# UNIT 3

## Chapter 1

# PROGRAMMING TECHNIQUES WITH ADDITIONAL INSTRUCTIONS

### 1.1 PROGRAMMING TECHNIQUES: LOOPING, COUNTING AND INDEXING

The programming technique used to instruct the microprocessor to repeat tasks is called **looping**. A loop is set up by instructing the microprocessor to change the sequence of execution and perform the task again. This process is accomplished by using 'Jump instructions'. In addition techniques such as **counting and indexing are used in setting up a conditional loop**.

Loops can be classified into two types:

- Continuous loop – repeats a task continuously
- Conditional loop – repeats a task until certain data conditions are met

### 1.1.1 Continuous Loop

A continuous loop is set up by using the 'unconditional Jump instructions' as shown in the flowchart (fig 3.1)

A program with a continuous loop does not stop repeating the tasks until the system is reset. Typical examples of such programs include a continuous counter or a continuous monitor system.

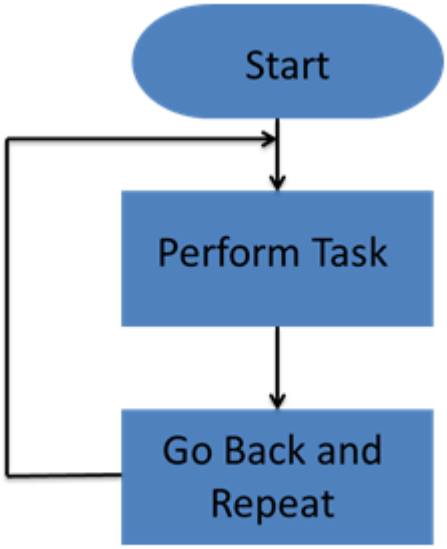



**Fig 3.1** Flowchart of a continuous loop

## 1.1.2 Conditional Loop

A conditional loop is set up by the conditional Jump instructions. These instructions check flags (Zero, Carry, etc.) and repeat the specified tasks if the conditions are satisfied. These loops usually include counting and indexing.

CONDITIONAL LOOP AND COUNTER

A counter is a typical application of a conditional loop. Example – for a microprocessor to repeat a task five times we set up a counter for 5 counts. Fig. 3.2 shows the flowchart for such a counter which is accomplished by using a conditional loop.

This flowchart can be translated to program using following steps-

1. Counter is set up by loading appropriate count in a register.
2. Counting is performed by either incrementing or decrementing counter register.
3. Loop is set up by conditional Jump instruction.
4. End of counting is indicated by a flag.

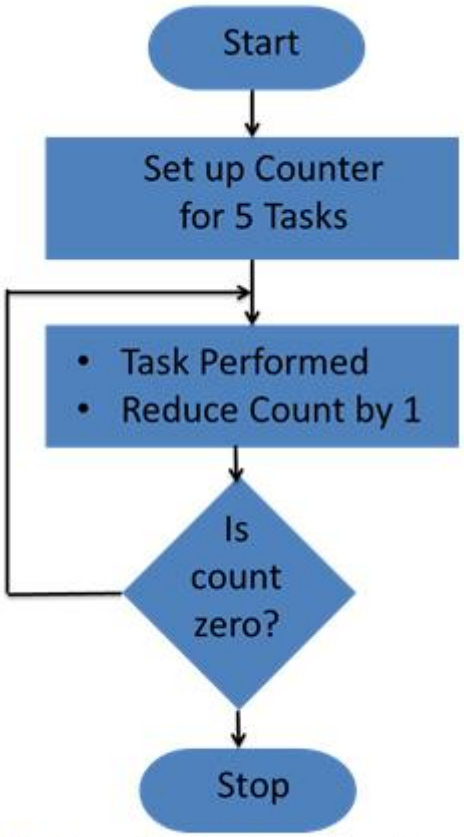



**Fig 3.2** Flowchart for Conditional Loop

It is easier to count down to zero than to count up because the zero flag is set when the register content is zero. While counting up requires compare instruction along with checking of flags. So this will increase the number of instructions.

CONDITIONAL LOOP, COUNTER AND INDEXING

Another type of loop includes indexing along with counter. Indexing means pointing or referencing objects with sequential numbers. In library, books are arranged according to numbers, and they are referred to or sorted by numbers. Similarly, data bytes are stored in memory locations, and those data bytes are referred to by their memory location.

Example –

Write a program to count the number of 1's in register A and the result is to be stored in register D.

https://www.youtube.com/watch?v=1d9sSoYYjcA&t=189s

## 1.2 ADDITIONAL DATA TRANSFER AND 16-BIT ARITHMETIC INSTRUCTIONS

This section introduces the instructions related to the data transfer between the microprocessor and memory. In addition, instructions for some 16-bit arithmetic operations are included. The opcodes are as follows:

1. **Loading 16-bit data in register pairs.**
   LXI $R_P$,16-bit data: Load Register Pair Immediate

2. **Data transfer from memory to microprocessor**
   MOV R,M: Move (from memory to register)
   LDAX B/D: Load Accumulator Indirect
   LDA 16-bit address: Load Accumulator Direct

3. **Data transfer from microprocessor to memory**
   MOV M,R: Move (from register to memory)
   STAX B/D: Store Accumulator Indirect
   STA 16-bit address: Store Accumulator Direct

4. **Loading 8-bit data to memory location**
   MVI M,8-bit data: Load 8-bit data in memory

5. **Incrementing/ Decrementing Register Pair**
   INX $R_P$: Increment Register Pair
   DCX $R_P$: Decrement Register Pair

### 1.2.1 16-Bit Data Transfer to Register Pairs (LXI $R_P$,16-bit data)

The LXI instructions perform functions similar to those of the MVI instructions, except that the LXI instructions load 16-bit data in register pairs and the stack pointer. These instructions do not affect the flags.

This is a 3-byte instruction. The register pair is loaded with 16-bit data specified in last two bytes of the instruction. The second byte is loaded in low- order register of register pair and the third byte is loaded in high-order register. There are four such instructions in the set as shown. The operands B, D and H represents BC, DE and HL register pairs respectively. SP represents Stack Pointer.

## Instructions:

| Opcode | Operand |
|---|---|
| LXI | B, 16-bit data |
| LXI | D, 16-bit data |
| LXI | H, 16-bit data |
| LXI | SP,16-bit data |

Fig 3.3 shows method to load register H with data 20H and register L with data 50H using single 'LXI' instruction (3bytes) while fig 3.4 shows the method to do the same using two 'MVI' instructions (4 Bytes).

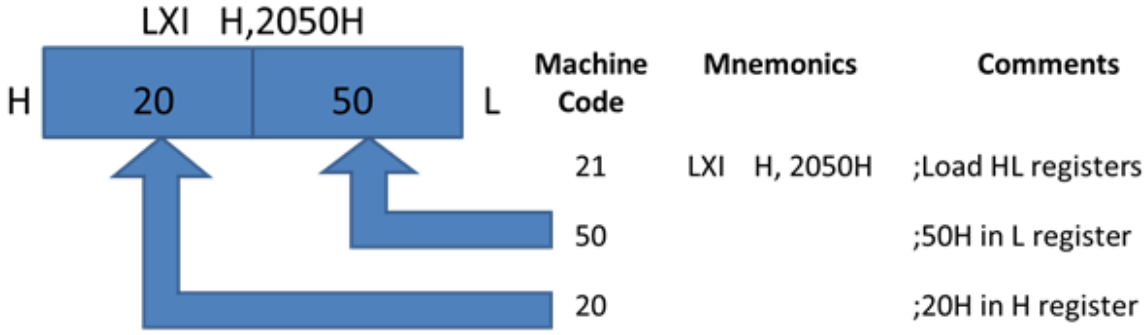



**Fig 3.3** Loading 16-bit data to H and L registers using LXI instruction

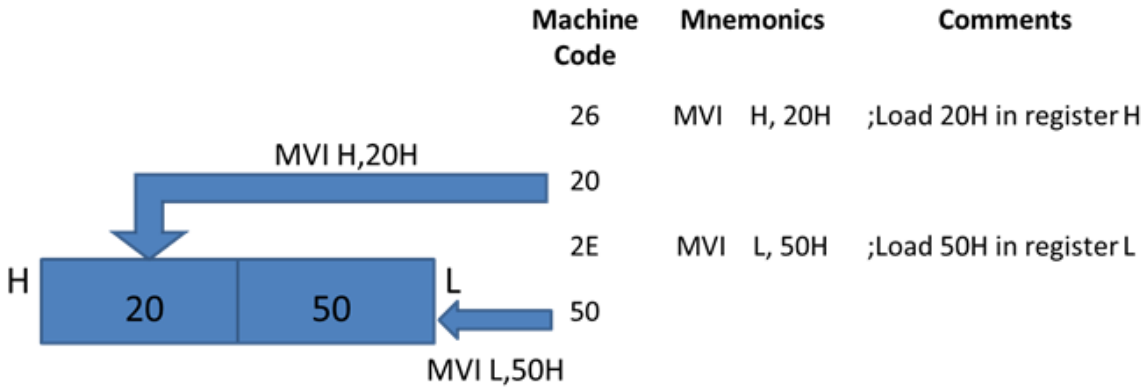



**Fig 3.4** Loading 16-bit data to H and L registers using MVI instructions

### 1.2.2 Data transfer (Copy) from Memory to the Microprocessor

XI instructions perform functions similar to those of the MVI instructions, except that the LXI instructions load 16-bit data in register pairs and the stack pointer. These instructions do not affect the flags.

1. **MOV R, M : Move from memory to register**
   - This is a 1 byte instruction.
   - Copies data from memory location to given register R.
   - R represents microprocessor registers A, B, C, D, E, H and L.
   - Address of the memory location is specified by content of HL register pair.
   - As the address is not specified directly hence this instruction belongs to indirect addressing mode.
2. **LDAX $R_p$ : Load Accumulator Indirect**
   - This is a 1 byte instruction.
   - Copies data from memory location to Accumulator.
   - $R_P$ represents microprocessor register pairs B-C and D-E. Hence the possible instructions are **LDAX B** and **LDAX D**.
   - Address of the memory location is specified by content of given register pair (register pair BC or DE)
   - As the address is not specified directly hence this instruction belongs to indirect addressing mode.
3. **LDA 16-bit address : Load Accumulator Direct**
   - This is a 3 byte instruction.
   - Copies data from memory location to Accumulator.
   - Address of the memory location is specified in last 2 bytes of the instruction.
   - As the address is specified directly hence this instruction belongs to direct addressing mode.

### 1.2.3 Data transfer (Copy) from the Microprocessor to Memory or directly into memory

These instructions are as follows.

1. **MOV M, R : Move from register to memory**
   - This is a 1 byte instruction.
   - Copies data from given register R to memory location.
   - R represents microprocessor registers A, B, C, D, E, H and L.
   - Address of the memory location is specified by content of HL register pair.
   - As the address is not specified directly hence this instruction belongs to indirect addressing mode.
2. **STAX $R_p$ : Store Accumulator Indirect**
   - This is a 1 byte instruction.
   - Copies data from Accumulator to memory location.
   - $R_P$ represents microprocessor register pairs B-C and D-E. Hence the possible instructions are **STAX B** and **STAX D**.
   - Address of the memory location is specified by content of given register pair (register pair BC or DE)
   - As the address is not specified directly hence this instruction belongs to indirect addressing mode.
3. **STA 16-bit address : Store Accumulator Direct**
   - This is a 3 byte instruction.
   - Copies data from Accumulator to memory location.
   - Address of the memory location is specified in last 2 bytes of the instruction.
   - As the address is specified directly hence this instruction belongs to direct addressing mode.
4. **MVI M, 8-bit data : Move immediate 8-bit data to Memory**
   - This is a 2 byte instruction.
   - Loads the memory location with given 8-bit data specified in $2^{nd}$ byte of instruction.
   - Address of the memory location is specified by the content of HL register pair.
   - This instruction belongs to Indirect and Immediate addressing mode.

### 1.2.4 Arithmetic Operations Related to 16-bits or Register Pairs

Though these instructions are related to incrementing or decrementing 16-bit content in register pairs, all these instructions are single byte instructions. These instructions do not affect any flag.

1. **INX $R_p$ : Increment Register Pair**

- 1 byte instruction.
- It treats the contents of two registers as one 16-bit number and increases the content by 1.
- This type includes 4 instructions, as follows –
    - INX B (Increment content of BC register pair)
    - INX D (Increment content of DE register pair)
    - INX H (Increment content of HL register pair)
    - INX SP (Increment content of Stack Pointer)

2. **DCX $R_p$ : Decrement Register Pair**

- 1 byte instruction.
- It treats the contents of two registers as one 16-bit number and decreases the content by 1.
- This type includes 4 instructions, as follows –
    - DCX B (Decrement content of BC register pair)
    - DCX D (Decrement content of DE register pair)
    - DCX H (Decrement content of HL register pair)
    - DCX SP (Decrement content of Stack Pointer)

## 1.3 ARITHMETIC INSTRUCTIONS RELATED TO MEMORY

Using 8085 instructions in this section we can copy data from microprocessor into memory and vice versa.

1. Indirect addressing using HL as memory pointer. HL register pair is used to copy data between registers and memory. Any instruction with operand M automatically assumes the HL pair as memory pointer.
    a. MOV R, M (1 byte)
    b. MOV M, R (1 byte)
    c. MVI M, 8-bit data (2 byte)

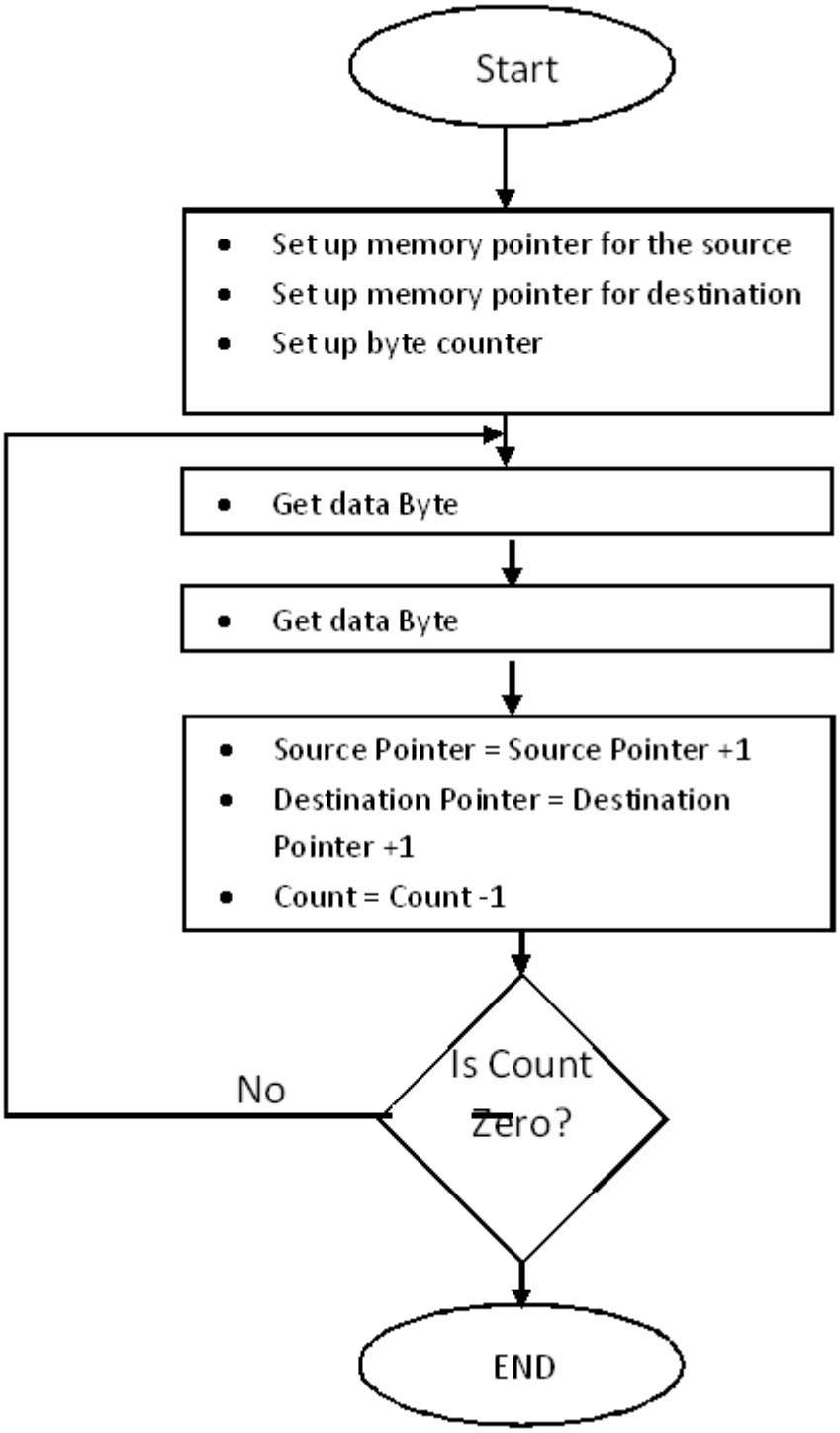
2. Indirect addressing using BC or DE as memory pointer. BC or DE register pair is used to copy data between registers and memory. However, this method is restricted to transfer data between memory and accumulator only. These are single byte instructions –
    a. LDAX B
    b. LDAX D
    c. STAX B
    d. STAX D
3. Direct addressing instructions. These instructions include memory address as operand. This method is also restricted to transfer data between memory and accumulator only. These are 3 byte instructions –
    a. LDA 16-bit address
    b. STA 16-bit address
4. Immediate addressing instructions. These instructions include 8-bit data as operand.–
    a. LDA 16-bit address
    b. STA 16-bit address

In addition to the above data transfer instructions, we also studied arithmetic instructions for 16-bit content of register pair i.e., 'INX $R_p$' and 'DCX $R_p$'

## 1.4 Illustrative Program: Block Transfer of Data Bytes

Problem Statement:

16 bytes of data are stored in memory locations at XX50H to XX5FH. Transfer the entire block of data to new memory locations starting from XX70H.



| Label | Mnemonics | | Explanation |
|---|---|---|---|
| | LXI | H, XX50H | Set up HL as pointer for source memory |
| | LXI | D, XX70H | Set up DE as pointer for destination memory |
| | MVI | C, 10H | Set up C as Byte Counter |
| LOOP | MOV | A, M | Get data byte from source memory |
| | STAX | D | Store data byte in destination memory |
| | INX | H | Increment Source memory pointer |
| | INX | D | Increment Destination memory pointer |
| | DCR | C | Decrement the counter by 1 |
| | JNZ | LOOP | Go back to get next data byte for transfer if Counter $\neq 0$ |
| | HLT | | End the program |

## 1.5 LOGIC OPERATIONS:

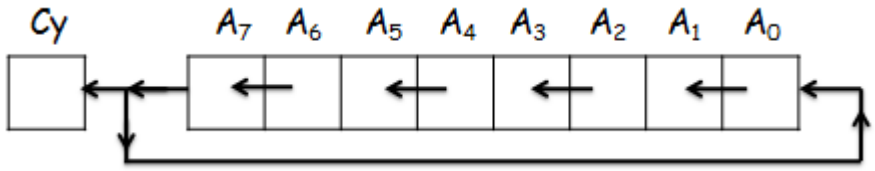
This group includes instructions for Logic AND, Logic OR, Logic EX-OR, Complement, Compare and Rotate.

### 1.5.1 Rotate

**RLC :** ROTATE ACCUMULATOR LEFT

- Format: $[A_n+1] \leftarrow [A_n]$, $[CY] \leftarrow [A_7]$, $[A_0] \leftarrow [A_7]$
- Implied addressing mode
- Logical group
- 1 Byte instruction
- Only CY Flag may be affected

The content of accumulator is rotated to left by 1-bit position. The bit $A_7$ is stored in bit $A_0$ as well as in carry flag.




**Example:**

Before Execution: Let [A] = 81 H and [CY] = 0

| CY | $A_7$ | $A_6$ | $A_5$ | $A_4$ | $A_3$ | $A_2$ | $A_1$ | $A_0$ |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

After Execution:

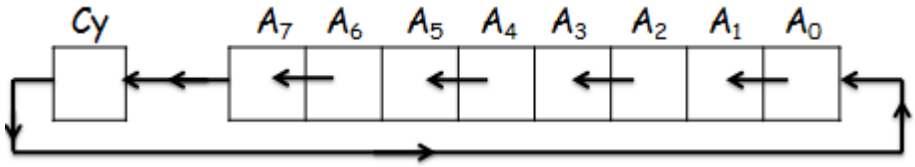
| CY | $A_7$ | $A_6$ | $A_5$ | $A_4$ | $A_3$ | $A_2$ | $A_1$ | $A_0$ |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |

[A] = 03 H & [CY] = 1

**RAL :** ROTATE ACCUMULATOR LEFT THROUGH CARRY

- Format: $[A_n+1] \leftarrow [A_n]$, $[CY] \leftarrow [A_7]$, $[A_0] \leftarrow [CY]$
- Implied addressing mode
- Logical group
- 1 Byte instruction
- Only CY Flag may be affected

The content of accumulator is rotated to left by 1-bit position through CY flag. CY flag is stored in bit $A_0$ of accumulator & bit $A_7$ of accumulator is stored in CY flag.




**Example:**

Before Execution: Let [A] = 81 H and [CY] = 0

| CY | $A_7$ | $A_6$ | $A_5$ | $A_4$ | $A_3$ | $A_2$ | $A_1$ | $A_0$ |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

After Execution:

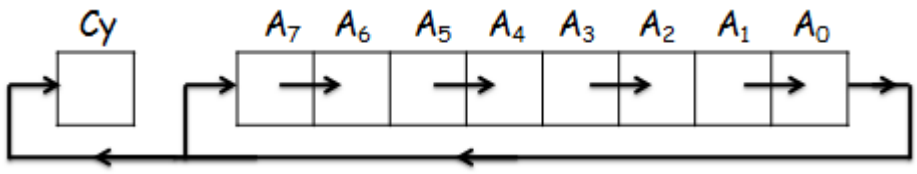
| CY | $A_7$ | $A_6$ | $A_5$ | $A_4$ | $A_3$ | $A_2$ | $A_1$ | $A_0$ |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

[A] = 02 H & [CY] = 1

**RRC :** ROTATE ACCUMULATOR RIGHT

- Format: $[A_n] \leftarrow [A_{n+1}]$, $[CY] \leftarrow [A_0]$, $[A_7] \leftarrow [A_0]$
- Implied addressing mode
- Logical group
- 1 Byte instruction
- Only CY Flag may be affected

The content of accumulator is rotated to right by 1-bit position. The bit $A_0$ is stored in bit $A_7$ as well as in carry flag.




**Example:**

Before Execution: Let [A] = 81 H and [CY] = 0

| CY | $A_7$ | $A_6$ | $A_5$ | $A_4$ | $A_3$ | $A_2$ | $A_1$ | $A_0$ |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

After Execution:

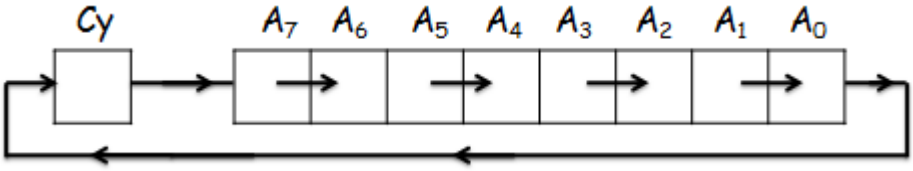
| CY | $A_7$ | $A_6$ | $A_5$ | $A_4$ | $A_3$ | $A_2$ | $A_1$ | $A_0$ |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

[A] = C0 H & [CY] = 1

**RAR :** ROTATE ACCUMULATOR RIGHT THROUGH CARRY

- Format: $[A_n] \leftarrow [A_n+1]$, $[CY] \leftarrow [A_7]$, $[A_0] \leftarrow [CY]$
- Implied addressing mode
- Logical group
- 1 Byte instruction
- Only CY Flag may be affected

The content of accumulator is rotated to right by 1-bit position through CY flag. CY flag is stored in bit $A_7$ of accumulator & bit $A_0$ of accumulator is stored in CY flag.




**Example:**

Before Execution: Let [A] = 81 H and [CY] = 0

| CY | $A_7$ | $A_6$ | $A_5$ | $A_4$ | $A_3$ | $A_2$ | $A_1$ | $A_0$ |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

After Execution:

| CY | $A_7$ | $A_6$ | $A_5$ | $A_4$ | $A_3$ | $A_2$ | $A_1$ | $A_0$ |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

[A] = C0 H & [CY] = 1

## 1.5.2 Compare

### 1. Compare Register: CMP R

- Register addressing Mode
- Logical Group
- 1 Byte instruction
- All flags affected
- This compares the content of the given register R with content of accumulator. Here R can be any register i.e., A, B, C, D, E, H or L.
- Comparison is done using subtraction of content of register from the content of accumulator.
- The content of accumulator remains unchanged.

(i) If [A] < [R] then CY = 1, Z = 0
(ii) If [A] = [R] then CY = 0, Z = 1
(iii) If [A] > [R] then CY = 0, Z = 0

Example : Let [A] = 88 H & [L] = 99 H
Instruction : CMP L
After Execution : [A] = 88 H, [L] = 99H, [CY] = 1, [Z] = 0

### 2. Compare Memory: CMP M

- Register Indirect addressing Mode
- Logical Group
- 1 Byte instruction
- This compares the content of the memory location whose address is pointed by HL register pair with content of accumulator.
- Comparison is done using subtraction of content of memory location from the content of accumulator.
- The content of accumulator remains unchanged. Only flags are modified.

(i) If [A] < [[HL]] then CY = 1, Z = 0
(ii) If [A] = [[HL]] then CY = 0, Z = 1

(iii) If [A] > [[HL]] then CY = 0, Z = 0

Example : Let [A] = 54 H, [[HL]] = 3000 H, [3000] = 12 H
Instruction : CMP L
After Execution : [A] = 54 H, [[HL]] = 3000 H, [3000] = 12 H
[CY] = 0, [Z] = 0

**3. Compare Data: CPI 8-bit data**

- Immediate addressing Mode
- Logical Group
- 2 Byte instruction
- This compares the 8-bit data specified in 2$^{nd}$ byte of the instruction with content of accumulator.
- The content of accumulator remains unchanged. Only flags are modified.

(i) If [A] < 8-bit data then CY = 1, Z = 0
(ii) If [A] = 8-bit data then CY = 0, Z = 1
(iii) If [A] > 8-bit data then CY = 0, Z = 0

Example : Let [A] = 36 H
Instruction : CPI 36 H
After Execution : [A] = 36 H, [CY] = 0, [Z] = 1

## 1.6 DYNAMIC DEBUGGING

**1. Single Step:**

- Allow to execute one instruction at a time and observe the result of each instruction
- This facility is built in with a hard wired circuit
- By pressing Single step key in microprocessor, we are able to observe addresses and codes as they are executed.
- This technique can spot – incorrect address, incorrect jump locations or loops, incorrect data or missing codes
- To effectively use this technique, reduce the number of loops and delay counts to minimum number.
- Better for short programs.
- This technique helps to infer the flag status.

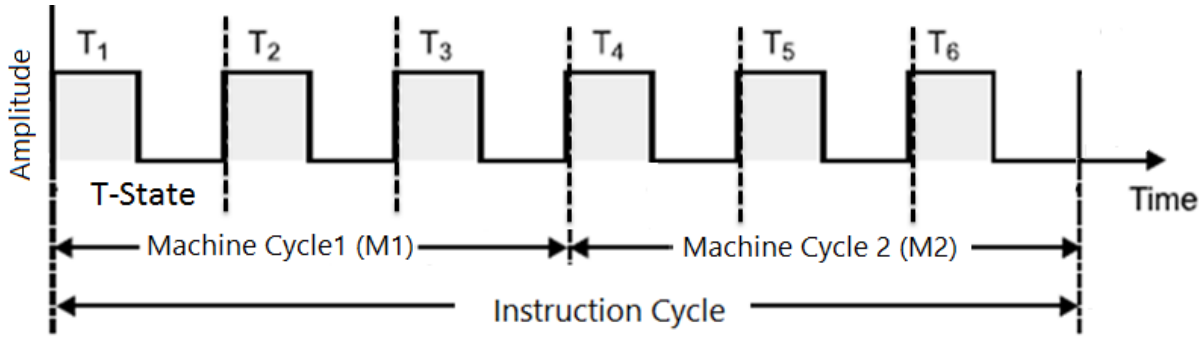
2. **Register Examine:**
   - Register examine key in microprocessor allows to examine the contents of microprocessor registers.
   - By pressing appropriate keys, the monitor program displays the content of each register.
   - This technique is used in conjunction with either the single step or break point facilities.
   - After executing a block of instructions, we can examine the register contents of the program at critical part and compare the observed content with expected.
3. **Break Point:**
   - Break point facility is a software routine that allows the program in sections.
   - The break point can be set using RST1 instruction.
   - When program is executed, it will be executed only until break point.
   - Hence we can check observed result of that much part with expected.
   - If that part of program is found satisfactory then second breakpoint can be set and again checked in the same manner.
   - With this facility we can isolate the segment of the program with errors.
   - This technique is used to check the errors in timing loops, I/O section and Interrupts.

# Chapter 2

# COUNTERS AND TIME DELAYS

## 2.1 COUNTERS AND TIME DELAYS

- T-state: The subdivision of an operation, which is performed in one clock period, is called as T-state.
- Machine Cycle: It is defined as the time required to complete any operation of accessing either memory or I/O which is the subpart of an instruction. (A machine cycle may consist of 3 to 6 T-states)
- Instruction Cycle: It is defined as the time required to complete the execution of an instruction. (A instruction cycle may consist of 1 to 5 Machine cycles)




- The time required for the execution of a program depends on the T-states.
- Each instruction requires different number of T-states to be executed.
- The time required for the execution of a program is given by

**Total Execution time = Number of T-states x Clock period**

Or

**Total Execution time = Number of T-states/Clock Frequency**

### 2.1.1 Delay Using 8-bit Counter

Steps to generate the delay using 8-bit counter

| Label | Instruction | Comment | T-States |
|---|---|---|---|
| | MVI D, Count | Load 8-bit count to reg D | 7 |
| LOOP | DCR D | Decrement count by 1 | 4 |
| | JNZ LOOP | If Count≠0 then repeat | 10/7 |

- The instruction 'MVI D, count' is executed only once.
- If JNZ condition is satisfied (Z=0) then 10 T-states are required and the program branches to DCR D instruction.
- If JNZ condition is not satisfied (Z=1) then 7 T-states are required and the program executes the next instruction after JNZ instruction.
- For every iteration in which JNZ condition is satisfied, the number of T-states is

**T-states for DCR D + T-states for JNZ satisfied = 4 + 10 = 14 T-states**

- These 14 T-states will be performed (count – 1) times.
- For last iteration in which JNZ condition is not satisfied, the number of T-states is

**T-states for DCR D + T-states for JNZ not satisfied = 4 + 7 = 11 T-states**

- Hence the total T-States required to execute the program are:

**Total T-States = 7 + [ (count – 1) x 14 ] + 11**

**For Example if Count = 09H = $(9)_{10}$**

Number of T-states = T-states for MVI D instruction

+ [(Count – 1) x (T-state of iterations for Loop)]

+ T-state for Last iteration

= 7 + [(9 – 1) x (4 + 10)] + (4 + 7)

= 7 + [8 x 14] + 11

= 7 + 112 + 11

= 130

**Problem:**

**Write a subroutine to produce longest delay using 8-bit counter. Also calculate total number of T-states**

**Solution:**

| Label | Instruction | Comment | T-States |
|---|---|---|---|
| | MVI D, FF H | Load largest 8-bit count FFH to reg D | 7 |
| LOOP | DCR D | Decrement count by 1 | 4 |
| | JNZ LOOP | If Count≠0 then repeat | 10/7 |
| | RET | Return to main program | 10 |

For maximum delay with 8-bit counter, Count = FFH

For Count = FFH = (255)10

Number of T-states = T-states for MVI D instruction

+ [(Count – 1) x (T-states of iterations for Loop)]

+ T-states for Last iteration

+ T-states for RET instruction

= 7 + [(255 – 1) x (4 + 10)] + (4 + 7) + 10

= 7 + [254 x 14] + 11 +10

= 7 + 3556 + 11 + 10

= 3584

Hence

**Maximum number of T-states in delay using 8-bit counter = 3584**

**Problem:**

**Write a subroutine to produce delay of 100 µs. Assume clock period to be 320ns**

**Solution:**

Total Execution Time = Number of T-states x Clock Period

Hence Number of T-states = Total Execution Time / Clock period

= 100µs / 320 ns

= $100 \times 10^{-6} / 320 \times 10^{-9} = 312.5 < 3584$

Hence Delay subroutine with 8-bit counter is enough

| Label | Instruction | Comment | T-States |
|---|---|---|---|
| | MVI D, Count | Load 8-bit count to reg D | 7 |
| LOOP | DCR D | Decrement count by 1 | 4 |
| | JNZ LOOP | If Count≠0 then repeat | 10/7 |
| | RET | Return to main Program | 10 |

Number of T-states = T-states for MVI D instruction

+ [(Count – 1) x (T-state of Loop)]

+ T-state for Last iteration

+ T-states for RET instruction

= 7 + [(count – 1) x (4 + 10)] + (4 + 7) + 10

But total T-states required = 312.5 ---- (from calculation)

Hence 312.5 = 7 + [(count – 1) x (4 + 10)] + (4 + 7) + 10

312.5 = 7 + [(count – 1) x 14] + 11 + 10

312.5 = 28 + [(count – 1) x 14]

Hence (count – 1) x 14 = 312.5 – 28 = 284.5

count – 1 = 284.5 / 14 = 20.32

Hence count = 20.32 + 1 = 21.32 ≈ 21 = 15H

### 2.1.2 Delay Using 16-bit Counter

Steps to generate the delay using 16-bit counter

| Label | Instruction | Comment | T-States |
|---|---|---|---|
| | LXI D, Count | Load 16-bit count to reg pair DE | 10 |
| LOOP | DCX D | Decrement count by 1 | 6 |
| | MOV A, D | Move reg D to Accumulator | 4 |
| | ORA E | Logically OR Accumulator (D) with E | 4 |
| | JNZ LOOP | If Count≠0 then repeat | 10/7 |

- The instruction 'LXI D, count' is executed only once.
- If JNZ condition is satisfied (Z=0) then 10 T-states are required and the program branches to DCX D instruction.
- If JNZ condition is not satisfied (Z=1) then 7 T-states are required and the program executes the next instruction after JNZ instruction.
- For every iteration in which JNZ condition is satisfied, the number of T-states is

**T-states for (DCX D + MOV A, D + ORA E + JNZ satisfied)**

= (6 + 4 + 4 + 10) = **24 T-states**

- These 24 T-states will be performed (count – 1) times.
- For last iteration in which JNZ condition is not satisfied, the number of T-states is

**T-states for (DCX D + MOV A, D + ORA E + JNZ not satisfied)**

= (6 + 4 + 4 + 7) = **21 T-states**

- Hence the total T-States required to execute the program are:

**Total T-States = 10 + [(count – 1) x 24] + 21**

**For Example Count = 0FFFH = $(4095)_{10}$**

Number of T-states = T-states for LXI D instruction

+ (Count – 1) x (T-state of iterations for Loop)

+ T-state for Last iteration

= 10 + (4095 – 1) x (6 + 4 + 4 + 10) + (6 + 4 + 4 + 7)

= 10 + 4094 x 24 + 21

= 10 + 98256 + 21

= 98287

**Problem:**

**Write a subroutine to produce longest delay using 16-bit counter. Also calculate total number of T-states.**

**Solution:**

For maximum delay with 16-bit counter, Count = FFFFH

| Label | Instruction | | T-States |
|---|---|---|---|
| | LXI | D, FFFF H | 10 |
| LOOP | DCX | D | 6 |
| | MOV | A, D | 4 |
| | ORA | E | 4 |
| | JNZ | LOOP | 10/7 |
| | RET | | 10 |

**For Count = FFFFH = $(65535)_{10}$**

Number of T-states = T-states for LXI D instruction

+ [(Count – 1) x (T-states of iterations for Loop)]

+ T-states for Last iteration

+ T-states for RET instruction

= 10 + [(65535 – 1) x (6+4+4+10)] + (6+4+4+7) + 10

= 10 + [65534 x 24] + 21 +10

= 10 + 1572816 + 21 + 10

= 1572857

**Hence Total number of T-states in delay using 16-bit counter = 1572857**

**Problem:**

**Write a subroutine to produce delay of 10 ms. Assume clock frequency to be 3MHz**

**Solution:**

Total execution time = Number of T-states x Clock Period

Total execution time = Number of T-states / Clock frequency

Hence Number of T-states = Total execution time x Clock frequency

= 10 ms x 3 MHz

= $(10 \times 10^{-3}) \times (3 \times 10^{6})$

= 30000 > 3584

Delay subroutine with 8-bit counter is not enough

Hence Delay Subroutine with 16-bit counter is needed

| Label | Instruction | T-States |
|---|---|---|
| | LXI D, Count | 10 |
| DELAY | DCX D | 6 |
| | MOV A, D | 4 |
| | ORA E | 4 |
| | JNZ DELAY | 10/7 |
| | RET | 10 |

Number of T-states = T-states for LXI D instruction

+ [(Count – 1) x (T-state of iterations for Loop)]

+ T-state for Last iteration

+ T-states for RET instruction

= 10 + [(count – 1) x (6+4+4+10)] + (6+4+4+7) + 10

= 10 + [(count – 1) x 24] + 21 + 10

But total T-states required = 30000 ---- (from calculation)

Hence 30000 = 10 + (count – 1) x 24 + 21 + 10

30000 = 41 + (count – 1) x 24

Hence (count – 1) x 24 = 30000 – 41 = 29959

count – 1 = 29959 / 24 = 1248.3

Hence count = 1248.3 + 1 = 1249.3 ≈ 1249 = 4E1H

Hence **count = 04E1H**

### 2.1.3 Delay Using Nested Loop

Steps to generate the delay using nested loop

| Label | Instruction | | Comment | T-States |
|---|---|---|---|---|
| | MVI | B,CountM | Initialize the Multiplier | 7 |
| LOUT | MVI | C,CountD | Initialize the Delay count | 7 |
| LIN | DCR | C | Decrement delay count by 1 | 4 |
| | JNZ | LIN | If delay count≠0 then repeat | 10/7 |
| | DCR | B | Decrement multiplier count by 1 | 4 |
| | JNZ | LOUT | If multiplier count≠0 then repeat | 10/7 |

- This method has more than one loop.
- The outer loop sets the multiplying count to the delays provided by the inner loop.
- From program we can see that Count2 forms inner loop while Count1 forms outer loop

Hence CountM = Multiplier count

CountD = Delay count

T-states for Inner Loop are

$T_{Lin}$ = [(delay count – 1) x (T-state of iterations for Inner Loop)]

+ T-state for Last iteration

= [(CountD – 1) x (4+10)] + (4+7)

= [(CountD – 1) x 14] + 11

T-states for Outer loop are

$T_{Lout}$ = [(CountM – 1) x (7+$T_{Lin}$ + 4+10)] + (7+$T_{Lin}$ +4+7)

$$= [(CountM - 1) \times (T_{Lin} + 21] + (T_{Lin} + 18)$$

Total number of T-states

$T_{Final}$ = T-states for MVI B + $T_{Lout}$

$= 7 + T_{Lout}$

## 2.2 ILLUSTRATIVE PROGRAMS

### 2.2.1 Hexadecimal Counter:

**Problem statement:**

Write an assembly language program to count continuously from 00H to FFH with delay of 1 second. (Hexadecimal UP Counter)

**Solution:**

Main Program:

| Label | Mnemonics | Explanation |
|---|---|---|
| | LXI SP,7050H | Initialize Stack pointer with 7050H |
| | MVI A, 00H | Load 00H to Accumulator |
| BACK | CALL DISPLAY | Call Display subroutine to display content of Accumulator |
| | CALL DELAY | Call Delay subroutine |
| | INR A | Increment content of Accumulator by 1 |
| | JMP BACK | Jump unconditionally to 'BACK' for continuous counter |
| | HLT | End the program |

Delay Subroutine:

(Since delay of 1s is required 8-bit counter is not enough. Hence delay with 16-bit counter is used)

| Label | Mnemonics | Explanation |
|---|---|---|
| **DELAY** | LXI D, count | Load 16-bit count in reg pair DE |
| LOOP | DCX D | Decrement content of reg pair DE by 1 |
| | MOV A, D | Copy content of reg D to Accumulator |
| | ORA E | Logical OR reg E with Accumulator |
| | JNZ LOOP | If Zero flag not set then jump to 'LOOP' |
| | RET | Return to Main Program |

**Problem statement:**

Write an assembly language program to count continuously from FFH to 00H with delay of 1 second. (Hexadecimal DOWN Counter)

**Solution:**

Main Program:

| Label | Mnemonics | Explanation |
|---|---|---|
| | LXI SP,7050H | Initialize Stack pointer with 7050H |
| | MVI A, FFH | Load FFH to Accumulator |
| BACK | CALL DISPLAY | Call Display subroutine to display content of Accumulator |
| | CALL DELAY | Call Delay subroutine |
| | DCR A | Decrement content of Accumulator by 1 |
| | JMP BACK | Jump unconditionally to 'BACK' for continuous counter |
| | HLT | End the program |

Delay Subroutine:

(Since delay of 1s is required 8-bit counter is not enough. Hence delay with 16-bit counter is used)

| Label | Mnemonics | Explanation |
|---|---|---|
| **DELAY** | LXI D, count | Load 16-bit count in reg pair DE |
| LOOP | DCX D | Decrement content of reg pair DE by 1 |
| | MOV A, D | Copy content of reg D to Accumulator |
| | ORA E | Logical OR reg E with Accumulator |
| | JNZ LOOP | If Zero flag not set then jump to 'LOOP' |
| | RET | Return to Main Program |

**Problem statement:**

Write an assembly language program to count continuously from 00H to FFH and then back from FFH to 00H with delay of 100 micro second.
(Hexadecimal UP-DOWN Counter)

**Solution:**

Main Program:

| Label | Mnemonics | Explanation |
|---|---|---|
| | LXI SP,7050H | Initialize Stack pointer with 7050H |
| REPEAT | MVI A, 00H | Load 00H to Accumulator |
| BACK1 | CALL DISPLAY | Call Display subroutine to display content of Accumulator |
| | CALL **DELAY** | Call Delay subroutine |
| | INR A | Increment content of Accumulator by 1 |
| | CPI 00H | Compare Accumulator with 00H |
| | JNZ BACK1 | If Zero flag not set jump to 'BACK1' for up counter |
| | MVI A, FFH | Load FFH to Accumulator |
| BACK2 | CALL DISPLAY | Call Display subroutine to display content of Accumulator |
| | CALL **DELAY** | Call Delay subroutine |
| | DCR A | Decrement content of Accumulator by 1 |
| | CPI FFH | Compare Accumulator with FFH |
| | JNZ BACK2 | If Zero flag not set jump to 'BACK2' for down counter |
| | JMP REPEAT | Jump unconditionally to 'REPEAT' to repeat the whole process |
| | HLT | End the program |

Delay Subroutine:
(Since delay of 500 μs is required 8-bit counter is enough)

| Label | Mnemonics | Explanation |
|---|---|---|
| **DELAY** | MVI D, count | Load 8-bit count in reg D |
| LOOP | DCR D | Decrement content of reg D by 1 |
| | JNZ LOOP | If Zero flag not set then jump to 'LOOP' |
| | RET | Return to Main Program |

## 2.2.2 Modulo-10 Counter (0 to 9):

**Problem statement:**

Write an assembly language program to count continuously from 0 to 9 with delay of 1 second. (Modulo-10 UP Counter)

| Label | Mnemonics | Explanation |
|---|---|---|
| | LXI SP,7050H | Initialize Stack pointer with 7050H |
| REPEAT | MVI A, 00H | Load 00H to Accumulator |
| BACK | CALL DISPLAY | Call Display subroutine to display content of Accumulator |
| | CALL **DELAY** | Call Delay subroutine |
| | INR A | Increment content of Accumulator by 1 |
| | CPI 0AH | Compare Accumulator with 0AH to check if Accumulator > 9 |
| | JNZ BACK | If Zero flag not set jump to 'BACK' for up counter |
| | JMP REPEAT | Jump unconditionally to 'REPEAT' to repeat the whole process |
| | HLT | End the program |

Delay Subroutine:

(Since delay of 1s is required 8-bit counter is not enough. Hence delay with 16-bit counter is used)

| Label | Mnemonics | Explanation |
|---|---|---|
| **DELAY** | LXI D, count | Load 16-bit count in reg pair DE |
| LOOP | DCX D | Decrement content of reg pair DE by 1 |
| | MOV A, D | Copy content of reg D to Accumulator |
| | ORA E | Logical OR reg E with Accumulator |
| | JNZ LOOP | If Zero flag not set then jump to 'LOOP' |
| | RET | Return to Main Program |

**Problem statement:**

Write an assembly language program to count continuously from 9 to 0 with delay of 1 second. (Modulo-10 DOWN Counter)

| Label | Mnemonics | Explanation |
|---|---|---|
| | LXI SP,7050H | Initialize Stack pointer with 7050H |
| REPEAT | MVI A, 09H | Load 09H to Accumulator |
| BACK | CALL DISPLAY | Call Display subroutine to display content of Accumulator |
| | CALL **DELAY** | Call Delay subroutine |
| | DCR A | Decrement content of Accumulator by 1 |
| | CPI FFH | Compare Accumulator with FFH to check if Accumulator crossed 00H |
| | JNZ BACK | If Zero flag not set jump to 'BACK' for up counter |
| | JMP REPEAT | Jump unconditionally to 'REPEAT' to repeat the whole process |
| | HLT | End the program |

Delay Subroutine:

(Since delay of 1s is required 8-bit counter is not enough. Hence delay with 16-bit counter is used)

| Label | Mnemonics | Explanation |
|---|---|---|
| **DELAY** | LXI D, count | Load 16-bit count in reg pair DE |
| LOOP | DCX D | Decrement content of reg pair DE by 1 |
| | MOV A, D | Copy content of reg D to Accumulator |
| | ORA E | Logical OR reg E with Accumulator |
| | JNZ LOOP | If Zero flag not set then jump to 'LOOP' |
| | RET | Return to Main Program |

## 2.2.3 Modulo-100 Counter (00 to 99):

**Problem statement:**

Write an assembly language program to count continuously from 00 to 99 with delay of 1 second. (Modulo-100 UP Counter)

| Label | Mnemonics | Explanation |
|---|---|---|
| | LXI SP,7050H | Initialize Stack pointer with 7050H |
| | MVI A, 00H | Load 00H to Accumulator |
| LOOP | CALL DISPLAY | Call Display subroutine to display content of Accumulator |
| | CALL **DELAY** | Call Delay subroutine |
| | ADI 01H | Increment content of Accumulator by 1 |
| | DAA | |
| | JMP LOOP | Jump unconditionally to 'LOOP' to repeat the whole process |
| | HLT | End the program |

Delay Subroutine:

(Since delay of 1s is required 8-bit counter is not enough. Hence delay with 16-bit counter is used)

| Label | Mnemonics | Explanation |
|---|---|---|
| **DELAY** | LXI D, count | Load 16-bit count in reg pair DE |
| LOOP | DCX D | Decrement content of reg pair DE by 1 |
| | MOV A, D | Copy content of reg D to Accumulator |
| | ORA E | Logical OR reg E with Accumulator |
| | JNZ LOOP | If Zero flag not set then jump to 'LOOP' |
| | RET | Return to Main Program |

## 2.2.4 Generating Pulse Waveforms

**Problem statement:**

Write an assembly language program to generate square waveform delay of 100μs.

| Label | Mnemonics | Explanation |
|---|---|---|
| | LXI SP,7050H | Initialize Stack pointer with 7050H |
| REPEAT | MVI A, 00H | Load 09H to Accumulator |
| | OUT PORT0 | Call Display subroutine to display content of Accumulator |
| | CALL **DELAY** | Call Delay subroutine |
| | MVI A, FFH | Decrement content of Accumulator by 1 |
| | OUT PORT0 | Call Display subroutine to display content of Accumulator |
| | CALL **DELAY** | Call Delay subroutine |
| | JMP REPEAT | Jump unconditionally to 'REPEAT' to repeat the whole process |
| | HLT | End the program |

Delay Subroutine:
(Since delay of 100 μs is required 8-bit counter is enough)

| Label | Mnemonics | Explanation |
|---|---|---|
| **DELAY** | MVI D, count | Load 8-bit count in reg D |
| LOOP | DCR D | Decrement content of reg D by 1 |
| | JNZ LOOP | If Zero flag not set then jump to 'LOOP' |
| | RET | Return to Main Program |

**Problem statement:**

Write an assembly language program to generate triangular waveform.

| Label | Mnemonics | Explanation |
|---|---|---|
| | LXI SP,7050H | Initialize Stack pointer with 7050H |
| REPEAT | MVI A, 00H | Load 09H to Accumulator |
| | OUT PORT1 | Output the content of accumulator to Port 1 |
| | INR A | |
| | CALL **DELAY** | Call Delay subroutine |
| | JMP REPEAT | Jump unconditionally to ‘REPEAT’ to repeat the whole process |
| | HLT | End the program |

# Chapter 3

# STACK AND SUBROUTINE

## 3.1 STACK

- The stack in 8085 microprocessor can be described as a set of memory locations in R/W memory, specified by programmer in the main program.
- These memory locations are used to store binary information temporarily during execution of a program.
- The beginning of the stack is defined in the main program by using instruction LXI SP, which loads 16-bit address in the Stack Pointer register. This location is termed as the top of the stack.
- Once stack location is defined, storing of data begins at the memory location that is one less than the address in Stack Pointer. Data will be stored by decrementing stack pointer each time continuously.
- Therefore stack is generally defined at highest R/W location and program is written from lowest R/W location just to prevent the program from being destroyed by Stack information.
- The programmer can store and retrieve the contents of register by using PUSH and POP instructions.
- The instructions necessary for using the stack are explained below:

| Opcode | Operand | Bytes | Addressing Mode | Explanation |
|---|---|---|---|---|
| LXI | SP, 16-bit | 3 | Immediate | Load stack pointer with a 16-bit address |
| PUSH | $R_P$ | 1 | Register | • Push content of register pair on Stack<br>• 1st Stack Pointer is decremented by 1 and content of higher register is stored<br>• Then Stack Pointer is again decremented by 1 and content of lower register is stored<br>Hence [SP] = [SP] – 2 at the end |
| PUSH | PSW | 1 | Register | • Push content of Program Status Word on Stack<br>• 1st Stack Pointer is decremented by 1 and content of Accumulator is stored<br>• Then Stack Pointer is again decremented by 1 and content of Flag register is stored<br>Hence [SP] = [SP] – 2 at the end |

| | | | | |
|---|---|---|---|---|
| POP | $R_P$ | 1 | Register | • Pop off content of Stack to register pair<br>• 1st content of Stack location is stored to lower register and Stack Pointer is incremented by 1<br>• Then content of this Stack location is stored to higher register and Stack Pointer is again incremented by 1<br>Hence [SP] = [SP] + 2 at the end |
| POP | PSW | 1 | Register | • Pop off content of Stack as Program Status Word<br>• 1st content of Stack location is stored to Flag egister and Stack Pointer is incremented by 1<br>• Then content of this Stack location is stored to Accumulator and Stack Pointer is again incremented by 1<br>Hence [SP] = [SP] + 2 at the end |

## 3.2 SUBROUTINE

- A Subroutine is a group of instructions written separately from the main program to perform a function that occurs repeatedly in the main program.
- Example: If a time delay is required between three successive events, then three delays can be written in the main program. To avoid this repetition of the same delay instructions, the subroutine technique is used.
- Delay instructions are written once, separately from main program, and are called by the main program when needed.
- The microprocessor 8085 has two instructions to implement subroutines – CALL (call the subroutine) and RET (Return to main program).
- CALL instruction is written in main program and is used to call the required subroutine.
    - This instruction is written as 'CALL 16-bit address' where this 16-bit address belongs to location from where the subroutine is written.
    - When CALL is executed the address of instruction next to the CALL is stored in Stack and program execution is transferred to subroutine address.
- RET instruction is written at the end of the subroutine and used to return to the main program after executing subroutine.
    - When RET is executed the address stored in Stack is retrieved and the execution is resumed in main program from where it was left.
- Further explanation of CALL and RET is given below.

## Unconditional Call and Unconditional Return:

| Opcode | Operand | Bytes | Addressing Mode | Explanation |
|---|---|---|---|---|
| CALL | 16-bit address | 3 | Immediate | Call Subroutine Unconditionally<br>• The content of program counter is stored on the stack.<br>• Stack pointer is decremented by 2<br>• Program jump unconditionally to address specified in last 2 bytes of CALL instruction<br>• This instruction must be accompanied by Return instruction at the end of the subroutine |
| RET | | 1 | Implied | Return from Subroutine Unconditionally<br>• Pops off the content of stack (address stored during Call) back to Program counter<br>• Stack Pointer is incremented by 2<br>• Execution unconditionally returns from the subroutine to main program.<br>Hence [SP] = [SP] – 2 at the end |

Stack and Subroutine:

https://www.youtube.com/watch?v=8nOl4WnwxbI

## 3.3 RESTART, CONDITIONAL CALL AND CONDITIONAL RETURN INSTRUCTIONS

In addition to CALL and RET instruction, 8085 instruction set includes eight Restart instructions, eight conditional Call instructions and eight conditional Return instructions

### 3.3.1 Restart (RST) instructions

- 1-byte instructions that transfer the program execution to location having monitor programs.
- These instructions can be termed as software interrupts
- All these are high priority and vectored interrupts
- When executed the execution is transferred to the monitor program stored from the vector location belonging to that Restart instruction.
- The list of eight RST instruction is as follows:

| Instruction | Vector Location |
|---|---|
| RST0 | 0000H |
| RST1 | 0008H |
| RST2 | 0010H |
| RST3 | 0018H |
| RST4 | 0020H |
| RST5 | 0028H |
| RST6 | 0030H |
| RST7 | 0038H |

### 3.3.2 Conditional Call and Conditional Return:

The conditional call and conditional return instructions are based on condition of four flags: Carry, Zero, Sign and Parity. In case of conditional call instruction, the program is transferred to subroutine if the condition is satisfied otherwise the main program is continued. In case of a conditional return instruction the sequence returns to the main program is the condition is satisfied otherwise the sequence of the subroutine is continued. The conditional call and return instructions are listed as follows.

## Conditional Call:

| | | |
|---|---|---|
| CC | Call on Carry | Call Subroutine if Carry flag is set (CY = 1) |
| CNC | Call on No Carry | Call Subroutine if Carry flag is reset (CY = 0) |
| CZ | Call on Zero | Call Subroutine if Zero flag is set (Z = 1) |
| CNZ | Call on No Zero | Call Subroutine if Zero flag is reset (Z = 0) |
| CM | Call on Minus | Call Subroutine if Sign flag is set (S = 1) |
| CP | Call on Plus | Call Subroutine if Sign flag is reset (S = 0) |
| CPE | Call on Parity Even | Call Subroutine if Parity flag is set (P = 1) |
| CPO | Call on Parity Odd | Call Subroutine if Parity flag is reset (P = 0) |

## Conditional Return:

| | | |
|---|---|---|
| RC | Return on Carry | Return if Carry flag is set (CY = 1) |
| RNC | Return on No Carry | Return if Carry flag is reset (CY = 0) |
| RZ | Return on Zero | Return if Zero flag is set (Z = 1) |
| RNZ | Return on No Zero | Return if Zero flag is reset (Z = 0) |
| RM | Return on Minus | Return if Sign flag is set (S = 1) |
| RP | Return on Plus | Return if Sign flag is reset (S = 0) |
| RPE | Return on Parity Even | Return if Parity flag is set (P = 1) |
| RPO | Return on Parity Odd | Return if Parity flag is reset (P = 0) |