**MARUDHAR KESARI JAIN COLLEGE FOR WOMEN,
VANIYAMBADI**

PG & RESEARCH DEPARTMENT OF PHYSICS

CLASS : III BSC PHYSICS
SUBJECT NAME : FUNDAMENTALS OF MICROPROCESSOR-8085
SUBJECT CODE : FEPH 63A

SYLLABUS

UNIT – II

INSTRUCTIONS & ADDRESSING MODES

Data transfer/ copy Instructions-Arithmetic, Logical - Two examples each instructions - Branch instructions-Unconditional and conditional jump - Call and Return instructions - Stack and Stack related instructions - I/O and Machine control instructions - Addressing modes.

4.3 DATA TRANSFER INSTRUCTIONS - I

The data transfer instructions transfer or copy a data from register to register, register to memory or memory to register. It also transfers an immediate data (data contained in the instruction itself) to register or memory.

Data transfer instructions do not affect any flags.

In this section, we will understand some simple instructions. Some of the slightly complicated data transfer instructions are discussed in the next chapter.

NOTE: The contents of registers and memory are in binary form. Only for convenience, the data are represented in Hexadecimal.

The complete instruction set of 8085, with their respective hex codes are given in Appendix-I. The hex codes of many instructions are freely used in many examples in this book. We can get the hex code of any instruction by simply referring to Appendix-I and it need not be memorized.

a) **Move Instructions:**

*i. **Move between Register and Register***

These instructions copy an 8-bit data from an 8-bit register to another 8-bit register. The registers used are A, B, C, D, E, H and L. The general form of move instruction is,

MOV r_d, r_s

where r_s is the register that acts as a source and r_d is the register that acts as a destination.

In this instruction, MOV is the opcode and r_d and r_s are operands.

This operation can also be represented as,

$$(r_d) \leftarrow (r_s)$$

The parantheses are short-hand for 'contents of '.

After the operation, only the contents of destination register is altered but the contents of source register is not changed.

For example, the instruction,

MOV B,A

moves (copies) the contents of A register to B register.

$$(B) \leftarrow (A)$$

Let us assume that register A has a data 45_H and register B has a data $7C_H$ to start with Let us see what happens after executing the instruction MOV B,A.

Before:.

A	B
45_H	$7C_H$

Instruction: MOV B,A

After:

A	B
45_H	45_H

After the operation, only the contents of B register (destination) has changed. The contents of A register (source) is not altered.

The move instructions are **one-byte instructions,** because both the opcode and operand are specified in a single byte. For example, the instruction MOV B,A has a single byte hex code 47_H. When the hex code of the instruction is loaded in memory, only one memory location is required.

ii. Move between Register and Memory:

The letter **M** is used to represent memory. But there are thousands of memory locations (2048 locations for a 2KX8 memory) and each location has a distinct address. In 8085, the desired 16-bit memory address is first loaded in HL register before using any instruction that has a memory reference, M. The general form of the instruction is,

MOV M,r and **MOV r,M**

where r is a general purpose register or accumulator. It is also an one-byte instruction.

The operation can be represented as,

M ((HL)) ← (r) and **(r) ← M((HL))**

(HL) represents content of HL register. ((HL)) represents contents of memory whose address is available in HL register pair. M with ((HL)) is optional.

For example, to move a data from accumulator (A) to memory (M) the corresponding instruction is MOV M,A. This instruction duplicates the contents of the accumulator on to a memory location whose address is stored in HL register pair.

Let us take the initial content of register A as 56_H and the content of register HL be 2050_H. Also, let us assume the memory whose address is 2050_H has a random data 99_H.

Before:

A
56_H

H	L
20_H	50_H

Memory Address	Data
2050_H	99_H

The instruction is MOV M,A.

After: The contents of A is moved to memory whose address is in HL register

A
56_H

H	L
20_H	50_H

Memory Address	Data
2050_H	56_H

We can see that the contents of A register is not altered. The HL register pair is used as a memory pointer.

Similarly, the instruction, MOV A,M copies the contents of a memory location whose address is in HL register on to A register.

b) Move Immediate Instruction:

The move immediate instruction has the form,

MVI r,data 8 and **MVI M,data 8**

where r is a general purpose register or accumulator and M refers to memory. This instruction transfers the byte of data specified within the instruction itself to a register or memory.

(r) ← **data 8** and **M((HL))** ← **data 8**

This is a **two-byte instruction.** The first byte represents the operation code and the second byte represents the 8-bit data. This type of instructions require two memory locations. The byte corresponding to the opcode is stored in the first memory location and the byte representing the data is stored in the second memory location.

For example, MVI B,25_H transfers the data 25_H to B register. For this instruction, the hex code 06_H is stored in the first memory location and the data 25_H is stored in the next memory location.

The instruction, MVI M, 25_H transfers the data 25_H to a memory location whose address is in HL register. Therefore, the desired address must be loaded in HL first, before executing the MVI M, 25_H instruction.

Example 4.1:

Write a sequence of instructions that will load FF_H in C register and transfer the byte to a memory location whose address is 2050_H. Also write the sequence of instructions as to how the operation is done without using C register.

Solution:

HL register pair is used as a memory pointer. The 16-bit address 2050_H is first loaded in HL register pair. The higher byte 20_H is moved to H register and the lower byte 50_H is moved to L register. The data FF_H is moved to C register. Then the data in C register is moved to the memory. The required sequence of instructions is:

```
MVI  H,20H        ; (H) = 20H
MVI  L,50H        ; (L) = 50H
MVI  C,FFH        ; (C) = FFH
MOV  M,C          ; (C) is moved to the memory whose
                  ; address is in HL register pair
```

To load FF_H in memory without using C register, the sequence of instructions is,

```
MVI  H,20H
MVI  L,50H
MVI  M,FFH        ; The immediate data FFH is moved to
                  ; the memory whose address is in HL
```

NOTE: Comments are written after a semi colon.

c) Load Immediate Register Pair (Load Extended Immediate):

The load immediate register pair instruction is used to load a 16-bit immediate data into a specified register pair. Since 16-bit number is involved, the word 'extended' is also used. The general form of this instruction is,

LXI r_p,data 16

where r_p stands for one of the register pairs BC, DE, HL or the 16-bit Stack Pointer SP.

(r_p) ← data 16

For example, the instruction,

LXI H,2050_H

will load 20_H into H register and 50_H into L register. These instructions are **three byte instructions.** The first byte is for the opcode and the remaining two bytes for the 16-bit data. In the 16-bit data 2050_H, which is made up of two bytes, 20_H is referred to as the higher byte and 50_H is referred to as the lower byte. For the instruction,

LXI H,2050_H

the opcode of the instruction 21_H is stored in the first memory location. The lower byte 50_H is stored in the second memory location followed by the higher byte 20_H in the third memory location.

Now that we have a single instruction, LXI H,2050_H to load HL register pair with a 16-bit data, 2050_H, we need not use two instructions like MVI H,20_H and MVI L,50_H as we have done in the solution to Example 4.1.

The instruction LXI B,1234_H loads the register pair BC with 1234_H (12_H in B register and 34_H in C register).

The instruction LXI SP, $27FF_H$ loads the stack pointer with $27FF_H$. This instruction is used to initialize the top of stack. Stack related operations are discussed in Chapter 5.

Example 4.2:

Write a program to transfer one byte from a memory location whose address is 2450_H to another memory location whose address is 2550_H.

Solution:

```
LXI   H,2450H   ; Load HL pair with 2450H
MOV   A,M       ; Move the data from memory whose address is
                ; 2450H to A
LXI   H, 2550H  ; Load HL pair with 2550H
MOV   M,A       ; Move the data from  A to memory whose
                ; address is 2550H
HLT             ; End of program
```

d) Store Accumulator direct:

Store accumulator direct, **STA addr** stores the contents of the accumulator in the memory whose address is specified in the instruction itself.

M(addr) ← (A)

For example the instruction,

STA 2550_H

stores the contents of the accumulator in the memory whose address is 2550_H (directly given in the instruction).The contents of the accumulator is not altered.

Before:

A
23_H

Memory Address	Data
2550_H	67_H

Instruction: STA 2550_H

After:

A
23_H

Memory Address	Data
2550_H	23_H

e) Load Accumulator direct:

Load accumulator direct, **LDA addr** loads the accumulator with a byte from the memory location whose address is specified in the instruction itself.

(A) ← M(addr)

For example the instruction,

LDA 2450_H

copies the contents of the memory location whose address is 2450_H (directly given in the instruction) into the accumulator. The contents of the memory location is not altered. It is a three byte instruction.

Before:

A
23_H

Memory Address	Data
2450_H	67_H

Instruction: LDA 2450_H

After:

A
67_H

Memory Address	Data
2450_H	67_H

Let us take Example 4.2 once again. With LDA and STA instructions, the program is much simpler.

```
LDA  2450H
STA  2550H
```

So far, we have seen different types of instructions to transfer data between register and register or register and memory. **We should remember that no instruction is available in 8085 (or 8086) to transfer data directly from one memory location to another memory location.**

Some more data transfer instructions are introduced in Chapter 5.

4.4 ARITHMETIC INSTRUCTIONS

The arithmetic group of instructions are used to perform addition, subtraction, increment and decrement operations.

a) Addition Instructions:

i. ADD with register / memory

The add instructions, **ADD r** and **ADD M** are used to add the contents of a specified register or memory to the contents of the accumulator. The result is stored in the accumulator.

$$\mathbf{(A) \leftarrow (A) + (r)}$$

The instruction ADD B, adds the contents of A and B registers and places the result in A register.

$$\mathbf{(A) \leftarrow (A) + (B)}$$

ADD instructions affect all flags.

Example 3.1 is repeated here.

(A)	=	79_H	=	0111 1001
(B)	=	68_H	=	0110 1000
ADD B		$E1_H$	=	1110 0001

The result in the accumulator is $E1_H$.

The result is less than FF_H. Therefore Carry flag is reset.

The MSB of the result is 1. Therefore the Sign flag is set. The sign flag is used only with signed numbers. For unsigned numbers, as in this example, sign flag need not be used.

The result is not zero. Hence the Zero flag is reset.

The result has even parity. So the parity flag is set.

During the addition, a carry is generated at position D_3 and passed on to D_4. Therefore the auxiliary carry flag is set.

The flag register bit position is as shown.

S	Z	X	AC	X	P	X	Cy

Assuming X = 0, the flag register will have the bit values as

1	0	0	1	0	1	0	0

The content of the flag register is 94_H. If we take X = 1, the content of the flag register is BE_H.

Instead of B register, we can use any other register. However, to add the contents of memory with accumulator, the instruction **ADD M** is used. As mentioned earlier, whenever we refer to a memory location using the letter M, we should remember that the 16-bit memory address is in HL register pair.

$$\mathbf{(A) \leftarrow (A) + M((HL))}$$

ii. ADD immediate

The add immediate instruction, **ADI data 8**, adds the contents of the accumulator with an 8-bit data that is included in the instruction itself.

$$\mathbf{(A) \leftarrow (A) + data\ 8}$$

For example, ADI 37_H will add 37_H to the contents of the accumulator and place the result in the accumulator.

iii. ADD with carry

The add with carry instruction is used to add the contents of a specified register or memory along with carry flag to the contents of the accumulator. The result is stored in the accumulator. The instruction is of the form,

ADC r and **ADC M**

The instruction ADC B, adds the contents of A and B registers with the carry flag and places the result in A register.

$$(A) \leftarrow (A) + (B) + Cy.$$

In this, Cy is taken as 1 when the Carry flag is in set condition and taken as 0 if the carry flag is in reset condition at the time of executing the add with carry instruction.

Let us take the Carry flag to be in set condition i.e.,1.

Let (A) = 79_H and (B) = 68_H. On executing ADC B instruction, the contents of A is calculated as,

(A)	=	79_H	=	0111 1001
(B)	=	68_H	=	0110 1000
Cy	=	1	=	1
ADC B		$E2_H$	=	1110 0010

The result in the accumulator is $E2_H$

The instruction ADC M, adds the contents of memory along with the Carry flag to the accumulator and places the result in the accumulator. We know that the address of the memory is available in HL register.

Example 4.3:

Write a program to add two 16-bit numbers 1234_H and $56FF_H$ and store the 16-bit result in two successive memory locations 2050_H and 2051_H.

Solution:

Let us take the first 16-bit number in BC register pair and the second 16-bit number in DE register pair.

(BC) = 1234_H

(DE) = $56FF_H$

First, the lower bytes in C register and E register are added with the help of the accumulator and the result is transferred to the memory location 2050$_H$. In the next step, the higher bytes in B register and D register are added, again, using the accumulator. But now, we have to take into account the carry (if there is a carry) produced by adding the lower bytes. The carry must be included while adding the higher bytes. The result in the accumulator is now transferred to the memory location 2051$_H$. The program is given below.

```
LXI   B,1234H      ; Load 1234H in BC
LXI   D,56FFH      ; Load 56FFH in DE
MOV   A,C          ; Copy C in A
ADD   E            ; (A)    ←     (A) + (E)
                   ; 34H   +   FFH   =   33H and carry 1
STA   2050H        ; Store the lower byte (33H) of result
                   ; in memory
MOV   A,B          ; Copy B in A
ADC   D            ; (A)    ←    (A)    +    (D)    +    Cy
                   ; 12H   +   56H   +   1    =   69H
STA   2051H        ; Store the higher byte (69H)of result in
                   ; memory, the 16-bit result is 6933H
HLT                ; End of program
```

NOTE: The 16- bit addition can be done in a more efficient way by using DAD instruction. However, this method helps us to understand ADC instruction.

iv. ADD immediate with carry

This instruction is of the form, **ACI data 8** and adds with carry, the contents of the accumulator with an 8-bit data that is included in the instruction itself.

(A) ← (A) + data 8 + Cy

All types of ADD instructions affect all flags.

b) Subtract Instructions:

i. SUB with register/memory

The subtract instructions, **SUB r** and **SUB M** are used to subtract the contents of a specified register or memory from the contents of the accumulator. The result is stored in the accumulator. For SUB r, we can write,

$$(A) \leftarrow (A) - (r)$$

The instruction SUB B, subtracts the contents of B register from the contents of A register and places the result in A register.

For example, let

(A)	=	79_H	=	0111 1001
(B)	=	65_H	=	0110 0101
SUB B		14_H	=	0001 0100

The result in the accumulator is 14_H.

On the other hand, if,

(A)	=	65_H	=	0110 0101	
(B)	=	79_H	=	0111 1001	
SUB B		-14_H	=	1110 1100	with a borrow of 1.

The result in the accumulator is EC_H. That is, the result -14_H (minus 14_H) is obtained in 2's complement form. To check ,

$+ 14_H$	=	0001 0100	
1's complement of 14_H	=	1110 1011	
2's complement of 14_H	=	1110 1100	(1's complement plus 1)

That is, -14_H is represented in 2's complement form as EC_H. Also, this operation has produced a borrow and so the Carry flag is set to 1.

The instruction SUB A clears the accumulator since (A) – (A) = 0.

NOTE: Negative numbers are always represented in 2's complement form in all microprocessors and hence in computers.

Instead of B register, we can use any other register. However, to subtract the contents of memory from accumulator, the instruction **SUB M** is used. As mentioned earlier, whenever we refer to a memory location using the letter M, we should remember that the 16-bit memory address is in HL register pair.

$$(A) \leftarrow (A) - M((HL))$$

ii. SUB immediate

The sub immediate instruction **SUI data 8**, subtracts the 8-bit data that is included in the instruction from the accumulator.

$$(A) \leftarrow (A) - \text{data 8}$$

For example, SUI 37_H will subtract 37_H from the contents of the accumulator and places the result in the accumulator.

iii. SUB with borrow

The subtract with borrow instruction is used to subtract the contents of a specified register or memory along with borrow (carry flag) from the contents of the accumulator. The result is stored in the accumulator. The instruction is of the form,

SBB r and **SBB M**

For SBB r, we can write,

$$(A) \leftarrow (A) - (r) - Cy.$$

The instruction SBB B, subtracts the contents of B and the carry flag from the accumulator and places the result in A register.

$$(A) \leftarrow (A) - (B) - Cy$$

In this, Cy is taken as 1 when the carry flag is in set condition and taken as 0 if the carry flag is in reset condition at the time of executing the sub with borrow instruction.

The instruction SBB M, subtracts the contents of memory along with the Carry flag from the accumulator and places the result in the accumulator. We know that the address of the memory is available in HL register.

iv. *SUB immediate with borrow*

This instruction is of the form, **SBI data 8** and subtracts a data byte with borrow (carry flag) from the contents of the accumulator and places the result in the accumulator. The 8-bit data is included in the instruction itself.

(A) ← (A) — data 8 — Cy

All types of SUB instructions affect all flags.

c) Increment / Decrement Instructions:

i. *Increment Register or Memory*

The instruction **INR r** increments the contents of the specified register by 1.

(r) ← (r) + 1

The result is in the specified register itself. For example, INR C increments the contents of C register by 1 and the new value is stored in C register.

The instruction **INR M** increments the contents of memory location, pointed to by HL register by 1.

ii. *Decrement Register or Memory*

The instruction **DCR r** decrements the contents of the specified register by 1.

(r) ← (r) — 1

The result is in the specified register itself. For example, DCR B, decrements the contents of B register by 1 and the new value is stored in B register.

The instruction **DCR M** decrements the contents of memory location, pointed to by HL register by 1.

Increment / Decrement register or memory instructions affect all flags except the carry flag.

iii. Increment Register pair

The instruction **INX r_p** increments the contents of the specified register pair by 1.

$$(r_p) \leftarrow (r_p) + 1$$

The register pairs used are BC, DE, HL and the 16-bit Stack Pointer. For example if the initial value in HL register pair is $14FF_H$, then the instruction INX H increments the 16-bit value in HL register pair by 1 and the new value 1500_H is stored in HL.

iv. Decrement Register pair

The instruction **DCX r_p** decrements the contents of the specified register pair by 1.

$$(r_p) \leftarrow (r_p) - 1$$

The register pairs used are BC, DE, HL and the 16-bit Stack Pointer. For example, if the initial value in DE register pair is 1500_H, then the instruction DCX D decrements 16-bit value in DE register pair by 1 and the new value $14FF_H$ is stored in DE.

Increment / Decrement register pair instructions do not affect any flags.

Example 4.4:

Specify the register and memory contents during each step after executing the following instructions.

```
LXI   H,2050H
MVI   B,75H
MOV   M,B
INR   M
INR   L
INR   H
INX   H
MOV   M,B
```

Solution:

```
LXI   H,2050H      ; (HL) = 2050H
MVI   B,75H        ; (B)  =  75H
MOV   M,B          ; The contents of B =  75H is moved to
                   ; memory whose address is in HL = 2050H
INR   M            ; Contents of memory = 75H is
                   ; incremented to 76H
INR   L            ; (L) = 50H is incremented to 51H
INR   H            ; (H) = 20H is incremented to 21H
INX   H            ; (HL) = 2151H is incremented to 2152H
MOV   M,B          ; (B) = 75H is moved to memory whose
                   ; address is now 2152H
```

d) Double-Add Instructions

The instruction **DAD r_p** adds the contents of the specified register pair to HL register pair. The result is left in the HL register pair.

$$\mathbf{(HL) \leftarrow (HL) + (r_p)}$$

If the result exceeds $FFFF_H$, the carry flag is set. No other flags are affected.

For example, DAD B instruction adds the contents of BC register pair with the contents of HL register pair and the 16-bit result is stored in HL register pair. This instruction is useful to add two 16-bit numbers with one instruction.

Let us take Example 4.3 once again but give a different type of solution.

Example 4.5:

Write a program to add two 16-bit numbers 1234_H and $56FF_H$ and store the 16-bit result in two successive memory locations 2050_H and 2051_H.

Solution:

Let us take the first 16-bit number in BC register pair and the second 16-bit number in HL register pair.

$$(BC) = 1234_H$$

$$(HL) = 56FF_H$$

Add them using DAD B instruction and get the result in HL and then transfer it to memory.

```
LXI B,1234H    ; Load BC with 1234H
LXI H,56FFH    ; Load HL with 56FFH
DAD B          ; Add (BC) with (HL), result in HL
MOV A,L        ; Move lower byte of result to memory
STA 2050H      ; whose address is 2050H
MOV A,H        ; Move higher byte of result to memory
STA 2051H      ; whose address is 2051H
HLT            ; End of program
```

4.5 LOGIC INSTRUCTIONS

Logic instructions in 8085 are useful for performing various logical operations with the contents of the accumulator. These instructions perform AND, OR and EX-OR operations. This logical group also includes instructions to compare two bytes and rotate the contents of the accumulator. Let us discuss them one by one.

a) AND Instructions

i. AND with register/memory

The instruction **ANA r**, performs logical AND operation between the specified register and accumulator on a bit by bit basis. The result is placed in the accumulator.

The AND instruction modifies S, Z and P flags. But Cy flag is always reset and AC flag is set .

For example, ANA B instruction performs logical AND operation between the 8-bits in the B register and A register and places the result in A register.

If (A) = 97_H = 1001 0111

and if (B) = $C5_H$ = 1100 0101

then ANA B results in, (A) = 1000 0101 = 85_H

The instruction **ANA M**, performs logical AND operation between the memory and accumulator on a bit by bit basis. The result is placed in the accumulator.

ii. AND Immediate

The instruction **ANI data 8** performs logical AND operation between the contents of the accumulator and the immediate data which is given in the instruction itself.

S, Z and P flags are modified. Cy flag is reset and AC flag is set.

ANA r and ANI data 8 can be used for **masking** specific bits. For example,

if (A) = 97_H, the instruction ANI $0F_H$ masks the upper 4-bits (or the upper nibble).

(A) = 97_H = 1001 0111

data 8 = $0F_H$ = 0000 1111

ANI $0F_H$ gives the result as 0000 0111 = 07_H in A register.

Similarly, the instruction ANI $F0_H$ masks the lower nibble.

b) OR Instructions

i. OR with register/memory

The instruction **ORA r**, performs logical OR operation between the specified register and accumulator on a bit by bit basis. The result is placed in the accumulator.

The OR instruction modifies S, Z and P flags. Both Cy flag and AC flag are reset.

For example, ORA B instruction performs logical OR operation between the 8-bits in the B register and A register and places the result in A register.

If (A) = 97_H = 1001 0111

and if (B) = $C5_H$ = 1100 0101

then ORA B results in, (A) = 1101 0111 = $D7_H$

The instruction **ORA M**, performs logical OR operation between the memory and accumulator on a bit by bit basis. The result is placed in the accumulator.

ii. OR Immediate

The instruction **ORI data 8** performs logical OR operation between the contents of the accumulator and the immediate data which is given in the instruction itself.

S, Z and P flags are modified. Cy flag and AC flag are reset.

c) EX-OR Instructions

i. EX-OR with register/memory

The instruction **XRA r**, performs logical EX-OR operation between the specified register and accumulator on a bit by bit basis. The result is placed in the accumulator.

The XRA r instruction modifies S, Z and P flags. Cy flag and AC flag are reset.

For example, XRA B instruction performs logical EX-OR operation between the 8-bits in the B register and A register and places the result in A register.

If (A) = 97_H = 1001 0111

and if (B) = $C5_H$ = 1100 0101

then XRA B results in, (A) = 0101 0010 = 52_H

With (A) = 97_H, let us perform XRA A.

(A) = 97_H = 1001 0111

(A) = 97_H = 1001 0111

XRA A results in, (A) = 0000 0000 = 00_H

i.e., XRA A clears the accumulator. With this single instruction, we can clear accumulator, Cy flag and AC flag.

The instruction **XRA M**, performs logical EX-OR operation between the memory a accumulator on a bit by bit basis. The result is placed in the accumulator.

ii. EX-OR Immediate

The instruction **XRI data 8** performs logical EX-OR operation between the contents the accumulator and the immediate data which is given in the instruction itself.

S, Z and P flags are modified. Cy flag and AC flag are reset.

d) Compare Instructions

i. Compare with register/memory

The instruction **CMP r**, compares the contents of the specified register with that of th accumulator. The comparison is performed by subtracting the contents of the specifie register from that of the accumulator. **The contents of the registers are not altered afte the execution of this instruction**. The result of the comparison is indicated by the C and Z flags as given below.

If (A) < (r), Cy flag is set and Z flag is reset.

If (A) = (r), Z flag is set and Cy flag is reset.

If (A) > (r), Cy and Z flags are reset.

The **CMP r** instruction also modifies S, P, AC flags.

For example, CMP B compares the contents of B register with the contents of th accumulator. This is done by subtracting the contents of B from that of A , but the content of both the registers are not altered. The result of the comparison is indicated by Cy and flags.

If (A) = 97_H and

(B) = $C5_H$

then CMP B instruction sets the Cy flag and resets the Z flag since (A) < (B). The content of A and B are not altered.

The instruction **CMP M**, compares the contents of memory with that of the accumulator.

ii. Compare Immediate

The instruction **CPI data 8** compares the immediate byte with the contents of the accumulator by subtraction. Only the flags are modified as in the previous example.

e) Rotate Instructions:

The rotate instructions are used to rotate the contents of the accumulator to the left or right, through carry or without carry.

i. Rotate Accumulator Left without carry

The instruction **RLC** rotates the contents of the accumulator by one bit position to the left. The MSB bit D_7 is shifted to LSB bit D_0. Also, the D_7 bit becomes the carry flag bit. Only carry flag is modified. This is shown in Fig (4.3).

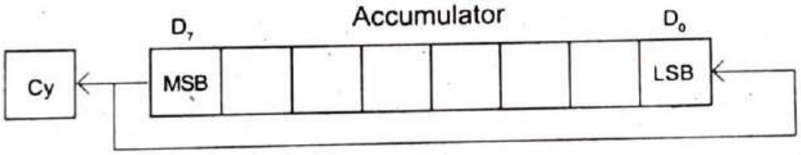
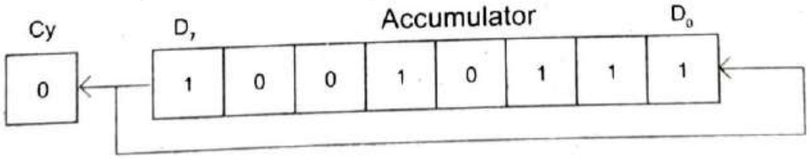
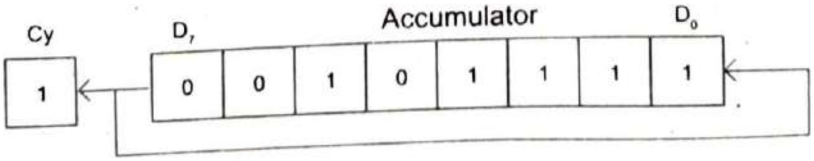


Fig (4.3)

For example, if (A) = 97_H and if Cy flag = 0, let us see what happens when RLC is executed.



After executing RLC instruction, the contents of accumulator and the status of the Cy flag is as shown below.



Contents of the accumulator become $2F_H$ and Cy = 1.

ii. Rotate Accumulator Left through carry

The instruction **RAL** rotates the contents of the accumulator by one bit position to the left through the carry flag. The MSB bit D_7 becomes the carry flag bit and the initial carry flag bit is shifted to LSB D_0. Only carry flag is modified. This is shown in Fig (4.4).

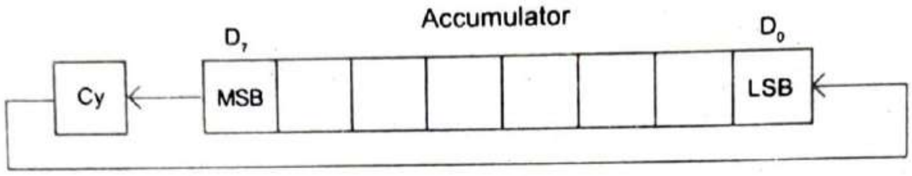
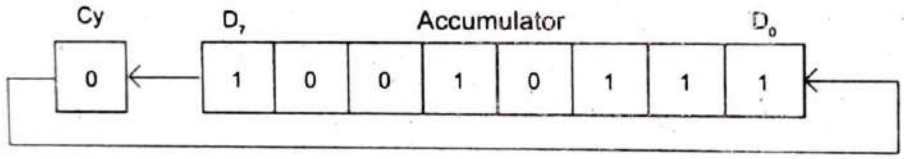
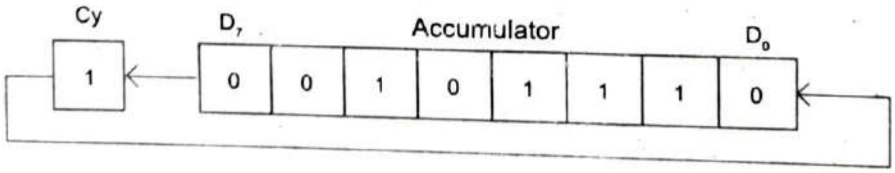


Fig (4.4)

For example, if (A) = 97_H and if Cy flag = 0, let us see what happens when RAL is executed.



After executing RAL instruction, the contents of accumulator and the status of the Cy flag is as shown below.



Contents of the accumulator become $2E_H$ and Cy = 1.

iii. Rotate Accumulator Right without carry

The instruction **RRC** rotates the contents of the accumulator by one bit position to the right. The LSB bit D_0 is shifted to MSB bit D_7. Also, the D_0 bit becomes the carry flag bit. Only carry flag is modified. This is shown in Fig (4.5).

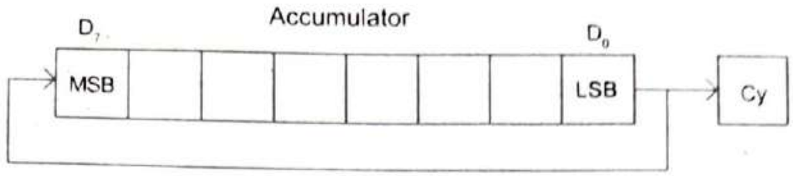
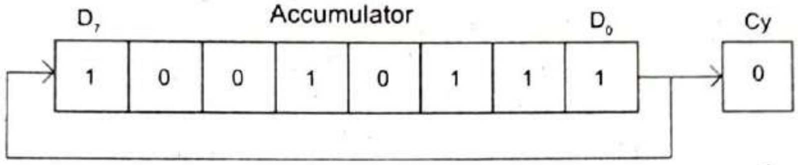
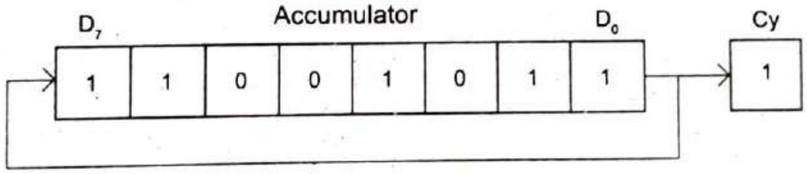


Fig (4.5)

For example, if (A) = 97$_H$ and if Cy flag = 0, let us see what happens when RRC is executed.



After executing RRC instruction, the contents of accumulator and the status of the Cy flag are as shown below.



Contents of the accumulator become CB$_H$ and Cy = 1.

iv. Rotate Accumulator Right through carry

The instruction **RAR** rotates the contents of the accumulator by one bit position to the right through the carry flag. The LSB bit D$_0$ becomes the carry flag bit and the initial carry flag bit is shifted to MSB D$_7$. Only carry flag is modified. This is shown in Fig (4.6).

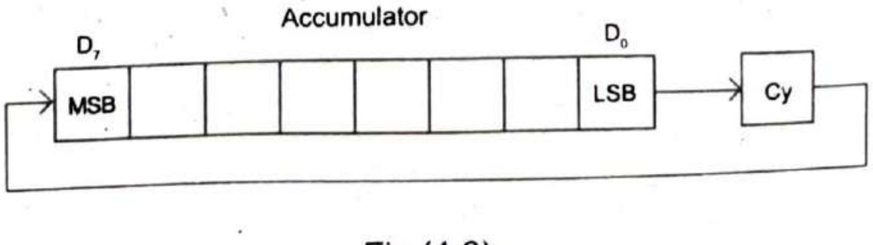
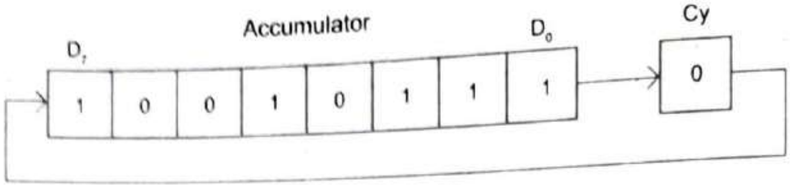
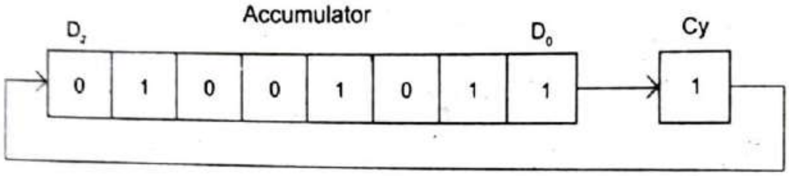


Fig (4.6)

For example, if (A) = 97_H and if Cy flag = 0, let us see what happens when RAR is executed.



After executing RAR instruction, the contents of accumulator and the status of the Cy flag are as shown below.



Contents of the accumulator become $4B_H$ and Cy = 1.

4.6 SPECIAL INSTRUCTIONS

a) Decimal Adjust Accumulator Instruction

The instruction **DAA** is used to produce a correct BCD result when BCD addition is performed. Refer Sec 1.4 for BCD addition.

The auxiliary carry is used to get the correct BCD result. As already mentioned, the auxiliary carry is used internally by the processor and not available for the user to use it in writing programs. A simple example will clarify the DAA instruction. Let us add two BCD numbers 05 and 07.

```
MVI  A,05        ; (A) = 05
MVI  B,07        ; (B) = 07
ADD  B           ; (A) = 0C_H , not a correct BCD result
DAA              ; (A) = 12, correct BCD result
```

However, we should remember that **DAA instruction is used only after an addition or increment operation which affect the auxiliary flag. Also, DAA instruction is applicable only for the data in the accumulator.**

That is,

```
        MVI     A,0CH
        DAA
```

will not produce the desired result of converting $0C_H$ to 12.

b) Complement Accumulator

The instruction **CMA** complements each bit in the accumulator. Flags are not affected.

For example if

$(A) = 26_H = 0010\ 0110$, then the

instruction CMA makes,

$(A) = D9_H = 1101\ 1001$

c) Set Carry

The instruction **STC** sets the carry flag to 1. No other flags are affected.

d) Complement Carry

The instruction **CMC** complements the carry flag. No other flags are affected.

5.1 DATA TRANSFER INSTRUCTIONS– II

We were introduced to a number of data transfer instructions like *move, move immediate, store accumulator, load accumulator* etc in Chapter 4. Those instructions are just sufficient for simple data manipulations. Having understood some basic instructions, now, we are ready to learn some slightly difficult data transfer instructions.

As mentioned in the previous chapter, data transfer instructions do not affect any of the flags.

a) Store Accumulator Indirect

We have already seen an instruction, STA Addr 16, which means, to store the contents of accumulator in a memory location whose address is directly given in the instruction itself. The accumulator contents can also be stored in a memory location whose address is indirectly given in register pairs BC or DE.

The instruction **STAX B** , stores the contents of the accumulator in a memory location whose address is in BC register pair.

$$((BC)) \leftarrow (A)$$

For example, if (A) = 45_H and (BC) = 2050_H, then the instruction STAX B, transfers the data 45_H in the accumulator to the memory location whose address is 2050_H.

Similarly, **STAX D** , stores the contents of the accumulator in a memory location whose address is in DE register pair.

((DE)) ← (A)

There is no instruction like STAX H. To store the contents of accumulator in memory whose address is in HL pair; we have a different instruction MOV M,A.

For example, to store the contents of accumulator in a memory location whose address is 2050$_H$, we can follow any one of the following four instructions.

i. STA 2050$_H$

ii. LXI B, 2050$_H$ followed by STAX B

iii. LXI D, 2050$_H$ followed by STAX D

iv. LXI H, 2050$_H$ followed by MOV M,A

b) Load Accumulator Indirect

We have an instruction, LDA Addr 16, which means to load the accumulator with the contents of a memory location whose address is directly given in the instruction itself. The accumulator can also be loaded from a memory location whose address is indirectly given in register pairs BC or DE.

The instruction **LDAX B** , loads the accumulator with the contents of a memory location whose address is in BC register pair.

(A) ← ((BC))

Similarly, **LDAX D** , loads the accumulator with the contents of a memory location whose address is in DE register pair.

(A) ← ((DE))

For example, to load the accumulator with the contents of a memory location whose ddress is 2050H, we can follow any one of the four instructions.

i. LDA 2050_H

ii. LXI B, 2050_H followed by LDAX B

iii. LXI D, 2050_H followed by LDAX D

iv. LXI H, 2050_H followed by MOV A,M

Example 5.1:

Write a program to exchange the contents of memory locations 2050_H and 2370_H without using STA.. and LDA.. instructions.

Solution:

```
LXI   B,2050H      ; Load BC pair with 2050H
LXI   D,2370H      ; Load DE pair with 2370H
LDAX  B            ; Load A with memory whose addr is 2050H
MOV   L,A          ; Save the first number in L
LDAX  D            ; Load A with memory whose addr is 2370H
STAX  B            ; Move (A) to memory whose addr is 2050H
MOV   A,L          ; Move (L) to A
STAX  D            ; Move (A) to memory whose addr is 2370H
HLT                ; End of program
```

c) Store H and L Direct

The instruction **SHLD Addr 16** is used to store the contents of H register and L register in two successive memory locations. The contents of low register L is moved to a memory having a lower address and the contents of high register H is moved to a memory having a higher address.

(Addr) ← (L) , (Addr + 1) ← (H)

For example, the instruction SHLD 2050_H, copies the L register into a memory location 2050_H and the H register into the memory location 2051_H. It is a three byte instruction and would be coded in three successive memory locations as 22, 50, 20.

d) Load H and L Direct:

The instruction **LHLD Addr16** performs the reverse operation to that of Store H and L direct. LHLD Addr16 loads the L register with data from the address given in the instruction and loads the H register with the contents of the following address.

(L) ← (Addr) , (H) ← (Addr + 1)

For example, the instruction LHLD 2060_H loads the L register with data from the address 2060_H (given in the instruction) and loads the H register with the contents of the following address, 2061_H.

Example 5.2:

(Same as Example 4.3)

Write a program to add two 16-bit numbers 1234_H and $56FF_H$ and store the 16-bit result in two successive memory locations 2050_H and 2051_H.

Solution:

```
LXI  B,1234H     ; Load BC with 1234H
LXI  H,56FFH     ; Load HL with 56FFH
DAD  B           ; Add (BC) and (HL) and put the result in HL
SHLD 2050H       ; Store (L) in 2050H and (H) in 2051H
HLT              ; End of program
```

e) Exchange the register pairs HL and DE:

The instruction **XCHG** exchanges the contents of HL pair with DE pair.

(HL) ⟷ (DE)

That is, if (HL) = 1234_H and (DE) = $ABCD_H$ initially, then the XCHG instruction makes, (HL) = $ABCD_H$ and (DE) = 1234_H.

e) Copy H and L registers to the Stack Pointer

The instruction **SPHL** loads the contents of the H and L registers into the Stack Pointer register.

(SP) ← (HL)

f) Exchange Stack-top with H and L

The instruction **XTHL** exchanges the contents of H and L registers with the contents of memory locations pointed to by SP and SP + 1.

The contents of L register are exchanged with the contents of memory location pointed to by the stack pointer (SP) and the contents of H register are exchanged with the contents of the memory location (next stack location) pointed to by stack pointer plus one (SP) + 1.

(L) ← ((SP))

(H) ← ((SP) + 1)

Let (H) = 45_H , (L) = 67_H and (SP) = $27F5_H$

Let the contents of memory locations pointed to by SP and SP + 1 be 22_H and 33_H respectively.

i.e., ((SP)) = ($27F5_H$) = 22_H

((SP) + 1) = ($27F6_H$) = 33_H

When the instruction XTHL is executed, the HL register contents and memory contents change. But the SP contents do not change.

i.e., (H) = 33_H

(L) = 22_H

((SP)) = ($27F5_H$) = 67_H

((SP) + 1) = ($27F6_H$) = 45_H

5.2 BRANCH INSTRUCTIONS

We have mentioned earlier, that the machine codes of the instructions of a program are stored in successive memory locations. The program counter of 8085 sends out the address of the memory location for a byte to be read or written into. In the mean time, the program counter is automatically incremented and is ready with the address of the next memory location. On certain occasions, the normal sequence has to be altered by changing the contents of the program counter. A number of instructions like jump, call, restart, move HL to PC etc., are available. Let us discuss them one by one.

a) Jump Instruction

The jump instruction is of two types, namely

i) Unconditional jump and

ii) Conditional jump

i. Unconditional jump

The unconditional jump instruction is of the form **JMP Addr 16**. This is a 3-byte instruction with the first byte containing the opcode (machine code) of the instruction and bytes 2 and 3 containing the address. When loaded in memory, the lower order address is loaded first, followed by higher order byte.

When the instruction JMP 2250_H is executed, the address 2250_H is loaded into the program counter. The 8085 will now pick up its next instruction from the memory at this new address and continues to execute the instructions sequentially from this address.

The instruction JMP 2250_H is loaded in the memory at address 2000_H as,

2000	C3
2001	50
2002	22

he jump is unconditional and does not depend on the status of the flags.

While writing programs, the jump instruction can be followed by a label instead of the tual address. That is, instead of writing

JMP 2250_H, we can write

JMP AHEAD or JMP START or JMP LOOP1 etc.,

where AHEAD or START or LOOP1 are the labels used. While loading the memory with machine codes, the label is converted into address and entered.

ii. Conditional Jump:

Conditional jump instructions such as **JZ** (jump on zero) or **JC** (jump on carry) will cause the program counter to be loaded with the 16-bit address given in the instruction only if the specified condition is true (Z = 1 or Cy = 1). Otherwise, the execution of instructions will continue in its normal sequence.

The various conditional jump instructions are listed below.

JZ	jump if zero flag is set.	Z = 1.	
JNZ	jump if zero flag is not set.	Z = 0.	
JC	jump if carry flag is set.	Cy = 1.	
JNC	jump if carry flag is not set.	Cy = 0.	
JPE	jump if parity flag is set.	P = 1.	Even parity.
JPO	jump if parity flag is not set.	P = 0.	Odd parity.
JM	jump if sign flag is set.	S = 1.	MSB bit = 1.
JP	jump if sign flag is not set.	S = 0.	MSB bit = 0.

Let us take one or two simple programs to understand the conditional jump instructions and to show how they are coded and loaded in memory.

Example 5.3:

Write a program to add two bytes and store the result in a memory location 2050_H. Also store the carry as 1 if there is a carry, in the next memory location. If there is no carry, store a 0 in the next location.

Solution:

```
          MVI C,00    ; C = 0,to indicate no carry
          MVI A,X     ; First number in A
          MVI B,Y     ; Second number in B
          ADD B       ; Add the numbers
Jmp,      JNC AHEAD   ; No carry, jump to store zero
```

```
        INR  C        ; If there is a carry, C = 1
AHEAD:  STA 2050      ; Store sum at 2050H
        MOV A,C       ; Bring carry status to accumulator
        STA 2051      ; Store carry at 2051H
        HLT           ; End of program
```

The same program with memory address, hex code and mnemonics is shown below.

Mem.Addr	Hex.code	Label	Opcode	Operands
2000	0E		MVI	C,00
2001	00			
2002	3E		MVI	A,X
2003	X			
2004	06		MVI	B,Y
2005	Y			
2006	80		ADD	B
2007	D2		JNC	AHEAD
2008	0B			
2009	20			
200A	0C		INR	C
200B	32	AHEAD:	STA	2050
200C	50			
200D	20			
200E	79		MOV	A,C
200F	32		STA	2051
2010	51			
2011	20			
2012	76		HLT	

In this program, AHEAD is the label of the address at which the instruction STA 2050_H stored. X and Y are 8-bit data to be entered in the memory.

Let us take one more example to understand the conditional jumps.

Example 5.4:

Write a program to transfer hundred bytes stored in one block of memory with starting address 2100_H to another block of memory with starting address 2400_H.

Solution:

It is a program for 'block transfer'. Hundred bytes are to be transferred and so we need a counter. Let us use C register as counter and load the C register with a count equal to 100 in decimal which is also equal to 64_H. We will transfer a byte from the first location in the first memory block to the first location in the second memory block using data transfer instructions, that we have already seen. After each transfer, the count in C register is decremented by one. The transfer is repeated for the succeeding memory locations till the count in C register becomes zero.

```
     LXI H,2100H   ; Starting address of first block in HL
     LXI D,2400H   ; Starting address of second block in DE
     MVI C,64H     ; Count in C = 64H
DO:  MOV  A,M      ; Take a byte from memory addressed by
                   ; HL to A
     STAX D        ; Transfer the byte in A to memory
                   ; addressed by DE
     INX H         ; Increment the address in HL
     INX D         ; Increment the address in DE
     DCR C         ; Decrement count by 1
     JNZ DO        ; If (C)is not zero, continue in DO
     HLT           ; If (C) is equal to zero, end task
```

Here, DO is the label of the address at which the instruction MOV A,M is stored.

b) Call and Return instructions

The **CALL** and **RET** instructions are discussed under branch instructions because these instructions also cause branching from regular sequence. First, let us get used to the words **'subroutines'** or subprograms.

While writing a program, let us assume that a group of instructions are repeatedly used a number of times at different points in the program. If we use this approach, the machine codes of the instructions are entered repeatedly and the memory space required will be large. This problem is solved in a very elegant way. The group of instructions which are to be repeatedly used are stored in a separate memory block, only once. This group of instructions are now called subprograms or subroutines. From different points from the main program, the subroutine is 'called'. Now the processor branches to execute the instructions in the subroutine. At the end of the subroutine, the processor has to 'return' back to the main program. While returning back to the main program, the processor has to come back to the point from where it has branched out. The procedure is achieved using the CALL and RET instructions with the help of stack operations. The subroutine can be called any number of times from the main program. This is shown in Fig (5.1).

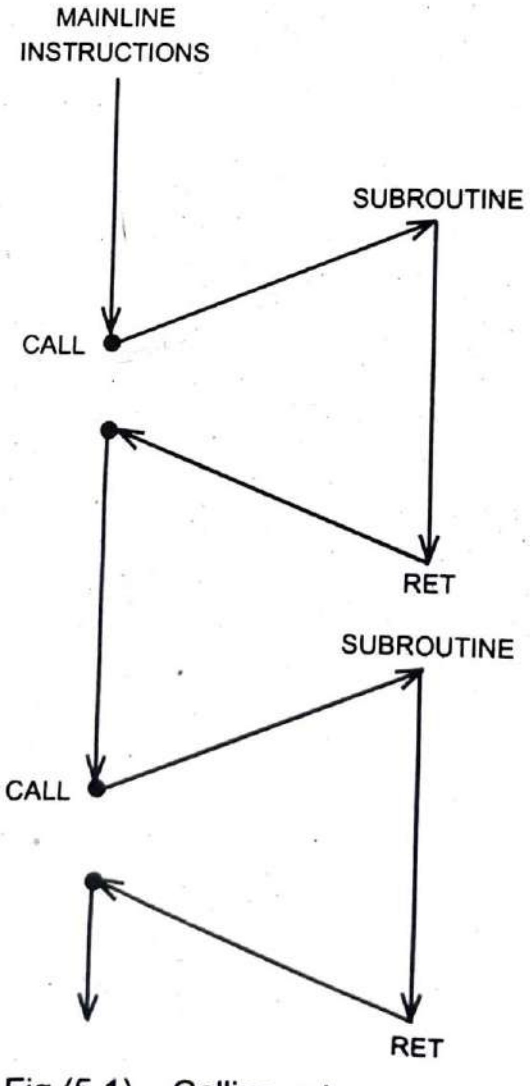


Fig (5.1) Calling subroutines

The subroutines can also be nested as shown in Fig (5.2). This means, one subroutine calls another subroutine.

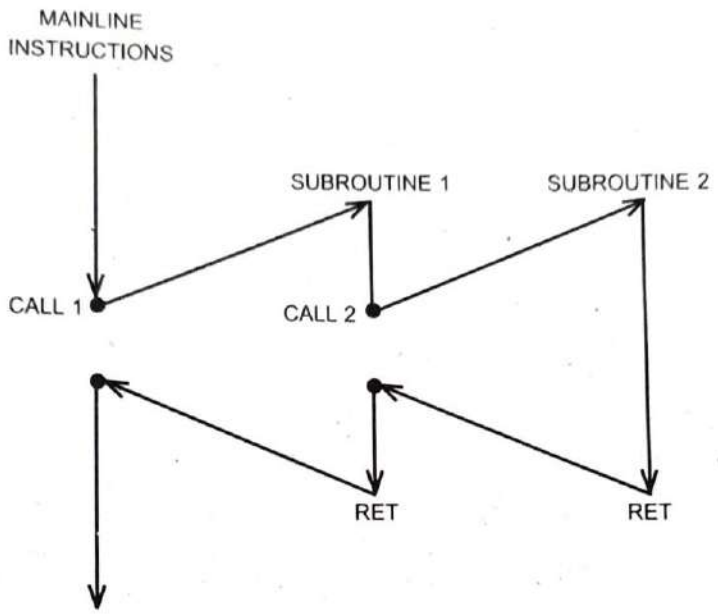


Fig (5.2) Nested Subroutines

But, we must be carefuı to have the RET instruction at the end of each subroutine. All return addresses are stored in the stack.

The call and return instructions are of two types, namely

i) Unconditional Call and Return.

ii) Conditional Call and Return.

i. Unconditional Call and Return instructions

The unconditional call instruction is of the form,

CALL Addr 16

This is a three byte instruction. The first byte is the Hex code (CD) for the CALL instruction. The second and the third byte gives the 16-bit starting address of the subroutine. When the processor encounters the CALL instruction, the program counter is modified to contain the starting address of the subroutine. Now, the instructions under the subroutine are executed sequentially. At the end of the subroutine, to return back to the main program, the **RET** instruction is given.

We can see that, when we want to go to a subprogram, we give the address of the subprogram. But, when we want to return to the main line program, we just give the RET instruction and not the address of the instruction in the main program. This is made possible in all microprocessors by making use of the stack. Before modifying the contents of the program counter with the address of the subroutine, the address of the next instruction in the main program is saved in the stack. Stack and related instructions are discussed in Sec 5.3. However, let us see how stack is used during CALL and RET instructions with an example.

In the programmer's model of 8085, we have taken the RAM area from 2000_H to $27FF_H$. The last few locations can be used for stack related operations. The first step is to initialize the top of the stack with an instruction LXI SP, Addr 16. Let us choose the address as $27F0_H$ for stack top. We initialize the stack top with the instruction,

LXI SP,$27F0_H$

Let us call a subroutine entered at location 2050_H from the main memory location 2010_H. The instruction CALL 2050_H is coded as CD, 50, 20 and entered at locations 2010_H, 2011_H, 2012_H. Therefore the address of the next instruction after the CALL instruction is 2013_H. This address is referred to as the 'return address'. This address is stored in two locations in the stack. The stack pointer (SP) is decremented by 1 to $27EF_H$ and the high order return address 20_H is stored here. The stack pointer is decremented once again to $27EE_H$ and the low order return address 13_H is stored here. The new value in the stack pointer is $27EE_H$. This is shown in Fig (5.3).

Main program

Address	Hex code
2010	CD
2011	50
2012	20
2013	next instruction

Stack

	address	Hex code
	27ED	xx
Top of stack :	27EE	13
	27EF	20
	27F0	xx

Fig (5.3)

The sequence of operations during the CALL instruction can be represented as follows.

((SP) – 1) ← (PC high)

((SP) – 2) ← (PC low)

(SP) ← (SP) - 2

(PC) ← call address

While writing programs, the CALL instruction can be followed by a label instead of the actual address. That is, instead of writing

CALL 2250_H, we can write

CALL DELAY or CALL MULTI or CALL BCD

where DELAY, MULTI and BCD are the labels used. While loading the memory with machine codes, the label is converted into address and entered.

At the end of the subroutine, the RET instruction is given. We have seen that now the stack top is $27EE_H$. The return address 2013_H is safely at $27EE_H$ and $27EF_H$. The contents of memory pointed by SP is transferred as a lower byte to the PC. The SP is incremented by 1 and the contents of the memory location pointed to by SP + 1 is transferred to as a higher byte to the program counter. The stack pointer is incremented once again so that SP + 2 is the new value for the stack pointer.

The sequence of operations during the RET instruction can be represented as follows.

(PC low)	←	((SP))
(PC high)	←	((SP) + 1)
(SP)	←	(SP) + 2

ii. Conditional Call and Return

Conditional call instructions such as **CZ** (call on zero) or **CC** (call on carry) will cause the program counter to be loaded with the 16-bit address given in the instruction only if the specified condition is true (Z = 1 or Cy = 1). Otherwise, the execution of instructions will continue in its normal sequence.

The various conditional call instructions are listed below.

CZ	call if zero flag is set.	Z = 1.	
CNZ	call if zero flag is not set.	Z = 0.	
CC	call if carry flag is set.	Cy = 1.	
CNC	call if carry flag is not set.	Cy = 0.	
CPE	call if parity flag is set.	P = 1.	Even parity.
CPO	call if parity flag is not set.	P = 0.	Odd parity.
CM	call if sign flag is set.	S = 1.	MSB bit = 1.
CP	call if sign flag is not set.	S = 0.	MSB bit = 0.

Return to the main program can also be conditional. Conditional return instructions such as **RZ** (return on zero) or **RC** (return on carry) will cause the program counter to be loaded with the 16-bit address from the stack only if the specified condition is true (Z = 1 or Cy = 1). Otherwise, the execution of instructions will continue in its normal sequence in the subroutine.

The various conditional return instructions are listed below.

RZ	return if zero flag is set.	Z = 1.	
RNZ	return if zero flag is not set.	Z = 0.	
RC	return if carry flag is set.	Cy = 1.	
RNC	return if carry flag is not set.	Cy = 0.	
RPE	return if parity flag is set.	P = 1.	Even parity.
RPO	return if parity flag is not set.	P = 0.	Odd parity.
RM	return if sign flag is set.	S = 1.	MSB bit = 1.
RP	return if sign flag is not set.	S = 0.	MSB bit = 0.

c) Restart Instructions:

Restart instructions **(RST n)** are special one byte unconditional call instructions. As in the case of call instruction, the content of the program counter (corresponding to the return address) is saved in the stack. Then the program jumps to the instruction starting at the restart location. There are eight restart instructions, RST 0 to RST 7. That is, the contents of the program counter is changed to one of the restart addresses.

The eight restart instructions and the corresponding addresses to which the control is transferred are given below.

Instruction	Addr to which control is transferred
RST 0	0000_H
RST 1	0008_H
RST 2	0010_H
RST 3	0018_H
RST 4	0020_H
RST 5	0028_H
RST 6	0030_H
RST 7	0038_H

The address calculation is simple. For RST n, the address is 8 x n in Hex. For example. for RST 5, the restart address is 5 x 8 = 40 in decimal which is equal to 28_H.

The restart instructions are mostly used with **interrupts**, discussed in Chapter 10.

The instruction **PCHL** moves the contents of HL register pair to the program counter

(PC high) ← (H)

(PC low) ← (L)

Following the execution of this instruction, program execution jumps to the address stored in HL register pair. Therefore, this instruction is also a type of branch instruction.

5.3 STACK AND STACK RELATED INSTRUCTIONS

A small area of the read/write memory RAM can be called as stack. When we write programs, the registers may have to be used again and again. The content of a register has to be saved for future use before using the same register in some other part of the program. The memory connected to 8085 has thousands of locations and the last few locations of the memory can be used to save the register contents. Also, when the processor executes a CALL instruction and during interrupts, the processor needs a few memory locations to store the 'return address'. For all these requirements, we can set aside a small portion of RAM, which is called as stack. Since, only the tail end of RAM is used for stack operations, it will not clash with the area where the programs are entered.

In the programmer's model of 8085, we have taken the RAM to have an address range of 2000_H to $27FF_H$. The machine codes of the instructions of the programs are entered in successive locations from 2000_H onwards. With each entry of machine code in the memory, the memory address goes on increasing.

We can choose the very last location, $27FF_H$ and a few locations below that for the stack operations. This area is used to save the register contents and retrieve them when needed. The same area is used to save the 'return address' during call instruction and during interrupts.

First, the stack is initialized with the instruction LXI SP, addr 16. Let us initialize the stack with the instruction,

LXI SP, 2800_H

Though the very last RAM location in our example is $27FF_H$, we have used the address as 2800_H. It is because, the processor decrements the stack pointer once, before saving the register content at that address. Therefore, the first byte will be stored at address $27FF_H$.

To save the registers **PUSH** instruction is used. To retrieve the contents **POP** instruction is used.

PUSH Instruction:

The PUSH instruction is used to push or store a register pair on stack. For example, the instruction, PUSH B pushes the contents of BC register pair on the stack. The sequence of operations for PUSH B is,

i) The SP is decremented once and the contents of B register is stored in the memory pointed to by (SP) – 1.

ii) The SP is decremented once again and the contents of C register is stored in the memory location pointed to by (SP) – 2.

iii) The (SP) – 2 becomes the new value of SP.

Let the initial values in BC, SP and stack be as shown in Fig (5.4)

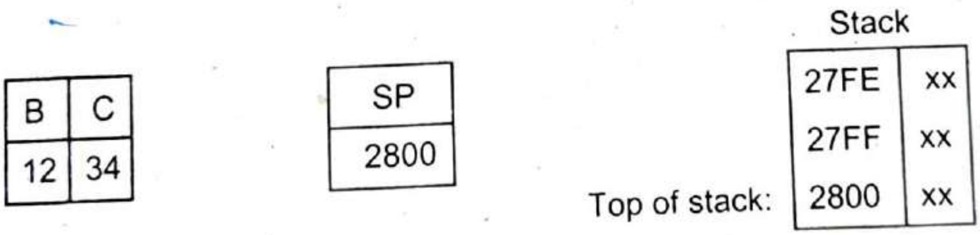


Fig (5.4)

Now, if the instruction PUSH B is executed, the contents of BC register pair is saved in the stack and the value in SP is decremented by 2 as shown in Fig (5.5).

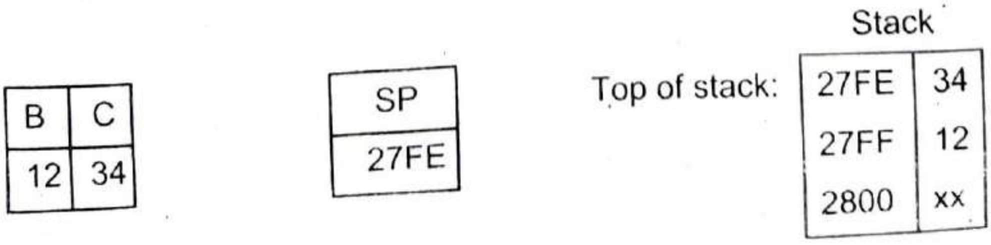


Fig (5.5)

The sequence of operations for PUSH B can represented as,

((SP) – 1)	←	(B)
((SP) – 2)	←	(C)
(SP)	←	(SP) – 2

Using the same procedure, we can push other register on to the stack.

PUSH D pushes registers D and E.

PUSH H pushes registers H and L.

PUSH PSW pushes A register and Flag register.

PSW stands for Processor Status Word or Program Status Word and represents the contents of the accumulator and flag register. The accumulator is taken as the high register and the flag register is taken as the low register for stack instructions.

Let us initialize the stack pointer and push all the registers. The sequence of instructions and how the data are stored in the stack is shown below in Fig (5.6).

Stack

LXI SP.2800	Top of stack :	27F8	(Flags)
PUSH B		27F9	(A)
PUSH D		27FA	(L)
PUSH H		27FB	(H)
PUSH PSW		27FC	(E)
		27FD	(D)
		27FE	(C)
		27FF	(B)
		2800	xx

Fig (5.6)

By looking at Fig (5.6), we can see that, every time a push operation is performed, 'stack grows' but in the 'downward' direction (as the address of the memory goes on decreasing).

POP Instruction:

The POP instruction is used to transfer two bytes from the top of the stack to the register pair specified in the instruction. For example, POP B transfers the contents of memory pointed by SP to C register and the contents of memory pointed by SP + 1 to B register. Let the initial values in BC, SP and stack be as shown in Fig (5.7).

B	C
xx	xx

SP
27F6

Stack

Top of stack:	27F6	89
	27F7	67
	27F8	xx

Fig (5.7)

Now, if the instruction POP B is executed, the contents of BC register pair is retrieved from the stack and the value in SP is incremented by 2 as shown in Fig (5.8)

B	C
67	89

SP
27F8

Stack

	27F6	89
	27F7	67
Top of stack:	27F8	xx

Fig (5.8)

The sequence of operation for POP B can be represented as,

$$(C) \leftarrow ((SP))$$

$$(B) \leftarrow ((SP) + 1)$$

$$(SP) \leftarrow (SP) + 2$$

The instruction POP D retrieves the contents of DE.

The instruction POP H retrieves the contents of HL.

The instruction POP PSW retrieves the contents of accumulator and flags.

The registers are saved and retrieved on a **Last In First Out (LIFO)** basis. This means the register pair that is pushed in last must be popped out first.

Let us assume that we want to call a subroutine from the main program. We can save all the registers in the beginning of the subroutine using PUSH instructions and then include the instructions of the subroutine. The registers can be retrieved using POP instructions before returning to the main program. This is shown below.

```
Main Program              Subroutine
..................        PUSH   B
..................        PUSH   D
..................        PUSH   H
CALL  subroutine          PUSH   PSW
..................        ..................
..................        Instructions
..................        ..................
                          POP   PSW
                          POP   H
                          POP   D
                          POP   B
                          RET
```

As seen earlier (Section 5.2.b), stack is used during CALL and RET instructions. When the microprocessor is interrupted using one of the interrupt pins (INTR, RST 5.5, RST 6.5, RST 7.5 and TRAP), the processor branches to execute a subroutine which is called as an Interrupt Service Routine (ISR). During this process also, the stack is used to save the 'return address'. Interrupts are discussed in Chapter 10.

In our discussion, we have initialized the SP with the very last location plus one. This is done so that not even single memory location is wasted. But the stack can be initialized to be any where in the RAM, as decided by the user.

Example 5.4:

Specify the register contents as the following instructions are executed one after another.

```
LXI    SP,2750H
LXI    B,1234H
LXI    D,5678H
PUSH B
PUSH D
POP    B
POP    D
```

Solution:

```
LXI     SP,2750H      ; Load SP with 2750H
LXI     B, 1234H      ; Load BC with 1234H
LXI     D, 5678H      ; Load DE with 5678H
PUSH     B            ; Save (BC) = 1234H in stack
PUSH     D            ; Save (DE) = 5678H in stack
POP      B            ; Retrieve 5678H from top of stack and
                      ; put it in BC
POP     D             ; Retrieve 1234H from top of stack and
                      ; put it in DE
```

This program has actually exchanged the contents of register pairs BC and DE.

5.4 I/O AND MACHINE CONTROL INSTRUCTIONS

a) I/O Instructions

To communicate with the outside world, the processor uses input ports and output ports. The 8085 uses **IN** instruction to input a byte from an input port to the accumulator. An **OUT** instruction is used to output a byte from the accumulator to the output port. These two instructions will be used in the I/O Interface, which is dealt in Chapter 9.

b) Machine Control Instructions

Machine control instructions only control the processor operations and do not perform any operation on data. They are all single byte instructions.

i. Halt Instruction

The **HLT** instruction stops the microprocessor and no further instructions are executed. The registers and flags are not affected. The address and data buses are placed in high impedance state.

The processor can come out of the halt state only by resetting the processor or by applying interrupt.

ii. No operation Instruction

The instruction **NOP** performs no operation. The processor fetches the Hex code of the instruction NOP from the memory and decodes the instruction but no operation is executed. The registers and flags are not affected.

NOP instructions can be introduced in delay loops to increase the T-states. (Refer Chapter 7). Also, NOP instructions can be introduced as dummy instructions, so that if we have to insert an instruction, it can be inserted in the place of NOP instruction. Adding or removing NOP instruction does not affect the program.

iii. Interrupt Related Instructions

There are four Interrupt related instructions in 8085.

a) **EI** – Enable Interrupts

b) **DI** – Disable Interrupts

c) **SIM** – Set Interrupt Mask

d) **RIM** – Read Interrupt Mask

ll these instructions are discussed in Chapter 10, where 8085 interrupts are introduced.

5.4 THE 8085 ADDRESSING MODES

We have seen that an assembly language instruction is made up of an operation code and operand/s. When 8085 executes an instruction, it performs a specified function on data (called operand/s). These data may be part of an instruction, reside in one of the internal registers or stored at an address in memory or I/O. The various formats of specifying the operands are called the addressing modes. The 8085 uses the following addressing modes:

i) Direct addressing

ii) Register addressing

iii) Register indirect addressing

iv) Immediate addressing

v) Implied addressing / Implicit addressing

i. Direct Addressing

In direct addressing mode, the address of the operand (data) is given in the instruction itself.

Examples:

1. STA 2050_H

In this instruction, 2050_H is the memory address where the data is to be stored, the address is given in the instruction itself.

2. IN FF_H

In this instruction, FF_H is the address of the input port from where data is brought in to the accumulator.

ii. Register Addressing

In register addressing mode, the operands are general purpose registers.

Examples:

1. MOV B,C

Both the source and destination operands are specified in the instruction.

2. ADD B

Here, the register B is the source operand.

iii. Register Indirect Addressing

In this addressing mode, the address of the operand is not specified directly but given indirectly in a register pair.

Example:

1. MOV A,M

This instruction moves a data from the memory whose address is in HL register pair to accumulator.

2. STAX B

This instruction moves the content of the accumulator to memory whose address is given in BC register pair.

iv. Immediate addressing

In this mode, an 8-bit or 16-bit data is specified as part of the instruction. When entered in memory, the data is entered immediately following the hexcode of the instruction.

1. MVI A, 25_H

The immediate data here is 25_H.

2. LXI H, 2050_H

In this instruction the immediate 16-bit data is 2050_H.

v. Implied Addressing / Implicit Addressing

In this mode, the instructions operate on the contents of the accumulator and has only the opcode field. (No operand field)

Example:

1. RLC

This instruction rotates the contents of the accumulator left by one bit position without carry. Here the register used is implied as accumulator.

2. CMA

This instruction complements the accumulator.

Practice Questions:

1. What is the function of the MOV instruction in the 8085 microprocessor?
2. Explain the difference between MVI and MOV instructions in 8085.
3. What does the ADD instruction do in the 8085 microprocessor?
4. What is the function of the SUB instruction in the 8085 microprocessor?
5. Explain the purpose of the ANA instruction in the 8085 microprocessor.
6. What is the role of JMP instruction in the 8085 microprocessor?
7. What does the JC instruction do in the 8085 microprocessor?
8. What is the significance of the CALL instruction in the 8085 microprocessor?
9. What does the POP instruction do in the context of stack operations?
10. What is the purpose of the IN instruction in the 8085 microprocessor?

Additional Resources :

https://www.geeksforgeeks.org/addressing-modes-8085-microprocessor/

https://www.youtube.com/watch?v=RyMT7GznQUo

References:

1. Fundamentals of Microprocessor 8085-V.Vijayedran