UNIT – II: 8086 PROGRAMMING

Program Development Steps:

Major steps in developing an assembly language program

- Defining the Problem
- Representing program operations
- Finding the right instructions
- Writing a program

Defining the problems

- Find out the problem.

  For example: Sensing temperature, detecting fire, detecting smock, decoder-encoder, Intelligent machine controller etc.

Representing program operations

- Formula or sequence of operations used to solve a programming problem is called as the algorithm.
- There are two ways of representing algorithms:
  - Flowchart
  - Structured programming and pseudo code

Finding the right instructions

- Instructions in 8086 are mainly divided into following categories
- Data Transfer Instructions
- Arithmetic Instruction
- Bit manipulation Instruction
- String Instruction
- Program execution transfer Instruction
- Processor control Instruction

Writing a program

- We need to do the following steps to write the program effectively:
  - INITIALIZATION INSTRUCTIONS: used to initialize various parts of the program like segment registers, flags and programmable port devices.
  - STANDARD PROGRAM FORMAT: it's a tabular format containing ADDRESS, DATA OR CODE, LABELS, MNEM, OPERAND(S) and COMMENTS as the columns.

- DOCUMENTATION: you should document the program. E.g. each page of the document contains page number and name of the program, heading block containing the abstract about the program, comments should be added wherever necessary.

Instruction Set of 8086:

The 8086 instructions are categorized into the following main types.

a. Data Copy / Transfer Instructions
b. Arithmetic and Logical Instructions
c. Branch Instructions
d. Loop Instructions
e. Machine Control Instructions
f. Flag Manipulation Instructions
g. Shift and Rotate Instructions
h. String Instructions

a) Data Transfer Instruction:

These types of instructions are used to transfer data from source operand to destination operand. All the store, load, move, exchange, input and output operations belong to this category.

b) Arithmetic and Logical Instructions:

All the instructions performing arithmetic, logical, increment, decrement, compare and scan instructions belong to this category.

c) Branch Instructions:

These instructions transfer control of execution to the specified address. All the call, jump, interrupt and return instructions belong to this category.

d) Loop Instructions:

The *LOOP*, *LOOPNZ* and *LOOPZ* instructions belong to this category. These are useful to implement different loop structures.

e) Machine control Instructions:

These instructions control the machine status. *NOP*, *HLT*, *WAIT* and *LOCK* instructions belongs to this category.

f) Flag Manipulation Instructions:

All instructions which directly affect the flag register belong to this category. The instructions *CLD, STD, CLI, STI* etc. belong to this category.

g) Shift and Rotate Instructions:

These instructions involve the bitwise shifting or rotation in either direction with or without a count in *CX*.

h) String Instructions:

These instructions involve string manipulation operations like load, scan, compare, store etc. These instructions are only to be operated upon the string.

a) Data Copy / Transfer Instructions: -

MOV:

This instruction copies a word or a byte of data from some source to a destination.

The destination can be a register or a memory location. The source can be a register, a memory location, or an immediate number.

MOV AX, BX

MOV AX, 5000H

MOV AX, [SI]

MOV AX, [2000H]

MOV AX, 50H [BX]

MOV [734AH], BX

MOV DS, CX

MOV CL, [357AH]

Direct loading of the segment registers with immediate data is not permitted.

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.

Ex: - PUSH AX

PUSH DS

PUSH [5000H]

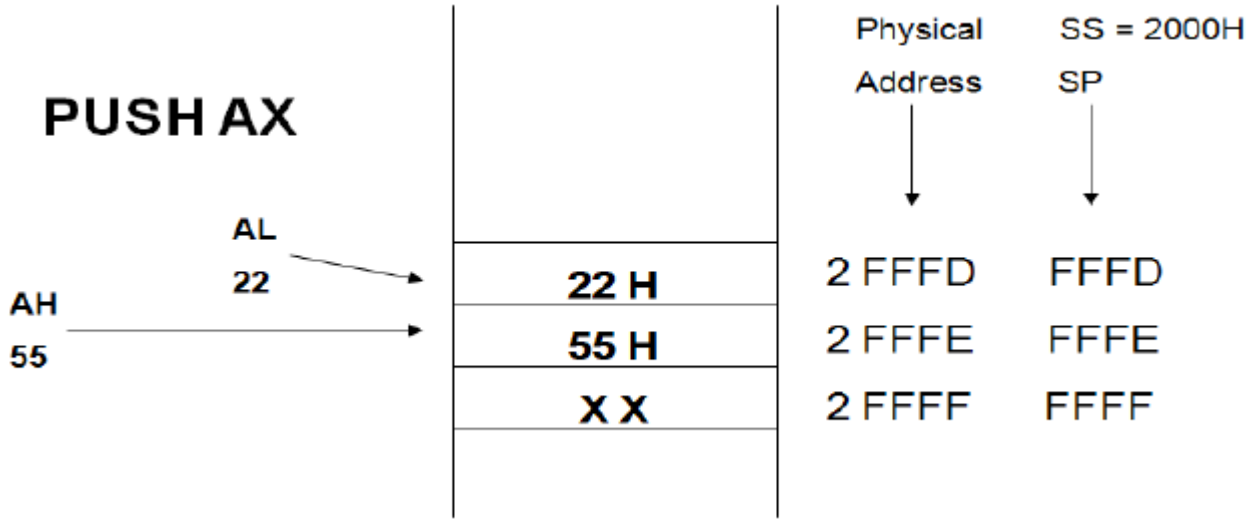


Fig: Push data to Stack memory

POP: Pop from Sack

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. The stack pointer is incremented by 2

Ex: - POP AX

POP DS

POP [5000H]

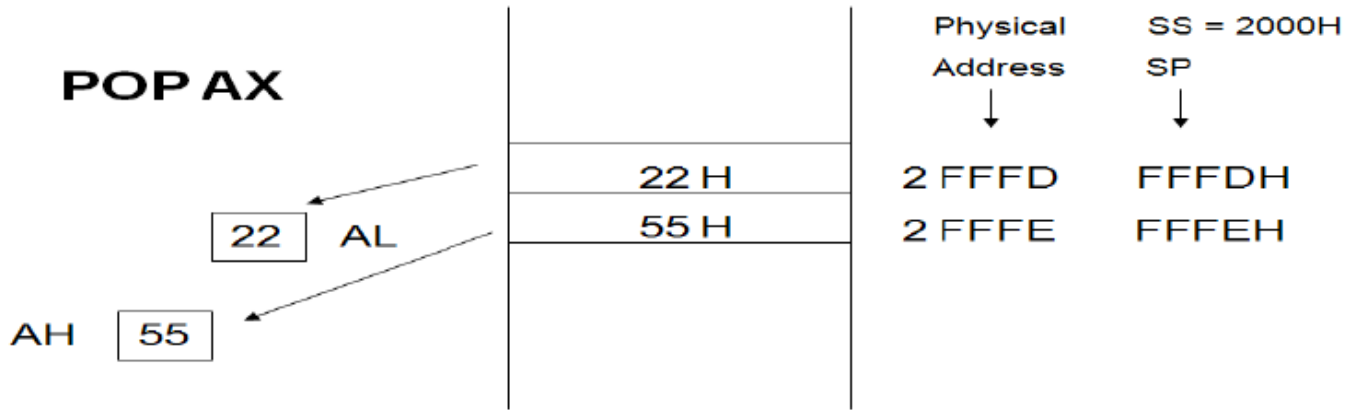


Fig: Poping Register content from stack memory

XCHG: Exchange byte or word

This instruction exchanges the contents of the specified source and destination operands

Ex: - XCHG [5000H], AX

XCHG BX, AX

XLAT: Translate byte using look-up table

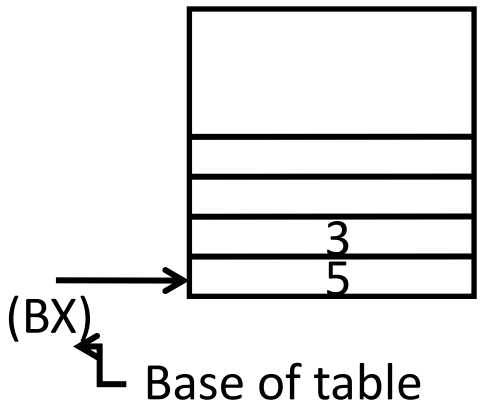
Ex: - MOV BX, OFFSET TABLE

MOV AL, 00H

XLAT

(AL) ← 5

AL← [BX+AL]



Simple input and output port transfer Instructions:

IN:

Copy a byte or word from specified port to accumulator.

Ex: - IN AL,03H

IN AX, DX

OUT:

Copy a byte or word from accumulator specified port.

Ex: - OUT 03H, AL

OUT DX, AX

LEA:

Load effective address of operand in specified register.

[reg] offset portion of address in DS

Ex: - LEA reg, offset

LDS:

Load DS register and other specified register from memory.

[reg] [mem]

[DS] [mem + 2]

Ex: - LDS reg, mem

LES:

Load ES register and other specified register from memory.

[reg] [mem]

[ES] [mem + 2]

Ex: - LES reg, mem

Flag transfer instructions:

LAHF:

Load (copy to) AH with the low byte the flag register.

[AH] [Flags low byte]

Ex: - LAHF

SAHF:

Store (copy) AH register to low byte of flag register.

[Flags low byte] ← [AH]

Ex: - SAHF

PUSHF:

Copy flag register to top of stack.

[SP] ← [SP] – 2

[[SP]] ← [Flags]

Ex: - PUSHF

POPF:

Copy word at top of stack to flag register.

[Flags] ← [[SP]]

[SP] ← [SP] + 2

Ex: - POPF

b) Arithmetic Instructions:

The 8086 provides many arithmetic operations: addition, subtraction, negation, multiplication and comparing two values.

ADD:

The add instruction adds the contents of the source operand to the destination operand.

Ex: - ADD AX, 0100H

ADD AX, BX

ADD AX, [SI]

ADD AX, [5000H]

ADD [5000H], 0100H

ADD 0100H

ADC: Add with Carry

This instruction performs the same operation as ADD instruction, but adds the carry flag to the result.

Ex: - ADC 0100H

ADC AX, BX

ADC AX, [SI]

ADC AX, [5000]

ADC [5000], 0100H

SUB: Subtract

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

Ex: - SUB AX, 0100H

SUB AX, BX

SUB AX, [5000H]

SUB [5000H], 0100H

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

Ex: - SBB AX, 0100H

SBB AX, BX

SBB AX, [5000H]

SBB [5000H], 0100H

INC: Increment

This instruction increases the contents of the specified Register or memory location by 1. Immediate data cannot be operand of this instruction.

Ex: - INC AX

INC [BX]

INC [5000H]

DEC: Decrement

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

Ex: - DEC AX

DEC [5000H]

NEG: Negate

The negate instruction forms 2's complement of the specified destination in the instruction. The destination can be a register or a memory location. This instruction can be implemented by inverting each bit and adding 1 to it.

Ex: - NEG AL

AL = 0011 0101 35H Replace number in AL with its 2's complement

AL = 1100 1011 = CBH

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

Ex: - CMP BX, 0100H

CMP AX, 0100H

CMP [5000H], 0100H

CMP BX, [SI]

CMP BX, CX

MUL: Unsigned Multiplication Byte or Word

This instruction multiplies an unsigned byte or word by the contents of AL.

Ex: - MUL BH; (AX) ← (AL) x (BH)

MUL CX; (DX) (AX) ← (AX) x (CX)

MUL WORD PTR [SI]; (DX) (AX) ← (AX) x ([SI])

IMUL: Signed Multiplication

This instruction multiplies a signed byte in source operand by a signed byte in AL or a signed word in source operand by a signed word in AX.

Ex: - IMUL BH

IMUL CX

IMUL [SI]

CBW: Convert Signed Byte to Word

This instruction copies the sign of a byte in AL to all the bits in AH. AH is then said to be sign extension of AL.

Ex: - CBW

AX= 0000 0000 1001 1000 Convert signed byte in AL signed word in AX.

Result in AX = 1111 1111 1001 1000

CWD: Convert Signed Word to Double Word

This instruction copies the sign of a byte in AL to all the bits in AH. AH is then said to be sign extension of AL.

Ex: - CWD

Convert signed word in AX to signed double word in DX: AX

DX= 1111 1111 1111 1111

Result in AX = 1111 0000 1100 0001

DIV: Unsigned division

This instruction is used to divide an unsigned word by a byte or to divide an unsigned double word by a word.

```
Ex: -  DIV CL      ; Word in AX / byte in CL
                   ; Quotient in AL, remainder in AH
       DIV CX      ; Double word in DX and AX / word in CX, and Quotient in AX, remainder in DX
```

AAA: ASCII Adjust After Addition

The **AAA** instruction is executed after an ADD instruction that adds two ASCII coded operand to give a byte of result in AL. The **AAA** instruction converts the resulting contents of AL to a unpacked decimal digits.

AAA operation:

1) In AL If rightmost nibble is >9 (ie) A to F or Auxiliary Flag=1
   ADD 6 to rightmost nibble
2) Clear left nibble form AL.
3) In AH ADD 1
4) Set Carry and Auxiliary Carry

```
Ex: -  ADD CL, DL          ; [CL] = 34H = ASCII for 4
                           ; [DL] = 38H = ASCII for 8
                           ; Result [CL] = 6CH
       AAA                 ; [AL] =02, unpacked BCD for 2
                           ; [AH] =01, unpacked BCD for 1
```

AAS: ASCII Adjust AL after Subtraction

This instruction corrects the result in AL register after subtracting two unpacked ASCII operands. The result is in unpacked decimal format. The procedure is similar to AAA instruction except for the subtraction of 06 from AL.

AAS operation:

1) AAS checks the rightmost nibble in AL If rightmost nibble is >9 (ie) A to F Or Auxiliary Flag=1, Then Subtract 6 from rightmost nibble
2) Clear left nibble in AL.
3) Subtracts 1 from AH

4) Set Carry and Auxiliary Carry

Ex: - MOV AL, 34H

SUB AL, 38H ; AX=00FC

AAS ; AX= FF06 ten's complement i.e -4 (Borrow one from AH)

OR AL, 30H ; AL=34

AAM: ASCII Adjust after Multiplication

This instruction, after execution, converts the product available In AL into unpacked BCD format.

AAM performs the following operations

1) Divides AL value by 10 (0AH)

2) Stores Quotient in AH

3) Store Remainder in AL

Ex: - MOV AL, 04 ; AL = 04

MOV BL, 09 ; BL = 09

MUL BL ; AX = AL*BL ; AX=0024H

AAM ; AH = 03H, AL=06H

AAD: ASCII Adjust before Division

This 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. In the instruction sequence, this instruction appears Before DIV instruction.

Operations done by AAD instruction

1) AAD multiplies the AH by 10(0Ah).

2) Then adds the product to AL and clears the AH

Ex: - 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. Where 3A is the hexadecimal equivalent of 58 (decimal).

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.

Ex: - AL = 53H, CL = 29H

ADD AL, CL ; AL ← (AL) + (CL)

; AL ← 53 + 29

; AL ← 7C

DAA ; AL ← 7C + 06 (as C > 9)

; AL 82

DAS: Decimal Adjust after Subtraction

This instruction converts the result of the subtraction of two packed BCD numbers to a valid BCD number. The subtraction has to be in AL only.

Ex: - AL = 75H, BH = 46H

```
SUB AL, BH      ; AL ← 2 F_H = (AL) - (BH)
                ; AF = 1
DAS             ; AL ← 29 (as F>9, F - 6 = 9)
```

Logical Instructions:

AND: Logical AND

This instruction bit by bit ANDs the source operand that may be an immediate register or a memory location to the destination operand that may a register or a memory location. The result is stored in the destination operand.

Ex: - AND AX, 0008H

AND AX, BX

OR: Logical OR

This instruction bit by bit ORs the source operand that may be an immediate, register or a memory location to the destination operand that may a register or a memory location. The result is stored in the destination operand.

Ex: - OR AX, 0008H

OR AX, BX

NOT: Logical Invert

This instruction complements the contents of an operand register or a memory location, bit by bit.

Ex: - NOT AX

NOT [5000H]

XOR: Logical Exclusive OR

This instruction bit by bit XORs the source operand that may be an immediate, register or a memory location to the destination operand that may a register or a memory location. The result is stored in the destination operand.

Ex: - XOR AX, 0098H

XOR AX, BX

TEST: Logical Compare Instruction

The TEST instruction performs a bit by bit logical AND operation on the two operands. The result of this AND ing operation is not available for further use, but flags are affected.

Ex: - TEST AX, BX

TEST [0500], 06H

c) Branch Instructions:

Branch Instructions transfers 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 to be transferred.

The Branch Instructions are classified into two types

i. Unconditional Branch Instructions.

ii. Conditional Branch Instructions.

Unconditional Branch Instructions:

In 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.

CALL: Unconditional Call

This instruction is used to call a Subroutine (Procedure) from a main program. Address of procedure may be specified directly or indirectly.

There are two types of procedure depending upon whether it is available in the same segment or in another segment.

i. Near CALL i.e., ±32K displacement.

ii. Far CALL i.e., anywhere outside the segment.

On execution this instruction stores the incremented IP & CS onto the stack and loads the CS & IP registers with segment and offset addresses of the procedure to be called.

RET: Return from 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 execution of the main program continues further.

INT N: Interrupt Type N.

In the interrupt structure of 8086, 256 interrupts are defined corresponding to the types from 00H to FFH. When 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 memory block in 0000 segment.

INTO: Interrupt on Overflow

This instruction is executed, when the overflow flag OF is set. 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. No Flags are affected by this instruction.

IRET: Return from ISR

When it is executed, the values of IP, CS and Flags are retrieved from the stack to continue the execution of the main program.

d) LOOP Instructions:

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 and JUMP IF NOT ZERO structure.

```
Ex: -       MOV CX, 0004H
            MOV BX, 7526H
    Label 1: MOV AX, CODE
            OR BX, AX
            LOOP Label 1
```

Conditional Branch Instructions:

When this instruction is executed, execution control is transferred to the address specified relatively in the instruction, provided the condition implicit in the opcode is satisfied. Otherwise execution continues sequentially.

JZ/JE Label

Transfer execution control to address 'Label', if ZF=1.

JNZ/JNE Label

Transfer execution control to address 'Label', if ZF=0

JS Label

Transfer execution control to address 'Label', if SF=1.

JNS Label

Transfer execution control to address 'Label', if SF=0.

JO Label

Transfer execution control to address 'Label', if OF=1.

JNO Label

Transfer execution control to address 'Label', if OF=0.

JNP Label

Transfer execution control to address 'Label', if PF=0.

JP Label

Transfer execution control to address 'Label', if PF=1.

JB Label

Transfer execution control to address 'Label', if CF=1.

JNB Label

Transfer execution control to address 'Label', if CF=0.

JCXZ Label

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

Conditional LOOP

LOOPZ / LOOPE Label

Loop through a sequence of instructions from label while ZF=1 and CX=0.

LOOPNZ / LOOPENE Label

Loop through a sequence of instructions from label while ZF=1 and CX=0.

e) Flag Manipulation and Processor Control Instructions:

These instructions control the functioning of the available hardware inside the processor chip. These instructions are categorized into two types:

1. Flag Manipulation instructions.
2. Machine Control instructions.

Flag Manipulation instructions:

The Flag manipulation instructions directly modify some of the Flags of 8086.

i. CLC – Clear Carry Flag.

ii. CMC – Complement Carry Flag.

iii. STC – Set Carry Flag.

iv. CLD – Clear Direction Flag.

v. STD – Set Direction Flag.

vi. CLI – Clear Interrupt Flag.

vii. STI – Set Interrupt Flag.

Machine Control instructions

The Machine control instructions control the bus usage and execution

i. WAIT – Wait for Test input pin to go low.

ii. HLT – Halt the process.

iii. NOP – No operation.

iv. ESC – Escape to external device like NDP

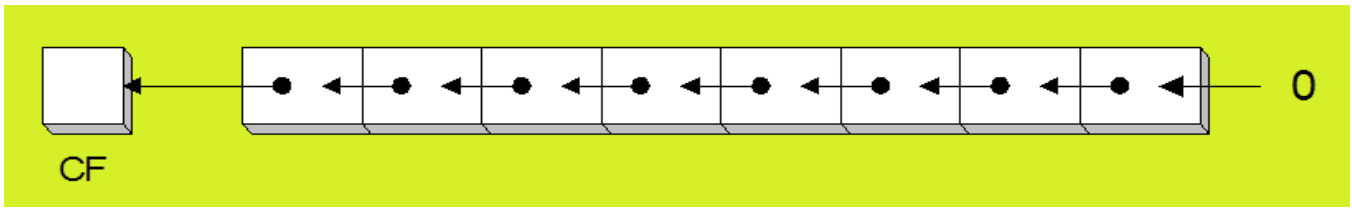
v. LOCK – Bus lock instruction prefix.

f) Shift & Rotate Instructions:

SAL/SHL: SAL / SHL destination, count

SAL and SHL are two mnemonics for the same instruction. This instruction shifts each bit in the specified destination to the left and 0 is stored at LSB position. The MSB is shifted into the carry flag. The destination can be a byte or a word.

It can be in a register or in a memory location. The number of shifts is indicated by count.



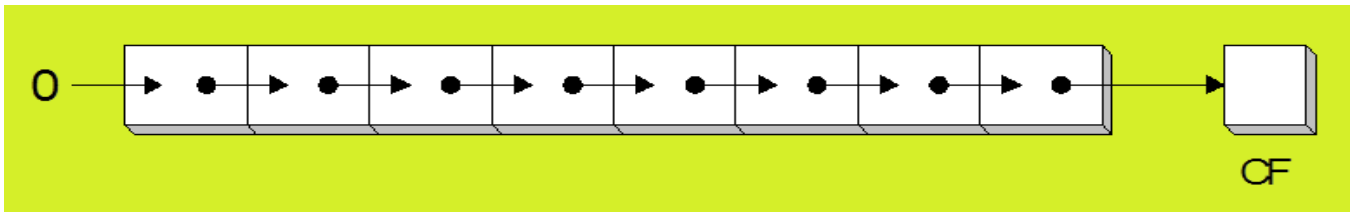
Ex: - SAL CX, 1

SAL AX, CL

SHR: SHR destination, count

This instruction shifts each bit in the specified destination to the right and 0 is stored at MSB position. The LSB is shifted into the carry flag. The destination can be a byte or a word.

It can be a register or in a memory location. The number of shifts is indicated by count.



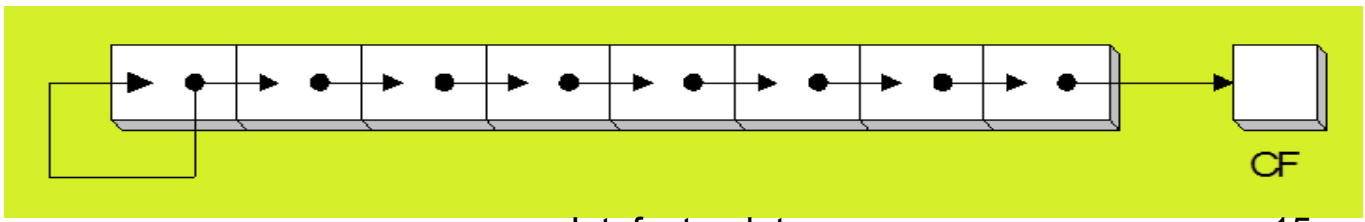
Ex: - SHR CX, 1

MOV CL, 05H

SHR AX, CL

SAR: SAR destination, count

This instruction shifts each bit in the specified destination some number of bit positions to the right. As a bit is shifted out of the MSB position, a copy of the old MSB is put in the MSB position. The LSB will be shifted into CF.



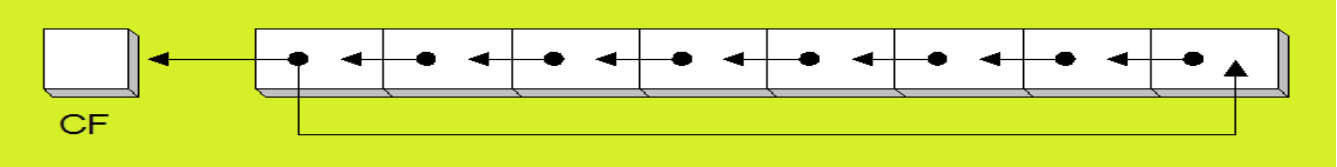
Ex: - SAR BL, 1

MOV CL, 04H

SAR DX, CL

ROL Instruction: ROL destination, count

This instruction rotates all bits in a specified byte or word to the *left* some number of bit positions. MSB is placed as a new LSB and a new CF.



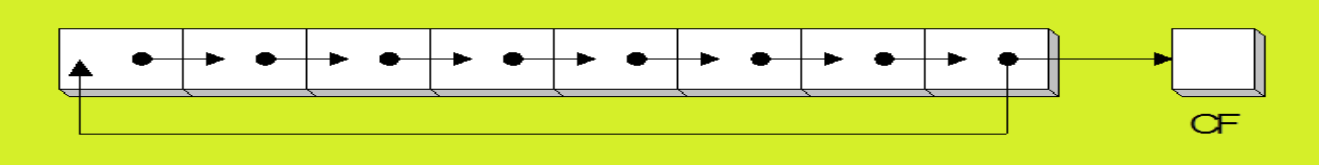
Ex: - ROL CX, 1

MOV CL, 03H

ROL BL, CL

ROR Instruction: ROR destination, count

This instruction rotates all bits in a specified byte or word to the *right* some number of bit positions. LSB is placed as a new MSB and a new CF.



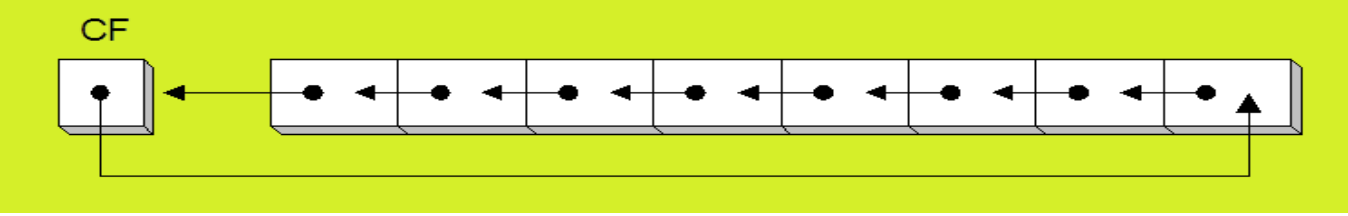
Ex: - ROR CX, 1

MOV CL, 03H

ROR BL, CL

RCL Instruction: RCL destination, count

This instruction rotates all bits in a specified byte or word some number of bit positions to the *left along with the carry flag*. MSB is placed as a new carry and previous carry is place as new LSB.



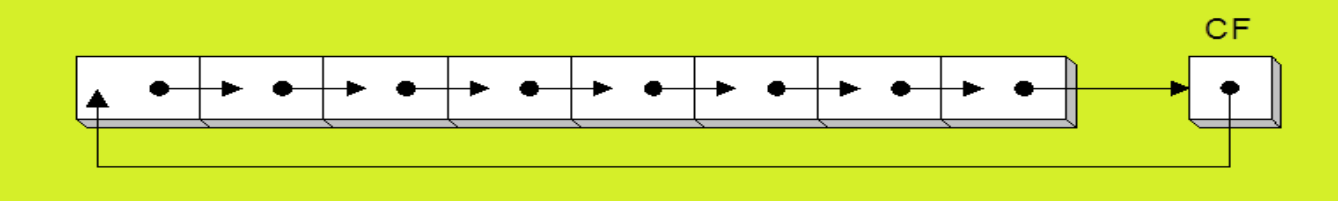
Ex: - RCL CX, 1

MOV CL, 04H

RCL AL, CL

RCR Instruction: RCR destination, count

This instruction rotates all bits in a specified byte or word some number of bit positions to the right *along with the carry flag*. LSB is placed as a new carry and previous carry is place as new MSB.



Ex: - RCR CX, 1

MOV CL, 04H

RCR AL, CL

g) String Manipulation Instructions:

A series of data byte or word available in memory at consecutive locations, to be referred as Byte String or Word String. A String of characters may be located in consecutive memory locations, where each character may be represented by its ASCII equivalent.

The 8086 supports a set of more powerful instructions for string manipulations for referring to a string, two parameters are required.

I. Starting and End Address of the String.

II. Length of the String.

The length of the string is usually stored as count in the CX register. The incrementing or decrementing of the pointer, in 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.

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).

i. REPE / REPZ - repeat operation while equal / zero.

ii. REPNE / REPNZ - repeat operation while not equal / not zero.

These are used for CMPS, SCAS instructions only, as instruction prefixes.

Prefix	Used with	Meaning
REP	MOVS STOS	Repeat while not end of string CX≠0
REPE/ REPZ	CMPS SCAS	CX≠0 & ZF=1
REPNE/REPNZ	CMPS SCAS	CX≠0 & ZF=0

Mnemonic	Meaning	Format	Operation	Flags affected
CLD	Clear DF	CLD	(DF) ← 0	DF
STD	Set DF	STD	(DF) ← 1	DF

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 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.

Ex: - Block move program using the move string instruction

```
        MOV AX, DATA SEG ADDR
        MOV DS, AX
        MOV ES, AX
        MOV SI, BLK 1 ADDR
        MOV DI, BLK 2 ADDR
        MOV CK, N
        CLD                    ; DF=0
NEXT:   MOV SB
        LOOP NEXT
        HLT
```

CMPSB/SW: Compare String Byte or String Word

- The CMPS instruction can be used to compare two strings of byte 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 REP instruction Prefix is used to repeat the operation till CX (counter) becomes zero or the condition specified by the REP Prefix is False.

SCANSB/SW: 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 s stored in CX.

The DF controls the mode for scanning of the string. 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.

LODSB/SW: 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, 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.

STOSB/SW: Store String Byte or String Word

- The STOS instruction Stores the AL / AX register contents to a location in the string pointer by ES: DI register pair. The DI is modified accordingly; No Flags are affected by this instruction.
- The direction Flag controls the String instruction execution, the source index SI and Destination Index DI are modified after each iteration automatically.
- If DF=1, then the execution follows auto-decrement mode, SI and DI are decremented automatically after each iteration.
- If DF=0, then the execution follows auto-increment mode. In this mode, SI and DI are incremented automatically after each iteration.

Ex:- Clearing a block of memory with a STOSB operation.

```
        MOV AX, 0
        MOV DS, AX
        MOV ES, AX
        MOV DI, A000
        MOV CX, OF
        CLD
AGAIN:  STOSB
        LOOPNE AGAIN
NEXT:   Clear A000 to A00F to 00H
```

Addressing Modes of 8086:

The default segment for the addressing modes using BP and SP is SS. For all other addressing modes, the default segments are DS or ES.

Addressing mode indicates a way of locating data or operands.

Different addressing modes of 8086:

1. Immediate:

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

Ex: - **MOV AX, 0050_H**

Here 0050_H is the immediate data and it is moved to register AX. The immediate data may be 8-bit or 16-bit in size.

2. Direct:

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

Ex: - **MOV AX, [1000_H]**

Here data resides in a memory location in the data segment, whose effective address is $10H \times DS + 1000H$

3. Register:

In 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.

Ex: - **MOV AX, BX**

4. Register Indirect:

In this addressing mode, the address of the memory location which contains data or operand is determined in an indirect way using offset registers. The offset address of data is in either BX or SI or DI register. The default segment register is either DS or ES.

Ex: - **MOV AX,** $[BX]$

The data is present in a memory location in DS whose offset is in BX. The effective address is $10H \times DS + [BX]$

5. Indexed:

In this addressing mode offset of the operand is stored in one of the index register. DS and ES are the default segments for index registers SI and DI respectively

Ex: - **MOV AX,** $[SI]$

The effective address of the data is $10H \times DS + [SI]$

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.

Ex: - **MOV AX, 50H**$[BX]$

The effective address of the data is $10H \times DS + 50H + [BX]$

7. Based Indexed:

In this addressing mode the effective address of the data is formed by adding the 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.

Ex: - **MOV AX,**$[BX][SI]$

The effective address is $10H \times DS + [BX] + [SI]$

8. Relative Based Indexed:

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

Ex: - **MOV AX, 50H**$[BX][SI]$

Here 50H is an immediate displacement. The effective address is $10H \times DS + [BX] + [SI] + 50H$.

9. Intrasegment Direct Mode:

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

10. Intrasegment Indirect Mode:

This mode is similar to intrasegment direct mode except the displacement to which control is to be transferred is passed to the instruction indirectly. Here the branch address is found as the content of a register or a memory location.

11. Intersegment Direct Mode:

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.

12. Intersegment Indirect Mode:

This mode is similar to intersegment direct mode except the address to which the control is to be transferred is passed to the instruction indirectly. This information is kept in a memory block of 4 bytes: IP (LSB), IP(MSB), LS(LSR) and CS(MSB) sequentially. The starting address of the memory block may be referred using any of the addressing modes, except immediate mode.

Assembler Directives and Operators:

The main advantage of machine language programming is that the memory control is directly in the hands of the programmer, so that, he may be able to manage the memory of the system more efficiently. On the other hand, the disadvantages are more prominent. The programming, coding and resource management techniques are tedious. The programmer has to take care of these functions hence the chances of human errors are more. The programs are difficult to understand unless one has a thorough technical knowledge of the processor architecture and instruction set.

The assembly language programming is simpler as compared to the machine language programming. The instruction mnemonics are directly written in the assembly language programs. The programs are now more readable to users than the machine language programs. The main improvement in assembly language over machine language is that the address values and the constants can be identified by labels. If the labels are suggestive, then certainly the program will become more understandable, and each time the programmer will not have to remember the different constants and the addresses at which they are stored, throughout the programs. The labels may help to identify the addresses and constants. Due to this facility, the tedious byte handling and manipulations are got rid of. Similarly, now different logical segments and routines may be assigned with the labels rather than the different addresses. The memory control feature of machine language programming is left unchanged by providing storage define facilities in assembly language programming. The documentation facility which was not possible with machine language programming is now available in assembly language.

An assembler is a program used to convert an assembly language program into the equivalent machine code modules which may further be converted to executable codes. The assembler decides the address of each label and substitutes the values for each of the constants and variables. It then forms the machine code for the mnemonics and data in the assembly language program. While doing these things, the assembler may find out syntax errors. The logical errors or other programming errors are not found out by the assembler. For completing all these tasks, an assembler needs some hints from the programmer, i.e. the required storage for a particular constant or variable, logical names of the segments, types of the different routines and modules, end of file, etc. These, types of hints are given to the assembler using some predefined alphabetical strings called assembler directives. Assembler directives help the assembler to correctly understand the assembly language programs to prepare the codes.

Another type of hint which helps the assembler to assign a particular constant with a label or initialize particular memory locations or labels with constants is called an operator. Rather, the operators perform the arithmetic and logical tasks unlike directives that just direct the assembler to correctly interpret the program to code it appropriately. The following directives are commonly used in the assembly language programming practice using Microsoft Macro Assembler (MASM) or Turbo Assembler (TASM).

DIRECTIVES:

DB: Define Byte

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

Example:

LIST DB 0lH, 02H, 03H, 04H

This statement directs the assembler to reserve four memory locations for a list named LIST and initialize them with the above specified four values.

MESSAGE DB 'GOOD MORNING'

This makes the assembler reserve the number of bytes of memory equal to the number of characters in the string named MESSAGE and initialize those locations by the ASCII equivalent of these characters.

VALUE DB 50H

This statement directs the assembler to reserve 50H memory bytes and leave them uninitialized for the variable named VALUE.

DW: Define Word.

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

Example:

WORDS DW 1234H, 4567H, 78ABH, 045CH

This makes the assembler reserve four words in memory (8 bytes), and initialize the words with the specified values in the statements.

Another option of the 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 label WDATA and initializes all the word locations with 6666H.

DQ: Define Quad word

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

DT: Define Ten Bytes.

The DT directive directs the assembler to define the specified variable requiring 10-bytes for its storage and initialize the 10 bytes with the specified values. The directive may be used in case of variables facing heavy numerical calculations, generally processed by numerical processors.

ASSUME: **Assume Logical Segment Name**

The ASSUME directive is used to inform the assembler, the names of the logical segments to be assumed for different segments used in the program.

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

END: END of Program

The END directive marks the end of an assembly language program.

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.

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

ENDP: END of Procedure.

In assembly language programming, the subroutines are called procedures. The ENDP directive is used to indicate the end of a procedure.

```
Ex: -   PROCEDURE STAR
                .
                .
                .
        STAR ENDP
```

EVEN: Align on Even Memory Address

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.

Ex: -

```
EVEN
PROCEDURE ROOT
.
.
.
ROOT ENDP
```

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.

Ex: -

```
LABEL EQU 0500H
ADDITION EQU ADD
```

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 MODULE 1 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. The statement of declaration EXTRN must be accompanied by the SEGMENT and ENDS directives of the MODULE 1, before it is called in MOBULE 2. Thus, the MODULE 1 and MODULE 2 must have the following declarations.

Ex: -

```
MODULEl SEGMENT

        PUBLIC FACTORIAL FAR


MODULEl ENDS
MODULE2 SEGMENT

        EXTRN FACTORIAL FAR


MODULE2 ENDS
```

GROUP: Group the Related segment

The directive is used to form logical groups of segments with similar purpose or type. This directive is used to inform the assembler to form a logical group of the following segment names.

Ex: - PROGRAM GROUP CODE, DATA, STACK

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

LABEL: Label

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

A LABEL directive may be used to make a FAR jump as shown below.

A FAR jump cannot be made at a normal label with a colon.

LENGTH: Byte Length of a Label

This directive is not available in TASM. This is used to refer to the length of a data array or a string.

Ex: - MOV CX, LENGTH ARRAY

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

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 while starting the assembly process for a module, the assembler initializes a location counter to keep track of the allotted addresses for the module.

If the ORG statement is not written in the program, the location counter is initialized to 0000.

PROC: Procedure

The PROC directive marks the start of a named procedure in the statement. Also, the types NEAR or FAR specify the type of the procedure, i.e whether it is to be called by the main program located within 64K of physical memory or not.

Ex: - RESULT PROC NEAR

ROUTINE PROC FAR

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.

OPERATORS:

OFFSET: Offset of a Label

When the assembler comes across the OFFSET operator along with a label, it first computes the 16-bit displacement (also called as offset interchangeably) of the particular label, and replaces the string 'OFFSET LABEL' by the computed displacement. This operator is used with arrays, strings, lables and procedures to decide their offsets in their default segments.

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

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.

```
Ex: -   MOV AL, BYTE PTR [SI]    ; Moves content of memory location addressed by SI (8-bit) to AL
        INC BYTE PTR [BX]        ; Increments byte contents of memory location addressed by BX
        MOV BX, WORD PTR [2000H] ; Moves 16-bit content of memory location 2000H to BX, i.e.
                                   [2000H] to BL [2001 H] to BH
        INC WORD PTR [3000H]     ; Increments word contents of memory location 3000H
                                   considering contents of 3000H (lower byte) and 3001 H
                                   (higher byte) as a 16-bit number
```

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'.

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

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).

```
Ex: -   JMP SHORT LABEL
```

UNIT - II

Assembly Language programs involving Logical, Branch & CALL Instructions:

write a program for the addition of a series of 8-bit numbers. The series contains 100 (numbers)

Sol:-

```
ASSUME CS: CODE , DS: DATA
DATA    SEGMENT
NUMLIST DB   52H , 23H , ....
COUNT   EQU  100D
RESULT  DW   01H  DUP (?)
DATA    ENDS
CODE    SEGMENT
        ORG  200H
        MOV  AX, DATA
        MOV  DS, AX
        MOV  CX, COUNT
        XOR  AX, AX
        XOR  BX, BX
        MOV  SI, OFFSET NUMLIST
AGAIN:  MOV  BL, [SI]
        ADD  AX, BX
        INC  SI
        DEC  CX
        JNZ  AGAIN
        MOV  DI, OFFSET RESULT
        MOV  [DI], AX
        MOV  AH, 4CH
        INT  21H
CODE    ENDS
END
```

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.

Sol:-

```
ASSUME  CS: CODE, DS: DATA
DATA    SEGMENT
LIST    DB  52H, 23H, 56H, 45H, ....
COUNT   EQU  OF
LARGEST DB  01H DUP (?)
DATA    ENDS

CODE    SEGMENT
START:      MOV   AX, DATA
            MOV   DS, AX
            MOV   SI, OFFSET LIST
            MOV   CL, COUNT
            MOV   AL, [SI]
AGAIN:      CMP   AL, [SI+1]
            JNL   NEXT
            MOV   AL, [SI+1]
NEXT:       INC   SI
            DEC   CL
            JNZ   AGAIN
            MOV   SI, OFFSET LARGEST
            MOV   [SI], AL
            MOV   AH, 4CH
            INT   21H
CODE    ENDS
END     START
```

AL: ~~52~~ 56

52 > 23

52 < 56

A

Write an ALP to arrange a given series of hexadecimal bytes in ascending order.

Sol:-

Note: In order to arrange the given series of hexadecimal bytes in ascending order, here we use Bubble sorting technique

Ex:- Series of nos 53, 25 19 02

Here $n=4$

- After $n-1$ (i.e 3) iterations we will get first largest number at the end of the series
- After $n-2$ (i.e 2) iterations we will get second largest number
- After $n-3$ (i.e 1) iteration we will get third largest number

i) 53 25 19 02

CMP 25 53 19 02 I
25 19 53 02 II
25 19 02 53 III } cycle 1

19 25 02 53 I
19 02 25 53 II } cycle 2

02 19 25 53 I cycle 3

```
DATA SEGMENT
    LIST DB 53H, 25H, 19H, 02H
    COUNT EQU 04H
DATA ENDS
```

```
ASSUME CS: CODE, DS: DATA
CODE SEGMENT
        START:  MOV  AX, DATA
                MOV  DS, AX
                XOR  AX, AX
                MOV  DX, COUNT-1
        AGAIN0: MOV  CX, DX
                MOV  SI, OFFSET LIST
        AGAIN1: MOV  AL, [SI]
                CMP  AL, [SI+1]
                JL   PR1
                XCHG [SI+1], AL
                XCHG [SI], AL
        PR1:    INC  SI
                LOOP AGAIN1
                DEC  DX
                JNZ  AGAIN0
                MOV  AH, 4CH
                INT  21H
CODE ENDS
        END START
```

#: Write an ALP to perform a one byte BCD addition (3)

Sol:-

Let 92_H

$+ \quad 59_H$

EB_H

Check the lower nibble in the result

i.e B > 9, then add 6 to it.

```
  1011
+ 0110
  ---
 10001
```

After that, check the higher nibble in the result,

i.e E > 9, then add 6 to it. also carry from previous nibble.

```
    1
  1110
+ 0110
  ----
 10101
```

Result:

cf	MSD	LSD
1	5	1

```
DATA SEGMENT
        OPR1  EQU  92H
        OPR2  EQU  59H
        RES   DB   02H DUP(00)
DATA ENDS
ASSUME CS:CODE, DS:DATA
CODE SEGMENT
        START: MOV AX, DATA
               MOV DS, AX
```

```
        MOV  BL, OPR1
        XOR  AL, AL
        MOV  AL, OPR2
        ADD  AL, BL
        DAA
        MOV  RES, AL
        JNC  MSB0
        INC  [RES+1].
MSB0:   MOV  AH, 4CH
        INT  21H
CODE ENDS
     END START
```

(4)

#: Write an ALP to find out the number of positive numbers and negative numbers from a given series of signed numbers.

Sol:-

Note: The status of carry flag, i.e, the MSB of the number will give the information about the sign of the number. during the number should ~~Shift~~ left ~~during~~

If CF = 1, the number is negative

If CF = 0, the number is positive

Ex:- i) 2579_H

i.e CF 0011 0101 0111 1001

SHL 0 0110 1010 1111 0010 ← padding

↓

Sign is +ve

ii) $A500_H$

i.e CF 1010 0101 0000 0000

SHL (1) 0100 1010 0000 0000 ←

↓

Sign is -ve

```
DATA SEGMENT
    LIST DW 2579H, 0A500H, 0C009H, 0159H, 0B900H
    COUNT EQU 05H
DATA ENDS
ASSUME CS: CODE, DS: DATA
CODE SEGMENT
        START: MOV AX, DATA
               MOV DS, AX
```

```
        XOR  BX, BX
        XOR  DX, DX
        XOR  AX, AX
        MOV  CL, COUNT
        MOV  SI, OFFSET LIST
AGAIN:  MOV  AX, [SI]
        SHL  AX, 01H
        JC   NEG
        INC  BX
        JMP  NEXT
NEG:    INC  DX.
NEXT:   ADD  SI, 02H
        DEC  CL.
        JNZ  AGAIN
        MOV  AH, 4CH
        INT  21H

CODE ENDS
     END START
```

Note: From the above program,

After the program Execution, the result must be stored in the Registers BX, & DX

BX ← No. of positive numbers

DX ← No. of negative numbers

A program to find out the number of even and odd numbers from a given series of 16-bit hexadecimal-numbers.

Sol:-

```
ASSUME CS: CODE, DS: DATA
DATA SEGMENT
LIST DW 2357H, 0A579H, 0C322H, 0C91EH, 0C000H, 0957H
COUNT EQU 006H.
DATA ENDS
CODE SEGMENT
START:  XOR  BX, BX
        XOR  DX, DX
        MOV  AX, DATA
        MOV  DS, AX
        MOV  CL, COUNT
        MOV  SI, OFFSET LIST
AGAIN:  MOV  AX, [SI]
        ROR  AX, 01
        JC   ODD
        INC  BX
        JMP  NEXT
ODD:    INC  DX
NEXT:   ADD  SI, 02
        DEC  CL
        JNZ  AGAIN
        MOV  AH, 4CH
        INT  21H
CODE ENDS
END START
```

Assembly Language Programs involving String Manipulation Instructions:

Write a program to move a string of data words from offset 2000h to offset 3000h, the length of the string is 0Fh.

```
ASSUME CS: CODE, DS: DATA
DATA SEGMENT
SOURCESTRT EQU 2000H
DESTSTRT EQU 3000H
COUNT EQU 0FH
DATA ENDS
CODE SEGMENT

        MOV AX, DATA
        MOV DS, AX
        MOV ES, AX
        MOV SI, SOURCESTRT
        MOV DI, DESTSTRT
        MOV CX, COUNT
        CLD
REP     MOVSB
        MOV AH, 4CH
        INT 21H
CODE ENDS
END
```

Write a program to find out whether a given byte is in the string or not. If it is in the string, find out the relative address of the byte from the starting location of the string.

Sol:- The given string is scanned for the given byte. If it is found in the string, the zero flag is set, else, it is reset. Use of SCASB instruction is quite obvious here. A count should be maintained to find out the relative address of the byte found out.

```
ASSUME CS: CODE; DS: DATA
CODE SEGMENT
        MOV AX, DATA
        MOV DS, AX
        MOV ES, AX
        MOV CX, COUNT
        MOV DI, OFFSET STRING
        MOV BL, 00H
        MOV AL, BYTE1
SCAN1:  NOP
        SCASB [DI]
        JZ XXX
        INC BL
        LOOP SCAN1
XXX:    MOV AH, 4CH
        INT 21H
CODE ENDS
DATA SEGMENT
    BYTE1  EQU 25H
    COUNT  EQU 06H
    STRING DB 12H, 13H, 20H, 20H, 25H, 21H
DATA ENDS
END
```

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

Sol:- A program to display the string

```
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
        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 & Run
        MOV AH, 4Ch     ; the interrupt
        INT 21h         ; Return to DOS
CODE ENDS
END
```

STACK STRUCTURE OF 8086:

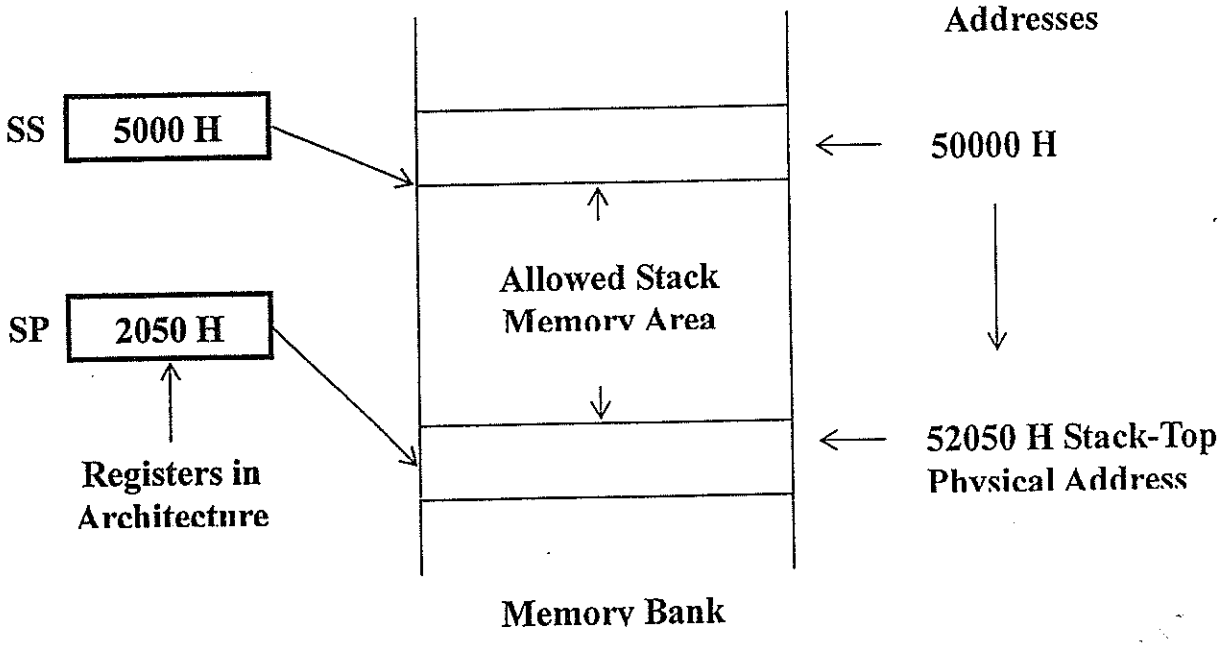
The stack is a block of memory that may be used for temporarily storing the contents of the registers inside the CPU. It is a top-down data structure whose elements are accessed using the stack pointer (SP) which gets decremented by two as we store a data word into the stack and gets incremented by two as we retrieve a data word from the stack back to the CPU register.

The process of storing the data in the stack is called 'pushing into' the stack and the reverse process of transferring the data back from the stack to the CPU register is known as 'popping off' the stack. The stack is essentially *Last-In-First-Out* (LIFO) data segment. This means that the data which is pushed into the stack last will be on top of stack and will be popped off the stack first. The stack pointer is a 16-bit register that contains the offset address of the memory location in the stack segment.

The stack segment, like any other segment, may have a memory block of a maximum of 64 Kbytes locations, and thus may overlap with any other segments. Stack Segment register (SS) contains the base address of the stack segment in the memory.

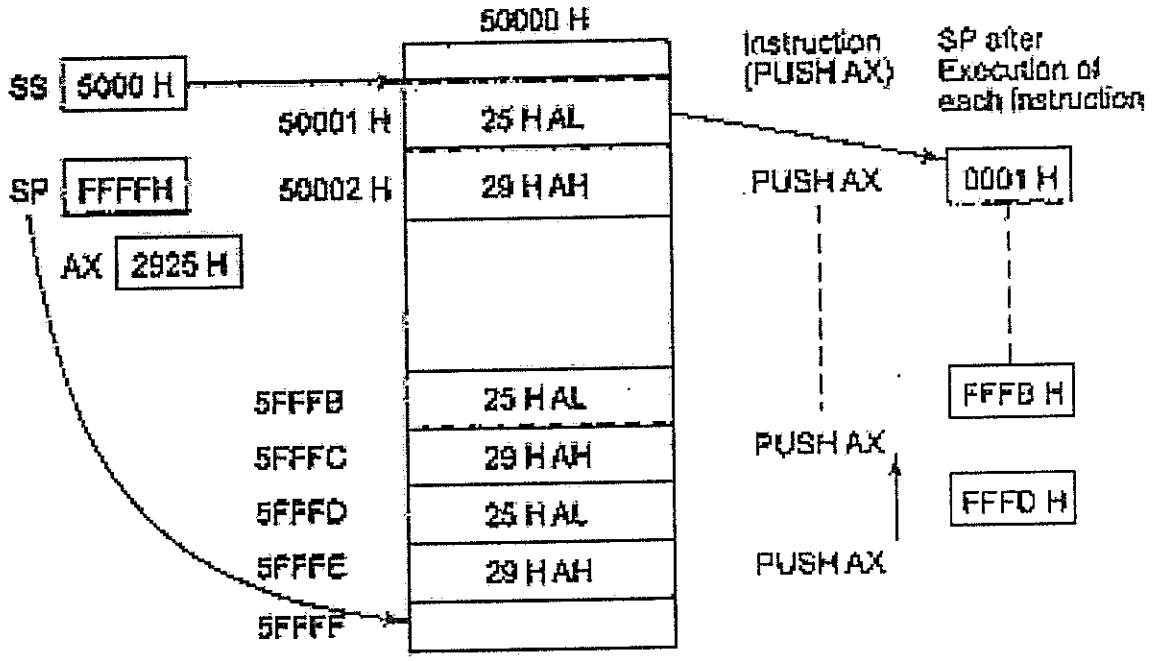
The Stack Segment register (SS) and Stack pointer register (SP) together address the stack-top as explained below:

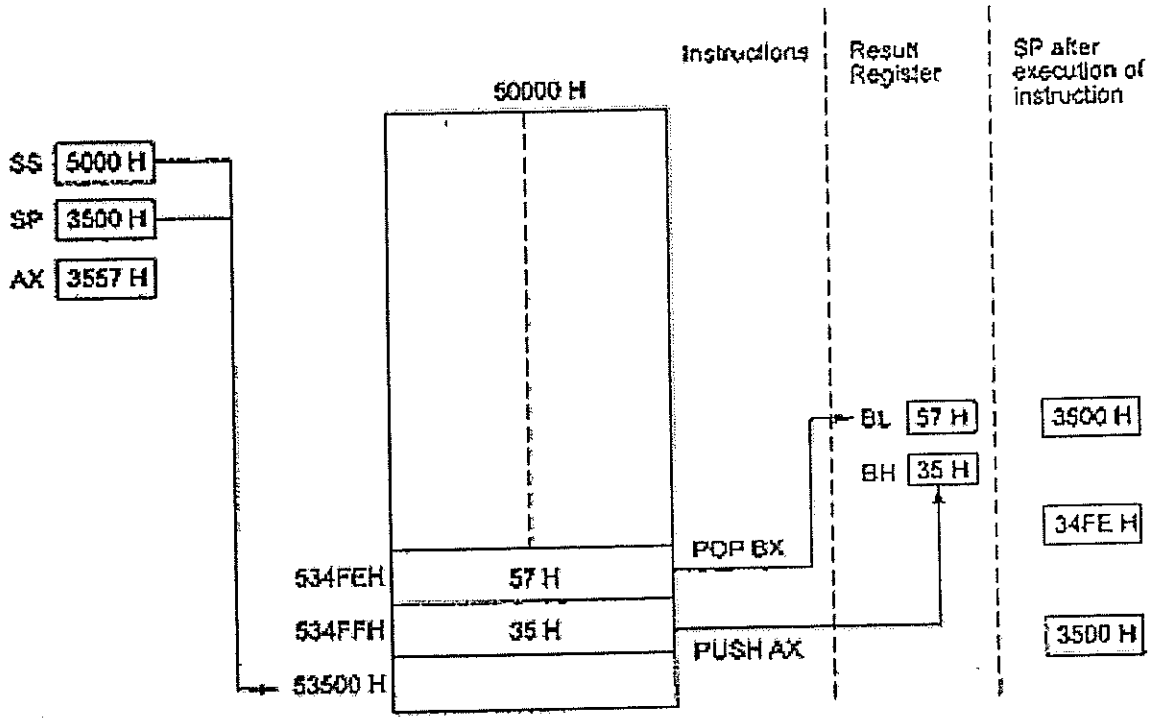
SS	⇒ 5000 H					
SP	⇒ 2050 H					
SS	⇒	0101	0000	0000	0000	
10H * SS	⇒	0101	0000	0000	0000	0000
	+					
SP	⇒		0010	0000	0101	0000
Stack-top address		0101 5	0010 2	0000 0	0101 5	0000 0



If the stack top points to a memory location 52050H, it means that the location 52050H is already occupied with the previously pushed data. 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 decremented contents of SP will be 204EH. This location will now be occupied by the recently pushed data.

Thus for a selected value of SS, the maximum value of SP=FFFFH and the segment can have maximum of 64K locations. If the SP starts with an initial value of FFFFH, it will be decremented by two, whenever a 16-bit data is pushed onto the stack. After successive push operations, when the stack pointer contains 0000H, any attempt to further push the data to the stack will result in stack overflow.





After a procedure is called using the CALL instruction, the IP is incremented to the next instruction. Then the contents of IP, CS and flag register are pushed automatically to the stack. The control is then transferred to the specified address in the CALL instruction i.e. starting address of the procedure. Then the procedure is executed.

Programming for Stack:

Program 1: Write an assembly language program to calculate squares of BCD numbers 0 to 9 and store them from 2000h offset onwards in the current data segment. The numbers and their squares are in the BCD format. Write a subroutine for the calculation of the square of a number.

Ans:

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

DATA SEGMENT
        ORG 2000H
        SQUARES DB 0FH DUP (?)
DATA ENDS


STACK SEGMENT
        SDATA DB 100H DUP (?)           ; Reserve 256 bytes for stack
STACK ENDS

CODE SEGMENT
        START: MOV AX,DATA              ;Initialize data segment
               MOV DS,AX
               MOV AX,STACK             ;Initialize stack segment
               MOV SS,AX
               MOV SP,OFFSET SDATA      ; Initialize stack pointer
               MOV CL,0AH               ; Initialize counter for numbers
               MOV SI,OFFSET SQUARES    ; Pointer for array of squares
               MOV AL,00H               ; Start from zero
      NEXTNUM: CALL SQUARE              ; Calculate square procedure
               MOV BYTE PTR [SI],AH     ; Store square in the array
               INC AL                   ; Go to next number
               INC SI                   ; Increment array pointer
               DEC CL                   ; Decrement counter
               JNZ NEXTNUM              ; Stop if CL=0, else continue
               MOV AH,4CH
               INT 21H
```

CL:10
AL:00

```
SQUARE PROC NEAR          ; Square is a near procedure
        MOV BH,AL
        MOV CH,AL
        XOR AL,AL
AGAIN:  ADD AL,BH         ; Successively add BH to AL
        DAA               ; Get BCD equivalent
        DEC CH            ; Decrement multiplier register
        JNZ AGAIN
        MOV AH,AL         ; Store the square of the number
        MOV AL,BH         ; Get back the number
        RET
SQUARE ENDP
CODE ENDS
    END START
```

Program 2: Write an ALP to change a sequence of sixteen 2-byte numbers from ascending to descending order. The numbers are stored in the data segment. Store the new series at addresses starting from 6000H. Use LIFO property of stack.

```
ASSUME CS: CODE, DS: DATA
DATA SEGMENT
    LIST DW 10H
    ORG 6000H
    RESULT DW 10H
    COUNT EQU 10H
    STACKDATA DB 100H DUP (?)
DATA ENDS

CODE SEGMENT
    START: MOV AX,DATA    ;Initialize data segment
           MOV DS,AX
           MOV SS,AX
           MOV SP,OFFSET LIST
           MOV CL,COUNT
           MOV BX, OFFSET RESULT+COUNT
    NEXT:  POP AX
```

```
        MOV DX,SP
        MOV SP,BX
        PUSH AX
        MOV BX,SP
        MOV SP,DX
        DEC CL
        JNZ NEXT
        MOV AH,4CH
        INT 21H
CODE ENDS
    END START
```

PROCEDURES: -

A procedure is a set of code that can be branched to and returned from in such a way that the code is as if it were inserted at the point from which it is branched to. The branch to procedure is referred to as the *call,* and the corresponding branch back is known as the *return.* The return is always made to the instruction immediately following the call regardless of where the call is located.

Calls, Returns, and Procedure Definitions

The CALL instruction not only branches to the indicated address, but also pushes the return address onto the stack. The RET instruction simply pops the return address from the stack. The registers used by the procedure need to be stored before their contents are changed, and then restored just before their contents are changed, and then restored just before the procedure is excited.

A CALL may be direct or indirect and, intra-segment or intersegment. If the CALL is intersegment, the return must be intersegment. Intersegment call must push both (IP) and (CS) onto the stack. The return must correspondingly pop two words from the stack. In the case of intra-segment call, only the contents of IP will be saved and retrieved when call and return instructions are used.

Procedures are used in the source code by placing a statement of the form at the beginning of the procedure

```
Procedure name    PROC    Attribute (i.e NEAR or FAR)
```

and by terminating the procedure with a statement

```
Procedure name ENDP
```

The attribute that can be used will be either NEAR or FAR. If the attribute is NEAR, the RET instruction will only pop a word into the IP register, but if it is FAR, it will also pop a word into the CS register.
A procedure may be in:

1. The same code segment as the statement that calls it.
2. A code segment that is different from the one containing the statement that calls it, but in the same source module as the calling statement.
3. A different source module and segment from the calling statement.

In the first case, the attribute could be NEAR provided that all calls are in the same code segment as the procedure. For the latter two cases the attribute must be FAR. If the procedure is given a FAR attribute, then all calls to it must be intersegment calls even if the call is from the same code segment. For the third case, the procedure name must be declared in EXTRN and PUBLIC statements.

Saving and Restoring Registers

When both the calling program and procedure share the same set of registers, it is necessary to save the registers when entering a procedure, and restore them before returning to the calling program.

```
MSK PROC NEAR
        PUSH AX
        PUSH BX
        PUSH CX
        POP CX
        POP BX
        POP AX
        RET
MSK ENDP
```

Procedure Communication

There are two general types of procedures; those operate on the same set of data and those that may process a different set of data each time they are called. If a procedure is in the same source module as the calling program, then the procedure can refer to the variables directly. When the procedure is in a separate source module it can still refer to the source module directly provided that the calling program contains the directive

```
PUBLIC ARY, COUNT, SUM

EXTRN ARY: WORD, COUNT: WORD, SUM: WORD
```

www.Jntufastupdates.com

Program 3: **Write an ALP to convert Binary number into its Decimal and then ASCII equivalent, and then display the number.**

```
ASSUME CS: CODE, SS: STACK
STACK SEGMENT
        STAKDATA DB 100H
STACK ENDS

CODE SEGMENT
        START: MOV AL,59H
                CALL BTA
                MOV AH,4CH
                INT 21H

        BTA PROC NEAR
                PUSH DX
                PUSH BX
                PUSH AX
                MOV AH,00H      ; Clear AH
                AAM             ; Convert to BCD
                ADD AX,3030H    ; Convert to ASCII
                MOV BX,AX       ; Save Result
                MOV DL,BH       ; Load first Digit (MSD)
                MOV AH,02H      ; Load function Number
                INT 21H         ; Display first Digit
                MOV DL,BL       ; Load second Digit
                INT 21H         ; Display second Digit
                POP AX
                POP BX
                POP DX
                RET
        BTA ENDP
CODE ENDS
        END START
```

INTERRUPTS AND INTERRUPT SERVICE ROUTINES:

Interrupt and its Need

The microprocessors allow normal program execution to be interrupted in order to carry out a specific task/work. The processor can be interrupted in the following ways

i) by an external signal generated by a peripheral,

ii) by an internal signal generated by a special instruction in the program,

iii) by an internal signal generated due to an exceptional condition which occurs while executing an instruction. (For example, in 8086 processors, divide by zero is an exceptional condition which initiates type 0 interrupt and such an interrupt is also called execution).

In general, the process of interrupting the normal program execution to carry out a specific task/work is referred to as interrupt.

The interrupt is initiated by a signal generated by an external device or by a signal generated internal to the processor. When a microprocessor receives, an interrupt signal it stops executing current normal program, save the status (or content) of various registers (IP, CS and flag registers in case of 8086) in stack and then the processor executes a subroutine/procedure in order to perform the specific task/work requested by the interrupt. The subroutine/procedure that is executed in response to an interrupt is also called Interrupt Service Subroutine. (ISR). At the end of ISR, the stored status of registers in stack is restored to respective registers, and the processor resumes the normal program execution from the point (instruction) where it was interrupted.

The external interrupts are used to implement interrupt driven data transfer scheme. The interrupts generated by special instructions are called software interrupts and they are used to implement system services/calls (or monitor services/calls). The system/monitor services are procedures developed by system designer for various operations and stored in memory. The user can call these services through software interrupts. The interrupts generated by exceptional conditions are used to implement error conditions in the system.

Interrupt Driven Data Transfer Scheme

The interrupts are useful for efficient data transfer between processor and peripheral. When a peripheral is ready for data transfer, it interrupts the processor by sending an appropriate signal. Upon receiving an interrupt signal, the processor suspends the current program execution, save the status in stack and executes an ISR to perform the data transfer between the peripheral and processor. At the end of ISR the processor status is restored from stack and processor resume its normal program execution. This type of data transfer scheme is called interrupt driven data transfer scheme.

The data transfer between the processor and peripheral devices can be implemented either by polling technique or by interrupt method. In polling technique, the processor has to periodically poll or

www.Jntufastupdates.com

check the status/readiness of the device and can perform data transfer only when the device is ready. In polling technique, the processor time is wasted, because the processor has to suspend its work and check the status of the device in predefined intervals.

Alternatively, if the device interrupts the processor to initiate a data transfer whenever it is ready then the processor time is effectively utilized because the processor need not suspend its work and check the status of the device in predefined intervals. For an example, consider the data transfer from a keyboard to the processor.

Normally a keyboard has to be checked by the processor once in every 10 milli seconds for a key press. Therefore, once in every 10 milli seconds the processor has to suspend its work and then check the keyboard for a valid key code. Alternatively, the keyboard can interrupt the processor, whenever a key is pressed and a valid key code is generated. In this way, the processor need not waste it's time to check the keyboard once in every 10 milli seconds.

Classification of Interrupts

In general, the interrupts can be classified in the following three ways:

1. Hardware and software interrupts
2. Vectored and Non-Vectored interrupt
3. Maskable and Non-Maskable interrupts

The interrupts initiated by external hardware by sending an appropriate signal to the interrupt pin of the processor is called hardware interrupt. The 8086 processor has two interrupt pins INTR and NMI. The interrupts initiated by applying appropriate signal to these pins are called hardware interrupts of 8086.

The software interrupts are program instructions. These instructions are inserted at desired locations in a program. While running a program, if software interrupt instruction is encountered then the processor initiates an interrupt. The 8086 processor has 256 types of software interrupts. The software interrupt instruction is INT n, where n is the type number in the range 0 to 255.

When an interrupt signal is accepted by the processor, if the program control automatically branches to a specific address (called vector address) then the interrupt is called vectored interrupt. The automatic branching to vector address is predefined by the manufacturer of processors. (In these vector addresses the interrupt service subroutines (ISR) are stored). In non-vectored interrupts the interrupting device should supply the address of the ISR to be executed in response to the interrupt. All the 8086 interrupts are vectored interrupts. The vector address for an 8086 interrupt is obtained from a vector table implemented in the first 1kb memory space (00000h to 03FFFh).

The processor has the facility for accepting or rejecting hardware interrupts. Programming the processor to reject an interrupt is referred to as masking or disabling and programming the processor to

accept an interrupt is referred to as unmasking or enabling. In 8086 the interrupt flag (IF) can be set to one to unmask or enable all hardware interrupts and IF is cleared to zero to mask or disable a hardware interrupts except NMI.

The interrupts whose request can be either accepted or rejected by the processor are called maskable interrupts. The interrupts whose request has to be definitely accepted (or cannot be rejected) by the processor are called non-maskable interrupts. Whenever a request is made by non-maskable interrupt, the processor has to definitely accept that request and service that interrupt by suspending its current program and executing an ISR. In 8086 processors, all the hardware interrupts initiated through INTR pin are maskable by clearing interrupt flag (IF). The interrupt initiated through NMI pin and all software interrupts are non-maskable.

Sources of Interrupts in 8086

An interrupt in 8086 can come from one of the following three sources.

1. One source is from an external signal applied to NMI or INTR input pin of the processor. The interrupts initiated by applying appropriate signals to these input pins are called hardware interrupts.
2. A second source of an interrupt is execution of the interrupt instruction "INT n", where n is the type number. The interrupts initiated by "INT n" instructions are called software interrupts.
3. The third source of an interrupt is from some condition produced in the 8086 by the execution of an instruction. An example of this type of interrupt is divide by zero interrupt. Program execution will be automatically interrupted if you attempt to divide an operand by zero. Such conditional interrupts are also, known as exceptions

Interrupts Cycle of 8086:

The 8086 microprocessor has 256 types of interrupts. INTEL has assigned a type number to each interrupt. The type numbers are in the range of 0 to 255. The 8086 processor has dual facility of initiating these 256 interrupts. The interrupts can be initiated either by executing "INT n" instruction where n is the type number or the interrupt can be initiated by sending an appropriate signal to INTR input pin of the processor.

For the interrupts initiated by software instruction" INT n ", the type number is specified by the instruction itself. When the interrupt is initiated through INTR pin, then the processor runs an interrupt acknowledge cycle to get the type number. (i.e., the interrupting device should supply the type number through D0- D7 lines when the processor requests for the same through interrupt acknowledge cycle).

The kinds of interrupts and their designated types are summarized in figure by illustrating the layout of their pointers within the memory. Only the first five types have explicit definitions; the other types may be used by interrupt instructions or external interrupts. From the figure, it is seen that the type associated with a division error interrupt is 0.

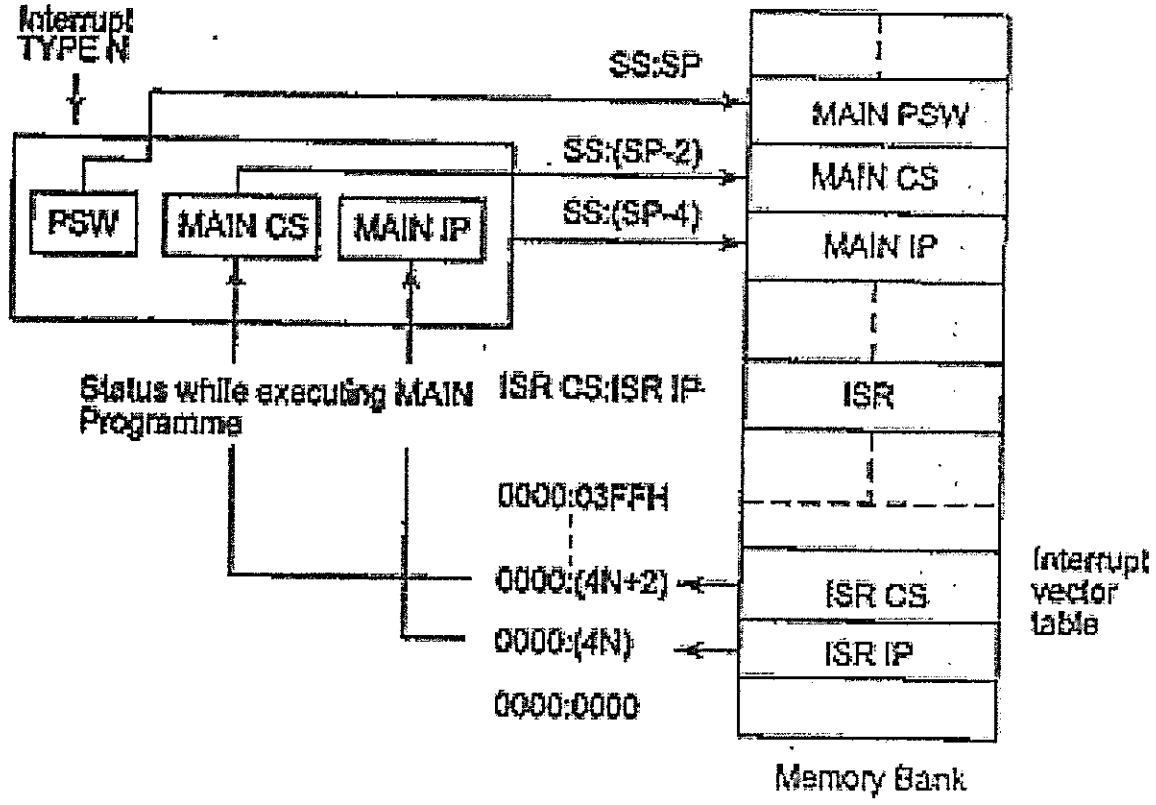
Therefore, if a division by 0 is attempted, the processor will push the current contents of the PSW, CS and IP into the stack, fill the IP and CS registers from the addresses 00000 to 00003, and continue executing at the address indicated by the new contents of IP and CS.

A division error interrupt occurs any time a DIV or IDIV instruction is executed with the quotient exceeding the range, regardless of the IF (Interrupt flag) and TF (Trap flag) status.

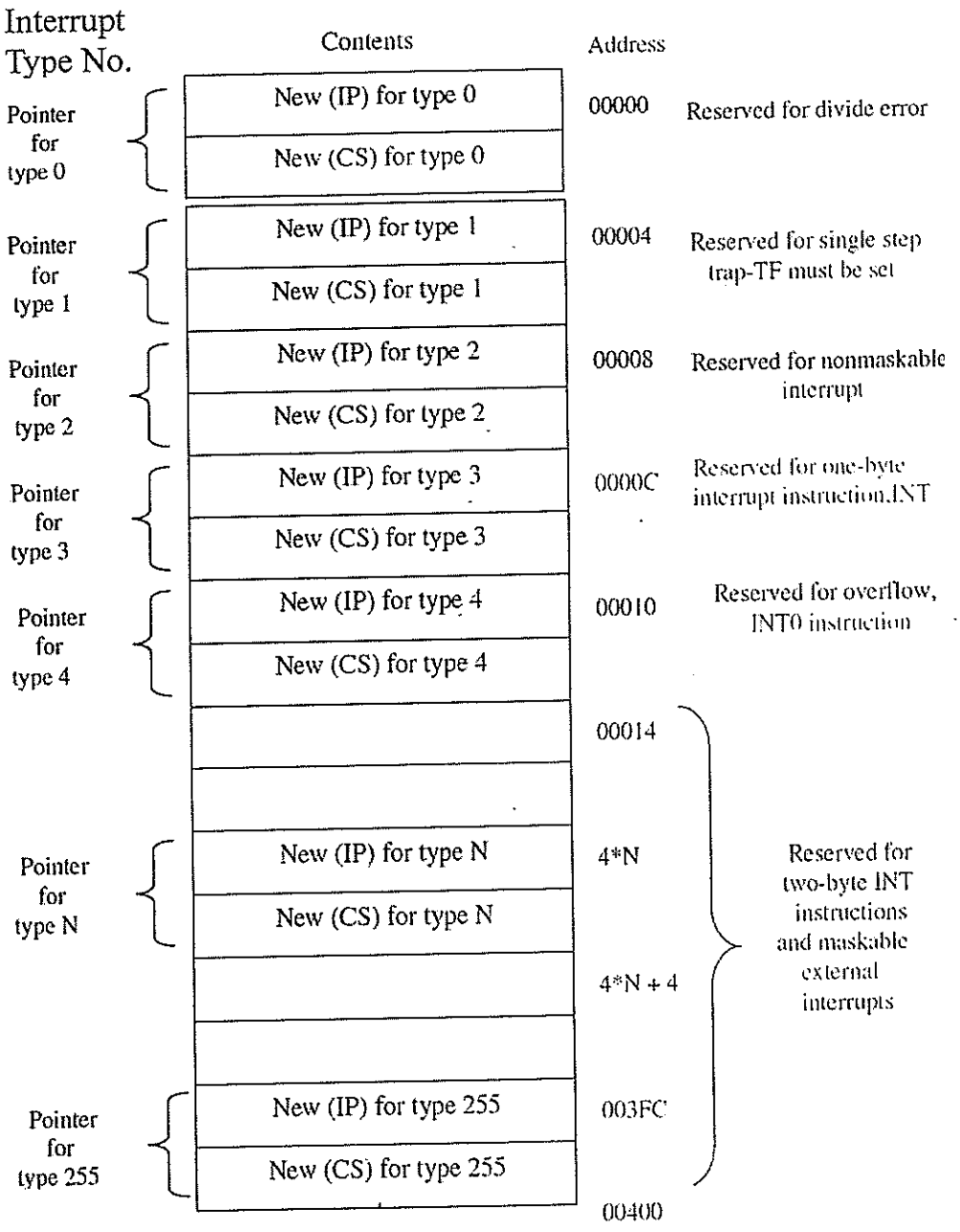
The type 1 interrupt is the single-step interrupt (Trap interrupt) and is the only interrupt controlled by the TF flag. If the TF flag is enabled, then an interrupt will occur at the end of the next instruction that will cause a branch to the location indicated by the contents of 00004H to 00007H.The single step interrupt is used primarily for debugging which gives the programmer a snapshot of his program after each instruction is executed.

The type 2 interrupt is the non-maskable external interrupt. It is the only external interrupt that can occur regardless of the IF flag setting. It is caused by a signal sent to the CPU through the non-maskable interrupt pin.

The remaining interrupt types correspond to interrupts instructions imbedded in the interrupt program or to external interrupts. The interrupt instructions are summarized below and their interrupts are not controlled by the IF flag.

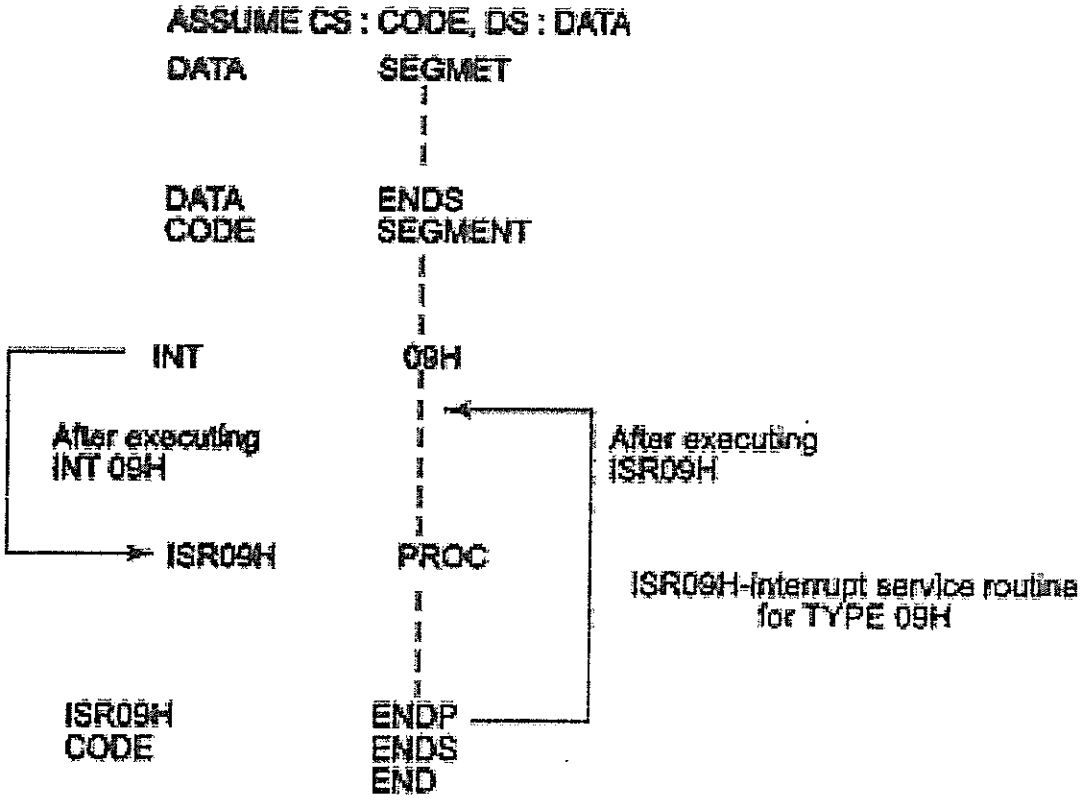


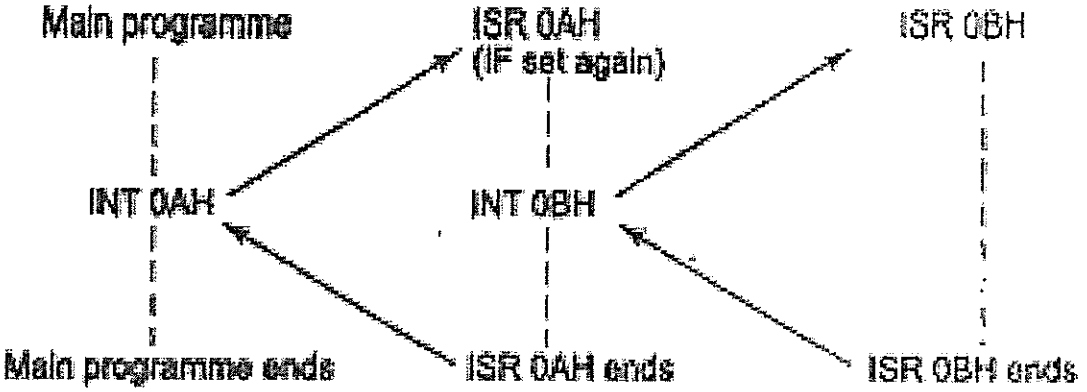
Interrupt Response Sequence



Structure of Interrupt vector table of 8086

Interrupt Programming:





Transfer of Control for Nested Interrupts

Program:
Write an ALP to a create a file RESULT and store in it 500H bytes from the memory block starting at 1000:1000, if either an interrupt occurs at INTR pin with Type 0AH or an instruction equivalent to the above interrupt is executed.

```
ASSUME CS: CODE, DS: DATA
DATA SEGMENT
        FILENAME DB "RESULT", "$"
        MESSAGE DB "FILE WASN'T CREATED SUCCESSFULLY", 0AH, 0DH,"$"
DATA ENDS
CODE SEGMENT
                START:  MOV AX, CODE
                        MOV DS, AX                  ; Set DS at code for setting IVT
                        MOV DX,OFFSET ISR0A         ; Set DX at offset of ISR0A.
                        MOV AX,250AH                ;Set IVT using function value 250AH
                        INT 21H                     ; in AX under INT 21H
                        MOV DX,OFFSET FILENAME ; Set pointer to filename.
                        MOV AX,DATA                 ; Set the DS at DATA for filename.
                        MOV DS,AX
                        MOV CX,00H
                        MOV AH,3CH                  ; Create file with the filename 'RESULT'
                        INT 21H
                        JNC FURTHER                 ; If no carry, create operation is successful
                        MOV DX,OFFSET MESSAGE ; else display message
                        MOV AH,09H
                        INT 21H
                        JMP STOP
                FURTHER: INT 0AH                    ; If the file is created successfully,
                STOP:   MOV AH,4CH                  ; write into it and return to DOS prompt
                        INT 21H
ISR0A PROC NEAR
                MOV BX,AX       ; Take file handle in BX,
                MOV CX,500H     ; Byte count in CX
                MOV DX,1000H    ; Offset of block in DX
                MOV AX,1000H    ; Segment value of block
                MOV DS,AX       ; in DS
                MOV AH,40H      ; Write in the file and return
                INT 21H
ISR0A ENDP
CODE ENDS
                END START
```

Program:
Write an ALP that gives display 'IRT2 is OK' if a hardware Signal appears on IRQ_2 pin and 'IRT3 is OK' if it appears on IRQ_3 pin of PC IO channel.

```
ASSUME CS: CODE, DS: DATA
DATA SEGMENT
        MSG1 DB "IRT2 is OK", 0AH, 0DH,"$"
        MSG2 DB "IRT3 is OK", 0AH, 0DH,"$"
DATA ENDS
CODE SEGMENT
          START: MOV AX,CODE
                 MOV DS,AX                ; Set IVT for Type 0AH
                 MOV DX, OFFSET ISR1
                 MOV AX,250AH             ; IRQ2 is equivalent to Type 0AH
                 INT 21H
                 MOV DX, OFFSET ISR2      ; Set IVT for Type 0BH
                 MOV AX,250BH             ; IRQ3 is equivalent to Type 0BH
                 INT 21H
           HERE: JMP HERE
```

ISR1 display the message

```
ISR1 PROC LOCAL
        MOV AX,DATA
        MOV DS,AX
        MOV DX,OFFSET MSG1    ; Display message MSG1
        MOV AH,09H
        INT 21H
        IRET
ISR1 ENDP
```

ISR2 display the message

```
ISR2 PROC LOCAL
        MOV AX,DATA
        MOV DS,AX
        MOV AX,OFFSET MSG2    ; Display message MSG2
        MOV AH,09H
        INT 21H
        IRET
ISR2 ENDP
CODE ENDS
              END START
```

Assembly Language Program Development Tools:

1. Editor

- An editor is a program which allows you to create a file containing the assembly language statements for your program.

  Example: PC-Write, Wordstar.
- As you type in your program, the editor stores the ASCII codes for the letters and numbers in successive RAM locations.
- When you have typed in your entire program, you then save the file on the hard disk. This file is called source file and the extension is .asm.

2. Assembler

- An assembler program is used to translate the assembly language mnemonics for instructions to corresponding binary codes. When you run the assembler, it reads the source file of your program from the disk where you have saved it after editing.
- On the first pass through the source program, the assembler determines the displacement of named data items, the offset of labels, etc. and puts this information in a symbol table.
- On the second pass through the source program, the assembler produces the binary code for each instruction and inserts the offsets, etc. that it calculated during the first pass.
- The assembler generates 2 files on the floppy disk or hard disk. The first file is called object file (.obj).
- The second file generated by assembler is called the assembler list file and is given extension (.lst).

3. Linker

- A linker is a program used to join several object files into one large object file.
- The linker produces a link file which contains the binary codes for all the combined modules.
- The linker also produces a link map file which contains the address information about the linked files (.exe).

4. Locator

- A locator is a program used to assign the specific address of where the segments of object code are to be loaded into memory.
- A locator program called EXE2BIN comes with the IBM PC Disk Operating System (DOS). EXE2BIN converts a .exe file to a .bin file which has physical addresses.

5. Debugger

- A debugger is a program which allows you to load your object code program into system memory, execute the program and troubleshoot or debug it.
- The debugger allows you to look at the contents of registers and memory locations after your program runs.
- It allows you to change the contents of registers and memory locations and re-run the program.
- Some debuggers allow you to stop execution after each instruction so that you can check or alter after each register contents.
- A debugger also allows you to set a breakpoint at any point in your program. If you insert a breakpoint at any point in your program, the debugger will run the program up to the instruction where you put the breakpoint and then stop the execution.

6. Emulator

- An emulator is a mixture of hardware and software.
- It is used to test and debug the hardware and software of an external system, such as the prototype of a microprocessor based instrument. Part of the hardware of an emulator is a multi wire cable which connects the host system to the system being developed.