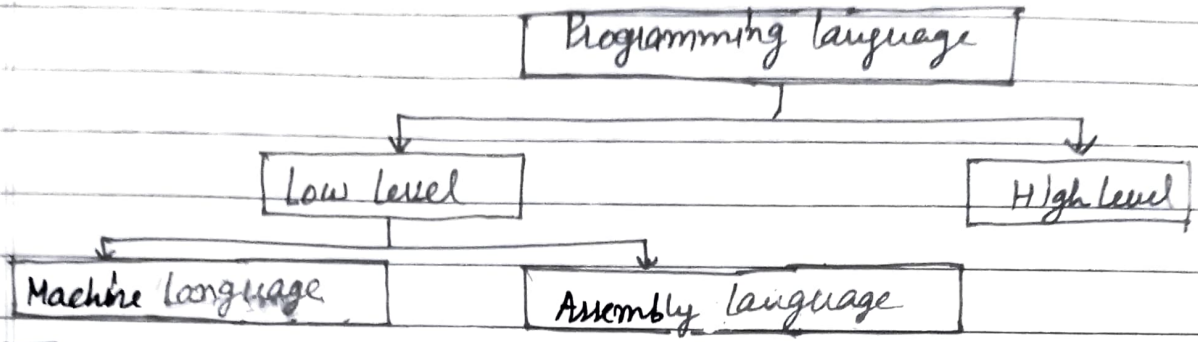
Assembly Language Programming based on 8085

Types of Programming Languages



There are mainly two types of programming languages
- Low level Programming language
- High level Programming language

- **Low level languages**

Low level language is a programming language in which each statement is directly translated into a single machine code. It is machine dependent.

→ **Machine language**

- Machine language is a language that a microprocessor actually understands.
- Machine language is a sequence of instruction written in the form of binary numbers consisting of 1's and 0's to which the microprocessor responds directly.
- The main advantage of machine language is that they execute fast as the computer understands them and does not need translation.
- The disadvantages of machine languages are
  - → Difficult to use
  - → Machine dependent
  - → Difficult to debug and modify

→ **Assembly language**

Assembly language is a symbolic representation (called Mnemonics) of machine code. They are close to machine code but the computer can not understand them. The assembly language program must be translated into machine code.

by a separate program called an assembler.
The assembler program recognizes the character strings that make up the symbolic names of machine operations and substitutes the required machine code for each instruction.

eg:- ADD
SUB
INR

- The advantage of Assembly language is
  → It is relatively easy for writing programs.
  → It is useful in writing program in embedded system.
- The disadvantages of Assembly language are
  → Hard to remember mnemonics
  → Machine Dependent
  → Less efficient than machine language.

• High Level Language
A high level language is a programming language that is more user friendly, to some extent platform independent and abstracts from low level processor operations.
They are similar to natural languages and so are easy to write and remember.

- Advantages are
  → Easy to use
  → Portability
  → Easy to debug
  → Easy and fast development of software.
- Disadvantages are
  → More execution time (slow in execution)
  → Needs own translator

Compiler and Interpreter:
• High level source program must be translated first into a form the machine can understand. This is done by the software called compiler. The compiler takes the source code as input and produces the machine language code (object code) for the machine on

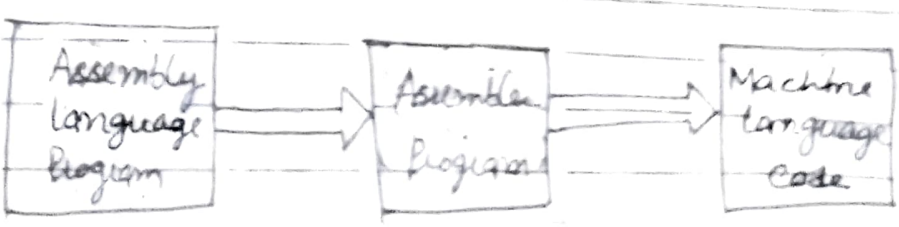
which it is to be executed as output.
During the process of translation, the compiler reads the source programs statements and checks for syntax errors. In case of any error the ~~computer~~ compiler generates message about error.
• An interpreter, is also translator which translates high level language code into a machine level language. The interpreter translates and executes the program line by line. Each time the program is executed every line is checked for syntax error and then convert it into equivalent machine code.

Assembly Language Programming

• Programs in assembly language are represented by instructions that use English language type commands called mnemonics. It is more convenient to write the programs in assembly than in machine language.
• A translator (Assembler) must be used to convert the assembly language program into binary machine language programs so that the processor can execute them.
• Assembly language program is machine dependent.

Tools for developing Assembly Language Programs

The program development utilities enable the user to write, assemble and test assembly language programs; they include
→ Editor → Assembler → Linker → Debugger
→ Editor: It allows the user to enter, modify and store a group of instructions under a filename. The editor programs can be line editors or full screen editors.
→ Assembler: It is a program that translates source code into the binary code called object code and generates a file called object file.



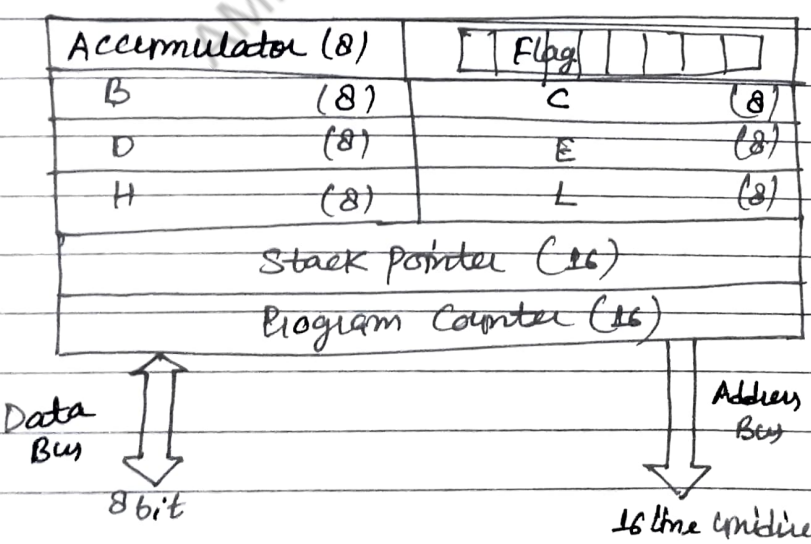
→ Loader (Linker): The loader is a program that takes the object file generated by assembler program and generates a file in binary code called the EXE file.

→ Debugger: It is a program that allows the user to test and debug the object file. The user can employ this program to perform the following functions:

- Make changes in the object code.
- Examine and modify the contents of memory.
- Set breakpoints, execute a segment of the program and display register contents after the execution.
- Trace the execution of the specified segment of the program and display the register and memory contents after the execution of each instruction.
- Disassemble a section of the program.

8085 Programming Model

The 8085 programming model includes six registers, one accumulator, and one flag register. In addition it has two 16-bit registers stack pointer and program counter.



→ Registers: 8085 has six general purpose registers to store 8 bit data B, C, D, E, H, L. They can be combined as register pair BC, DE, HL to perform 16 bit operations. The programmer can use these registers.

→ Accumulator: It is an 8-bit register that is a part of ALU. This register is used to store 8-bit data to perform ALU operations. The result of operation is stored in accumulator.

→ Flag: The ALU includes five flags which are set or reset after operation according to data conditions of the result in accumulator and other register. The flags used are Sign (S), zero (Z), Auxiliary Carry (AC), Parity (P), Carry (CY) flags.

→ Program Counter: 16-bit register deals with sequencing the execution of instructions. This register is a pointer to memory.
It holds the address of next instruction to be executed.

→ Stack Pointer: 16-bit register used as a memory pointer. It points to a memory location in R/W memory called the stack. The beginning of the stack is defined by loading 16-bit address in the stack pointer.

Instruction Set of 8085:

- The 8085 has 74 operation codes. It consists of 246 instructions in total.
- 8085 instructions can be classified as:

1. Data Transfer/Copy: These instructions move data between registers or memory and registers from source to destination.
2. Arithmetic instructions: These instructions perform the operations like, addition, subtraction, increment, decrement, etc.
3. Logical and bit manipulation instructions: Logical operations like AND, OR, XOR, complement are performed. Additionally Rotate, shift operation can also be performed.
4. Branching operations: These operations are used to control the flow of program execution.
5. Machine Control Instruction: Includes the instructions for maintaining the stack reading/writing ports, setting and reading interrupt mask and setting/clearing flags.

Addressing Modes of 8085

- 8085 has five addressing modes

→ Immediate addressing: In this mode 8 or 16 bit data is specified in the instruction itself as its one of the operand.

eg: MVI A, 12H

→ Register addressing: In this mode operands are in microprocessor registers only.

eg: MOV A, B

→ Direct addressing: In this mode one operand is data stored in memory. The memory address of operand is directly given in instruction.

eg: STA 2500H

→ Indirect addressing: In this mode, one of the operands is data stored in memory. The memory address of operand specified by register pair.

eg: LDAX D

→ Implied addressing: This mode the data is specified by opcode itself.

eg: CMA.

WRITING ASSEMBLY LANGUAGE PROGRAMS

- To write a program, one needs to draw up a plan of logical thoughts. A given task should be broken down into small units that can be built independently. This is called the modular design approach.
- To write a program is equivalent to give specific command to the microprocessor in a sequence of to perform a task.

There are following steps to write a program:

Step 1: Read the problem carefully.
Step 2: Break it down into small steps.
Step 3: Represent these small steps in a possible sequence with a flow chart.
Step 4: Translate each block of flowchart into appropriate mnemonic instructions.
Step 5: Translate mnemonics into machine code.
Step 6: Enter the machine code in memory and execute.
Step 7: Start troubleshooting.

Example:

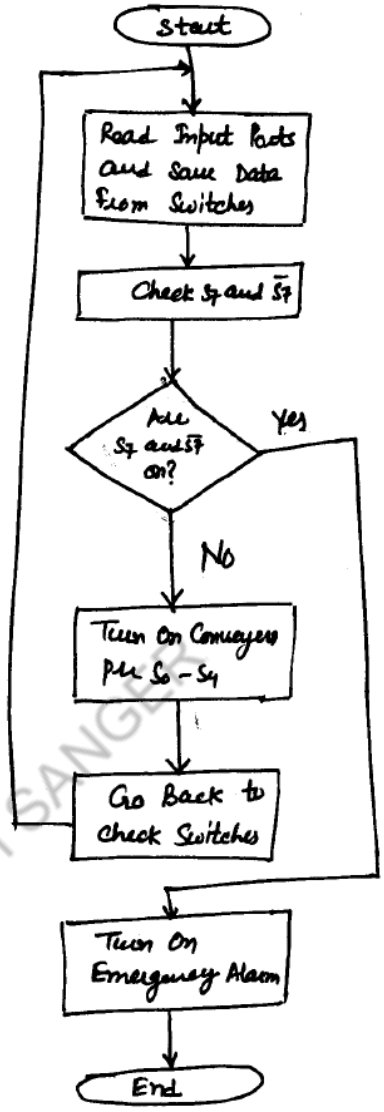
A microcomputer is designed to monitor various processes on the floor of a manufacturing plant, The microcomputer has two input ports with the addresses F1H and F2H and an output port with the address F3H. Input port F1H has six switches, five of which ($D_0 - D_4$) control the conveyer belts through the output port F3H. Switch S_7, corresponding to the data line D_7, is reserved to indicate an emergency on the floor. As a precautionary measure, input port F2H is controlled by the foreman, and its switch, $\overline{S_7}$, is also used to indicate an emergency. Output line D_6 of port F3H is connected to the emergency alarm. Write a program to

(i.) turn on the five conveyer belts according to the ON/OFF positions of switches $S_4 - S_0$ at port F1H.

(ii.) turn off the conveyer belt and turn on the emergency alarm only when both switches - S_7 and $\overline{S_7}$ are triggered.

(iii.) monitor the switches continuously.

```
START:   IN F1H
         MOV B,A
         IN F2H
         ANI 80H
         MOV C,A
         MOV A,B
         ANI 80H
         ANA C
         JNZ SHTDWN
         MOV A,B
         ANI 1FH
         OUT F3H
         JMP START
SHTDWN:  MVI A,40H
         OUT F3H
         HLT
```



- Debugging a program

- Debugging a program is similar to troubleshooting hardware.
- The debugging process can be divided into two parts:
  1. Static Debugging 2. Dynamic Debugging.

1. Static debugging: It is similar to visual inspection of a circuit board; It is done by a paper-and-pencil check of a flowchart and machine code.

2. Dynamic debugging: It involves observing the output, or register contents, following the execution of each instruction or of a group of instructions.

- What is Assembler?
An assembler is a program that translates the mnemonics into their machine code. It is generally not available on a single-board microcomputer.

- What is monitor program for keys?
The key monitor program is a set of instructions that continuously checks whether a key is pressed and stores the binary equivalent of a pressed key in a memory location.

- DATA TRANSFER OPERATIONS

- One of the primary functions of the microprocessor is copying data from one register to another.
- The copying function is labeled as the data transfer functions.
- Data transfer instructions do not affect the flags.
- Instructions used for data transfer operation are as follows:

→ **MOV Rd, Rs**: This is a 1-byte instruction, copies data from source register to destination register.

→ **MVI R, 8-bit**: This is a 2-byte instruction, loads 8 bits of second byte into the register specified.

→ **OUT 8-bit port address**: This is a 2-byte instruction, sends the content of accumulator (A) to the output port specified in second byte.

→ **IN 8-bit port address**: This is a 2-byte instruction, accepts data from the input port specified in second byte and loads into the accumulator

→ **HLT**: This is a 1-byte instruction, when this instruction is executed the processor stops executing and enters wait state, and the address and data bus are placed in high impedance state, No register contents are affected.

→ **NOP**: This is a 1-byte instruction and No operation is performed. It is used to increase processing time as substitute in place of an instruction.

NOTE: Rd, Rs, R are used to represent 8-bit register A, B, C, D, E, H, L

Examples:

1. Load accumulator A with the data byte 82_H and save the data in register B.

```
MVI A, 82H
MOV B, A
HLT
```

2. Write instructions to read eight ON/OFF switches connected to the input port with the address 00H and turn on the devices connected to the output port with the address 01_H.

```
IN 00H
OUT 01H
HLT
```

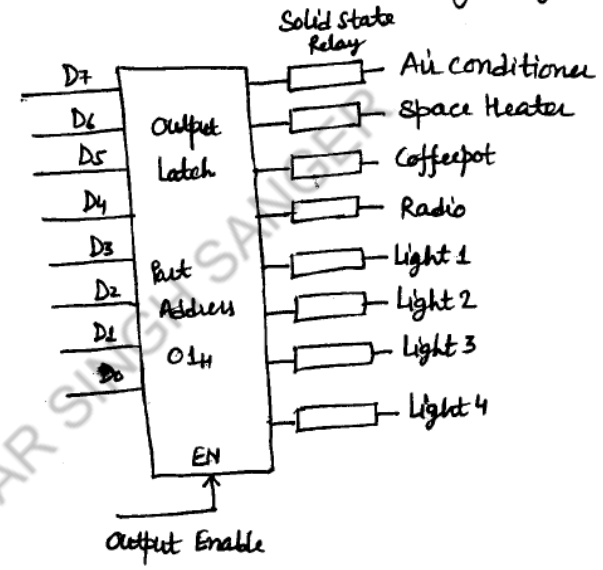
3. Load the Hexadecimal number 37_H in register B, and display the number at the output port labeled PORT 1.

```
MVI  B, 37H
MOV  A, B
OUT  PORT1
HLT
```

4. A microcomputer is designed to control various appliances and lights in your house. The system has an output port with the address 01_H, and various units are connected to the bits D_7 to D_0 as shown in figure. On a cool morning you want to turn on the radio, the coffeepot, and the space heater, write appropriate instructions for the microcomputer. Assume the R/W memory in your system begins at 2000_H.

$D_7\ D_6\ D_5\ D_4\ D_3\ D_2\ D_1\ D_0$
$0\ 1\ 1\ 1\ \ 0\ 0\ 0\ 0 = 70_H$

```
MVI A, 70H
OUT 01H
HLT
```



- ARITHMETIC OPERATIONS

- 8085 performs various arithmetic operations, such as addition, subtraction, increment and decrement.
- Instructions used for arithmetic operations are:
  - → **ADD R**: This is a 1-byte instruction and adds the contents of R to the accumulator.
  - → **ADI 8-bit**: Add immediate. This is a 2-byte instruction and adds the second byte to the contents of the accumulator.
  - → **SUB R**: This is a 1-byte instruction. Subtracts the contents of R from the contents of the accumulator.
  - → **SUI 8-bit**: Subtract immediate. This is a 2-byte instruction and subtracts the second byte from the the contents of the accumulator.
  - → **INR R**: This is 1-byte instruction and increases the contents of register R by 1. This instruction affects all the flags except CY.
  - → **DCR R**: This is 1-byte instruction and decreases the contents of R by 1. This instruction affects all the flags except CY.

NOTE: R is a register.

- Some important points regarding the arithmetic instruction are as:
  - → For using ADD, ADI, SUB, SUI.
    - Above four instructions assume implicitly that accumulator is one of the operand.
    - Above four instructions modify all the flags according to the data conditions of the result.
    - The result of all four instructions is placed in accumulator.
    - All the four instructions do not affect the contents of the operand register.
  - → For INR and DCR.
    - Both instructions affect the contents of the specified register
    - Both instructions affects all the flags except the CY flag.

Examples

1. The contents of the accumulator are 93_H and the contents of register C are $B7_H$. Add both contents.

A [93H] C [B7H]

ADD C

	CY	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
93_H =		1	0	0	1	0	0	1	1
$B7_H$ =		1	0	1	1	0	1	1	1
	1	0	1	0	0	1	0	1	0

CY [1] A [4AH]

```
MVI A, 93H
MVI C, B7H
ADD C
HLT
```

Flags: S=0, Z=0, CY=1

2. The contents of accumulator are $4A_H$ and CY is set. Add 35_H directly to the accumulator.

ADI 35_H

CY	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	0	1	0	0	1	0	1	0
	0	0	1	1	0	1	0	1
[0]	0	0	1	1	1	1	1	1

CY A

Flag status: S=0, Z=0, CY=0

3. Assume the accumulator holds the data byte FF_H. Illustrate the differences in the flags status

(i) By adding 01_H, and

(ii) By incrementing the accumulator contents.

(i) ADI 01_H

	CY	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
A = FF_H =		1	1	1	1	1	1	1	1
Data = 01_H		0	0	0	0	0	0	0	1
	[1]	0	0	0	0	0	0	0	0

CY A

Flags: S=0, Z=1, CY=1

(ii) INR A

A: FF_H = CY [0] A [1 1 1 1 1 1 1 1]

After INR A [0 0 0 0 0 0 0 0]

A

Flags: S=0, Z=1, CY=NA

4. Write a program to perform the following functions.
(i.) Load the number 8BH in register D.
(ii.) Load the number 6FH in register C.
(iii.) Increment the contents of register C by one.
(iv.) Add the contents of registers C and D and display the sum at output PORT1.

```
MVI  D, 8BH
MVI  C, 6FH
INR  C
MOV  A, C
ADD  D
OUT  PORT1
HLT
```

Flags : S = 1, Z = 0, CY = 0

```
  1 0 0 0 1 0 1 1
+ 0 1 1 0 1 1 1 1
-----------------
[0] 1 1 1 1 0 1 0
 CY  S
```

- Examples of subtraction

NOTE: 8085 performs subtraction by using the method of 2's complement.

1. Register B has 65H and the accumulator has 97H. Subtract the contents of register B from the contents of the accumulator.

```
MVI B, 65H
MVI A, 97H
SUB B
HLT
```

(B) : 65H = 0 1 1 0 0 1 0 1
1's Complement of 65H = 1 0 0 1 1 0 1 0
2's complement of 65H = + 1
1 0 0 1 1 0 1 1

(A) : 97H = 1 0 0 1 0 1 1 1
2's Comp. of 65H = 1 0 0 1 1 0 1 1
+

1 (CY) | 0 0 1 1 (3) | 0 0 1 0 (2)

Complement the carry, 0. The result is
(A) : 32H

Flags : S = 0, Z = 0, CY = 0

2. Write a program to do the following :
(i.) load the number 30H in register B and 39H in register C.
(ii.) Subtract 39H from 30H.
(iii.) Display output at PORT1.

```
MVI B, 30H
MVI C, 39H
MOV A, B
SUB C
OUT PORT1
HLT
```

Flags : S = 1, Z = 0, CY = 1

* The result F7H is the 2's complement of 09H. The PORT1 displays F7H.

- LOGIC OPERATIONS

- A microprocessor is basically a programmable logic chip. It can perform all the logic functions of the hard-wired logic through its instruction set. The 8085 instruction set includes such logic functions as AND, OR, Ex OR and NOT.
- The instructions of these logic operations are described as:
  - → **ANA R**: This is a 1-byte instruction and logically ANDs the contents of the register R with the contents of the accumulator.
    Flags CY is reset and AC is set.
  - → **ANI 8bit**: This is AND immediate with accumulator. This is a 2-byte instruction. This logically ANDs the second byte with the accumulator contents.
    Flags CY is reset and AC is set.
  - → **ORA R**: This is 1-byte instruction and logically OR the contents of R with the accumulator.
  - → **ORI 8bit**: This is OR immediate with accumulator. This is a 2-byte instruction and logically ORs the second byte with the contents of the accumulator.
  - → **XRA R**: This is a 1-byte byte instruction and Exclusive-ORs the contents of ~~cont~~ R with the accumulator.
  - → **XRI 8bit**: This is Exclusive-OR immediate with accumulator. This is a 2-byte instruction and Exclusive-ORs the second byte with the contents of the accumulator.
  - → **CMA**: This is complement accumulator. This is a 1-byte instruction that complements the contents of accumulator. And this affects no flags.
- Some important points are,
  - → Logic instructions implicitly assume that the accumulator is one of the operands.
  - → Reset the CY flag. CMA instruction is exception and does not affect any flags.
  - → Logic instructions modify Z, P, S flags according to the data conditions of result
  - → Logic instructions place the result in the accumulator.
  - → Logic instructions do not affect the contents of operand register.

- Example

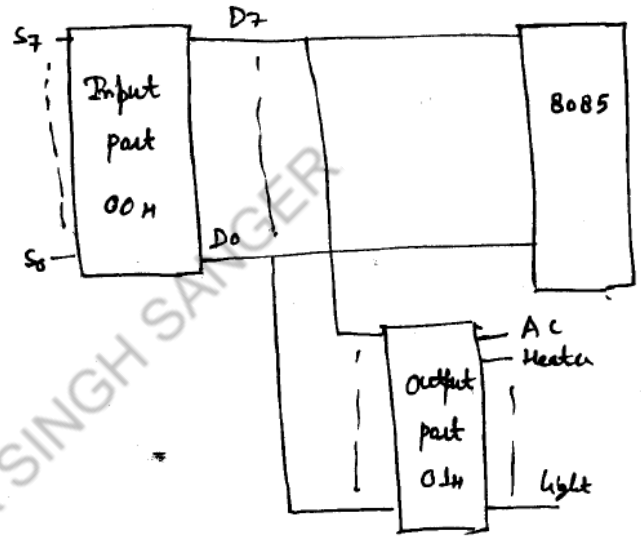
1. (Data Masking with logic AND) To conserve energy and to avoid an electrical overload on a hot afternoon, implement the following procedures to control the appliances throughout the house. Assume that the control switches are located in the kitchen, and they are available to anyone in the house. Write a set of instructions to

(i.) turn on the air conditioner if switch S_7 of the input port 00_H is on.

(ii.) ignore all other switches of the input port even if someone attempts to turn on other appliances.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	
1	0	0	0	0	0	0	0	$= 80_H$

```
MVI  A, Data
ANI  80H
OUT  01H
HLT
```



2. Assume register B holds 93_H and the accumulator holds 15_H. Show the results of the instructions ORA B, XRA B, and CMA.

(i.) ORA B,

$$\begin{array}{rl} & (B) = 1001\ 0011 \\ \text{OR} & (A) = 0001\ 0101 \\ \hline & (A) = 1001\ 0111 \quad (97_H) \end{array}$$

Flags : S = 1, Z = 0, CY = 0

(ii.) XRA B,

$$\begin{array}{rl} & (B) = 1001\ 0011 \\ \text{XOR} & (A) = 0001\ 0101 \\ \hline & (A) = 1000\ 0110 \quad (86_H) \end{array}$$

Flags : S = 1, Z = 0, CY = 0

(iii.) CMA,

$$(A) = 0001\ 0101$$

$$\text{CMA } (A) = 1110\ 1010 \ (EA_H)$$

3. Two input ports with eight switches are connected to the microcomputer having address 00_H and 01_H respectively, and control the same appliances. Write instruction to turn on devices from the any input port.

```
IN 00H
MOV B, A
IN 01H
ORA B
OUT PORT1
HLT
```

- BRANCH OPERATIONS

- Branch instructions are the most powerful instructions because they allow the microprocessor to change the sequence of a program, either unconditionally or under certain test conditions
- Branch instructions are classified in three categories:
    1. Jump instructions
    2. Call and Return instructions
    3. Restart instructions.
- Jump instructions specify the memory location explicitly. They are 3-byte instructions:
    - One byte for operation code
    - Two bytes for memory address
- Jump instructions are classified into two categories:
  (i) Unconditional Jump (ii) Conditional Jump

(i) Unconditional Jump: The unconditional Jump instruction enables the programmer to set up continuous loops.

- **JMP 16-bit**: This is a three byte instruction. The second byte specifies the low order and third byte specifies the high order address.

Example: Write a program to read the switch position continuously and turn on the appliances accordingly.

```
START: IN 00H
       OUT 01H
       JMP START
```

(ii) Conditional Jump: - These instructions allow the microprocessor to make decision based on certain conditions indicated by the flags.
- The 8085 flag register has five flags. The four flags used by Jump instruction, which are given below
    1. Carry flag
    2. Zero flag
    3. Sign flag
    4. Parity flag
- Two Jump instructions are associated with each flag i.e. present or absent.

→ JC 16-bit : Jump on carry

→ JNC 16-bit : Jump on No carry

→ JZ 16-bit : Jump on Zero

→ JNZ 16-bit : Jump on No Zero

→ JP 16-bit : Jump on Plus

→ JM 16-bit : Jump on Minus

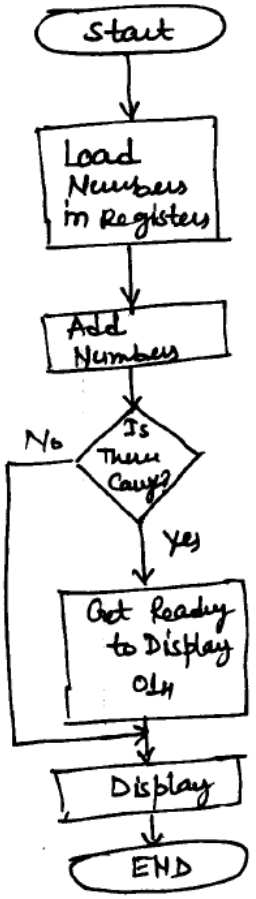
→ JPE 16-bit : Jump on Even Parity

→ JPO 16-bit : Jump on odd Parity

Example:

Load a HEX number 9BH and A7H in registers D and E respectively, and add the numbers. If the sum is greater than FFH display 01H at output PORT0; Otherwise display the sum.

```
        MVI D, 9BH
        MVI E, A7H
        MOV A, D
        ADD E
        JNC DISPLAY
        MVI A, 01H
DISPLAY: OUT 00H
        HLT
```



- PROGRAMMING TECHNIQUES : LOOPING, COUNTING AND INDEXING

- Programming techniques 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.
- Techniques such as counting and indexing are used in setting up a loop.
- Loops can be ~~as~~ classified into two groups:
  - → Continuous loop : repeats a task continuously
  - → Conditional loop : repeats a task until certain data conditions are met.

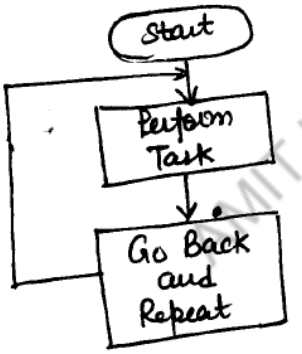
→ Continuous Loop:- It is set up by using the unconditional Jump instruction.

- A program with continuous loop does not stop repeating the task until the system is reset.

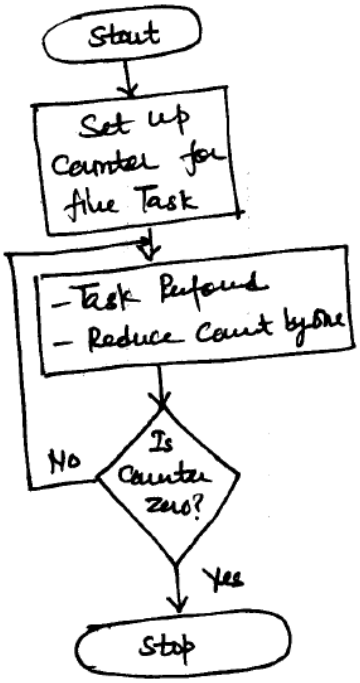
eg: continuous counter

→ Conditional loop:- A conditional loop is set up by the conditional Jump instructions. These instructions check flags and repeat the specified tasks if the conditions are satisfied. These loops usually include counting and indexing.

eg: Microprocessor repeats a task five times.



Continuous loop:-

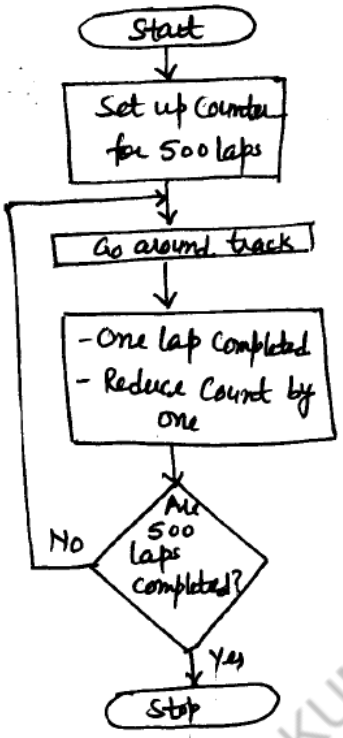


Task Repetition

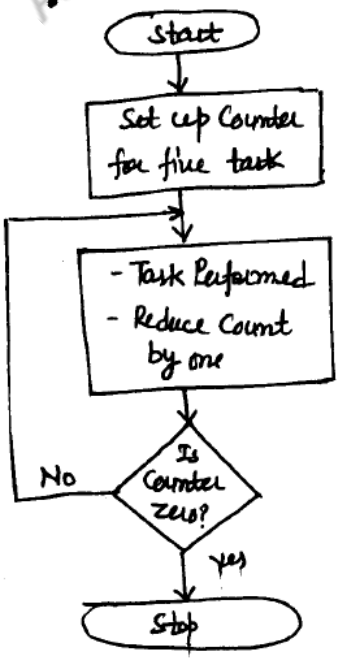
- Conditional loop, counter and indexing

- Another type of loop includes indexing along with a counter.
- Indexing means pointing or referencing objects with sequential numbers and are referred to by numbers. This is called indexing.

eg:- 1. The process of a car racer in the Indy 500 going around the track 500 times.



2. Microprocessor needs a to repeat the task five times.



Examples of Conditional loop with counter.

ADDITIONAL DATA TRANSFER AND 16-bit ARITHMETIC INSTRUCTIONS

- 16 bit Data transfer to Register pairs

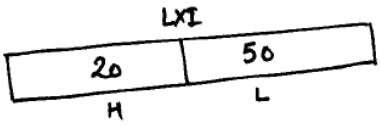
* The LXI instructions perform load operation of 16-bit data in register pairs and the stack pointer register.
* This instruction does not affect the flags.

- `LXI Rp, 16-bit` : Load Register pair, 3 byte instruction.
- `LXI SP, 16-bit`

* Rp represents Register pair, BC, DE, HL denoted by B, D, H.

eg: Write instruction to load the 16-bit number 2050H in register pair HL.

```
LXI H, 2050H
```



- Data transfer (copy) from Memory to the Microprocessor

* 8085 instruction set includes three types of memory transfer instructions: Two use the indirect addressing mode and one uses the direct addressing mode.
* These instructions do not affect the flags.

(i.) `MOV R, M` : Move from Memory to Register; 1-byte instruction;

It copies the data byte from memory location into a register.

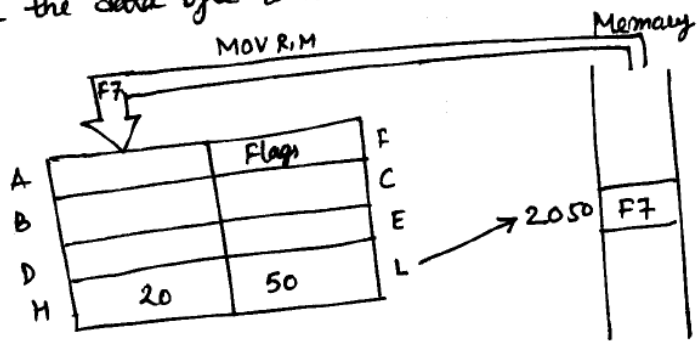
R ⇒ microprocessor registers. A, B, C, D, E, H and L.

M ⇒ memory location specified by the contents of HL register.

As the memory location specification is ~~indirect~~ indirect, it is called the indirect addressing mode.

(ii) eg: The memory location 2050H holds the data byte F7H. Write instructions to transfer the data byte to accumulator.

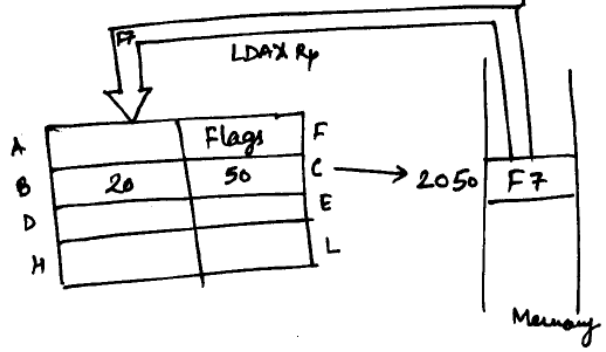
```
LXI H, 2050H
MOV A, M
HLT
```



(ii.) **LDAX B/D** : Load Accumulator indirect; 1-byte instruction; It copies data byte from memory location into the accumulator; the memory location is specified by the contents of registers BC or DE.

eg:- Write instruction to transfer data byte to the accumulator from memory location 2050H.

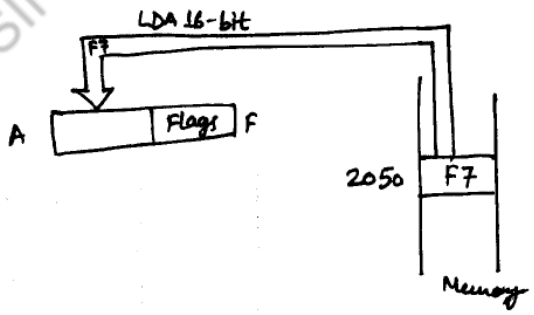
```
LXI  B,2050H
LDAX B
HLT
```



(iii.) **LDA 16-bit** : Load accumulator direct; 3-byte instruction; It copies the data byte from memory location specified by 16-bit address. The second byte is a line number and third byte is a page number. The addressing mode is direct.

eg:- Write instruction to transfer data byte to the accumulator from memory location 2050H.

```
LDA 2050H
HLT
```



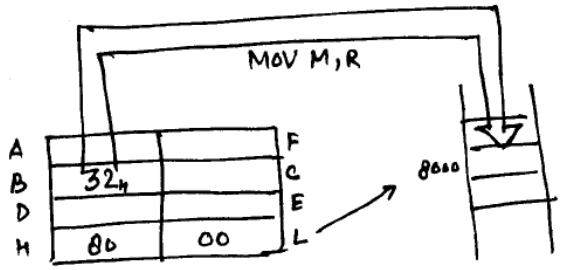
- Data transfer (copy) from the Microprocessor to Memory or Directly into Memory.

* The instructions for copying data from the microprocessor to a memory location.

1. **MOV M,R** : 1byte instruction that copies data from a register, R into the memory location specified by the contents of HL registers.

eg:- Register B contains 32H. Write instructions to copy the contents of B into memory location 8000H using indirect addressing.

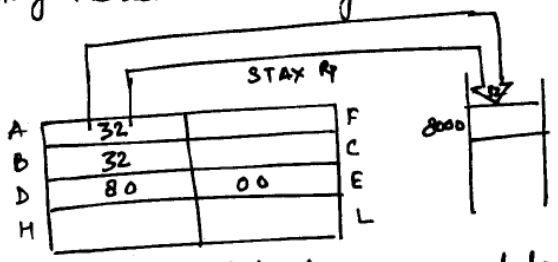
```
LXI H, 8000H
MOV M, B
HLT
```



2. **STAX B/D** : 1-byte instruction that copies data from accumulator to memory location specified by the contents of either BC or DE registers.

eg: Register B contains 32H. Write instruction to copy contents of ~~accumulator~~ B into memory location 8000H using indirect addressing.

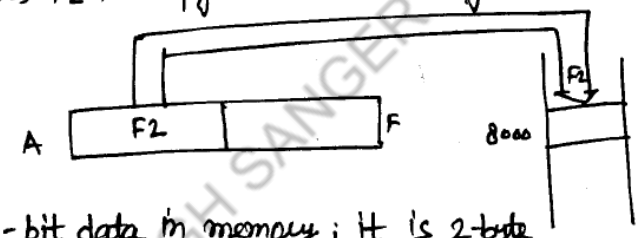
```
LXI  D, 8000H
MOV  A, B
STAX D
HLT
```



3. **STA 16-bit** : 3-byte instruction that copies data from accumulator into the memory location specified by the 16-bit operand.

eg: The accumulator contains F_2H. Copy A into memory location 8000_H using direct addressing.

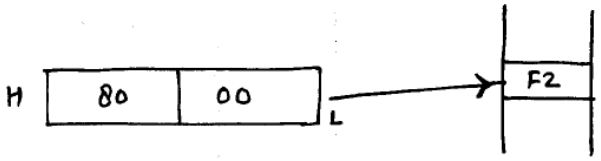
```
STA 8000H
HLT
```



4. **MVI M, 8-bit** : It loads 8-bit data in memory; it is 2-byte instruction, the second byte specifies 8-bit data. The memory location is specified by the contents of the HL registers.

eg: Load $F2_H$ directly in memory location 8000_H using indirect addressing.

```
LXI H, 8000H
MVI M, F2H
HLT
```



- Arithmetic Operations Related to 16-bits or Registers Pairs

- instructions related to increment / decrement 16-bit contents in a register pair are included and these instructions do not affect the flags.

1. **INX Rp** : This is 1-byte instruction and used to increment Register pair. It treats the contents of two register as one and increases the contents by 1.

Rp ⇒ Register Pair. eg:- INX B, INX D, INX H, INX SP

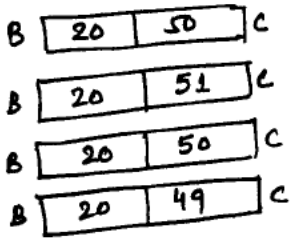
2. **DCX Rp** : This is 1-byte instruction and decrements the contents of register pair.

Rp ⇒ Register Pair. eg:-

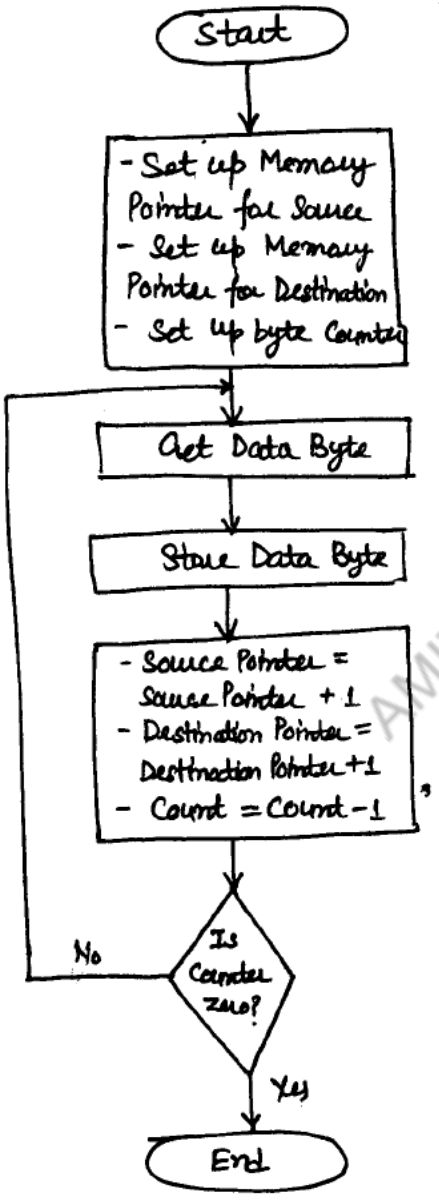
```
DCX B
DCX D
DCX H
DCX SP
```

eg:- Write the instruction to load the number 2050H in register pair BC. First increment one and then decrement twice the contents of BC pair.

```
LXI  B, 2050H
INX  B
DCX  B
DCX  B
```



eg:- Write instructions to transfer a block of data bytes.




```
        LXI H, 2050H
        LXI D, 2A50H
        MVI B, 10H
NEXT:   MOV A, M
        STAX D
        INX H
        INX D
        DCR B
        JNZ NEXT
        HLT
```

– ARITHMETIC OPERATIONS RELATED TO MEMORY

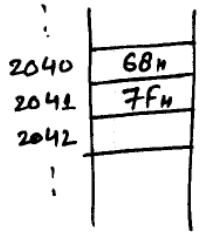
- These instructions perform two tasks: one is to copy a byte from memory location to the microprocessor and other is to perform the arithmetic operation.
- Accumulator (A) is assumed as an operand implicitly.
- After operation the previous contents of the accumulator are replaced by the result.
- All flags are modified accordingly.

- **ADD M**: 1-byte instruction, adds (M) to (A) and stores the result in (A). The memory location is specified by the contents of HL register.
- **SUB M**: 1-byte instruction, subtracts (M) from (A) and stores the ~~result~~ result in A. The memory location is specified by the contents of HL register.
- **INR M**: 1-byte instruction that increments the contents of a memory location by 1 not the memory address. The memory address is specified by HL. This instruction modifies all the flags except the Carry flag.
- **DCR M**: 1-byte instruction that decreases (M) by 1; the memory location is specified by HL. This instruction modifies all the flags except the Carry flag.

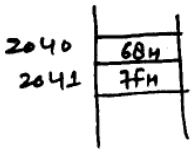
Example

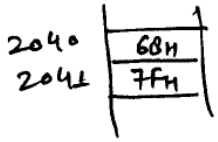
- Write instructions to add the contents of the memory location 2040H to (A) and subtract the contents of the memory location 2041 from the first sum. Assume that the accumulator has 30_H, the memory location 2040H has 68_H, and location 2041H has 7FH.

```
LXI  H, 2040H
ADD  M
INX  H
SUB  M
HLT
```



After ADDM





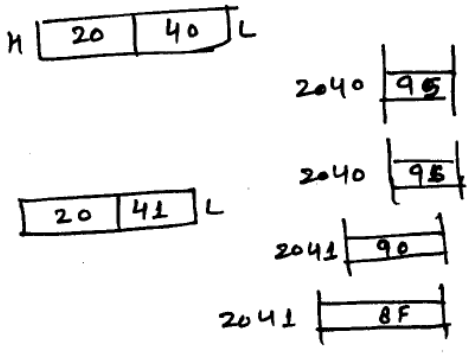
After SUB M

- Write instruction to

-(1.) Load 95_H in memory location 2040_H and increment the contents of the memory location.

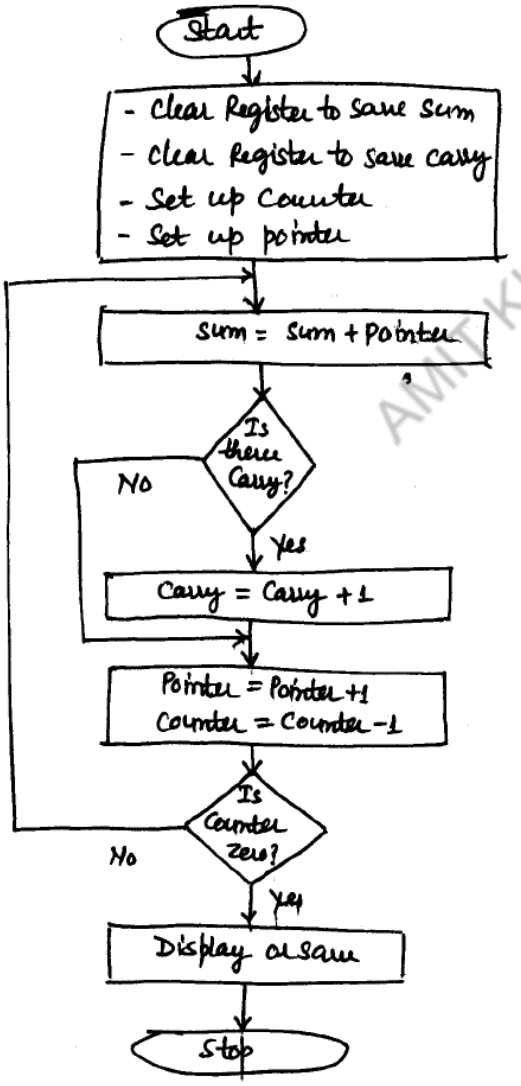
(2.) load 90_H in memory location 2041_H and decrement the contents of the memory location.

```
LXI  H, 2040H   →
MVI  M, 95H     →
INR  M          →
INX  H          →
MVI  M, 90H     →
DCR  M          →
HLT             →
```



- Addition with carry: Six bytes of data are stored in memory locations starting at 2050_H. Add all the data bytes. Use register B to save any carries ~~generation~~ generated, while adding the data bytes. Store the result at 2070_H and 2071_H.

Data(H) A2, FA, DF, E5, 98, 8B




```
        XRA  A
        MOV  B, A
        MVI  C, 06H
        LXI  H, 2050H
NXTBYT: ADD  M
        JNC  NXTMEM
        INR  B
NXTMEM: INX  H
        DCR  C
        JNZ  NXTBYT
        LXI  H, 2070H
        MOV  M, A
        INX  H
        MOV  M, B
        HLT
```

LOGIC OPERATIONS

– ROTATE

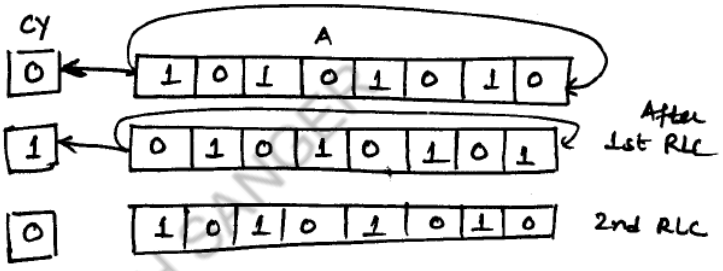
This group has four operations; two are for rotating left and two are for rotating right.

(1.) **RLC** : Rotate Accumulator Left

During this operation each bit is shifted to the adjacent Left position. Bit D_7 becomes D_0. The carry flag CY is modified according to bit D_7.

eg: Assume the contents of accumulator are AAH and CY = 0. Show the accumulator contents after the execution of RLC instruction twice.

```
MVI A, AAH
RLC
RLC
HLT
```

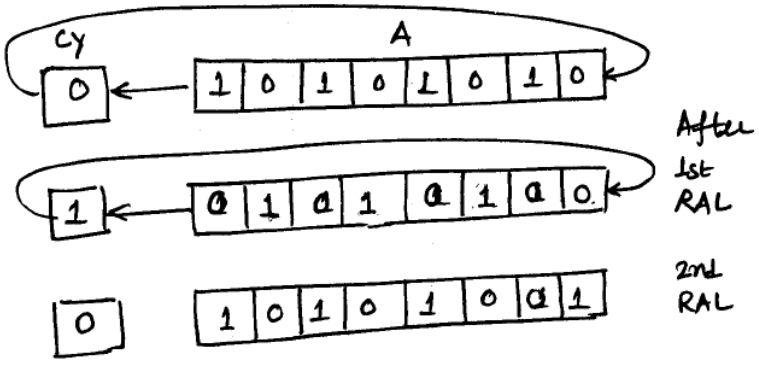


(2.) **RAL** : Rotate Accumulator through Carry

This instruction shifts each bit to its adjacent left position and D_7 becomes the carry bit and carry bit is shifted into D_0.

eg: Assume the accumulator contents are AAH and CY = 0. Show the accumulator contents after execution of RAL twice.

```
MVI A, AAH
RAL
RAL
HLT
```

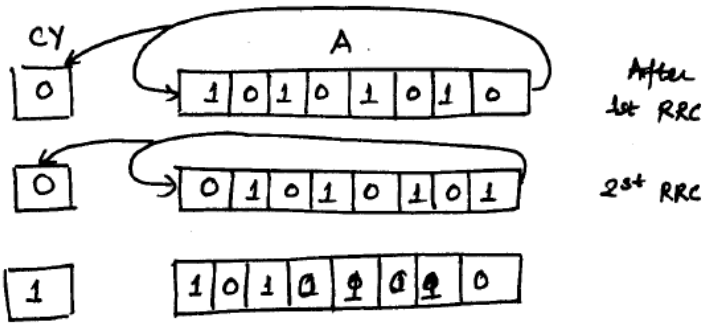


(3.) **RRC** : Rotate Accumulator Right

Each bit is shifted right to the adjacent position. Bit D_0 becomes D_7 and the Carry flag is modified according to bit D_0.

eg:- Assume the accumulator contents are AA$_H$ and CY = 0. Show the accumulator contents after the execution of the ~~RAL too~~ RRC.

```
MVI A, AAH
RRC
RRC
HLT
```

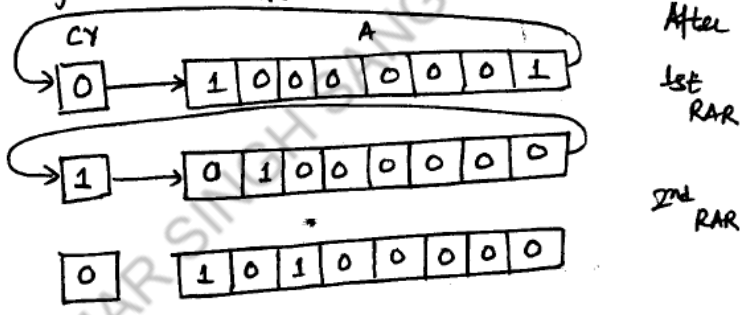


(4.) RAR : Rotate Accumulator Right Through carry

Each bit is shifted right to the adjacent position. Bit D_0 becomes the carry bit and the carry bit is shifted into D_7.

eg:- Assume the contents of the accumulator are 81$_H$ and CY = 0. Show the accumulator contents after two RAR.

```
MVI A, 81H
RAR
RAR
HLT
```



- Checking Sign with Rotate instruction: A set of ten current readings is ~~stort~~ stored in memory locations starting at 2060$_H$. The reading are expected to be positive (< 127_{10}). Write a program to

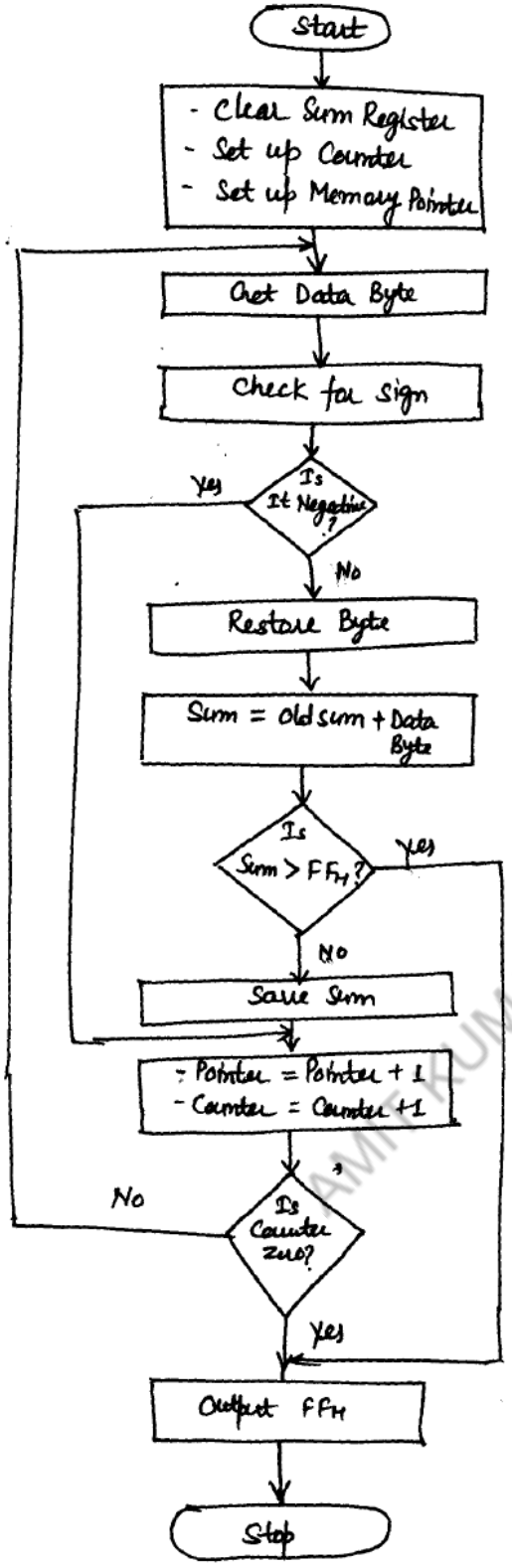
(i.) check each reading to determine whether it is positive or negative.

(ii.) reject all negative readings.

(iii.) add all positive readings.

(iv.) ~~Output FF$_H$ to~~ Store FF$_H$ in memory location 2070$_H$ when sum exceeds eight bits; otherwise store the sum.

Data (H) 28, D8, C2, 21, 24, 30, 2F, 19, F2, 9F




```
        MVI  B,00H
        MVI  C,0AH
        LXI  H,2060H
NEXT:   MOV  A,M
        RAL
        JC   REJECT
        RAR
        ADD  B
        JC   OVRLOD
        MOV  B,A
REJECT: INX  H
        DCR  C
        JNZ  NEXT
        STA  2070H
OVRLOD: MVI  A,FFH
        STA  2070H
        HLT
```


- APPLICATIONS OF ROTATE INSTRUCTIONS

The rotate instructions are primarily used in arithmetic multiply and divide operations and for serial transfer.

– COMPARE

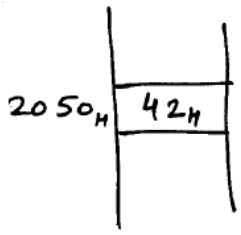
- 8085 instruction set has two types of compare operations:
  - CMP : Compare with accumulator
  - CPI : Compare Immediate (with accumulator)

- **CMP R/M** :
  - * 1-byte instruction that compares (Register or Memory) with Accumulator.
  - * If (A) < (R/M), CY is set and Zero (Z) is reset.
    (A) = (R/M), Z is set and CY is reset.
    (A) > (R/M), CY and Z are reset.
  - * When memory is an operand its address is specified by (HL).
  - * No contents are modified
  - * All remaining flags (S, P, AC) are affected according to the result of subtraction.

- **CPI 8-bit** :
  - * 2-byte instruction that compares second byte with (A).
  - * If (A) < 8-bit data, CY is set and Z is reset.
    (A) = 8-bit data, Z is set and CY is reset.
    (A) > 8-bit data, CY and Z are reset.
  - * No contents are modified
  - * All remaining flags (S, P, AC) are affected according to the result of Subtraction.

eg: Write an instruction to load the accumulator with the data byte 64_H, and verify whether the data byte in memory location 2050_H is equal to the accumulator contents.

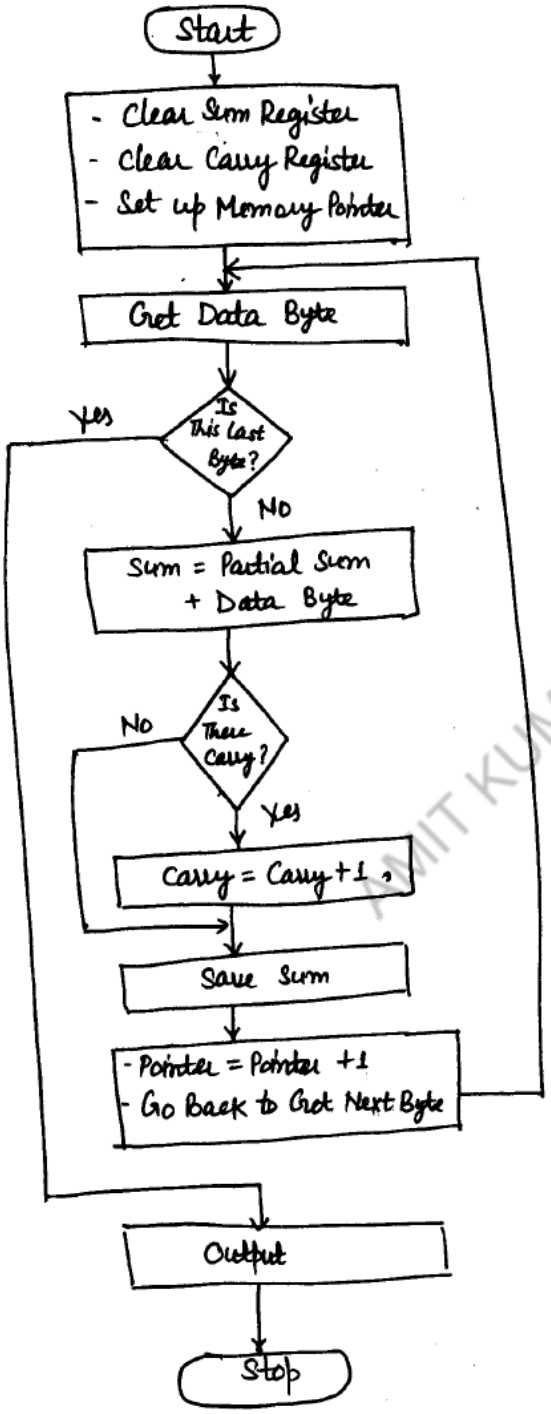
```
LXI H, 2050H
MVI A, 64H
CMP M
HLT
```



- Use of compare instruction to indicate End of Data String:

A set of current readings is stored in memory locations starting at 2050_H. The end of data string is indicated by the data byte 00_H. Add the set of readings. Store the result in memory location $200A_H$ and $200B_H$.

Data (H)	32, 52, F2, A5, 00




```
        LXI H, 2050H
        MVI C, 00H
        MOV B, C
NXTBYT: MOV A, M
        CPI 00H
        JZ  DISPLAY
        ADD C
        JNC SAVE
        INR B
SAVE :  MOV C, A
        INX H
        JMP NXTBYT
DISPLAY: ~~MOV~~ LXI H, 200AH
        MOV M, C
        INX H
        MOV M, B
        HLT
```

- A set of three readings is stored in memory starting at 2050_H. Sort the readings in ascending order.

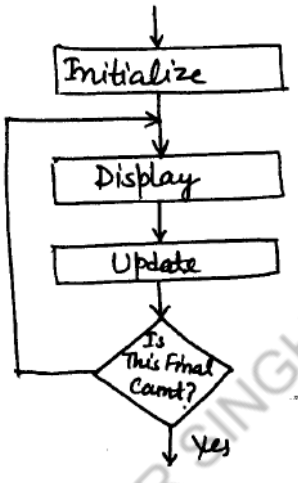
Data (H)	87, 56, 42

```
START:  LXI H, 2050H
        MVI D, 00H
        MVI C, 02H
CHECK:  MOV A, M
        INX H
        CMP M
        JC NXTBYT
        MOV B,M
        MOV M,A
        DCX H
        MOV M,B
        INX H
        MVI D,01
NXTBYT: DCR C
        JNZ CHECK
        MOV A,D
        RRC
        JC START
        HLT
```

COUNTER and TIME DELAYS

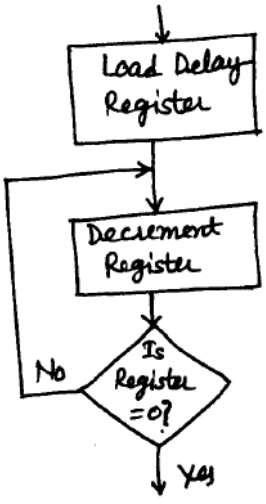
- To design a counter is a frequent programming application.
- Counters are used to keep track of events.
- Time delays are important in setting up reasonably accurate timing between two events.

- **Counter:** It is designed simply by loading an appropriate number into one of the register and using INR or the 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.



- Counter is shown in flowchart. It has a major drawback; the counting is performed at such high speed that only the last count can be observed.

- **Time delay:** - The procedure used to design a specific delay is similar to that used to set up a counter.
- A register is loaded with a number, depending on the time delay required.
- Then the register is decremented until it reaches to zero by setting up a loop with a conditional jump instruction.
- The loop causes the delay, depending upon the clock period of the system.



eg:

		T-State
	MVI C, FFH	7
LOOP:	DCR C	4
	JNZ LOOP	10/7

T-States in JNZ are 10/7.

10-Tstates: to execute a conditional Jump when it changes the sequence of program.

7-T-states: when program falls through the Loop.

eg:- 1. Calculate Time delay for following loop assuming frequency of microprocessor 2 MHz.

			T-states
	MVI	C, FFH	7
LOOP:	DCR	C	4
	JNZ	LOOP	10/7

Total Delay = Time to execute instruction outside loop + Time to execute loop instructions
$T_D = T_0 + T_{LA}$

clock period $T = 1/f = \frac{1}{2} \times 10^{-6} = 0.5\ \mu s$

$$T_L = T \times \text{Loop T-States} \times N_{10}$$

$T_L \Rightarrow$ Time delay in loop, T = System clock period, $N_{10} = (\text{Count Loaded in register})_{10}$

$T_L = 0.5\ \mu s \times 14 \times 255 = 1785\ \mu s = 1.785\ ms$

$T_{LA} = T_L - (3\ \text{Tstates} \times \text{clock period})$

$T_{LA} = 1785.0\ \mu s - 3 \times 0.5\ \mu s = (1785 - 1.5)\ \mu s = 1783.5\ \mu s$

$T_D = T_0 + T_{LA} = (7 \times 0.5\ \mu s) + 1783.5\ \mu s = 3.5\ \mu s + 1783.5\ \mu s$

$T_D = 1787\ \mu s$

2. Calculate time delay for following loop assuming frequency of microprocessor 2 MHz.

			T-state
	LXI	B, 2384H	10
LOOP:	DCX	B	6
	MOV	A, C	4
	ORA	B	4
	JNZ	LOOP	10/7

$(N_{10}) = (9092)_{10}$

[Since $(2384)_H = (9092)_{10}$]

$T_L = 0.5\ \mu s \times 24 \times (9092)_{10} = 109104\ \mu s$

$T_{LA} = 109104\ \mu s - 1.5\ \mu s = 109102.5\ \mu s$

$T_0 = 0.5\ \mu s \times 10 = 5\ \mu s$

$T_D = (109102.5 + 5)\ \mu s$

$T_D = (109.1075)\ ms \approx 109\ ms$

- Time Delay using a loop within a loop technique

eg:

```
        MVI B, 38H      7T
LOOP2 : MVI C, FFH      7T
LOOP1 : DCR C           4T
        JNZ LOOP1       10/7T
        DCR B           4T
        JNZ LOOP2       10/7T
```

let $T = 0.5\,\mu s$

$T_{L1} = T_{LA1} = (14 \times 255 - 3) \times 0.5\,\mu s$

$= 1783.5\,\mu s$

$T_{L2} = T_{O2} + T_{LA2} = (7 \times 0.5\,\mu s) + [(1783.5\,\mu s + 21 \times 0.5\,\mu s) \times 56 - 3 \times 0.5\,\mu s]$

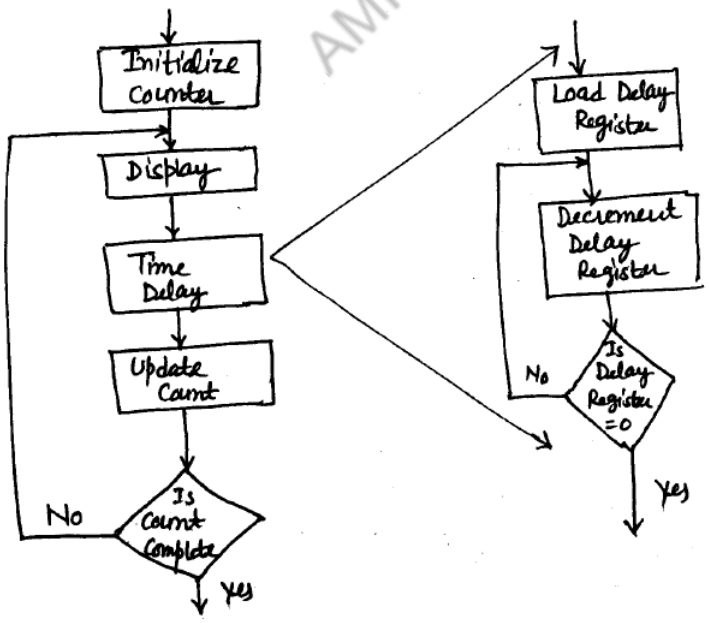
$= 3.5\,\mu s + 100464\,\mu s - 1.5\,\mu s = 100466\,\mu s$

$= 100.466\,ms$

- Time delay within a loop can be increased by using instruction that will not affect the program except to increase the delay.
- Disadvantages in using software delay technique for real time applications in which the demand for time accuracy is high are as follows:
  1. Accuracy of the time delay depends on the accuracy of system clock.
  2. The microprocessor is occupied simply in a waiting loop.
  3. The task of ~~each~~ calculating accurate time delay is tedious.

- Counter design with time delay:

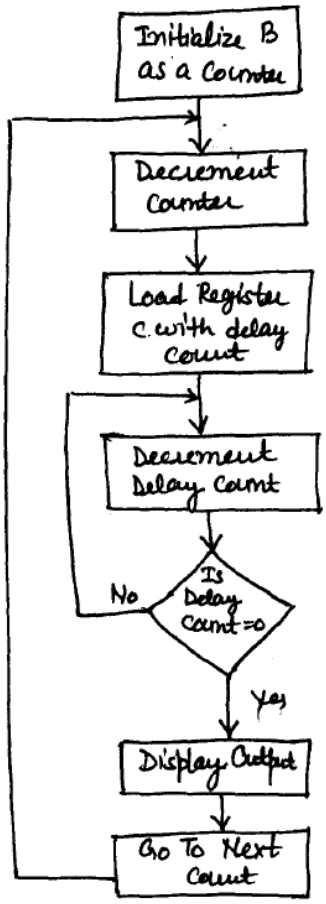
To design a counter with a time delay counter and timer design technique is combined.



- The flowchart shown is basic building blocks.
- This flowchart shows that count is displayed after initialization. Another way for displaying may be after updating the i count.

HEXADECIMAL COUNTER

Write a program to count continuously in Hexadecimal from FF_H to 00_H in a system with a 0·5 μs clock period. Use register C to set up a one millisecond delay between each count and display the numbers at one of the output ports.




```
        MVI  B, 00H
NEXT:   DCR  B
        MVI  C, COUNT
DELAY:  DCR  C
        JNZ  DELAY
        MOV  A, B
        OUT  PORT#
        JMP  NEXT
```

Delay Calculation

- Delay loop includes ~~DCR C~~ and
  DCR C (4T) and JNZ (10/7 T)
  with 14 T states.
- ∴ T_L in the loop is

T_L = 14 T-states × T × COUNT

T_L = 14 × 0·5 μs × COUNT = 7 μs × COUNT

- The delay outside the loop includes the following instructions

DCR B	4T
MVI C, COUNT	7T
MOV A, B	4T
OUT PORT #	10T
JMP	10T
	35T-states.

T_O = 35 × 0·5 μs = 17·5 μs

Total Delay $T_D = T_O + T_L = 17{\cdot}5 \times 10^{-6} + (7 \times 10^{-6} \times \text{COUNT})$

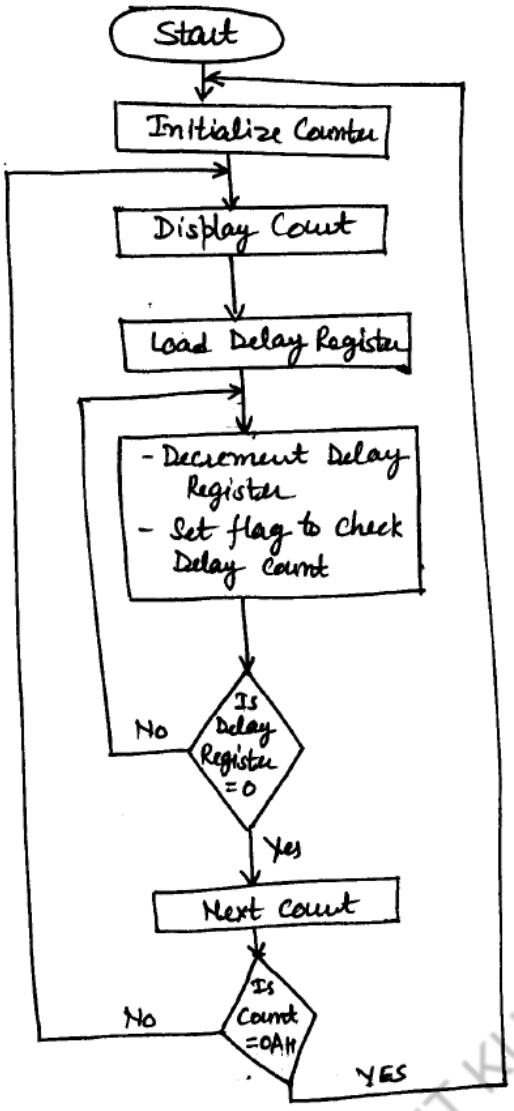
$1 \times 10^{-3} = 17{\cdot}5 \times 10^{-6} + (7 \times 10^{-6} \times \text{COUNT})$

$$\text{COUNT} = \frac{1 \times 10^{-3} - 175 \times 10^{-6}}{7 \times 10^{-6}} = (140)_{10}$$

COUNT = $(140)_{10} = 8C_H$.

To obtain 1 ms delay 8C is loaded in C Register.

ZERO-TO-NINE (MODULO TEN) COUNTER




```
                                        T-state
START :   MVI  B, 00H
          MOV  A, B
DISPLAY:  OUT  PORT #        10   } TD = 20
          LXI  H, 16-bit     10   }
LOOP  :   DCX  H             6    }
          MOV  A, L          4    } TL = 24
          ORA  H             4    }
          JNZ  LOOP          10/7
          INR  B             4    }
          MOV  A, B          4    } TD = 25
          CPI  0AH           7    }
          JNZ  DISPLAY       10/7
          JZ   START
```

Delay Calculation

Major delay between two counts is provided by 16 bit number in delay register HL.

Loop delay, T_L = 24 T-states × T × COUNT

$$1 \text{ sec.} = 24 \times 1 \times 10^{-6} \times \text{COUNT}$$

$$\text{COUNT} = \frac{1}{24 \times 10^{-6}} = (A2C2H)_{16} \text{ or } (41666)_{10}$$

- The delay count A2C2H in HL would provide approximately a 1-sec. delay between two counts.
- For more accurate calculation in delay instructions outside the loop and difference of 3 states in last JNZ must be accounted for calculating delay.

Total delay $T_D = T_0 + T_L$

$$1 \text{ sec.} = (45 \times 1.0 \times 10^{-6}) + (24 \times 1.0 \times 10^{-6} \times \text{COUNT})$$

$$\text{COUNT} = 41665$$

The difference between the two delay counts is very little.

GENERATING PULSE WAVEFORMS

Write a program to generate a continuous square wave with the period of 500 μs. Assume the system clock period is 325 ns and use bit D0 to output the square wave.

```
        MVI  D, AA      7 T
ROTATE: MOV  A, D       4 T
        RLC             4 T
        MOV  D, A       4 T
        ANI  01H        7 T
        OUT  PORT1      10 T
        MVI  B, COUNT   7 T
DELAY:  DCR  B          4 T
        JNZ  DELAY      10/7 T
        JMP  ROTATE     10 T
```

Delay Calculation

Pulse width = 250 μs

Total delay should include the delay in loop and the execution time of the instruction outside the loop.

1. The number of instruction outside the loop is seven.

   Delay outside the loop : T_O = 46 T-states × 325 ns = 14.95 μs

2. The delay loop includes two instructions (DCR and JNZ) with 14-T states, except the last cycle, which has 11 T-states.

   Loop delay : T_L = 14 T-state × 325 ns (COUNT − 1) + 11 T-states × 325 ns.

   = 4.5 μs (COUNT − 1) + 3.575 μs

3. The total delay required is 250 μs. Therefore, the count can be calculated as follows:

   $T_D = T_O + T_L$

   250 μs = 14.95 μs + 4.5 μs (COUNT − 1) + 3.575 μs

   $COUNT = (52.4)_{10} = 34_H$

Output: Square wave with a period of 500 μs.

- Debugging Counter and Time delay Programs

The following is a list of common errors:

1. Error in counting T-states in a delay loop.
2. Error in recognizing how many times a loop is repeated.
3. Failure to convert a delay count from a decimal number into its hexadecimal equivalent.
4. Conversion error in coverting a delay count from decimal to hexadecimal number or vice versa.
5. Specifying a wrong Jump location.
6. Failure to set a flag, especially with 16-bit Decrement / Increment instruction.
7. Using a wrong Jump instruction.
8. Failure to display either the first or the last count.
9. Failure to provide a delay between the last and last but one count.

STACK and SUBROUTINES

- Stack in an 8085 microprocessor system can be defined as a set of memory locations in the R/W memory. It is specified by a programmer in a ~~a math memory~~ main program. These memory locations are used to store binary information temporarily during the execution of a program.
- The beginning of the stack is defined in the program by 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 stack pointer and continues in reversed numerical order.
- The size of the stack is limited only by the available memory.
- Data bytes in the register pairs of the microprocessor can be stored on the stack in reverse order by using the instruction PUSH.
- Data bytes can be transferred from stack to respective registers by using the instruction POP.
- The stack pointer register tracks the storage and retrieval of the information.
  - → When two bytes are being stored at a time the contents of stack pointer is decremented by two;
  - → When data bytes are retrieved, the address is incremented by two.

  An address in the stack pointer register indicates that the next two memory locations can be used for storage.
- The stack is shared by the programmer and microprocessor.
  - → programmer uses PUSH and POP instruction to store and retrieve contents of register pair.
  - → Microprocessor automatically stores the contents of Program counter when subroutine is called.

- Instructions used for stack.

→ **LXI SP, 16-bit** :

- Load stack pointer instruction, that loads the stack pointer register with a 16-bit address. The operands for this instruction are B, D, H. (B ⇒ BC, D ⇒ DE, H ⇒ HL)

→ **PUSH Rp** :

- This is one byte instruction that copies the contents of specified register pair on stack.
- The SP is decremented and contents of high order register is are copied in the location shown by SP; the SP is again decremented and contents of low order register are copied in that location.
- The operand Rp implies the register pair B, D, H.
- The operand PSW represents Program Status Word, meaning the contents of the accumulator and Flags.

eg:-
```
PUSH B
PUSH PSW
```

→ **POP Rp** :

- Retrieve Register pair from stack
- This is one byte instruction that copies the contents of the top two memory locations of stack into specified register pair.
- First the contents of memory location indicated by the SP are copied into the lower order register, and then SP is incremented by 1. The contents of next location are copied into high-order register and SP is again incremented by 1.

eg:
```
POP B
POP PSW
```

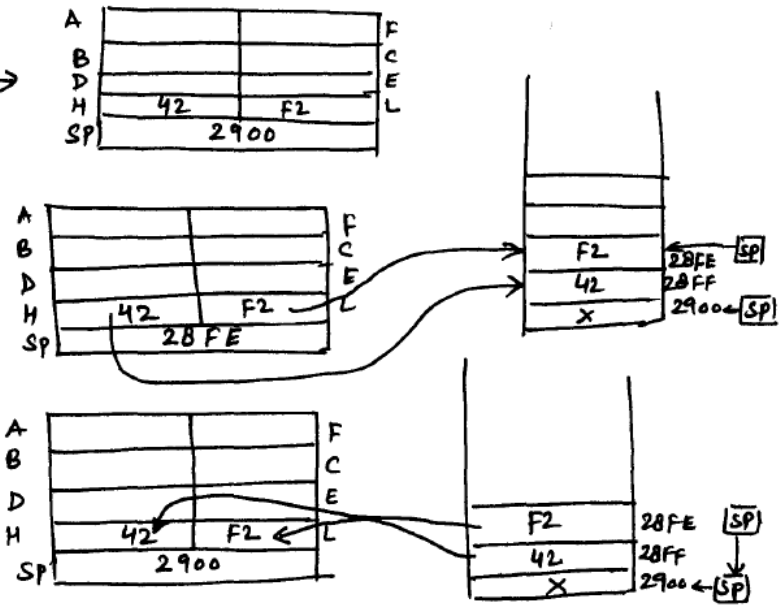
Example

1. Consider the following instructions

```
LXI  SP, 2900H
LXI  H, 42F2H
PUSH H
DELAY COUNTER
  ↓
POP H
```

Show the contents of various register when PUSH and POP instructions are executed.

```
LXI   SP, 2900H
LXI   H , 42F2H
PUSH  H
DELAY COUNTER
POP   H
```



2. Write a program to perform the following function

(i.) Clear all the flags

(ii.) Load 00H in the accumulator and demonstrate that the Zero flag is not affected by the data transfer instruction

(iii.) Logically OR the accumulator with itself to set the Zero flag, and ~~display the flag at PORT1~~ Store all the flags on the stack.

```
LXI  SP, 2099 H
MVI  L, 00H
PUSH H
POP  PSW
```

```
MVI  A, 00H
PUSH PSW
POP  H
MOV  A,L
```

Program Output:

40H

```
MVI  A,00H
ORA  A
PUSH PSW
POP  H
MOV  A,L
ANI  40H
```

```
MVI  A,00H
ORA  A
PUSH PSW
HLT
```

Storing the flags in stack

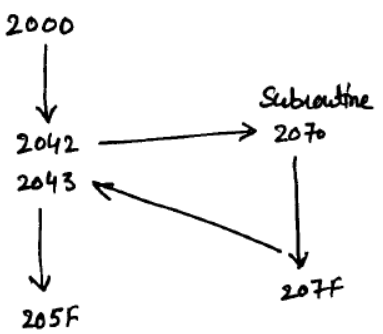
- SUBROUTINE

- A subroutine is a group of instructions written separately from the main program to perform a function that occurs repeatedly in the main program.
  eg: To insert successive delays, a separate set of instructions is written to do it. This is known as subroutine technique.
- The 8085 microprocessor has two instructions to implement subroutines: CALL and RET.
  → The CALL is used in the main program to call a subroutine.
  → The RET is used at the end of the subroutine to return to the main program.
- When a subroutine is called, the contents of program counter (PC), is stored on stack and program execution is transferred to the subroutine address. When a RET instruction is executed at the end of the subroutine the memory address stored on the stack is retrieved and the sequence of execution is resumed in the main program.

- Instructions to implement subroutine

- | CALL | 16-bit memory address of subroutine | : This is 3-byte instruction that transfers the program sequence to a subroutine address.
  - This instruction saves the contents of program counter on stack, decrements the stack pointer register by two and jumps unconditionally to the memory location specified by the second and third bytes.
  - This instruction is accompanied by a return instruction in subroutine.
- RET : This is 1-byte instruction and inserts the two bytes from the top of the stack into the program counter and increments the stack pointer register by two.

- Program execution : The sequence of program execution and events in execution of CALL and subroutine are shown in figure



- Starting address = 2000 of program
- Address of subroutine call instruction = 2042
- Address of subroutine starting from 2070 and ending at 207F.
- At 207F, there is RET instruction.
- After execution of RET, main program continues from 2043.

- Execution of CALL instruction

The CALL instruction requires five machine cycles and eighteen T-states. The sequence of events in each machine cycles is as follows:

1. M1-machine cycle: (opcode fetch) In this machine cycle, the contents of program counter are placed on the address bus and instruction code CD is fetched using the data bus. At the same time program counter is upgraded to the next memory address.
   - After the instruction is decoded and executed, the stack pointer register is decremented by one.

2. M2 and M3 - Memory Read: There are two Memory Read cycles during which the 16-bit address of CALL instruction is fetched. The low-order address is fetched first and placed in internal register Z. The high-order address is fetched next and placed in register W.
   - During M3 the program counter is upgraded to point the next instruction.

3. M4 and M5 : Storing of Program Counter: - At the beginning of the M4 cycle, the normal operation of placing the contents of program counter on the address bus is suspened.
   - The contents of stack pointer are placed on the address bus.
   - The contents of program counter are placed on data bus and stored in the stack.
     (PCH is placed on data bus and stored in stack and SP is decrementated by 1)
   - During M5, the contents of stack pointer (SP-1) are placed on the address bus. and PCL is placed on data bus and stored in stack.

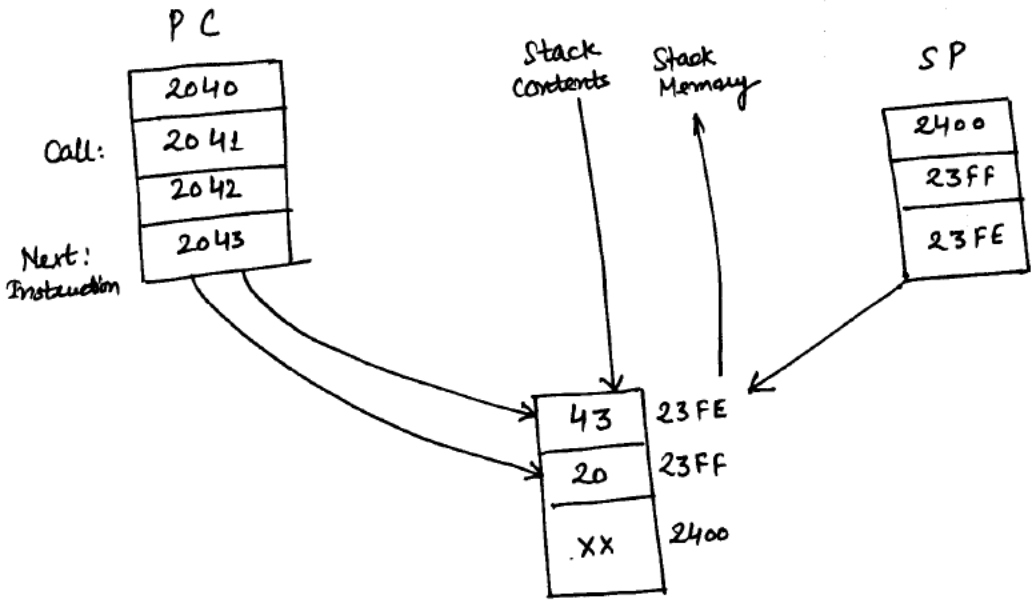
4. Next instruction cycle: - The program execution sequence is transfered to call CALL location by placing the contents of W and Z registers on the address bus.
   - During M1 of next instruction cycle, the PC is upgraded to location (W, Z+1)

- Execution of RET instruction:

- At the end of subroutine, when the instruction RET is executed, the program execution sequence is transferred to the address stored in the top two locations of the stack during CALL instruction.
- The sequence of events that occurs as the RET is executed are as follows:
  - → M_1 is normal opcode fetch cycle.
  - → During M_2 the contents of the stack pointer register are placed on the address bus, rather than those of PC. The data byte from the top of the stack is fetched and stored in the Z register, and the SP is upgraded to the next location.
  - → During M_3, the next byte is copied from the stack and stored in register W and SP is again upgraded.
  - → The program sequence is transferred by placing the contents of W/Z registers on the address bus at the beginning of the next instruction cycle.

eg:- For CALL instruction.

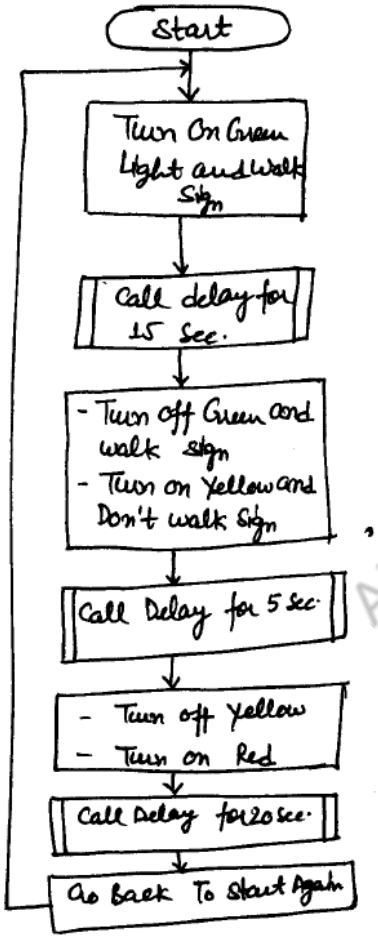
2040	CD	CALL 2070
2041	70	
2042	20	
2043	NEXT,	NEXT INSTRUCTION

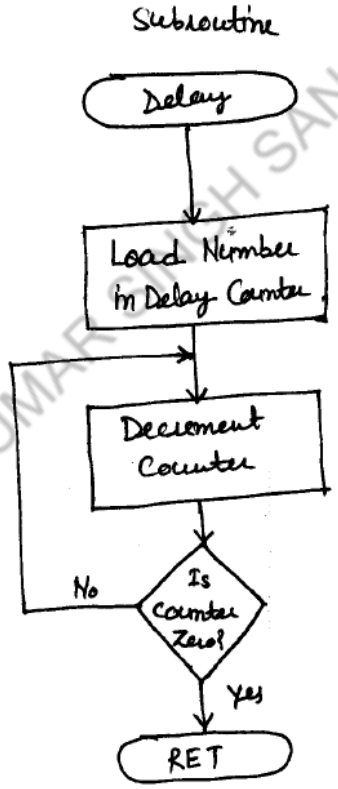


Example: Write a program to provide the given on/off time to three traffic lights and two pedestrian. The signal lights and signs are turned on/off by the data bits of an output port as shown below:

Lights:	Data bits	On time
Green	D0	15 sec.
Yellow	D2	05 sec.
Red	D4	20 sec.
Walk	D6	15 sec.
Don't walk	D7	25 sec.

The traffic and ~~pedestrian~~ pedestrian flow are in the same direction; the pedestrian should cross the road when the Green light is on.





- The on/off times for traffic signals and pedestrians signs are as follows:

Time Sequence in sec	Don't walk	walk	Red			Yellow	Green		Hex Code
	D7	D6	D5	D4	D3	D2	D1	D0	
0 ↓ 15	0	1	0	0	0	0	0	1	= 41H
↓ 20	1	0	0	0	0	1	0	0	= 84H
↓ 40	↓ 1	0	0	1	0	0	0	0	= 90H

```
        LXI  SP, 2099
START:  MVI  A, 41H      ; bit pattern for Green light and WALK
        OUT  PORT#
        MVI  B, 0FH
        CALL DELAY
        MVI  A, 84H      ; bit pattern for Yellow light and Don't walk
        OUT  PORT#
        MVI  B, 05H
        CALL DELAY
        MVI  A, 90H      ; bit pattern for Red light and Don't walk
        OUT  PORT#
        MVI  B, 14H
        CALL DELAY
        JMP  START

DELAY:  PUSH D           ; Save contents of DE and Accumulator
        PUSH PSW         ;
SECOND: LXI  D, COUNT    ; load Register pair with COUNT for 1sec delay
LOOP :  DCX  D
        MOV  A, D
        ORA  E           ; OR D and E for setting Z flag.
        JNZ  LOOP
        DCR  B           ; Is this end of time needed?
        JNZ  SECOND
        POP  PSW
        POP  D
        RET
```


- Subroutine Documentation and parameter passing

- Various information is passed between a calling program and a subroutine, a procedure called parameter passing.
- The documentation should include at least the following:
  1. Functions of the subroutine
  2. Input / output parameters
  3. Registers used or modified
  4. List of other subroutines called by this subroutine

RESTART, CONDITIONAL CALL and RETURN INSTRUCTIONS

- Restart (RST) instructions

* RST is 1-byte call instructions that transfer the program execution to a specific location on page 00H.
* They are executed the same way as call instructions.
* When an RST instruction is executed, 8085 stores the contents of the program counter (PC) on the top of the stack and transfers the program to the Restart location. These instructions are generally used in conjunction with the interrupt process.
* These instructions are listed here to emphasize that they are Call instructions and not necessarily always associated with the interrupts.
* The list of eight RST instructions is as follows:

RST 0	Call 0000H	RST 4	Call 0020H
RST 1	Call 0008H	RST 5	Call 0028H
RST 2	Call 0010H	RST 6	Call 0030H
RST 3	Call 0018H	RST 7	Call 0038H

- Conditional Call and Return Instruction

* The conditional call and Return instructions are based on four data conditions: Carry, Zero, Sign, and Parity.
* The Conditions are tested by checking the respective flags.
* In case of a conditional call instruction, the program is transferred to the subroutine if condition is met; otherwise, the main program is continued.
* In case of a conditional return instruction, the sequence returns to the main program if the condition is met; otherwise the sequence in the subroutine is continued.
* If the call instruction in the main program is conditional, the Return instruction in the subroutine can be conditional or unconditional.
* The Conditional Call and Return instructions are listed for reference.
* Conditional CALL:

CC - Call subroutine if Carry flag is set (CY=1)
CNC - Call subroutine if Carry flag is reset (CY=0)
CZ - Call subroutine if Zero flag is set (Z=1)
CNZ - Call Subroutine if Zero flag is reset (Z=0)
CM - Call Subroutine if Sign flag is set (S=1), negative number
CP - Call Subroutine if Sign flag is reset (S=0), positive number
CPE - Call Subroutine if Parity flag is set (P=1), even parity
CPO - Call Subroutine if Parity flag is reset (P=0), Odd parity

* Conditional RETURN

RC	- Return	if	Carry flag	is set	(CY = 1)
RNC	- Return	if	Carry flag	is reset	(CY = 0)
RZ	- Return	if	Zero flag	is set	(Z = 1)
RNZ	- Return	if	Zero flag	is reset	(Z = 0)
RM	- Return	if	Sign flag	is set	(S = 1), negative number
RP	- Return	if	Sign flag	is reset	(S = 0), positive number
RPE	- Return	if	Parity flag	is set	(P = 1), even parity
RPO	- Return	if	Parity flag	is reset	(P = 0), odd parity

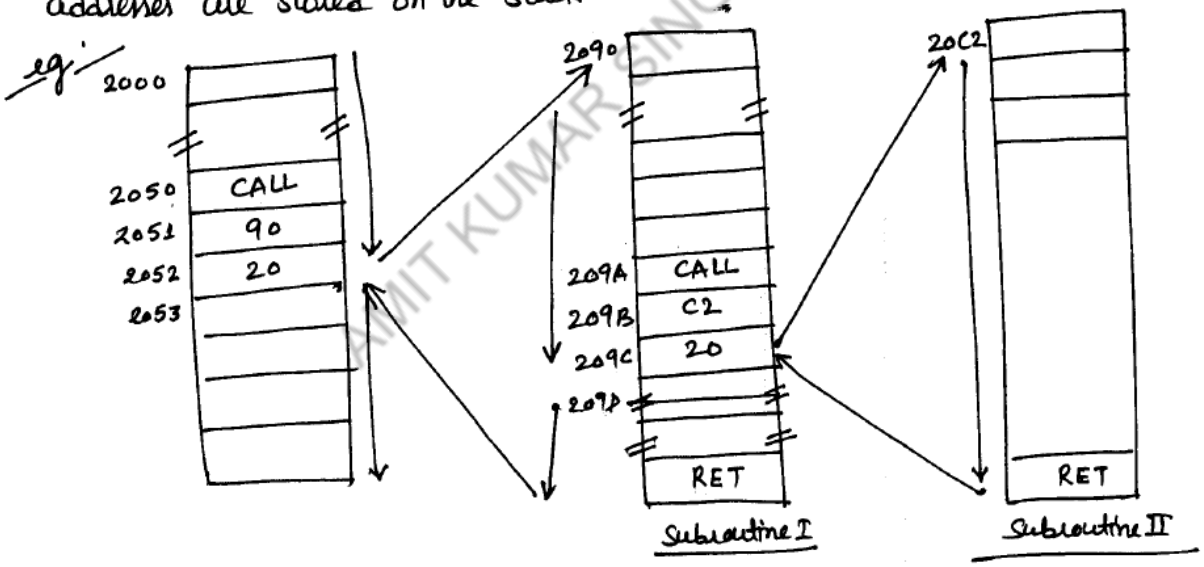
ADVANCED SUBROUTINE CONCEPTS

Some advanced subroutine concepts are nesting and multiple ending.

- **Nesting** * The programming technique of a subroutine calling another subroutine is called nesting.

* This process of nesting is limited only by the number of available stack locations. When a subroutine calls another subroutine, all return addresses are stored on the stack.

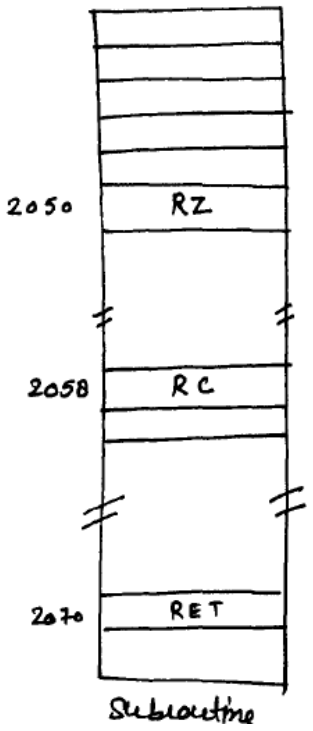
eg:



- **Multiple - Ending Subroutines**

* It is possible to have multiple endings i.e. multiple return for single CALL instruction.

eg: one CALL instruction can have three possible endings as shown in figure. The subroutine has two conditional returns (RZ and RC) and one unconditional return (RET).



- if Z = 1,
  Subroutine returns from 2050H
- if CY = 1,
  subroutine returns from 2058H
- if neither Z nor CY is set it returns from 2070H.