CHAPTER 2
Introduction to Assembly Language Programming (10 Hrs.)

PRIME COLLEGE
ROLISHA STHAPIT

CONTENTS

Assembly Language Programming Basics, Classifications of Instructions and Addressing Mode, 8085 Instruction Sets, Assembling, Executing and debugging the Programs, Developing Counters and Time Delay Routines, Interfacing Concepts.

Introduction

A microprocessor executes instructions given by the user

- Instructions should be in a language known to the microprocessor
- Microprocessor understands the language of 0’s and 1’s only
- This language is called **Machine Language**

Assembly Language Programming Basics

- An assembly language is the most basic programming language available for any processor.
- With assembly language, a programmer works only with operations that are implemented directly on the physical CPU.
- Assembly languages generally lack high-level conveniences such as variables and functions, and they are not portable between various families of processors.
- They have the same structures and set of commands as machine language, but allow a programmer to use names instead of numbers. This language is still useful for programmers when speed is necessary or when they need to carry out an operation that is not possible in high-level languages.
- Assembly language is specific to a given processor. For e.g. assembly language of 8085 is different than that of Motorola 6800 microprocessor.

Microprocessor cannot understand a program written in Assembly language. A program known as Assembler is used to convert Assembly language program to machine language.

Assembly language program to add two numbers

```
MVI A, 2H  ;Copy value 2H in register A
MVI B, 4H  ;Copy value 4H in register B
ADD B              ;A = A + B
```

Advantages of Assembly Language

a) The symbolic programming of Assembly Language is easier to understand and saves a lot of time and effort of the programmer.
b) It is easier to correct errors and modify program instructions.
c) Assembly Language has the same efficiency of execution as the machine level language.

Disadvantages of Assembly Language

a) One of the major disadvantages is that assembly language is machine dependent. A program written for one computer might not run in other computers with different hardware configuration.
b) If you are programming in assembly language, you must have detailed knowledge of the particular microcomputer you are using.
c) Assembly language programs are not portable

Low-level/High-level languages

Machine language and Assembly language are both
–Microprocessor specific (**Machine dependent**)
–Low-level languages
•**Machine independent** languages are called
–High-level languages
–For e.g. BASIC, PASCAL,C++,C,JAVA, etc.
–A software called **Compiler** is required to convert a high-level language program to machine code

8085 Programming Model

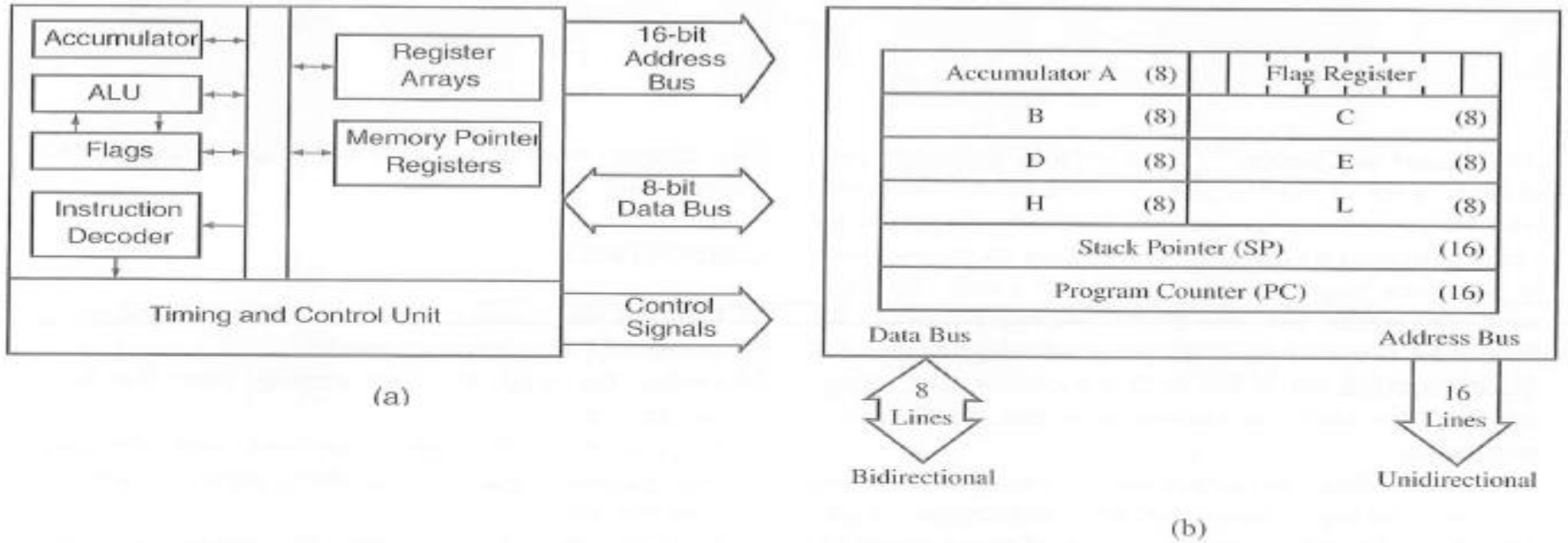


FIGURE 2.1
8085 Hardware Model (a), Programming Model (b), and Flag Register (c)

8085 HARDWARE MODEL

The hardware model in fig (a) shows two major segments. One segment includes arithmetic logic unit [ALU] and an 8 bit register called an accumulator, instruction decoder, and flags. The second segment shows 8 bit and 16 bit registers. Both segments are connected with various internal connections called an internal bus. The arithmetic and logic operations are performed in the arithmetic logic unit [ALU]. Results are stored in the accumulator, and flip-flops, called flags, are set or reset to reflect the results. There are 3 buses- a 16 bit unidirectional address bus, an 8 bit bidirectional data bus, and a control bus.

The 8085 Programming Model

The 8085 programming model includes six registers, one accumulator, and one flag register, Figure. In addition, it has two 16-bit registers: the stack pointer and the program counter. They are described briefly as follows.

Registers

The 8085 has six general-purpose registers to store 8-bit data; these are identified as B,C,D,E,H, and L as shown in the figure. They can be combined as register pairs - BC, DE, and HL - to perform some 16-bit operations. The programmer can use these registers to store or copy data into the registers by using data copy instructions.

Accumulator

- The accumulator is an 8-bit register that is a part of arithmetic/logic unit (ALU). This register is used to store 8-bit data and to perform arithmetic and logical operations. The result of an operation is stored in the accumulator. The accumulator is also identified as register A.

ACCUMULATOR A (8) FLAG REGISTER

B (8)

D (8)

H (8)

Stack Pointer (SP) (16)

Program Counter (PC) (16)

C (8)

E (8)

L (8)

Data Bus Address Bus

- 8 Lines Bidirectional 16 Lines unidirectional

- **Flags**
  The ALU includes five flip-flops, which are set or reset after an operation according to data conditions of the result in the accumulator and other registers. They are called Zero(Z), Carry (CY), Sign (S), Parity (P), and Auxiliary Carry (AC) flags; their bit positions in the flag register are shown in the Figure below. The most commonly used flags are Zero, Carry, and Sign. The microprocessor uses these flags to test data conditions.
- For example, after an addition of two numbers, if the sum in the accumulator is larger than eight bits, the flip-flop uses to indicate a carry called the Carry flag (CY) is set to one. When an arithmetic operation results in zero, the flip-flop called the Zero(Z) flag is set to one. The first Figure shows an 8-bit register, called the flag register, adjacent to the accumulator. However, it is not used as a register; five bit positions out of eight are used to store the outputs of the five flip-flops. The flags are stored in the 8-bit register so that the programmer can examine these flags (data conditions) by accessing the register through an instruction.
- These flags have critical importance in the decision-making process of the microprocessor. The conditions (set or reset) of the flags are tested through the software instructions. For example, the instruction JC (Jump on Carry) is implemented to change the sequence of a program when CY flag is set. The thorough understanding of flag is essential in writing assembly language programs.

Program Counter (PC)
This 16-bit register deals with sequencing the execution of instructions. This register is a memory pointer. Memory locations have 16-bit addresses, and that is why this is a 16-bit register.
The microprocessor uses this register to sequence the execution of the instructions. The function of the program counter is to point to the memory address from which the next byte is to be fetched. When a byte (machine code) is being fetched, the program counter is incremented by one to point to the next memory location

Stack Pointer (SP)
The stack pointer is also a 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.
This programming model will be used in subsequent tutorials to examine how these registers are affected after the execution of an instruction.

Instruction description and format:

An instruction manipulates the data and a sequence of instructions constitutes a program. Generally each instruction has two parts: one is the task to be performed, called the **operation code** (Op-Code) field, and the second is the data to be operated on, called the **operand** or address field**.** The operand (or data) can be specified in various ways. It may include 8-bit (or 16-bit) data, an internal register, a memory location, or an 8-bit (or 16-bit) address. The Op-Code field specifies how data is to be manipulated and address field indicates the address of a data item.

For example: ADD R1, R0
Op-code address
Here R0 is the source register and R1 is the destination register. The instruction adds the contents of R0 with the content of R1 and stores result in R1.

Classification of instruction on the basisof address specified / Instruction Word Size

Depending on the number of address specified in instruction sheet, the instruction format can be classified into the categories.

One address format (1 byte instruction):

- Here 1 byte will be Op-code and operand will be default. E.g. ADD B, MOV A,B

Two address format (2 byte instruction) :

- Here first byte will be Op-code and second byte will be the operand/data. E.g. IN 40H, MVI A, 8-bit Data

Three address format (3 byte instruction):

- Here first byte will be Op-code, second and third byte will be operands/data. That is

  2nd byte- lower order data.

  3rd byte – higher order data

  E.g. LXI B, 4050

Examples

ONE-BYTE INSTRUCTIONS

A 1-byte instruction includes the opcode and the operand in the same byte. For example:

Task	Opcode	Operand*	Binary Code	Hex Code
Copy the contents of the accumulator in register C.	MOV	C,A	0100 1111	4FH

TWO-BYTE INSTRUCTIONS

In a 2-byte instruction, the first byte specifies the operation code and the second byte specifies the operand. For example:

Task	Opcode	Operand	Binary Code	Hex Code	
Load an 8-bit data byte in the accumulator.	MVI	A,32H	0011 1110	3E	First Byte
			0011 0010	32	Second Byte

Examples

THREE-BYTE INSTRUCTIONS

In a 3-byte instruction, the first byte specifies the opcode, and the following two bytes specify the 16-bit address. Note that the second byte is the low-order address and the third byte is the high-order address. For example:

Task	Opcode	Operand	Binary Code	Hex Code*	
Load contents of memory 2050H into A.	LDA	2050H	0011 1010	3A	First Byte
			0101 0000	50	Second Byte
			0010 0000	20	Third Byte

Data Format

- The 8085 is an 8-bit microprocessor, and it processes only binary numbers.
- We need to code binary numbers into different media.
- Common codes and data formats are ASCII, BCD, signed integers, and unsigned integers:
  - **ASCII Code - 7-bit alphanumeric code that represents decimal numbers, English alphabets, and nonprintable characters such as carriage return. Extended ASCII is an 8-bit code.**
  - **BCD Code - Binary-coded decimal; it is used for decimal numbers.**

- Signed Integer - A signed integer is either a positive number or a negative number.
  - In an 8-bit processor, the most significant digit, D7, is used for the sign; 0 represents the positive sign and I represents the negative sign.
  - The remaining seven bits, D6—D0, represent the magnitude of an integer. Therefore, the largest positive integer that can be processed by the 8085 at one time is 0111 1111 (7FH); the remaining Hex numbers, 80H to FFH, are considered negative numbers.

- Unsigned Integers - An integer without a sign can be represented by all the 8 bits in a microprocessor register.
  - **Therefore, the largest number that can be processed at one time is FFH.**
  - **However, this does not imply that the 8085 microprocessor is limited to handling only 8-bit numbers. Numbers larger than 8 bits (such as 16-bit or 24-bit numbers) are processed by dividing them in groups of 8 bits.**

Addressing Modes of 8085

Type	Instruction	Source	Destination
Direct	STA 2005 H	Accumulator	Memory 2005H
Register	MOV A, B	Register B	Accumulator
Register Indirect	MOV A, M	Memory [[HL]]	Accumulator
Immediate	MVI A, 18H	Data 18H	Accumulator
Implicit	CMA		

Note: For details refer chapter 1

Instruction Set of 8085

- An instruction is a binary pattern designed inside a microprocessor to perform a specific function. The entire group of instructions that a microprocessor supports is called Instruction Set.
- 8085 has 246 instructions. Each instruction is represented by an 8-bit binary value. These 8-bits of binary value is called Op-Code or Instruction Byte.
- Following are the classification of instructions:

a) Data Transfer Instruction
b) Arithmetic Instructions
c) Logical Instructions
d) Branching Instructions
e) Control Instructions

a) Data Transfer Instruction

These instructions move data between registers, or between memory and registers. These instructions copy data from source to destination. While copying, the contents of source are not modified.

Example: MOV, MVI

b) Arithmetic Instructions

These instructions perform the operations like addition, subtraction, increment and decrement.

Example: ADD, SUB, INR, DCR

c) Logical Instructions

These instructions perform logical operations on data stored in registers and memory. The logical operations are: AND, OR, XOR, Rotate, Compare and Complement.

Example: ANA, ORA, RAR, RAL, CMP, CMA

d) Branching Instructions

Branching instructions refer to the act of switching execution to a different instruction sequence as a result of executing a branch instruction. The three types of branching instructions are: Jump, Call and Return.

e) Control Instructions

The control instructions control the operation of microprocessor. Examples: HLT, NOP, EI (Enable Interrupt), DI (Disable Interrupt).

Instruction Set of 8085

8085 instructions can be classified as

1. **Data Transfer (Copy) instructions**
2. **Arithmetic instructions**
3. **Logical and Bit manipulation instructions**
4. **Branching instructions**
5. **Miscellaneous instructions / Control**

Data Transfer Operations

- The data transfer instructions load given data into register, copy data from register to register, copy data from register to memory location, and vice versa.
- In other words we can say that data transfer instructions copy data from source to destination.
- Source can be data or contents of register or contents of memory location whereas destination can be register or memory location.
- These instructions do not affect the flag register of the processor.

Instructions

1) MOV Rd, Rs(move register instruction)

- 1 byte instruction
- Copies data from source register to destination register.
- Rd & Rsmay be A, B, C, D, E, H &L
- E.g. MOV A, B

2) MVI R, 8 bit data (move immediate instruction)

- 2 byte instruction
- Loads the second byte (8 bit immediate data) into the register specified.
- Rmay be A, B, C, D, E, H & L
- E.g. MVI C, 53H

Cont..

3) MOV M, R(Move to memory from register)

- Copy the contents of the specified register to memory. Here memory is the location specified by contents of the ***HL register pair***.
- E.g. MOV M, B

4) MOV R, M (move to register from memory)

- Copy the contents of memory location specified by **HL pair** to specified register.
- E. g. MOV B, M

Cont..

5) LXI R_P, 2 bytes data (load register pair)

- 3-byte instruction
- Load immediate data to register pair – Register pair may be BC, DE, HL & SP(Stack pointer)
- 1st byte- Op-code
- 2nd byte – lower order data
- 3rd byte- higher order data
- E.g. LXI B, 4532H; B← 45,C←32H

Cont..

6) MVI M, data (load memory immediate)

- 2 byte instruction.
- Loads the 8-bit data to the memory location whose address is specified by the contents of HL pair. E.g. MVI M ,35H; [HL]← 35H

7) LDA 4035H (Load accumulator direct)

- 3-byte instruction
- Loads the accumulator with the contents of memory location whose address is specified by 16 bit address.
- A←[4035H]

Cont..

8. STA16-bit address (store accumulator contents direct)

- 3-byte instruction.
- Stores the contents of accumulator to specified address
- E.g. STAFA00H

9. LDAX RP (Load accumulator indirect)

- 1 byte instruction.
- Loads the contents of memory location pointed by the contents of register pair to accumulator.
- E. g. LDAX B

```
LXI B, 9000H      B= 90, C= 00
LDAX B            A= [9000]
```

Cont..

10) STAX RP

- Stores the contents of accumulator to memory location specified by the contents of register pair.
- 1 byte instruction
- Example

```
LXI B, 9500H
LXI D, 9501H
MVI  A,  32H
STAXB
MVI A, 7AH
STAX D
```

Cont..

11) IN 8-bit address

- 2-byte instruction
- Read data from the input port address specified in the second byte and loads data into the accumulator.
- E.g. IN 40H

12) OUT 8-bit address

- 2-byte instruction
- Copies the contents of the accumulator to the output port address specified in the 2nd byte
- E.g. OUT 40H

Cont..

13) LHLD16-bit address (Load HL directly)

- 3-byte instruction.
- Loads the contents of specified memory location to L register and contents of next higher location to H-register.
- Eg. LXI H, 9500H
  MVI M, 32H
  MVI L, 01H
  MVI m, 7AH
  LHLD 9500H

Cont..

14) SHLD16-bit address (store HL directly)

- Opposite to LHLD.
- Stores the contents of Lregister to specified memory location and contents of H register to next higher memory location.
- E.g. LXI H, 9500H

  SHLD 8500H

Cont..

15) XCHG (Exchange)

- Exchanges DE pair with HL pair.
- E. g. LXI H, 7500H — H= 75, L=00

  LXI D, 9532H — D=95. E=32

  XCHG — H=95, L=32

  D=75 E=00

Arithmetic group Instructions

- The arithmetic operation add and subtract are performed in relation to the contents of accumulator. The features of these instructions are

1) They assume implicitly that the accumulator is one of the operands.
2) They modify all the flags according to the data conditions of the result.
3) They place the result in the accumulator.
4) They do not affect the contents of operand register or memory.

Arithmetic group Instructions

- The 8085 microprocessor performs various arithmetic operations such as addition, subtraction, increment and decrement. These arithmetic operations have the following mnemonics.

1) ADD R/M

- 1 byte add instruction.
- Adds the contents of register/memory to the contents of the accumulator and stores the result in accumulator.
- E.g. ADD B; A ← A + B

Cont..

2) ADI 8 bit data

- 2 byte add immediate instruction.
- Adds the 8 bit data with the contents of accumulator and stores result in accumulator.
- Eg. ADI 9BH ; A←A+9BH

3) SUB R/M

- 1 byte subtract instruction.
- Subtracts the contents of specified register / m with the contents of accumulator and stores the result in accumulator.
- E. g. SUB D ; A←A-D

Cont..

4) SUI 8 bit data

- 2 byte subtract immediate instruction.
- Subtracts the 8 bit data from the contents of accumulator stores result in accumulator.
- E. g. SUI D3H A ← A – D3H

Cont..

5) INR R/M, DCR R/M

- 1 byte increment and decrement instructions
- Increase and decrease the contents of R(register) or M(memory) by 1 respectively.

E. g. DCR B ; B=B-1
DCR M ; [HL] = [HL]-1
INR A ; A=A+1
INR M ; [HL] +1

For these, all flags are affected except carry.

Cont..

6. INX Rp, DCX RP

- Increase and decrease the register pair by 1.
- Acts as 16 bit counter made from the contents of 2 registers (1 byte instruction)
- E.g. INX B ;BC=BC+1
- DCX D ;DE=DE-1
- No flags affected

Cont..

7) ADC R/M and ACI 8-bit data (addition with carry (1 byte))

- ACI 8-bit data= immediate (2 byte).
- Adds the contents of register or 8 bit data whatever used suitably with the Previous carry.
- – E.g. ADC B ; A = A+B+CY

  ACI 70H ; A = A + 70+CY

Cont..

8) SBB B/M

- 1 byte instruction. – Subtracts the contents of register or memory from the contents of accumulator and stores the result in accumulator.
- –e. g.SBB D; A ← A-D-Borrow

SBI 8 bit data

- 2 byte instruction.
- Subtracts the 8-bit immediate data from the content of the accumulator and stores the result in accumulator.
- E.g. SBI 70H; A ← A-70-Borrow

Cont..

9) DAD Rp(double addition)

- 1 byte instruction.
- Adds register pair with HL pair and store the 16 bit result in HL pair.
- E. g. LXI H, 7320H
  LXI B, 4220H
  DAD B; HL=HL+BC

Cont..

10) DAA (Decimal adjustment accumulator)

- Used only after addition.
- 1 byte instruction.
- The content of accumulator is changed from binary to two 4-bit BCD digits.
- E. g MVI A, 78H ; A=78
  MVI B, 42H ; B=42
  ADD B ; A=A+B = BA
  DAA ; A=20, CY=1

Logical Group Instructions:

- Microprocessor can perform all the logic functions of the hardwired logic through its instruction set. The 8085 instruction set includes such logic functions as AND, OR, XOR and NOT (Complement):
- The following features hold true for all logic instructions:

1. The instructions implicitly assume that the accumulator is one of the operands.
2. All instructions reset (clear) carry flag except for complement where flag remain unchanged.
3. They modify Z, P& S flags according to the data conditions of the result.
4. Place the result in the accumulator.
5. They do not affect the contents of the operand register

1) ANA R/M (the contents of register/memory)

- Logically AND the contents of register/memory with the contents of accumulator.
- 1 byte instruction.
- **CY flag is reset and AC is set**.

2) ANI 8 bit data

- Logically AND 8 bit immediate data with the contents of accumulator.
- 2 byte instruction.
- **CY flag is reset and AC is set**. Others as per result

3) ORA R/M

- Logically OR the contents of register/memory with the contents of accumulator.
- 1 byte instruction.
- **CY and AC is reset and other as per result**.

4) ORI 8 bit data

- Logically OR 8 bit immediate data with the contents of the accumulator.
- 2 byte instruction.
- **CY and AC is reset and other as per result**.

5) XRA R/M

- Logically exclusive ORthe contents of register memory with the contents of accumulator.
- 1 byte instruction.
- **CYand ACis reset and other as per result.**

6) XRI 8 bit data

- Logically Exclusive OR8 bit data immediate with the content of accumulator.
- 2 byte instruction.
- **CYand ACis reset and other as per result**.

7) CMA (Complement accumulator)

- 1 byte instruction.
- Complements the contents of the accumulator.
- No flags are affected

Logically Compare instructions

7. **CMP R/M (1 byte instruction)**
   CPI 8 bit data (2 byte instruction)

- Compare the contents of register/ memory and 8 bit data with the contents of accumulator.
- Status is shown by flags & all flags are modified.

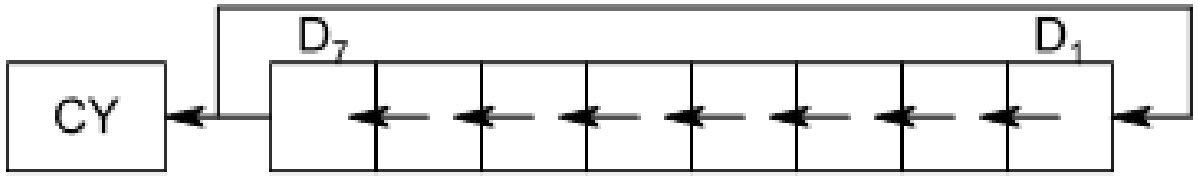
Case	CY	Z	
[A]<[R/M] or 8 bit	1	0	A-R<0
[A]=[R/M] or 8 bit	0	1	A-R=0
[A]>[R/M] or 8 bit	0	0	A-R>0

Logical Rotate instructions

- This group has four instructions, two are for rotating left and two are for rotating right. The instructions are:

1) RLC: Rotate accumulator left

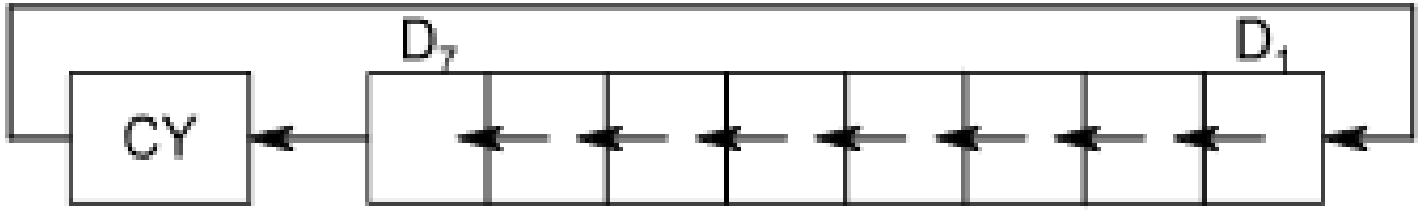
- Each bit is shifted to the adjacent left position.
- Bit D7 becomes D0.
- The carry flag is modified according to D7



$CY = D_7,\ D_7 = D_6,\ D_6 = D_5, \ldots\ldots, D_1 = D_0,\ D_0 = D_7$

2) RAL: Rotate accumulator left through carry

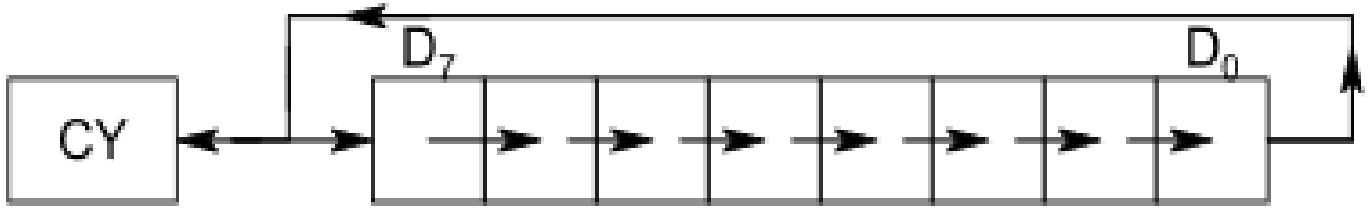
- Each bit is shifted to the adjacent left position.
- Bit D7 becomes the carry bit and the carry bit is shifted into D0.
- The carry flag is modified according to D7.



$CY = D_7,\ D_7 = D_6,\ D_6 = D_5, \ldots\ldots, D_1 = D_0,\ D_0 = CY$

3) RRC: rotate accumulator right

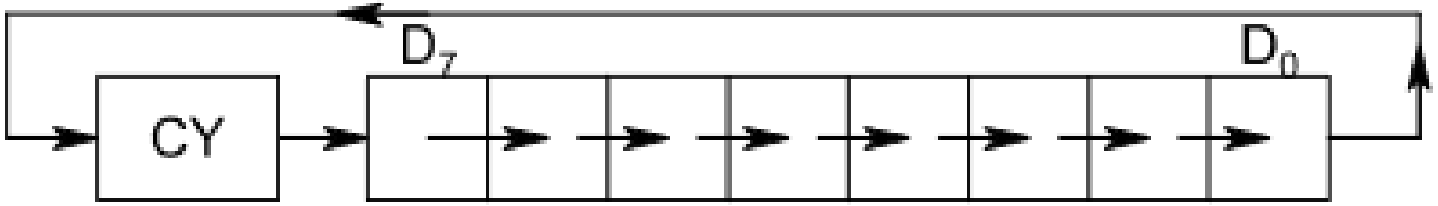
- Each bit is shifted right to the adjacent position.
- Bit D0 becomes D7.
- The carry flag is modified according to D0.



CY= D_0, D_7= D_0,.....,D_0=D_1

4) RAR: Rotate accumulator right through carry

- Each bit is shifted right to the adjacent position.
- Bit D0 becomes the carry bit and the carry bit is shifted into D7.



$CY = D_0$, $D_0 = D_1$, ...$D_7 = CY$

Branching Group Instructions:

- The microprocessor is a sequential machine; it executes machine codes from one memory location to the next.
- The branching instructions instruct the microprocessor to go to a different memory location and the microprocessor continues executing machine codes from that new location.
- The branching instruction code categorized in following three groups:
  - Jump instructions
  - Call and return instruction
  - Restart instruction

Jump Instructions

- The jump instructions specify the memory location explicitly.
- They are 3 byte instructions, one byte for the operation code followed by a 16 bit (2 byte) memory address. Jump instructions can be categorized into **unconditional and conditional jump.**

1. Unconditional Jump

- 8085 includes unconditional jump instruction to enable the programmer to set up continuous loops without depending only type of conditions.
- E.g. JMP 16 bit address: loads the program counter by 16 bit address and jumps to specified memory location.

E.g. JMP 4000H

Conditional Jump

- The conditional jump instructions allow the microprocessor to make decisions based on certain conditions indicated by the flags.
- After logic and arithmetic operations, flags are set or reset to reflect the condition of data.
- These instructions check the flag conditions and make decisions to change or not to change the sequence of program.
- The four flags namely **carry, zero, sign and parity** used by the jump instruction.

Cont…

Mnemonics	Description
JC16 bit	Jump on carry (if CY=1)
JNC16 bit	Jump on if no carry (if CY=0)
JZ16bit	Jump on zero (if Z=1)
JNZ16bit	jump on if no zero (if Z=0)
JP 16bit	jump on positive (if S=0)
JM 16bit	jump on negative (if S=1)
JPE16bit	Jump on parity even (if P=1)
JPO 16bit	Jump on parity odd (if P=0)

WAP to move 10 bytes of data from starting address 9500 H to 9600H

```
2000    MVI B, 0AH
2002    LXI H, 9500H
2005    LXI D, 9600H
2008    MOV A, M
2009    STAX D    ; Store the contents of accumulator to register pair.
200A    INX H     ; Increment the register pair by 1.
200B    INX D
200C    DCR B
200D    JNZ       2008
2010    HLT
```

Call and return instructions: (Subroutine)

- Call and return instructions are associated with subroutine technique.
- A subroutine is a group of instructions that perform a subtask. A subroutine is written as a separate unit apart from the main program and the microprocessor transfers the program execution sequence from main program to subroutine whenever it is called to perform a task.
- After the completion of subroutine task microprocessor returns to main program.
- The subroutine technique eliminates the need to write a subtask repeatedly, thus it uses memory efficiently.

- To implement subroutine there are two instructions CALL and RET.

1. CALL16 bit memory

- Call subroutine unconditionally.
- 3 byte instruction.
- Saves the contents of program counter on the stack pointer. Loads the PC by jump address (16 bit memory) and executes the subroutine.

2. RET

- Returns from the subroutine unconditionally.
- 1 byte instruction
- Inserts the contents of stack pointer to program counter

Restart Instruction:

- 8085 instruction set includes 8 restart instructions(RST).These are 1 byte instructions and transfer the program execution to a specific location.

Restart instruction	Hex Code	Call location in hex
RST 0	C7	0000H
RST 1	CF	0008H
RST 2	D7	0010H
RST 3	DF	0018H
RST 4	E7	0020H
RST 5	EF	0028H
RST 6	F7	0030H
RST 7	FF	0038H

Cont..

- When RST instruction is executed, the 8085 stores the contents of PC on SP and transfers the program to the restart location.
- Actually these restart instructions are inserted through additional hardware.
- These instructions are part of interrupt process.

Miscellaneous Group Instructions:

STACK

- The stack is defined as a set of memory location in R/W memory, specified by a programmer in a main memory. 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,16 bit address.
- Once the stack location is defined, it loads 16 bit address in the stack pointer register. Storing of data bytes for this operation takes place at the memory location that is one less than the address
- e.g. LXI SP,2099H

- The stack instructions are:

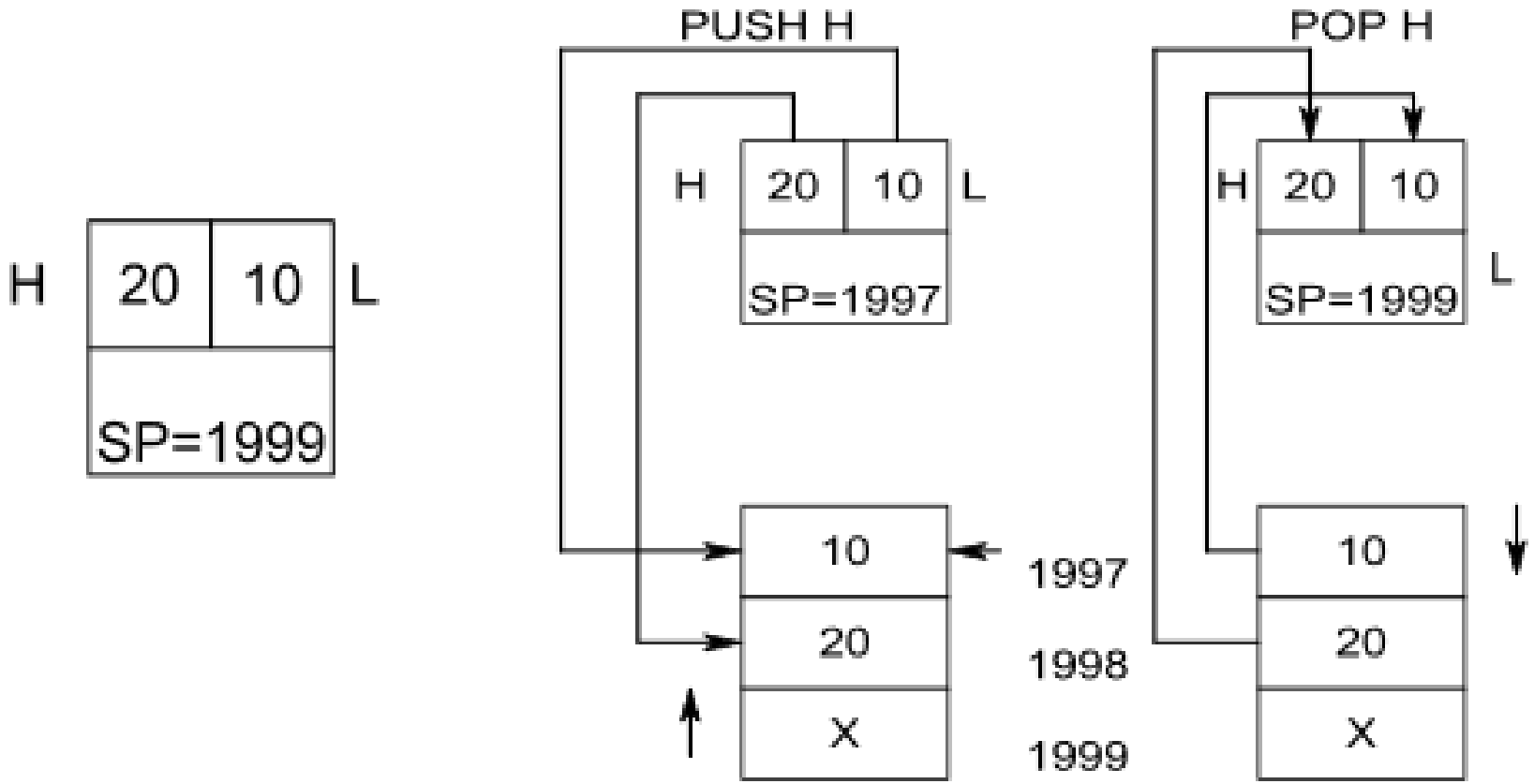
1. PUSH Rp/PSW (Store register pair on stack)

- 1 byte instruction.
- Copies the contents of specified register pair or program status word (accumulator and flag) on the stack.
- Stack pointer is decremented and content of high order register is copied. Then it is again decremented and content of low order register is copied.

2. POP Rp/PSW (retrieve register pair from stack)

- 1 byte instruction.
- Copies the contents of the top two memory locations of the stack into specified register pair or program status word.

- A content of memory location indicated by SP is copied into low order register and SP is incremented by 1. Then the content of next memory location is copied into high order register and SP is incremented by 1.



Cont..

3. XTHL – exchanges top of stack (TOS) with HL
4. SPHL – move HL to SP
5. PCHL – move HL to PC

Example

```
LXI SP,1FFFH
LXI H, 9320H
LXI B, 4732H
LXI D, ABCDH
MVI A, 34H
PUSH H
PUSH B
PUSH D
PUSH PSW
POP H
POP B
POP D
POP PSW
HLT
```

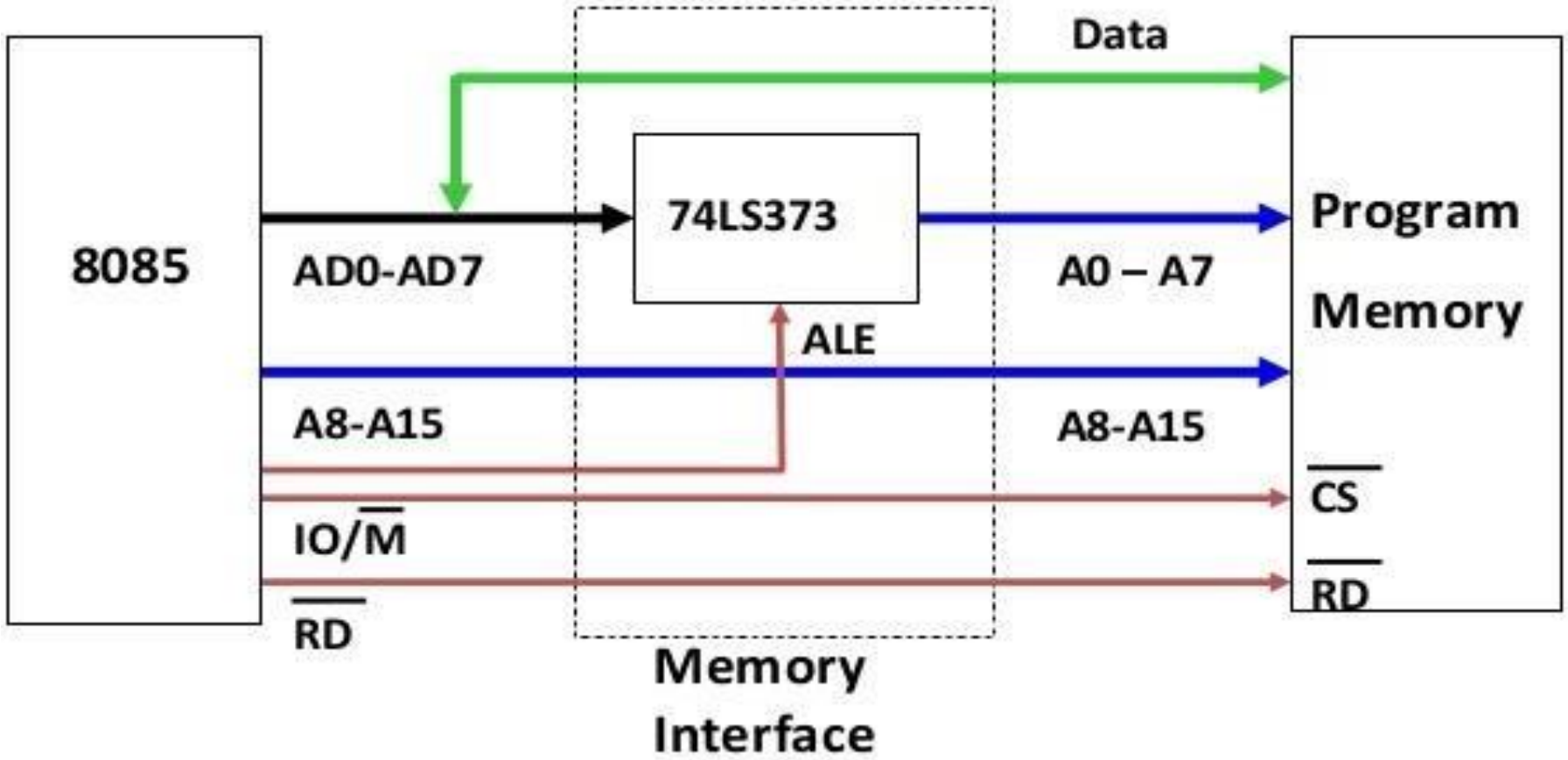
BEORE EXECUTION

H= 93	L= 20
B= 47	C=32
D= AB	E=CD
A= 34	F= 10

AFTER EXECUTION

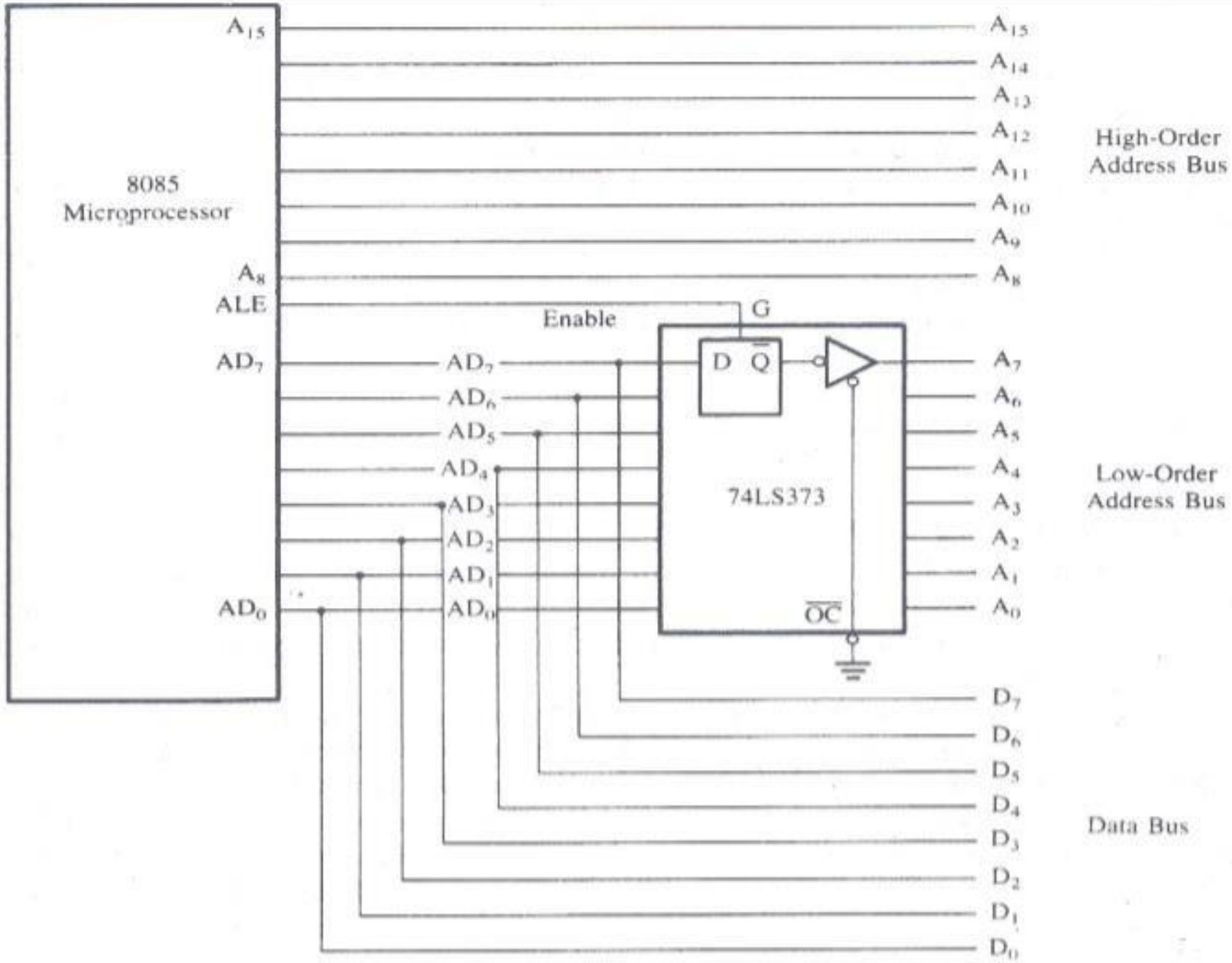
H= 34	L=10
B=AB	C=CD
D= 47	E=32
A= 93	F=20

8085 Interfacing with Memory chips



Multiplexing and Demultiplexing of Address Bus

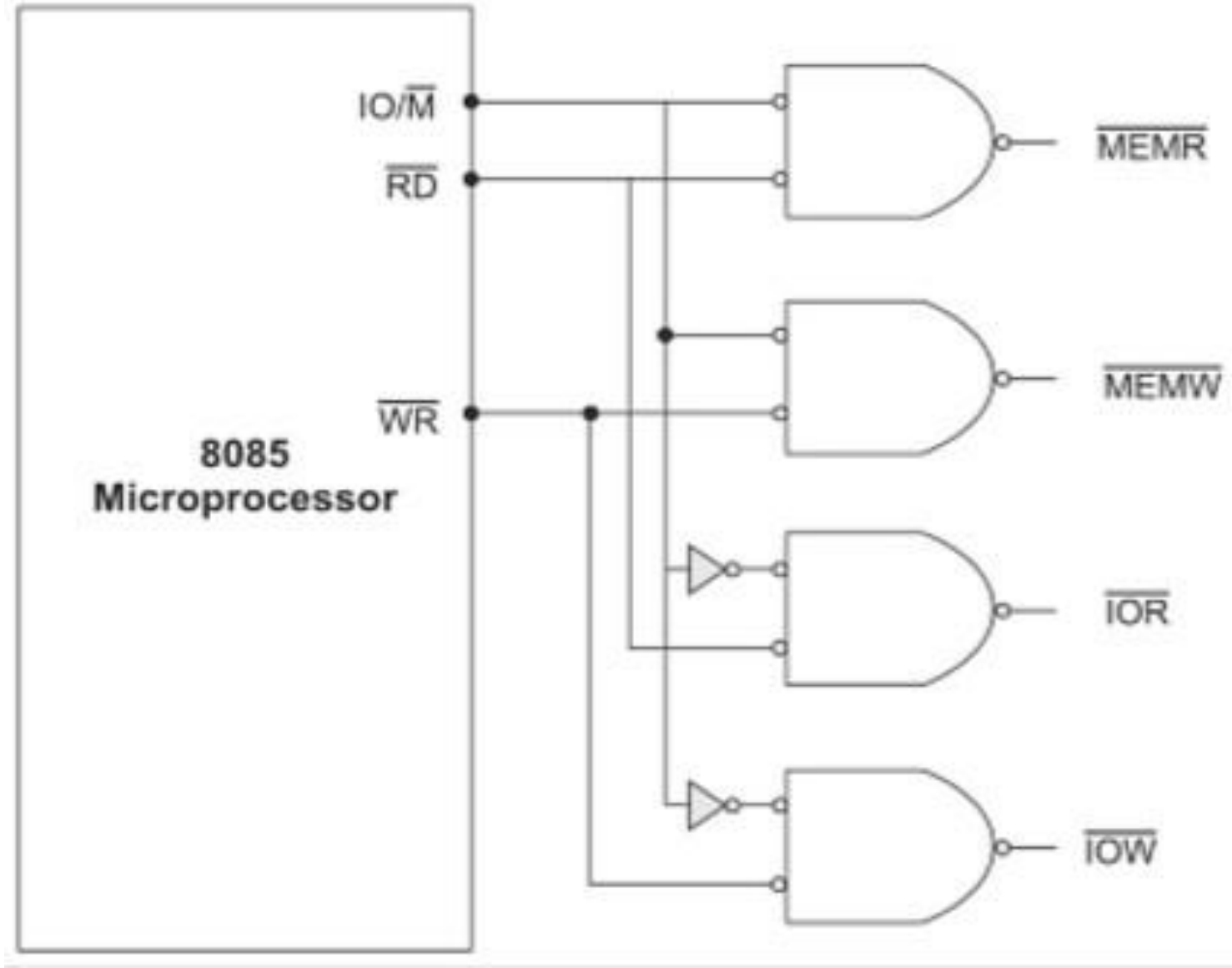
- The higher order address bus(A15-A8)is a uni-directional bus.
- It carries most significant 8-bits of a 16-bit address of memory or I/O device.
- Address remains on lines as long operation is not completed.
- The lower order bus(AD7-AD0) is bidirectional and works on time division multiplexing between address and data.
- During first clock cycle, it serves as a least significant 8-bits of memory/ IO address.
- For second and third clock cycles it acts as data bus and carries data provided that control signal ALE=0


8085
Microprocessor
A15
A8
ALE
AD7
AD0
Enable
G
D Q
74LS373
OC
AD7
AD6
AD5
AD4
AD3
AD2
AD1
AD0
A15
A14
A13
A12
A11
A10
A9
A8
A7
A6
A5
A4
A3
A2
A1
A0
High-Order
Address Bus
Low-Order
Address Bus
D7
D6
D5
D4
D3
D2
D1
D0
Data Bus

Generating control signal

- The Mp provides $\overline{RD}$ and $\overline{WR}$ signals to initiate read and write cycle.
- Because these signals are used both for reading / writing memory or reading writing an input/output device, it is necessary to generate separate read and write signals for memory and I/O devices.
- 8085 provides IO/$\overline{M}$ signal to indicate that initiated cycle is for I/O device or for memory device.
- Using IO/$\overline{M}$ signal along with $\overline{RD}$ and $\overline{WR}$, it is possible to generate four signals as shown in figure below.

Generating control



$IO/\overline{M}$	$\overline{RD}$	$\overline{WR}$	$\overline{MEMR}$ $\overline{RD} + IO/\overline{M}$	$\overline{MEMW}$ $\overline{WR} + IO/\overline{M}$	$\overline{IOR}$ $\overline{RD} + IO/\overline{M}$	$\overline{IOW}$ $\overline{WR} + IO/\overline{M}$
0	0	0	Condition never exists, because $\overline{RD}$ and $\overline{WR}$ signals does not go low simultaneously			
0	0	1	0	1	1	1
0	1	0	1	0	1	1
0	1	1	1	1	1	1
1	0	0	Condition never exists, because $\overline{RD}$ and $\overline{WR}$ signals does not go low simultaneously			
1	0	1	1	1	0	1
1	1	0	1	1	1	0
1	1	1	1	1	1	1

Basic Interfacing Concepts

- **Interface** is the path for communication between two components. Interfacing is of two types, memory interfacing and I/O interfacing.

Memory Interfacing:

- While executing an instruction, there is a necessity for the microprocessor to access memory frequently for reading various instruction codes and data stored in the memory. The interfacing circuit aids in accessing the memory.
- Memory requires some signals to read from and write to registers. Similarly the microprocessor transmits some signals for reading or writing a data.
- The interfacing process involves matching the memory requirements with the microprocessor signals. The interfacing circuit therefore should be designed in such a way that it matches the memory signal requirements with the signals of the microprocessor.

- For example for carrying out a READ process, the microprocessor should initiate a read signal which the memory requires to read a data. In simple words, the primary function of a memory interfacing circuit is to aid the microprocessor in reading and writing a data to the given register of a memory chip.

I/O Interfacing:

- We know that keyboard and Displays are used as communication channel with outside world. So it is necessary that we interface keyboard and displays with the microprocessor. This is called I/O interfacing. In this type of interfacing we use latches and buffers for interfacing the keyboards and displays with the microprocessor.
- But the main disadvantage with this interfacing is that the microprocessor can perform only one function. It functions as an input device if it is connected to buffer and as an output device if it is connected to latch. Thus the capability is very limited in this type of interfacing.

- There are two ways of communication in which the microprocessor can connect with the outside world.
  - Serial Communication Interface
  - Parallel Communication interface
- **Serial Communication Interface** − In this type of communication, the interface gets a single byte of data from the microprocessor and sends it bit by bit to the other system serially and vice-a-versa.
- **Parallel Communication Interface** − In this type of communication, the interface gets a byte of data from the microprocessor and sends it bit by bit to the other systems in simultaneous (or) parallel fashion and vice-a-versa.

Programmable Peripheral Devices

Programmable peripheral devices were introduced by Intel to increase the overall performance of the system. These devices along with I/O functions, they perform various other functions such as time delays, counters and interrupt handling. These devices are nothing but a combination of many devices on a single chip. A programmable device can be set up to perform specific function by writing a code in the internal register. As this code controls the function of the device it's called **control word** and internal register in which it is stored is called **Control Register**.

INTEL developed some peripheral devices for processors like 8085/8086/8088. The peripheral devices includes

8255 – Parallel Communication Interface (PPI)

8251 – Serial communication Interface (USART- Universal Synchronous/Asynchronous Receiver/Transmitter)

8257 – DMA Controller

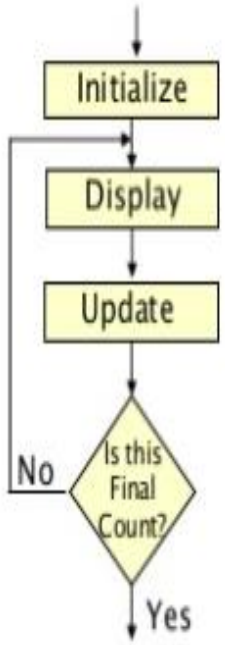
8279 – Keyboard/Display Controller

8259 – Programmable Interrupt controller

8254 – Programmable Timer

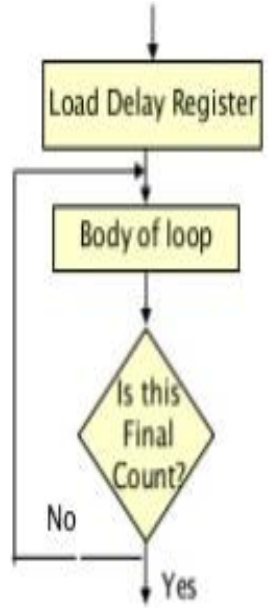
Developing Counters and Time Delay Routines

- Counters are used primarily to keep track of events.
- Time delays are important in setting up reasonably accurate timing between two events.
- The process of designing counters and time delays using software instructions is far more flexible and less time consuming than the design process using hardware.



Flowchart of a Counter

A Counter is designed simply by loading an appropriate number into one of the registers and using the INR (Increment by one) or the DCR (Decrement by one) 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.



Flowchart of a 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, and then the register is decremented until it reaches zero by setting up a loop with a conditional jump instruction. The loop causes the delay, depending upon the clock period of the system.

Calculating Time Delays

- Each instruction passes through different combinations of Opcode Fetch, Memory Read, and Memory Write cycles.
- Knowing the combinations of cycles, one can calculate how long such an instruction would require to complete.
  - Number of Bytes
  - Number of Machine Cycles
  - Number of T-State.

- Knowing how many T-States an instruction requires, and keeping in mind that a T-State is one clock cycle long, we can calculate the time delay using the following formula:

Time Delay = No. of T-States * Clock Period

Clock period = 1 / frequency

- For example,

"MVI" instruction uses 7 T-States. Therefore, if the Microprocessor is running at 2 MHz, the instruction would require 3.5 μS to complete.

We can design Time Delay using following three Techniques:

1. Using One Register.
2. Using a Register Pair.
3. Using a Loop with in a Loop

Using One Register

- A count is loaded in a register, and We can use a loop to produce a certain amount of time delay in a program.
- The following is an example of a delay using One Register:

```
        MVI C, FFH      7 T-States
LOOP    DCR C           4 T-States
        JNZ LOOP        10 T-States / 7 T- States
```

- The first instruction initializes the loop counter and is executed only once requiring only 7 T-States.
- The following two instructions form a loop that requires 14 T-States to execute and is repeated 255 times until C becomes 0. FFH -255

Using One Register

- We need to keep in mind though that in the last iteration of the loop, the JNZ instruction will fail and require only 7 T-States rather than the 10.
- Therefore, we must deduct 3 T-States from the total delay to get an accurate delay calculation.
- To calculate the delay, we use the following formula:

$$T_{delay} = T_O + T_L$$

T_{delay} = total delay
T_O = delay outside the loop
T_L = delay of the loop

- T_O is the sum of all delays outside the loop.

- T_L is calculated using the formula

$$T_L = T * \text{Loop T-States} * N \text{ (no. of iterations)}$$

Using One Register

- Using these formulas, we can calculate the time delay for the previous example:
- T_O = 7 T-States
  (Delay of the MVI instruction)
- T_L = (14 X 255) - 3 = 3567 T-States
  (14 T-States for the 2 instructions repeated 255 times
  ($FF_{16} = 255_{10}$) reduced by the 3 T-States for the final JNZ.)
- $T_{delay} = [(T_O + T_L)/f]$
  $= (7 + 3567)/2MHz$
  $= (3574)$ X 0.5 μSec
  $= 1.787$ mSec
  (Assuming **f** = 2 MHz)

Using a Register Pair

- Using a single register, one can repeat a loop for a maximum count of 255 times.
- It is possible to increase this count by using a register pair for the loop counter instead of the single register.
  - A minor problem arises in how to test for the final count since DCX and INX do not modify the flags.
  - However, if the loop is looking for when the count becomes zero, we can use a small trick by ORing the two registers in the pair and then checking the zero flag.

Using a Register Pair

- The following is an example of a delay loop set up with a register pair as the loop counter.

	LXI B, 1000H	10 T-States
LOOP	DCX B	6 T-States
	MOV A, C	4 T-States
	ORA B	4 T-States
	JNZ LOOP	10 T-States / 7 T- States

Using a Register Pair

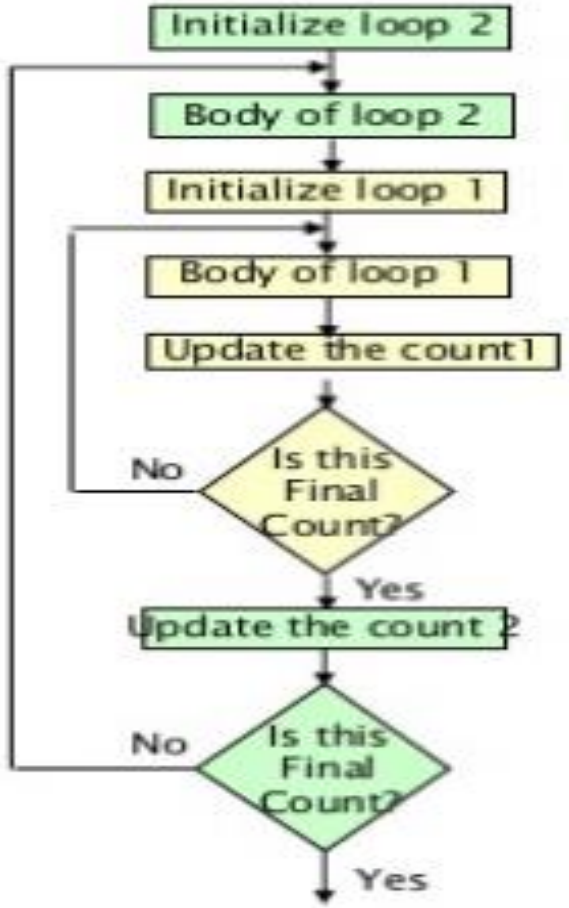
- Using the same formula from before, we can calculate:

- T_O = 10 T-States
  (The delay for the LXI instruction)

- T_L = (24 X 4096) - 3 = 98301 T- States
  (24 T-States for the 4 instructions in the loop repeated 4096 times ($1000_{16} = 4096_{10}$) reduced by the 3 T-States for the JNZ in the last iteration.)

- T_{Delay} = (10 + 98301) X 0.5 mSec = 49.155 mSec

Using a Loop within a Loop



- Nested loops can be easily setup in Assembly language by using two registers for the two loop counters and updating the right register in the right loop.

Using a Loop with in a Loop

- Instead (or in conjunction with) Register Pairs, a nested loop structure can be used to increase the total delay produced.

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

Using a Loop with in a Loop

- The calculation remains the same except that the formula must be applied recursively to each loop.
  Start with the inner loop, then plug that delay in the calculation of the outer loop.

- Delay of inner loop,

  T_{O1} = 7 T-States
  (MVI C, FFH instruction)
  T_{L1} = (255 X 14) - 3 = 3567 T-States
  (14 T-States for the DCR C and JNZ instructions repeated 255 times ($FF_{16} = 255_{10}$) minus 3 for the final JNZ.)
  T_{LOOP1} = 7 + 3567 = 3574 T-States

Using a Loop with in a Loop

- **Delay of outer loop**

  T_{O2} = 7 T-States

  (MVI B, 10H instruction)

  $T_{L1} = (16 \text{ X } (14 + 3574)) - 3 = 57405$ T-States

  (14 T-States for the DCR B and JNZ instructions and 3574 T-States for loop1 repeated 16 times ($10_{16} = 16_{10}$) minus 3 for the final JNZ.)

  $T_{Delay} = 7 + 57405 = 57412$ T-States

- **Total Delay**

  $T_{Delay} = 57412 \text{ X } 0.5\ \mu\text{Sec} = 28.706$ mSec

PROGRAMS