2.3 INSTRUCTION SET OF 8086/8088

The 8086/8088 instructions are categorised into the following main types. This section explains the function of each of the instructions with suitable examples wherever necessary.

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

(ii) **Arithmetic and Logical Instructions** All the instructions performing arithmetic, logical, increment, decrement, compare and scan 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 instructions belong to this class.

(iv) **Loop Instructions** If these instructions have REP prefix with CX used as count register, they can be used to implement unconditional and conditional loops. The LOOP, LOOPNZ and LOOPZ instructions belong to this category. These are useful to implement different loop structures.

(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 affect 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 bitwise shifting or rotation in either direction with or without a count in CX.

(viii) **String Instructions** These instructions involve various string manipulation operations like load, move, scan, compare, store, etc. These instructions are only to be operated upon the strings.

2.3.1 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 registers or other general or special purpose registers or a memory location and, another register or memory location may act as destination.

(1)

MOV : Move

→ This data transfer instructions transfer data from one register / memory location to another register / memory location.

→ The source may be one of the segment register or other general or special purpose registers or a memory location.

→ The destination may be another register or memory location.

(Note : The destination cannot be a segment register ; in case of immediate addressing mode).

Ex :- Load DS with 5000H

(1) MOV DS, 5000H
Invalid instruction

(2) MOV AX, 5000H
MOV DS, AX
valid instruction

Ex :-
MOV AX, 5000H → Immediate
MOV AX, BX → Register
MOV AX, [SI] → Indirect
MOV AX, [2000H] → Direct

MOV AX, 50H[BX]
→ Based Relative 50H displacement value.

(2) PUSH : Push to Stack

→ This instruction pushes the contents of the specified register / memory location onto 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 then stores 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 decremented stack addresses the higher byte occupies the higher address and lower byte occupies the lower address.

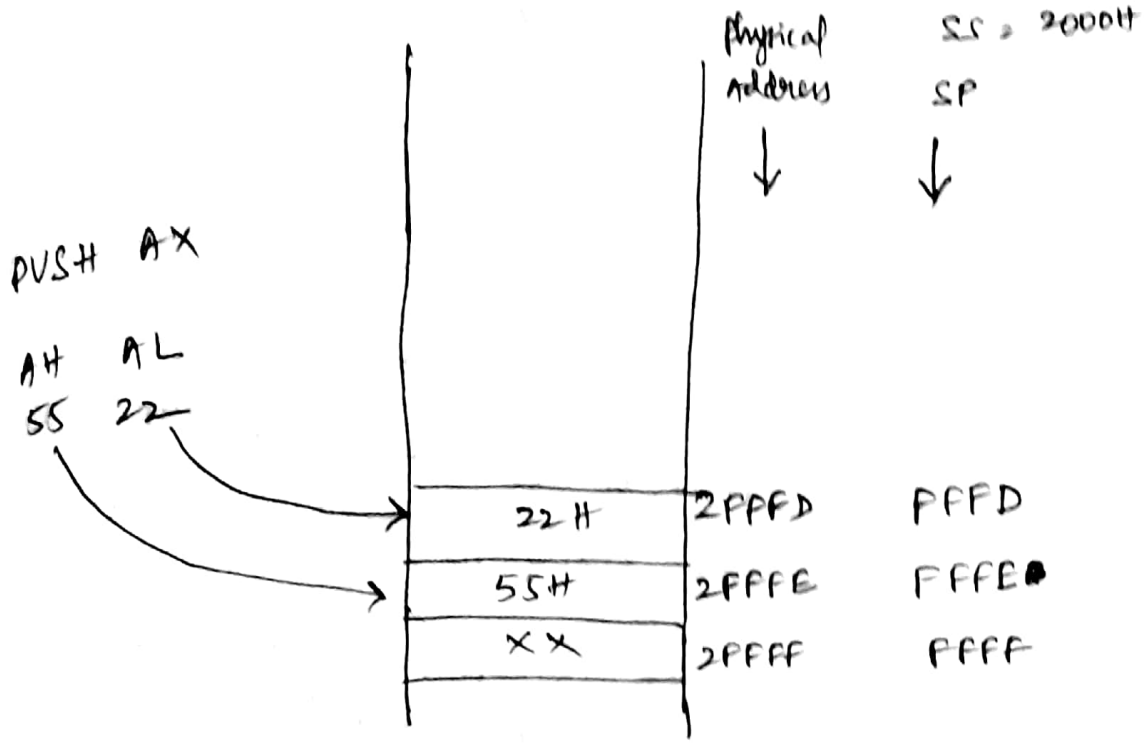


fig: Pushing Data to Stack memory.

The sequence of operation is as follows:

(1) Current stack top is already occupied so decrement SP by one, then store AH into the address pointed to by SP

(2) Further decrement SP by one and store AL into the location pointed to by SP.

Ex:- PUSH AX ; PUSH DS

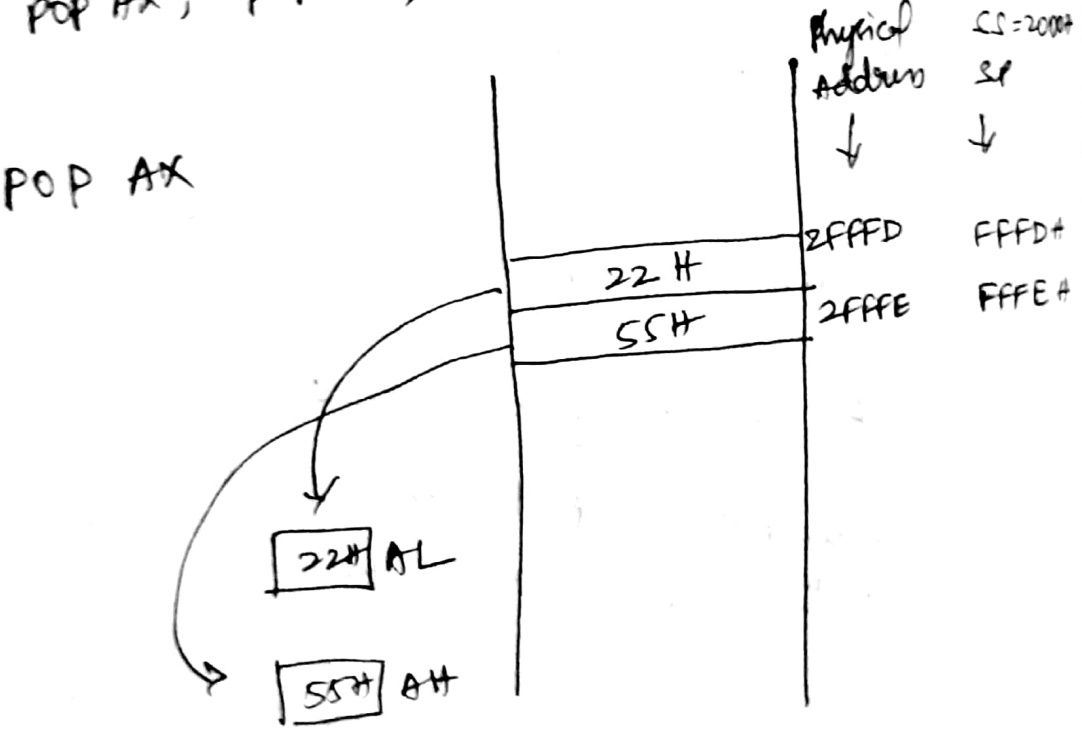
PUSH [5000H] ; contents of location 5000H and 5001H in DS are pushed on to the stack.

(3) POP : Pop from stack :

→ This instruction when executed, loads the specified register / memory location with the contents of memory location of which the address is formed using the current stack segment and stack pointer as usual.

The stack pointer is incremented by 2.

Ex:- POP AX ; POP DS ; POP [5000H]



Sequence :

(1) contents of stack top memory location stored in AL and SP ~~dec~~ incremented by one

(2) contents of memory location pointed by SP are moved into AH and SP incremented by one.

(9) XCHG : Exchange.

→ This instruction exchanges the contents of the specified source and destination operands, which may be registers or one of them can be a memory location.

→ Note : Exchange of contents of two memory locations is not permitted.

→ Note : Immediate data is also not allowed in this instruction.

Ex :-

```
XCHG [5000H], AX ; exchanges data b/w AX and memory location 5000H in DS.

XCHG BX, AX ; exchanges data b/w AX, BX registers.
```

(5) IN : Input the port

→ This instruction used for reading an input port.

→ The address of the input port may be specified in the instruction directly or indirectly.

→ AL or AX are the allowed destinations for 8 or 16 bit input operation.

→ DX is the only register (implicit) which is allowed to carry the port address. If the port address is 16 bit, it must be in DX.

Ex:-

(1) IN AL, 03H ; reads data from an 8bit port with address 03H into AL.

(2) IN AX, DX ; read data from an 16 bit port whose address is provided by DX and stores it in AX.

(3) MOV DX, 0800H ; 16 bit port address is taken in DX
IN AX, DX ; Read the content of port into AX.

(6) OUT : output to the port.

→ This instruction is used for writing into an output port.

→ The address of the output port may be specified in the instruction directly or implicitly in DX.

→ contents of AX or AX are transferred to a directly or indirectly addressed port after executing of this instruction.

→ Data to an odd addressed port is transferred on $D_8 - D_{15}$, while that to an even addressed port is transferred on $D_0 - D_7$.

→ Registers AL, AX are the allowed source operands.

→ If the port address is 16 bit, it must be in DX.

Ex:- (1) OUT 03H, AL ; data in AL is moved to a port with address 03H

(2) OUT DX, AX ; data in AX is moved to a port whose address is provided by DX

(3) MOV DX, 0300H ; 16 bit port address into DX

OUT DX, AX ; data from AX move into port with address provided by DX.

(7) XLAT : Translate.

→ This instruction is used for finding out the codes in case of code conversion problems, using look up table technique.

Ex:-

A hexadecimal key pad with 16 keys for 0 to F is interfaced with 8086 using 8255. When a key is pressed, the code of that key (0-F) is returned in AL.

For displaying the number corresponding to the pressed key on a 7-segment display device, it is required that 7-segment code corresponding to the key pressed is found out and sent to the display port.

This translation from the code of the key pressed to the corresponding seven segment code is performed using XLAT instruction.

For this purpose, one is required to prepare a look up table of codes, starting from an offset say 2000H and store 7-segment codes for 0 to F at locations 2000H to 200FH sequentially.

For executing the XLAT instruction, the code of the pressed key obtained from key board is moved in AL and the base address of the look up table containing 7-segment code is kept in BX. After executing the XLAT instruction the 7-segment code corresponding to the key pressed is returned in AL, replacing the key code which was in AL prior to the execution of XLAT.

```
MOV AX, SEG TABLE ; address of segment containing look up table is transferred into DS
MOV DS, AX
MOV AL, CODE      ; code of key pressed in AL
MOV BX, OFFSET TABLE ; offset of code lookup table in BX
XLAT              ; find equivalent code and store in AL.
```

(8) LEA: Load Effective Address

The load effective address instruction loads the effective address formed by destination operand into the specified source register.

The instruction is more useful for assembly language rather than machine language.

Ex -

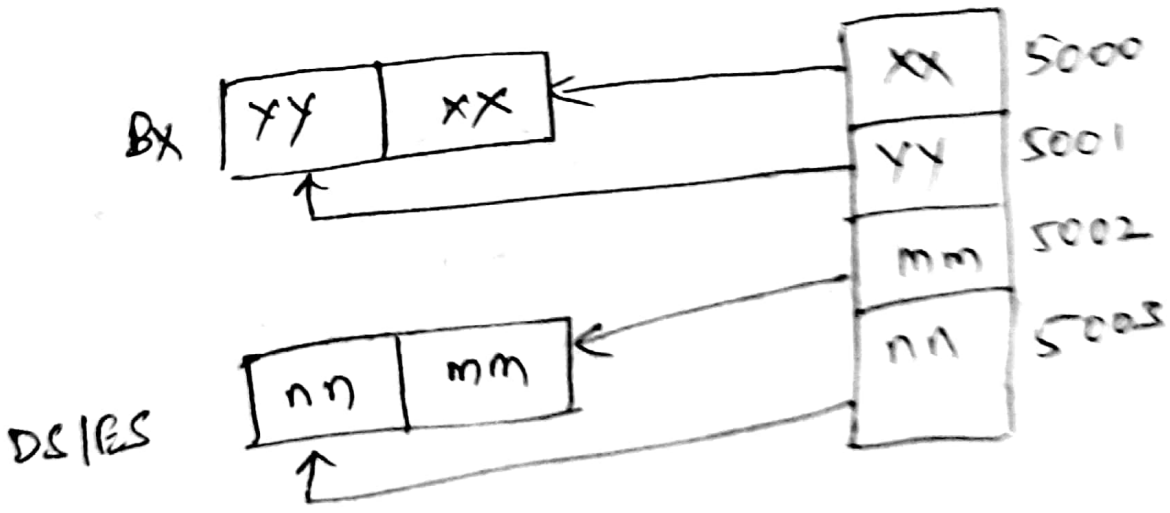
```
LEA BX, ADR    ; effective address of label
                 ADR ie. offset of ADR
                 will be transferred to BX

LEA SI, ADR[BX] ; offset of label ADR
                  will be added to content
                  of BX to form effective
                  address and it will be
                  loaded in SI.
```

(9) LDS / LES : Load pointer to DS / ES.

→ This instruction loads the DS or ES register and the specified destination register in instruction with the content of memory location specified as the source in the instruction.

Ex:- LDS BX, 5000H

LES BX, 5000H



(10) LAHF : load AH from lower byte of flag register

→ It load AH Reg. with the lower byte of flag register.
This can be used to observe the status of all condition code flags (except overflow) at a time.

(11) SAHF : Store AH to lower byte of flag register :

This instruction sets or resets the condition code flags depending upon the corresponding bit position in AH.

(12) PUSHF : Push flags to stack.

→ pushes the flag register on to stack

(13) POPF : pop flags from stack

pop flags instruction loads the flag reg. completely from the word contents of memory location currently addressed by SP and SS.

Arithmetic Instructions:

→ These instructions perform arithmetic operations like addition, subtraction, multiplication, 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 condition code flags.

→ The operands are either registers or memory locations or immediate data depending on the addressing mode.

(1) ADD: Add

→ This instructions adds an immediate data or contents of a memory location or register to the contents of another register or memory location. The result will be stored in the destination operand.

Note: Both source and destination operands cannot be memory locations

Note: Also contents of segment registers cannot be added using this instruction.

→ All condition code flags are affected depending on the result.

Ex:-

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)

(2) ADC : Add with carry

This instruction performs the same operation as ADD instruction, but adds the carry flag bit generated from any previous calculations to the result.

All condition code flags are affected.

Ex:-

(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

(3) INC : Increment

This instruction increases the contents of specified register or memory location by 1. All condition code flags affected except carry flag CF.

Note : Immediate data cannot be an operand in this instruction.

Ex:-

(1) INC AX ; Register

(2) INC [BX] ; Register Indirect

(3) INC [5000H] ; Direct

④ DEC : Decrement

→ The decrement instruction subtracts 1 from the contents of specified register or memory location.

→ All condition code flags except carry flag modified.

Ex:-

① DEC AX ; Register

② DEC [5000H] ; Direct

(5) SUB : Subtract

→ The subtract instruction subtracts the source operand from the destination operand and the result is left in the destination operand.

→ Source may be a register, memory location or immediate data

→ destination may be a register or memory location.

→ Source and destination operands both must not be memory operands.

→ All condition code flags affected.

Ex:-

(1) SUB AX, 0100H ; Immediate (destination AX)

(2) SUB AX, BX ; Register

(3) SUB AX, [5000H] ; Direct

(4) SUB [5000H], 0100 ; Immediate

(6) SBB : Subtract with borrow

→ This instruction subtracts the source operand and borrow flag (CF) which may reflect the result of previous calculations from the destination operand.

→ The result is stored in the destination operand

→ All the flags are affected.

Ex:-

(1) SBB AX, 0100H ; Immediate (dest. AX)

(2) SBB AX, BX ; register

(3) SBB AX, [5000H] ; Direct

(4) SBB [5000H], 0100 ; Immediate

(7) 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 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.

→ If both operands are same, zero flag is set

→ If the source operand greater than distination operand, the carry flag is set.

Ex:-

(1) CMP BX, 0100H ; Immediate

(2) CMP AX, 0100H ; Immediate

(3) CMP [5000H], 0100H ; Direct

(4) CMP BX, [SI] ; Reg. Indirect

(5) CMP BX, CX ; Register

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 high 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 in the 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 higher 4 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 Fig. 2.6. This instruction does not give exact ASCII codes of the sum, but they can be obtained by adding 3030H to AX.

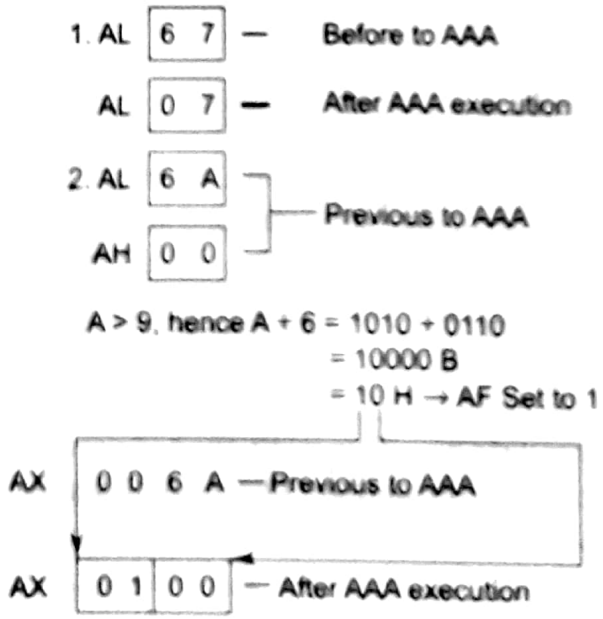


Fig. 2.6 *ASCII Adjust after Addition Instruction*

AAS: ASCII Adjust AL 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 1, the AL is decremented by 6 and AH register is decremented by 1, the CF and AF are set to 1. Otherwise, the CF and AF are set to 0, the result needs 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 is similar to the AAA instruction except for the subtraction of 06 from AL. AH is modified as difference of the previous contents (usually zero) 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. The AAM—ASCII Adjust After Multiplication—instruction follows a multiplication instruction that multiplies two unpacked BCD operands, i.e. higher nibbles of the multiplication operands should be O. The multiplication of such operands is carried out using MUl instruction. Obviously the result of multiplication is available in AX. The following AAM instruction replaces content of AH by tens of the decimal multiplication and AL by singles of the decimal multiplication.

Example 2.31

```
MOV AL, 04   ;  AL ← 04
MOV BL, 09   ;  BL ← 09
MUL BL       ;  AH-AL ← 24H (9 × 4)
AAM          ;  AH ← 03
             ;  AL ← 06
```

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 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. In the instruction sequence, this instruction appears before DIV instruction unlike AAM appears after MUL. Let AX contains 0508 unpacked BCD for 58 decimal, and DH contains 02H.

Example 2.32

```
AX              05  08
AAD result in AL    00  3A      58D = 3A H in AL
```

The result of AAD execution will give the hexadecimal number 3A in AL and 00 in AH. Note that 3A is the hexadecimal equivalent of 58 (decimal). Now, instruction DIV DH may be executed. So rather than ASCII adjust for division, it is ASCII adjust before division. All the ASCII adjust instructions are also called as unpacked BCD arithmetic instructions. Now, we will consider the two instructions related to packed BCD arithmetic.

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 examples given below explain the instruction.

Example 2.33

```
(i) AL = 53        CL = 29
    ADD AL, CL     ; AL ← (AL) + (CL)
                   ; AL ← 53 + 29
                   ; AL ← 7C
    DAA            ; AL ← 7C + 06 (as C>9)
                   ; AL ← 82
```

```
(ii) AL = 73        CL = 29
     ADD AL, CL     ; AL ← AL + CL
                    ; AL ← 73 + 29
                    ; AL ← 9C
     DAA            ; AL ← 02 and CF = 1
                      AL = 7 3
                         +
                    CL = 2 9
                    ------
                         9 C
                       + 6
                    ------
                         A 2
                       + 6 0
                    ------
                    CF = 1  0 2  in AL
```

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

DAS: Decimal Adjust after Subtraction This instruction converts the result 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 subtraction sets the carry flag or if upper nibble is greater than 9, it subtracts 60H from AL. This instruction modifies the AF, CF, SF, PF and ZF flags. The OF is undefined after DAS instruction. The examples are as follows:

Example 2.34

```
(i) AL = 75        BH = 46
    SUB AL, BH     ; AL ← 2 F = (AL) - (BH)
                   ; AF = 1
    DAS            ; AL ← 2 9 (as F > 9, F - 6 = 9)
(ii) AL = 38       CH = 6 1
    SUB AL, CH     ; AL ← D 7  CF = 1 (borrow)
    DAS            ; AL ← 7 7  (as D > 9, D - 6 = 7)
                   ; CF = 1 (borrow)
```

DAA and DAS instructions are also called packed BCD arithmetic instructions.

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.

MUL: Unsigned Multiplication Byte or Word This instruction multiplies an unsigned byte or word by the contents of AL. The unsigned byte or word may be in any one of the general purpose registers or memory locations. The most significant word of the 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. The example instructions are as shown. Immediate operand is not allowed in this instruction. If the most significant byte or word of the result is '0' CF and OF both will be set.

Example 2.35

```
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 a signed word in source operand by a 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 DX contain parts of 16 and 32-bit result respectively, CF and OF both will be set. The AL and AX are the implicit operands in case of 8 bits and 16 bits multiplications respectively. The unused higher bits of the result are filled by sign bit and CF, AF are cleared. The example instructions are given as follows:

Example 2.36

```
1. IMUL     BH
2. IMUL     CX
3. IMUL     [SI]
```

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 the DX register. This operation is to be done before signed division. It does not affect any flag.

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) and an 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.

IDIV: Signed Division This instruction performs the same operation as the DIV instruction, but 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 0 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.

2.3.3 Logical Instructions

These type of instructions are used for carrying out the bit by bit shift, rotate, or basic logical operations. All the condition code flags are affected depending upon the result. Basic logical operations available with 8086 instruction set are AND, OR, NOT, and XOR. The instruction for each of these operations are discussed as follows.

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
```

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.

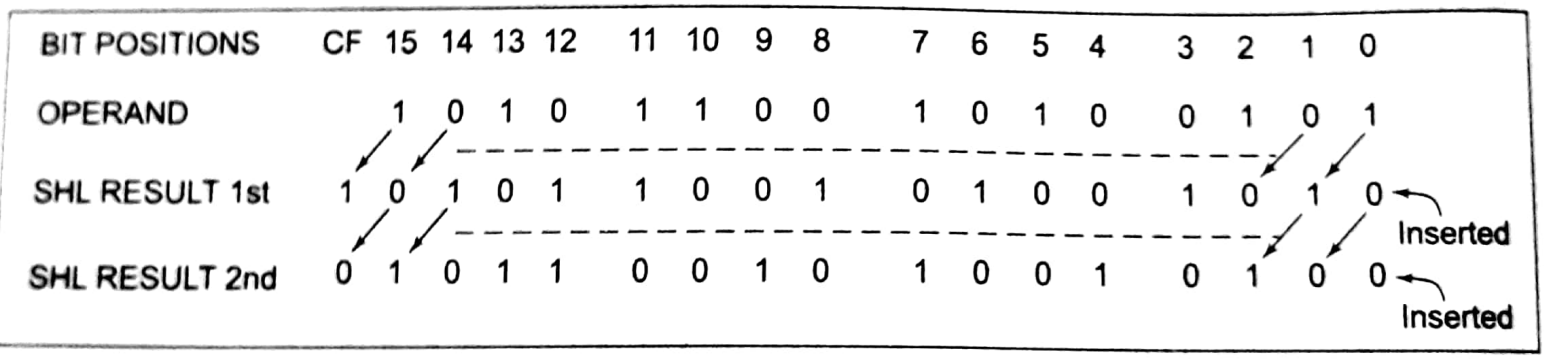


Fig. 2.7 *Execution of SHL/SAL Instruction*

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.

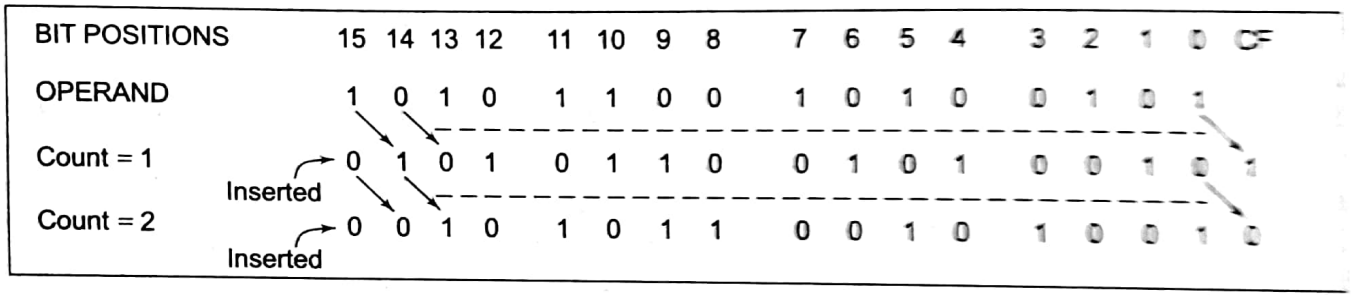


Fig. 2.8 *Execution of SHR Instruction*

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.

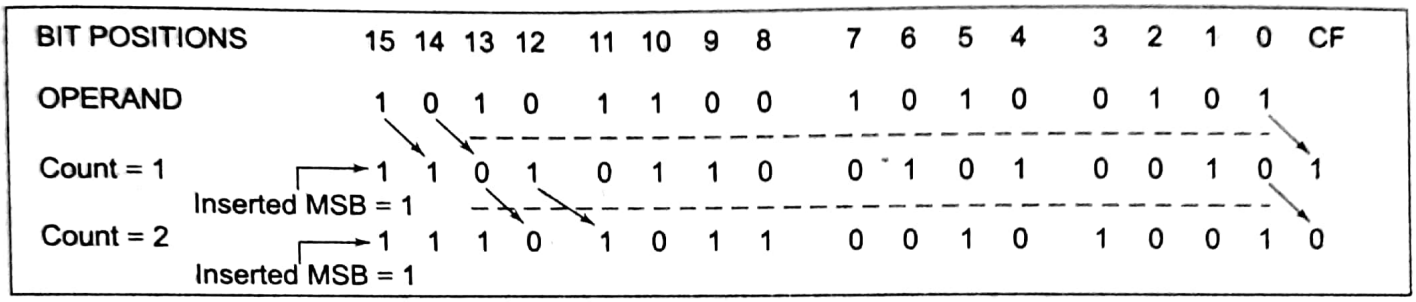


Fig. 2.9 *Execution of SAR Instruction*

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

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.

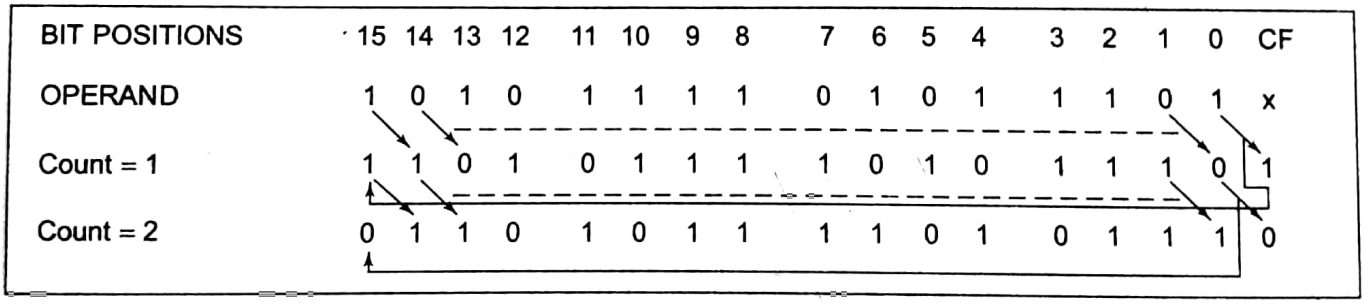


Fig. 2.10 *Execution of ROR Instruction*

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.

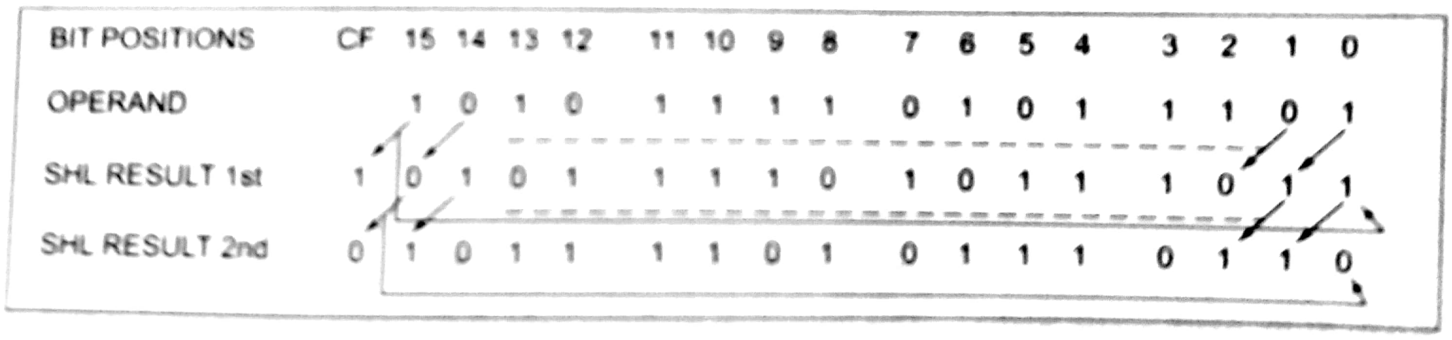


Fig. 2.11 *Execution of ROL Instruction*

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.

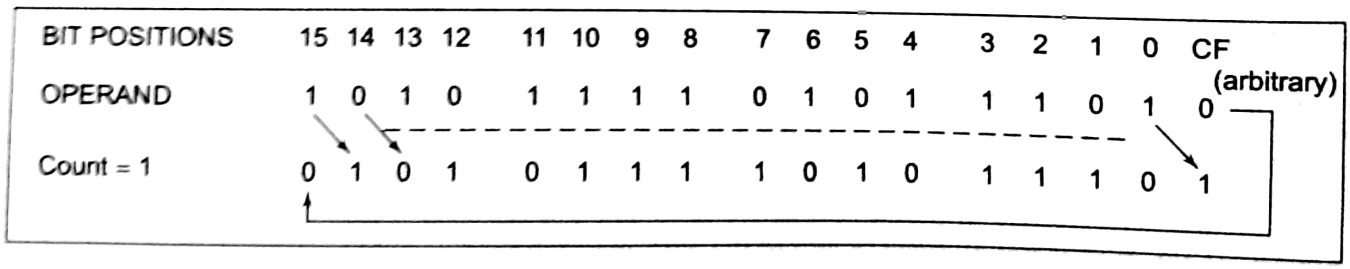


Fig. 2.12 *Execution of RCR Instruction*

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.

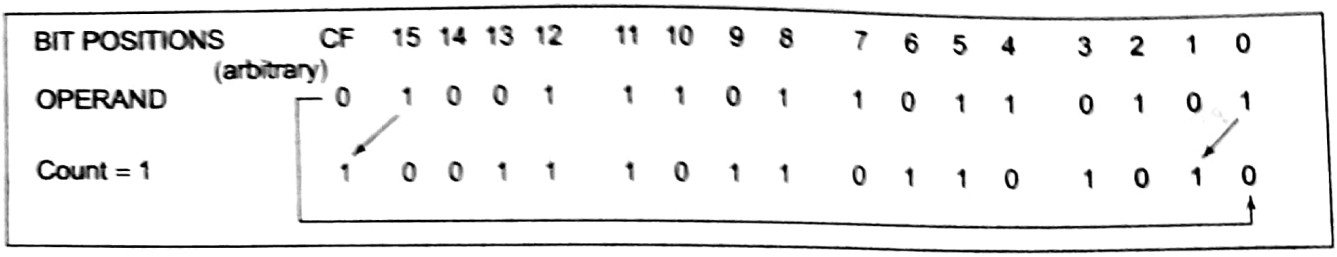


Fig. 2.13 *Execution of RCL Instruction*

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

2.3.4 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*. For example, a string of characters may be located in consecutive memory locations, where each character may be represented by its ASCII equivalent. For referring to a string, two parameters are required, (a) starting or end address of the string and (b) length of the string. The length of a string is usually stored as count in the CX register. In case of 8085, similar structures can be set up by the pointer and counter arrangements which may be modified at each iteration, till the required condition for proceeding further is satisfied. On the other hand, the 8086 supports a set of more powerful instructions for string manipulations. The incrementing or decrementing of the pointer, in case of 8086 string instructions, depends upon the Direction Flag (DF) status. If it is a byte string operation, the index registers are updated by one. On the other hand, if it is a word string operation, the index registers are updated by two. The counter in both the cases, is decremented by one.

→ The string instructions operate on elements of strings, bytes or words.

→ The SI register contains the offset address of an element (byte or word) in the source string, which is present in Data Segment

→ The DI register contains the offset address of an element (byte or word) in the destination string, which is present in extra segment.

→ After each string operation, SI and/or DI are automatically incremented or decremented by 1 or 2 (for byte or word operation) according to the D flag in the flag register

If D = 0, SI and/or DI are incremented

If D = 1, SI and/or DI are decremented.

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). The REP instruction prefix is used with MOVS instruction to repeat it by a value given in the counter (CX). The length of the byte string or word string must be stored in CX register. No flags are affected by this instruction.

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 the string manipulation instructions. The following string of instructions explain the execution of the MOVS instruction.

Example 2.42

```
MOV AX, 5000H     ; Source segment address is 5000h
MOV DS, AX        ; Load it to DS
MOV AX, 6000H     ; Destination segment address is 6000h
MOV ES, AX        ; Load it to ES
MOV CX, 0FFH      ; Move length of the string to counter register CX
MOV SI, 1000H     ; Source index address 1000H is moved to SI
MOV DI, 2000H     ; Destination index address 2000H is moved to DI
CLD               ; Clear DF, i.e. set autoincrement mode
REP MOVSB         ; Move 0FFH string bytes from source address to destination
```

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 while equal/zero. The second is REPNE/REPNZ allows for repeating the operation while not equal/not zero. These options are used for CMPS, SCAS instructions only, as instruction prefixes.

CMPS: Compare String Byte or String Word The CMPS instruction can be 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 byte or word of the string, after each comparison the index registers are updated depending upon 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.

Example 2.43

```
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. SEG2 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, 010H              ; Length of the string is moved to CX
CLD                       ; Clear DF, i.e. set autoincrement 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. The following string of instructions elaborates the use of SCAS instruction.

Example 2.44

```
  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 programme 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 upon DF. The DF plays exactly the same role as in case of MOVSB/MOVSW instruction. If it is a byte transfer(LODSB), the SI is modified by 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 autodecrement mode. In this mode, SI and DI are decremented automatically after each iteration (by 1 or 2 depending upon byte or word operations). Hence, in autodecrementing mode, the strings are referred to by their ending addresses. If DF = 0, then the execution follows autoincrement mode. In this mode, SI and DI are incremented automatically (by 1 or 2 depending upon byte or word operation) after each iteration, hence the strings, in this case, are referred to by their starting addresses. Chapter 3 on assembly language programming explains the use of some of these instructions in assembly language programs.

2.3.5 Control Transfer or Branching Instructions

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. Depending upon the addressing modes specified in Chapter 1, the CS may or may not be modified. This type of instructions are classified in two types:

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.

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 this type of instruction the control will be transferred to a particular specified location, if a particular flag satisfies the condition.

2.3.6 Unconditional Branch Instructions

CALL: Unconditional Call This instruction is used to call a subroutine from a main program. In case of assembly language programming, the term procedure is used interchangeably with subroutine. The address of the procedure may be specified directly or indirectly depending upon the addressing mode. There are again two types of procedures depending upon whether it is available in the same segment (Near CALL, i.e. ±32K displacement) or in another segment (FAR CALL, i.e. anywhere outside the segment). The modes for them are called as intrasegment and intersegment addressing modes respectively. This instruction comes under unconditional branch instructions and can be described as shown with the coding formats. On execution, this instruction stores the incremented IP (i.e. address of the next instruction) and CS onto the stack and loads the CS and IP registers, respectively, with the segment and offset addresses of the procedure to be called. In case of NEAR CALL it pushes only IP register and in case of FAR CALL it pushes IP and CS both onto the stack. The NEAR and FAR CALLS are discriminated using opcode.

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. The procedure may be a near or a far procedure. 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 upon the type of procedure and the SP contents, the RET instruction is of four types.

1. Return within segment
2. Return within segment adding 16-bit immediate displacement to the SP contents.
3. Return intersegment
4. 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 IF must be enabled.

Example 2.45

Thus the instruction INT 20H will find out the address of the interrupt service routine as follows:

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

Pointer to IP and CS of the ISR is 0000 : 0080 H

INTO: Interrupt on Overflow This command is executed, when the overflow flag OF is set. The new contents of IP and CS are taken from the address 0000:0010 as explained in INT type instruction. This is equivalent to a Type 4 interrupt instruction.

JMP: Unconditional Jump This instruction unconditionally transfers 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 far). No flags are affected by this instruction. Corresponding to the methods of specifying jump addresses, the JUMP instruction may have the following three formats. For other JMP types the reader may refer to the following datasheet.

JUMP DISP 8-bit		Intrasegment, relative, short jump
JUMP	DISP.16-bit (LB) | DISP.16-bit (HB)	Intrasegment, relative, short jump
JUMP	IP (LB) | IP (HB) | CS (LB) | S (HB)	Intrasegment, direct, far jump

IRET: Return from ISR When an 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 values of 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. The following sequence explains the execution. At each iteration, CX is decremented automatically. In other words, this instruction implements DECREMENT COUNTER and JUMP IF NOT ZERO structure.

Example 2.46

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

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/JNC	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).

Table 2.4 *Conditional Loop Instructions*

Mnemonic	*Displacement*	*Operation*
LOOPZ/LOOPE (Loop while ZF = 1; equal)	Label	Loop through a sequence of instructions from 'Label' while ZF=1 and CX ¹ 0.
LOOPNZ/LOOPENE (Loop while ZF = 0; not equal)	Label	Loop through a sequence of instructions from 'Label' while ZF=0 and CX ¹ 0.

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.