Instruction Timing and Execution in 8085

8085 based system

- To make a complete system, the 8085 microprocessor needs to be interfaced with the **memories** and **input** and **output devices**.
- The memories required are of two types – RAM and ROM.
- ROM is necessary to store some amount of fixed programs.
- These programs are executed when the system is powered on and are essential for the system.

8085 based system Contd..

- RAM is required in any system to store temporary **programs** and **data**.
- A particular memory location is selected from the memory devices by properly issuing the address for that memory location and issuing the control signals discussed in the previous section.

8085 based system contd..

- The microprocessor is the **master** in any microcomputer system and the microprocessor issues the required control signals to the peripherals.
- Each location in a memory is given a number, called an **address**. The maximum number of locations that can be addressed will depend on the number of bits in the address.
- In general, **2^n** is the number of memory locations addressed where **n** is the number of bits in the address.

MICROPROCESSOR INSTRUCTIONS

- Every microprocessor has its own **instruction set**.
- Based on the design of the ALU and the decoding unit, the microprocessor manufacturers generally list out the instruction set for the every microprocessor.
- The instruction set consists of both the assembly language mnemonics and the corresponding machine code.

INSTRUCTION SET

- An instruction is a binary **bit pattern** that can be **decoded** inside a microprocessor to perform a specific function.
- The assembly language mnemonics are the codes for these binary patterns so that the **user can easily** understand the function performed by these instructions.
- The entire group of instructions is called the instruction set, and this determines the functionalities that the microprocessor can perform.

Classification of Instruction set

- Based on the **length** of the instructions, the instruction set can be classified into following types.

a) **One-byte** instructions,

b) **Two-byte** instructions and

c) **Three-byte** instructions etc.

One-Byte Instructions

- Instructions that require **only one byte** in **machine language** are called one-byte instructions.
- These instructions just have the machine code or opcode alone to represent the operation to be performed.
- The common examples are the instructions that have their operands within the processor itself **e.g. MOV A,B** (opcode 78).

Example - One-Byte Instructions

Opcode	Operand	Machine code/Opcode/ Hex code
MOV	A, B	78
ADD	M	86
XRA	A	AF

Two-Byte Instructions

Opcode	Operand	Machine code / Opcode/ Hex code	Byte description
MVI	A, 7FH	3E 7F	First Byte Second Byte
ADI	0FH	C6 OF	First Byte Second Byte
IN	40H	DB 40	First Byte Second Byte

In a two-byte instruction, the **first byte** specifies the **operation code** and the **second byte** specifies the **operand.**

Three-Byte Instructions

- Instructions that require three bytes in machine code are called three-byte instructions.
- In 8085 machine language, the first byte of the three-byte instructions is the opcode which specifies the operation to be performed.
- The next two bytes refer to the 16-bit operand, which is either a 16-bit number or the **address** of a memory location.

Three-Byte Instructions Examples

Opcode	Operand	Hex Code	Byte description
JMP	2085H	C3 85 20	First byte Second Byte Third Byte
LDA	8850H	3A 50 88	First byte Second Byte Third Byte
LXI	H, 0520H	21 20 05	First byte Second Byte Third Byte

Format of Assembly language instructions and programs

- The assembly language programs are written for performing the specific function and are converted into the machine language code and then stored in the memory of the microprocessor based system.
- The conversion of assembly language program into machine language code is called as Assembling and the application that performs this task is called **assembler**.

Format of Assembly language programs contd..

Memory address	Machine code / opcode	Label	Mnemonics with operands	comments
8000	3E	START:	MVI A, 5FH	load data in the accumulator
8001	5F			
8002				Address of the next memory location

INSTRUCTION EXECUTION AND TIMING DIAGRAM WITH MACHINE CYCLES AND T STATES

- The 8085 microprocessor is designed to **fetch** the instruction **pointed** by the program counter and then **decode** and **execute** the instruction within the processor.
- If necessary, further operand fetch will take place before completing the execution.
- Each instruction has two parts:

The operation code, known as **opcode** and another part - **operand.**

TIMING DIAGRAM

- The opcode is a command such as ADD, and the operand is an object to be operated on, for ex. a byte or the contents of a register.
- To complete the execution, 8085 needs to perform various operations such as **opcode fetch, Memory Read / Write** or **I/O Read / Write.**

External Communication Function

- The microprocessor external communication function can be divided into three categories:

a) Memory read/write

b) I/O read and write

c) Interrupt Request Acknowledge

Definitions

- **1. Instruction Cycle**

  Defined as the **time** taken by the processor to **complete, *execution of an instruction*** (i.e. **time required to execute one instruction**). Instruction cycle consists of 1 to 5 machine cycles.

- **2. Machine Cycle**:

  The time required to complete **one operation** of accessing memory or an I/O device. The machine cycle consists of 3 to 6 T-states.

- **3. T state**

  The time corresponding to **one clock period**. The T state forms the basic unit to calculate the execution of instructions and programs in a processor.

Different Machine Cycles

- The 8085 microprocessor has 5 (Five) **basic** machine cycles. They are

a) **Opcode fetch** cycle (4T)

b) **Memory read** cycle (3 T)

c) **Memory write** cycle (3 T)

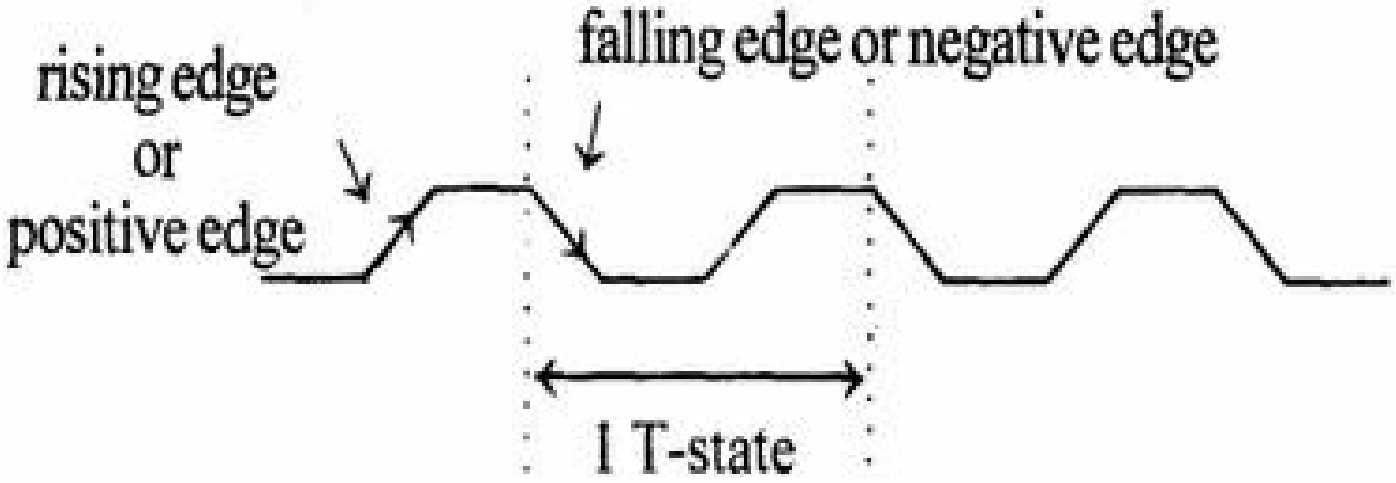
d) **I/O read cycle** (3 T)

e) **I/O write cycle** (3 T)

Machine Cycles - Description

- Each instruction of the 8085 processor consists of one to five machine cycles.
- An **instruction execution will have compulsorily, the opcode fetch cycle**.
- Then depending upon the instruction, an instruction cycle will have one or two or more other cycles in the list.
- When the 8085 processor executes an instruction, it will need to access external memory and I/O devices, in other words need many machine cycles in a specific order.

Clock Signal Shape



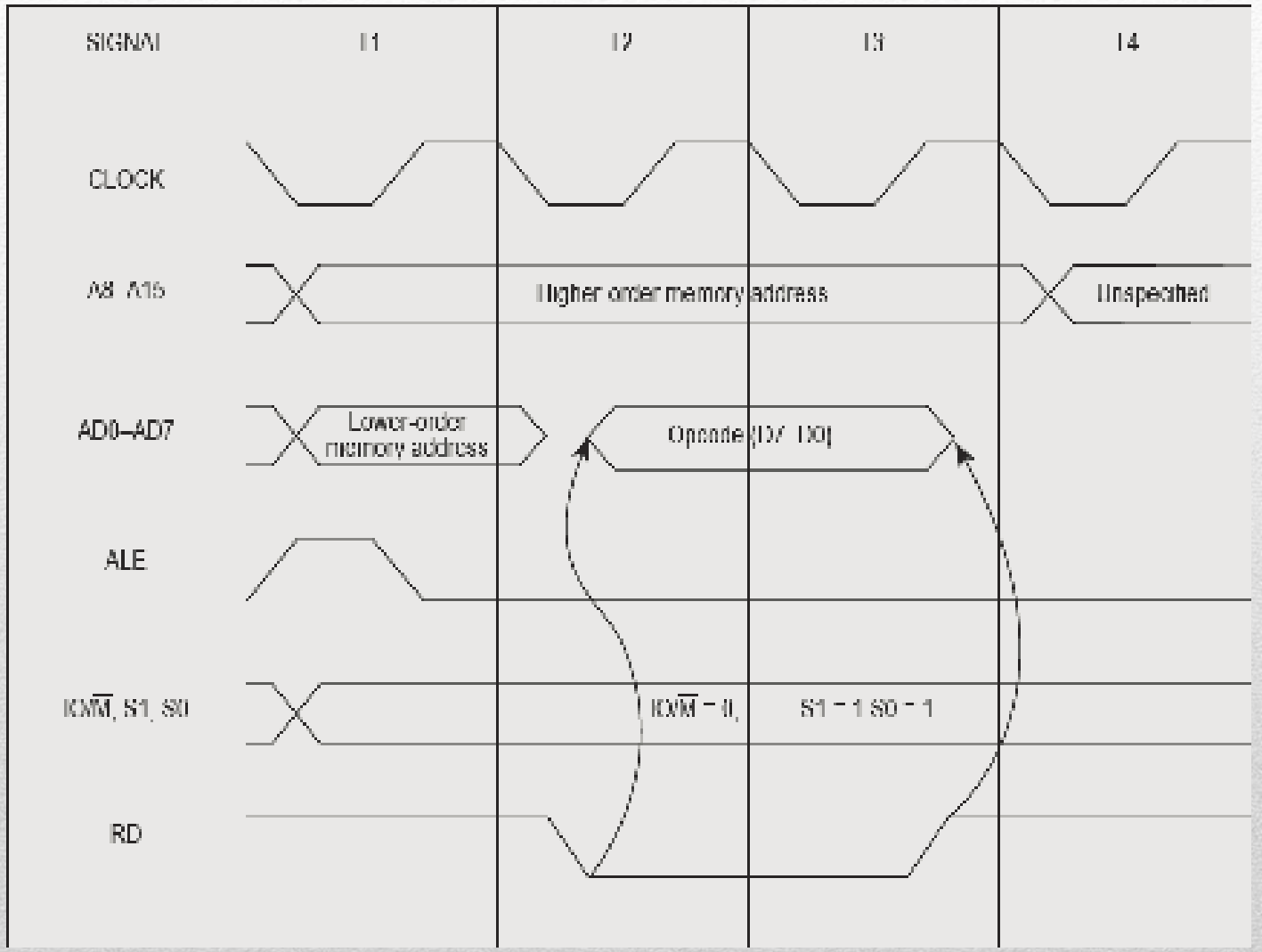
Time period, T = 1/f ; where f = Internal clock frequency

Timing Diagram

- The **timing diagram** of an instruction are obtained by drawing the **binary levels** on the various **signals** of 8085.
- It is drawn with respect to the **clock** of the microprocessor.
- It explains the **execution** of the instruction with the basic machine cycles of that instruction, one by one in the order of execution.

- *Representation of Various Control signals generated during Execution of an Instruction.*
- *Following **Buses** and **Control Signals** must be shown in a Timing Diagram:*
- ***Clock***
- **Higher Order Address Bus.**
- **Lower Address/Data bus**
- **ALE**
- **RD~**
- **WR~**
- **IO/M~**
- **S0,S1**

Opcode fetch machine cycle



Opcode fetch machine cycle

- At **T1**, the high order 8 address bits are placed on the address lines A8 – A15 and the low order bits are placed on AD7–AD0.
- The **ALE** signal goes high to indicate that AD0 – AD7 are carrying an address and now that needs to be **latched**.
- At exactly the same time, the **IO/M~** signal goes low to indicate a memory access operation.

Opcode fetch machine cycle

- At the beginning of the **T2** cycle, the low order 8 address bits are removed from AD7– AD0 and **latched** and the controller sends the Read **(RD~)** signal to the memory.
- The **RD~** signal remains low (active) for two clock periods to allow for slow devices.
- During **T2**, memory places the data (i.e. **opcode**) from the memory location on the lines AD7– AD0.

Opcode fetch machine cycle

- During **T3** the **RD~** signal is Disabled (goes high).
- That makes the AD7– AD0 lines go to high impedance (or tri- stated) mode.
- The opcode is placed into **instruction register**, and then sent to **decoder**.
- During **T4**, the opcode is **decoded** by the processor and necessary action or control is initiated for the **execution** of the instruction fetched.

MPU Communication and Bus Timing

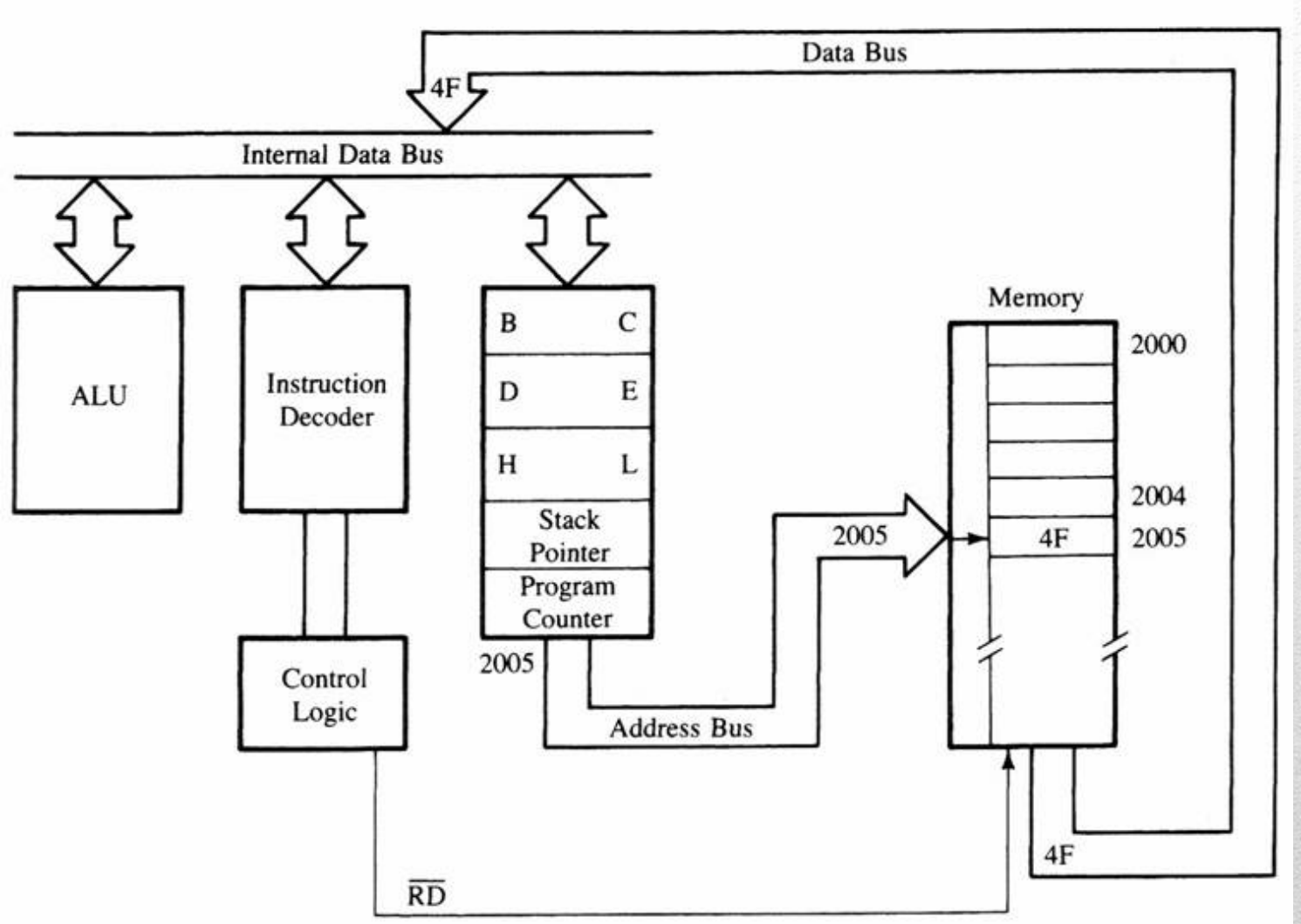


Figure :Execution of **MOV C, A** instruction form memory to MPU (machine code is 4FH = 0100 1111)

MPU Communication and Bus Timing

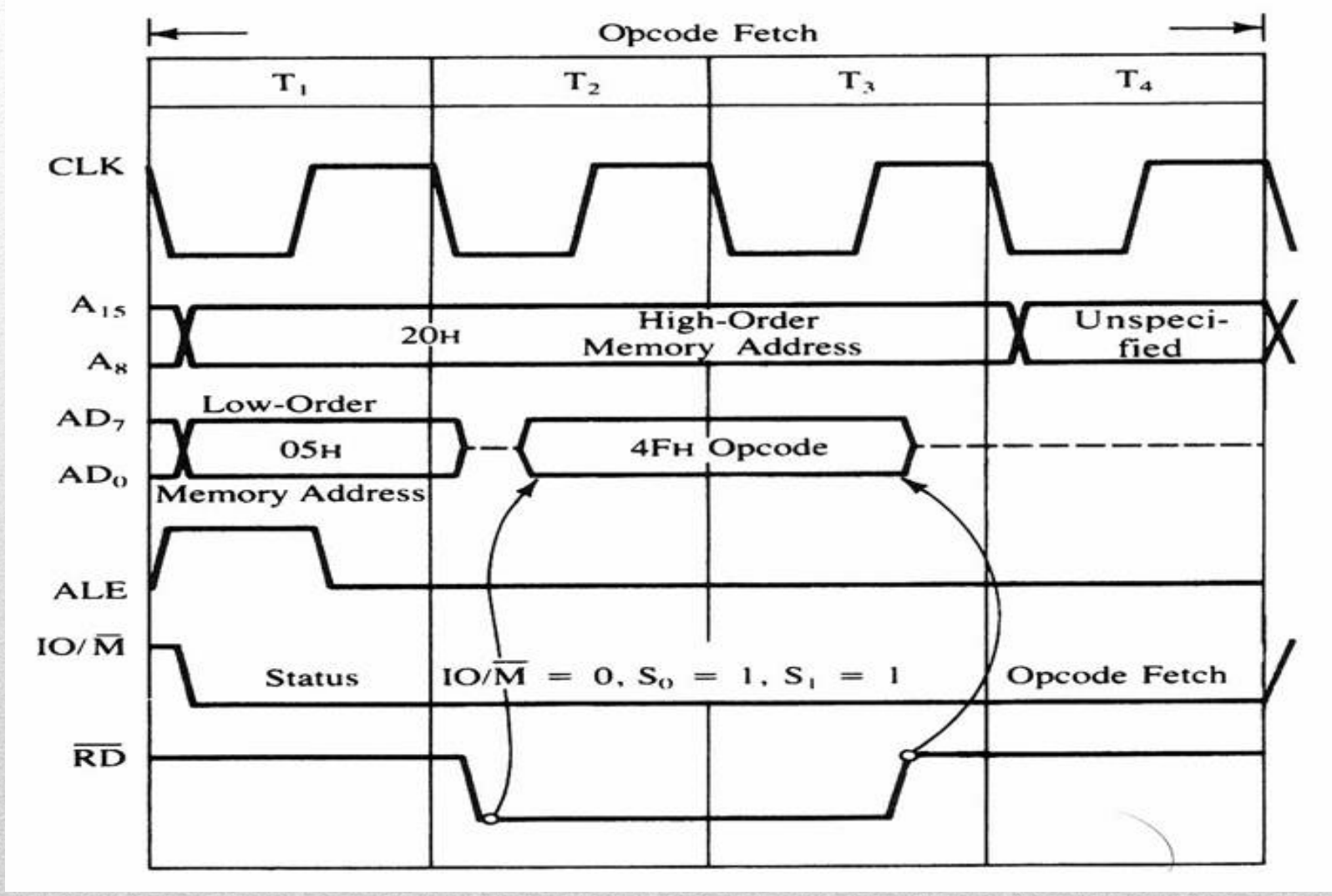
- The **Fetch Execute** Sequence :
  1. The µp places a 16 bit memory address from PC (program counter) to address bus.
     - Figure : at **T1**
       - The high order address, 20H, is placed at A15 – A8.
       - the low order address, 05H, is placed at AD7 - AD0 and ALE is active high.
       - Simultaneously the IO/M**~** is in active low condition to show it is a memory operation.
  2. At **T2** the active low control signal, RD**~**, is activated so as to activate **read** operation; At the same time **S0** and **S1** become 1 and 1 respectively, it is to indicate that the MPU is in **fetch** mode operation.

Continue…

3. **T3**: The active low **RD~** signal enabled the byte instruction, **4FH**, to be **placed** on AD7 – AD0 and transferred to the **MPU**. While **RD~** high, the data bus will be in high impedance mode. **4FH** will then be placed in the **instruction register** and then sent to **decoder** circuit.

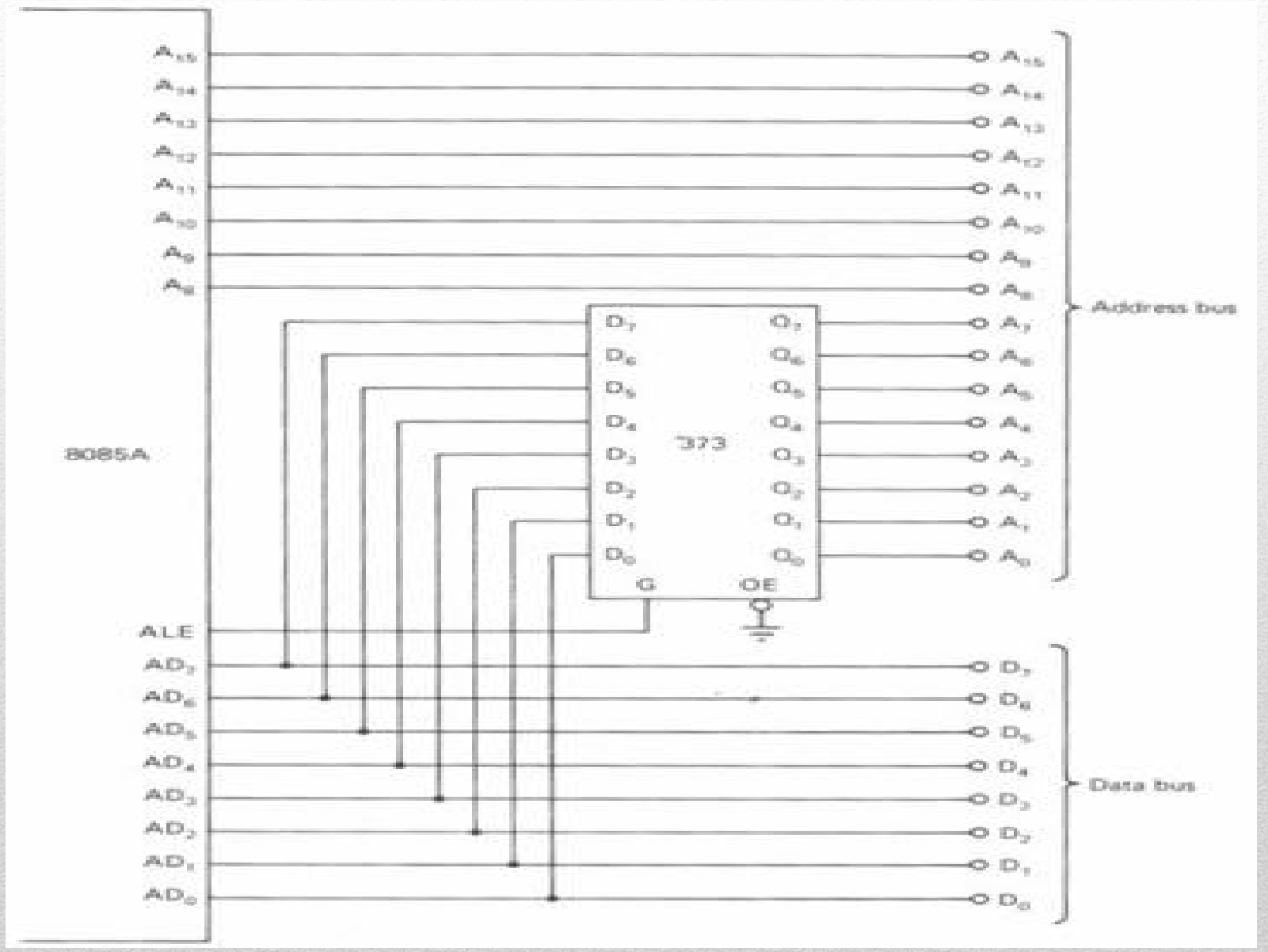
4. **T4**: The machine code, 4FH, will then be **decoded** in instruction **decoder**. **The content of accumulator (A) will then copied into C register** at state T4.

8085 timing diagram for Opcode fetch cycle for MOV C, A .

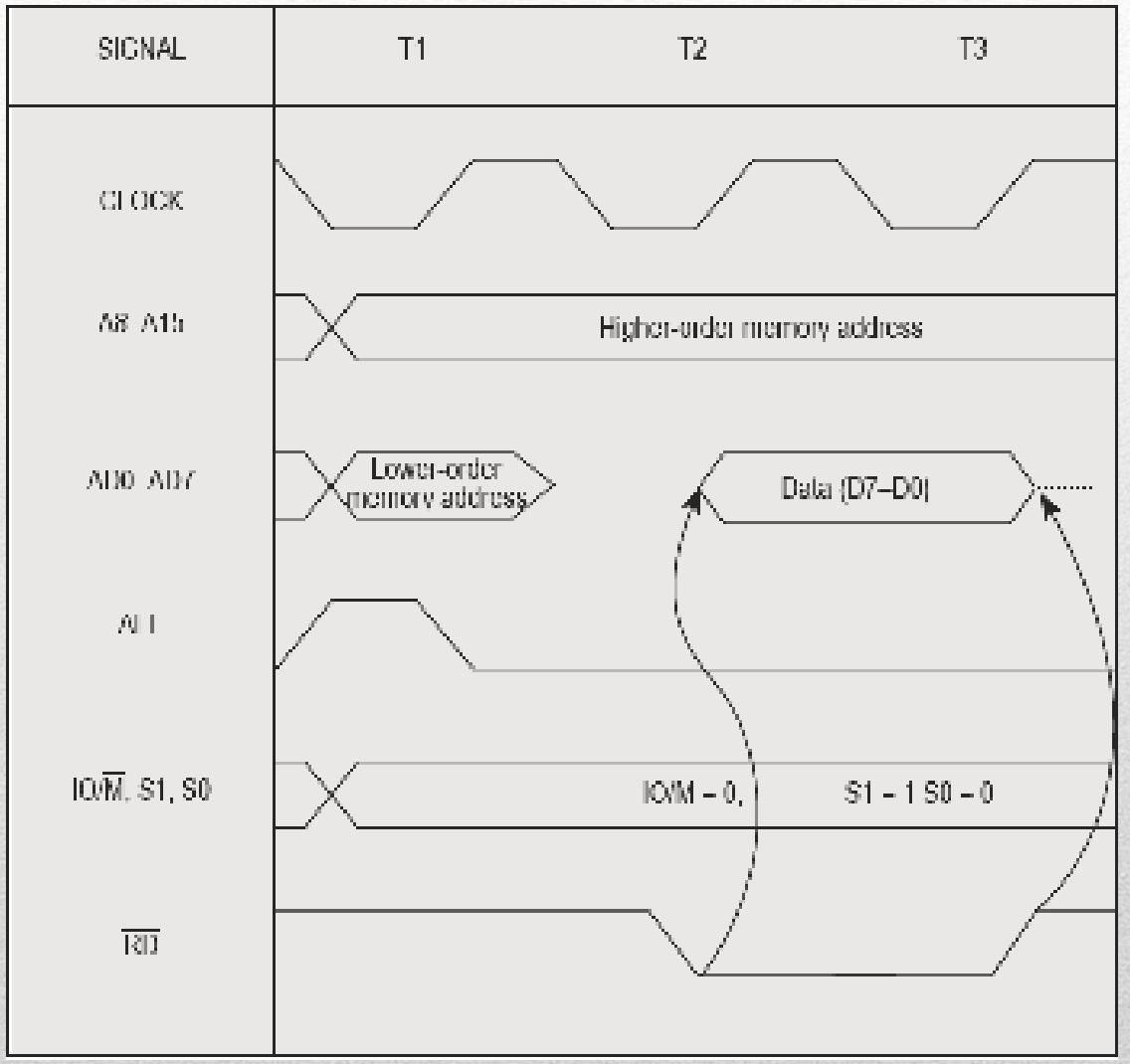


8085 timing diagram for Opcode fetch cycle for MOV C, A .

ALE used to demultiplex address/data bus



Memory Read Machine Cycle



Memory Read Machine Cycle

- The memory read machine cycle is exactly the same as the opcode fetch except:

a) It has only 3 T-states

b) The **S0** signal is set to **0**.

c) The memory read machine cycle is executed by the processor to **read** a **data** byte from memory.

d) The processor takes 3T states to execute this cycle.

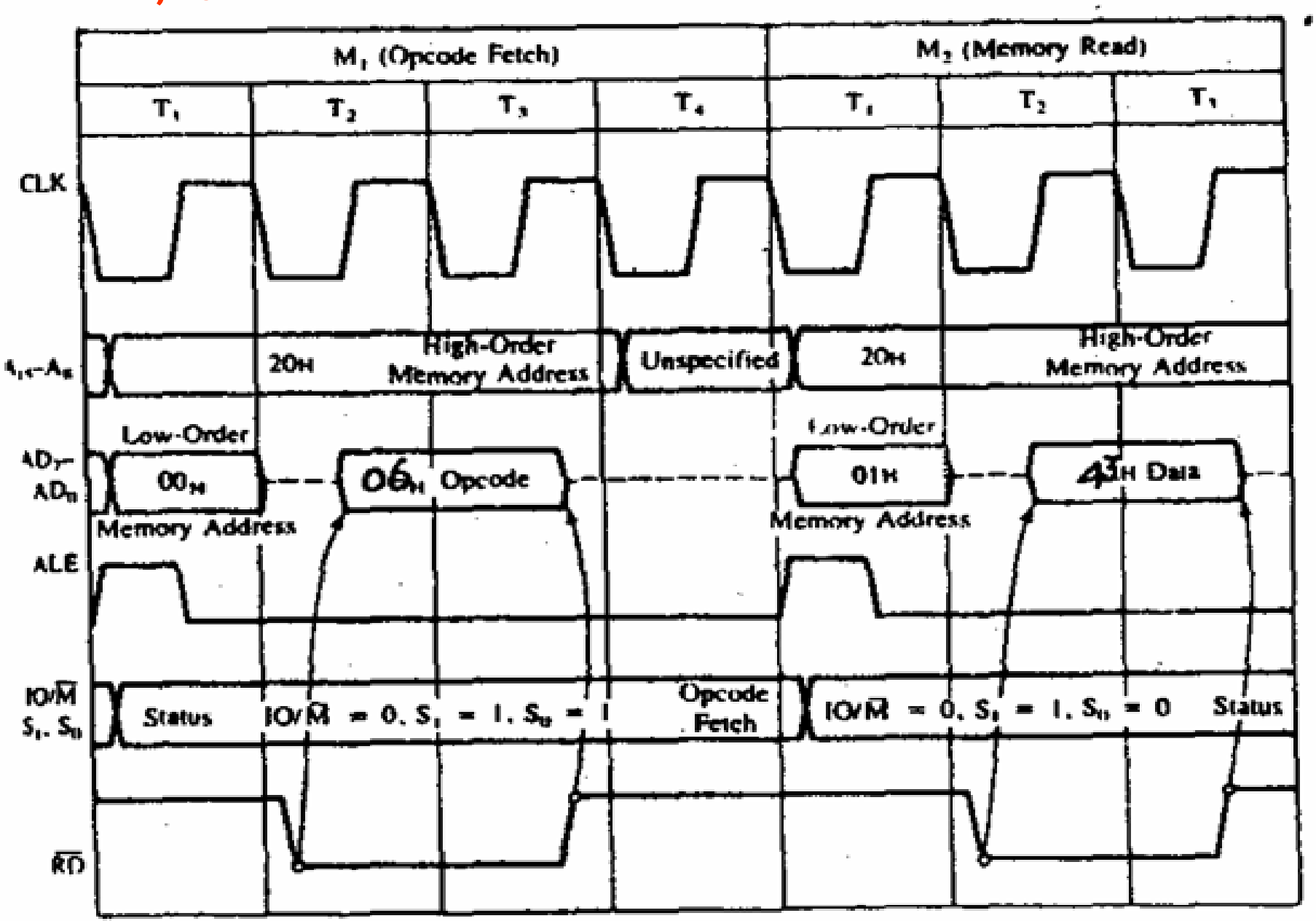
e) The instructions which have more than one byte word size will use the machine cycle after the opcode fetch machine cycle.

Execution of MVI B,43

- Instruction:
- **2000H MVI B, 43H**

- Corresponding Coding:
- (Machine code is 06, and data is 43)
- Memory Location M/C code
- **2000H 06H**
- **2001H 43H**

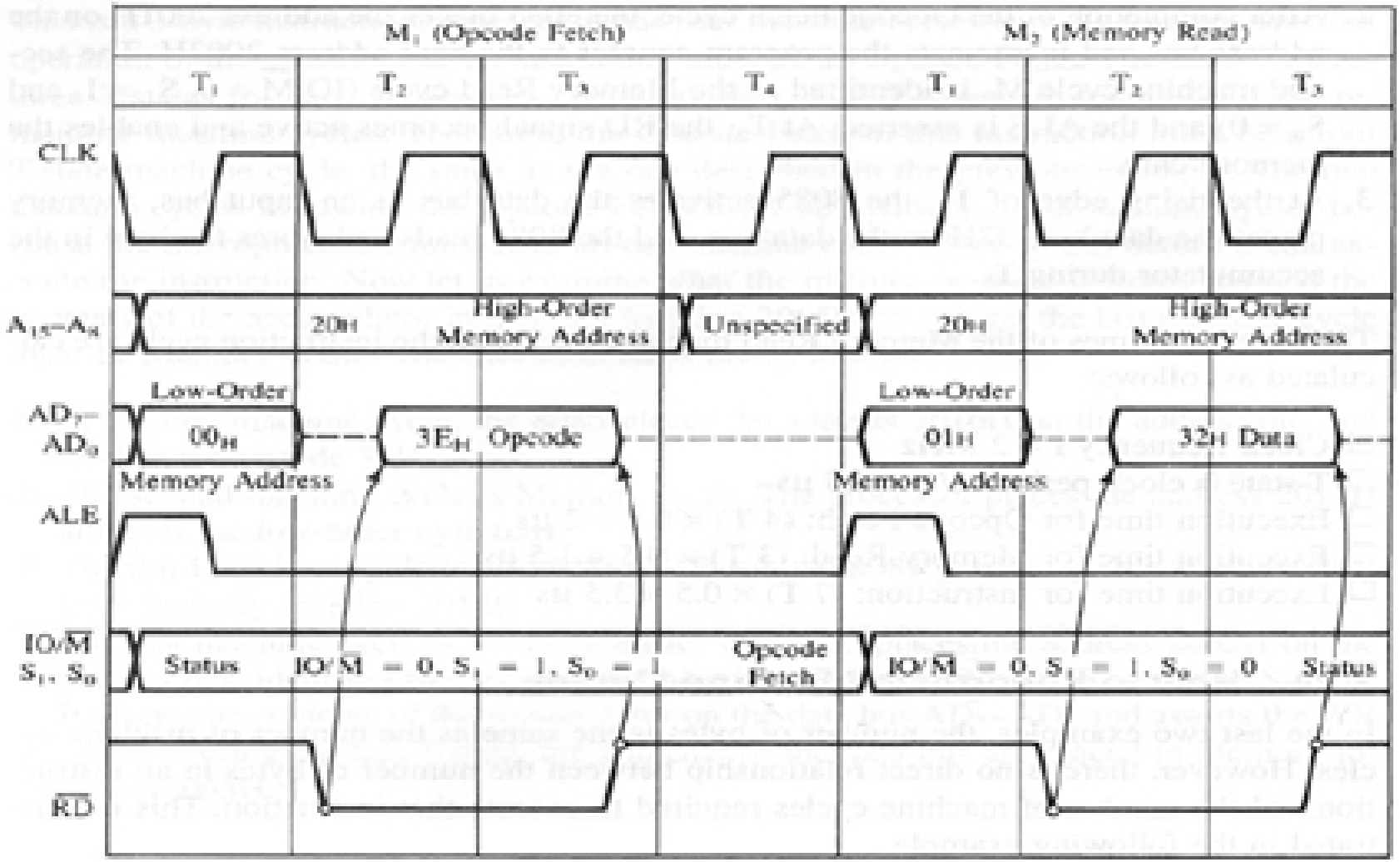
MVI B, 43

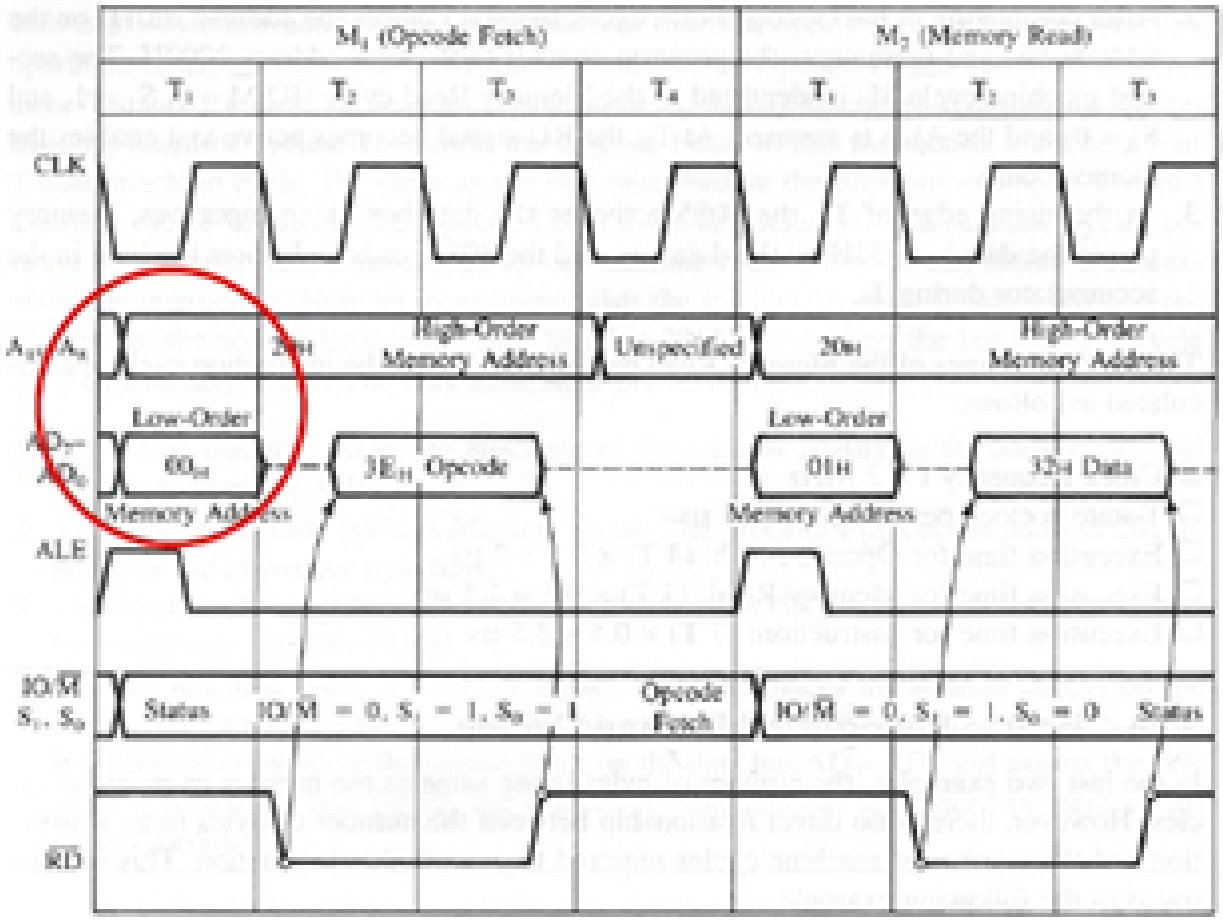


Example :- MVI A, DATA

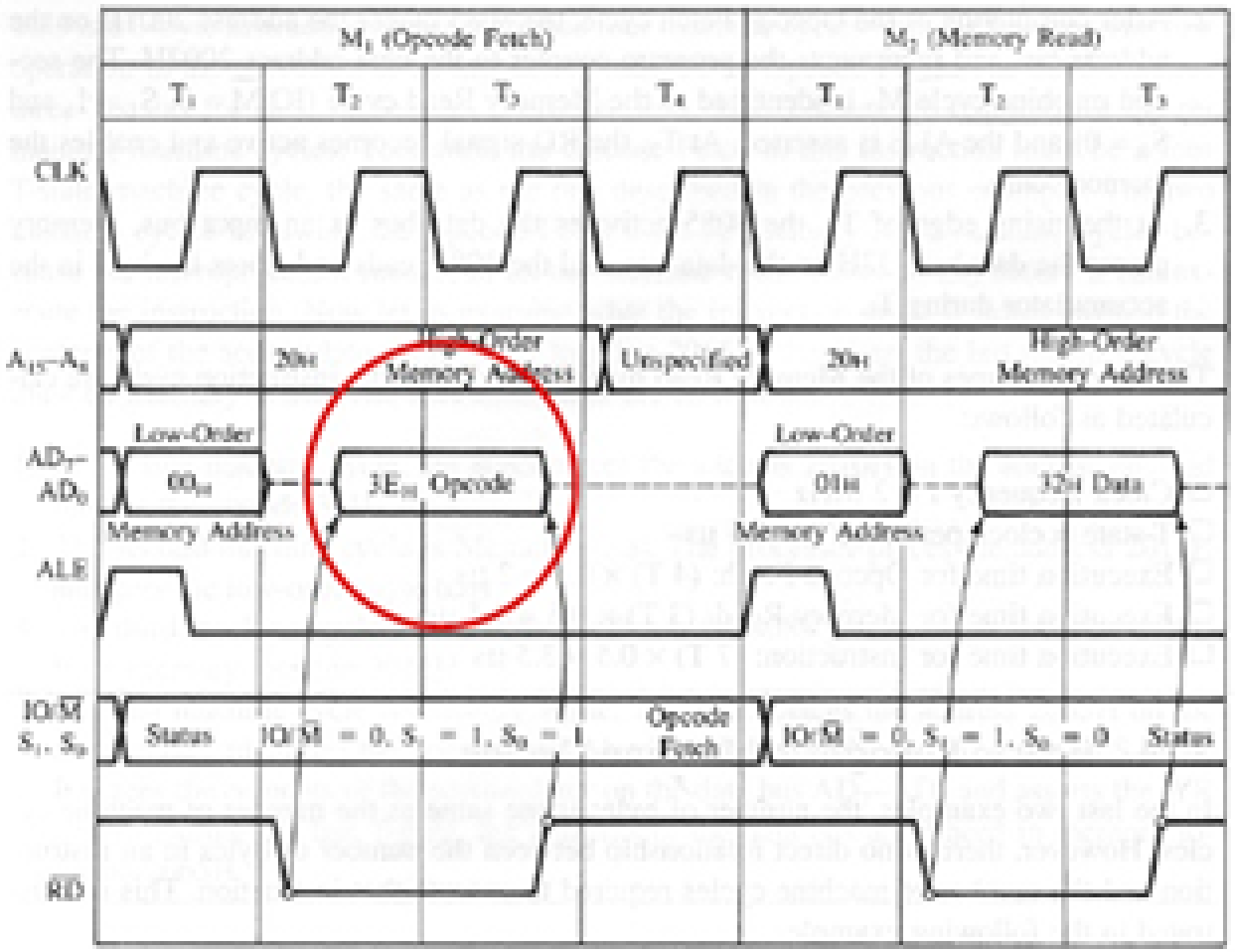
ADDRESS	MNEMONICS	OP-CODE
2000	MVI A,32_H	3E
2001		32

TIMING DIAGRAM OF MVI A,32 H

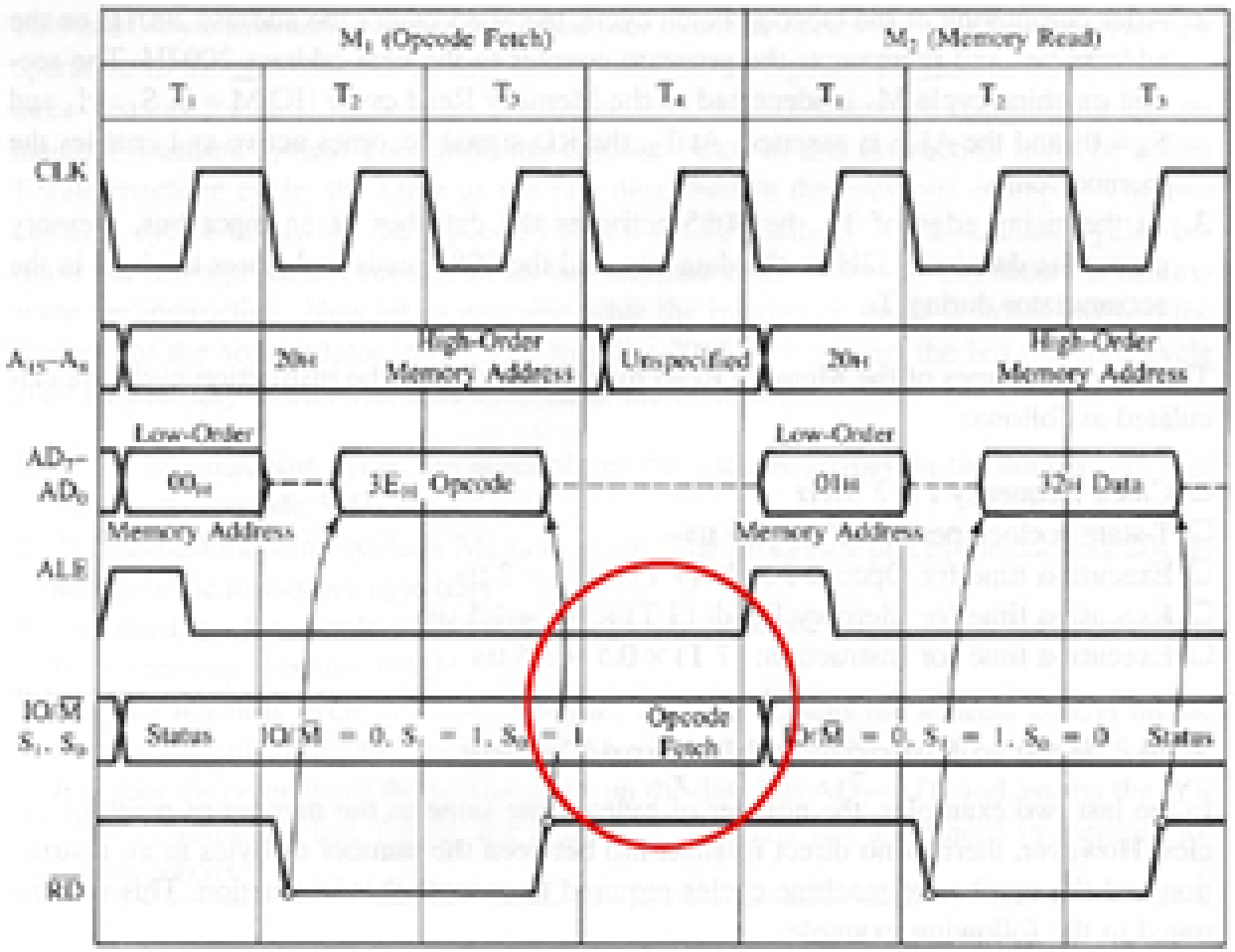




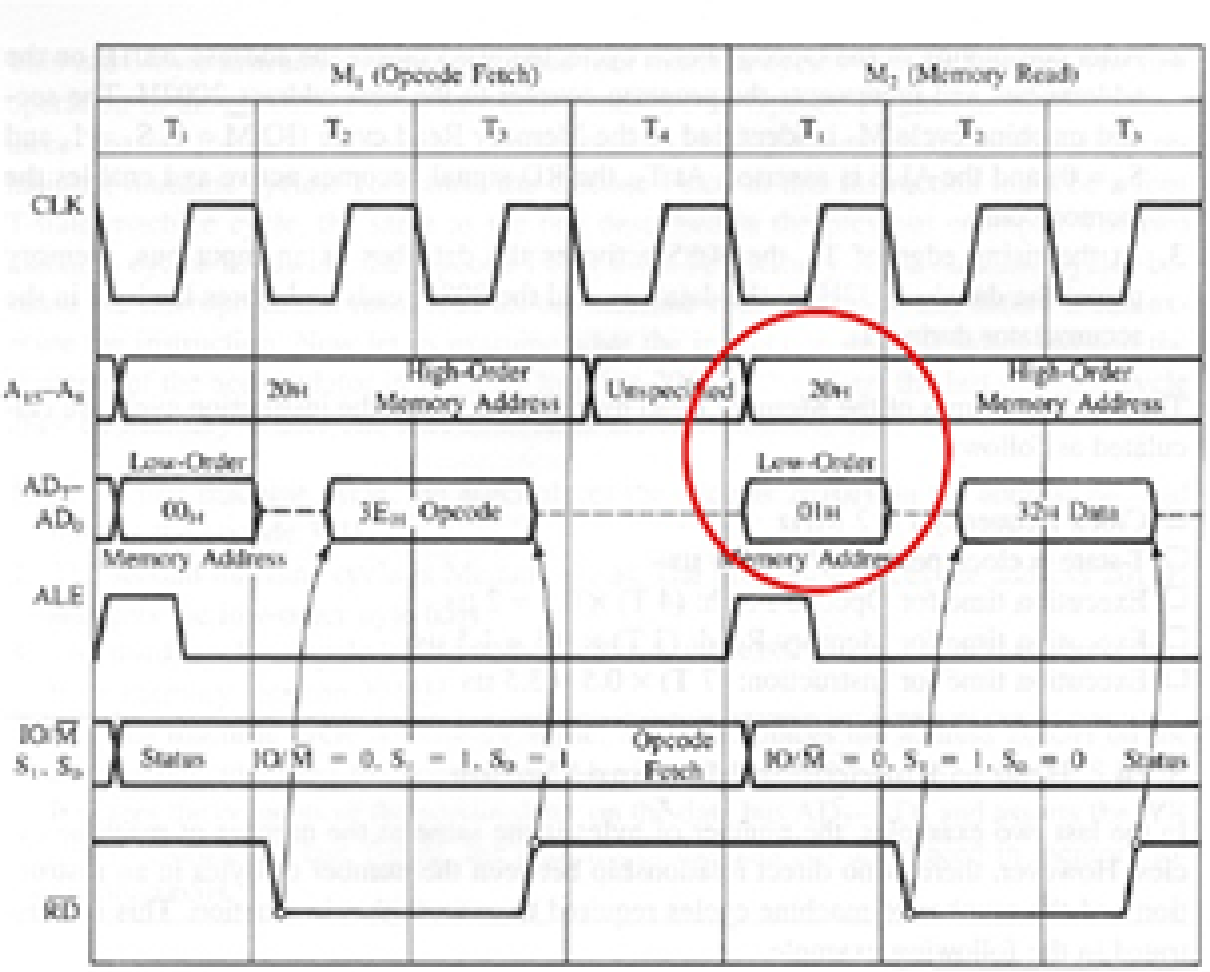
Put the first memory Location on the address bus(2000 H)



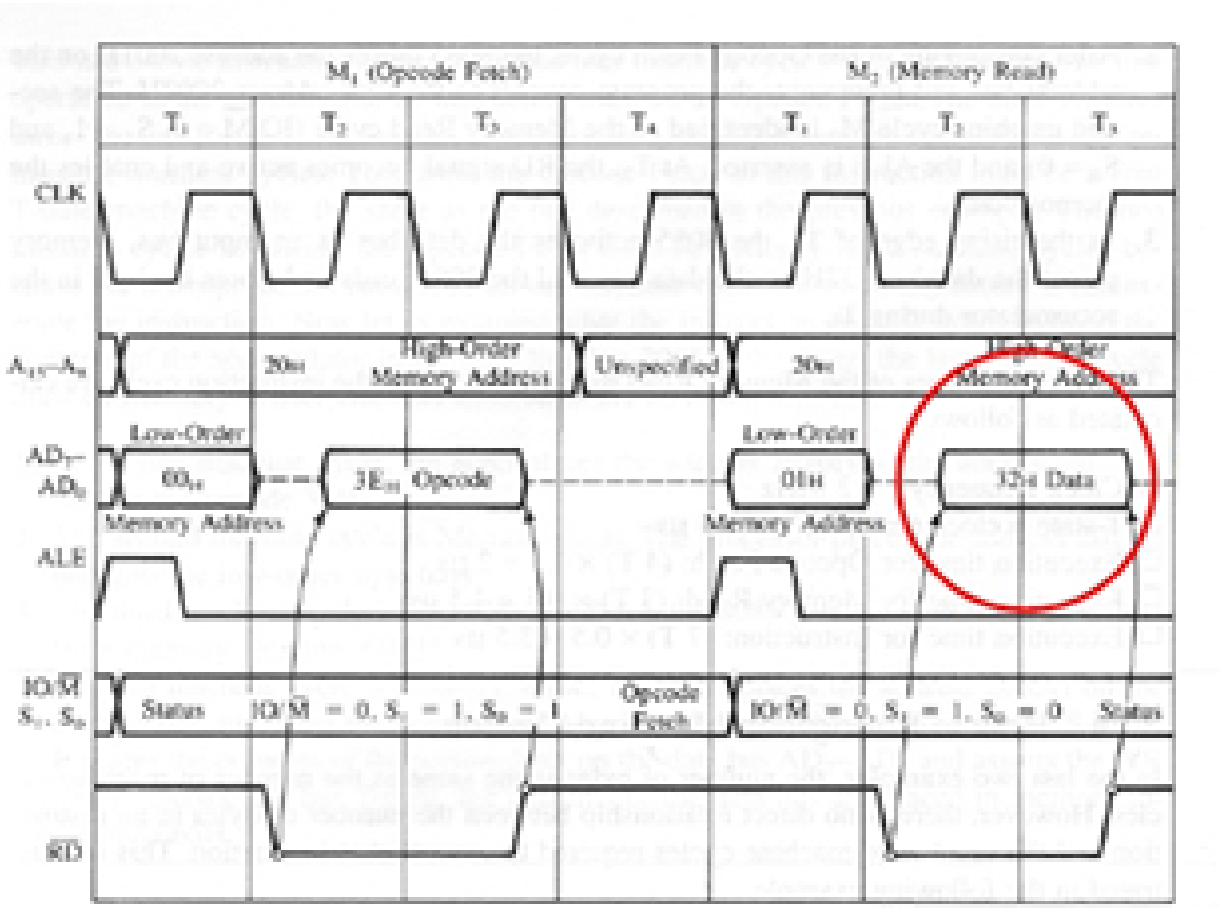
Get the instruction(op-code) byte from memory



Interpret the Instruction : Wait for the data byte



Put the next memory location on the address bus (2001 H)



Get the data byte from the memory

Through data lines
This will be finally placed into accumulator

Timing diagram for STA 526AH.

Assume that **Accumulator** has the data **C7**H.

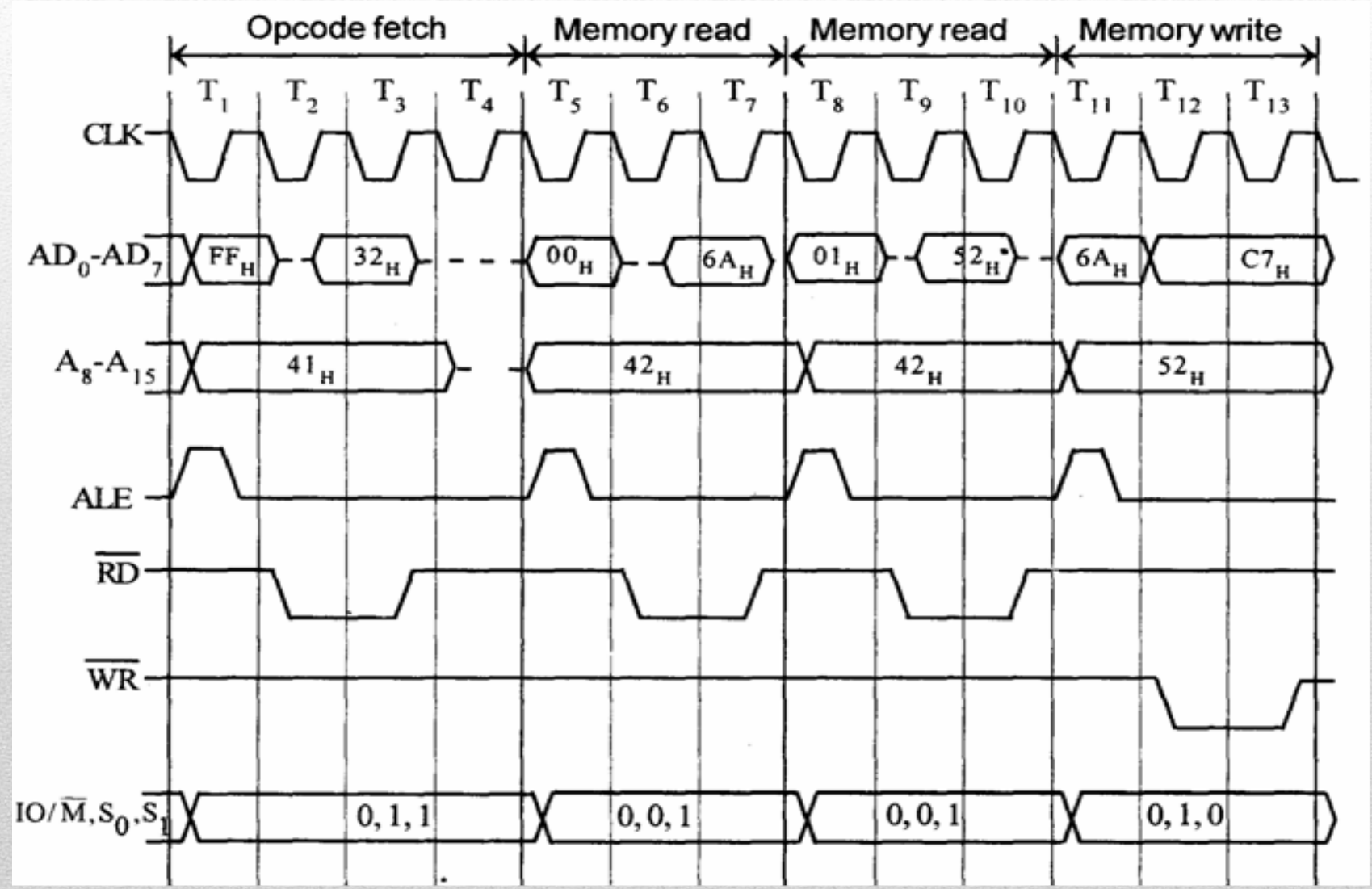
- This instruction is going to store content of A (i.e. C7) in the memory location **526A**.
- The opcode of the STA instruction is said to be 32H

Address	mnemonic	opcode
41FFH	**STA 526A**	32H
4200H		6AH
4201H		52H

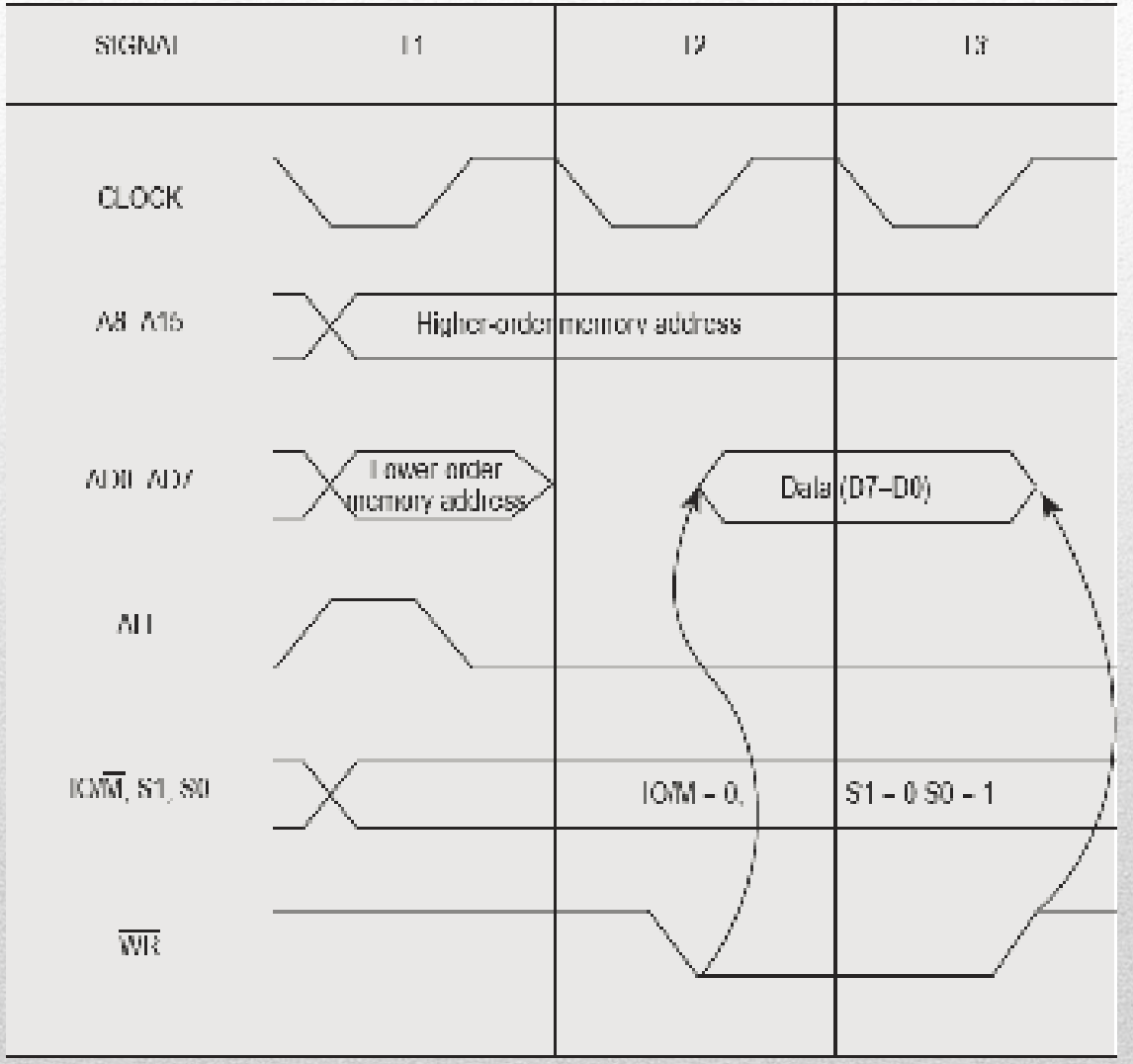
Timing diagram for STA 526AH.

- STA means **Store Accumulator** -The contents of the accumulator is stored in the specified address(**526A**H).
- The opcode of the STA instruction is said to be **32**H. It is fetched from the memory **41FFH**.
- Then **read** the lower order memory address which is (**6A**H). - *Memory Read Machine Cycle,*
- Read the higher order memory address (**52**H).- *Memory Read Machine Cycle,*
- The combination of both the addresses are considered and the **content** from accumulator is written in **526AH**. - *Memory Write Machine Cycle,*
- This will be the memory address for the instruction and let the content of accumulator is C7H. So, C7H from accumulator is now stored in 526A.

STA 526A



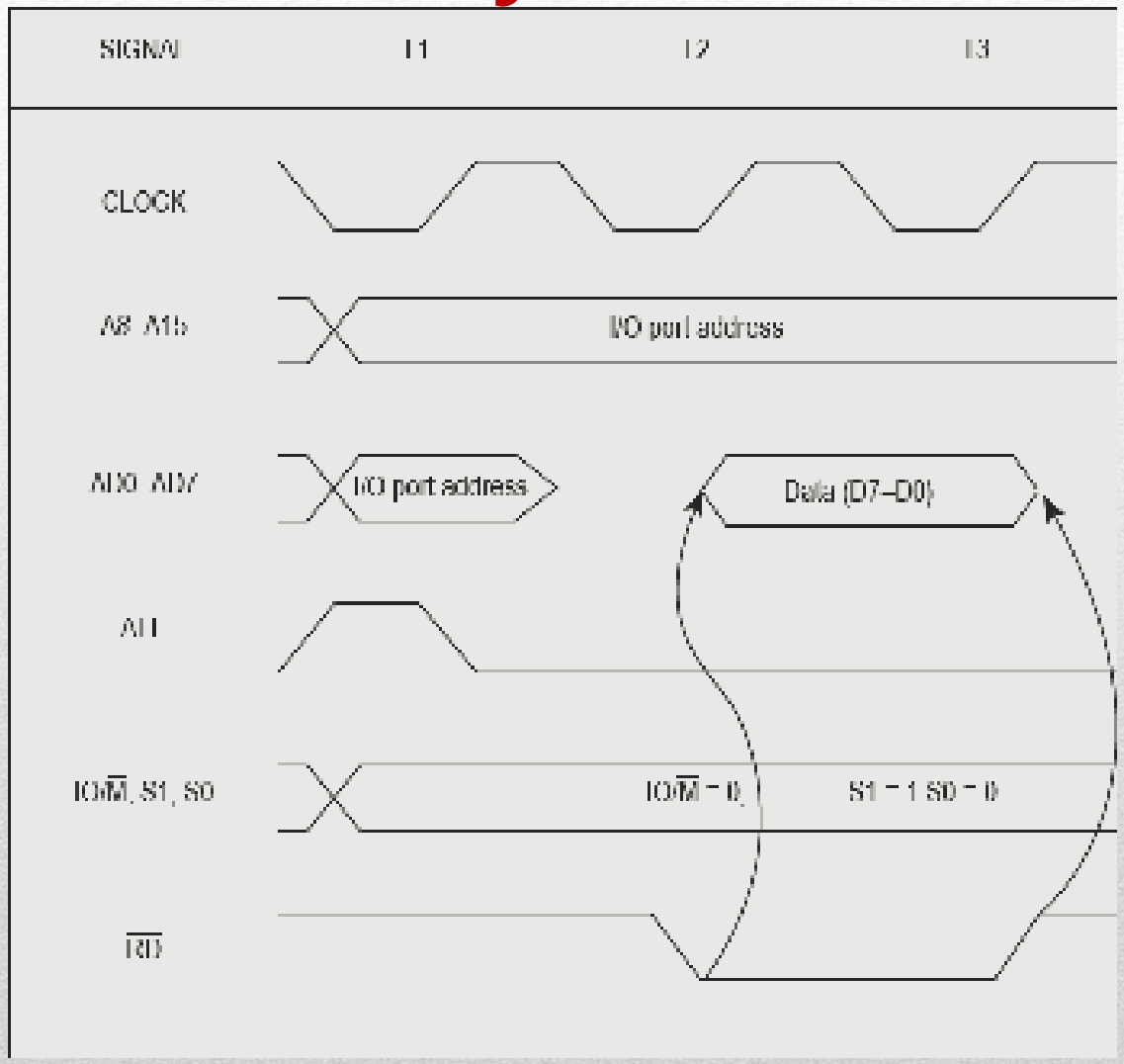
Memory Write Machine Cycle



Memory Write Machine Cycle

- The memory write machine cycle is executed by the processor to write a data byte in a memory location.
- The processor takes **3T states** to execute this machine cycle.
- Now, the active low **WR~** signal is made to low indicating a write operation to the memory chips.

I/O Read Cycle of 8085



I/O Read Cycle

- The I/O Read cycle is executed by the processor to read a data byte from **I/O port** or from the peripheral, which is I/O, mapped in the system.
- 8085 uses 8-bit **port address**. So, the port address is placed in **lower order** address bus.
- At the same time, the port address is also placed in the **higher order** address bus.
- The **IN** instruction uses this machine cycle during the execution.

Timing diagram for **IN** instruction

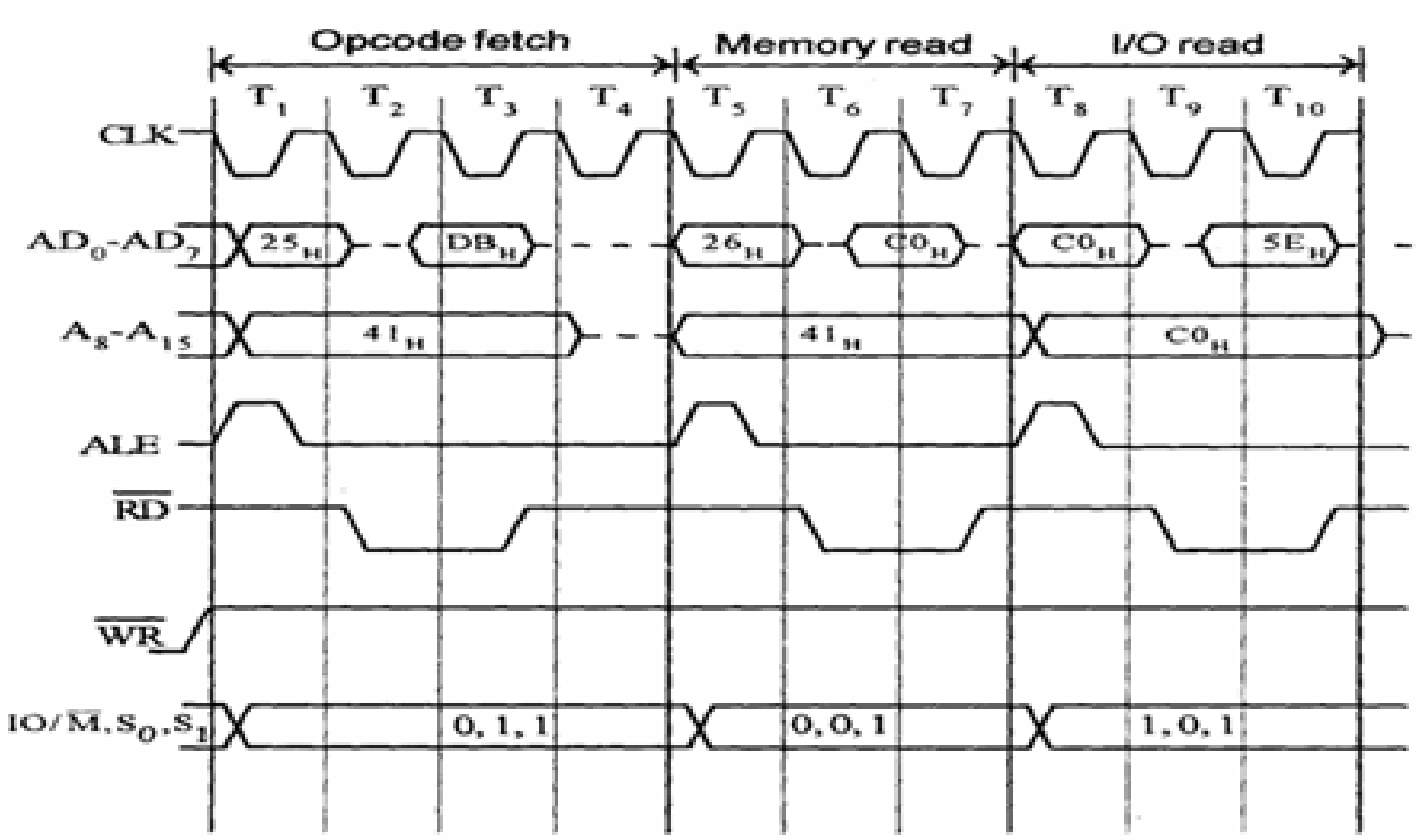
TIMING DIAGRAM FOR IN port address.

1. Fetching the Op-code DBH from the memory 4125H.
2. Read the port address C0H from 4126H.
3. Read the content of port C0H and send it to the accumulator.
4. Let the content of port is 5EH.

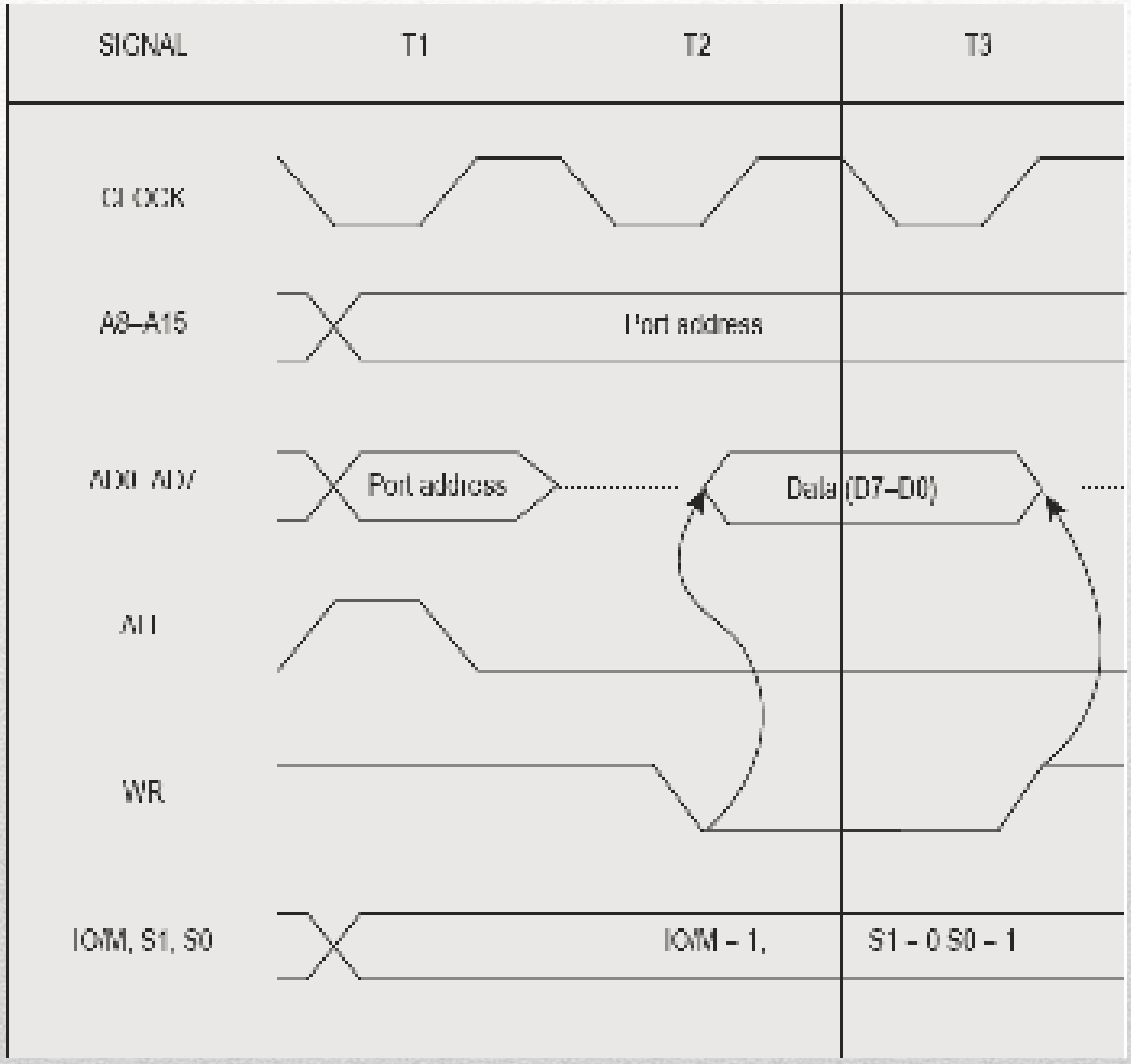
ADDRESS	MNEMONICS	OP-CODE
4125	IN $C0_H$	DB_H
4126		$C0_H$

Timing diagram for **IN** instruction

IN C0H



I/O Write Cycle of 8085



Introduction - I/O Write Cycle

- The I/O write machine cycle is executed by the processor to **write** a data byte in the **I/O port** or to a peripheral, which is I/O, mapped in the system.
- The processor takes 3T states to execute this machine cycle.

8086

- **Paging** (physical division of program and memory)
  -Pages and page frames
  20 address lines
  16 data lines
- **Segmentation** (Code Segment (IP), Data Segment (BX/SI/DI), Stack Segment (SP), Extra segment)
- Fragmentation (wastage of memory)
- Swapping
- Thrashing

Counters

- This is also called as software counter or timer.
- Counters and time delays are important techniques.
- They are commonly used in applications such as **traffic signals, digital clocks, process control** and serial data transfer.

Counters…

- counter is used to count certain value e.g. binary counter, decimal counter etc.
  - A loop is set up by loading a register with a certain value
  - Then using the DCR (to decrement) and/or 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.

Counter…

- The following is an example of a counter to display the number at the output port (from FF to 00):

```
        MVI  C, FFH    initialize the counter
LOOP    MOV A,C          move it to A
        OUT 01H          display it at output port (01)
        DCR C          decrement the number
      JMP   LOOP     go back to given location
```

Timing concepts

- Each instruction passes through different combinations of Fetch, Memory Read, and Memory Write cycles etc.

- Knowing the combinations of cycles, one can calculate **how long an instruction would require** to complete.
- Intern, one can calculate **how long (how much time) a processor will take, to execute a complete program.**
- **An instruction have one or more bytes (i.e. size).**
- **An instruction have one or more** Machine Cycles.
- **An instruction have one or more T-states.**
  - B for Number of Bytes
  - M for Number of Machine Cycles
  - T for Number of T-State.

Delays…

- 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, using the following formula:
- T = 1/f, (one clock period)

  Delay = No. of T-States * T

  or Delay = No. of T-States / Frequency

- For example "MVI" instruction uses 7 T-States. Therefore, if the Microprocessor is running at 2 MHz(2*10^6 Hz), the instruction would require 7*1/2 *10^6 = 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

- We need to keep in mind 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

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

Delay Loops

- Using these formulas, we can calculate the time delay for the previous example:
- T_0 = 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} = (7 + 3567) X 0.5 μSec = 1.787 mSec**
  - **Assuming f = 2 MHz**
  - **Next question is 'How to increase this delay?'**

Using a Register Pair as a Loop Counter

- Using a single register, one can repeat a loop for a maximum count of 255 times (FF is max number).
- It is possible to further 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 as a Loop Counter

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

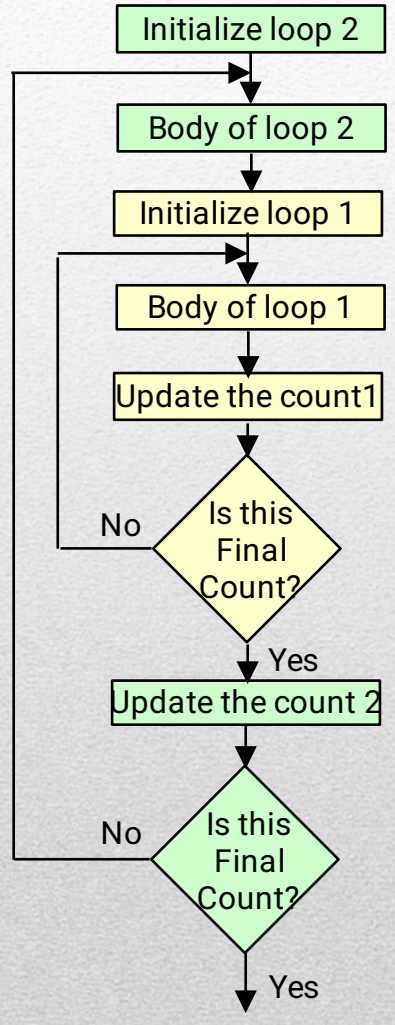
```
      LXI B, 1000H 10 T-States
LOOP  DCX B  6 T-States
      MOV A, C   4 T-States
      ORA B  4 T-States
      JNZ LOOP  10 T-States
```

Using a Register Pair as a Loop Counter

- Using the same formula as earlier, we can calculate:
- T_0 = 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 **µSec** = 49.155 mSec

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 is before and after loop1.



Nested Loops for Delay

- Instead 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_{01} = 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

Delay Calculation of Nested Loops

- Delay of outer loop
  - T_{O2} = 7 T-States
    - MVI B, 10H instruction
  - T_{L2} = (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
  - Assuming f = 2MHz

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.

Problem statement

- Write a program to count continuously from FF to 00 in a system with a clock frequency of 2 MHz. Provide 1m sec delay between each count and display the number at output port.

- First, Only the counter program

Label	Instruction	T-state	Comments
	MVI B,00H	7	no. to be displayed
GO:	DCR B	4	
	MOV A,B	4	
	OUT 01	10	display no. at o/p port 01
	JMP GO	10	

- Delay program
-

	MVI C, COUNT	7	counter for delay
DELAY:	**DCR C**	**4**	
	JNZ DELAY	**10**	

Label	Instruction	T-state	Comments
	MVI B,00H	7	no. to be displayed
GO:	DCR B	4	
	MVI C, COUNT	7	counter for delay
DELAY:	**DCR C**	**4**	
	JNZ DELAY	**10**	
	MOV A,B	4	
	OUT 01	10	display no. at o/p port 01
	JMP GO	10	

Delay Calculation

- T (Loop)= (10+4)* T*Count
- = 14* 0.5*10^-6 *Count = 7*10^-6* Count
- T (outside)= (4+7+4+10+10)* T = 35*0.5*10^-6 = 17.5 μSec.
- Total time delay = T (Loop) + T (outside)
- 1 ms = 17.5*10^-6 + 7*10^-6* Count

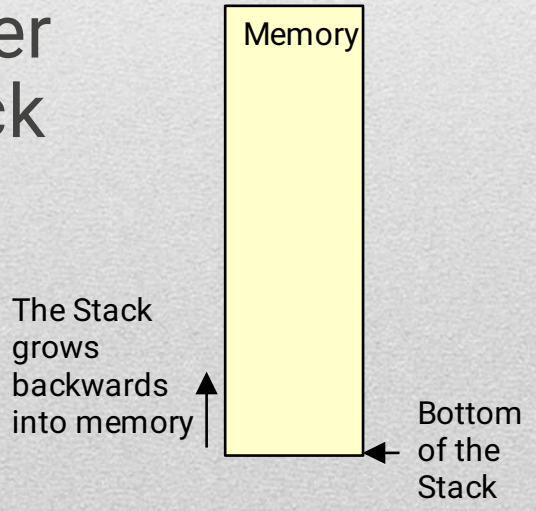
- 1*10^-3 = 17.5*10^-6 + 7*10^-6* Count

- (1*10^-3 - 17.5*10^-6) / (7*10^-6) = Count

- => Count = 140 (decimal)
- = 8CH

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

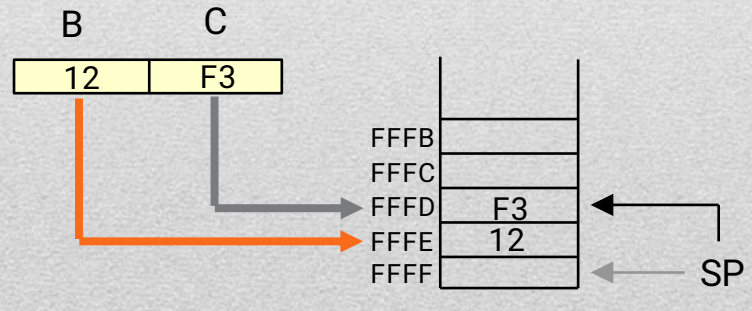
- This sets the Stack Pointer to location FFFFH (end of memory for the 8085).

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.

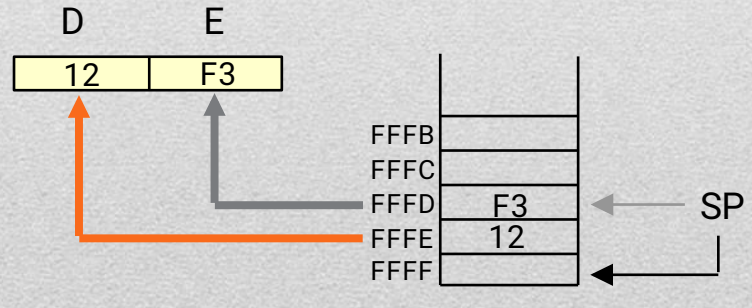
The PUSH Instruction

- PUSH B
  - Decrement SP
  - Copy the contents of register B to the memory location pointed to by SP
  - Decrement SP
  - Copy the contents of register C 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 E
  - Increment SP
  - 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
```

- Reversing the order of the POP instructions will result in the exchange of the contents of BC and DE.

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

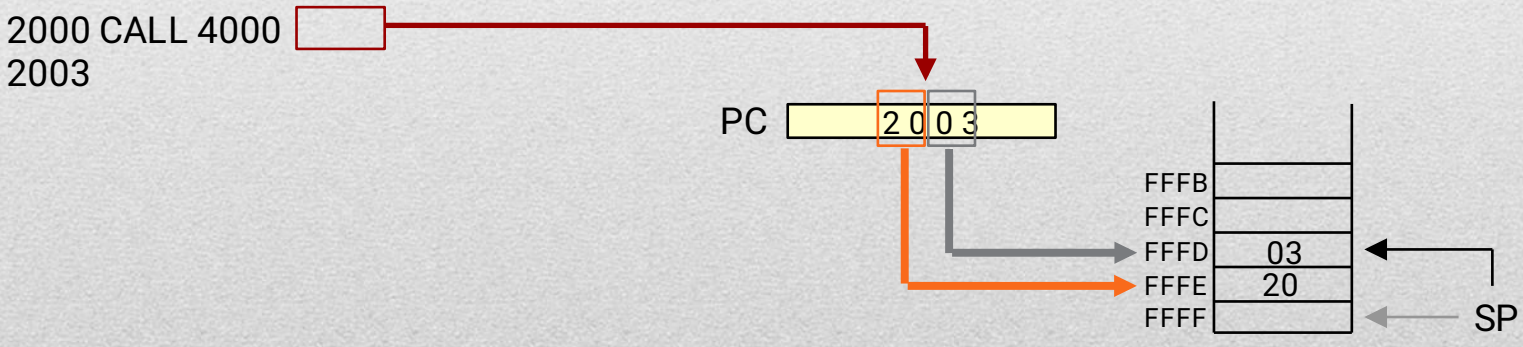
- A subroutine is a group of instructions that will be used repeatedly in different locations of the program.
  - Rather than repeat the same instructions several times, they can be grouped into a subroutine that is called from the different locations.
- In Assembly language, a subroutine can exist anywhere 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 the program counter with the 16-bit address supplied with the CALL instruction.

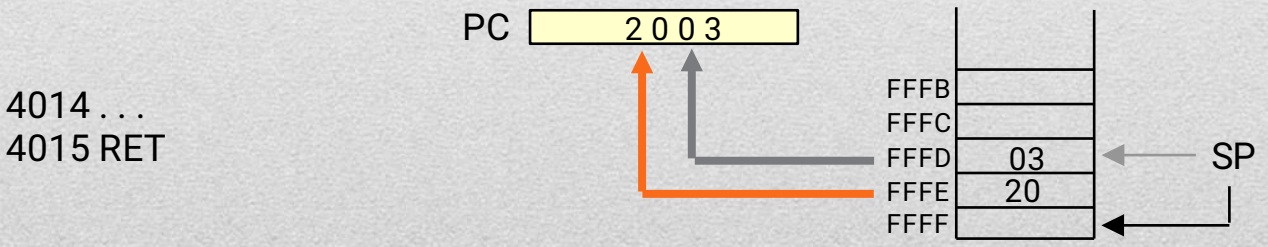


The CALL Instruction

- MP Reads the subroutine address from the next two memory locations and stores the higher order 8bits of the address in the W register and stores the lower order 8bits of the address in the Z register,
- – Pushes the address of the instruction immediately following the CALL onto the stack [Return address],
- – Loads the program counter with the 16-bit address supplied with the CALL instruction from WZ register.

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 RTE 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.

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.