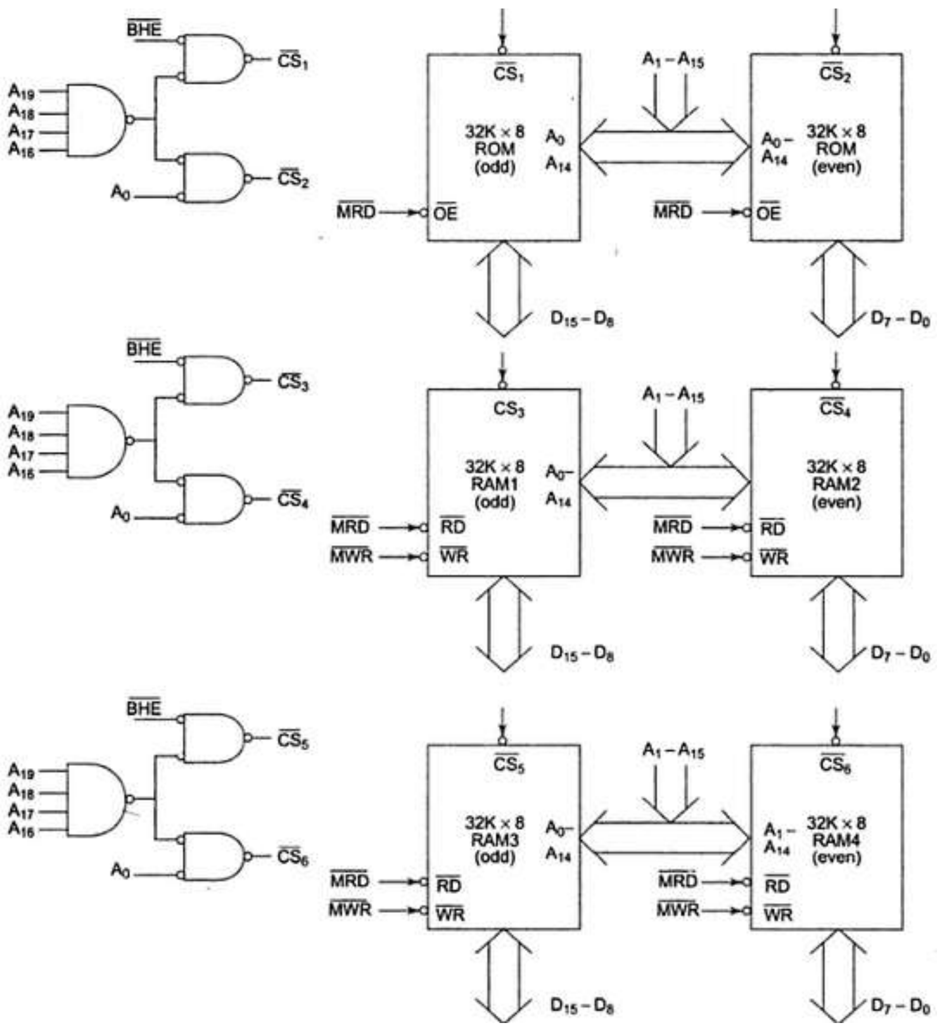
ROM 1 and ROM 2 F0000H - FFFFFH, RAM 1 and RAM 2 D0000H - DFFFFH, RAM 3 and RAM 4 E0000H - EFFFFH. Show the implementation of this memory system.

Solution:



Address Map for Problem

Address	A_{19}	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_{09}	A_{08}	A_{07}	A_{06}	A_{05}	A_{04}	A_{03}	A_{02}	A_{01}	A_0
F0000H	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ROM 1and2											64K									
FFFFFH	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
D0000H	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
RAM 1and2											64K									
DFFFFH	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
E0000H	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
RAM 3and4											64K									
EFFFFH	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0



Methods of Interfacing I/O Devices

Memory Mapping	IO mapping
1. 20-bit addresses are provided for IO devices.	1. 8-bit or 16-bit address are provided for IO devices
2. The IO ports or peripherals can be treated like memory locations and so all instructions related to memory can be used for data transfer.	2. Only IN and OUT instructions can be used for data transfer between IO device and the processor.
3. In memory mapped ports, the data can be moved from any register to port and vice versa	3. In IO mapped ports, the data transfer can take only between the accumulator and the ports
4. When memory mapping is used for IO devices, the full memory address space cannot be used for addressing memory.	4. When IO mapping is used for IO devices, then the full address space can be used for addressing memory.