Anjali

1. What do you mean by addressing mode. Describe the addressing modes in 8086.

- Addressing modes describe the type of operands and the way they are accessed for executing an instruction.

a) Immediate
- data appears in the instruction as 8bit or 16bit
- Eg:- MOV AX, 0005H

b) Direct
- the 16 bit memory offset address where the data resides is there in the instruction in square bracket.

Eg:- MOV AX, [5000H]
The effective address is 10H × DS + 5000.

c) Register
- the data will be there in the register.

Eg:- MOV AX, BX

d) Register Indirect.
- the 16 bit memory offset address where the data resides will be there in the register

Eg:- MOV AX, [BX]

BX 5555
DS 2000

effective address =
20000H + 5555H.
= 25555H

e) Indexed.
- the 16 bit offset address where data resides will be there in Index Registers.

Eg:- MOV AX, [SI]

SI 2020
DS 2000

effective address = 20000 + 2020
= 22020

Scanned by CamScanner

Scanned by CamScanner

f) Register relative:-

- The 16 bit offset value and a displacement value from this offset is also specified in the instruction.

Eg: MOV AX, 50H[BX]

BX 2020
DS 2000

effective address. $10_H \times 2000_H + 2020_H + 0050_H$.

$= 22070_H$

g) Based Indexed.

- The 16 bit offset value and a displacement value from this offset is specified in a register in the instruction.

Eg:- MOV AX, [BX][SI]

BX 2020
DS 2000
SI 0050

effective address:- $10_H \times 2000_H +$
$2020_H +$
0050_H

22070_H.

h) Relative Based Indexed.

- 16 bit offset in register, displacement in Index register and an immediate displacement in register is specified in the instruction.

Eg:- MOV AX, 10_H [BX][SI].

BX 2020
DS 2000
SI 0050

effective address:- $10_H \times 2000_H +$
$2020_H +$
$0050_H +$
0010_H

22080_H

Scanned by CamScanner

i) Addressing modes for control transfer instruction :-

- the addressing mode for control transfer instruction depends on whether the data lies in same segment or in other segment.
- If the location lies in same segment it is called Intra segment mode.
- If the location lies in different segment it is called Inter segment mode.
- If the control is to be transfered to 8 bit displacement from IP content it is short jump.
- If the control is to be transfered to 16 bit displacement from IP Content it is long jump.
- If the displacement is directly specified in the instruction it is direct mode.
- If the displacement is kept in a register and that register is specified in the instruction it is indirect mode.

1) Intrasegment Direct.

- Same segment.
- Displacement address in instruction.

Eg:- JMP SHORT LABEL

LABEL is an 8 bit address.

2) Intrasegment Indirect.

- Same segment.
- displacement is in the register specified.

Eg:- JMP [BX]

Scanned by CamScanner

3) Intersegment Direct

- different Segment.
- Address in the instruction.

Eg:- JMP 5000H : 2000H.

4) Intersegment Indirect.

- different Segment.
- Address is there in the memory location specified in the instruction. There will be two address :- The base address will be in CS and Offset in IP. It is specified as 4 bytes in memory location.

IP (LSB)
IP (MSB)
CS (LSB)
CS (MSB)

Eg:- JMP [2000_H]

If DS = 1000.

$1000_H \times 10_H + 2000_H$

12000	30
12001	30
12002	00
12003	50

effective address where address is kept = 12000.

The instruction execution will jump to

$5000_H \times 10_H + 3030_H = 53030_H$.

Scanned by CamScanner

2. INSTRUCTION SET of 8086.

There are 8 types of Instructions.

1) Data Copy / Transfer
2) Arithmetic & Logical
3) Branch
4) Loop.
5) Machine Control
6) Flag Manipulation.
7) Shift & Rotate.
8) String.

1) Data Copy / Transfer.

- To move data from one location to another

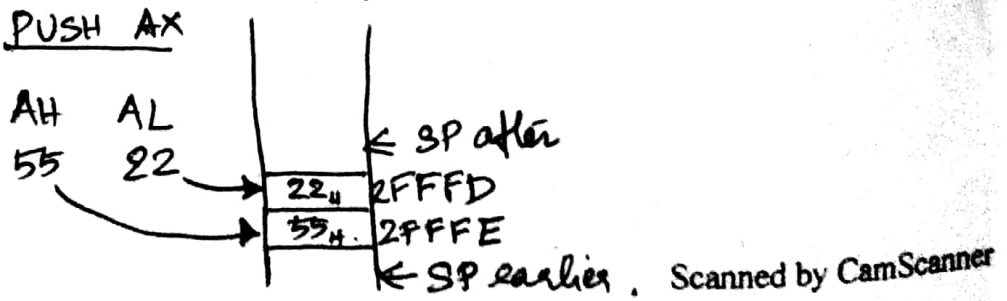
a) MOV - To move data from register / memory to another register / Memory

Eg:- MOV AX, 5000_H
MOV AX, BX
MOV AX, [SI]
MOV AX, [2000_H]
MOV AX, 50_H[BX]

b) PUSH - To push to Stack.

To push a register / memory content to the stack. Before pushing Stack Pointer is decremented by 2.

PUSH AX

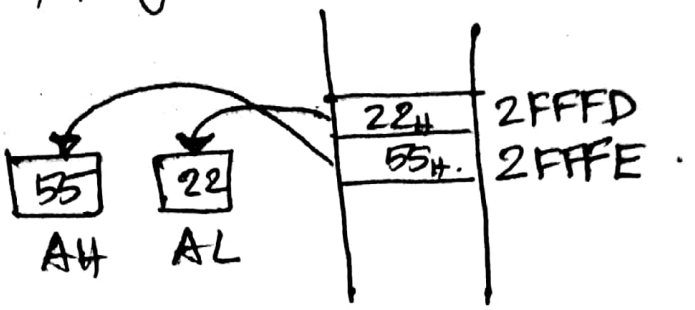


Scanned by CamScanner

Scanned by CamScanner

Eg:- PUSH AX
PUSH DS
PUSH [5000$_H$]

c) POP : Pop from stack to a register or memory location specified. SP is incremented by 2 after poping.



Eg:- POP AX
POP DS
POP [5000$_H$]

d) XCHG : Exchange
— Exchange the content of source and destination.

Eg:- XCHG [5000H], AX
XCHG BX, AX.

e) IN : Input the Port.
- For reading Input Port.
- Address of input port will be in DX
- Default destination registers are
  AL - for 8 bit reading
  AX - for 16 bit reading.

Eg:- IN AL, 30$_H$; Read from addres port 30$_H$ to AL
IN AX, DX ; Read from address port specified in DX

Scanned by CamScanner

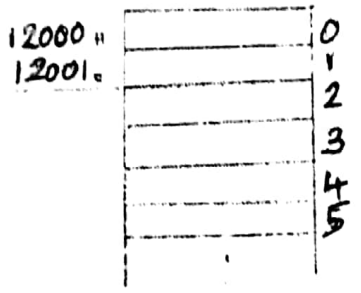
```
MOV  DX,  0800H
IN   AX,  DX
```

f) OUT : write to the port address.

- Address of the port will be in DX.
- The data to be transfered is kept in AL (8 bit) or AX (16 bit).
- Data to even address is transfered through $D_0 - D_7$ and odd address through $D_8 - D_{15}$.

Eg:-
```
OUT   03H, AL
OUT   DX, AX
MOV   DX, 0300H
OUT   DX, AX.
```

g) XLAT : To find the code of the pressed key using lookup table.



Eg:- The code corresponding to key pressed in 7 segment display is stored in a lookup table starting from address 12000_H.
So DS = 1000
offset address = 2000.

```
MOV  AL, NUMBER  (Number pressed is stored in AL)
MOV  BX, 2000H.  (Offset address of lookup table is
                  kept in BX)
XLAT.            (The corresponding code will be in AL)
```

h) LEA : Load effective address.

Eg:- LEA BX, Label (the address corresponding to the label is kept in BX)

Scanned by CamScanner

Scanned by CamScanner

LDS/LES : Load Pointer to DS/ES

Eg:- LDS BX, 5000H
Content from 5000 & 5001 loaded to BX
Content from 5002 & 5003 loaded to DS.

LAHF : Load from lower Byte of Flag Register.

SAHF : Store AH to lower byte of Flag Register.

PUSHF : Push flag to Stack.

POPF : Pop flag from Stack.

2) Arithmetic Instruction.

a) ADD : Add.

- Add immediate data, contents of register or memory.
- Both contents cannot be in memory
- Contents of segment register cannot be added.

Eg:- ADD AX, 0001_H
ADD AX, BX
ADD AX, [SI]
ADD AX, [5000_H].

b) ADC : Add with Carry.

- Same as ADD, but adds carry created from previous instruction.

Eg:- ADC AX, 0001_H.

c) INC : Increment.

- Content of Register or memory location is incremented by 1 (Except flag)

Eg:- INC AX

Scanned by CamScanner

Scanned by CamScanner

d) DEC : Decrement.

- Decrement the content of Memory or register by 1.

Eg :- DEC AX.

e) SUB :- Subtract.

- the source operand is subtracted from the destination operand.

Eg :- SUB AX, 0100_H.
SUB AX, BX
SUB AX, $[5000_H]$

f) SBB :- Subtract with Borrow.

- Subtract the source operand and borrow flag (CF) from previous operation, from the destination.

Eg :- SBB AX, 0100_H.
SBB AX, BX
SBB AX, $[5000_H]$

g) CMP : Compare.

- Source and destination is compared.
- For this Source Operand is subtracted from destination. Result is not stored any where, but flags are affected. If Source is greater CF = 1.

Eg :- CMP BX, 0100_H
CMP AX, 0100_H.
CMP $[5000_H]$, 0100_H
CMP BX, [SI]
CMP BX, CX

Scanned by CamScanner

8) ASCII Adjust after addition . AAA

- It converts results in AL to unpacked decimal digits while adding the ascii coded operands.

Set AH = 0
If (Lower nibble of AL is less than 9)
    If (AF = 0)
    The higher nibble of AL is 0.
    If (AF = 1)
    Then AH++
    The higher nibble of AL is 0.
    The lower nibble of AL = AL+6
Else
    Then higher nibble of AL = 0.
    AH++
    Lower nibble of AL = AL + 06
    AF & CF = 1

Eg:- 1. AL 37 ← Ascii of 07
BL 39 ← Ascii of 09

ADD AL, BL

AL = 6B (69, 80

AAA AL = 11

B = 1011 +
6 = 0110
1 0001
1 1

2) AAS = ASCII adjust after subtraction
- Converts the results in AL to unpacked decimal format.
- Operations are similar AAA except the subtraction of 06 from AL.

3) AAM - ASCII adjust after multiplication.
- Convert the result to unpacked decimal after multiplication.

Scanned by CamScanner

Scanned by CamScanner

Eg:- MOV AL, 04.
MOV BL, 09
MUL BL : AH AL = 24_H.
AAM.

After AAM the result 24_H is there in AL.
This is divided by 10 and Quotient is kept in AH and remainder in AL.

Then AH = 03.
AL = 06.

$$A_H \overline{)\,2\,4_H} \quad 03$$
$$1\,E_H$$
$$06$$

1	B	15	1F
2	C	16	20
3	D	17	21
4	E	18	22
5	F	19	23
6	10	1A	24
7	11	1B	
8	12	1C	
9	13	1D	
A	14	1E	

j) AAD - ASCII adjust before division.
- It has totally different operation.
- It has to be used just before multiplication.
- It converts the two unpacked BCD in AH and AL to equivalent binary in AL.

Eg:- AX = 05-08....
After AAD AL = 3A

3A is the hexadecimal of 58.

k) DAA - Decimal adjust accumulator.
- Converts the result of addition of two packed BCD number to a valid BCD number.
- The result will be in AL.
- If lower nibble is greater than 9, 06 is added. If at that time AF is set or upper nibble is greater than 9, 60 is added to upper nibble.

Scanned by CamScanner

Scanned by CamScanner

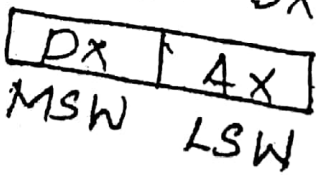
l) DAS :- Decimal adjust after subtraction.

- Convert the result of two packed BCD number to a valid BCD.
- If lower nibble of AL (result) is greater than 09, 06 is subtracted from it. If a Carry is generated or higher nibble is greater than 09_H, then 60 is subtracted from it.

m) NEG :- Negate.

- Keeps 2's compliment.
- For this the content is subtracted from Zero. If OF=1 then negate operation failed.

n) MUL :- Multiplies unsigned byte or Word by the content of AL.
Result is stored in DX & AX



Scanned by CamScanner

Scanned by CamScanner

Eg:- MUL BH.
MUL CX.

2) IMUL: Signed multiplication.

3) CBW: Convert signed byte to word.
- Sign bit is copied to a MSB of AX.

4) CWD: Convert signed word to double word.
- Sign bit is copied to DX and result will be in AX.

5) DIV :- Unsigned division.
- Divisor can be in register or memory
- Divident is AX.
- Result - Quotient in AL
Remainder in AH.
- If 32 bit divident then it is kept in

DX	&	AX
MSW		LSW

6) IDIV : Signed division.

3) LOGICAL INSTRUCTIONS

AND: logical AND.

Scanned by CamScanner

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 operands should be a register or a memory operand. Both the operands cannot be memory locations or immediate operands. An immediate operand cannot be a destination operand. The examples of this instruction are as follows:

Example 2.37

```
1.AND AX, 0008H
2.AND AX, BX
3.AND AX, [5000H]
4.AND [5000H], DX
```

If the content of AX is 3F0FH, the first example instruction will carry out the operation as given below. The result 3F9FH will be stored in the AX register.

```
0 0 1 1     1 1 1 1     0 0 0 0     1 1 1 1     = 3F0F H [AX]
↓ ↓ ↓ ↓     ↓ ↓ ↓ ↓     ↓ ↓ ↓ ↓     ↓ ↓ ↓ ↓       AND
0 0 0 0     0 0 0 0     0 0 0 0     1 0 0 0     = 0008 H
-------------------------------------------------------
0 0 0 0     0 0 0 0     0 0 0 0     1 0 0 0     = 0008 H [AX]
```

The result 0008H will be in AX.

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. The examples are as follows:

Example 2.38

```
1.OR AX, 0098H
2.OR AX, BX
3.OR AX, [5000H]
4.OR [5000H], 0008H
```

The contents of AX are say 3F0FH, then the first example instruction will be carried out as given below.

```
0 0 1 1     1 1 1 1     0 0 0 0     1 1 1 1     = 3F0F H
↓ ↓ ↓ ↓     ↓ ↓ ↓ ↓     ↓ ↓ ↓ ↓     ↓ ↓ ↓ ↓       OR
0 0 0 0     0 0 0 0     1 0 0 1     1 0 0 0     = 0098 H
-------------------------------------------------------
0 0 1 1     1 1 1 1     1 0 0 1     1 1 1 1     = 3F9F H
```

Thus the result 3F9FH will be stored in the AX register.

NOT: Logical Invert The NOT instruction complements (inverts) the contents of an operand register or a memory location, bit by bit. The examples are as follows:

Example 2.39

```
NOT AX
NOT [5000H]
```

If the content of AX is 200FH, the first example instruction will be executed as shown.

```
  AX      = 0 0 1 0     0 0 0 0     0 0 0 0     1 1 1 1
Invert      ↓ ↓ ↓ ↓     ↓ ↓ ↓ ↓     ↓ ↓ ↓ ↓     ↓ ↓ ↓ ↓
            1 1 0 1     1 1 1 1     1 1 1 1     0 0 0 0
```

Scanned by CamScanner

Result

In AX = 0 F F 0

The result DFF0H will be stored in the destination register AX.

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

Example 2.40

```
1.XOR    AX, 0098H
2.XOR    AX, BX
3.XOR    AX, [5000H]
```

If the content of AX is 3F0FH, then the first example instruction will be executed as explained. The result 3F97H will be stored in AX.

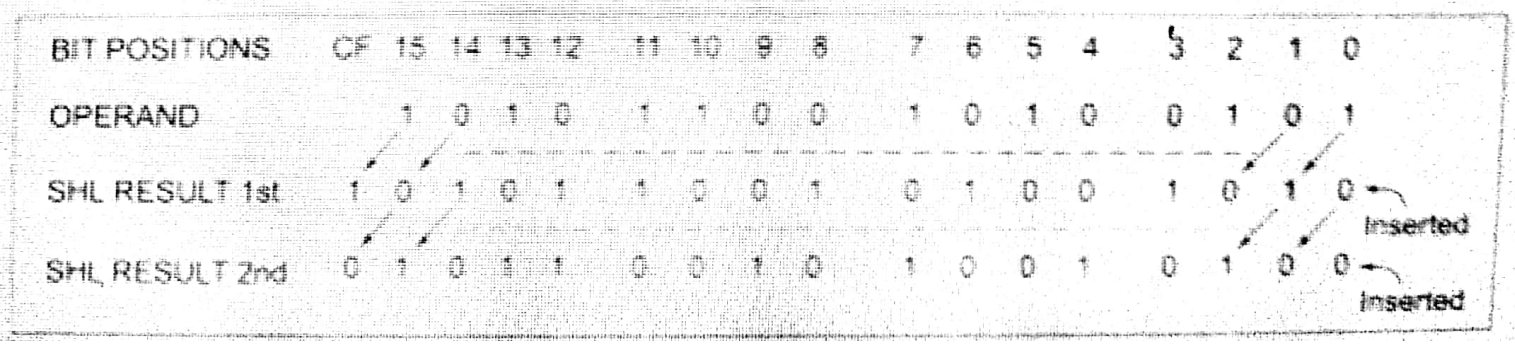
```
AX = 3F0FH =    0 0 1 1    1 1 1 1    0 0 0 0    1 1 1 1
      XOR       ↓ ↓ ↓ ↓    ↓ ↓ ↓ ↓    ↓ ↓ ↓ ↓    ↓ ↓ ↓ ↓
   0098H =      0 0 0 0    0 0 0 0    1 0 0 1    1 0 0 0
AX = Result =   0 0 1 1    1 1 1 1    1 0 0 1    0 1 1 1
        = 3F97H
```

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 are 1, else the result bit is reset to 0. The result of this ANDing operation is not available for further use, but flags are affected. The affected flags are OF, CF, SF, ZF and PF. The operands may be registers, memory or immediate data. The examples of this instruction are as follows:

Example 2.41

```
1.TEST    AX, BX
2.TEST    [0500], 06H
3.TEST    [BX] [DI], CX
```

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 a memory location but cannot be an immediate data. All flags are affected depending upon the result. Figure 2.7 explains the execution of this instruction. It is to be noted here that the shift operation is through carry flag.



Scanned by CamScanner

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. Figure 2.8 explains execution of this instruction. This instruction shifts the operand through the carry flag.

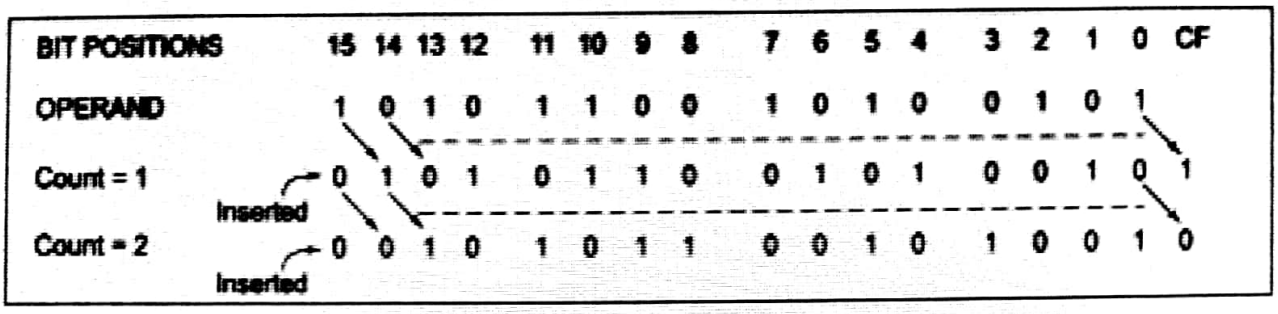


Fig. 2.8 *Execution of SHR Instruction*

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. It inserts the most significant bit of the operand in the newly inserted positions. The result is stored in the destination operand. Figure 2.9 explains execution of the instruction. All the condition code flags are affected. This shift operation shifts the operand through the carry flag.

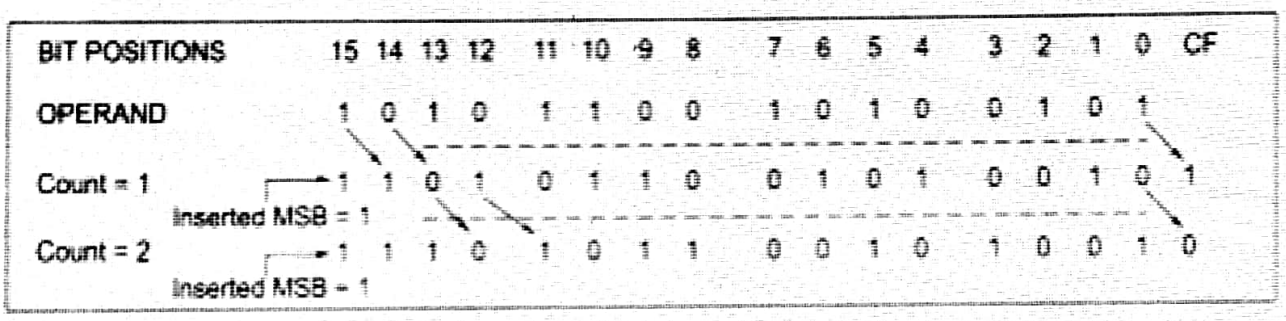


Fig. 2.9 *Execution of SAR Instruction*

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

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 cannot be an immediate operand. Figure 2.10 explains the operation. The destination operand may be a register (except a segment register) or a memory location.

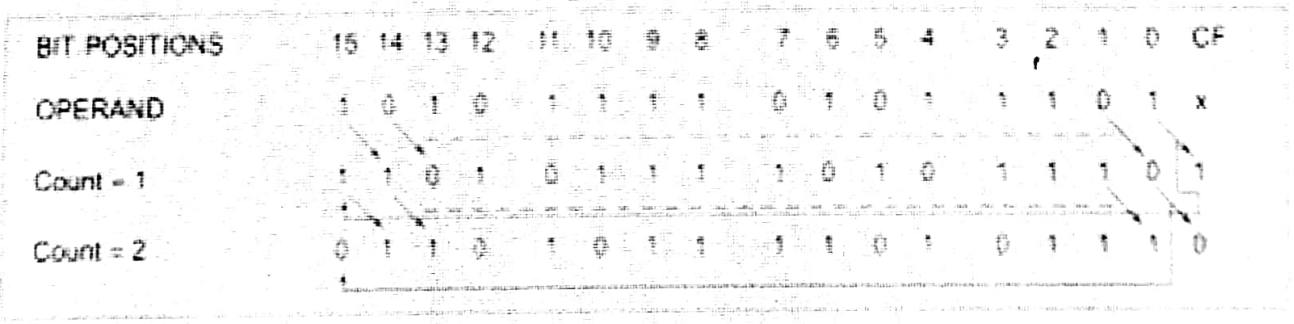


Fig. 2.10 *Execution of ROR Instruction*

Scanned by CamScanner

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 in this rotate operation. The operand may be a register or a memory location. Figure 2.11 explains the operation.

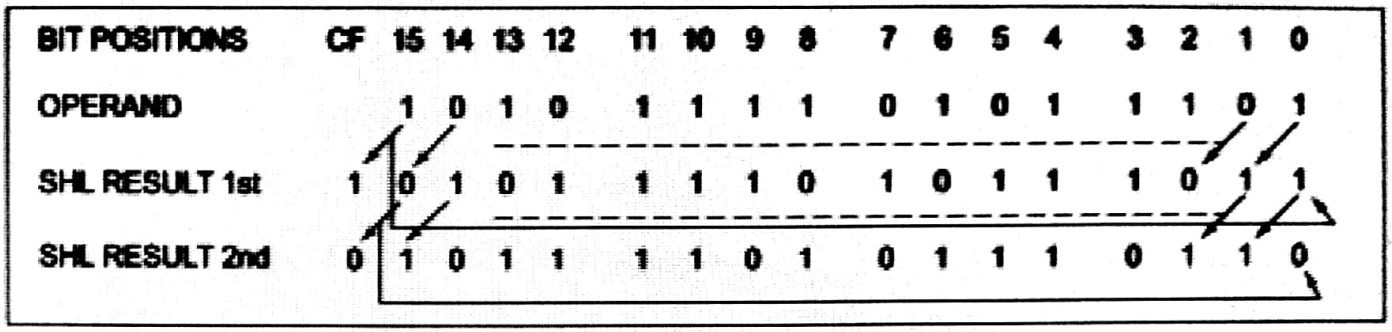


Fig. 2.11 *Execution of ROL Instruction*

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 a memory location. Figure 2.12 explains the operation.

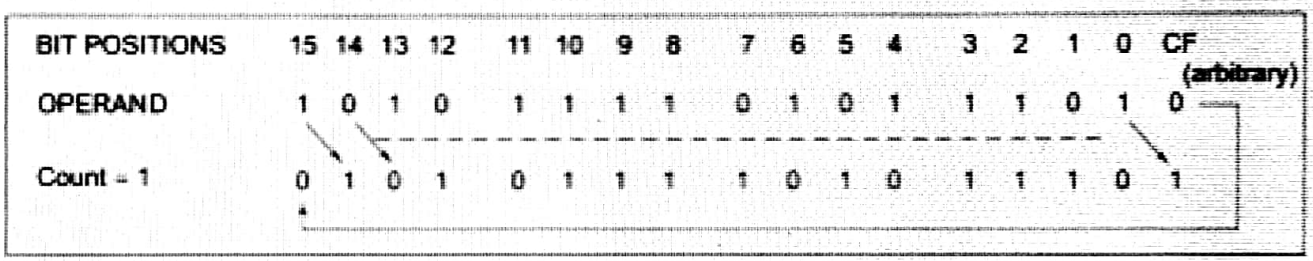


Fig. 2.12 *Execution of RCR Instruction*

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. Figure 2.13 explains the operation.

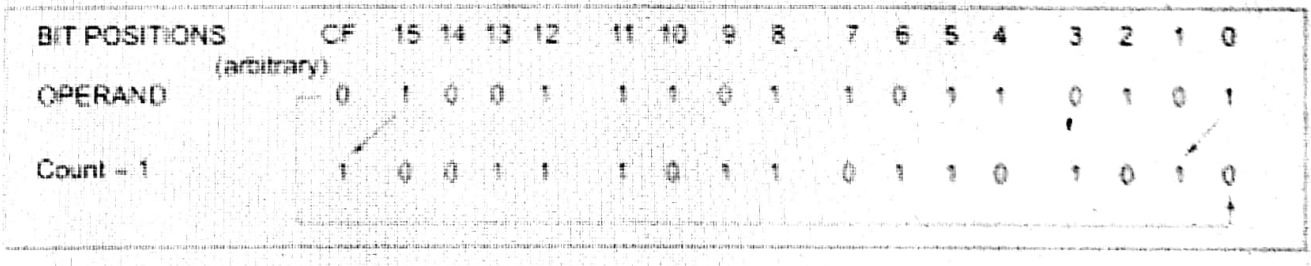


Fig. 2.13 *Execution of RCL Instruction*

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

Scanned by CamScanner

STRING MANIPULATION INSTRUCTIONS.

String is a set of bytes stored in consecutive memory locations. To refer a string we need start or end address and length of the string. The length is stored in CX register. The characters are addressed by using Index Registers. Index Registers are automatically incremented (DF=0) and decremented (if DF=1) during string operations.

REP:- Repeat Instruction Prefix.
- Used along with other instruction
- That suffixed instruction is repeated until CX=0

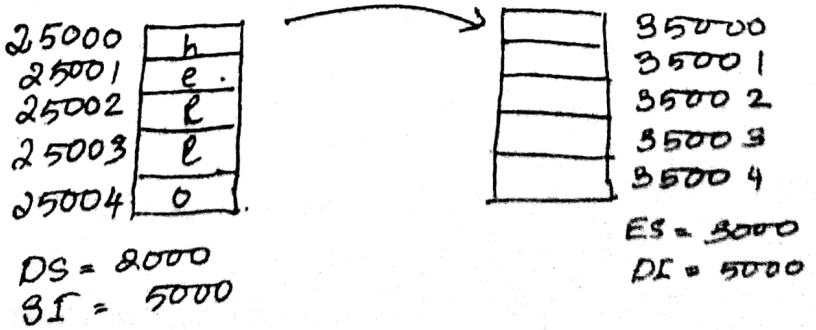
REPE - Repeat Till Equal.

REPZ - Repeat Till Zero.

REPNE - Repeat Till Not Equal.

REPNZ - Repeat Till Not Zero.

MOVSB/MOVSW :- Move String byte or String word
- To move a string from one location to another
- The offset of source location (Starting address of String) is stored in SI register.
- The offset of destination is stored in DI.



Scanned by CamScanner

CMPS : Compare String byte or String word.
- To compare two strings.
- Length of the String will be in CX
- DS : SI point to first String and ES : DI point to Second String.
- Flags are affected.
- Compare each character and stops when a mismatch is found.

SCAS :- Scan String byte or string word.
- Scan for a character in AL or AX in the String specified from location ES : DI.
- If a match is found execution stops and Zero flag is set.

LODS :- Load String Byte or String word.
Store AL/AX register with content of String pointed by SI / DS : SI

STOS :- Store String Byte or String word.
Store the AL/AX register content to location pointed by ES : DI.

CONTROL TRANSFER OR BRANCHING INSTRUCTION

Unconditional Branch :- Branching without checking any condition.

CALL :- Unconditional call.
To call a subroutine from a main program. The call can be intersegment and intrasegment.

Scanned by CamScanner

Scanned by CamScanner

CMPS : Compare String byte or String word.
- To compare two strings
- Length of the string will be in CX
- DS : SI point to first string and ES : DI point to second string.
- Flags are affected
- Compare each character and stops when a mismatch is found.

SCAS :- Scan String byte or string word
- Scan for a character in AL or AX in the string specified from location ES : DI.
- If a match is found execution stops and zero flag is set.

LODS :- Load String Byte or String word.
Store AL/AX register with content of string pointed by SI / DS : SI

STOS :- Storage String Byte or String word.
Store the AL/AX register content to location pointed by ES : DI.

CONTROL TRANSFER OR BRANCHING INSTRUCTION

Unconditional Branch :- Branching without checking any condition.

CALL :. Unconditional call.
To call a subroutine from a main program. The call can be intersegment and intra Segment.

Scanned by CamScanner

Scanned by CamScanner

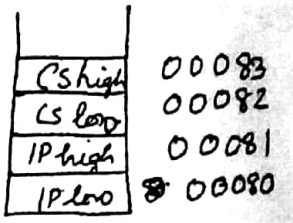
The displacement can be
- 32 bit displacement.
- 16 bit displacement.

RET :- Return from procedure.
- When procedures are called the CS and IP is stored in Stack.
- When it is return from intersegment procedure both CS & IP is fetched. Otherwise only IP.

INT N :- Software interrupt.
- N is the type of the interrupt.
- Type no: can be used to fetch the location of Interrupt Service Routine.
- Type No: × 4 will give correct offset of ISR. Base address is 00000.
- Eg:- INT 20H.

Offset = 20 × 4 = 80_H.

Effective address = 00000 + 80
= 00080



INTO : Interrupt on overflow
- If OF flag is set.

JMP : For unconditional jump.
- displacement can be 8 bit or 16 bit.

Eg:- JMP 80_H.
JMP 1080_H.

IRET : Return from Interrupt Service Routine.

LOOP : Loop unconditionally.

Scanned by CamScanner

Scanned by CamScanner

2.3.7 Conditional Branch Instructions

When these instructions are executed, execution control is transferred to the address specified relatively in the instruction, provided the condition implicit in the opcode is satisfied. If not the execution continues sequentially. The conditions, here, means the status of condition code flags. These type of instructions do not affect any flag. 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 lies within the above specified range. The different 8086/8088 conditional branch instructions and their operations are listed in Table 2.3.

Table 2.3 *Conditional Branch Instructions*

	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/JAE/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.	JLE/JNG	Label	Transfer execution control to address 'Label', if ZF=1 or neither SF nor OF is 1.
16.	JNLE/JE	Label	Transfer execution control to address 'Label', if ZF=0 or at least any one of SF and OF is 1(Both SF and OF are not 0).

Scanned by CamScanner

2.3.8 Flag Manipulation and Processor Control Instructions

These instructions control the functioning of the available hardware inside the processor chip. These are categorized into two types; (a) flag manipulation instructions and (b) machine control instructions. *The flag manipulation instructions directly modify some of the flags of 8086. The machine control instructions control the bus usage and execution.* The flag manipulation instructions and their functions are listed in Table 2.5.

Table 2.5 *Flag Manipulation Instructions*

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 modified using the flag manipulation instructions, further control the processor operation; like interrupt responses and autoincrement or autodecrement modes. Thus, the respective instructions may also be called machine or processor control instructions. The other flags can be modified using POPF and SAHF instructions, which are termed as data transfer instructions, in this text. No direct instructions, are available for modifying the status flags except carry flag.

The machine control instructions supported by 8086 and 8088 are listed in Table 2.6 along with their functions. They do not require any operand.

Table 2.6 *Machine Control Instructions*

WAIT	-	Wait for Test input pin to go low
HLT	-	Halt the processor
NOP	-	No operation
ESC	-	Escape to external device like NDP (numeric co-processor)
LOCK	-	Bus lock instruction prefix

Scanned by CamScanner

3. What are assembler directives? Give examples for assembler directives. (3 marks)

or.

Explain the assembler directives available for 8086 assembly language programming. (10 marks)

- Assembler is a system program that converts assembly language program to machine code
- Assembler directives are another set of mnemonics that give direction to assembler in the machine translation of the ALP

It gives information about the label values, storage space in memory, logical names of the segments etc. Some of the assembler directives are given below.

a) DB:- Define byte.

- Reserve Byte

Eg:-
```
RANKS   DB   01H, 02H, 03H
MSG1    DB   'HELLO'
VAL     DB   50H.
```

b) DW: DEFINE WORD

Eg:-
```
WORDS   DW   1234H, 3568H.
WDATA   DW   5 DUP (6666H)
```

This allocates 5 word spaces and initialises the word location with 6666.

c) DQ: Define quad word.

- define 4 words (8 bytes).

d) DT: Define Ten Bytes.

e) ASSUME: Assume logical segment name.

- it gives information about the names

Scanned by CamScanner

Scanned by CamScanner

of the segments used in the program.
It is write as

ASSUME CS: CODE, DS: DATA.

It informs that the segment named CODE is having the base address in CS and segment named DATA is having the base address in DS.

END: Marks end of ALP.

ENDP: End of the procedure.

Eg:-

```
PROCEDURE STAR
    :
    :
STAR ENDP
```

ENDS: End of the segment.

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

EVEN: Align on Even Memory Address.

- While starting an Assembly program, program counter initialises to next available location.
- If EVEN assembler directive is used the location counter is updated to next available even address.

```
EVEN
PROCEDURE ROOT
    :
ROOT ENDP
```

Scanned by CamScanner

Scanned by CamScanner

EQU: To assign a label with a value or symbol.

- A label is given to a value and that label can be used any where in ALP.
- During assembling this label is replaced with value.
- If a value is to be used 10 times in a program and if we try to change the value we need to update this 10 times. If we are using label, a change in label definition with EQU is enough.

Eg:- NUM EQU 0005H.

EXTRN. External and PUBLIC: Public.

EXTRN informs the assembler that the names, procedures with EXTRN label is defined in some other ALP.

Those modules where its definition is kept should be declared public by assembler directive PUBLIC.

```
MODULE1  SEGMENT
PUBLIC   FACTORIAL  FAR.
MODULE1  ENDS.
MODULE2  SEGMENT
EXTRN    FACTORIAL  FAR
MODULE 2 ENDS.
```

GROUP: Group the related segments.

- To group the segments with similar purpose or type.

Scanned by CamScanner

Scanned by CamScanner

- If declared so all these segments will lie in same 64kB memory space. So that the base address will be same for all segments

Eg:- PROGRAM GROUP CODE, DATA, STACK

ASSUME CS: PROGRAM , DS: PROGRAM , SS: PROGRAM

LABEL: Label.

- It is used to assign a name to current content of the location counter.
- When the assembly process reaches the LABEL it assign the declared label with current content of location counter.

Eg:- DD LABEL BYTE FAR

This means it is byte labla which is FAR and the current content of LC is assigned to DD.

LENGTH: Byte length of a label.

MOV CX, LENGTH ARRAY.

LOCAL: Labels, Variables, Constants or procedures declared LOCAL are to be used only by the particular module.

Eg:- LOCAL a, b, DATA, ARRAY, ROUTINE.

NAME: Logical Name of the Module.

- to assign a name to an assembly language program module.

OFFSET: Offset of a label.

- it computes the 16 bit displacement (offset) from base address of arrays, String, labels

Scanned by CamScanner
Scanned by CamScanner

and procedures.

Eg:- MOV 31, OFFSET LIST.

ORG: Origin

- It directs the assembler to allocate the memory location declared with ORG for particular segment, block or code.
- If ORG is not used, it start from 0000

PROC: Procedure.

- To mark the start of new procedure

RESULT PROC NEAR. (located within 64KB memory).

PTR: Pointer.

- to define the data type, byte or word.
- PTR is always prefixed with BYTE or WORD.
- It denotes that all elements pointed from there are of that type.

Eg:- MOV AL, BYTE PTR [SI].

SEG: To decide the segment address of the label, variable, produce procedure.

SEGMENT: Mark the starting of logical segments.

Scanned by CamScanner

Scanned by CamScanner