Table of Contents

1.0 DATA MANIPULATION-

1.1 ARITHMETIC INSTRUCTIONS.

These are used to perform arithmetic and operations. The arithmetic instructions include *add, subtract, increment, decrement* and *shift*. Generally, the *add* and *subtract* instructions involve the A-register (the accumulator) and either another processor register or a memory location. The *increment* and *decrement* operation can generally be performed on various registers and locations through the use of different mnemonics. Almost all instructions in the arithmetic group affect the flag register.

1.1.1 Addition Instructions

(a) Addition (8-bits)

One of the byte to be added is always stored in the accumulator. The mnemonic used depends on the source of the byte to be added (i.e. the second operand)

i. If the second operand is in another register

The mnemonic used is:

ADD reg. e.g. ADD C – This causes the contents of the accumulator. The result is stored in the accumulator. Register C is unchanged.

(A)+ (C) - - - ▸ (A)

ii. If the second operand is specified, the mnemonic is

ADI data –the immediate data are added to the accumulator

e.g. ADI 48H –Causes 48H to be added to the contents of the accumulator.

The sum is stored in the accumulator.

(A)+48H - - - - ▸ (A)

iii. If the second operand is in memory (whose address is in the H-L register pair)

The mnemonic is **ADD M**

(A) + ((HL)) - - - - ▸ (A)

The contents of the Accumulator are added to the contents of memory location whose address is in HL register pair.

The sum is placed in the accumulator. In all cases, all the flags are affected accordingly.

(b) Addition (16-bit)

These instruction provided in the 8085 instruction set adds two 16-bits numbers simultaneously. One of the numbers is stored in the HL pair register. The other number comes from another register pair.

e.g **DAD B** Causes the 16-bit contents of the register pair BC to be added to the 16-bit contents of the register pair HL . The result is placed in the register pair HL. *Only the carry flag is affected.*

e.g.	DAD B	(HL) + (BC)----► (HL)
Others	DAD D	(HL) + (DE)----► (HL)
	D AD H	(HL) + (HL)----► (HL)

1.1.2 Subtraction Instruction

The subtraction instructions for the 8085 have the same format as the addition instructions. The contents of the location defined in the instruction is subtracted from the contents of the accumulator; with the result placed in the accumulator. All flags are affected, the carry-flag is now a borrow flag, being set if a borrow-out is required.

(i) If the second number is in another processor register

SUB reg e.g. SUB B -causes the contents of register B to be added to the contents of the accumulator. The result is placed in the accumulator.

(A) - (B)----► (A)

(ii) If the second number is specified

The mnemonics used is **SUI data** (8-bits)

(A) - data ----►(A) e.g. SUI 86H ----► 86H is subtracted from the contents of the accumulator. (A)-86H----► (A)

(iii)If the second number is in memory (whose address is in HL register pair).

SUB M ----► which causes the contents of memory location whose address is in HL register pair to be subtracted from the contents of the accumulator.

(A)- ((HL)) ----► (A)

1.1.3 Increment

The increment instructions simply add one to the contents of a location.

(i) If the location is a processor register

INR reg e.g. INR C -which causes the contents of register C to be increment by one (C)+1----► (C)

All flags are affected

(ii) If the location is a memory location(whose address is in HL pair registers)

INR M The contents of memory location whose address is in HL pair is incremented by one ((HL)) +1---► ((HL)) .

All flags affected.

(iii) If the location is a register pair

INX rp e.g. INX D - increments the 16-bit number id the DE register pair by one (DE) +1----►(DE)

No flag is affected.

1.1.4 Decrement Instructions

Decrement instructions in the 8085 are very similar to the increment instructions. The contents of the addressed location are decremented by one.

(i) Register

DCR reg e.g. DCR E

(E)-1----►(E)

(ii) Memory

DCR M

((HL))-1----►((HL))

All flags are affected in both cases

(iii) Register pair

e.g. DCX H

(HL)-1----► (HL)

No flag is affected.

1.1.5 Multi-precision Arithmetic –Carry Flag.

If more than 16-bit accuracy is required, there are no single instructions available. Instead, a number of instructions must be used. For example, the addition of two 24-bit numbers would require three separate 8-bit additions.

e.g.

Number 1	Byte3	Byte 2	Byte 1
Number 2	Byte3’	Byte 2’	Byte 1’
Sum	Byte3+3’+cy	Byte2+2’+cy	Byte 1+Byte 1’

It is necessary to add the carry bit (CY) when adding together the second and third pairs of bytes. The 8085 as well as most other such devices provides additional add and subtract instructions that use the carry, or borrow bit.

For the 8085, the instructions are

(i) **ADC reg.**

e.g. ADC B -causes the contents of register B and the carry flag to be added to the contents of the accumulator. (A)+ (B) + (CY) - - - - ▶ (A)

(ii) **ADC data** e.g. ADC 4AH

(A)+4AH+ (CY) - - - - ▶ (A)

(iii) **ADC M** (A) + ((HL)) + (CY) - - - - ▶ (A)

All flags are affected.

The multi-precision subtraction instructions for the 8085 are the SBB instructions

(i) SBB C (A)-(C)-(CY)- - - - ▶ (A)

(ii) SBB M (A)-((HL))-(CY) - - - - ▶ (A)

(iii) SBI data (A)-(data)-(CY)- - - - ▶ (A)

All flags are affected.

1.1.6 BCD Arithmetic

It is advantageous in some microprocessor applications to represent numbers in BCD form. Most microprocessors therefore provide instructions that allow BCD data to be manipulated.

The 8085 uses a two stage process for BCD arithmetic. First straight binary operation, such as addition is performed. This is followed by an instruction that adjust the resulting number into its proper BCD format. The special instruction is unique to BCD operations. It uses the carry flag, the auxiliary carry flag (AC), and the accumulator contents to make the required adjustments.

For the 8085, the instruction is

DAA (**D**ecimal **A**ccumulator **A**djust).

This instruction performs the following operations:

1. If the value of the least significant nibble of the accumulator is greater than 9, or if the AC-flag is set, six (6) is added to the accumulator contents.
2. If the value of the most significant nibble of the accumulator is now greater than 9 or if the CY flag is set, six (6) is added to the most significant nibble of the accumulator. All flags are affected by this addition.

Because the single register increment instruction (INR) are really binary additions instructions, they can be used for BCD data, provided that the incremented byte exists in, or is moved to the A register before **DAA** instruction occur.

1.2 LOGICAL OPERATIONS

In addition to the arithmetic and instructions, most microprocessors have a number of data manipulation instructions that are primarily intended to work with non-numerical data. These are logical instructions and are used to perform logical operations. *Flag register is affected.*

1.2.1 Logical AND

The logical AND instructions perform the bit-by-bit AND operation between the contents of the A register and;

(i) Immediate data

ANI data e.g. ANI 54H (A) AND 54H----► (A)

(ii) the contents of another processor register

ANA reg e.g. ANA B (A) AND (B)----► (A)

(iii) the contents of memory location pointed by HL pair

ANA M (A) AND ((HL))- - - - ▸ (A)

A typical application of this instruction is to test the state of a specific bit in a given byte. For example, to test whether bit 6 of A- register contents is a one or a zero.

ANI 40H is used (A) AND 40H- - - - ▸ (A)

(A) ≡ xxxxxxxx

40H ≡ 01000000 AND

0x000000

If bit 6 were a zero, the result of AND operation would be zero and the zero flag would be set. If this bit were a one, the result would be 01000000 which is non zero; hence the zero flag would be reset to zero.

CY and AC flags are always cleared for the AND instructions of the 8085.

1.2.2 Logical OR

The logical OR is simply a bit-by-bit OR operation between the accumulator and the addressed byte. The result is placed in the accumulator: all flags are affected, with both the CY and AC flags being cleared. The instructions are:

ORI data	(A) OR data - - - - ▸ (A)	
	e.g. ORI C3H	(A) OR C3H- - - - ▸ (A)
ORA reg.	e.g . ORA L	(A) OR (L)- - - - ▸ (A)
ORA M		(A OR ((HL)) - - - - ▸ (A)

Whereas the AND operation allowed testing of bits in a word, the OR operation allows the setting of selected bits without disturbing the others.

E.g. to set bit 3 ORI 08H

(A) ≡ xxxxxxxx

08H≡00001000 OR

xxxx1xxx

1.2.3 Exclusive-OR operation

Is carried out using EX-OR instructions just like the OR and AND operations (instruction), one of the operand must be in the accumulator. The instructions affect the flags with CY and AC flags being cleared by the instructions

XRI data e.g XRI 24H (A) EX-OR 24H ----► (A)

XRA reg e.g XRA D (A) EX-OR (D) ----► (A)

XRA M (A) EX-OR ((HL))----► (A)

Exclusive –OR operations are often used to detect errors in binary data transmission. The received word is returned to the transmitter.

If no transmission errors have occurred an exclusive-OR on the two words will yield all zeros. If an error occurred, one or more bits will be one after the EX-OR operations.

Many programmers use the Exclusive-OR instruction to clear the accumulator (XRA A). It is also commonly used to complement a number by doing an exclusive-OR with FFH ie.XRI FFH e.g.

(A) ≡55H

XRI FFH…….. 01010101

11111111 EX-OR

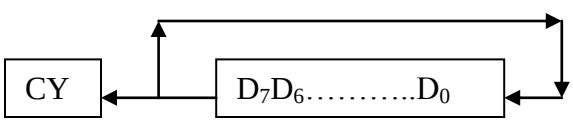
(A)=AAH ……. 10101010

1.2.4 Rotate /Shift

The three instruction types (AND, OR and EX-OR) perform bit by bit logical operation between two 8-bit words. In addition, the logical group contains instructions to move the bits within a byte (in the accumulator). The two primary forms of this operation are the rotate and the shift function. The 8085 does not provide a shift instruction; however, it does provide two forms of rotate in two directions.

(a) Rotate Left instructions

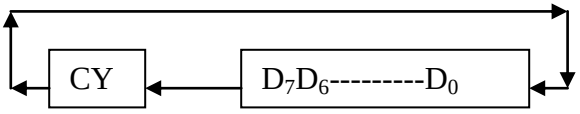
(i) **RLC** causes each bit in the accumulator to move to the next higher order location. The MSB is rotated back to the now vacant LSB position; it is also moved into the CY-flag bit



RLC

Only the CY flag is affected

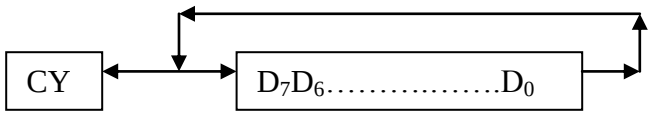
(ii) RAL



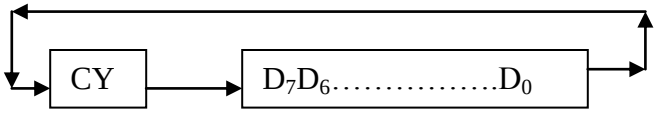
CY is included in the rotation path

(b) Rotate Right

(i) **RRC**- the LSB is rotated around to the most significant position and the CY bit. CY is not included in the rotation path as for RAR.



(ii) **RAL**



1.2.5 Compare Instructions

The compare instructions provide the keystone of computer decision-making operations. The compare instructions set the various flags as though a subtraction had occurred without actually completing the subtraction. The only states affected are the flags.

All 8085 compare instructions are performed using the accumulator with the other byte being in a processor register, memory or specified location

(i) If the other byte is in a processor register

CMP reg. e.g CMP B

The contents of the B-register are compared with the contents of the A-register

Z flag=1 if (B) = (A) [CPU performs (A)-(B)]

CY flag=1 if (B)> (A)

The other flags are also affected as though a subtraction had occurred.

(ii) If the other byte is specified

CPI data

e.g. CPI 56H

(ii) If the other byte is in memory (pointed by HL)

CMP M

1.2.6 Other Operations

The 8085 instructions set include three other operations under the logical group. None of the three affect any flags-other than the carry.

(i) **CMA** - This causes the bit-by bit complement of the accumulator with no flags affected.

(A) ----► ($\bar{A}$)

(ii) **CMC** --Complements the carry flag

(CY) ----► ($\overline{CY}$)

(iii) **STC** ----Set the carry flag

(CY) ----► 1

Whichever microprocessor is used, the programmer should study such miscellaneous instructions. They are often provided to take advantage of various processor characteristics or to overcome limitations.

Example

Q1 (a) Write 8085 assembly instruction to perform the following:

i. -Load 32H, F2H and EAH into registers A, D, and E respectively
ii. - Transfer the contents of register E to register C.
iii. -Store the contents of register A into locations 8400H
iv. -Decrement contents of register C
v. -Increment the content of register A
vi. -Add the contents of register A to that of register D
vii. -Store the result in memory location 8402H
viii. -Subtract the content of B from the contents of register C
ix. -Stop

(c) Hand-assemble the program in 1(a) and determine the memory capacity of the program in bits.

Solutions

```
(i)     MVI A, 32H,
        MVI D, F2H,
        MVI E, EAH
(ii)    MOV C,E
(iii)   STA 8400H
(iv)    DCR C
(v)     INR A
(vi)    ADD D
(vii)   STA 8402H
(viii)  MOV A,C,        ;since there  is no one instruction to perform this action, and also
        SUB B           ;since the subtrahend must be in register A, two instructions shall
                        ;be used.
(ix)    HLT
```

Q1 b Hand- assembling

	Mnemonics	**Machine codes**
(i)	MVI A, 32H,	3E32
	MVI D, F2H,	16F2
	MVI E, EAH	1EEA
(ii)	MOV C,E	4B
(iii)	STA 8400H	320084
(iv)	DCR C	0D
(v)	INR A	3C
(vi)	ADD D	82
(vii)	STA 8402H	320284
(viii)	MOV A,C,	79
	SUB B	90
(ix)	HLT	76

The memory capacity of the programs is 19 bytes which is equivalent to 19 X 8 bits = **152 bits**

2. TRANSFER OF CONTROL (BRANCH) INSTRUCTIONS

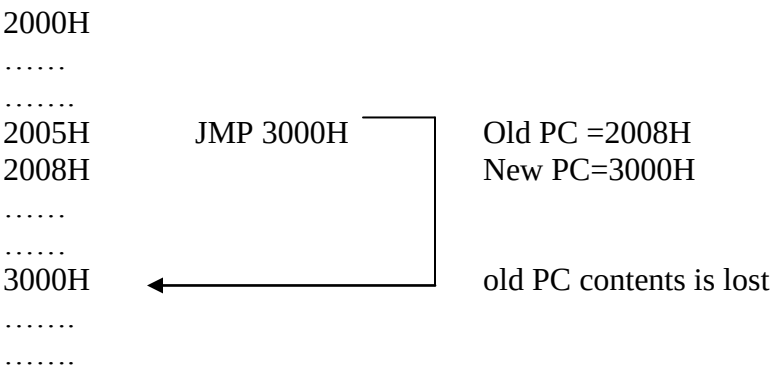
These instructions enable the CPU to transfer control of program execution from one program to another or from one section of a program to another section. The instructions include JUMPs, CALLs and RETurns.

2.1 Jumps

A jump instruction is used to break the normal sequential program execution. The PC usually steps one by one through instructions in the memory. When a jump instruction is encountered, execution is moved to another part of the program. This is done by replacing the two bytes in the PC with the address of the start of the new routine. The processor is thus forced to fetch the contents of this new location for its next instruction. The PC register is then incremented as usual through the new part of the program. This process is virtually universal for microprocessor program branching.

The new address for execution is included in the instruction statement for many transfer of control instructions.

e.g. JMP 4200H – causes the contents of the 8085's PC to be changed to 4200H. The old address in the **PC is lost**. The microprocessor fetches and executes the instruction at 4200H where program execution will continue.



2.1.1 Unconditional Jumps

JMP causes a branch to the new routine under all conditions when it is encountered. It is called an *unconditional jump* instruction.

2.1.2 Conditional Jumps

These causes the processor to branch to a new routine only if a specific condition is satisfied otherwise the PC is not changed and execution continues with no branch occurring. The conditions are determined by the state of the various flags (status bits) in the CPU's flag register. Each processor family has its own set of conditional jump parameters. *The state of these flags is determined by the operation that has occurred before the conditional jump instruction*. It is very important to note that the flags are changed only for certain instructions. The programmer must take care to verify that the flags are established before a conditional jump is executed.
The conditional branch instructions give the microprocessor its capacity for making decisions i.e. its ability to execute one or another instruction or program segment, depending on the previous results.

Conditional Jump Instructions.

Opcode	**Condition**	**Flag status**
JNZ	Not zero	Z=0
JZ	Zero	Z=1
JNC	No carry	C=0
JC	Carry	C=1
JPO	Parity odd	P=0
JPE	Parity even	P=1
JP	Plus	S=0
JM	Minus	S=1

2.2 Control Structures

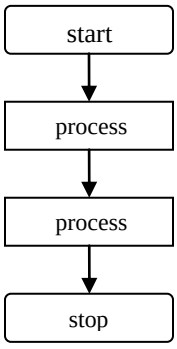
This describes the flow of a program. The basis of structured programming is a set of logical structures, each of which has only a single entry and a single exit point. The three basic control structures are *sequence, selection and repetition*.

By combining these basic structures, more complex logical structures are formed that also possess a single entry/single exit point. The advantage of this technique is in the development of programs that are more reliable and easier to understand, document and modify.

The syntax of some high level programming languages directly supports the implementation of the basic logic structures of structured programming. When programming in assembly language, however, it is more difficult to adhere strictly to the structured programming concepts because implementation of the basic logic structures necessitates programs that themselves require more instructions than their unstructured counterparts.

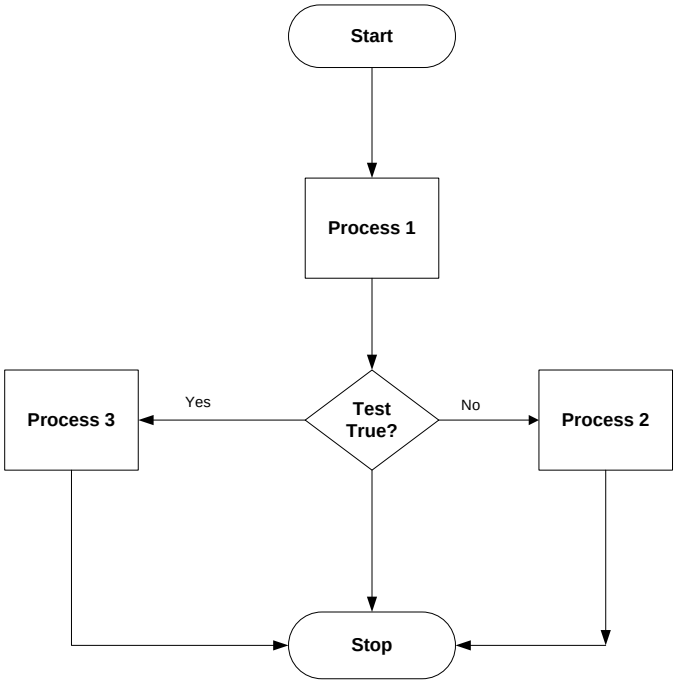
However, it is helpful to study these basic structures and how they can be implemented in assembly language. In practice, these logic structures and their variations are the foundation of program design.

(i) ***Sequence*** means that program statements are performed one after the other in order (unless another control structure takes precedence). This control structure is ensures that all instructions in the program are executed once.

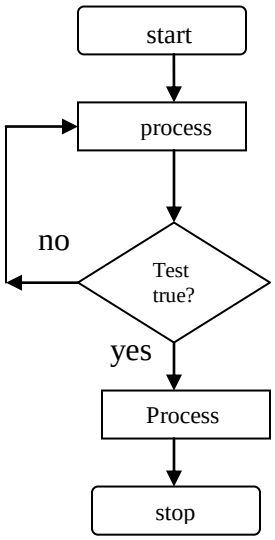


(ii) ***Selection*** means that one part or another of a program is executed, based on the outcome of a test for which part should be the one executed. Selection involves making a choice. This is sometimes called *branching* (choosing one branch or the other).

Branching can be done in most high level languages using the ***If- Then-Else*** and ***Select*** construct. This causes the statement(s) in < true block (process 1) > to be performed if < condition > is true; otherwise the statement(s) in < false block (process 2) > are performed.



(iii) ***Repetition*** means that a part of a program is to be executed over and over until a criterion for stopping the repetition is satisfied.



Repetition (*looping*) may be implemented in most high level programming languages using the ***Do… While***, ***repeat... until***, ***Do….Until*** constructs.

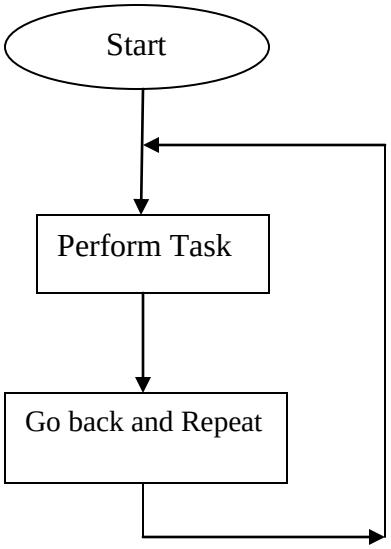
2.2.1 Looping

The programming technique used to instruct the microprocessor to repeat tasks is called ***looping***. Thus a loop in a program is a closed sequence of instructions whose execution is repeated until either a test or terminal state shows that a specified condition is satisfied, at which point the program branches or exits from the loop.

Loops are implemented using Jump instructions in assembly language for most microprocessors and can be classified into two groups.

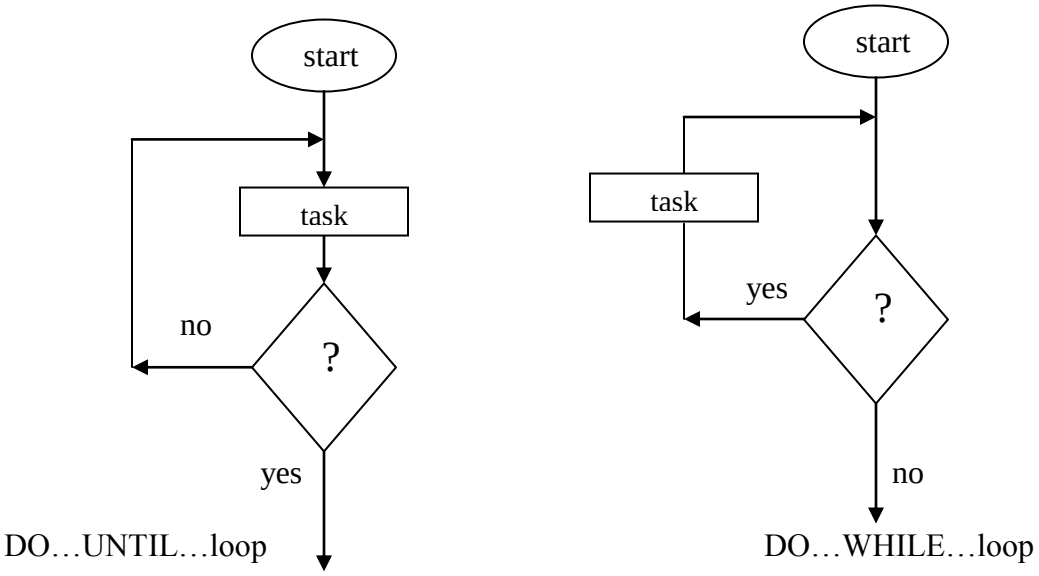
(i) Continuous loops

This is a set up by using the Unconditional jump instructions. A program with a continuous loop does not stop repeating the task until the system is reset.



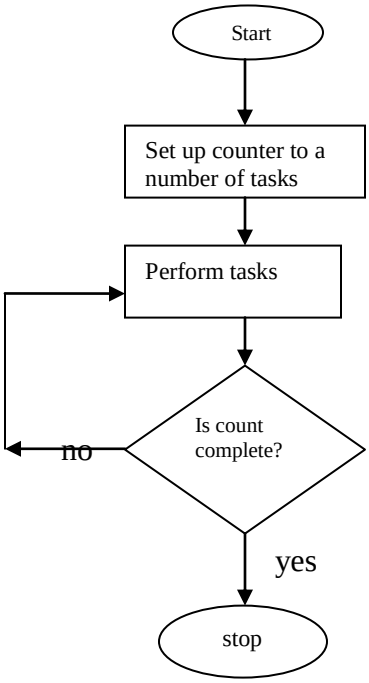
(ii) Conditional Loop

This is set up by the conditional Jump instructions and repeats a specified task until a set condition is satisfied. These loops usually include *counting and indexing*.



2.2.2 Counter

The microprocessor needs a counter to repeat certain tasks and when the counting is completed, it needs to examine a flag before exiting the loop. This is accomplished with the conditional loop.



It is easier to control down to zero than to count up because the zero flag is set when the register (counter) contents become zero. Counting up requires the use of the compare instructions.

2.2.3 Indexing

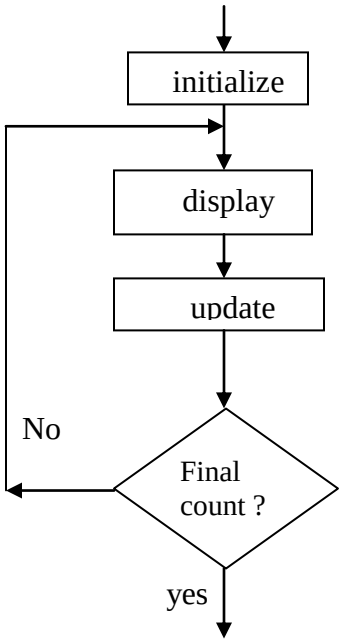
Another type of loop includes indexing along with a counter. Data bytes stored in memory locations can be referred to by their memory locations (addresses).

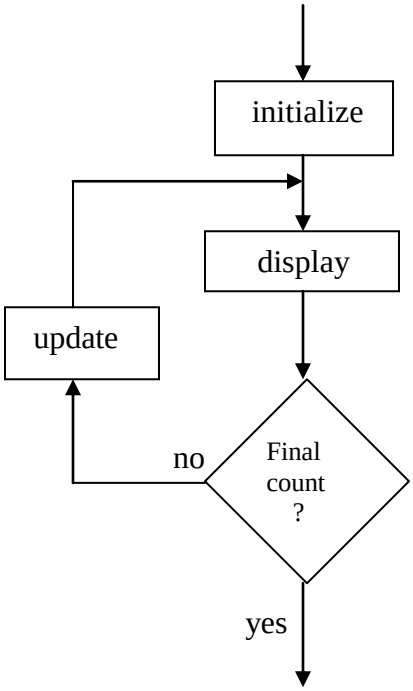
2.3 Applications of Jump Instructions

2.3.1 Counters

A counter is designed simply by loading an appropriate number into one of the registers and using the increment or decrement (INR or DCR) instructions.

A loop is established to update the count and each count is checked to determine whether it has reached the final number; if not, the loop is repeated.





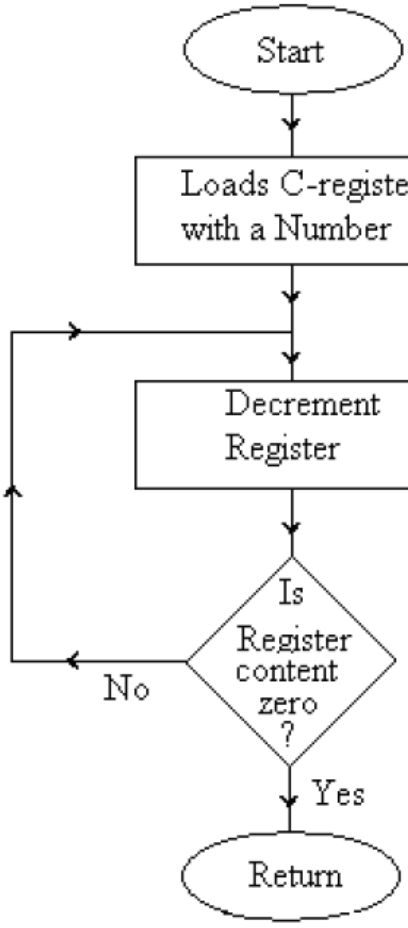

```
        XRA A
LOOP:   OUT 22H
        INR A
        CPI 0AH
        JNZ LOOP
```

The counting is performed at such a high speed that only the last count can be observed. To observe counting, there must be an appropriate time delay between the counts.

2.3.2 Time delays

Through programming time delay can be introduced, which is very useful in various applications such as digital clocks, traffic controls, digital process control and other data transfer controls. The procedure used to design a specific Time delay is similar to that used to set up a counter. A register is loaded with a number, depending on the time delay required and then the register is decremented until it reaches zero by setting up a loop with a conditional jump instruction. The delay introduced in the system will depend on the clock period of the system and the number of times (count) the instructions are executed inside the loop.

(a) **Using one Register (short delay)**



MVI C,FFH

Loop: DCR C

JNZ Loop

RET

Mnemonics	**T-States**
MVI C,FFH	7
Loop: DCR C	4
JNZ Loop	10/7
RET	10

In this program the instruction MVI C, FFH is executed only once and it takes 7 T-states to execute. The instruction DCR C is executed 255 times and thus takes 255 X 4 = 1020 T-states, since DCR instruction takes 4 T-states for its execution.

For the execution of JNZ instruction, it will go to loop 254 times, as the content of C-register will not be zero. So 254 x10 = 2540 T-states will be used in its calculation (till the content of C-register is not zero).

When the contents of C-register become zero, it will NOT jump to loop and this will take 7 T-states.

For the execution of RET statement it will take 10 T-states.

Thus total number of T-states taken for the execution of this program will be given by:

(7+1020+2540+7+10) T states = **3584** T-states

Alternative solution Approach

To calculate time delay, T-states for each instruction must be taken into account and the number of times instructions are executed in the loop. Assume the system clock is 3.125 MHz

The loop is repeated 255 times until Register C=0.
Thus
T_L i.e. No. of T-states *in the loop* = (254 X (4+10)) + (1 X (4+7))
= 3556 + 11
= 3567 T-states
T_O i.e. No. of T-states *outside the loop* = (7 +10) T-states

Total No. of T-states = 3567 + 17 T-states
= **3584** T-states

Clock frequency = 3.125 MHz

1 T-state = T (period) = 1/f = 1/ (3.125 X 10^6) seconds

1 T = 0.32 X 10^{-6} seconds

Thus the delay is 3574 T-states = 3584 X 0.32 X 10^{-6} seconds

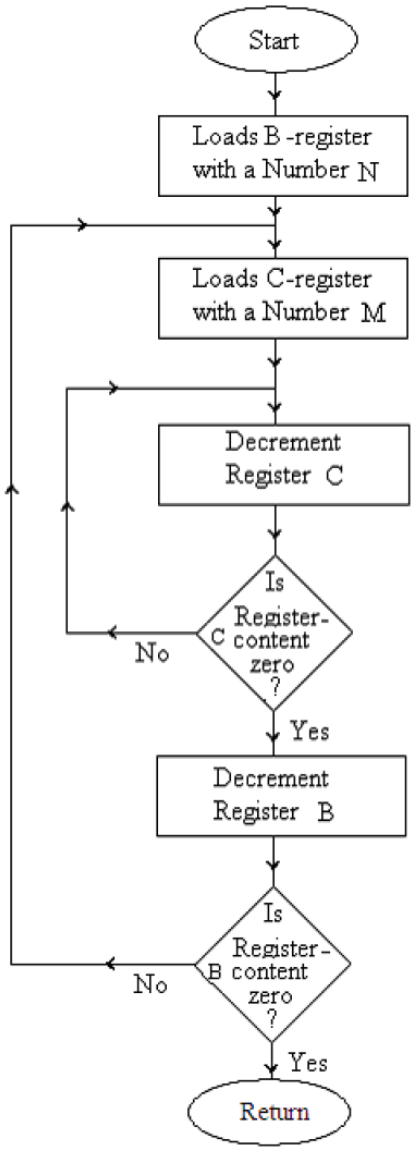
= **1.14688 msec**

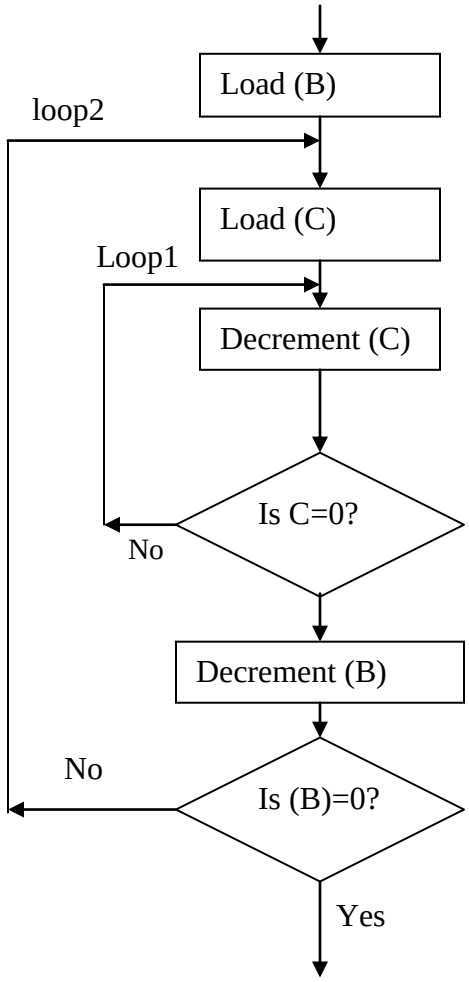
The time delay can be varied by changing the count number. However to increase the time delay, additional instructions are used or a register-pair is used to store the time delay. A longer time delay can also be achieved by using the technique of two loops or a register pair.

(a) Long Delays

i. Using two loops

Using register C in the inner loop and register B in the outer loop.





MVI B,B4H……7 T-states

MVI C,FFH…….7 T-states

Loop1: DCR C…….4 T-states

JNZ Loop1…….10/7 T-states

DCR B……..4 T-states

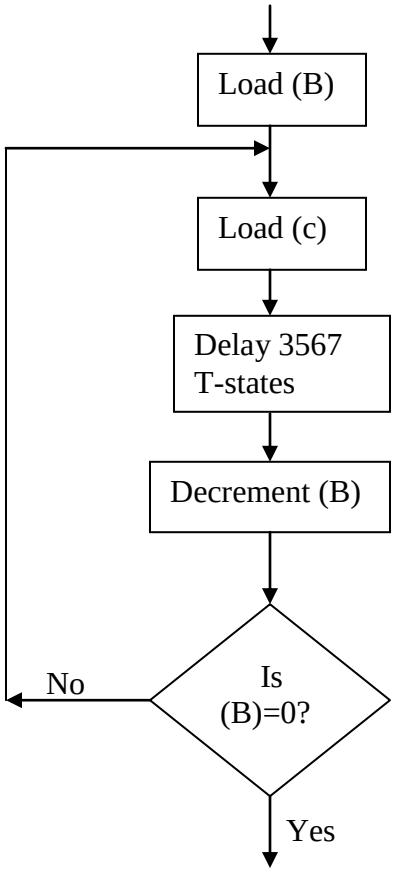
JNZ Loop 2…..10/7 T-states

$T_{\text{inner loop}}$ = 254(4+10) + (4+7)

= (254 X 14) + 11

= 3567 T-states

Re-drawing the flowchart:



$T_{\text{outer loop}}$ = 179(7+3567+4+10) + 1(7+3567+4+7)
= (179 X 3588) + 3585
= 642252 + 3585
= 645837 T-states

$T_{\text{outside loop}}$ = 7 T-states
Total No. of T-states = 645837 + 7
= 645844 T-states

Clock frequency = 3.125 MHz

1 T-state = T (period) = 1/f = 1/(3.125 X $10^{6)}$ seconds

1 T = 0.32 X 10^{-6} seconds

Thus the delay is 3574 T-states = 645844 X 0.32 X 10^{-6} seconds
= 0.20667008 sec
= **206.67 msec**

ii. Using a register-pair.

The time delay can be considerably increased by setting a loop and using a register-pair with a 16-bit number (Maximum FFFFH).

The 16-bit number is decremented by using the DCX instruction. Since DCX Instruction does not set or reset any flag, and without a test flag, the conditional Jump instruction cannot be executed, additional techniques must be used to set the flags. (e.g. zero flag)

e.g.

```
        LXI   H,2088H
LOOP:   DCX   B
        MOV   A,B       ; To set the Z flag. Load A with Contents of
        ORA   C         ; B and then OR (B) with (C). This affects
                        ; the flags.
        JNZ   LOOP
```

The OR instruction sets the zero-flag only when the contents of A, B and C are simultaneously zero. Therefore the loop is repeated 2088H times, equal to the count set in the registration pair.

	T-states
DCX	6
MOV	4
ORA	4
JNZ	10/7
LXI	10

$2088H = 8328_{(10)}$

T_L i.e. No. of T-states *in the loop* = 8327(6+4+4+10) + (6+4+4+7)
= 199869 T-states

T_O i.e. No. of T-states *outside the loop* = 10 T-states

Total No. of T-states, T_D = $T_L + T_O$
= 199869 + 10
= 199879 T-states

Clock frequency = 3.125 MHz
1 T-state = T (period) = 1/f = 1/ 3.125 X 10^6 seconds
1 T = 0.32 X 10^{-6} seconds

Thus the delay is 3574 T-states = 199879 X 0.32 X 10^{-6} seconds
= **63.96128 msec**

A register pair can also be used in conjunction with a single register or another register pair in a two loop structure to obtain even longer delays.

When a very precise delay is required, a loop is written and initialized with a count value that comes as close as possible to the required delay as possible. The delay is then "tuned" by adding instructions outside the loop to bring it to the required time.
One instruction that is particularly appropriate for this purpose is the *no-operation* instruction (NOP). This instruction can be used to add 4 T-states in the delay loop without affecting any of the registers or flags. Its only effect is to cause a delay of 4 clock periods.
If WAIT states occur in the execution of the instructions that comprise a delay routine, they must be included in the computation of the initial values of the loop control values.

The desired time delay can be obtained by using any or all available registers.

Exercise

A system has a clock operating at 3.072 MHz. Determine the count to be loaded into the register-pair BC in order to obtain a delay of 600 microseconds.

			T-states
START:	LXI	B, YY H	10
LOOP:	DCX	B	6
	MOV	A, C	4
	ORA	B	4
	JNZ	LOOP	10/7
	RET		10

Solution

Let YY Hex = X_{10}

T_{loop} = (X – 1) (6+4+4+10) + (6+4+4+7)
= (X – 1) 24 + 21
= 24X – 24 + 21 = 24X – 3 T-states

$T_{outside}$ = 10 + 10
= 20 T-states

Total T-states = 24X – 3 + 20 = 24X + 17 T-states

Clock period T = 1/3.072 x 10^6 seconds

Thus $(1/3.072 \times 10^6$ seconds) = 1 T

600 microseconds $= 1/(3.072 \times 10^6) \times 600 \times 10^{-6}$

$= 3.072 \times 10^6 \times 600 \times 10^{-6}$

= 1843.2 T-states

Therefore $24X + 17 = 1843.2$

$24X = 1826.2$

$X \approx 76.09$

$X \approx 76_{10}$

$X \approx 76_{10=}$ YY Hex = 004CH

2.3 Additional techniques for time delay

The disadvantages in using software delays techniques for real-time applications in which the demand for time accuracy is high, such as digital clocks, are as follows:

1. The accuracy of the time delay depends on the accuracy of the system clock
2. The microprocessor is occupied simply in a waiting loop when it could otherwise be employed to perform other useful functions.
3. The task of calculating accurate time delays is tedious.

In real-time applications, timers (integrated time circuits) are commonly used. The Intel 8253 is an example of a programmed timer programmed to provide timing with considerable accuracy. The disadvantage of using the hardware chip includes additional expense and the need for an extra chip in the system.

3.0 Subroutines and the stack

3.1 Subroutine

A subroutine is a group of instructions written separately from the main program to perform a function that occurs frequently in the main program i.e. it performs a subtask e.g. time delay, arithmetic operations (like finding cosines, sines, square root, cube roots etc.) of repeated occurrence.

The subroutine is written as a separate unit apart from the main program. The microprocessor transfers the program execution from the main program to the subroutine whenever it is called to perform a task. After completion of the task, the microprocessor returns to the main program. The subroutine technique eliminates the need of writing a sub-task repeatedly; thus it uses memory more efficiently and saves program development time.

3.2 Stack

A stack is a group of memory locations in the RAM (R/W Memory) that is used for the temporary storage of binary information during the execution of a program.

The starting memory location of the stack is defined in the main program and space is reserved, usually at the high end of the memory map. Initialization of the stack is done using the instruction LXI SP, which loads a 16-bit memory address in the stack pointer register of the microprocessor. Once the stack location is defined, storing of data bytes begins at the memory address that is one less than the address in the stack pointer register.

e.g. If the SP is loaded with the memory address 2348H (LXI SP, 2348H) the storing of data begins at 2347H and continues in the reverse numerical order (decreasing memory address).

Therefore, as a general practice, the stack is initialized at the highest available memory location to prevent the program from being destroyed by the stack information. The size of the stack is only limited by the available memory.

Data bytes in the register-pairs of the microprocessor can be stored on the stack (i.e. two bytes at a time) by using the instruction **PUSH**:

e.g. PUSH H – stores the contents of register pair HL on to the stack

PUSH PSW – stores the contents of the accumulator and flag register on the stack.

Data bytes can be transferred from the stack back to the respective register pair by using the instruction **POP**.

e.g. POP B – loads the register pair BC with the contents of the two top locations of the stack.

POP PSW – loads the accumulator and the flag register with the contents of the two top locations of the stack.

The stack is shared by the programmer and the microprocessor. The programmer can store and retrieve the contents of a register pair by using PUSH and POP instructions.

Similarly, the microprocessor automatically stores the contents of the PC when a subroutine is called and retrieves the contents of the stack on to the PC whenever it returns from a subroutine.

The storage and retrieval of data bytes on the stack follows the **LIFO** (Last-In-First-Out) sequence. Information in the stack locations is not destroyed (during read/POP operations) until new information is stored in these locations.

Uses of the stack

i) Used for saving register data during a branch to subroutine and return from subroutine. This technique is called “status saving”.

ii) Can be used to conveniently for saving microprocessor status before a program change. The same is true for storing the last program status on return to the previous task. This eliminates the need for saving register and status registers in memory locations reservations.

iii) The stack is also used to hold/store operands temporarily within a subroutine. It used as temporary storage in manipulation and computation routines.

iv) The stack may be used to transfer parameters to a subroutine (parameter passing).

The stack and the subroutine offer a great deal of flexibility in writing programs. A large software project is usually divided into subtasks called modules. These modules are developed independently as subroutines by different programmers. Each programmer can use all the microprocessor registers to write a subroutine without affecting the other part of the program. At the beginning of the subroutine module, the registers’ contents of the main program are stored in the stack and these are retrieved before returning to the main program.

3.3 Subroutine Execution

3.3.1 Calling subroutine

The process of transferring control to the subroutine is by means of a subroutine **CALL** instruction.

CALL is the mnemonic for ‘Call the subroutine’. Every CALL instruction must include the starting address (or label) of the desired subroutine. If a subroutine for squaring a number starts at address 5000H and a time delay subroutine at 8400H:

The execution of CALL 5000H will cause the microprocessor control to jump from the main program to the square subroutine while a CALL 8400H provides a jump to the time delay subroutine.

When a CALL is executed in the 8085 CPU, the contents of the PC are automatically saved in the stack. The CALL address is then loaded into the PC so that execution begins with the first instruction in the subroutine.

3.3.2 Return from subroutine

The final instruction in every subroutine is a one-byte RETurn instruction, which causes the program control to be returned to the main program at the address immediately following the CALL instruction.

When a RET instruction is executed, the address in the stack is loaded back into the PC, which returns control to the original program.

If one forgets to use a RET at the end of a subroutine, the computer will be unable to get back to the original program.

3.3.2.1 Unconditional CALL and RETurn

CALL is unconditional just like JMP. Once CALL has been fetched into the Instruction Register (IR), the computer will jump to the starting address of the subroutine. RET is also unconditional. Once encountered, the program control returns to the main program.

3.3.2.2 Conditional CALLs and RETurns

These are only obeyed when certain conditions are satisfied

Conditional Calls

Command	**Meaning**	**Action**
CNZ addr	Call if not zero	Branch to subroutine only if the zero flag is reset
CZ addr	Call if zero	Branches only if the zero flag is set
CNC addr	Call if not carry	Branches only if the carry flag is reset
CC addr	Call if carry	Branches only if the carry flag is set
CPO addr	Call if parity odd	Branches only if the parity flag is reset (i.e. odd)
CPE addr	Call if parity even	Branches only if the parity flag is set (i.e. even)
CP addr	Call if positive; S=0	Branches only if the sign flag is reset
CM addr	Call if minus; S=1	Branches only if the zero flag is set

Conditional Returns

Command	**Meaning**
RNZ	Return if not zero (Z=0)
RZ	Return if zero (Z=1)
RNC	Return if not carry (C=0)
RC	Return if carry (C=1)
RPO	Return if parity odd (P=0)
RPE	Return if parity even (P=1)
RP	Return if positive (S=0)
RM	Return if minus (S=1)

3.4 Subroutine Documentation and Parameter Passing

In a large program, subroutines are scattered all over the memory map and are called from the many locations. Various information is passed between a calling program and a subroutine. This procedure of data exchange between two programs is called ***parameter passing***.

It is therefore important to document a subroutine clearly and carefully. The documentation should include at least the following:

i) Function of the subroutine – state clearly what the subroutine does (for the user)

ii) I/O parameters – the parameter passed to a subroutine are listed as inputs and parameters returned to the calling program are listed as outputs.

iii) List of other subroutines called by this subroutine and also if a subroutine is calling another subroutine.

iv) List of registers used or modified in a subroutine – the register used by a subroutine may also require to be used by the calling program. Therefore it is necessary to save the register contents of the calling program on the stack at the beginning of the subroutine and to retrieve the contents before returning from the subroutine.

3.4.1 Methods of parameter passing

i) **Through registers** – registers can be used to pass parameters. This is an advantageous solution provided that registers are available; since one does not need to use a fixed memory location, the subroutine remains memory-independent.

Main program

```
    --
    --
MVI     B, 4CH
MVI     A,62H
CALL    SUBR
OUT     22H
    ---
```

Subroutine

```
SUBR:   DCR   B
        MOV   M, B
         --
         --
        RET
         --
         --
```

ii) **Through memory** – using memory has the advantage of greater flexibility (more data) but results in poorer performance and also in typing the subroutine in a given memory area.

Main program

```
LXI     8600H
MOV     M,C
CALL    SUBR2
    ---
    ---
```

Subroutine

```
SUBR2: INR    M
       ADD    B
          --
             --
       RET
```

The data being incremented in the subroutine is in memory where it was stored in the main program.

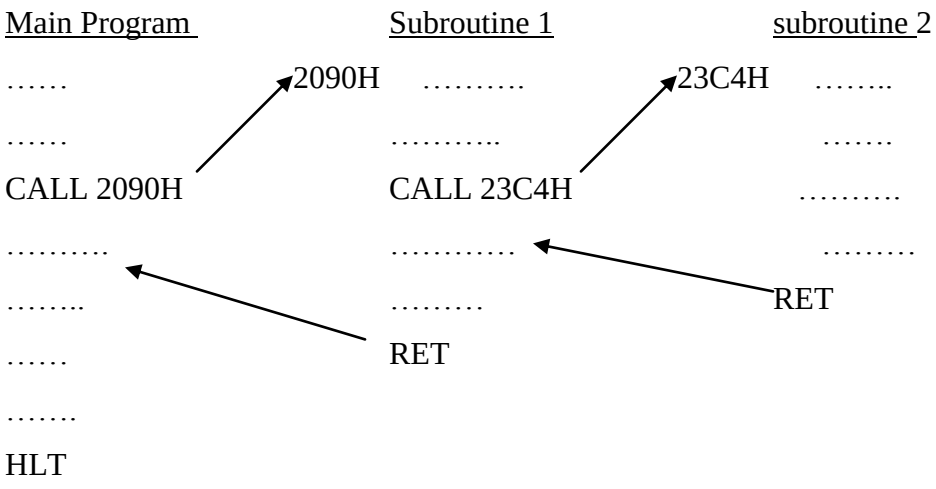
(iii) Through stack

```
Main program                    Subroutine
                                SUBR3:    POP    H   ; Pops return addresses
    --                                    POP    B   ; Retrieves data
    --                                                ; previously stored in DE
    --                                    PUSH   H   ; Restores return address
    --                                    MOV    D,A
PUSH      D                                 --
CALL      SUBR3                 --
  --                                      RET
  --
```

Data stored in the stack from register-pair DE in the main program is utilized in the subroutine from the stack. Using the stack has the same advantage as using registers; it is memory-independent. The only drawback is that it clutters the stack with data and therefore reduces the number of possible levels of subroutine calls. It also complicates the use of the stack and may require multiple stacks. The choice of the method to be used is up to the programmer.

3.4.2 Nesting

The programming technique of a subroutine calling another subroutine is called ***nesting.*** This process is limited only by the number of available stack locations. Such limitations generally occur when the stack overwrites, or is overwritten by, other storage. If the stack location is carefully established, deep subroutines nesting is rarely a problem for processors similar to the 8085. When a subroutine calls another subroutine, all return addresses are stored on the stack.



3.5 Other instructions involving the stack:

XTHL - This instruction exchanges the contents of the HL register-pair with the 2 bytes at the top or the stack. The SP value is not changed. The current stack top at location SP is exchanged with the contents of the L register. The contents of the H-register are exchanged with the byte one step down the stack (SP+1)

It allows access to the contents of the stack without changing the position of the stack pointer or losing the contents of the H and L registers.

SPHL – This instruction holds the contents of the HL register-pair into the stack pointer. When a designer wishes to set the stack pointer to a value that has been computed by the program, this value is placed in H and L, and then moved to the stack, using the *move HL to SP* instruction SPHL.

SPHL

(SP) ⟵ (HL)

3.6 Other jump related instruction

PCHL

The selection of one process from among many possible processes is facilitated by nesting several IF-THEN/ELSE structures or simply by placing several IF-THEM/ELSE structure in sequence. Or even more appropriate is the SELECT structure.

This structure tests data for multiple cases and selects the appropriate process for the given case. SELECT has only a single entry and a single exit.

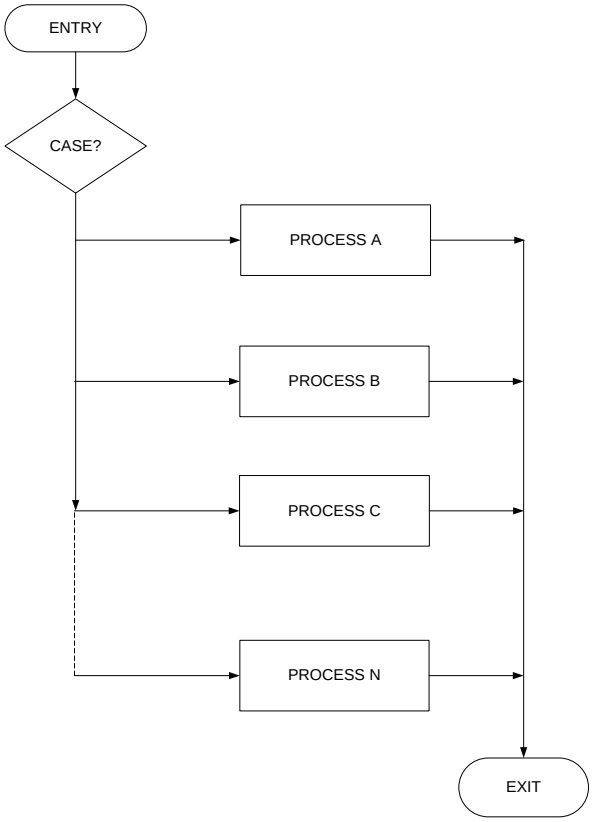
An instruction useful in creating a SELECT structure is the *move H and L to PC* instruction, PCHL

PCHL

(PCH) ⟵ (H)

(PCL) ⟵ (L)

This is a register indirect jump instruction. H and L contain the address to be branched to before the PCHL is executed. With the execution of PCHL, the program counter is set to the value of H and L.



SELECT structure

The PCHL instruction can be used to create a jump table, which is another type of select structure. The jump table contains starting addresses of alternative processes. Each address has two bytes; the low order address byte is in the first location associated with that entry, and the high order byte is in the second. The determination of which routine to branch to is based on a number that provides an index to the table. To maintain the single exit nature of a SEQUENCE structure, the last instruction of each routine is an unconditional jump to the same location.

**********’

References

1. Digital Computers Electronics; "An introduction to Microcomputers". 2nd edition by Albert Paul Malvino Ph.D.
2. Microprocessors and Programmed Logic, second edition, by Kenneth L. Short
3. "Introduction to Microprocessor 8085", By Dr. D. K. Kaushik, Dhanpat Rai Publishing co., New Delhi
4. Microprocessor Fundamentals 2nd edition by F. Halsall and P.F. Lister