## Instruction Set

- Instruction set of 8085 can be classified in following groups:
  - Data Transfer Instructions
    - These instructions can perform data transfer operations between
      - Registers of 8085 e.g. MOV
      - 8085 registers and main memory e.g. LDA, STA, MOV, LDAX, STAX, MVI, LXI etc.
      - Accumulator register and I/O devices e.g. IN, OUT
    - Data transfer instructions never affect the flag bits

## Instruction Set Contd..

- Arithmetic Instructions
  - 8085 can perform only 8-bit addition, subtraction and compare operations. These operations are always performed with accumulator as one of the operands. The status of the result can be verified by the contents of the flag register.
  - Op-codes for arithmetic instructions include ADD, ADI, ADC, ACI, SUB, SUI, SBB, SBI, CMP, CPI
- Logical Instructions
  - 8085 can perform 8-bit basic logical operations -AND, OR, XOR, NOT with some special operations such as rotate and shift operations
  - Logical instructions also modify the flag bits.
  - Op-codes for logical instructions include ANA, ANI, ORA, ORI, XRA, XRI, CMA, RAL, RLC, RAR, RRC etc.

## Instruction Set Contd..

- Program Control Instructions
  - These instructions are used to transfer the program control:
    - to jump from one memory location to any other memory location within a program
    - from one program to another program called as a subroutine
  - 8085 Instruction set consists of following program control instructions:
    - Jump Instructions
    - Call & Return Instructions
    - Restart instructions

## Instruction Set Contd..

- Program control instructions
  - Unconditional or Conditional
    - Unconditional program control instructions perform branching operation unconditionally
    - Conditional program control instructions perform branching operation with reference to the condition of flag bits.

## Instruction Set Contd..

- Unconditional Program control instructions are
  - JMP
  - Call & RET
  - RST n (n=0-7)
- Conditional Program control instructions are
  - JNC, JC, JNZ, JZ, JP, JM, JPE, JPO
  - CNC, CC, CNZ, CZ, CP, CM, CPE, CPO
  - RNC, RC, RNZ, RZ, RP, RM, RPE, RPO

## Instruction Set Contd..

- Machine control Instructions
- These instructions include special instructions such as
  - HLT – To halt the CPU
  - NOP – To perform no operation
  - SIM – To set the masking of hardware interrupts and serial output data
  - RIM – To read the status of interrupt mask and serial input data
  - EI – Enable Interrupt
  - DI – Disable Interrupt

## Addressing Modes

- 8085 instructions can be classified in following addressing modes
  - Register Addressing mode
    - Instructions which have their operands in registers only e.g. MOV, ADD, SUB, ANA, ORA, XRA etc.
  - Immediate Addressing mode
    - Instructions in which operand immediately follows the op-code e.g. MVI, LXI, ADI, SUI, ANI, ORI etc.
  - Direct Addressing mode
    - Instructions have their operands in memory and the 16-bit memory address is specified in the instruction e.g. LDA, STA, LHLD, SHLD etc.

## Addressing Modes Contd..

- Register Indirect Addressing mode
  - Instructions have their operand in memory and the 16-bit memory address is specified in a register pair e.g. LDAX, STAX, PUSH, POP etc.
- Implicit Addressing mode
  - These instruction have their operand implied in the op-code itself e.g. CMA, CMC, STC etc.

## Instruction size

- An instruction is assembled in the memory of a microcomputer system in binary form. The size of an instruction signifies how much memory space is required to load an instruction in the memory. 8085 instructions are of following sizes:
  - One-byte Instructions
    e.g. MOV, ADD, ANA, SUB, ORA etc.
  - Two-byte instructions
    e.g. MVI, ADI, ANI, ORI, XRI etc.
  - Three-byte instructions
    e.g. LXI, LDA, STA, LHLD, SHLD etc.

## Instruction Format

An **instruction** is a command to the microprocessor to perform a given task on a specified data. Each instruction has two parts: one is task to be performed, called the **operation code** (opcode), and the second is the data to be operated on, called the **operand.** The operand (or data) can be specified in various ways. It may include 8-bit (or 16-bit) data, an internal register, a memory location, or 8-bit (or 16-bit) address. In some instructions, the operand is implicit.

### *Instruction word size*

The 8085 instruction set is classified into the following three groups according to word size:

**1.** One-word or 1-byte instructions
**2.** Two-word or 2-byte instructions
**3.** Three-word or 3-byte instructions

In the 8085, "byte" and "word" are synonymous because it is an 8-bit microprocessor.
However, instructions are commonly referred to in terms of bytes rather than words.

## One-Byte Instructions

A 1-byte instruction includes the opcode and operand in the same byte. Operand(s) are internal register and are coded into the instruction.
Example: MOV A,B

## Two-Byte Instructions

In a two-byte instruction, the first byte specifies the operation code and the second byte specifies the operand. Source operand is a data byte immediately following the opcode.
Example: MVI A, 32H

## Three-Byte Instructions

In a three-byte instruction, the first byte specifies the opcode, and the following two bytes specify the 16-bit address. Note that the second byte is the low-order address and the third byte is the high-order address.
Three byte instructions - opcode + data byte + data byte
Example: LXI 21H, 0520H

# 8085 INSTRUCTION SET

## INSTRUCTION DETAILS

### DATA TRANSFER INSTRUCTIONS

| Opcode | Operand | Description |
|---|---|---|
| Copy from source to destination | | |
| MOV | Rd, Rs<br>M, Rs<br>Rd, M | This instruction copies the contents of the source register into the destination register; the contents of the source register are not altered. If one of the operands is a memory location, its location is specified by the contents of the HL registers.<br>Example: MOV B, C or MOV B, M |
| Move immediate 8-bit | | |
| MVI | Rd, data<br>M, data | The 8-bit data is stored in the destination register or memory. If the operand is a memory location, its location is specified by the contents of the HL registers.<br>Example: MVI B, 57H or MVI M, 57H |
| Load accumulator | | |
| LDA | 16-bit address | The contents of a memory location, specified by a 16-bit address in the operand, are copied to the accumulator. The contents of the source are not altered.<br>Example: LDA 2034H |
| Load accumulator indirect | | |
| LDAX | B/D Reg. pair | The contents of the designated register pair point to a memory location. This instruction copies the contents of that memory location into the accumulator. The contents of either the register pair or the memory location are not altered.<br>Example: LDAX B |
| Load register pair immediate | | |
| LXI | Reg. pair, 16-bit data | The instruction loads 16-bit data in the register pair designated in the operand.<br>Example: LXI H, 2034H or LXI H, XYZ |
| Load H and L registers direct | | |
| LHLD | 16-bit address | The instruction copies the contents of the memory location pointed out by the 16-bit address into register L and copies the contents of the next memory location into register H. The contents of source memory locations are not altered.<br>Example: LHLD 2040H |

Store accumulator direct

STA 16-bit address

The contents of the accumulator are copied into the memory location specified by the operand. This is a 3-byte instruction, the second byte specifies the low-order address and the third byte specifies the high-order address.
Example: STA 4350H

Store accumulator indirect

STAX Reg. pair

The contents of the accumulator are copied into the memory location specified by the contents of the operand (register pair). The contents of the accumulator are not altered.
Example: STAX B

Store H and L registers direct

SHLD 16-bit address

The contents of register L are stored into the memory location specified by the 16-bit address in the operand and the contents of H register are stored into the next memory location by incrementing the operand. The contents of registers HL are not altered. This is a 3-byte instruction, the second byte specifies the low-order address and the third byte specifies the high-order address.
Example: SHLD 2470H

Exchange H and L with D and E

XCHG none

The contents of register H are exchanged with the contents of register D, and the contents of register L are exchanged with the contents of register E.
Example: XCHG

Copy H and L registers to the stack pointer

SPHL none

The instruction loads the contents of the H and L registers into the stack pointer register, the contents of the H register provide the high-order address and the contents of the L register provide the low-order address. The contents of the H and L registers are not altered.
Example: SPHL

Exchange H and L with top of stack

XTHL none

The contents of the L register are exchanged with the stack location pointed out by the contents of the stack pointer register. The contents of the H register are exchanged with the next stack location (SP+1); however, the contents of the stack pointer register are not altered.
Example: XTHL

Push register pair onto stack

| | | |
|---|---|---|
| PUSH | Reg. pair | The contents of the register pair designated in the operand are copied onto the stack in the following sequence. The stack pointer register is decremented and the contents of the high-order register (B, D, H, A) are copied into that location. The stack pointer register is decremented again and the contents of the low-order register (C, E, L, flags) are copied to that location.<br>Example: PUSH B or PUSH A |

Pop off stack to register pair

| | | |
|---|---|---|
| POP | Reg. pair | The contents of the memory location pointed out by the stack pointer register are copied to the low-order register (C, E, L, status flags) of the operand. The stack pointer is incremented by 1 and the contents of that memory location are copied to the high-order register (B, D, H, A) of the operand. The stack pointer register is again incremented by 1.<br>Example: POP H or POP A |

Output data from accumulator to a port with 8-bit address

| | | |
|---|---|---|
| OUT | 8-bit port address | The contents of the accumulator are copied into the I/O port specified by the operand.<br>Example: OUT F8H |

Input data to accumulator from a port with 8-bit address

| | | |
|---|---|---|
| IN | 8-bit port address | The contents of the input port designated in the operand are read and loaded into the accumulator.<br>Example: IN 8CH |

**ARITHMETIC INSTRUCTIONS**

| Opcode | Operand | Description |
|---|---|---|
| Add register or memory to accumulator | | |
| ADD | R<br>M | The contents of the operand (register or memory) are added to the contents of the accumulator and the result is stored in the accumulator. If the operand is a memory location, its location is specified by the contents of the HL registers. All flags are modified to reflect the result of the addition.<br>Example: ADD B or ADD M |
| Add register to accumulator with carry | | |
| ADC | R<br>M | The contents of the operand (register or memory) and the Carry flag are added to the contents of the accumulator and the result is stored in the accumulator. If the operand is a memory location, its location is specified by the contents of the HL registers. All flags are modified to reflect the result of the addition.<br>Example: ADC B or ADC M |
| Add immediate to accumulator | | |
| ADI | 8-bit data | The 8-bit data (operand) is added to the contents of the accumulator and the result is stored in the accumulator. All flags are modified to reflect the result of the addition.<br>Example: ADI 45H |
| Add immediate to accumulator with carry | | |
| ACI | 8-bit data | The 8-bit data (operand) and the Carry flag are added to the contents of the accumulator and the result is stored in the accumulator. All flags are modified to reflect the result of the addition.<br>Example: ACI 45H |
| Add register pair to H and L registers | | |
| DAD | Reg. pair | The 16-bit contents of the specified register pair are added to the contents of the HL register and the sum is stored in the HL register. The contents of the source register pair are not altered. If the result is larger than 16 bits, the CY flag is set. No other flags are affected.<br>Example: DAD H |

Subtract register or memory from accumulator

| | | |
|---|---|---|
| SUB | R<br>M | The contents of the operand (register or memory ) are subtracted from the contents of the accumulator, and the result is stored in the accumulator. If the operand is a memory location, its location is specified by the contents of the HL registers. All flags are modified to reflect the result of the subtraction.<br>Example: SUB B or SUB M |

Subtract source and borrow from accumulator

| | | |
|---|---|---|
| SBB | R<br>M | The contents of the operand (register or memory ) and the Borrow flag are subtracted from the contents of the accumulator and the result is placed in the accumulator. If the operand is a memory location, its location is specified by the contents of the HL registers. All flags are modified to reflect the result of the subtraction.<br>Example: SBB B or SBB M |

Subtract immediate from accumulator

| | | |
|---|---|---|
| SUI | 8-bit data | The 8-bit data (operand) is subtracted from the contents of the accumulator and the result is stored in the accumulator. All flags are modified to reflect the result of the subtraction.<br>Example: SUI 45H |

Subtract immediate from accumulator with borrow

| | | |
|---|---|---|
| SBI | 8-bit data | The 8-bit data (operand) and the Borrow flag are subtracted from the contents of the accumulator and the result is stored in the accumulator. All flags are modified to reflect the result of the subtracion.<br>Example: SBI 45H |

Increment register or memory by 1

| | | |
|---|---|---|
| INR | R<br>M | The contents of the designated register or memory) are incremented by 1 and the result is stored in the same place. If the operand is a memory location, its location is specified by the contents of the HL registers.<br>Example: INR B or INR M |

Increment register pair by 1

| | | |
|---|---|---|
| INX | R | The contents of the designated register pair are incremented by 1 and the result is stored in the same place.<br>Example: INX H |

Decrement register or memory by 1

DCR R
M

The contents of the designated register or memory are decremented by 1 and the result is stored in the same place. If the operand is a memory location, its location is specified by the contents of the HL registers.
Example: DCR B or DCR M

Decrement register pair by 1

DCX R

The contents of the designated register pair are decremented by 1 and the result is stored in the same place.
Example: DCX H

Decimal adjust accumulator

DAA none

The contents of the accumulator are changed from a binary value to two 4-bit binary coded decimal (BCD) digits. This is the only instruction that uses the auxiliary flag to perform the binary to BCD conversion, and the conversion procedure is described below. S, Z, AC, P, CY flags are altered to reflect the results of the operation.

If the value of the low-order 4-bits in the accumulator is greater than 9 or if AC flag is set, the instruction adds 6 to the low-order four bits.

If the value of the high-order 4-bits in the accumulator is greater than 9 or if the Carry flag is set, the instruction adds 6 to the high-order four bits.

Example: DAA

**BRANCHING INSTRUCTIONS**

| Opcode | Operand | Description |
|---|---|---|
| Jump unconditionally | | |
| JMP | 16-bit address | The program sequence is transferred to the memory location specified by the 16-bit address given in the operand.<br>Example: JMP 2034H or JMP XYZ |

Jump conditionally

Operand: 16-bit address

The program sequence is transferred to the memory location specified by the 16-bit address given in the operand based on the specified flag of the PSW as described below.
Example: JZ 2034H or JZ XYZ

| Opcode | Description | Flag Status |
|---|---|---|
| JC | Jump on Carry | CY = 1 |
| JNC | Jump on no Carry | CY = 0 |
| JP | Jump on positive | S = 0 |
| JM | Jump on minus | S = 1 |
| JZ | Jump on zero | Z = 1 |
| JNZ | Jump on no zero | Z = 0 |
| JPE | Jump on parity even | P = 1 |
| JPO | Jump on parity odd | P = 0 |

Unconditional subroutine call

CALL 16-bit address

The program sequence is transferred to the memory location specified by the 16-bit address given in the operand. Before the transfer, the address of the next instruction after CALL (the contents of the program counter) is pushed onto the stack.
Example: CALL 2034H or CALL XYZ

Call conditionally

Operand: 16-bit address

The program sequence is transferred to the memory location specified by the 16-bit address given in the operand based on the specified flag of the PSW as described below. Before the transfer, the address of the next instruction after the call (the contents of the program counter) is pushed onto the stack.
Example: CZ 2034H or CZ XYZ

| Opcode | Description | Flag Status |
|---|---|---|
| CC | Call on Carry | CY = 1 |
| CNC | Call on no Carry | CY = 0 |
| CP | Call on positive | S = 0 |
| CM | Call on minus | S = 1 |
| CZ | Call on zero | Z = 1 |
| CNZ | Call on no zero | Z = 0 |
| CPE | Call on parity even | P = 1 |
| CPO | Call on parity odd | P = 0 |

Return from subroutine unconditionally

RET none

The program sequence is transferred from the subroutine to the calling program. The two bytes from the top of the stack are copied into the program counter, and program execution begins at the new address.
Example: RET

Return from subroutine conditionally

Operand: none

The program sequence is transferred from the subroutine to the calling program based on the specified flag of the PSW as described below. The two bytes from the top of the stack are copied into the program counter, and program execution begins at the new address.
Example: RZ

| Opcode | Description | Flag Status |
|---|---|---|
| RC | Return on Carry | CY = 1 |
| RNC | Return on no Carry | CY = 0 |
| RP | Return on positive | S = 0 |
| RM | Return on minus | S = 1 |
| RZ | Return on zero | Z = 1 |
| RNZ | Return on no zero | Z = 0 |
| RPE | Return on parity even | P = 1 |
| RPO | Return on parity odd | P = 0 |

Load program counter with HL contents

PCHL none

The contents of registers H and L are copied into the program counter. The contents of H are placed as the high-order byte and the contents of L as the low-order byte.
Example: PCHL

Restart

RST 0-7

The RST instruction is equivalent to a 1-byte call instruction to one of eight memory locations depending upon the number. The instructions are generally used in conjunction with interrupts and inserted using external hardware. However these can be used as software instructions in a program to transfer program execution to one of the eight locations. The addresses are:

| Instruction | Restart Address |
|---|---|
| RST 0 | 0000H |
| RST 1 | 0008H |
| RST 2 | 0010H |
| RST 3 | 0018H |
| RST 4 | 0020H |
| RST 5 | 0028H |
| RST 6 | 0030H |
| RST 7 | 0038H |

The 8085 has four additional interrupts and these interrupts generate RST instructions internally and thus do not require any external hardware. These instructions and their Restart addresses are:

| Interrupt | Restart Address |
|---|---|
| TRAP | 0024H |
| RST 5.5 | 002CH |
| RST 6.5 | 0034H |
| RST 7.5 | 003CH |

**LOGICAL INSTRUCTIONS**

| Opcode | Operand | Description |
|---|---|---|
| Compare register or memory with accumulator | | |
| CMP | R<br>M | The contents of the operand (register or memory) are compared with the contents of the accumulator. Both contents are preserved . The result of the comparison is shown by setting the flags of the PSW as follows:<br>if (A) < (reg/mem): carry flag is set<br>if (A) = (reg/mem): zero flag is set<br>if (A) > (reg/mem): carry and zero flags are reset<br>Example: CMP B or CMP M |
| Compare immediate with accumulator | | |
| CPI | 8-bit data | The second byte (8-bit data) is compared with the contents of the accumulator. The values being compared remain unchanged. The result of the comparison is shown by setting the flags of the PSW as follows:<br>if (A) < data: carry flag is set<br>if (A) = data: zero flag is set<br>if (A) > data: carry and zero flags are reset<br>Example: CPI 89H |
| Logical AND register or memory with accumulator | | |
| ANA | R<br>M | The contents of the accumulator are logically ANDed with the contents of the operand (register or memory), and the result is placed in the accumulator. If the operand is a memory location, its address is specified by the contents of HL registers. S, Z, P are modified to reflect the result of the operation. CY is reset. AC is set.<br>Example: ANA B or ANA M |
| Logical AND immediate with accumulator | | |
| ANI | 8-bit data | The contents of the accumulator are logically ANDed with the 8-bit data (operand) and the result is placed in the accumulator. S, Z, P are modified to reflect the result of the operation. CY is reset. AC is set.<br>Example: ANI 86H |

Exclusive OR register or memory with accumulator

| | | |
|---|---|---|
| XRA | R<br>M | The contents of the accumulator are Exclusive ORed with the contents of the operand (register or memory), and the result is placed in the accumulator. If the operand is a memory location, its address is specified by the contents of HL registers. S, Z, P are modified to reflect the result of the operation. CY and AC are reset.<br>Example: XRA B or XRA M |

Exclusive OR immediate with accumulator

| | | |
|---|---|---|
| XRI | 8-bit data | The contents of the accumulator are Exclusive ORed with the 8-bit data (operand) and the result is placed in the accumulator. S, Z, P are modified to reflect the result of the operation. CY and AC are reset.<br>Example: XRI 86H |

Logical OR register or memory with accumulaotr

| | | |
|---|---|---|
| ORA | R<br>M | The contents of the accumulator are logically ORed with the contents of the operand (register or memory), and the result is placed in the accumulator. If the operand is a memory location, its address is specified by the contents of HL registers. S, Z, P are modified to reflect the result of the operation. CY and AC are reset.<br>Example: ORA B or ORA M |

Logical OR immediate with accumulator

| | | |
|---|---|---|
| ORI | 8-bit data | The contents of the accumulator are logically ORed with the 8-bit data (operand) and the result is placed in the accumulator. S, Z, P are modified to reflect the result of the operation. CY and AC are reset.<br>Example: ORI 86H |

Rotate accumulator left

| | | |
|---|---|---|
| RLC | none | Each binary bit of the accumulator is rotated left by one position. Bit $D_7$ is placed in the position of $D_0$ as well as in the Carry flag. CY is modified according to bit $D_7$. S, Z, P, AC are not affected.<br>Example: RLC |

Rotate accumulator right

| | | |
|---|---|---|
| RRC | none | Each binary bit of the accumulator is rotated right by one position. Bit $D_0$ is placed in the position of $D_7$ as well as in the Carry flag. CY is modified according to bit $D_0$. S, Z, P, AC are not affected.<br>Example: RRC |

Rotate accumulator left through carry

| | | |
|---|---|---|
| RAL | none | Each binary bit of the accumulator is rotated left by one position through the Carry flag. Bit $D_7$ is placed in the Carry flag, and the Carry flag is placed in the least significant position $D_0$. CY is modified according to bit $D_7$. S, Z, P, AC are not affected.<br>Example: RAL |

Rotate accumulator right through carry

| | | |
|---|---|---|
| RAR | none | Each binary bit of the accumulator is rotated right by one position through the Carry flag. Bit $D_0$ is placed in the Carry flag, and the Carry flag is placed in the most significant position $D_7$. CY is modified according to bit $D_0$. S, Z, P, AC are not affected.<br>Example: RAR |

Complement accumulator

| | | |
|---|---|---|
| CMA | none | The contents of the accumulator are complemented. No flags are affected.<br>Example: CMA |

Complement carry

| | | |
|---|---|---|
| CMC | none | The Carry flag is complemented. No other flags are affected.<br>Example: CMC |

Set Carry

| | | |
|---|---|---|
| STC | none | The Carry flag is set to 1. No other flags are affected.<br>Example: STC |

**CONTROL INSTRUCTIONS**

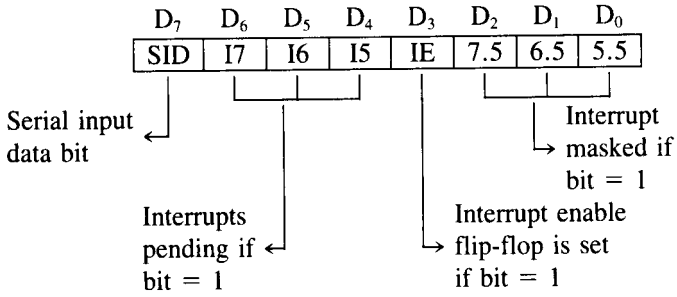
| Opcode | Operand | Description |
|---|---|---|
| No operation | | |
| NOP | none | No operation is performed. The instruction is fetched and decoded. However no operation is executed.<br>Example: NOP |
| Halt and enter wait state | | |
| HLT | none | The CPU finishes executing the current instruction and halts any further execution. An interrupt or reset is necessary to exit from the halt state.<br>Example: HLT |
| Disable interrupts | | |
| DI | none | The interrupt enable flip-flop is reset and all the interrupts except the TRAP are disabled. No flags are affected.<br>Example: DI |
| Enable interrupts | | |
| EI | none | The interrupt enable flip-flop is set and all interrupts are enabled. No flags are affected. After a system reset or the acknowledgement of an interrupt, the interrupt enable flip-flop is reset, thus disabling the interrupts. This instruction is necessary to reenable the interrupts (except TRAP).<br>Example: EI |

Read interrupt mask

RIM none

This is a multipurpose instruction used to read the status of interrupts 7.5, 6.5, 5.5 and read serial data input bit. The instruction loads eight bits in the accumulator with the following interpretations.

Example: RIM

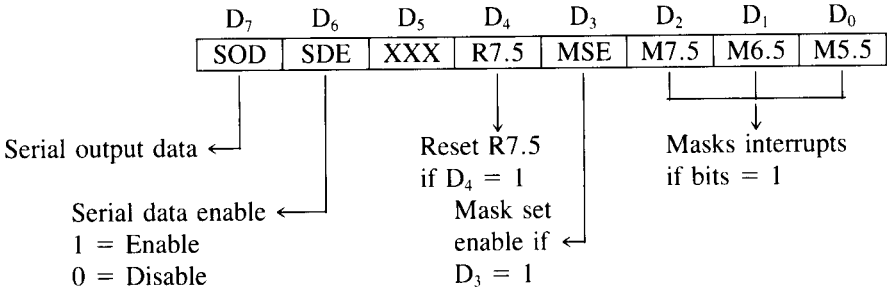



Set interrupt mask

SIM none

This is a multipurpose instruction and used to implement the 8085 interrupts 7.5, 6.5, 5.5, and serial data output. The instruction interprets the accumulator contents as follows.

Example: SIM




- ☐ SOD — Serial Output Data: Bit $D_7$ of the accumulator is latched into the SOD output line and made available to a serial peripheral if bit $D_6 = 1$.
- ☐ SDE — Serial Data Enable: If this bit = 1, it enables the serial output. To implement serial output, this bit needs to be enabled.
- ☐ XXX — Don't Care
- ☐ R7.5 — Reset RST 7.5: If this bit = 1, RST 7.5 flip-flop is reset. This is an additional control to reset RST 7.5.
- ☐ MSE — Mask Set Enable: If this bit is high, it enables the functions of bits $D_2$, $D_1$, $D_0$. This is a master control over all the interrupt masking bits. If this bit is low, bits $D_2$, $D_1$, and $D_0$ do not have any effect on the masks.
- ☐ M7.5 — $D_2$ = 0, RST 7.5 is enabled.
  = 1, RST 7.5 is masked or disabled.
- ☐ M6.5 — $D_1$ = 0, RST 6.5 is enabled.
  = 1, RST 6.5 is masked or disabled.
- ☐ M5.5 — $D_0$ = 0, RST 5.5 is enabled.
  = 1, RST 5.5 is masked or disabled.

## 3.7 Assembly Language Programming

A program is a set of instructions arranged in the specific sequence to do the specific task. It tells the microprocessor what it has to do. The process of writing the set of instructions which tells the microprocessor what to do is called **"Programming"**. In other words, we can say that programming is the process of telling the processor exactly how to solve a problem. To do this, the programmer must **"speak"** to the processor in a language which processor can understand.

### 3.7.1 Steps Involved in Programming

- **Specifying the problem :**
  The first step in the programming is to find out which task is to be performed. This is called specifying the problem. If the programmer does not understand what is to be done, the programming process cannot begin.
- **Designing the problem-solution :**
  During this process, the exact step by step process that is to be followed (program logic) is developed and written down.
- **Coding :**
  Once the program is specified and designed, it can be implemented. Implementation begins with the process of coding the program. Coding the program means to tell the processor the exact step by step process in its language. Each processor has a set of instructions. Programmer has to choose appropriate instructions from the instruction set to build the program.
- **Debugging :**
  Once the program or a part of program is coded, the next step is debugging the code. Debugging is the process of testing the code to see if it does the given task. If program is not working properly, debugging process helps in finding and correcting errors.

**To write a program, programmer should know :**

- How to develop program logic?
- How to tell the program to the processor?
- How to code the program?
- How to test the program?

### 3.7.2 Flowchart

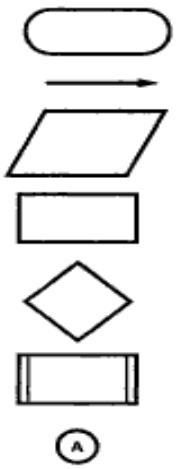



**Fig. 3.7 Graphic symbols used in flowchart**

To develop the programming logic programmer has to write down various actions which are to be performed in proper sequence. The flow chart is a graphical tool that allows programmer to represent various actions which are to be performed. The graphical representation is very useful for clear understanding of the programming logic.

The Fig. 3.7 shows the graphic symbols used in the flow chart.

**Oval :** It indicates start or stop operation.

**Arrow :** It indicates flow with direction.

**Parallelogram :** It indicates input/output operation.

**Rectangle :** It indicates process operation.

**Diamond :** It indicates decision making operation.

**Double sided rectangle :** It indicates execution of pre-defined process (subroutine).

**Circle with alphabet :** It indicates continuation.

**A: Any alphabet**

The Fig. 3.8 shows sample flowchart.

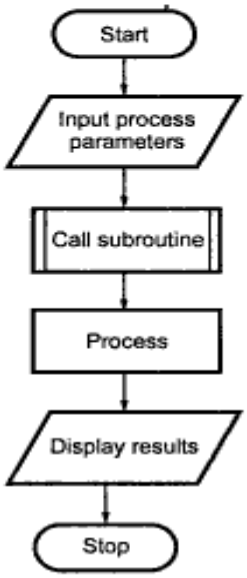



**Fig. 3.8 Sample flowchart**

### 3.7.3 Assembly Language Program

Let us define a program statement as 'write an assembly language program to add two numbers'. The three tasks are involved in this program :

- Load two hex numbers
- Add numbers and
- Store the result in the memory

These tasks can be symbolically presented as flowchart, as shown in the Fig. 3.9.

Next job is to find the suitable 8085 assembly language instruction/s for each task. These instructions are as follows :

**Task 1 instructions :**

```
MVI A, 20H    ; Load 20H as a first
              ; number in register A
MVI B, 40H    ; Load 40H as a second number
              ; in register B
```

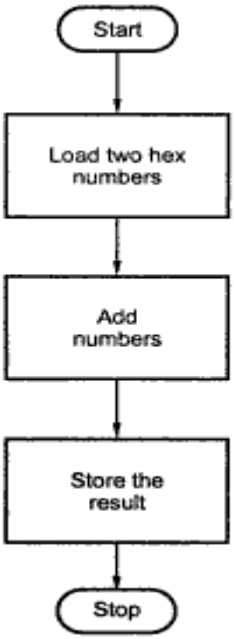



**Fig. 3.9 Flowchart for addition of two numbers**

**Task 2 Instruction :**

```
ADD B          ; Add two numbers and save
               ; result in register A
```

**Task 3 Instruction :**

```
STA 2200H      ; Store the result in memory
               ; location 2000H
HLT            ; Stop the program execution
```

We want to execute three tasks in a sequence, thus writing corresponding instructions in the same sequence constitutes an assembly language program.

### 3.7.4 Assembly Language Program to Machine Language Program

Once the assembly language program is ready, it is necessary to convert it in the machine language program. It is possible to do this by referring the proper hex code for each assembly instruction from the 8085 instruction set manual. This process is known as **hand assembly** and the resulted machine language program is also known as **hex code**. Let us see the hex code for our program.

| Mnemonics | Hex code | |
|---|---|---|
| MVI A, 20H | 3EH ← | Opcode |
| | 20H ← | Operand |

| | | |
|---|---|---|
| MVI B, 40H | 06H ← | Opcode |
| | 40H ← | Operand |
| ADD B | 80H ← | Opcode |
| STA 2200H | 32H ← | Opcode |
| | 00H ← | Operand (lower byte of address) |
| | 22H ← | Operand (higher byte of address) |
| HLT | 76H ← | Opcode |

### 3.7.5 Storing Hex Code in the Memory

Once the hex code is ready, it has to be loaded in the memory of specially designed microprocessor system (Microprocessor training kit) for execution. To perform this task we should know the address range of read/write memory in the system. Let us assume that the read/write memory ranges from 2000H to 22FFH. The microprocessor training kit has keypad to enter the hex code in the memory. It provides a special routines (monitor program) to enter a hex code byte by byte and execute the program. Typical steps for storing hex code in the memory from address from address 2000H are as follows :

1. Reset the microprocessor system by pressing the RESET key.
2. Enter into store mode by pressing SET key.
3. Enter the address of the memory 2000H, where the first hex code (starting address of the program) is to be stored using hex keys.
4. Enter the hex code using hex keys.
5. Increment the memory address by 1 using INC key.
6. Repeat steps 4 and 5 until the last hex code.

### 3.7.6 Executing the Program

The microprocessor training kit provides a procedure to execute the program. To activate the procedure we have to enter the starting address of the program (2000H in our example). To enter this address we have to go into execute mode by pressing GO key and enter the starting address using hex keys. Once the starting address is entered, the program can be executed by pressing EXECUTE key. The EXECUTE key procedure loads the starting address of our program, 2000H into the program counter and program control is transferred from monitor program to our program.

After this microprocessor reads one hex code at a time, and when it fetches the complete instruction, it executes that instruction. Then it fetches the next instruction and this process continues until the last instruction in the program is executed.

## 4.1 Looping, Counting and Indexing

Before going to implement these techniques, we will get conversant with these techniques and understand their use.

**Looping :** In this technique, the program is instructed to execute certain set of instructions repeatedly to execute a particular task number of times. For example, to add ten numbers stored in the consecutive memory locations we have to perform addition ten times.

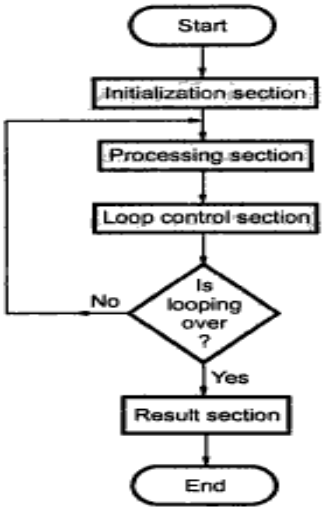
**Counting :** This technique allows programmer to count how many times the instruction/set of instructions are executed.

**Indexing :** This technique allows programmer to point or refer the data stored in sequential memory locations one by one. Let us see the program loop to understand looping, counting and indexing.

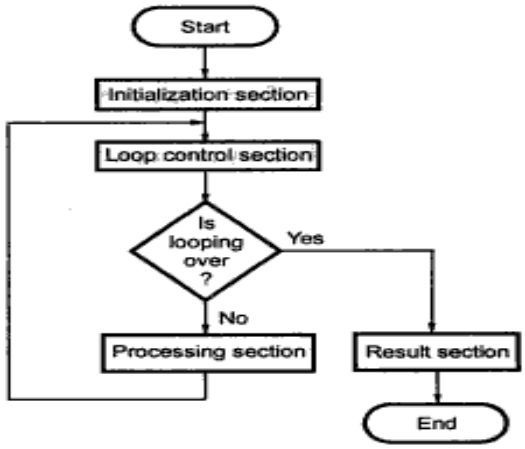
The program loop is the basic structure which forces the processor to repeat a sequence of instructions. Loops have four sections.

1. Initialization section.
2. Processing section.
3. Loop control section
4. Result section.

**Flowchart**




**Flowchart 1**




**Flowchart 2**

1. The initialization section establishes the starting values of
   - loop counters for counting how many times loop is executed,
   - address registers for indexing which give pointers to memory locations and
   - other variables
2. The actual data manipulation occurs in the processing section. This is the section which does the work.
3. The loop control section updates counters, indices (pointers) for the next iteration.
4. The result section analyzes and stores the results.

**Note :** The processor executes initialization section and result section only once, while it may execute processing section and loop control section many times. Thus, the execution time of the loop will be mainly dependent on the execution time of the processing section and loop control section. The flowchart 1 shows typical program loop. The processing section in this flowchart is always executed at least once. If you interchange the position of the processing and loop control section then it is possible that the processing section may not be executed at all, if necessary. Refer flowchart 2.

# 5 Stacks and Subroutines

The stack is a part of read/write memory that is used for temporary storage of binary information during the execution of a program. The binary information is basically the intermediate results and the return address in case of subroutine calls.

1. For the application programs, the internal memory of the microprocessor (registers) is not sufficient to store the intermediate results. These intermediate results can be stored temporarily on the stack and can be referred back when required.
2. A subroutine is a group of instructions, performs a particular subtask which is executed number of times. It is written separately. The microprocessor executes this subroutine by transferring program control to the subroutine program. After completion of subroutine program execution, the program control is returned back to the main program. The use of subroutines is a very important technique in designing software for microprocessor systems because it eliminates the need to write a subtasks repeatedly; thus it uses memory more efficiently. For implementation of subroutine technique, it is necessary to define stack. In the stack, the address of the instruction in the main program which follows the subroutine call is stored.

## 5.1 Concept of Stack and Subroutines

This section explains the concept of stack and subroutines in detail.

### 5.1.1 Stack

The stack is a portion of read/write memory set aside by the user for the purpose of storing information temporarily. When the information is written on the stack, the operation is called PUSH. When the information is read from stack, the operation is called POP.

The microprocessor stores the information, much like stacking plates. Using this analogy of stacking plates it is easy to illustrate the stack operation.

Fig. 5.1 shows the stacked plates. Here, we realize that if it is desired to take out the first stacked plate we will have to remove all plates above the first plate in the reverse order. This means that to remove first plate we will have to remove the third plate, then the second plate and finally the first plate. This means that, the first information pushed on to the stack is the last information popped off from the stack. This type of operation is known as a first in, last out (FILO). This stack is implemented with the help of special memory pointer register. The special pointer register is called the **stack pointer.** During PUSH and POP operation, stack pointer register gives the address of memory where the information is to be stored or to be read. The stack pointer's contents are automatically manipulated to point to stack top. The memory location currently pointed by stack pointer is called **top of stack.**


**Fig. 5.1 Stacked plates**

The stack pointer SP, is a 16-bit register in the 8085A which is manipulated by the microprocessor's control section, during stack related instructions.

#### 5.1.1.1 Stack Related Instructions

The 8085A supports following stack related instructions :

**LXI SP, data (16) :**

It initializes stack pointer with 16-bit address.

**SPHL :**

It copies the contents of HL register pair into the stack pointer.

**PUSH rp :**

It is used to write 16-bit data in the stack.

**POP rp :**

It is used to read 16-bit data from the stack.

**CALL addr :**

It transfers the program control to the subroutine program after storing the return address in the stack.

**RET :**

It reads the return address from the stack and transfers the program control back to the instruction following the CALL.

**INX SP :**

It increments the contents of stack pointer by one.

**DAD SP :**

It adds the contents of stack pointer into the contents of HL register pair and stores the result in the HL register pair.

**XTHL :**

It exchanges the contents of memory location pointed by the stack pointer with the contents of L register and the contents of the next memory location with the contents of H register.

Now we will see the detail operation and the use of these instructions.

### 5.1.1.2 Detail Operation and the use of Stack Related Instructions

**LXI SP, Data and SPHL : Initializes Stack Pointer.**

Before execution of any stack related instruction, stack pointer must be initialized with a valid memory address. The stack pointer can be initialized by two ways.

1. **Direct way :** LXI SP, data (16-bit) ; Loads 16-bit data into SP
2. **Indirect way :** LXI H, data (16-bit) ; Loads 16-bit data into HL
   SPHL ; Loads the contents of HL into SP

Normally, the stack pointer is initialized by the direct way. When a programmer wishes to set the stack pointer to a value that has been computed by the program, indirect way is used. The computed value is placed in H and L and the contents of HL register pair then moved into the stack pointer.

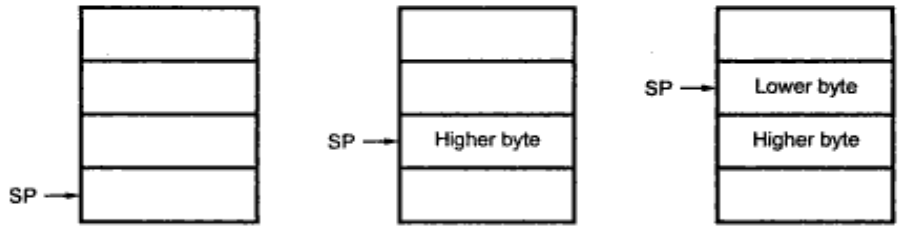
**Note :**

1. The stack pointer can be initialized anywhere in the read/write memory map. However, as a general practice, the stack pointer is initialized at the highest Read/Write memory location so that it will be less likely to interfere with a program.
2. Since the 8085A's stack pointer is decremented before data is written to the stack, the stack pointer can actually be initialized to a value one higher than the highest read/write memory location available.

**PUSH and POP :** Temporarily stores the contents of register pair and program internal status word, and retrieves when required.

When programmer realizes the shortage of the registers, he stores the present contents of the registers in the stack with the help of PUSH instruction and then uses the registers for other function. After completion of other function programmer loads the previous contents of the register from the stack with the help of POP instruction.

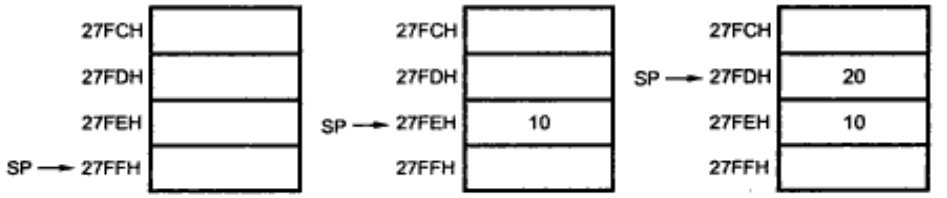
**PUSH Operation :** In PUSH operation, 16-bit data is stored on the stack. This 16-bit data is stored in two operations. In the first operation, stack pointer is decremented by one and then the higher byte of the 16-bit data is stored at the memory location pointed by stack pointer. In the second operation, stack pointer is again decremented by one and then the lower byte of the 16-bit data is stored at the memory location pointed by stack pointer. The Fig. 5.2 shows steps involved in the PUSH operation.




**(a) Initial Position** **(b) Decrements SP & stores higher byte** **(c) Decrements SP & stores lower byte**

**Fig. 5.2 Steps involved in PUSH Operation**

**Example 1 :** The contents of BC register pair are 1020H and contents of stack pointer are 27FFH. Then after execution of PUSH B instruction the stack contents are as shown in the Fig. 5.3.

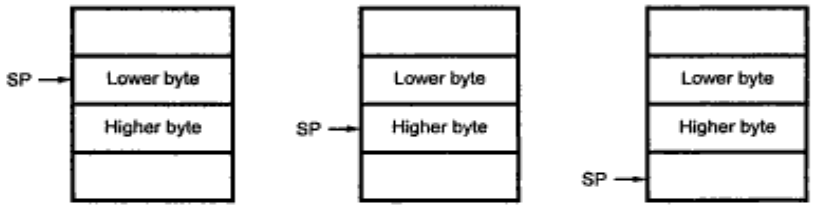



**(a) Initial position** **(b) Decrements SP & stores higher bytes** **(c) Decrements SP & stores lower bytes**

**Fig. 5.3 Steps involved in PUSH Operation with example**

**POP Operation :** In POP operation, 16-bit data is read from the stack. This 16-bit data is read in two operations. In the first operation, the contents from the memory location pointed by stack pointer are loaded into lower byte of register pair and then the stack pointer is incremented by one. In the second operation, the contents from the

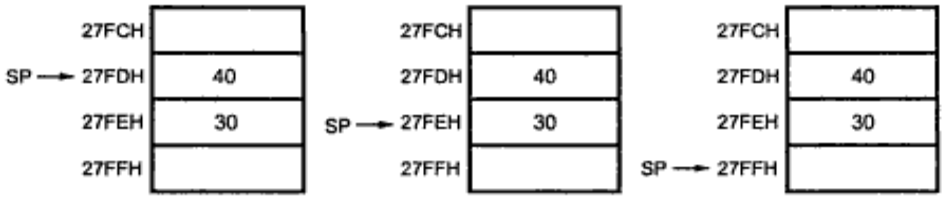
memory location pointed by stack pointer are loaded into higher byte of register pair and then the stack pointer is incremented by one. Fig. 5.4 shows steps involved in the POP operation.




**(a) Initial position, reads lower bytes** **(b) Increments SP and reads higher bytes** **(c) Increments SP**

**Fig. 5.4 Steps Involved in POP Operation**

**Example 2 :** If the initial contents of stack and contents of stack pointer are as shown in Fig. 5.5 (a) then after execution POP B contents of BC register will be 3040 (B = 30H and C = 40H).




**(a) Initial position, reads lower bytes** **(b) Increments SP and reads higher bytes** **(c) Increments SP**

**Fig. 5.5 Steps Involved in POP operation with example**

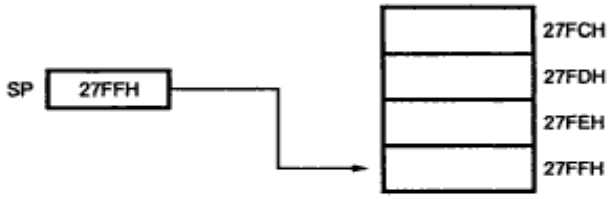
**Example 3 :** Let us consider the following program

```
LXI SP, 27FFH
LXI B, 2030H
LXI D, 4045H
PUSH D
PUSH B
MOV A, C
ADD E
MOV D, A
POP B
POP D
```

We will see the contents of stack and stack pointer after execution of each instruction in the above program.

**Instruction 1 :**

LXI SP, 27FFH




**Instruction 2 :**

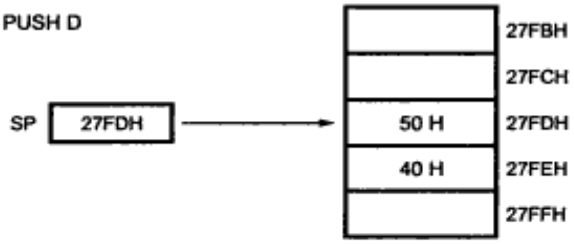
LXI B, 2030H


**Instruction 3 :**

LXI D, 4050H

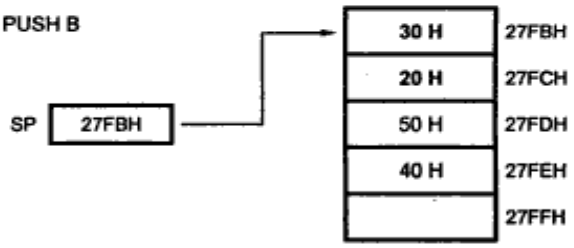

**Instruction 4 :**

PUSH D



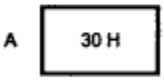

**Instruction 5 :**

PUSH B



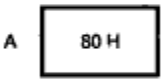
**Instruction 6 :**

MOV A, C



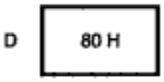

**Instruction 7 :**

ADD E



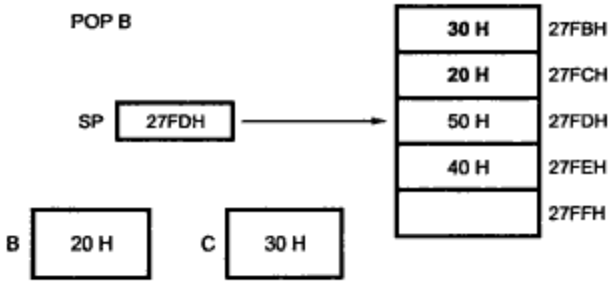

**Instruction 8 :**

MOV D, A



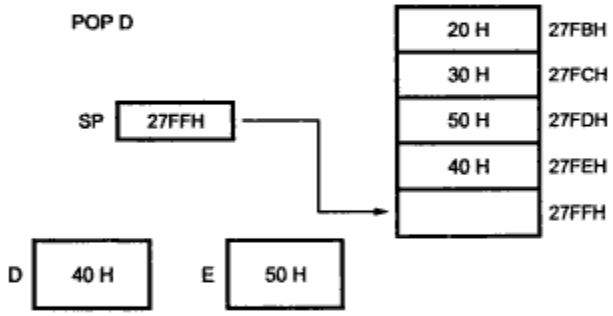

**Instruction 9 :**

POP B




**Instruction 10 :**

POP D




This program shows us how registers can be used for more than one purpose. This program initializes the stack pointer at 27FFH and stores the original contents of BC and DE register pairs in the stack. Now registers B, C, D and E are free to use for intermediate

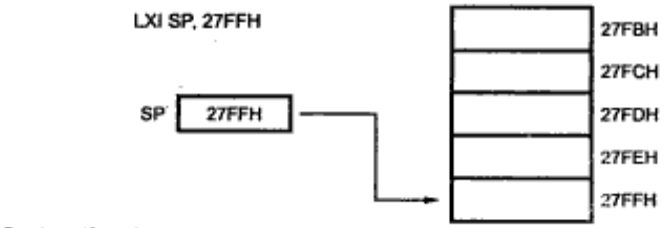
calculations. Once these calculations are over, we can get back the original contents of B, C, D and E registers from stack by POP B and POP D instructions. We know that stack works in FILO fashion. Therefore, the sequence of getting the contents back from the stack should be exactly in the reverse order that of the sequence of storing the contents in the stack. In this example the contents of BC register pair are stored after storing the contents of DE register pair in the stack. But while getting the contents of register pairs from the stack, first BC register pair contents are retrieved and then DE register pair contents are retrieved.

➠ **Example 5.1 :** *Exchange the contents of BC and DE registers without using any other MPU general purpose register.*

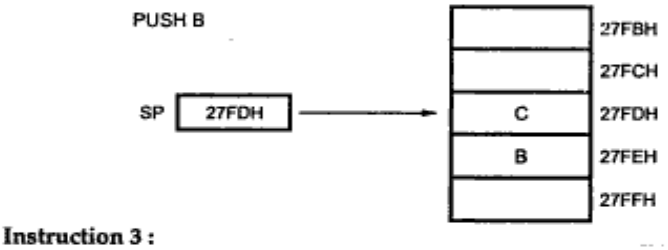
**Solution :**
```
LXI  SP,  27FFH
PUSH B
PUSH D
POP  B
POP  D
```

We will see the contents of stack and stack pointer after execution of each instruction in the above program.

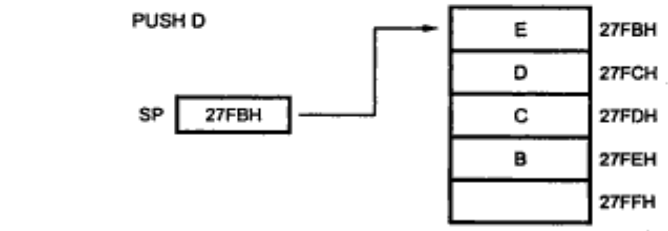
**Instruction 1 :**



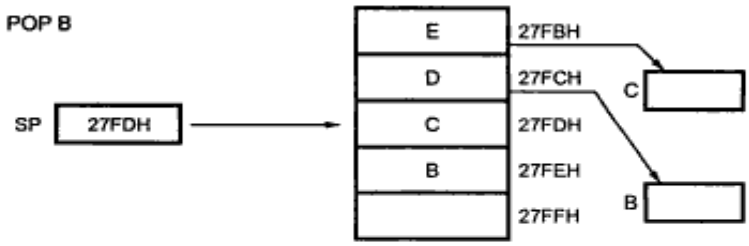

**Instruction 2 :**



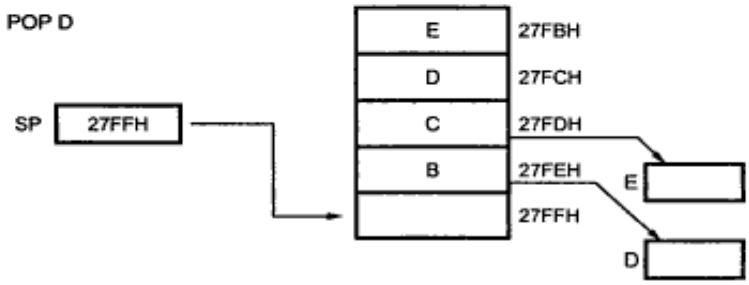

**Instruction 3 :**



**Instruction 4 :**




**Instruction 5 :**




This program shows us that if you reverse the order of retrieving the contents from the stack we will not get the original contents of the register pairs. Here, the contents of DE and BC register pairs will be exchanged after execution of the program.

**Example 5.2 :** *Write a program to load the flag register contents in C register.*

**Solution :** We know that 8085 does not provide any instruction to transfer the contents of flag register to any general purpose register. Therefore, to load flag register contents into any register, it is necessary to use stack. Following program explains how to use stack to load flag register contents in the C register.

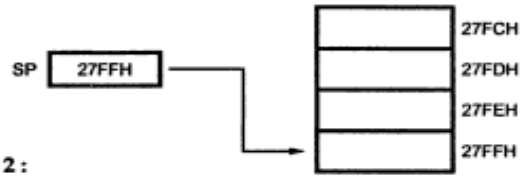
**Program :**

```
LXI SP, 27FFH
PUSH PSW
POP B
```

We will see the contents of stack and stack pointer after execution of each instruction in the program.

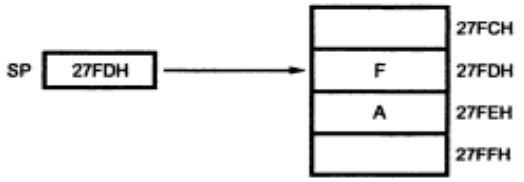
**Instruction 1 :**

LXI SP, 27FFH



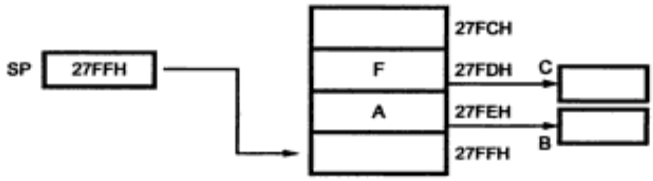

**Instruction 2 :**

PUSH PSW




**Instruction 3 :**

POP B




## CALL Address and RET : Implements Subroutines

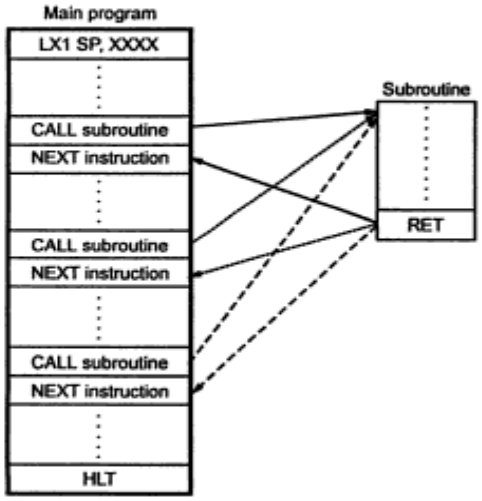



**Fig. 5.6 Program flow**

Whenever we need to use a group of instructions several times throughout a program there are two ways to avoid rewriting of the group of instructions. One way is to write the group of instructions as a separate subroutine. We can then just CALL the subroutine whenever we need to execute that group of instructions. For calling the subroutine we have to store the return address on the stack. This process takes some time. If the group of instructions is bit enough then this overhead time and execution time are comparable. In such cases, it is not desirable to write subroutines. For these cases, we can use macros. Macro is also a group of

instructions. Each time we "CALL" a macro in our program, the assembler will insert the defined group of instructions in place of the "CALL". An important point here is that the assembler generates machine codes for the group of instructions each time macro is called. So there is not overhead time involved in calling and returning from a subroutine. The disadvantage of macro is that it generates in line code each time when the macro is called which takes more memory.

### 5.1.2 Subroutines

From the previous discussions, we know that the subroutine is a group of instructions stored as a separate program in the memory and it is called from the main program whenever required.

The 8085A microprocessor has two instructions to implement subroutines: **CALL** and **RET**. CALL instruction is used to call a subroutine in the main program and RET instruction is the last instruction in the subroutine to return it back to the main program. The CALL instruction saves the address of the instruction following it and then transfers the program control to the first instruction in the subroutine. When subroutine execution is completed the RET instruction reads the return address from the stack and transfers control back to the instruction following the CALL.

**Main program :**

```
6000H    LXI  SP,3000H
6003H         -
              -
              -
6010H    CALL DELAY(6500H)
6013H         -
```

**DELAY subroutine**

```
6500H    DELAY :    MVI C, FFH
6502H    BACK  :    DCR C
6503H               JNZ BACK
6506H               RET
```

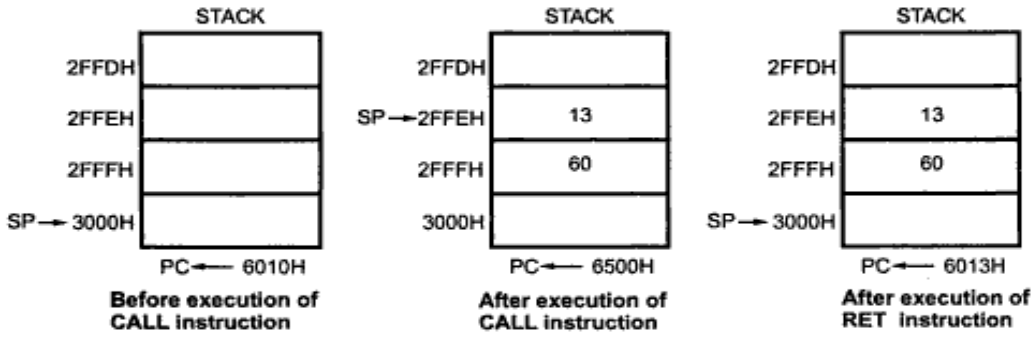



**Fig. 5.7 Details in the execution of CALL and RETURN instructions**

Here, the main program initializes stack pointer at 3000H memory location and executes instructions in sequence till the execution of CALL instruction. After execution of CALL instruction program control is transferred to the delay subroutine stored at memory address 6500H. Before transfer of control to the subroutine the address of the instruction (6013H) which is after the CALL instruction, is stored in the stack. At the end of delay subroutine RET instruction is executed, which reads the return address (6013H) from the stack and transfers the program control to the instruction which is after CALL instruction.

### Conditional Call and Return Instructions :

In addition to the unconditional CALL and RET instructions, the 8085A instruction set includes eight conditional CALL instructions and eight conditional RET instructions. These conditions are checked by reading the status of respective flags. If the condition associated with the conditional CALL is not met, the instruction following the CALL is executed. If the condition is met, the program counter contents are saved on the stack, and the address contained in the CALL instruction is loaded into program counter. The number of machine cycles and T-states required by a conditional CALL depends on whether or not the condition is satisfied. When the condition is not satisfied, two machine cycles with a total of nine T-state are required to fetch, decode and execute the instruction. When the condition is satisfied, five machine cycles with 18 T-states are required.

### Conditional CALL Instructions

| Instruction code | Description | Condition for CALL |
|---|---|---|
| CC | Call on carry | CY = 1 |
| CNC | Call on not carry | CY = 0 |
| CP | Call on positive | S = 0 |
| CM | Call on minus | S = 1 |
| CPE | Call on parity even | P = 1 |
| CPO | Call on parity odd | P = 0 |
| CZ | Call on zero | Z = 1 |
| CNZ | Call on not zero | Z = 0 |

**Table 5.1 Conditional calls**

In case of a conditional return instruction, the sequence returns to the main program if the condition is met otherwise, the sequence in the routine is continued.

| Instruction code | Description | Condition for RET |
|---|---|---|
| RC | Return on carry | CY = 1 |
| RNC | Return on not carry | CY = 0 |
| RP | Return on positive | S = 0 |
| RM | Return on minus | S = 1 |
| RPE | Return on parity even | P = 1 |
| RPO | Return on parity odd | P = 0 |
| RZ | Return on zero | Z = 1 |
| RNZ | Return on not zero | Z = 0 |

**Table 5.2 Conditions for return**

## 5.2 Parameters Passing Techniques

We often want a subroutine to process some data or address variable from the main program. For processing it is necessary to pass these address variables or data, usually referred to as passing parameters to the subroutine. There are four ways to pass parameters to and from the subroutine :

1. Using registers
2. Using general memory
3. Using pointers
4. Using stack

### 5.2.1 Passing Parameters using Registers

The data, to be passed is stored in the registers and these registers are accessed in the subroutine to process the data. In this technique, the main program loads internal registers with appropriate values before calling the subroutine and subroutine then obtains these values by referring pre-defined registers, as shown in the following example.

**Example :**

**Passing Parameters Using Registers**

```
; Main program
            .
            .
            MVI C, 08H    ; Data to be passed is loaded in the
                          ; register
            CALL SUB1
            MOV A, D      ; Main program accesses result from
                          ; register.
            .
            .
```

```
; subroutine
    SUB1:    MOV B, C      ; Subroutine accesses the data from C
                           ; register
             .
             MOV D, A      ; Stores result in D register.
             RET
```

### 5.2.2 Passing Parameters using Memory

For the cases where we have to pass few parameters to and from a subroutine, registers are a convenient way to do it. However in cases where we need to pass a large number of parameters to subroutine we use memory. This memory may be a dedicated section of general memory or a part of stack. In this technique, the main program loads the pre-defined memory locations with appropriate values before calling the subroutine and subroutine then obtains these values by referring these predefined memory locations, as shown in the following example.

**Example :**

**Passing Parameters Using General Memory**

```
; Main program
             LXI H, 2200H ; Initialize memory pointer
             MVI M, 50H   ; Load data into memory
             .
             .
             CALL SUB1
             LDA 2300H    ; Main program accesses result from
                          ; memory location 2300H.
          .
; Subroutine
    SUB1:    LDA 2200H    ; Subroutine accesses the data from
                          ; memory location.
             .
             .
             STA 2300H    ; Stores result in memory location
                          ; 2300H
             RET
```

The subroutine program stores the generated results in the memory locations before execution of RET. The main program accesses results from the these memory locations for further processing.

### 5.2.3 Passing Parameters using Pointers

In this technique, the main program stores the parameters to be passed in the memory, usually in the consecutive memory locations. Then it loads the internal register pair or pre-defined memory locations with the starting address of the parameter list. The subroutine obtains parameter list by accessing it in sequence from the given address.

**Example :**

**Passing Parameters Using Pointers**

```
; Main program
```

```
      LXI  H, 2200H     ; Initialize memory
      MOV  M, A
      INX  H
      MOV  M, B
        :
        :
      INX  H
      MVI  M, 30H
        :
      SHLD 3000H        ; Store memory pointer
      CALL SUB1
        .
        .
; Subroutine
SUB1: LHLD 3000H        ; Subroutine access pointer (address) to data
        MOV A, M
          .
          .
        RET
```

### 5.2.4 Passing Parameters using Stack

The stack can be used to pass parameters. To pass parameters to the subroutine using stack, it is necessary to push them on the stack before the call for the subroutine in the main program. The instructions in the subroutine read these parameters from the stack. Whenever the stack is used to pass parameters, it is very important to keep track of what is pushed on the stack and where the stack pointer points all the time in the program.

**Example :**

**Passing Parameters Using Stack**

```
; Main program
    LXI B, 1020H ; Load BC register pair with 1020H
    PUSH B       ; Store BC register pair on stack.
    CALL SUB1
       :
; Subroutine
    SUB1
    POP H        ; Store the return address into HL register
                 ; pair.
    POP B        ; The subroutine accesses data from stack using
                 ; BC register pair.
    PCHL         ; Here it is not possible to use return
                 ; instruction, since stack pointer not pointing
                 ; the return address. But the address is available
                 ; in HL. Thus the PCHL instruction is used to
                 ; transfer program control to the main program.
```

**Example :** Write a program to exchange the higher and lower nibble of ten 8-bit numbers stored from location 2200H. Make use of subroutine and explain different parameter passing techniques.

**Solution :**

**a) Parameter passing using register A**

**Main program :**

```
            LXI  SP, 27FFH      ; Initializes stack pointer
            LXI  H, 2200H       ; Initializes memory pointer
            MVI  C, 0AH         ; Initializes counter
BACK:       MOV  A, M           ; Stores number (passing parameter)
                                ; in A register
            CALL EXCHANGE       ; Calls subroutine exchange
            MOV  M, A           ; Stores the result from register A
            INX  H              ; Increments memory pointer
            DCR  C              ; Decrements counter
            JNZ  BACK           ; If not zero, repeat
            HLT                 ; Stop
```

**Subroutine program :**

```
EXCHANGE:   RLC
            RLC
            RLC
            RLC                 ; Rotate 4 times left to exchange
                                ; the nibbles
            RET                 ; Return to the main program
```

**b) Passing parameters using memory location**

**Main program :**

```
            LXI  SP, 27FFH      ; Initializes stack pointer
            LXI  H, 2200H       ; Initializes memory pointer
            MVI  C, 0AH         ; Initializes counter
BACK :      MOV  A, M           ; Gets the number
            STA  2300H          ; Stores the number (parameter) at
                                ; 2300H
            CALL EXCHANGE       ; Calls subroutine exchange
            LDA  2300H          ; Gets the result
            MOV  M, A           ; Stores the result
            INX  H              ; Increments memory pointer
            DCR  C              ; Decrements counter
            JNZ  BACK           ; If not zero, repeat
            HLT                 ; Stop
```

**Subroutine program :**

```
EXCHANGE:   LDA  2300H          ; Gets the parameter
            RLC
            RLC
            RLC
            RLC                 ; Rotate 4 times left to exchange
                                ; the nibbles
            STA  2300H          ; Store the result
            RET                 ; Return to main program
```

**c) Passing parameters using pointer**

```
            LXI SP, 27FFH      ; Initializes stack pointer
            LXI H, 2200H       ; Initializes memory pointer
            MVI C, 0AH         ; Initializes counter
BACK:       SHLD 2300H         ; Store pointer to passing
                               ; parameter
            CALL EXCHANGE      ; Calls subroutine exchange
            INX H              ; Increments memory pointer
            DCR C              ; Decrements counter
            JNZ BACK           ; If not zero, repeat
            HLT                ; Stop
```

**Subroutine program :**

```
EXCHANGE:   LHLD 2300H         ; Gets pointer to parameter
            MOV A, M           ; Gets number
            RLC
            RLC
            RLC
            RLC                ; Rotate 4 times left to exchange
                               ; the nibbles
            MOV M, A           ; Stores the result
            RET                ; Return to main program.
```

**d) Passing parameters using stack.**

```
            LXI SP, 27FFH      ; Initializes stack pointer
            LXI H, 2200H       ; Initializes memory pointer
            MVI C, 0AH         ; Initializes counter
BACK :      MOV A, M           ; Gets the number
            PUSH PSW           ; Stores number in stack as a
                               ; parameter
            CALL EXCHANGE      ; Calls subroutine exchange
            MOV M, A           ; Stores the result
            INX H              ; Increments memory pointer
            DCR C              ; Decrements counter
            JNZ BACK           ; If not zero repeat
            HLT                ; Stop
```

**Subroutine program :**

```
EXCHANGE:   POP H              ; Stores return address in HL
            POP PSW            ; Gets the number in accumulator
            RLC
            RLC
            RLC
            RLC                ; Rotate 4 times left to exchange
                               ; the nibbles
            PCHL               ; Return to main program.
```

## 5.3 Subroutine Documentation

Subroutine program must provide enough information so that other users can utilize the subroutine without having to examine its internal structure. So along with subroutine program it is necessary to give the following guidelines.

1. Description of the purpose of the subroutine
2. A list of passing parameters
3. Return value
4. Registers and memory and memory locations used
5. Sample code

If these guidelines are followed while writing the subroutine then the subroutine can be easily used as a library function in other applications if required.

## 5.4 Advanced Subroutine Concepts

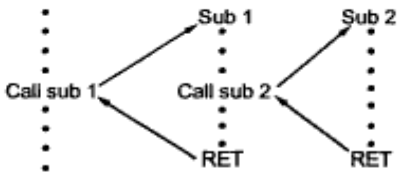



**Fig. 5.8 Transfer of control with nested subroutines**

When one subroutine calls another subroutine to complete a particular task, the operation is called **nesting**. The second subroutine may in turn call a third subroutine and so on each successive CALL without an intervening return creates an additional level of nesting. These routines are called **nested subroutines**. Fig. 5.8 shows the transfer of control with nested subroutines.

Nested subroutines are commonly classified as :

- Re-entrant Subroutine
- Recursive Subroutine

### 5.4.1 Re-entrant Subroutine

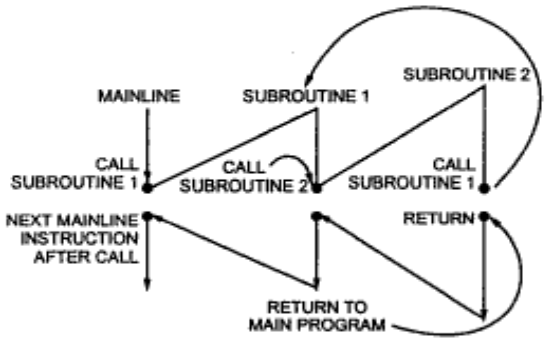



**Fig. 5.9 Flow of program execution for re-entrant subroutine**

In some situations it may happen that subroutine1 is called from main program, subroutine2 is called from subroutine1 and subroutine1 is again called from subroutine2. In this situation program execution flow re-enters in the subroutine1. This type of subroutines are called re-entrant subroutines. The flow of program execution for re-entrant subroutine is shown in Fig. 5.9.

### 5.4.2 Recursive Subroutine

A recursive subroutine is a subroutine which calls itself. Recursive subroutines are used to work with complex data structures called trees. If the subroutine is called with N (recursion depth) = 3, then the N is decremented by one after each subroutine CALL and the subroutine is called until N = 0. Fig. 5.10 shows the flow diagram and pseudo code for recursive subroutine.

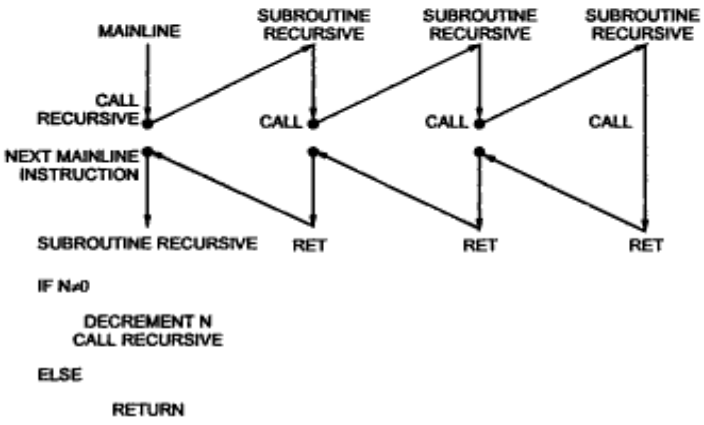



**Fig. 5.10 Flow diagram and pseudo code for recursive subroutine**

**Limitations :**

The level of subroutine nesting cannot exceed that supported by the available stack memory in the system. Since the stack pointer is usually initialized to the highest available read/write memory location, and the stack grows toward lower addresses, it is possible that too many PUSH or CALL operations without a sufficient number of intervening POP or RET operations result in the overflow of stack. For example, assume that read/write memory allocated in the system for stack has address range from 2000H to 27FFH. In this case, if stack grows beyond 2000H then it is called as overflow of stack. Care must be taken in memory allocation and program design to prevent this.

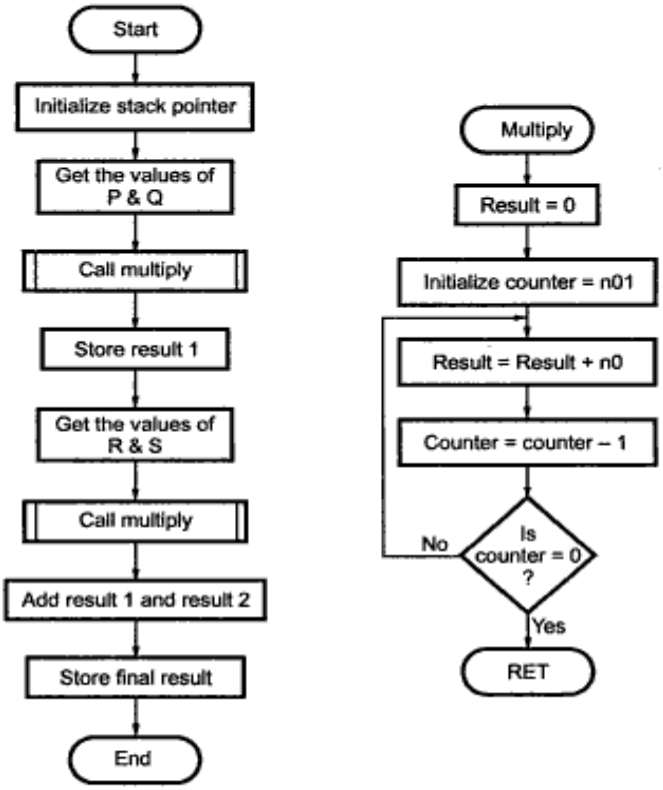
### Lab Experiment 61 : Solve given equation.

**Statement :** Write a program to solve following equation

$X = (P \times Q) + (R \times S)$

Assume P, Q, R and S numbers are stored at 2200H, 2201H, 2202H and 2203H memory locations respectively. Store the result in memory locations 2204H and 2205H. Make use of subroutine.

**Flowchart :**




**Main program :**

```
        LXI SP, 27FFH      ; Initialize stack pointer
        LXI H, 2200H       ; Initialize memory pointer
        CALL MULTIPLY      ; Call subroutine multiply
        SHLD 2204H         ; Store the result 1
        LXI H, 2202H       ; Set memory pointer for next two numbers
        CALL MULTIPLY      ; Call subroutine multiply
        XCHG               ; Save result in DE register pair
        LHLD 2204H         ; Get the result 1
        DAD D              ; Add result 1 and result 2
        SHLD 2204H         ; Store final result
        HLT                ; Stop
```

**Multiply subroutine :**

```
MULTIPLY:    MOV C, M      ; Initialize number1 as a counter
             MVI D, 00H    ;
```

```
            INX H          ;
            MOV E, M       ; Get number 2
            LXI H, 0000H   ; Result = 0
BACK :      DAD D          ; Result = Result + number2
            DCR C          ; Decrement counter
            JNZ BACK       ; If not zero, repeat
            RET            ; Return to main program
```

## Review Questions

1. *What is stack ? Explain the use of the stack, and stack pointer and how they are affected by instructions such as PUSH, POP, CALL and RET.*
2. *What is subroutine ? How is it useful ? Explain the use of stack in CALL and RETURN instructions.*
3. *The first four instructions of a typical subroutine are :*

   *PUSH PSW, PUSH H*

   *PUSH B, PUSH D*

   *What will be the last five instructions of the subroutine? Explain clearly.*
4. *If the CALL and RET instructions are not provided in the 8085, could it be possible to write subroutines for this microprocessor ? If so how will you call and return from the subroutine ?*
5. *Discuss the passing parameters techniques in the subroutine.*
6. *Explain the necessity of subroutine documentation.*
7. *What do you mean by nested subroutines ?*
8. *Explain the re-entrant subroutine.*
9. *Explain the recursive subroutine with the help of example.*

❑❑❑