APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

KeralaNotes

SYLLABUS | STUDY MATERIALS | TEXTBOOK

PDF | SOLVED QUESTION PAPERS

www.keralanotes.com

KeralaNotes

KTU STUDY MATERIALS

MICROPROCESSORS AND MICROCONTROLLERS

CST 307

Module 2

Related Link :

- KTU S5 STUDY MATERIALS
- KTU S5 NOTES
- KTU S5 SYLLABUS
- KTU S5 TEXTBOOK PDF
- KTU S5 PREVIOUS YEAR SOLVED QUESTION PAPER

- **Addressing modes of 8086**

The addressing modes for sequential and control transfer instructions are explained as follows:

1. Immediate In this type of addressing, immediate data is a part of instruction, and appears in the form of successive byte or bytes.

Example

```
MOV AX, 0005H
MOV BL, 06H
```

In the above examples 0005H and 06H are the immediate data. The immediate data may be 8-bit or 16-bit in size.

2. Direct In the direct addressing mode, a 16-bit memory address (offset) or an IO address is directly specified in the instruction as a part of it.

Example :

```
MOV AX, [5000H]
IN 80H
```

Here, data resides in a memory location in the data segment, whose effective address may be computed using 5000H as the offset address and content of DS as segment address. The effective address, here, is 10H*DS+5000H. In the second instruction 80H is IO address.

3. Register In the register addressing mode, the data is stored in a register and it is referred using the particular register. All the registers, except IP, may be used in this mode.

Example

```
MOV BX, AX.
ADC AL, BL
```

The operands in these instructions are provided in registers BX, AX and AL, BL respectively.

4. Register Indirect Sometimes, the address of the memory location which contains data or operand is determined in an indirect way, using the offset registers. This mode of addressing is known as register indirect mode. In this addressing mode, the offset address of data is in either BX or SI or DI register. The default segment is either DS or ES. The data is supposed to be available at the address pointed to by the content of any of the above registers in the default data segment.

Example

```
MOV AX, [BX]
```

Here, data is present in a memory location in DS whose offset address is in BX. The effective address of the data is given as 10H*DS+[BX].

5. Indexed In this addressing mode, offset of the operand is stored in one of the index registers. DS is the default segment for index registers SI and DI. In case of string instructions DS and ES are default segments for SI and DI respectively. This mode is a special case of register indirect addressing mode.

Example

```
MOV AX, [SI]
MOV CX, [DI]
```

Here, data is available at an offset address stored in SI in DS. The effective address, in this case, is computed as 10H*DS+[SI]. The content of address 10H*DS+[SI] will be transferred into register CX.

6. Register Relative In this addressing mode, the data is available at an effective address formed by adding an 8-bit or 16-bit displacement with the content of any one of the registers BX, BP, SI and DI in the default (either DS or ES) segment.

Example

```
MOV AX, 50H[BX]
MOV 10H[SI], DX
```

Here, the effective address is given as 10H*DS+50H+[BX] and 10H*DS+10H+[SI] respectively.

7. Based Indexed The effective address of data is formed, in this addressing mode, by adding content of a base register (any one of BX or BP) to the content of an index register (any one of SI or DI). The default segment register may be ES or DS.

Example

```
MOV AX, [BX] [SI]
MOV [BX] [DI], AX
```

Here, BX is the base register and SI is the index register. The effective address is computed as 10H*DS+[BX]+[SI].

8. Relative Based Indexed The effective address is formed by adding an 8 or 16-bit displacement with the sum of contents of any one of the base registers (BX or BP) and any one of the index registers, in a default segment.

Example

```
MOV AX,50H [BX][SI]
ADD 50H [BX] [SI], BP
```

Here, 50H is an immediate displacement, BX is a base register and SI is an index register. The effective address of data is computed as 10H*DS+[BX]+[SI]+50H. The second instruction adds content of B with memory location of which offset is given by adding 50H of content of BX and SI. The result is stored in the memory location.

For the control transfer instructions, the addressing modes depend upon whether the destination location is within the same segment or in a different one. It also depends upon the method of passing the destination address to the processor. Basically, there are two addressing modes for the control transfer instructions, viz. intersegment and intrasegment addressing modes.

If the location to which the control is to be transferred lies in a different segment other than the current one, the mode is called intersegment mode. If the destination location lies in the same segment, the mode is called intrasegment mode.

Figure shows the modes for control transfer instructions.

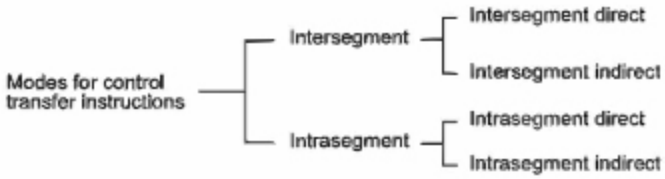

Fig. *Addressing Modes for Control Transfer Instructions*

9. Intrasegment Direct Mode In this mode, the address to which the control is to be transferred lies in the same segment in which the control transfer instruction lies and appears directly in the instruction as an immediate displacement value. In this addressing mode, the displacement is computed relative to the content of the instruction pointer IP.

The effective address to which the control will be transferred is given by the sum of 8 or 16 bit displacement and current content of IP. In case of jump instruction, if the signed displacement (d) is of 8 bits (i.e. $-128 < d < +127$), we term it as *short jump* and if it is of 16 bits (i.e. $-32768 < d < +32767$), it is termed as *long jump*.

Example

```
JMP SHORT LABEL; LABEL lies within –128 TO +127 from the current IP
content.
```

Thus SHORT LABEL is 8-bit signed displacement.

A 16-bit target address of a label indicates that it lies within – 32768 to + 32767. But a problem arises when one requires a forward jump at a relative address greater than 32767 or backward jump at relative address – 32768; in the same segment. Suppose current contents of IP are 5000H then a forward jump may be allowed at all the displacement DISP so that IP + DISP = FFFFH or DISP = FFFF – 5000 = AFFFH. Thus forward jumps may be allowed for all 16-bit displacement values from 0000H to AFFFH. If displacement exceeds AFFFH i.e. from B000H to FFFFH, then all such jumps will be treated as backward jumps. All such jumps are called NEAR PTR jumps and coded as below.

```
JMP NEAR PTR LABEL
```

10. Intrasegment Indirect Mode In this mode, the displacement to which the control is to be transferred, is in the same segment in which the control transfer instruction lies, but it is passed to the instruction indirectly. Here, the branch address is found as the content of a register or a memory location. This addressing mode may be used in unconditional branch instructions.

Example

```
JMP [BX]; Jump to effective address stored in BX.
                          JMP [ BX + 5000H ]
```

11. Intersegment Direct In this mode, the address to which the control is to be transferred is in a different segment. This addressing mode provides a means of branching from one code segment to another code segment. Here, the CS and IP of the destination address are specified directly in the instruction.

Example

```
JPM 5000H : 2000H;
Jump to effective address 2000H in segment 5000H.
```

12. Intersegment Indirect In this mode, the address to which the control is to be transferred lies in a different segment and it is passed to the instruction indirectly, i.e. contents of a memory block containing four bytes, i.e. IP(LSB), IP(MSB), CS(LSB) and CS(MSB) sequentially. The starting address of the memory block may be referred using any of the addressing modes, except immediate mode.

Example

```
JMP [2000H];
```

Jump to an address in the other segment specified at effective address 2000H in DS, that points to the memory block as said above.

Forming the Effective Addresses The following examples explain forming of the effective addresses in the different modes.

Example

The contents of different registers are given below. Form effective addresses for different addressing modes.
Offset (displacement) – 5000H
[AX]-1000H, [BX]-2000H, [SI]-3000H, [DI]-4000H, [BP]-5000H,
[SP]-6000H, [CS]-0000H, [DS]-1000H, [SS]-2000H, [IP]-7000H.
Shifting a number four times is equivalent to multiplying it by 16_D or 10_H.

(i) Direct addressing mode

```
                          MOV AX, [5000H]
DS:OFFSET ⇔  1000H: 5000H
10H* DS ⇔     10000
Offset ⇔    + 5000
            ---------
            15000H - Effective address
```

(ii) Register indirect

```
                          MOV AX, [BX]
DS:BX ⇔   1000H:2000H
10H*DS ⇔    10000
[BX] ⇔    + 2000
          ----------
          12000H - Effective address
```

(iii) Registor relative

```
                                MOV AX, 5000 [BX]
DS: [5000 + BX]
    10H*DS ⇔   10000
    Offset ⇔  + 5000
      [BX] ⇔  + 2000
              ---------
              17000H - Effective address
```

(iv) Based indexed

```
                                MOV AX, [BX] [SI]
DS:[BX + SI]
    10H*DS ⇔   10000
      [BX] ⇔  + 2000
      [SI] ⇔  + 3000
              ---------
              15000H - Effective address
```

(v) Relative based indexed

```
                          MOV AX, 5000 [BX] [SI]
DS: [BX + SI + 5000]
    10H*DS ⇔   10000
      [BX] ⇔  + 2000
      [SI] ⇔  + 3000
    Offset ⇔  + 5000
              ---------
              1A000 - effective address
```

we present examples of address formation in control transfer instructions.

Example

Suppose our main program resides in the code segment where CS = 1000H. The main program calls a subroutine which resides in the same code segment. The base register contains offset of the subroutine, i.e. BX = 0050H. Since the offset is specified indirectly, as the content of BX, this is indirect addressing. The instruction CALL [BX] calls the subroutine located at an address 10H*CS + [BX] = 10050H, i.e. in the same code segment. Since the control goes to the subroutine which resides in the same segment, this is an example of intrasegment indirect addressing mode.

Example

Let us now assume that the subroutine resides in another code segment, where CS = 2000H. Now CALL 2000H:0050H is an example of intersegment direct addressing mode, since the control now goes to different segment and the address is directly specified in the instruction. In this case, the address of the subroutine is 20050H.

- **8086 instruction set**

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.

in case of immediate addressing mode, a segment register cannot be a 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.

Example

Load DS with 5000H.

1. MOV DS, 5000H; Not permitted (invalid)
   Thus to transfer an immediate data into the segment register, the correct procedure is given below.
2. MOV AX, 5000H
   MOV DS, AX
   It may be noted here that both the source and destination operands cannot be memory locations (except for string instructions). Other MOV instruction examples are given below with the corresponding addressing modes.
3. MOV AX, 5000H; Immediate
4. MOV AX, BX; Register
5. MOV AX, [SI]; Indirect

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 then stores 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 decremented stack addresses the higher byte occupies the higher address and the lower byte occupies the lower address.

The actual operation takes place as given below SS : SP points to the stack top of 8086 system as shown in Fig. and AH, AL contains data to be pushed.

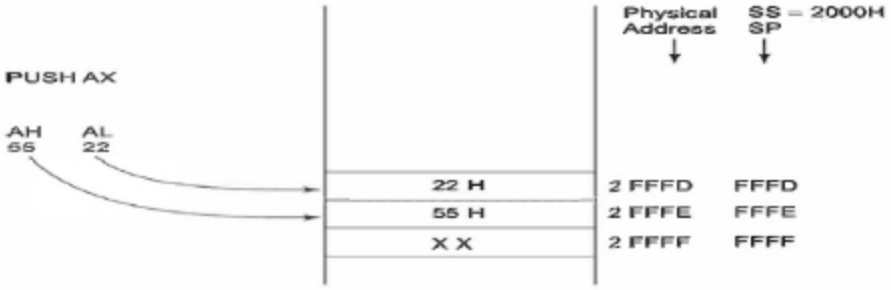

Fig. ***Pushing Data to Stack Memory***

The sequence of operation as below:

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.

Thus SP is decremented by 2 and AH–AL contents are stored in stack memory as shown in Fig. Contents of SP points to a new stack top.

The examples of these instructions are as follows:

Example

1. PUSH AX
2. PUSH DS
3. PUSH [5000H]; Content of location 5000H and 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 the 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.

16-bit contents of current stack top are poped into the specified operand as follows.

The sequence of operation is as below.

1. Contents of stack top memory location is stored in AL and SP is incremented by one
2. Further contents of memory location pointed to by SP are copied to AH and SP is again incremented by 1

Effectively SP is incremented by 2 and points to next stack top.

The examples of these instructions are shown as follows:

Example

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

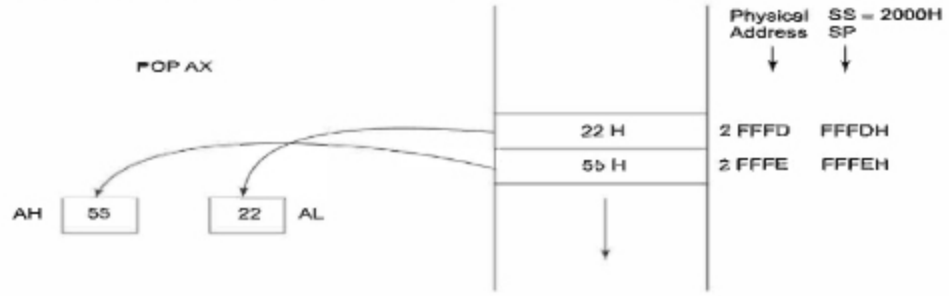

Fig. *Popping Register Contents from Stack Memory*

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 contents of two memory locations is not permitted. Immediate data is also not allowed in these instructions. The examples are as follows:

Example

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

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. If the port address is of 16 bits it must be in DX. The examples are given as shown:

Example

```
1. IN AL,03H        ; This instruction reads data from an 8-bit port whose address
                    ; is 03H and stores it in AL.
2. IN AX, DX        ; This instruction reads data from a 16-bit port whose
                    ; address is in DX (implicit) and stores it in AX.
3. MOV DX,0800H     ; The 16-bit address is taken in DX.
   IN AX, DX        ; Read the content of the port 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. If the port address is of 16 bits it must be in DX. The examples are given as shown:

Example

```
1. OUT    03H,AL    ; This sends data available in AL to a port whose
                    ; address is 03H.
2. OUT    DX, AX    ; This sends data available in AX to a port whose
                    ; address is specified implicitly in DX.

3. MOV DX, 0300H    ; The 16-bit port address is taken in DX.
   OUT DX, AX       ; Write the content of AX to a port of which address is in DX.
```

XLAT: Translate The translate instruction is used for finding out the codes in case of code conversion problems, using look up table technique.

Suppose, a hexadecimal key pad having 16 keys from 0 to F is interfaced with 8086 using 8255. Whenever a key is pressed, the code of that key (0 to F) is returned in AL. For displaying the number corresponding to the pressed key on the 7-segment display device, it is required that the 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 7-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 the 7-segment codes for 0 to F at the locations 2000H to 200FH sequentially. For executing the XLAT instruction, the code of the pressed key obtained from the keyboard (i.e. the code to be translated) is moved in AL and the base address of the look up table containing the 7-segment codes is kept in BX. After the execution of the XLAT instruction, the 7-segment code corresponding to the pressed key is returned in AL, replacing the key code which was in AL prior to the execution of the XLAT instruction. To find out the exact address of the 7-segment code from the base address of look up table, the content of AL is added to BX internally, and the contents of the address pointed to by this new content of BX in DS are transferred to AL. The following sequence of instructions perform the task.

Example

```
MOV AX, SEG TABLE    ; Address of the segment containing look-up-table
MOV DS,AX            ; is transferred in DS
MOV AL, CODE         ; Code of the pressed key is transferred in AL
MOV BX, OFFSET TABLE; Offset of the code look-up-table in BX
XLAT                 ; Find the equivalent code and store in AL
```

LEA: Load Effective Address The load effective address instruction loads the effective address formed by destination operand into the specified source register. This instruction is more useful for assembly language rather than for machine language.

Example

```
LEA BX,ADR      ; Effective address of Label ADR i.e. offset of ADR will be
                  transferred to Reg ; 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
```

LDS/LES: Load Pointer to DS/ES This instruction loads the DS or ES register and the specified destination register in the instruction with the content of memory location specified as source in the instruction. The example in Fig. explains the operation.

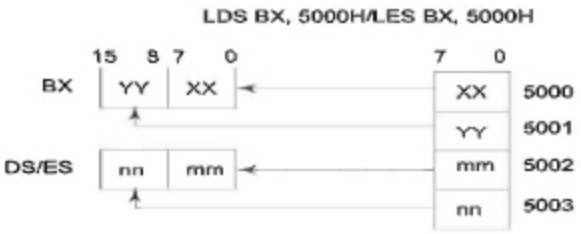

Fig. *LDS/LES Instruction Execution*

LAHF : Load AH from Lower Byte of Flag This instruction loads the AH register with the lower byte of the flag register. This command may be used to observe the status of all the condition code flags (except over flow) at a time.

SAHF: Store AH to Lower Byte of Flag Register This instruction sets or resets the condition code flags (except overflow) in the lower byte of the flag register depending upon the corresponding bit positions in AH. If a bit in AH is 1, the flag corresponding to the bit position is set, else it is reset.

PUSHF: Push Flags to Stack The push flag instruction pushes the flag register on to the stack; first the upper byte and then the lower byte is pushed on to it. The SP is decremented by 2, for each push operation. The general operation of this instruction is similar to the PUSH operation.

POPF: Pop Flags from Stack The pop flags instruction loads the flag register completely (both bytes) from the word contents of the memory location currently addressed by SP and SS. The SP is incremented by 2 for each pop operation.

Arithmetic Instructions

These instructions 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 condition code flags. The operands are either the registers or memory locations or immediate data depending upon the addressing mode.

ADD: Add 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. 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.

Example

```
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.

Example

```
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
```

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

Example:

```
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 the carry flag, are affected depending upon the result. Immediate data cannot be operand of the instruction.

Example

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

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, memory location or immediate data and the destination operand may be a register or a memory location, but source and destination operands both must not be memory operands. Destination operand can not be an immediate data. All the condition code flags are affected by this instruction.

Example

```
1.SUB     AX, 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 flags are affected (Condition code) by this instruction.

Example

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

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 upon the result of the subtraction. If both of 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.

Example

```
1.CMP BX, 0100H          Immediate
2.CMP AX, 0100H          Immediate
3.CMP [5000H], 0100H     Direct
4.CMP BX, [SI]           Register 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 ●●) 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.

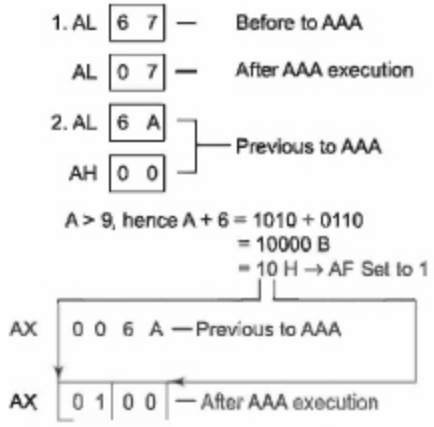

Fig. ***ASCII Adjust after Addition Instruction***

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.

Example

```
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

```
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, 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.

```
(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.

```
(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.

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:

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.

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.

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.

Example

```
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.

Example

```
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.

Example

```
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
```

Result

In AX = D 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.

Example

```
1.XOR   AX, 0098H
```

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.

Example

```
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 explains the execution of this instruction. It is to be noted here that the shift operation is through carry flag.

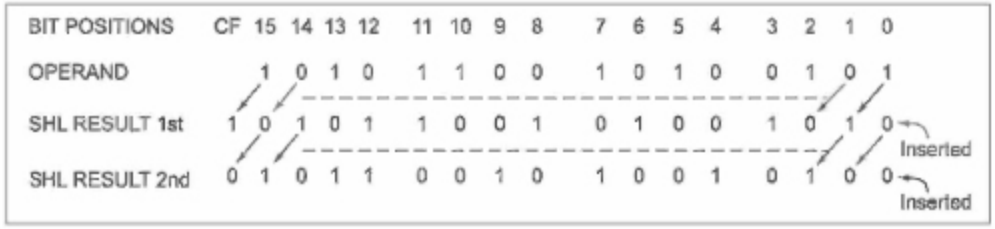

Fig. *Execution of SHL/SAL Instruction*

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

Fig. *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 explains execution of the instruction. All the condition code flags are affected. This shift operation shifts the operand through the carry flag.

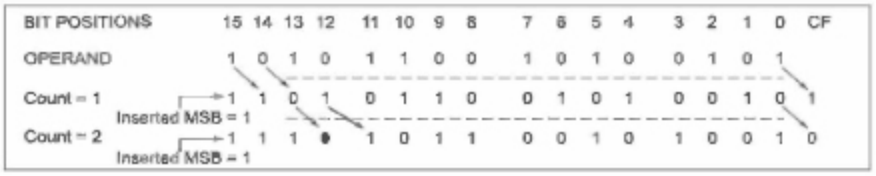

Fig. : *Execution of SAR Instruction*

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 explains the operation. The destination operand may be a register (except a segment register) or a memory location.

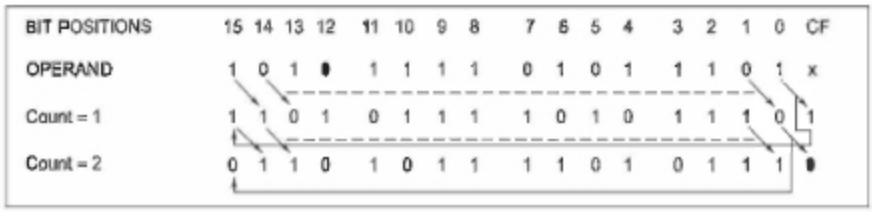

Fig. *Execution of ROR Instruction*

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 explains the operation.

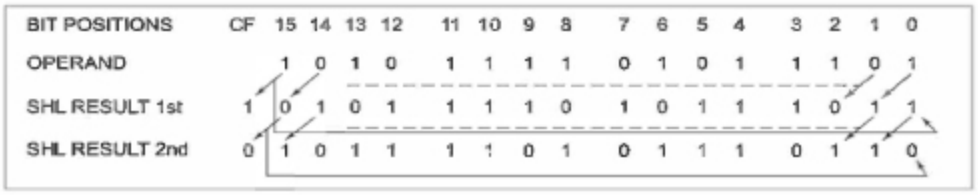

Fig. *Execution of ROL Instruction*

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 explains the operation.

Fig. *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 : explains the operation.

Fig. *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.

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.

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.

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.

```
MOV CX, OFFH      ; 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 auto increment mode
REP MOVSB         ; Move OFFH string bytes from source address to destination
```

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 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.

```
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.

```
  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.

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, 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.

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 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.

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

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.

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.

At each iteration, CX is decremented automatically. In other words, this instruction implements DECREMENT COUNTER and JUMP IF NOT ZERO structure.

Example

```
        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
```

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

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

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).

While the remaining instructions can be used for unsigned binary operations, the last four instructions are used in case of decisions based on signed binary number operations. The terms above and below are generally used for unsigned numbers, the terms less and greater are used for signed numbers. A conditional jump instruction, that does not check status flags for condition testing, is given as follows:

```
JCXZ    'Label'   Transfer execution control
                  to address 'Label', if CX=0.
```

The conditional LOOP instructions are given in Table with their meanings. These instructions may be used for implementing structures like DO_WHILE, REPEAT_UNTIL, etc.

Table ***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.

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

Table *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, i 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 along with their functions. They do not require any operand.

i *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.

after executing the HLT instruction, the processor enters the halt state. The two ways to pull it out of the halt state are to reset the processor or to interrupt it. When NOP instruction is executed, the processor does not perform any operation till 4 clock cycles, except for incrementing the IP by one. It then continues with further execution after 4 clock cycles. ESC instruction when executed, frees the bus for an external master like a coprocessor or peripheral devices. The LOCK prefix may appear with another instruction. When it is executed, the bus access is not allowed for another master till the lock prefixed instruction is executed completely. This instruction is used in case of programming for multiprocessor systems. The WAIT instruction when executed, holds the operation of processor with the current status till the logic level on the $\overline{\text{TEST}}$ pin goes low. The processor goes on inserting WAIT states in the instruction cycle, till the $\overline{\text{TEST}}$ pin goes low. Once the $\overline{\text{TEST}}$ pin goes low, it continues further execution.

3)ASSEMBLER DIRECTIVES

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

Example

DW directive is explained with the DUP operator.

```
WDATA    DW 5 DUP (6666H)
```

This statement reserves five words, i.e. 10-bytes of memory for a word lable WDATA and initialises all the word locations with 6666H.

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

DT: Define Ten Bytes The DT directive directs the assembler to define the specified variable requiring 10-bytes for its storage and initialise the 10-bytes with the specified values.

ASSUME: Assume Logical Segment Name The ASSUME directive is used to inform the assemble, the names of the logicals segments to be assumed for different segments used in the program.

. The statement ASSUME CS : CODE directs the assembler that the machine codes are available in a segment named CODE, and hence the CS register is to be loaded with the address (segment) allotted by the operating system for the label CODE,

ASSUME DS : DATA indicates to the assembler that the data items related to the program, are available in a logical segment named DATA, and the DS register is to be initialised by the segment address

value decided by the operating system for the data segment, It then considers the segment DATA as a default data segment for each memory operation, related to the data and the segment CODE as a source segment for the machine codes of the program. The ASSUME statement is a must at the starting of each assembly language program, v

END: END of Program The END directive marks the end of an assembly language program. When the assembler comes across this END directive, it ignores the source lines available later on.

ENDP: END of Procedure , the subroutines are called procedures. They may be independent program modules which return particular results or values to the calling programs. The ENDP directive is used to indicate the end of a procedure. A procedure is usually assigned a name, i.e. label. To mark the end of a particular procedure, the name of the procedure, i.e. label may appear as a prefix with the directive ENDP.

```
PROCEDURE STAR
      ⋮
   STAR ENDP
```

ENDS: END of Segment This directive marks the end of a logical segment. The logical segments are assigned with the names using the ASSUME directive.
Whatever are the contents of the segments, they should appear in the program before ENDS.

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

data related to the program must lie between the DATA SEGMENT and DATA ENDS statements.
all the executable instructions must lie between CODE SEGMENT and CODE ENDS statements.

EVEN: Align on Even Memory Address The assembler, while starting the assembling procedure of any program, initialises a location counter and goes on updating it, as the assembly proceeds.

The EVEN directive updates the location counter to the next even address, if the current location counter contents are not even, and assigns the following routine or variable or constant to that address.

```
      EVEN
PROCEDURE     ROOT
                ⋮
      ROOT    ENDP
```

The above structure shows a procedure ROOT that is to be aligned at an even address.
When the assembler comes across the directive EVEN, it
checks the contents of the location counter. If it is odd, it is updated to the next even value and then the ROOT procedure is assigned to that address, i.e. the updated contents of the location counter. If the content of the location counter is already even, then the ROOT procedure will be assigned with the same address.

EQU: Equate The directive EQU is used to assign a label with a value or a symbol. The use of this directive is just to reduce the recurrence of the numerical values or constants in a program code. The recurring value is assigned with a label, and that label is used in place of that numerical value, throughout the program.

Using the EQU directive, even an instruction mnemonic can be assigned with a label, which can then be used in the program in place of that mnemonic.

Example

```
LABEL     EQU    0500H
```

The statement assigns the constant 500H with the label LABEL,

EXTRN: External and PUBLIC: Public The directive EXTRN informs the assembler that the names, procedures and labels declared after this directive have already been defined in some other assembly language modules. While in the other module, where the names, procedures and labels actually appear, they must be declared public, using the PUBLIC directive. If one wants to call a procedure FACTORIAL appearing in MODULE1 from MODULE 2; in MODULE1, it must be declared PUBLIC using the statement PUBLIC FACTORIAL and in module 2, it must be declared external using the declaration EXTRN FACTORIAL. Consider the following declarations.

```
MODULE1     SEGMENT
PUBLIC      FACTORIAL FAR
MODULE1     ENDS
MODULE2     SEGMENT
EXTRN       FACTORIAL FAR
MODULE2     ENDS
```

GROUP: Group the Related Segments

This directive is used to inform the assembler to form a logical group of the following segment names.

the group declared segments or operands must lie within a 64Kbyte memory segment.

Eg:

```
PROGRAM GROUP CODE, DATA, STACK
```

CODE, DATA and STACK

segment must lie within a 64kbyte memory segment that is named as PROGRAM. for the ASSUME statement, one can use the label PROGRAM rather than CODE, DATA and STACK

LABEL: Label The Label directive is used to assign a name to the current content of the location counter.

As the program assembly proceeds, the contents of the location counter are updated. The assembler assigns the declared label with the current contents of the location counter. The type of the label must be specified, i.e. whether it is a NEAR or a FAR label, BYTE or WORD label, etc.

The label CONTINUE can be used for a FAR jump,

The LABEL directive can be used to refer to the data segment along with the data type, byte or word as shown.

```
DATA        SEGMENT
DATAS DB 50H DUP (?)
DATA-LAST LABEL BYTE FAR
DATA ENDS
```

After reserving 50H locations for DATAS, the next location will be assigned a label DATA-LAST and its type will be byte and far.

LENGTH: Byte Length of a Label . This is used to refer to the length of a data array or a string.

```
MOV CX, LENGTH ARRAY
```

This statement, when assembled, will substitute the length of the array ARRAY in bytes, in the instruction.

LOCAL The lables, variables, constants or procedures declared LOCAL in a module are to be used only by that particular module. Thus the same label may

serve different purposes for different modules of a program. With a single declaration statement, a number of variables can be declared local,

```
LOCAL a, b, DATA, ARRAY, ROUTINE
```

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 referred to by its declared name.'

OFFSET: Offset of a Label When the assembler comes across the OFFSET operator along with a label, it first computes the 16-bit displacement of the particular label,
and replaces the string 'OFFSET LABEL' by the computed displacement.

Example

```
CODE SEGMENT
MOV SI, OFFSET LIST
CODE ENDS
DATA SEGMENT
LIST DB 10H
DATA ENDS
```

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. If an ORG

200H statement is present at the starting of the code segment of that module, then the cods will start from 200H address in code segment. In other words, the location counter will get initialised to the address 0200H instead of 0000H.

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

For example, the statement RESULT PROC NEAR marks
the start of a routine RESULT, which is to be called by a program located in the same segment of memory. The FAR directive is used for the procedures to be called by the programs located in different segments of memory.

Example

```
RESULT    PROC  NEAR
ROUTINE   PROC  FAR
```

PTR: Pointer The POINTER operator is used to declare the type of a label, variable or memory operand. The operator PTR is prefixed by either BYTE or WORD. If the prefix is BYTE, then the particular label, variable or memory operand is treated as an 8-bit quantity, while if WORD is the prefix, then it is treated as a 16-bit quantity.

Example

```
INC BYTE PTR [BX]-          Increments byte contents of memory
                            location addressed by BX
INC WORD PTR [3000H] -      Increments word contents of memory
                            location 3000H considering contents
                            of 3000H (lower byte) and 3001H
                            (higher byte) as a 16-bit number
```

In case of JMP instructions, the PTR operator is used to specify the type of the jump, i.e. near or far,

```
JMP NEAR PTR [BX]-NEAR Jump
JMP FAR PTR [BX]-FAR Jump.
```

PUBLIC the PUBLIC directive is used along with the EXTRN directive. This informs the assembler that the labels, variables, constants, or procedures declared PUBLIC may be accessed by other assembly modules to form their codes, but while using the PUBLIC declared lables, variables, constants or procedures the user must declare them externals using the EXTRN directive.

SEG: Segment of a Label The SEG operator is used to decide the segment address of the label, variable, or procedure and substitutes the segment base address in place of "SEG" label.

Example

```
MOV AX, SEG ARRAY  ; This statement moves the segment address of ARRAY in
MOV DS, AX         ; which it is appearing, to register AX and then to DS.
```

SEGMENT: Logical Segment The SEGMENT directive marks the starting of a logical segment. The started segment is also assigned a name, i.e. label, by this statement.
In some cases, the segment may be assigned a type like PUBLIC or GLOBAL

```
EXE.CODE SEGMENT GLOBAL; Start of Segment named EXE.CODE,
                       ; that can be accessed by any other module.
EXE.CODE ENDS          ; END of EXE.CODE logical segment.
```

SHORT The SHORT operator indicates to the assembler that only one byte is required to code the displacement for a jump (i.e. displacement is within –128 to +127 bytes from the address of the byte next to the jump opcode).

```
JMP SHORT LABEL
```

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 the word type variable, the data type is 2, for double word type, it is 4, and for byte type, it is 1. Suppose, the STRING is a word array. The instruction MOV AX, TYPE STRING moves the value 0002H in AX.

GLOBAL The labels, variables, constants or procedures declared GLOBAL may be used by other modules of the program.
following statement declares the procedure ROUTINE as a global label.

```
ROUTINE PROC      GLOBAL
```

'+ & –' Operators These operators represent arithmetic addition and subtraction respectively and are typically used to add or subtract displacements (8 or 16 bit) to base or index registers or stack or base pointers

Example

```
MOV AL, [ SI +2 ]
MOV BX, [ OFFSET LABEL + 10 H ]
```

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 offset followed by 2 bytes segment address.

Example

```
JMP FAR PTR LABEL
CALL FAR PTR ROUTINE
```


4) MACROS

The difference between a macro and a subroutine is that in the macro the complete code of the instructions string is inserted at each place where the macro-name appears. Hence the EXE file becomes lengthy. Macro does not utilise the service of stack. There is no question of transfer of control as the program using the macro inserts the complete code of the macro at every reference of the macroname.
subroutine is called whenever necessary, i.e. the control of execution is transferred to the subroutine, every time it is called. The executable code in case of the subroutines becomes smaller as the subroutine appears only once in the complete code. Thus, the EXE file is smaller as compared to the program using macro. The control is transferred to a subroutine whenever it is called, and this utilizes the stack service. The program using subroutine requires less memory space for execution than that using macro. Macro requires less time for execution, as it does not contain CALL and RET instructions as the subroutines do.

Defining a MACRO

A MACRO can be defined anywhere in a program using the directives MACRO and ENDM. The label prior to MACRO is the macro name which should be used in the actual program. The ENDM directive marks the end of the instructions or statements sequence assigned with the macro name. The following macro DISPLAY displays the message MSG on the CRT. The syntax is as given:

```
DISPLAY MACRO
        MOV AX, SEG MSG
        MOV DS, AX
        MOV DX, OFFSET MSG
        MOV AH, 09 H
        INT 21 H
ENDM
```

The above definition of a macro assigns the name DISPLAY to the instruction sequence between the directives MACRO and ENDM. While assembling, the above sequence of instructions will replace the label 'DISPLAY', whenever it appears in the program

A macro may also be used in a data segment. a macro may also be used to represent statements and directives. The following
example shows a macro containing statements. The macro defines the strings to be displayed.

```
STRINGS MACRO
      MSG1 DB OAH,0DH, "Program terminated normally",0AH,0DH, "$"
      MSG2 DB OAH,0DH, "Retry , Abort, Fail",OAH,0DH, "$"
ENDM
```

A macro may be called by quoting its name, along with any values to be passed to the macro. Calling a macro means inserting the statements and instructions represented by the macro directly at the place of the macroname in the program.

Passing Parameters to a MACRO

Using parameters in a definition, the programmer specifies the parameters of the macro those are likely to be changed each time the macro is called. For example, the DISPLAY macro can be
made to display two different messages MSG1 and MSG2,

```
DISPLAY MACRO MSG
        MOV AX, SEG MSG
        MOV DS, AX
```

```
                    MOV DX, OFFSET MSG
                    MOV AH, 09 H
                    INT 21 H
                ENDM
```

This parameter MSG can be replaced by MSG1 or MSG2 while calling the macro as shown.

```
                    :
                    :
                DISPLAY MSG1
                    :
                    :
                DISPLAY MSG2
                    :
                    :
MSG1 DB OAH,ODH, "Program Terminated Normally",OAH,ODH, "$"
MSG2 DB OAH,ODH, "Retry, Abort, Fail",OAH,ODH, "$"
```

There may be more than one parameter appearing in the macro definition, meaning thereby that there may be more than one parameters to be passed to the macro, All the parameters are specified in the definition sequentially and also in the call with the same sequence.

A macro may be defined in another macro or in other words a macro may be called from inside a macro. This type of macro is called a nested macro.

- Stack

STACK STRUCTURE OF 8086/88

the stack contains a set of sequentially arranged data bytes , with the last item appearing on top of the stack. This item will be popped off the stack first for use. The stack pointer (SP) register is a 16-bit register that contains the offset of the address that lies in the stack segment. The stack segment, have a memory block of a maximum of 64 Kbyte locations, and thus may overlap with any other segments. The Stack Segment register (SS) and Stack Pointer register (SP) together address the stack-top

Let the content of SS be 5000 H and the content of the stack pointer register be 2050 H. To find out the current stack-top address, the stack segment register content is shifted left by four bit positions (multiplied by 10 H) and the resulting 20-bit content is added with the 16-bit offset value, stored in the stack pointer register. the stack top address can be calculated as shown:

SS ⇒ 5000 H						
SP ⇒ 2050 H						
SS	⇒	0101	0000	0000	0000	
10H * SS	⇒	0101	0000	0000	0000	0000
	+					
SP	⇒		0010	0000	0101	0000
Stack-top address		0101 5	0010 2	0000 0	0101 5	0000 0

the stack top address is 52050 H.

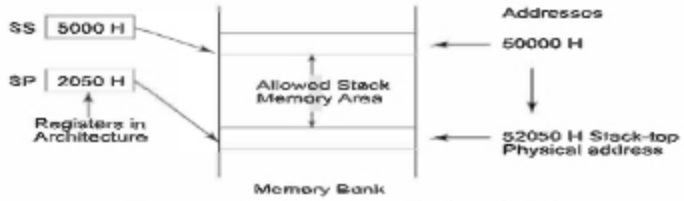

Fig. *Stack-top Address Calculation*

If the stack top points to a memory location 52050 H, previously pushed data is available at 52050 H. The next 16-bit push operation will decrement the stack pointer by two, so that it will point to the new stack-top 5204EH, and the contents of SP will be 204E H.

two locations will be re quired for a 2-byte (16-bit) data.'

Thus for a selected value of SS, the maximum value of SP = FFFF H and the segment can have maximum of 64K locations. starting with an initial value of FFFFH, the Stack Pointer (SP) is decremented by two, whenever a 16-bit data is pushed onto the stack. when the Stack Pointer contains 0000 H, any attempt to further push the data to the stack will result in stack overflow. each POP operation increments the SP.

The POP operation is used to retrieve the data stored on to the stack. In case there is a subroutine CALL instruction there is a possibility that all or some of the registers of the main program may be pushed onto the stack one by one. After each PUSH operation SP will be modified:

At the end of the execution of the subroutine, all the registers can get back their original contents by popping the data from the stack.'

the register or memory location that is pushed into the stack at the end should be popped off first.

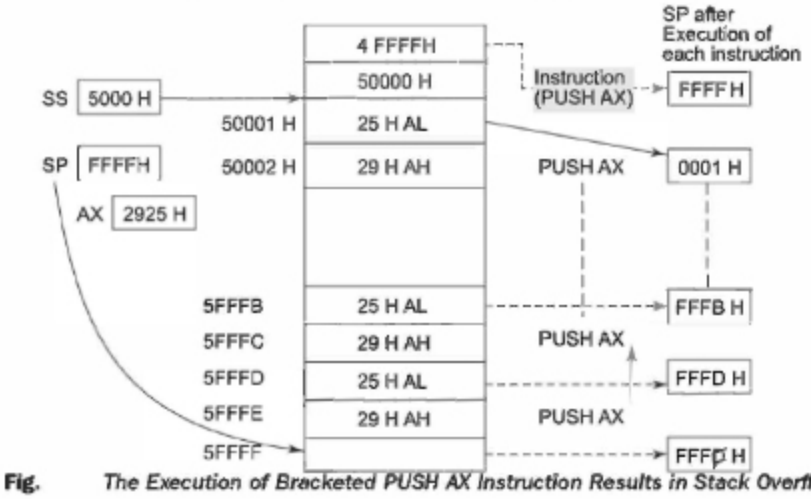

Fig. *The Execution of Bracketed PUSH AX Instruction Results in Stack Overflow*

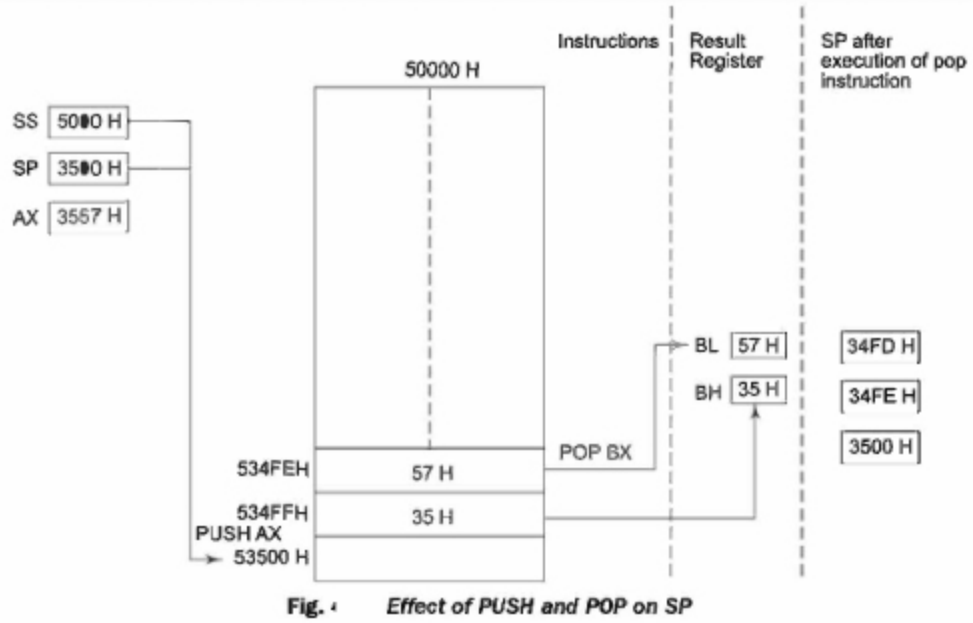

Fig. *Effect of PUSH and POP on SP*

Push registers to the stack at the start of subroutine. Stack mechanism is also used in interrupt service routines to store IP and CS. Stack size is set using DW/DB directive.

- **Passing parameters to procedures**

```
                MOV DS,AX
                  •
                  •
                MOV AX,NUMBER
                  •
                CODE1 ENDS
ASSUME CS:CODE2
CODE2 SEGMENT
          MOV AX,DATA
          MOV DS,AX
          MOV BX,NUMBER

    CODE2 ENDS
          END START
```

The CPU general purpose registers may be used to pass parameters to the procedures. The main program may store the parameters to be passed to the procedure in the available CPU registers and the procedure may use the same register contents for execution. The original contents of the used CPU register may change during execution of the procedure. This may be avoided by pushing all the register content to be used to the stack sequentially at the start of the procedure and by poping all the register contents at the end of the procedure in opposite sequence.

Example

```
ASSUME CS:CODE
CODE SEGMENT
    START :     MOV AX,5555H
                MOV BX,7272H
```



```
                  •
                CALL PROCEDURE1
                  •
                  •
                  •
PROCEDURE     PROCEDURE1 NEAR
                  •
                  •
                ADD AX,BX
                  •
                  •
                RET
PROCEDURE1    ENDP
CODE ENDS
                END START
```

Memory locations may also be used to pass parameters to a procedure in the same way as registers. A main program may store the parameter to be passed to a procedure at a known memory address location and the procedure may use the same location for accessing the parameter.

Example

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
NUM DB (55H)
COUNT EQU 10H
DATA ENDS
CODE SEGMENT
    START :     MOV AX,DATA
                MOV DS,AX
                  •
                  •
                CALL ROUTINE
                  •
                  •
                  •
    PROCEDURE   ROUTINE NEAR
                MOV BX,NUM
                MOV CX,COUTN
                  •
    ROUTINE     ENDP
    CODE        ENDS
                END START
```

Stack memory can also be used to pass parameters to a procedure. A main program may store the parameters to be passed to a procedure in its CPU registers. The registers will further be pushed on to the stack. The procedure during its execution pops back the appropriate parameters as and when required. This procedure of poping back the parameters must be implemented carefully

Example

```
ASSUME CS:CODE, SS:STACK
CODE SEGMENT
    START :     MOV AX,STACK
                MOV SS,AX
                MOV AX,5577H
                MOV BX,2929H
                  •
                PUSH AX
                PUSH BX
                CALL ROUTINE ; Decrements SP by 2 (by 4 far routine)
        PROCEDURE    ROUTINE NEAR
                           •
              •
              MOV BX,SP
              ADD SP,02     ;      Leave initial two stack bytes of
                                   return offset and
                                   segment address after executing
                                   subroutine
                                   POP BX              ;      The data is
                                   POP AX              ;      Passes in BX,AX
                                   MOV SP,DX
                                        •
                                        •
                                        •
    STACK SEGMENT
    STACKDATA DB 200H DUP (?)
    STACK ENDS
```

For passing the parameters to procedures using the PUBLIC & EXTRN directives, must be declared PUBLIC (for all routines) in the main routine and the same should be declared EXT RN in the procedure.

Example

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
PUBLIC NUMBER EQU 200H
         DATA ENDS
         CODE SEGMENT
    START :    MOV AX,DATA
               MOV DS,AX
                 •
                 •
                 •
               CALL ROUTINE
                 •
                 •
                 •
               PROCEDURE ROUTINE NEAR
           EXTRN NUMBER
               MOV AX,NUMBER
                 •
                 •
                 •
           ROUTINE   ENDP
```

HANDLING PROGRAMS OF SIZE MORE THAN 64K

the maximum size of an 8086 segment is 64 KB. This obviously puts limitation on the maximum size of a program and thus how to write programs of size more than 64 K is the question,

The big programming task should be divided into independent modules, which may be developed and tested individual functions of the module

there are two approaches

(i) Writing programs with more than one segment for Data, Code or Stack .
(ii) Writing programs with FAR subroutines each of which can be of size up to 64 K.

Example :

```
ASSUME CS:CODE1, DS:DATA1
CODE1 SEGMENT
    START :     MOV AX,DATA1
                MOV DS,AX
                  .
                  .
    CODE1           ENDS
ASSUME CS:CODE2, DS:DATA2
CODE2 SEGMENT
                MOV AX,DATA2
                MOV DS,AX
                  .
                  .
  CODE2         ENDS
    DATA1               SEGMENT
            .
            .
    DATA1               ENDS

    DATA2               SEGMENT
                        .
                        .
    DATA2               ENDS
                        END START
```

A FEW MACHINE LEVEL PROGRAMS

Example

Write a program to add a data byte located at offset 0500H in 2000H segment to another data byte available at 0600H in the same segment and store the result at 0700H in the same segment.

```
MOV   AX, 2000H    ; Initialising DS with value
MOV   DS, AX       ; 2000H
MOV   AX, [500H]   ; Get first data byte from 0500H
                     offset
ADD   AX, [600H]   ; Add this to the second byte
                     from 0600H
MOV   [700H], AX   ; Store AX in 0700H (result).
HLT                ; Stop
```

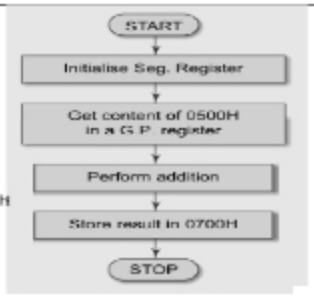

Fig. *Flow Chart for Example*

Example

Move a byte string, 16-bytes long, from the offset 0200H to 0300H in the segment 7000H.

Solution According to the program statement, a string that is 16-bytes long is available at the offset address 0200H in the segment 7000H. The required program should move this complete string at offset 0300H, in the same segment.

```
MOV AX,7000H
MOV DS,AX
MOV ES,AX
MOV CX,0010H
MOV SI,0200H
MOV DI,0300H
CLD
REP MOVSB
HLT
```

Example.

Find out the largest number from an unordered array of sixteen 8-bit numbers stored sequentially in the memory locations starting at offset 0500H in the segment 2000H.

Solution . The first number of the array is taken in a register, say AL. The second number of the array is then compared with the first one. If the first one is greater than the second one, it is left unchanged. However, if the second one is greater than the first, the second number replaces the first one in the AL register. The procedure is repeated for every number in the array and thus it requires 15 iterations. At the end of 15th iteration the largest number will reside in the register AL.

```
        MOV CX, OF H   ; Initialize counter for number of iterations
        MOV AX, 2000H  ; Initialize data segment
        MOV DS, AX     ;
        MOV SI, 0500H  ; Initialize source pointer
        MOV AL, [SI]   ; Take first number in AL
BACK :  INC SI         ; Increment source pointer
        CMP AL, [SI]   ; Compare next number with the previous
        JNC NEXT       ; If the next number is larger
        MOV AL, [SI]   ; replace the previous one with the next
NEXT :  LOOPBACK       ; Repeat the procedure 15 times
        HLT
```

- ## Assembly language programs

Write a program for addition of two numbers.

Solution The following program adds two 16-bit operands. There are various methods of specifying operands depending upon the addressing modes

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
OPR1 DW 1234H                   ; 1st operand
OPR2    DW 0002H                ; 2nd operand
RESULT  DW 01 DUP(?)            ; A word of memory reserved for re-
                                  sult
DATA    ENDS
CODE    SEGMENT
START:  MOV AX, DATA            ; Initialize data segment
        MOV DS, AX              ;
        MOV AX, OPR1            ; Take 1st operand in AX
        MOV BX, OPR2            ; Take 2nd operand in BX
        CLC                     ; Clear previous carry if any
        ADD AX, BX              ; Add BX to AX
        MOV DI, OFFSET RESULT   ; Take offset of RESULT in DI
        MOV [DI], AX            ; Store the result at memory address in DI
        MOV AH, 4CH             ; Return to DOS prompt
        INT 21H
CODE    ENDS                    ; CODE segment ends
        END START               ; Program ends
```

A program to find out the largest number from a given unordered array of 8-bit numbers, stored in the locations starting from a known address.

Solution Compare the *i*th number of the series with the (*i*+1)th number using CMP instruction. It will set the flags appropriately, depending upon whether the ith number or the (*i*+1)th number is greater. If the *i*th number is greater than (*i*+1)th, leave it in AX (any register may be used). Otherwise, load the (*i*+1)th number in AX, replacing the *i*th number in AX. The procedure is repeated till all the members in the array have been compared.

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT                        ; Data segment starts
LIST DB 52H, 23H, 56H, 45H, --      ; List of byte numbers
COUNT EQU OF                        ; Number of bytes in the list
LARGEST DB 01H DUP(?)               ; One byte is reserved for the largest
                                      number.
DATA ENDS                           ; Data segment ends
CODE SEGMENT                        ; Code segment starts.
START:      MOV AX, DATA            ; Initialize data segment.
            MOV DS, AX
            MOV SI, OFFSET LIST
            MOV CL, COUNT           ; Number of bytes in CL.
            MOV AL, [SI]            ; Take the first number in AL
AGAIN:      CMP AL, [SI+1]          ; and compare it with the next number.
            JNL NEXT

            MOV AL, [SI+1]
NEXT:       INC SI                  ; Increment pointer to the byte list.
            DEC CL                  ; Decrement counter.
            JNZ AGAIN               ; If all numbers are compared, point to
                                      result
            MOV SI, OFFSET LARGEST  ; destination and store it.
            MOV [SI], AL
            MOV AH, 4CH             ; Return to DOS.
            INT 21H
            CODE ENDS
END         START
```

Program to find the largest of unordered array of series of words using subroutine

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
LIST DW 1234H, 2354H, 0056H, 045AH, -
COUNT EQU OF
LARGEST DW 01H DUP(?)
DATA ENDS
CODE SEGMENT
START:      MOV AX, DATA
            MOV DS, AX
            MOV SI, OFFSET LIST

                  MOV CL, COUNT
                  MOV AX, [SI]

 CALL LARGE
 MOV SI,OFFSET LARGEST
 MOV [SI],AX
MOV AH,4CH
INT 21H
PROCEDURE  LARGE NEAR


 AGAIN:           CMP AX, [SI+2]
                  JNL NEXT
                  MOV AX, [SI+2]
 NEXT:            INC SI
                  INC SI
                  DEC CL
                  JNZ AGAIN
RET
LARGE ENDP

 CODE             ENDS
                  END        START
```

- Write a ALP to perform BCD to binary conversion
- ASSUME CS:CODE
- CODE SEGMENT
- ORG 1000H
- MOV BX,1100H
- MOV AL,[BX]
- MOV DL,AL
- AND DL, OFH
- MOV CL,4
- ROR AL,CL
- AND AL,OFH
- MOV DH,OAH
- MUL DH
- ADD AL,DL
- MOV [BX+1],AL
- HLT
- CODE ENDS
- END

//67
DL=7
AL=6
6x10+7=AL
//43

Write a program for the addition of two 3 × 3 matrices. The matrices are stored in the form of lists (row wise). Store the result of addition in the third list.

Solution In the addition of two matrices, the corresponding elements are added to form the corresponding elements of the result matrix as shown:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}+b_{11}+a_{12}+b_{12}+a_{13}+b_{13} \\ a_{21}+b_{21}+a_{22}+b_{22}+a_{23}+b_{23} \\ a_{31}+b_{31}+a_{32}+b_{32}+a_{33}+b_{33} \end{bmatrix}$$

$$[A] \quad + \quad [B] \quad = \quad [A + B]$$

The matrix *A* is stored in the memory at an offset MAT1, as given:
a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, a_{31}, a_{32}, a_{33}, etc.
A total of 3 × 3 = 9 additions are to be done. The assembly language program is written as

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
            DIM EQU 09H
            MAT1 DB 01, 02, 03, 04, 05, 06, 07, 08, 09
            MAT2 DB 01, 02, 03, 04, 05, 06, 07, 08, 09
            RMAT3 DW 09H DUP(?)
DATA ENDS
CODE SEGMENT
START:      MOV AX, DATA
            MOV DS, AX
            MOV CX, DIM
            MOV SI, OFFSET MAT1
            MOV DI, OFFSET MAT2
            MOV BX, OFFSET RMAT3
```

```
NEXT:       XOR AX, AX
            MOV AL, [SI]
            ADD AL, [DI]
            MOV WORD PTR [BX], AX
            INC SI
            INC DI
            ADD BX, 02
            LOOP NEXT
            MOV AH, 4CH
            INT 21H
CODE        ENDS
            END START
```

Write an assembly language program to arrange a given series of hexadecimal bytes in ascending order.

Solution The algorithm used here is called *bubble sorting*. To start with, the first number of the series is compared with the second one. If the first number is greater than second, exchange their positions in the series otherwise leave the positions unchanged. Then, compare the second number in the recent form of the series with third and repeat the exchange part that you have carried out for the first and the second number, and for all the remaining numbers of the series. Repeat this procedure for the complete series $(n-1)$ times. After $(n-1)$ iterations, you will get the largest number at the end of the series, where n is the length of the series. Continue this till the complete series is arranged in ascending order. Let the series be as given:

```
53 , 25 , 19, 02          n = 4
25 , 53 , 19, 02          1st operation
25 , 19 , 53 , 02         2nd operation
25 , 19 , 02 , 53         3rd operation
largest no.          ⇒    4 – 1 = 3 operations
19 , 25 , 02 , 53         1st operation
19 , 02 , 25 , 53         2nd operation
2nd largest number   ⇒    4 – 2 = 2 operations
02 , 19 , 25 , 53         1st operation
3rd largest number   ⇒    4 – 3 = 1 operations
```

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
LIST DW 53H, 25H, 19H, 02H
COUNT EQU 04
DATA ENDS
CODE SEGMENT
START:      MOV AX, DATA
            MOV DS, AX
            MOV DX, COUNT-1
AGAINO:     MOV CX, DX
            MOV SI, OFFSET LIST
AGAIN1:     MOV AX, [SI]
```

```
            CMP AX, [SI+2]
            JL  PR1
            XCHG [SI+2], AX
            XCHG [SI], AX
PR1:        ADD SI, 02
            LOOP AGAIN1
            DEC DX
            JNZ AGAINO
            MOV AH, 4CH
            INT 21H
CODE        ENDS
            END START
```

Display the message "The study of microprocessors is interesting." on the CRT screen of a microcomputer.

Solution

```
ASSUME     CS :CODE, DS :DATA
DATA       SEGMENT
MESSAGE    DB ODH, OAH,"  STUDY OF MICROPROCESSORS IS INTERESTING",
                          ODH, OAH, "$"
                          ;PREPARING STRING OF THE MESSAGE
DATA       ENDS
CODE       SEGMENT
START:     MOV AX, DATA    ;INITIALIZE DS
           MOV DS, AX
           MOV AH, 09H     ;SET FUNCTION VALUE  FOR DISPLAY
           MOV DX, OFFSET MESSAGE
           INT 21H         ;POINT TO MESSAGE AND RUN
           MOV AH, 4CH     ;THE INTERRUPT
           INT 21H         ;RETURN TO DOS
CODE       ENDS            ;STOP
           END START
```