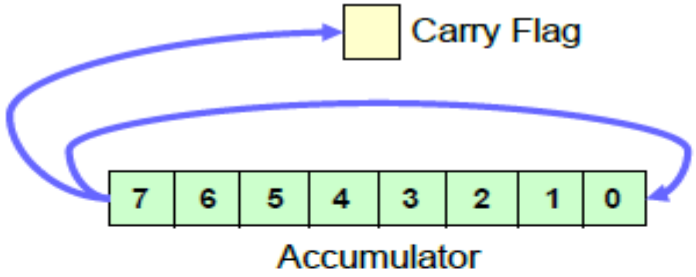
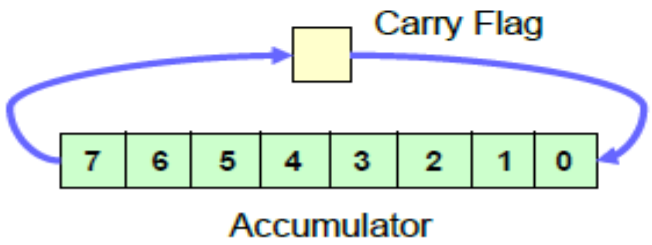
RLC vs. RAL

- RLC



- RAL



Logical Operations

- Compare
  - Compare the contents of a register or memory location with the contents of the accumulator.
    - CMP R/M Compare the contents of the register or memory location to the contents of the accumulator.
    - CPI # Compare the 8-bit number to the contents of the accumulator.
  - The compare instruction sets the flags (Z, Cy, and S).
  - The compare is done using an internal subtraction that does not change the contents of the accumulator.

    A – (R / M / #)

Branch Operations

- Two types:
  - Unconditional branch.
    - Go to a new location no matter what.
  - Conditional branch.
    - Go to a new location if the condition is true.

Unconditional Branch

- JMP Address
  - Jump to the address specified (Go to).
- CALL Address
  - Jump to the address specified but treat it as a subroutine.
- RET
  - Return from a subroutine.
- The addresses supplied to all branch operations must be 16-bits.

Conditional Branch

- Go to new location if a specified condition is met.
  - JZ Address (Jump on Zero)
    - Go to address specified if the **Zero flag is set**.
  - JNZ Address (Jump on NOT Zero)
    - Go to address specified if the **Zero flag is not set**.
  - JC Address (Jump on Carry)
    - Go to the address specified if the **Carry flag is set**.
  - JNC Address (Jump on No Carry)
    - Go to the address specified if the **Carry flag is not set**.
  - JP Address (Jump on Plus)
    - Go to the address specified if the **Sign flag is not set**
  - JM Address (Jump on Minus)
    - Go to the address specified if the **Sign flag is set**.

Machine Control

- HLT
  - Stop executing the program.
- NOP
  - No operation
  - Exactly as it says, do nothing.
  - Usually used for delay or to replace instructions during debugging.

Operand Types

- There are different ways for specifying the operand:
  - There may not be an operand (implied operand)
    - CMA
  - The operand may be an 8-bit number (immediate data)
    - ADI 4FH
  - The operand may be an internal register (register)
    - SUB B
  - The operand may be a 16-bit address (memory address)
    - LDA 4000H

Instruction Size

- Depending on the operand type, the instruction may have different sizes. It will occupy a different number of memory bytes.
  - Typically, all instructions occupy one byte only.
  - The exception is any instruction that contains immediate data or a memory address.
    - Instructions that include immediate data use two bytes.
      - One for the opcode and the other for the 8-bit data.
    - Instructions that include a memory address occupy three bytes.
      - One for the opcode, and the other two for the 16-bit address.

Instruction with Immediate Data

- Operation: Load an 8-bit number into the accumulator.
  - MVI A, 32
    - Operation: MVI A
    - Operand: The number 32
    - Binary Code:

0011 1110	3E	1st byte.
0011 0010	32	2nd byte.

Instruction with a Memory Address

- Operation: go to address 2085.
  - Instruction: JMP 2085
    - Opcode: JMP
    - Operand: 2085
    - Binary code:

1100 0011	C3	1st byte.
1000 0101	85	2nd byte
0010 0000	20	3rd byte

Addressing Modes

- The microprocessor has different ways of specifying the data for the instruction. These are called "addressing modes".
- The 8085 has four addressing modes:

– Implied	CMA
– Immediate	MVI B, 45
– Direct	LDA 4000
– Indirect	LDAX B

Data Formats

- In an 8-bit microprocessor, data can be represented in one of four formats:
  - ASCII
  - BCD
  - Signed Integer
  - Unsigned Integer.
- It is important to recognize that the microprocessor deals with 0's and 1's.
  - It deals with values as strings of bits.
  - It is the job of the user to add a meaning to these strings.

Data Formats

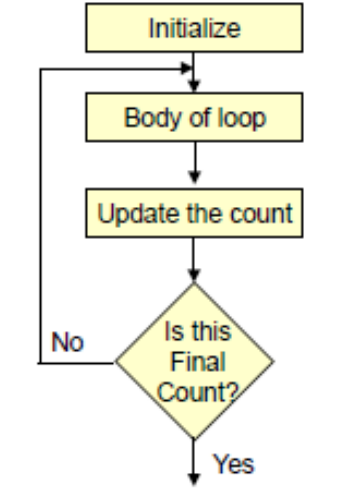
- Assume the accumulator contains the following value: 0100 0001. There are four ways of reading this value:
  - It is an unsigned integer expressed in binary, the equivalent decimal number would be 65.
  - It is a number expressed in BCD (Binary Coded Decimal) format. That would make it, 41.
  - It is an ASCII representation of a letter. That would make it the letter A.
  - It is a string of 0's and 1's where the 0^{th} and the 6^{th} bits are set to 1 while all other bits are set to 0.

Counters

- A loop counter is set up by loading a register with a certain value
- Then using the DCR (to decrement) and INR (to increment) the contents of the register are updated.
- A loop is set up with a conditional jump instruction that loops back or not depending on whether the count has reached the termination count.

Counters

- The operation of a loop counter can be described using the following flowchart.



Sample ALP for implementing a loop Using DCR instruction

```
        MVI C, 15H
LOOP    DCR C
        JNZ LOOP
```

Register Pair as a Loop Counter

- 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.

Register Pair as a Loop Counter

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

```
        LXI B, 1000H
LOOP    DCX B
        MOV A, C
        ORA B
        JNZ LOOP
```

Delays

- Knowing how many T-States an instruction requires, we can calculate the time using the following formula:
    - Delay = No. of T-States / Frequency
- For example a “MVI” instruction uses 7 T-States. Therefore, if the Microprocessor is running at 2 MHz, the instruction would require 3.5 μSeconds to complete.

Delay loops

- We can use a loop to produce a certain amount of time delay in a program.

- The following is an example of a delay loop:

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

Delay Loops (Contd.)

- 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.

Delay Loops (Contd.)

- 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.

Register Pair as a Loop Counter

- 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
```

Register Pair as a Loop Counter

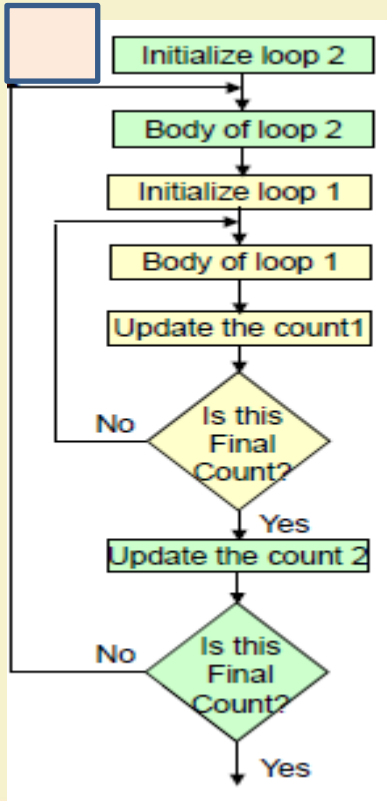
- Using the same formula from before, we can calculate:

- 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.

Nested Loops

- 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.
  - –In the figure, the body of loop2 can be before or after loop1.



Nested Loops for Delay

- 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
	DCR B	4 T-States
	JNZ LOOP2	10 T-States

Delay Calculation of Nested Loops

- The calculation remains the same except that it 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= 255) minus 3 for the final JNZ

Delay Calculation of Nested Loops

- Delay of outer loop
  - T_{O2} = 7 T-States
    - MVI B, 10H instruction
  - T_{L1} = (16 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 X 0.5 μSec = 28.706 mSec

Increasing the delay

- The delay can be further increased by using register pairs for each of the loop counters in the nested loops setup.
- It can also be increased by adding dummy instructions (like NOP) in the body of the loop.

Timing Diagram

- Representation of Various Control signals generated during Execution of an Instruction.
- Following Buses and Control Signals must be shown in a Timing Diagram:
  - Higher Order Address Bus.
  - Lower Address/Data bus
  - ALE
  - RD
  - WR
  - IO/M

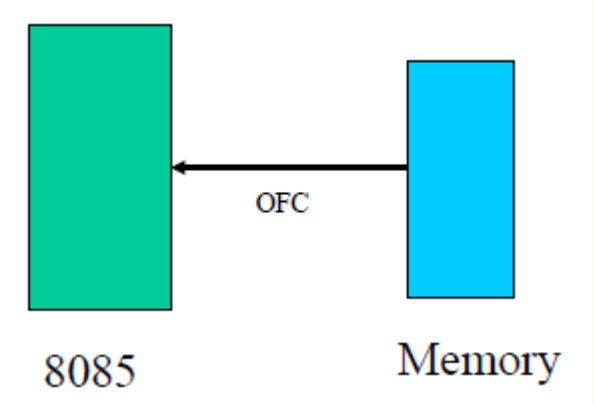
Timing Diagram

Instruction:

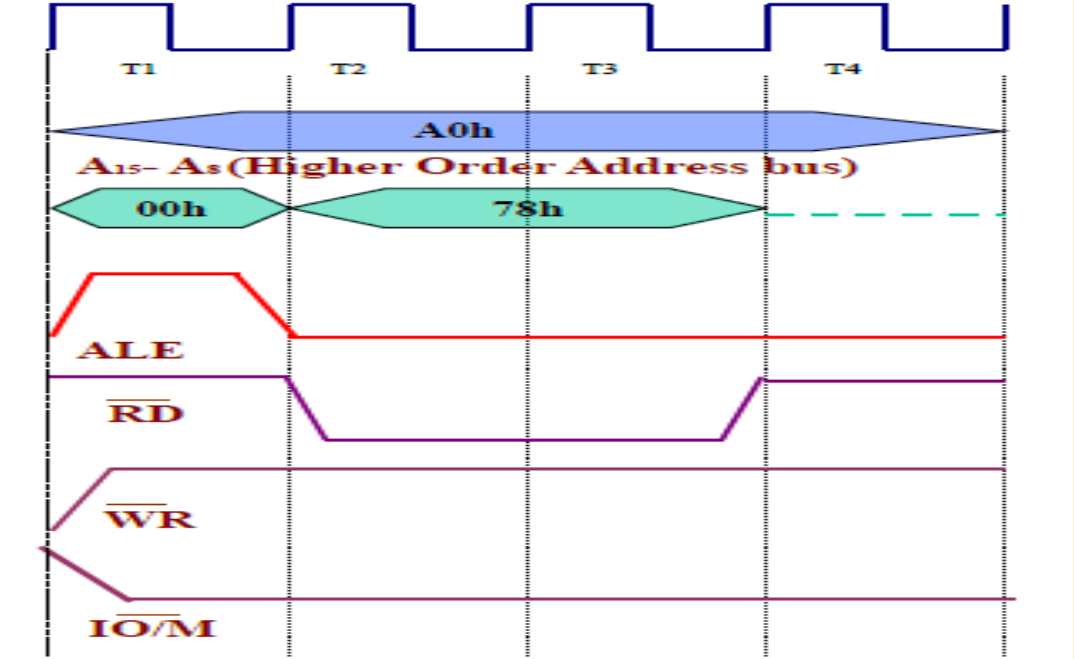
A000h MOV A,B

Corresponding Coding:

A000h 78



Timing Diagram



Op-code fetch Cycle

Timing Diagram

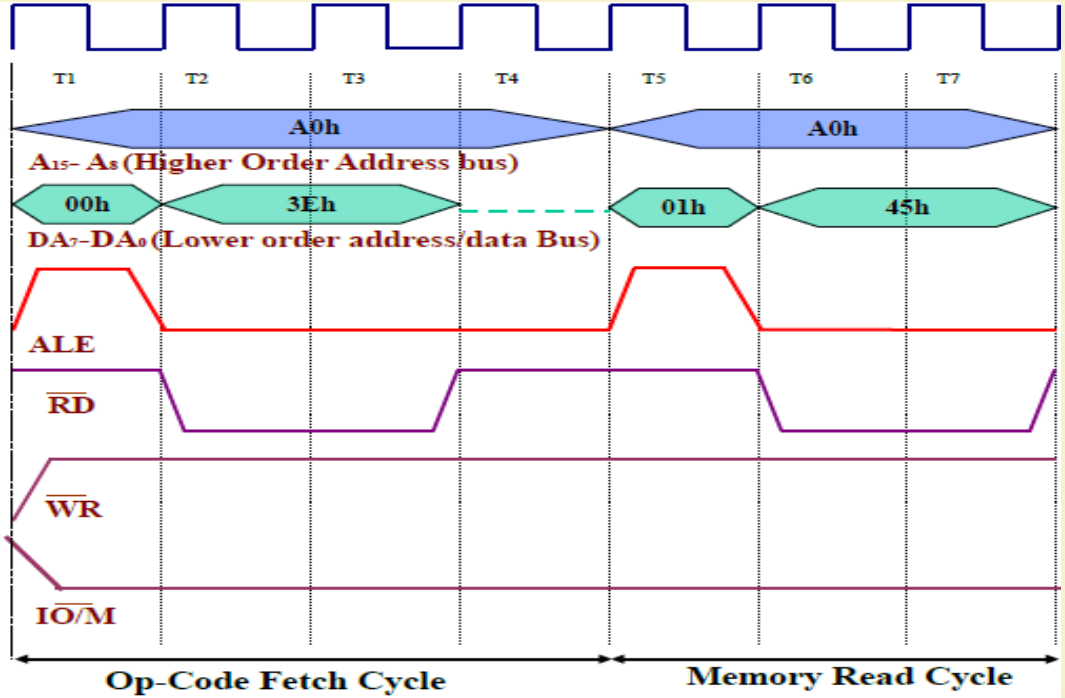
Instruction:

A000h MVI A,45h

Corresponding Coding:

A000h 3E

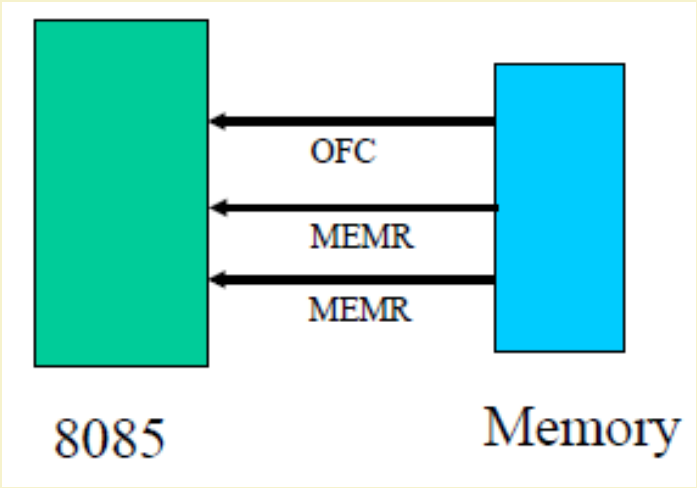
A001h 45



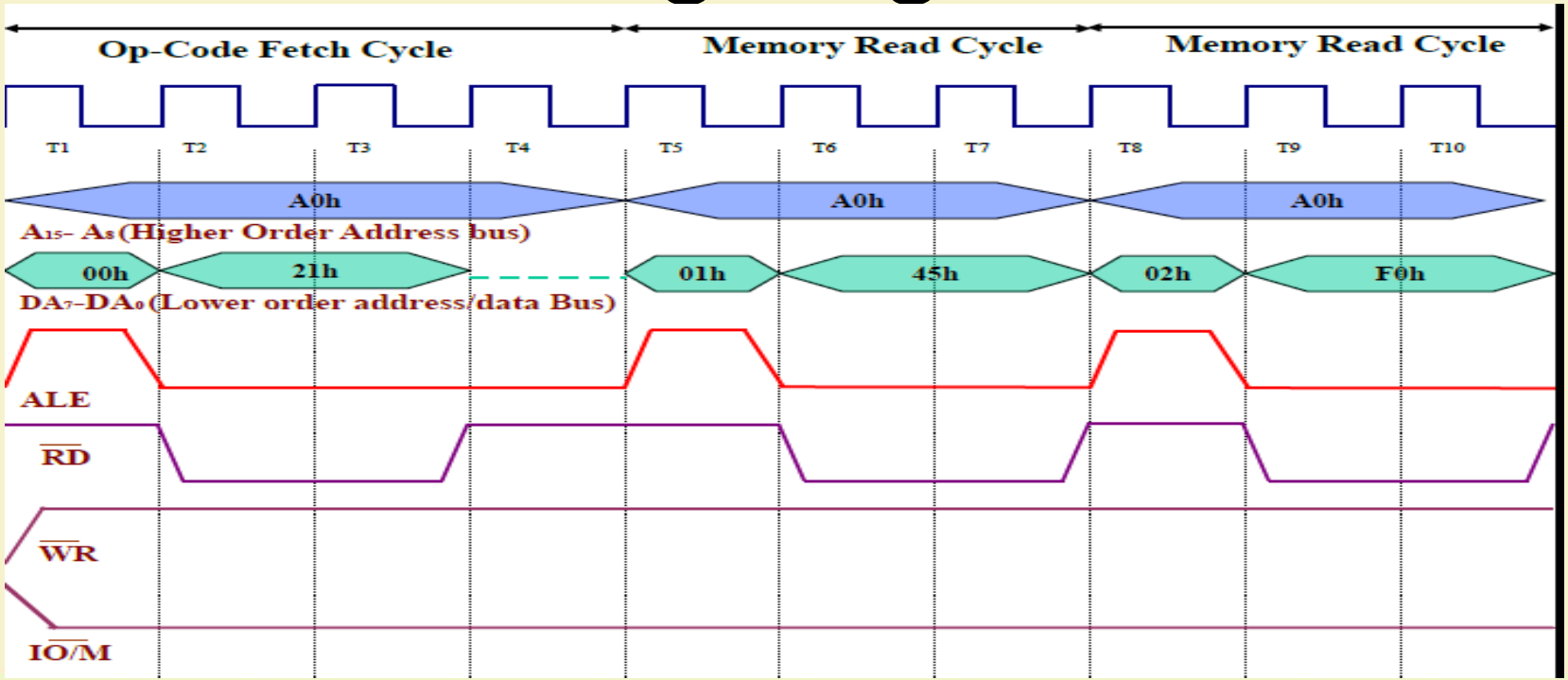
Timing Diagram

- Instruction:
- A000h LXI A,FO45h
- Corresponding Coding:

A000h	21
A001h	45
A002h	F0



Timing Diagram



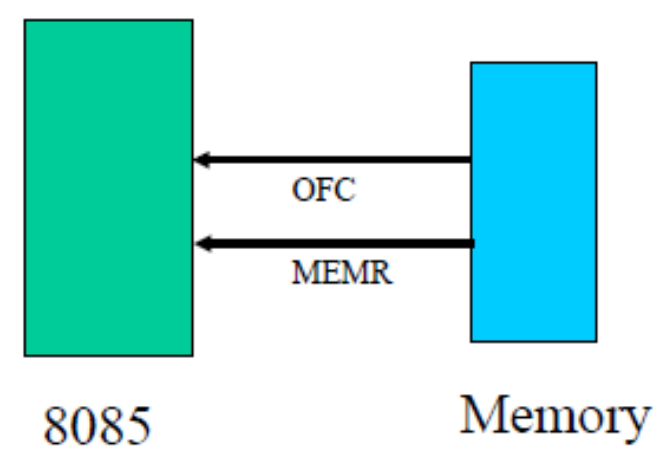
Timing Diagram

Instruction:

A000h MOV A,M

Corresponding Coding:

A000h 7E



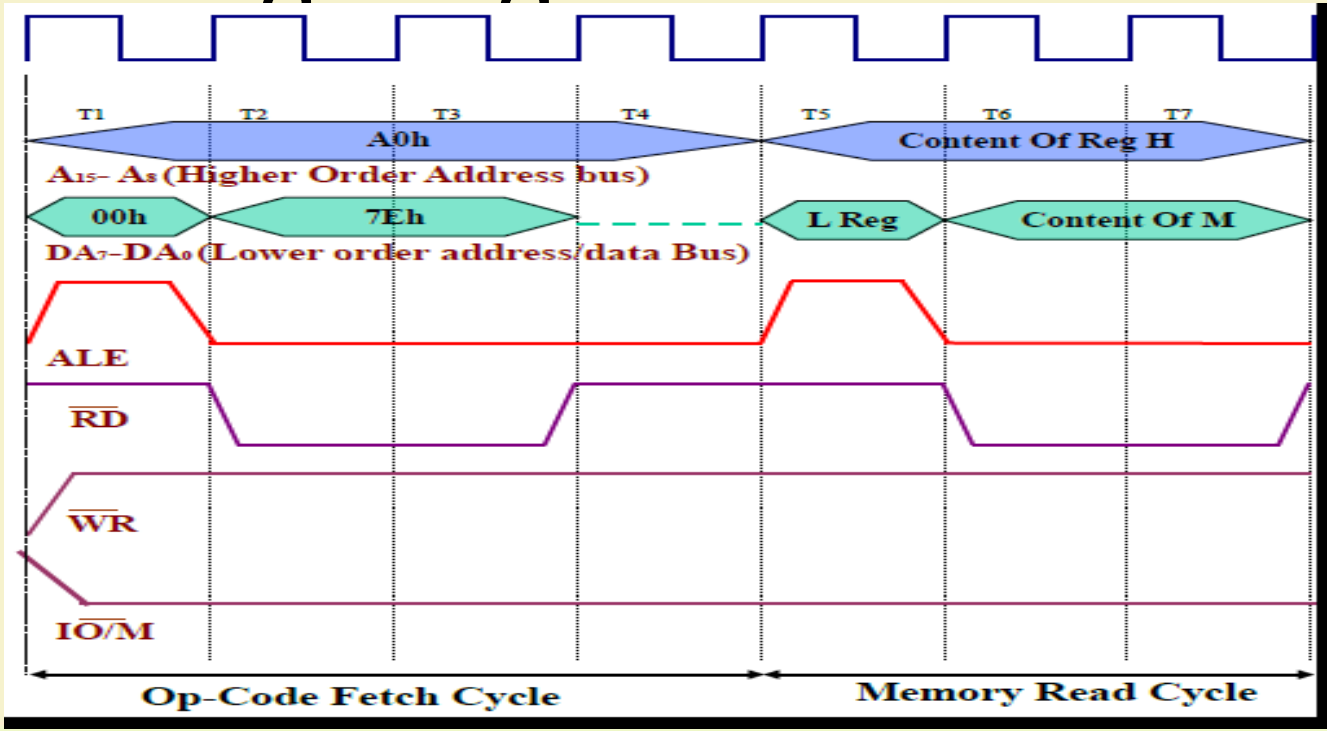
Timing Diagram

Instruction:

A000h MOV A,M

Corresponding Coding:

A000h 7E



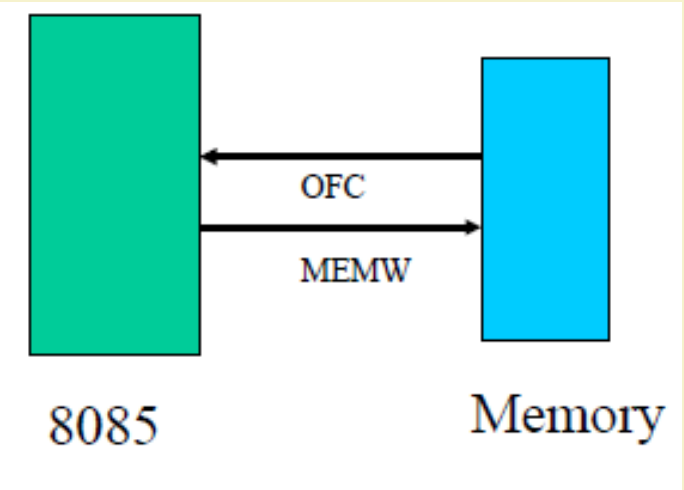
Timing Diagram

Instruction:

A000h MOV M,A

Corresponding Coding:

A000h 77



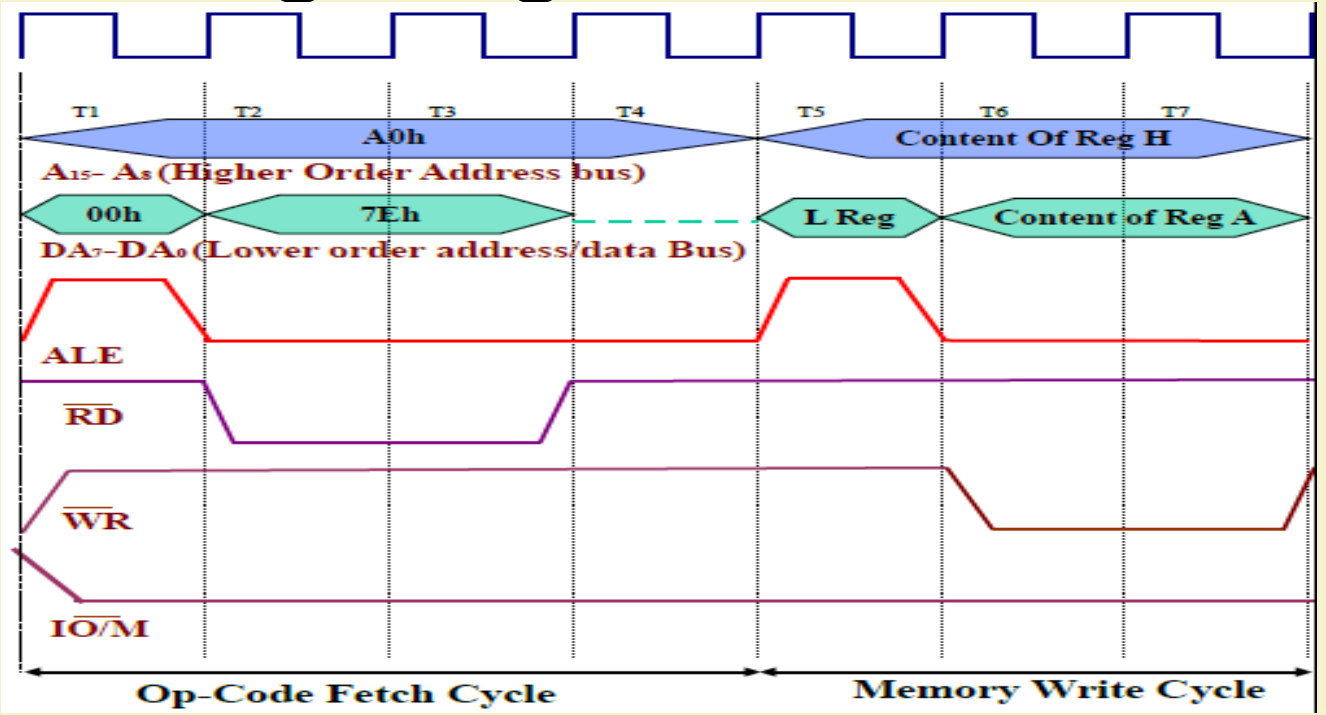
Timing Diagram

Instruction:

A000h MOV M,A

Corresponding Coding:

A000h 77



The Stack

- The stack is an area of memory identified by the programmer for temporary storage of information.
- The stack is a LIFO structure.
  - Last In First Out.

- The stack normally grows backwards into memory.
  - In other words, the programmer defines the bottom of the stack and the stack grows up into reducing address range.

The Stack

- Given that the stack grows backwards into memory, it is customary to place the bottom of the stack at the end of memory to keep it as far away from user programs as possible.
- In the 8085, the stack is defined by setting the SP (Stack Pointer) register.
- LXI SP, FFFFH
- This sets the Stack Pointer to location FFFFH (end of memory for the 8085).

Saving Information on the Stack

- Information is saved on the stack by PUSHing it on.
  - It is retrieved from the stack by POPing it off.
- The 8085 provides two instructions: PUSH and POP for storing information on the stack and retrieving it back.
  - Both PUSH and POP work with register pairs ONLY.

The PUSH Instruction

- PUSH B
  - Decrement SP
  - Copy the contents of register B to the memory location pointed to by SP

The POP Instruction

- POP D
  - Copy the contents of the memory location pointed to by the SP to register D
  - Increment SP

Operation of the Stack

- During pushing, the stack operates in a "decrement then store" style.
  - The stack pointer is decremented first, then the information is placed on the stack.
- During poping, the stack operates in a "use then increment" style.
  - The information is retrieved from the top of the the stack and then the pointer is incremented.
- The SP pointer always points to "the top of the stack".

LIFO

- The order of PUSHs and POPs must be opposite of each other in order to retrieve information back into its original location.

```
PUSH B
PUSH D
...
POP D
POP B
```

The PSW Register Pair

- The 8085 recognizes one additional register pair called the PSW (Program Status Word).
  - This register pair is made up of the Accumulator and the Flags registers.
- It is possible to push the PSW onto the stack, do whatever operations are needed, then POP it off of the stack.
  - The result is that the contents of the Accumulator and the status of the Flags are returned to what they were before the operations were executed.

Subroutines

- A subroutine is a group of instructions that will be used repeatedly in different locations of the program.
  - Rather than repeat the same instructions several times, they can be grouped into a subroutine that is called from the different locations.
- In Assembly language, a subroutine can exist anywhere in the code.
  - However, it is customary to place subroutines separately from the main program.

Subroutines

- The 8085 has two instructions for dealing with subroutines.
  - The CALL instruction is used to redirect program execution to the subroutine.
  - The RET instruction is used to return the execution to the calling routine.

The CALL Instruction

- CALL 4000H
  - Push the address of the instruction immediately following the CALL onto the stack
  - Load 4000H to PC

The RET Instruction

- RET
  - Retrieve the return address from the top of the stack
  - Load the program counter with the return address

Cautions

- The CALL instruction places the return address at the two memory locations immediately before where the Stack Pointer is pointing.
  - You must set the SP correctly BEFORE using the CALL instruction.
- The RET instruction takes the contents of the two memory locations at the top of the stack and uses these as the return address.
  - Do not modify the stack pointer in a subroutine. You will loose the return address.

Passing Data to a Subroutine

- In Assembly Language data is passed to a subroutine through registers.
  - The data is stored in one of the registers by the calling program and the subroutine uses the value from the register.
- The other possibility is to use agreed upon memory locations.
  - The calling program stores the data in the memory location and the subroutine retrieves the data from the location and uses it.

Call by Reference and Call by Value

- If the subroutine performs operations on the contents of the registers, then these modifications will be transferred back to the calling program upon returning from a subroutine.
  - Call by reference
- If this is not desired, the subroutine should PUSH all the registers it needs on the stack on entry and POP them on return.
  - The original values are restored before execution returns to the calling program.

Cautions with PUSH and POP

- PUSH and POP should be used in opposite order.
- There has to be as many POP's as there are PUSH's.
  - –If not, the RET statement will pick up the wrong information from the top of the stack and the program will fail.
- It is not advisable to place PUSH or POP inside a loop.

Conditional CALL and RET Instructions

- The 8085 supports conditional CALL and conditional RET instructions.
  - The same conditions used with conditional JUMP instructions can be used.
  - CC, call subroutine if Carry flag is set.
  - CNC, call subroutine if Carry flag is not set
  - RC, return from subroutine if Carry flag is set
  - RNC, return from subroutine if Carry flag is not set
  - Etc.

A Proper Subroutine

- According to Software Engineering practices, a proper subroutine:
  - Is only entered with a CALL and exited with an RET
  - Has a single entry point
    - Do not use a CALL statement to jump into different points of the same subroutine.
  - Has a single exit point
    - There should be one return statement from any subroutine.
- Following these rules, there should not be any confusion with PUSH and POP usage.

Interrupts

- Interrupt is a process where an external device can get the attention of the microprocessor.
  - The process starts from the I/O device
  - The process is asynchronous.
- Interrupts can be classified into two types:
  - Maskable (can be delayed)
  - Non-Maskable (can not be delayed)
- Interrupts can also be classified into:
  - Vectored (the address of the service routine is hard-wired)
  - Non-vectored (the address of the service routine needs to be supplied externally)

Interrupts

- An interrupt is considered to be an emergency signal.
  - The Microprocessor should respond to it as soon as possible.
- When the Microprocessor receives an interrupt signal, it suspends the currently executing program and jumps to an Interrupt Service Routine (ISR) to respond to the incoming interrupt.
  - Each interrupt will most probably have its own ISR.

Responding to Interrupts

- Responding to an interrupt may be immediate or delayed depending on whether the interrupt is maskable or non-maskable and whether interrupts are being masked or not.
- There are two ways of redirecting the execution to the ISR depending on whether the interrupt is vectored or non-vectored.
  - The vector is already known to the Microprocessor
  - The device will have to supply the vector to the Microprocessor

The 8085 Interrupts

- The maskable interrupt process in the 8085 is controlled by a single flip flop inside the microprocessor. This Interrupt Enable flip flop is controlled using the two instructions “EI” and “DI”.
- The 8085 has a single Non-Maskable interrupt.
  - The non-maskable interrupt is not affected by the value of the Interrupt Enable flip flop.

The 8085 Interrupts

- The 8085 has 5 interrupt inputs.
  - The INTR input.
    - The INTR input is the only non-vectored interrupt.
    - INTR is maskable using the EI/DI instruction pair.
  - RST 5.5, RST 6.5, RST 7.5 are all automatically vectored.
    - RST 5.5, RST 6.5, and RST 7.5 are all maskable.
  - TRAP is the only non-maskable interrupt in the 8085
    - TRAP is also automatically vectored

The 8085 Interrupts

Interrupt name	Maskable	Vectored
INTR	Yes	No
RST 5.5	Yes	Yes
RST 6.5	Yes	Yes
RST 7.5	Yes	Yes
TRAP	No	Yes

Interrupt Vectors and the Vector Table

- An interrupt vector is a pointer to where the ISR is stored in memory.
- All interrupts (vectored or otherwise) are mapped onto a memory area called the Interrupt Vector Table (IVT).
  - The IVT is usually located in memory page 00 (0000H- 00FFH).
  - The purpose of the IVT is to hold the vectors that redirect the microprocessor to the right place when an interrupt arrives.
  - The IVT is divided into several blocks. Each block is used by one of the interrupts to hold its "vector"