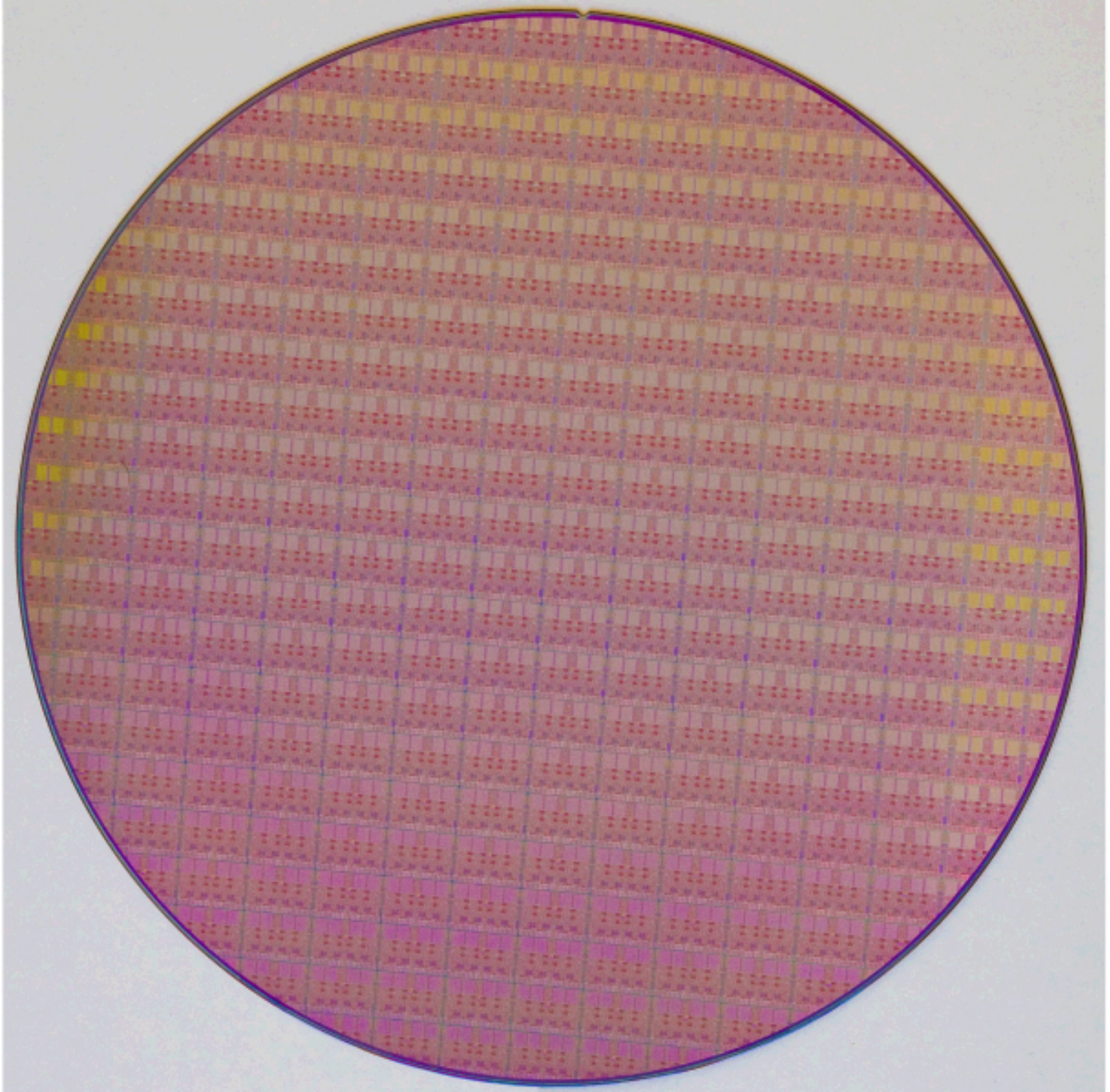
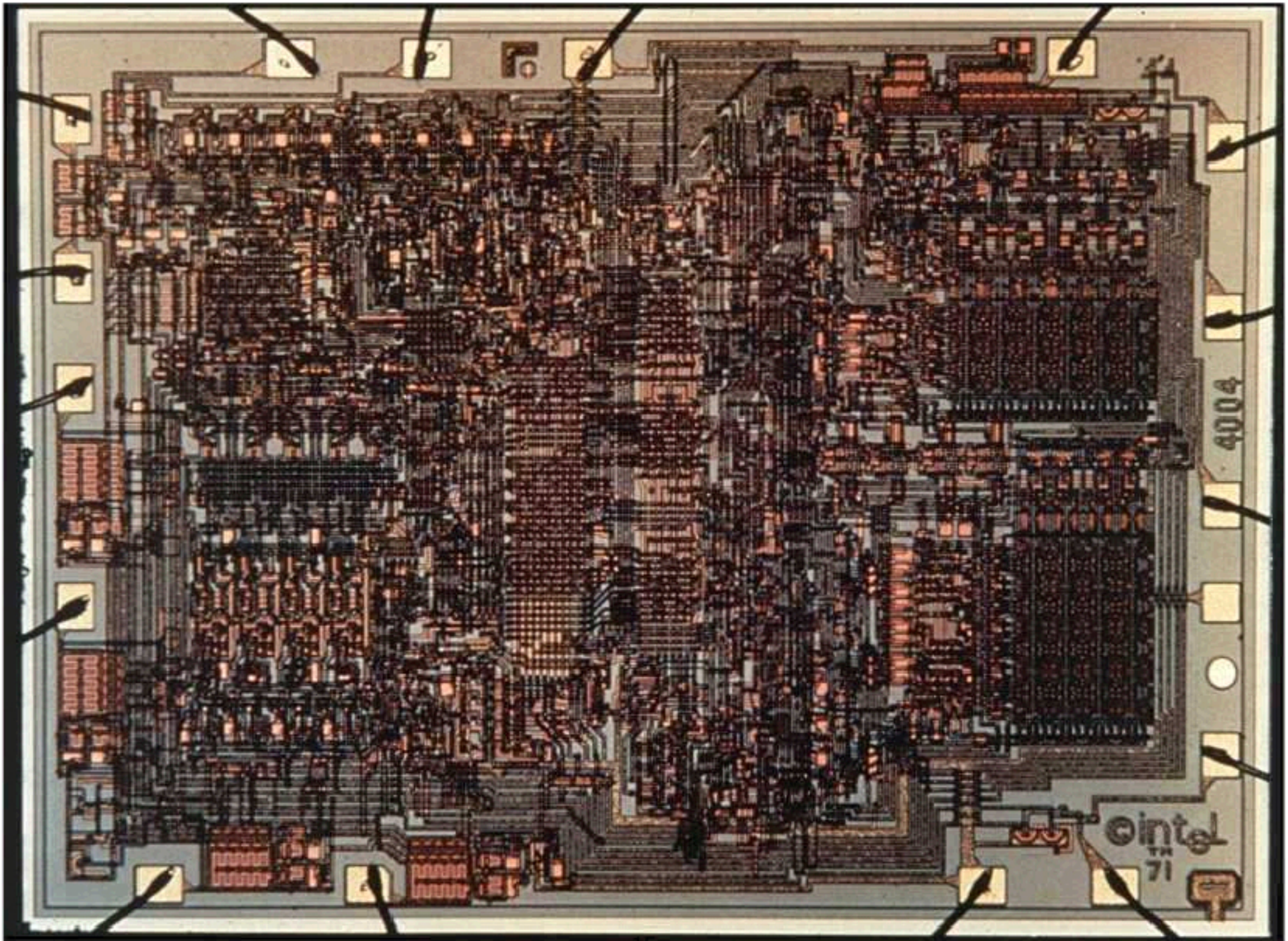
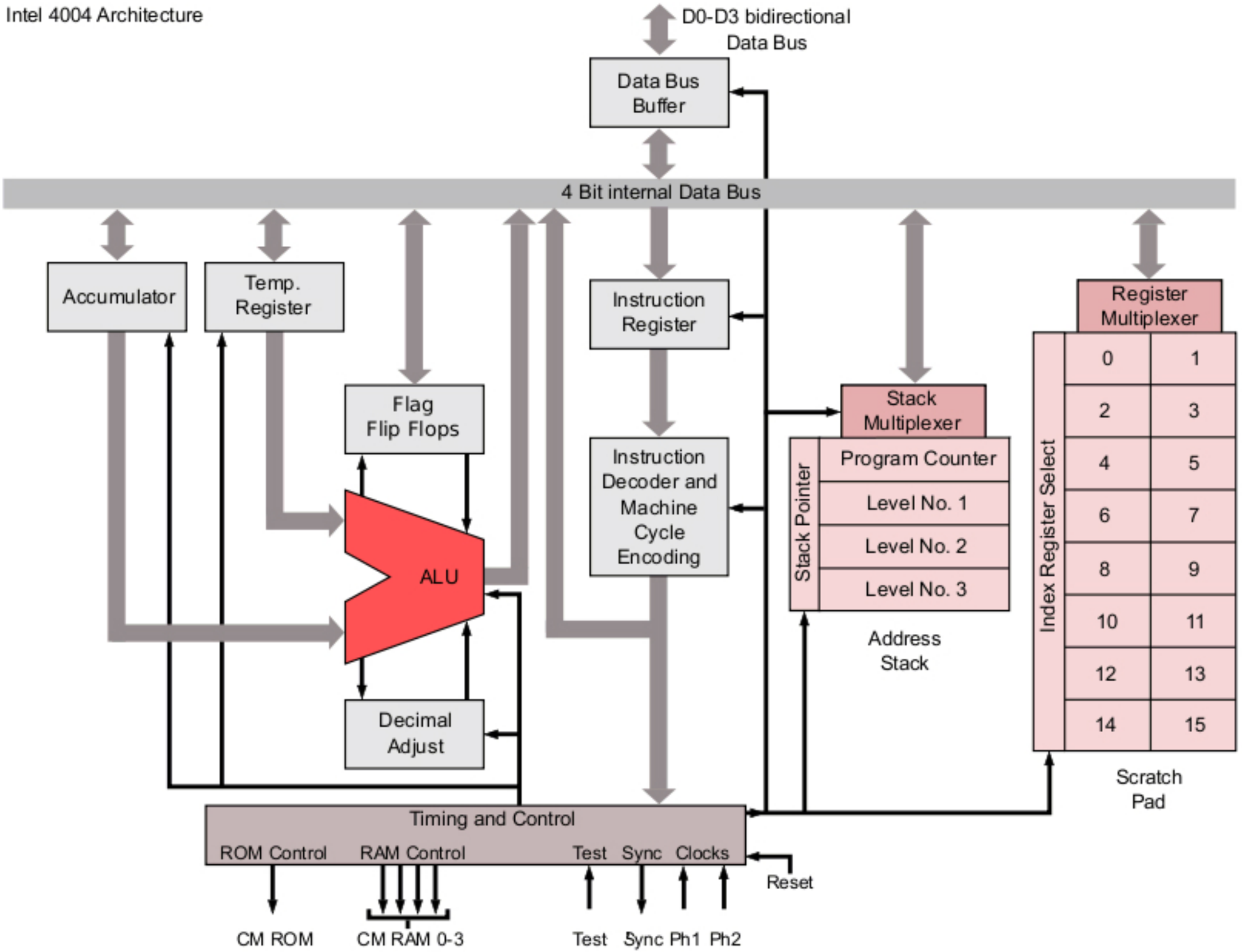
Single crystal silicon



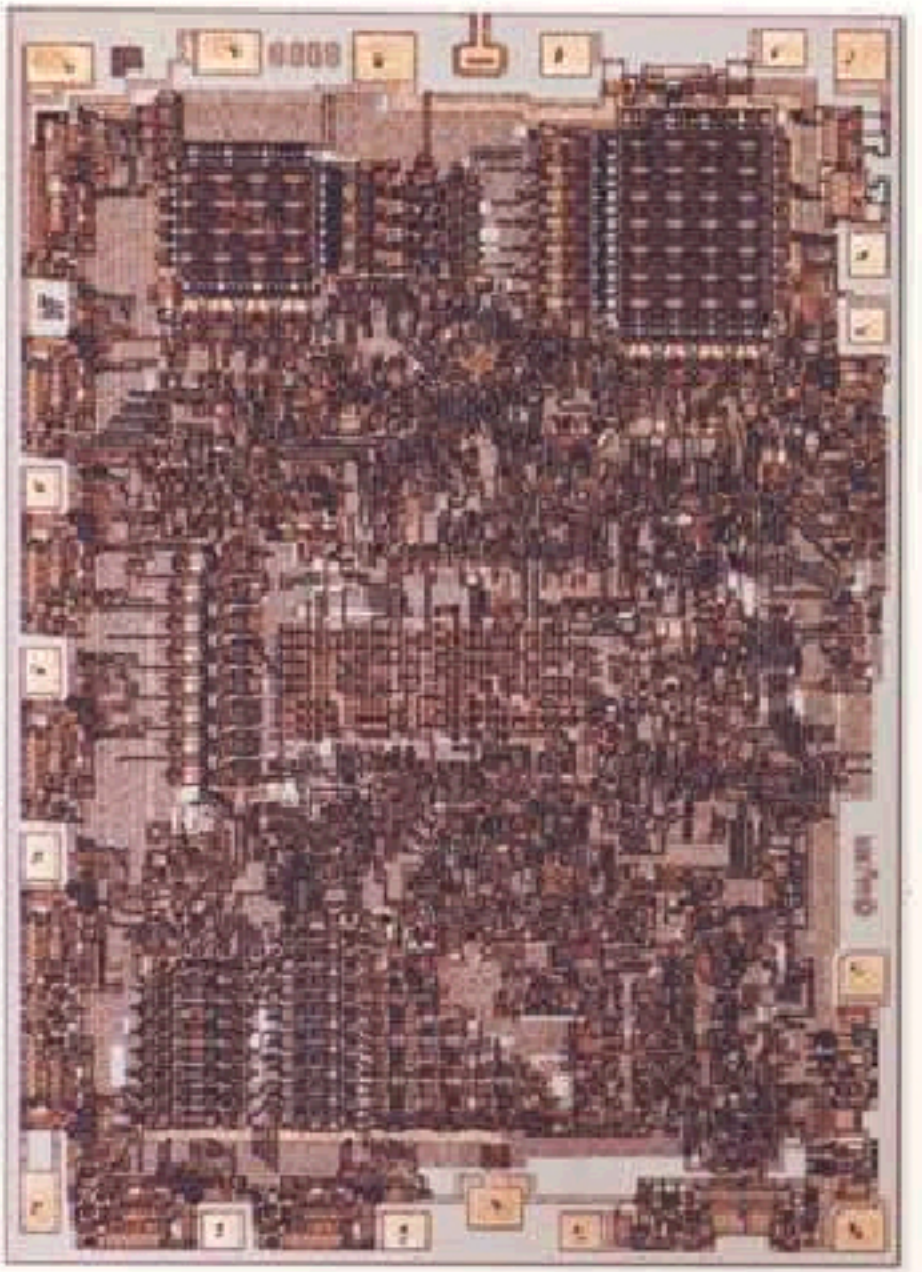
Intel 4004 (1971), 4 bit, 2300 transistors



Intel 4004 (1971), 4 bit, 2300 transistors



Intel 8008 (1972), 8/14 bit, 2500 transistors



Intel 8086 (1978), 16 bit, 20 000 transistors

Basic Concepts of Microprocessors

- Differences between:
  - Microcomputer – a computer with a microprocessor as its CPU. Includes memory, I/O etc.
  - Microprocessor – silicon chip which includes ALU, register circuits & control circuits
  - Microcontroller – silicon chip which includes microprocessor, memory & I/O in a single package.

What is a Microprocessor?

- The word comes from the combination micro and processor.
  - Processor means a device that processes whatever. In this context processor means a device that processes numbers, specifically binary numbers, 0’s and 1’s.
    - To process means to manipulate. It is a general term that describes all manipulation. Again in this content, it means to perform certain operations on the numbers that depend on the microprocessor’s design.

What about micro?

- Micro is a new addition.
  - In the late 1960's, processors were built using discrete elements.
    - These devices performed the required operation, but were too large and too slow.
  - In the early 1970's the microchip was invented. All of the components that made up the processor were now placed on a single piece of silicon. The size became several thousand times smaller and the speed became several hundred times faster. The "Micro"Processor was born.

Definition of the Microprocessor

The microprocessor is a programmable device that takes in numbers, performs on them arithmetic or logical operations according to the program stored in memory and then produces other numbers as a result.

Definition (Contd.)

- Lets expand each of the underlined words:
  - Programmable device: The microprocessor can perform different sets of operations on the data it receives depending on the sequence of instructions supplied in the given program.
    By changing the program, the microprocessor manipulates the data in different ways.

  - Instructions: Each microprocessor is designed to execute a specific group of operations. This group of operations is called an instruction set. This instruction set defines what the microprocessor can and cannot do.

Definition (Contd.)

- Takes in: The data that the microprocessor manipulates must come from somewhere.
  - It comes from what is called “input devices”.
  - These are devices that bring data into the system from the outside world.
  - These represent devices such as a keyboard, a mouse, switches, and the like.

Definition (Contd.)

- Numbers: The microprocessor has a very narrow view on life. It only understands binary numbers.

  A binary digit is called a bit (which comes from **b**inary dig**it**).

  The microprocessor recognizes and processes a group of bits together. This group of bits is called a “word”.

  The number of bits in a Microprocessor’s word, is a measure of its “abilities”.

Definition (Contd.)

- Words, Bytes, etc.
  - The earliest microprocessor (the Intel 8088 and Motorola's 6800) recognized 8-bit words.
    - They processed information 8-bits at a time. That's why they are called "8-bit processors". They can handle large numbers, but in order to process these numbers, they broke them into 8-bit pieces and processed each group of 8-bits separately.
  - Later microprocessors (8086 and 68000) were designed with 16-bit words.
    - A group of 8-bits were referred to as a "half-word" or "byte".
    - A group of 4 bits is called a "nibble".
    - Also, 32 bit groups were given the name "long word".
  - Today, all processors manipulate at least 32 bits at a time and there exists microprocessors that can process 64, 80, 128 bits

Definition (Contd.)

- Arithmetic and Logic Operations:
  - Every microprocessor has arithmetic operations such as add and subtract as part of its instruction set.
    - Most microprocessors will have operations such as multiply and divide.
    - Some of the newer ones will have complex operations such as square root.
  - In addition, microprocessors have logic operations as well. Such as AND, OR, XOR, shift left, shift right, etc.
  - Again, the number and types of operations define the microprocessor's instruction set and depends on the specific microprocessor.

Definition (Contd.)

- Stored in memory :
  - First, what is memory?
    - Memory is the location where information is kept while not in current use.
    - Memory is a collection of storage devices. Usually, each storage device holds one bit. Also, in most kinds of memory, these storage devices are grouped into groups of 8. These 8 storage locations can only be accessed together. So, one can only read or write in terms of bytes to and form memory.
    - Memory is usually measured by the number of bytes it can hold. It is measured in Kilos, Megas and lately Gigas. A Kilo in computer language is 2^{10} =1024. So, a KB (KiloByte) is 1024 bytes. Mega is 1024 Kilos and Giga is 1024 Mega.

Definition (Contd.)

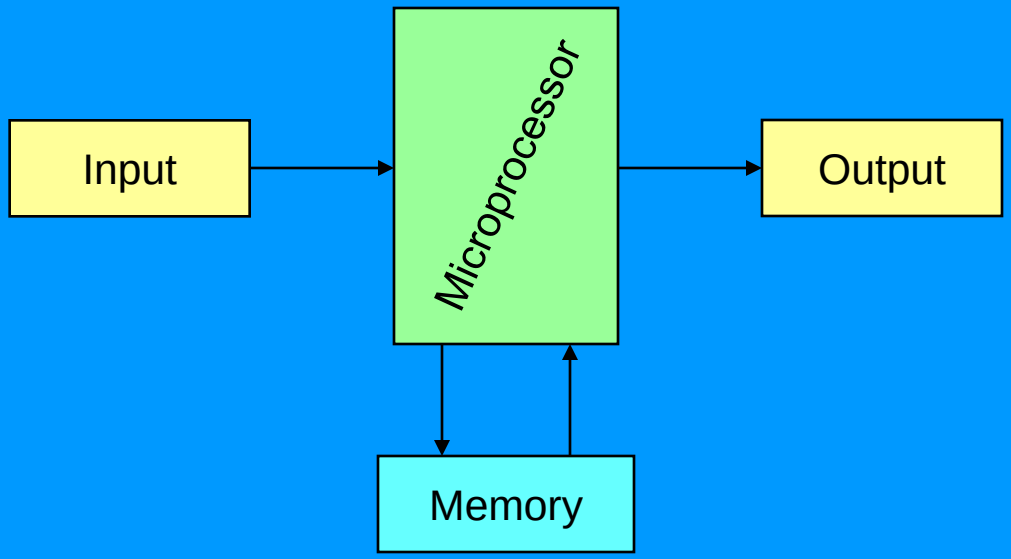
- Stored in memory:
  - When a program is entered into a computer, it is stored in memory. Then as the microprocessor starts to execute the instructions, it brings the instructions from memory one at a time.
  - Memory is also used to hold the data.
    - The microprocessor reads (brings in) the data from memory when it needs it and writes (stores) the results into memory when it is done.

Definition (Contd.)

- Produces: For the user to see the result of the execution of the program, the results must be presented in a human readable form.
  - The results must be presented on an output device.
  - This can be the monitor, a paper from the printer, a simple LED or many other forms.

A Microprocessor-based system

From the above description, we can draw the following block diagram to represent a microprocessor-based system:

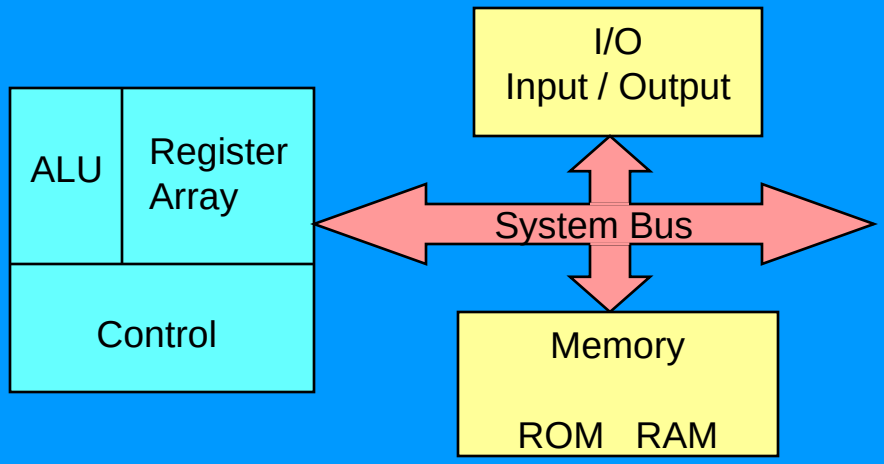


Inside The Microprocessor

- Internally, the microprocessor is made up of 3 main units.
  - The Arithmetic/Logic Unit (ALU)
  - The Control Unit.
  - An array of registers for holding data while it is being manipulated.

Organization of a microprocessor-based system

- Let’s expand the picture a bit.



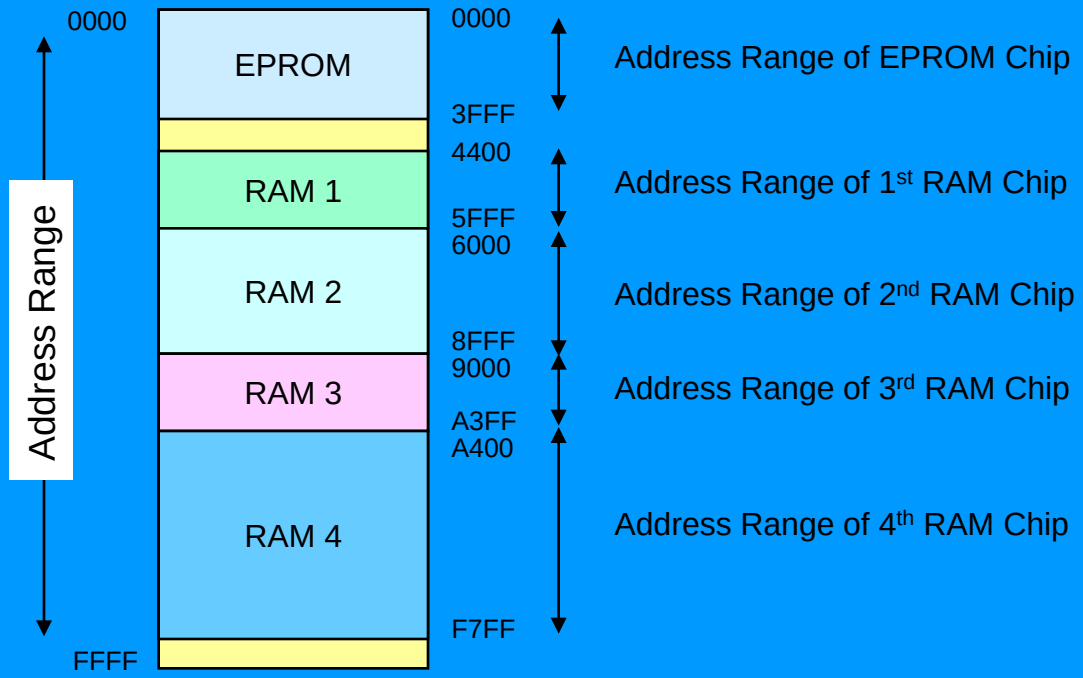
Memory

- Memory stores information such as instructions and data in binary format (0 and 1). It provides this information to the microprocessor whenever it is needed.

- Usually, there is a memory “sub-system” in a microprocessor-based system. This sub-system includes:
  - The registers inside the microprocessor
  - Read Only Memory (ROM)
    - used to store information that does not change.
  - Random Access Memory (RAM) (also known as Read/Write Memory).
    - used to store information supplied by the user. Such as programs and data.

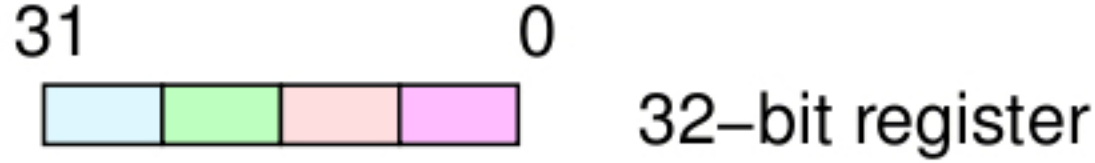
Memory Map and Addresses

- The memory map is a picture representation of the address range and shows where the different memory chips are located within the address range.



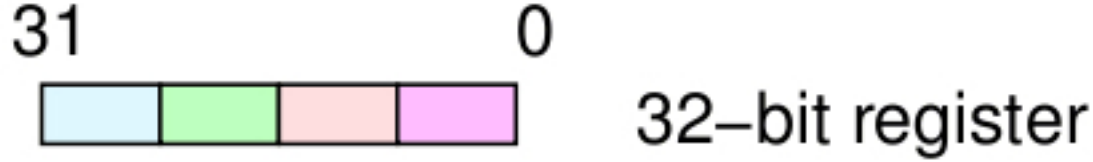
Endianess

How to store integers with a size larger than 8 bits?



Endianess

How to store integers with a size larger than 8 bits?



- affects only integers (floats are standardized)
- ARM: bi-endian (endianess is selected during startup)

Memory

- To execute a program:
  - the user enters its instructions in binary format into the memory.
  - The microprocessor then reads these instructions and whatever data is needed from memory, executes the instructions and places the results either in memory or produces it on an output device.

The three cycle instruction execution model

- To execute a program, the microprocessor “reads” each instruction from memory, “interprets” it, then “executes” it.

- To use the right names for the cycles:
  - The microprocessor **fetches** each instruction,
  - **decodes** it,
  - Then **executes** it.

- This sequence is continued until all instructions are performed.

Machine Language

- The number of bits that form the “word” of a microprocessor is fixed for that particular processor.
  - These bits define a maximum number of combinations.
    - For example an 8-bit microprocessor can have at most $2^8 = 256$ different combinations.

- However, in most microprocessors, not all of these combinations are used.
  - Certain patterns are chosen and assigned specific meanings.
  - Each of these patterns forms an instruction for the microprocessor.
  - The complete set of patterns makes up the microprocessor’s machine language.

The 8085 Machine Language

- The 8085 (from Intel) is an 8-bit microprocessor.
  - The 8085 uses a total of 246 bit patterns to form its instruction set.
  - These 246 patterns represent only 74 instructions.
    - The reason for the difference is that some (actually most) instructions have multiple different formats.
  - Because it is very difficult to enter the bit patterns correctly, they are usually entered in hexadecimal instead of binary.
    - For example, the combination 0011 1100 which translates into “increment the number in the register called the accumulator”, is usually entered as 3C.

Assembly Language

- Entering the instructions using hexadecimal is quite easier than entering the binary combinations.
  - However, it still is difficult to understand what a program written in hexadecimal does.
  - So, each company defines a symbolic code for the instructions.
  - These codes are called “mnemonics”.
  - The mnemonic for each instruction is usually a group of letters that suggest the operation performed.

Assembly Language

- Using the same example from before,
  - 00111100 translates to 3C in hexadecimal (OPCODE)
  - Its mnemonic is: “INR A”.
  - INR stands for “increment register” and A is short for accumulator.
- Another example is: 1000 0000,
  - Which translates to 80 in hexadecimal.
  - Its mnemonic is “ADD B”.
  - “Add register B to the accumulator and keep the result in the accumulator”.

Assembly Language

- It is important to remember that a machine language and its associated assembly language are completely machine dependent.
  - In other words, they are not transferable from one microprocessor to a different one.

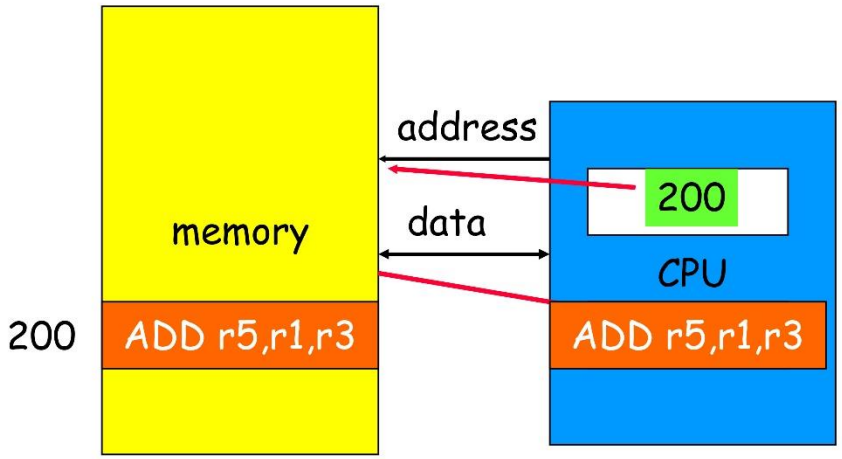
- For example, Motorolla has an 8-bit microprocessor called the 6800.
  - The 8085 machine language is very different from that of the 6800. So is the assembly language.
  - A program written for the 8085 cannot be executed on the 6800 and vice versa.

“Assembling” The Program

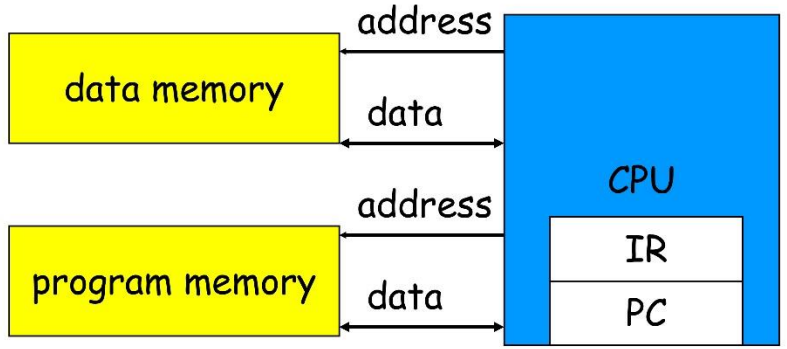
- How does assembly language get translated into machine language?
  - There are two ways:
  - 1st there is “**hand assembly**”.
    - The programmer translates each assembly language instruction into its equivalent hexadecimal code (machine language). Then the hexadecimal code is entered into memory.
  - The other possibility is a program called an “**assembler**”, which does the translation automatically.

- System Bus – wires connecting memory & I/O to microprocessor
  - Address Bus
    - Unidirectional
    - Identifying peripheral or memory location
  - Data Bus
    - Bidirectional
    - Transferring data
  - Control Bus
    - Synchronization signals
    - Timing signals
    - Control signal

von Neumann



Harvard Architecture



von Neumann vs. Harvard

- von Neumann
  - Same memory holds data, instructions.
  - A single set of address/data buses between CPU and memory
- Harvard
  - Separate memories for data and instructions.
  - Two sets of address/data buses between CPU and memory

von Neumann vs. Harvard

- Harvard allows two simultaneous memory fetches.
- Most DSPs use Harvard architecture for streaming data:
  - greater memory bandwidth;
  - more predictable bandwidth.

RISC vs. CISC

- Reduced Instruction Set Computer (RISC)
  - Compact, uniform instructions → facilitate pipelining
  - More lines of code → large memory footprint
  - Allow effective compiler optimization
- Complex Instruction Set Computer (CISC)
  - Many addressing modes and long instructions
  - High code density
  - Often require manual optimization of assembly code for embedded systems

Microprocessors

	von Neumann	Harvard
RISC	**ARM7**	**ARM9**
CISC	**Pentium**	**SHARC (DSP)**

Intel 8085 Microarchitecture

INTR
INTA#
RST 5.5
RST 6.5
RST 7.5
TRAP
SID
SOD
Interrupt Control
Serial I/O Control
8 Bit internal Data Bus
Accumulator (8 Bit)
Temp. Register
Flag Register (8 Bit)
ALU
Instruction Register (8 Bit)
Instruction Decoder and Machine Cycle Encoding
Register
B Reg. (8 Bit)
C Reg. (8 Bit)
D Reg. (8 Bit)
E Reg. (8 Bit)
H Reg. (8 Bit)
L Reg. (8 Bit)
Stack Pointer (16 Bit)
Program Counter (16 Bit)
Incrementer/Decrementer Address Latch
8 Bit
Address Bus (High). 8 Bit
Address Bus (Low). 8 Bit
Multiplexer
Timing and Control
X2
X1
CLK GEN
RESET
DMA
STATUS
CONTROL
CLK OUT
RESET IN#
RESET OUT
HOLD
HLDA
S0
S1
IO/M#
READY
RD#
WR#
Address Buffer
Data/Address Buffer
AD0-AD7
Decoder
ALE
Address Latch
MEMR#
MEMW#
IOR#
IOW#
A8-A15 Address Bus
A0-A7 Address Bus
D0-D7 Data Bus

Intel 8085 Microprocessor

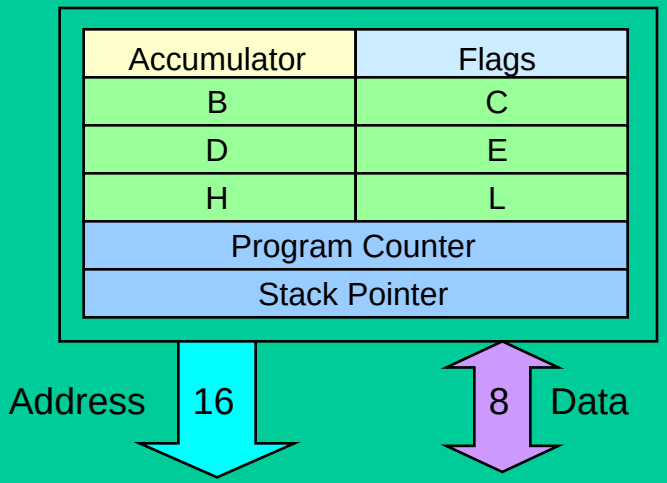
- Microprocessor consists of:
  - **Control unit**: control microprocessor operations.
  - **ALU**: performs data processing function.
  - **Registers**: provide storage internal to CPU.
  - **Interrupts**
  - **Internal data bus**

The ALU

- In addition to the arithmetic & logic circuits, the ALU includes the accumulator, which is part of every arithmetic & logic operation.

- Also, the ALU includes a temporary register used for holding data temporarily during the execution of the operation. This temporary register is not accessible by the programmer.

- Registers
  - General Purpose Registers
    - B, C, D, E, H & L (8 bit registers)
    - Can be used singly
    - Or can be used as 16 bit register pairs
      - BC, DE, HL
    - H & L can be used as a data pointer (holds memory address)
  - Special Purpose Registers
    - Accumulator (8 bit register)
      - Store 8 bit data
      - Store the result of an operation
      - Store 8 bit data during I/O transfer



- Flag Register
  - 8 bit register – shows the status of the microprocessor before/after an operation
  - S (sign flag), Z (zero flag), AC (auxillary carry flag), P (parity flag) & CY (carry flag)

D7	D6	D5	D4	D3	D2	D1	D0
S	Z	X	AC	X	P	X	CY

  - Sign Flag
    - Used for indicating the sign of the data in the accumulator
    - The sign flag is set if negative (1 – negative)
    - The sign flag is reset if positive (0 –positive)

- Zero Flag
  - Is set if result obtained after an operation is 0
  - Is set following an increment or decrement operation of that register

```
  10110011
+ 01001101
----------
1 00000000
```

- Carry Flag
  - Is set if there is a carry or borrow from arithmetic operation

```
    1011 0101
+   0110 1100
    ---------
Carry 1 0010 0001
```

```
    1011 0101
-   1100 1100
    ---------
Borrow 1 1110 1001
```

- Auxillary Carry Flag
  - Is set if there is a carry out of bit 3
- Parity Flag
  - Is set if parity is even
  - Is cleared if parity is odd

The Internal Architecture

- We have already discussed the general purpose registers, the Accumulator, and the flags.

- The Program Counter (PC)
  - This is a register that is used to control the sequencing of the execution of instructions.
  - This register always holds the address of the next instruction.
  - Since it holds an address, it must be 16 bits wide.

The Internal Architecture

- The Stack pointer
  - The stack pointer is also a 16-bit register that is used to point into memory.
  - The memory this register points to is a special area called the stack.
  - The stack is an area of memory used to hold data that will be retreived soon.
  - The stack is usually accessed in a Last In First Out (LIFO) fashion.

Non Programmable Registers

- Instruction Register & Decoder
  - Instruction is stored in IR after fetched by processor
  - Decoder decodes instruction in IR

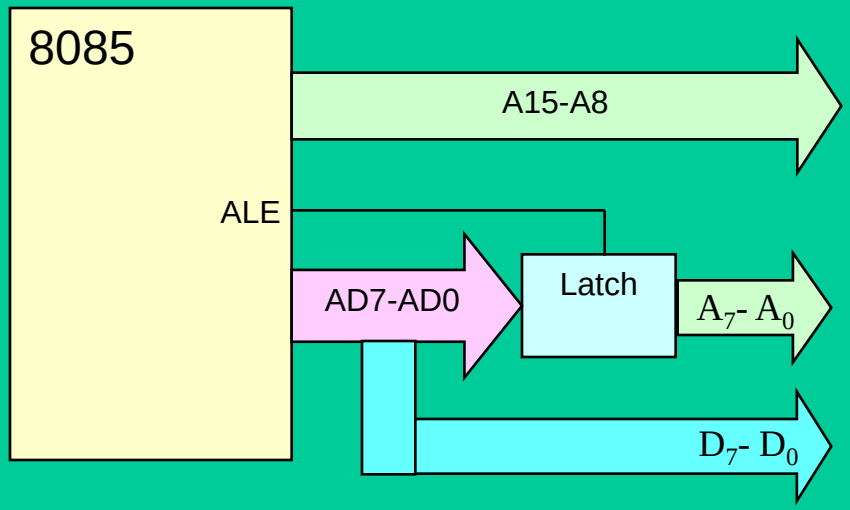
Internal Clock generator

- 3.125 MHz internally
- 6.25 MHz externally

The Address and Data Busses

- The address bus has 8 signal lines A8 – A15 which are unidirectional.
- The other 8 address bits are multiplexed (time shared) with the 8 data bits.
  - So, the bits AD0 – AD7 are bi-directional and serve as A0 – A7 and D0 – D7 at the same time.
    - During the execution of the instruction, these lines carry the address bits during the early part, then during the late parts of the execution, they carry the 8 data bits.
  - In order to separate the address from the data, we can use a latch to save the value before the function of the bits changes.

Demultiplexing AD7-AD0



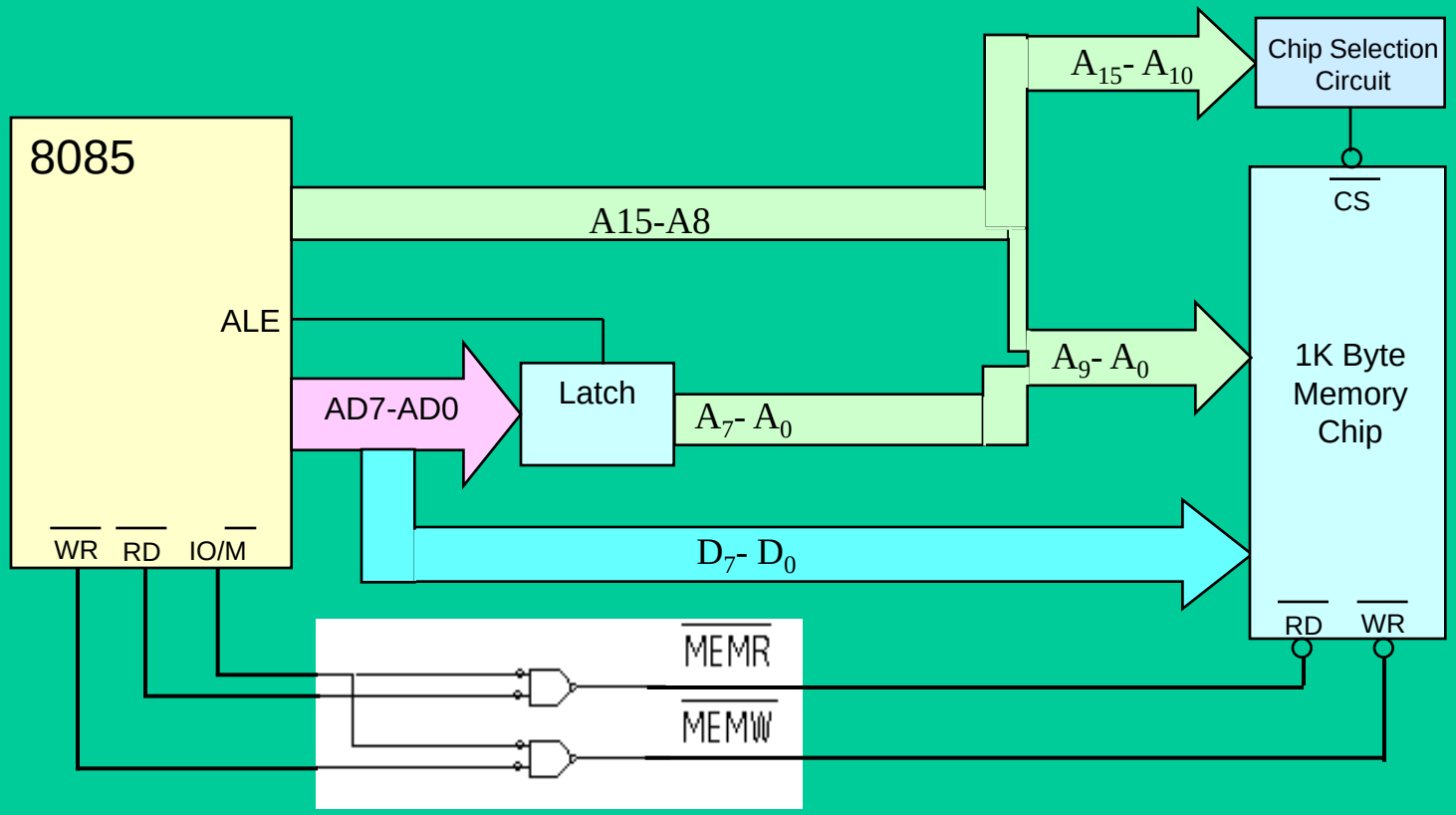
– Given that ALE operates as a pulse during T1, we will be able to latch the address. Then when ALE goes low, the address is saved and the AD7– AD0 lines can be used for their purpose as the bi-directional data lines.

Demultiplexing the Bus $AD_7 – AD_0$

- The high order address is placed on the address bus and hold for 3 clk periods,
- The low order address is lost after the first clk period, this address needs to be hold however we need to use latch
- The address AD7 – AD0 is connected as inputs to the latch 74LS373.
- The ALE signal is connected to the enable (G) pin of the latch and the OC – Output control – of the latch is grounded

The Overall Picture

- Putting all of the concepts together, we get:



Introduction to 8085 Instructions

The 8085 Instructions

- Since the 8085 is an 8-bit device it can have up to 2^8 (256) instructions.
  - However, the 8085 only uses 246 combinations that represent a total of 74 instructions.
    - Most of the instructions have more than one format.

- These instructions can be grouped into five different groups:
  - Data Transfer Operations
  - Arithmetic Operations
  - Logic Operations
  - Branch Operations
  - Machine Control Operations

Instruction and Data Formats

- Each instruction has two parts.
  - The first part is the task or operation to be performed.
    - This part is called the "opcode" (operation code).
  - The second part is the data to be operated on
    - Called the "operand".

Data Transfer Operations

- These operations simply **COPY** the data from the source to the destination.
- MOV, MVI, LDA, and STA

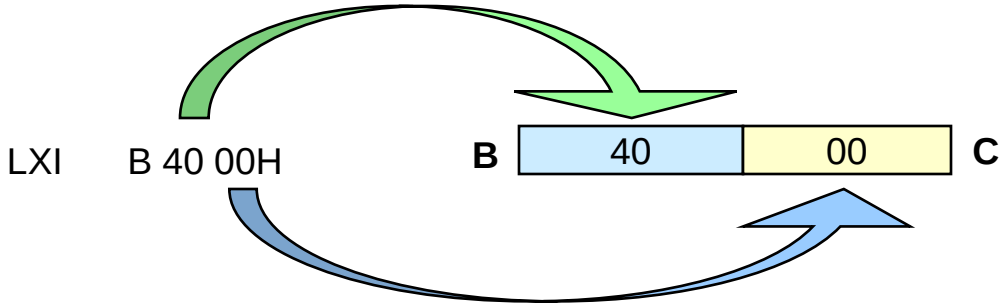
- They transfer:
  - Data between registers.
  - Data Byte to a register or memory location.
  - Data between a memory location and a register.
  - Data between an I\O Device and the accumulator.

- The data in the source is not changed.

The LXI instruction

- The 8085 provides an instruction to place the 16-bit data into the register pair in one step.
  - **LXI Rp, <16-bit address>** (Load eXtended Immediate)

  – The instruction **LXI B 4000H** will place the 16-bit number 4000 into the register pair B, C.
    - The upper two digits are placed in the 1st register of the pair and the lower two digits in the 2nd .



The Memory "Register"

- Most of the instructions of the 8085 can use a memory location in place of a register.
  - The memory location will become the "memory" register M.
    - **MOV M B**
      - copy the data from register B into a memory location.
  - Which memory location?

- The memory location is identified by the contents of the HL register pair.
  - The 16-bit contents of the HL register pair are treated as a 16-bit address and used to identify the memory location.

Using the Other Register Pairs

- There is also an instruction for moving data from memory to the accumulator without disturbing the contents of the H and L register.
  - **LDAX Rp** (LoaD Accumulator eXtended)
    - Copy the 8-bit contents of the memory location identified by the Rp register pair into the Accumulator.
    - This instruction only uses the **BC** or **DE** pair.
    - It does not accept the **HL** pair.

Indirect Addressing Mode

- Using data in memory directly (without loading first into a Microprocessor's register) is called Indirect Addressing.

- Indirect addressing uses the data in a register pair as a 16-bit address to identify the memory location being accessed.
  - The HL register pair is always used in conjunction with the memory register "M".
  - The BC and DE register pairs can be used to load data into the Accumultor using indirect addressing.

Arithmetic Operations

- Addition (ADD, ADI):
  - Any 8-bit number.
  - The contents of a register.
  - The contents of a memory location.
  - Can be added to the contents of the accumulator and the result is stored in the accumulator.
- Subtraction (SUB, SUI):
  - Any 8-bit number
  - The contents of a register
  - The contents of a memory location
  - Can be subtracted **from** the contents of the accumulator. The result is stored in the accumulator.

Arithmetic Operations Related to Memory

- These instructions perform an arithmetic operation using the contents of a memory location while they are still in memory.
  - ADD M
    - Add the contents of M to the Accumulator
  - SUB M
    - Sub the contents of M from the Accumulator
  - INR M / DCR M
    - Increment/decrement the contents of the memory location in place.
  - All of these use the contents of the HL register pair to identify the memory location being used.

Arithmetic Operations

- Increment (INR) and Decrement (DCR):
  - The 8-bit contents of any memory location or any register can be directly incremented or decremented by 1.
  - No need to disturb the contents of the accumulator.

Manipulating Addresses

- Now that we have a 16-bit address in a register pair, how do we manipulate it?
  - It is possible to manipulate a 16-bit address stored in a register pair as one entity using some special instructions.
    - **INX Rp** (Increment the 16-bit number in the register pair)
    - **DCX Rp** (Decrement the 16-bit number in the register pair)
  - The register pair is incremented or decremented as one entity. No need to worry about a carry from the lower 8-bits to the upper. It is taken care of automatically.

Logic Operations

- These instructions perform logic operations on the contents of the accumulator.
  - ANA, ANI, ORA, ORI, XRA and XRI
    - Source: Accumulator and
      - An 8-bit number
      - The contents of a register
      - The contents of a memory location
    - Destination: Accumulator

ANA R/M	AND Accumulator With Reg/Mem
ANI #	AND Accumulator With an 8-bit number
ORA R/M	OR Accumulator With Reg/Mem
ORI #	OR Accumulator With an 8-bit number
XRA R/M	XOR Accumulator With Reg/Mem
XRI #	XOR Accumulator With an 8-bit number

Logic Operations

– Complement:
  - 1’s complement of the contents of the accumulator.

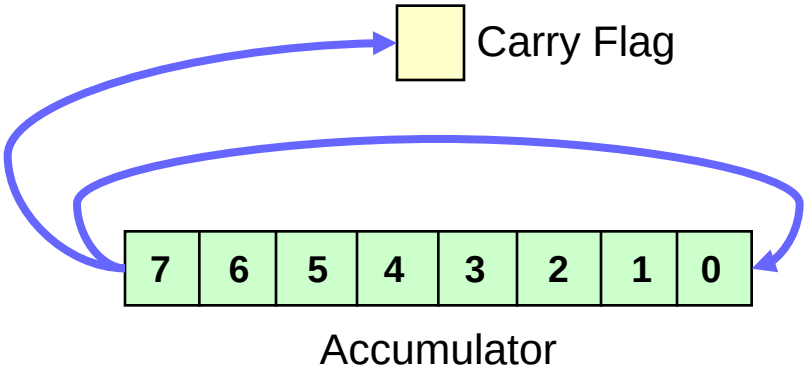
    CMA No operand

Additional Logic Operations

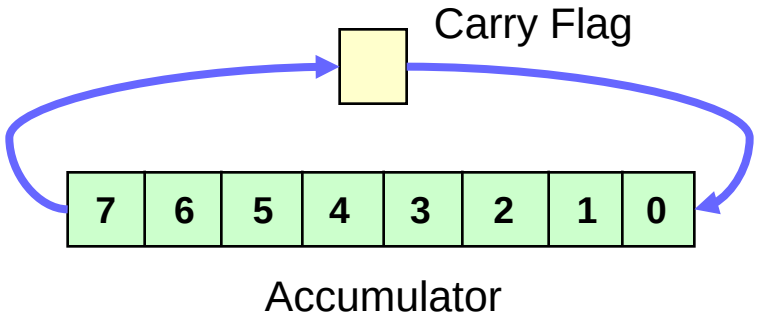
- Rotate
  - Rotate the contents of the accumulator one position to the left or right.
    - RLC — Rotate the accumulator left.
      Bit 7 goes to bit 0 AND the Carry flag.
    - RAL — Rotate the accumulator left through the carry.
      Bit 7 goes to the carry and carry goes to bit 0.
    - RRC — Rotate the accumulator right.
      Bit 0 goes to bit 7 AND the Carry flag.
    - RAR — Rotate the accumulator right through the carry.
      Bit 0 goes to the carry and carry goes to bit 7.

RLC vs. RLA

- 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 Date

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

      0011 1110 3E 1^{st} byte.

      0011 0010 32 2^{nd} 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
    - Load the accumulator with the contents of the memory location whose address is stored in the register pair BC).

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.

ASCII stands for American Standard Code for Information Interchange.

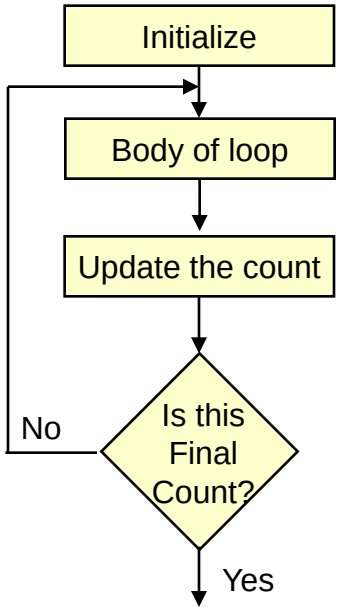
Counters & Time Delays

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
```

Using a 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.

Using a 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

- It was shown in Chapter 2 that each instruction passes through different combinations of Fetch, Memory Read, and Memory Write cycles.
- Knowing the combinations of cycles, one can calculate how long such an instruction would require to complete.
- The table in Appendix F of the book contains a column with the title B/M/T.
  - 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:

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.

Using a 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.

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

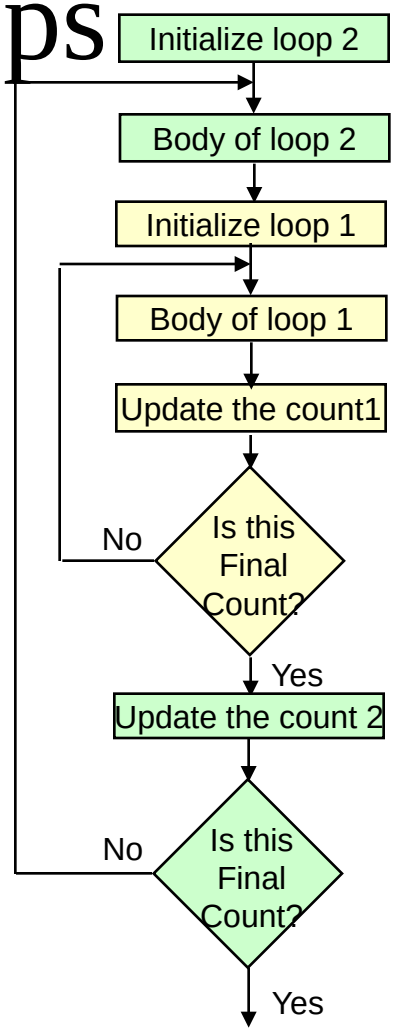
- Using the same formula from before, we can calculate:

- 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 ($FF_{16} = 255_{10}$) 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

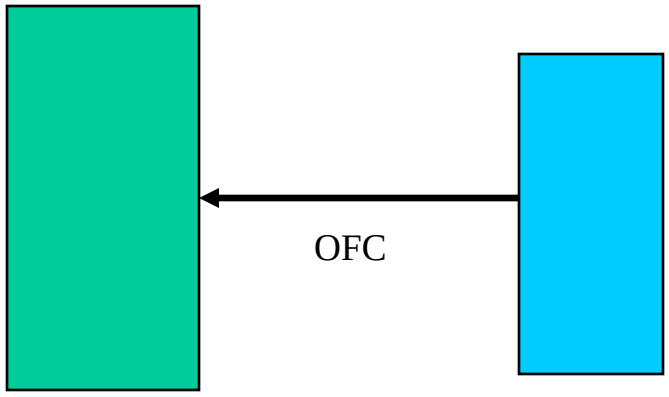
Timing Diagram

Instruction:

A000h MOV A,B

Corresponding Coding:

A000h 78



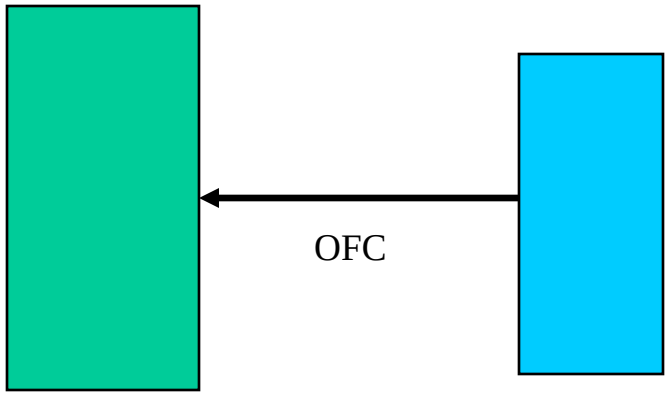
Timing Diagram

Instruction:

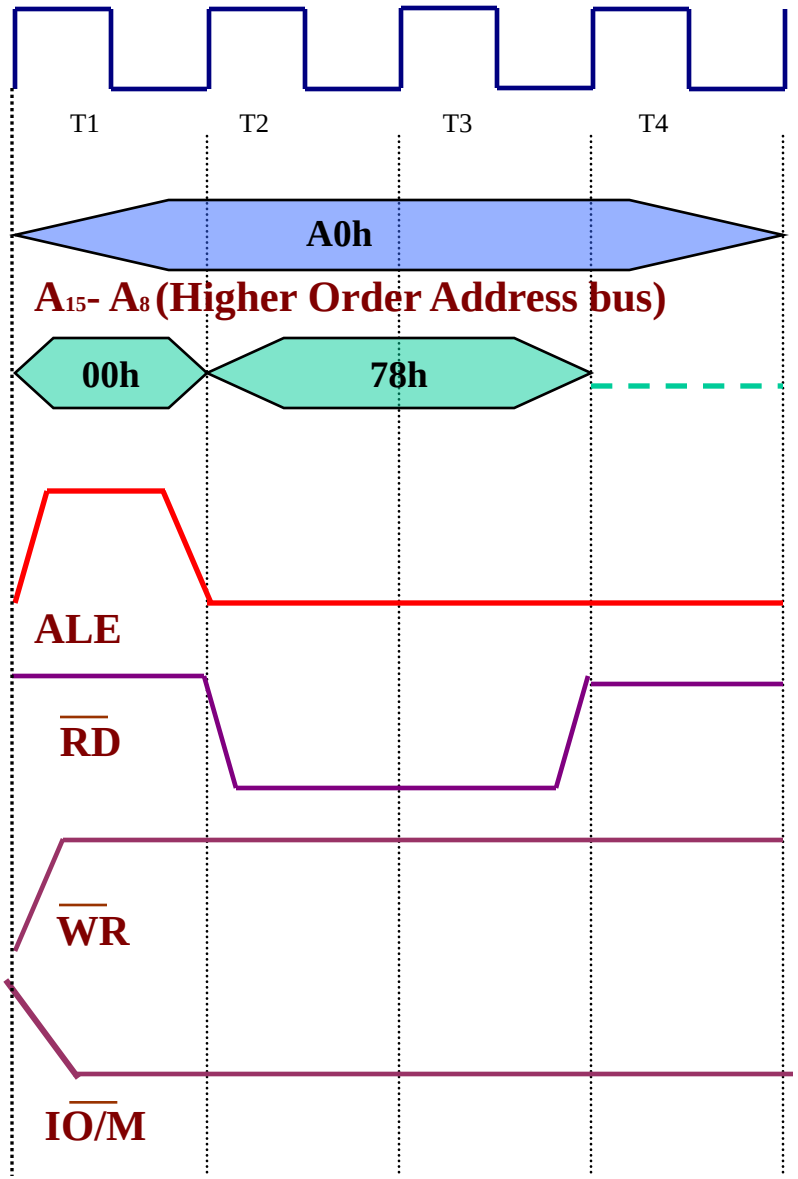
A000h MOV A,B

Corresponding Coding:

A000h 78



8085 Memory



Op-code fetch Cycle

Timing Diagram

Instruction:

A000h MVI A,45h

Corresponding Coding:

A000h 3E

A001h 45

Timing Diagram

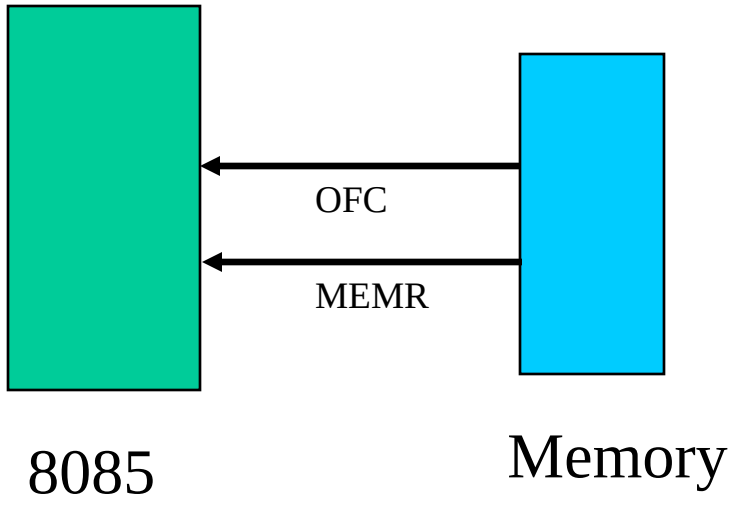
Instruction:

A000h MVI A,45h

Corresponding Coding:

A000h 3E

A001h 45



Timing Diagram

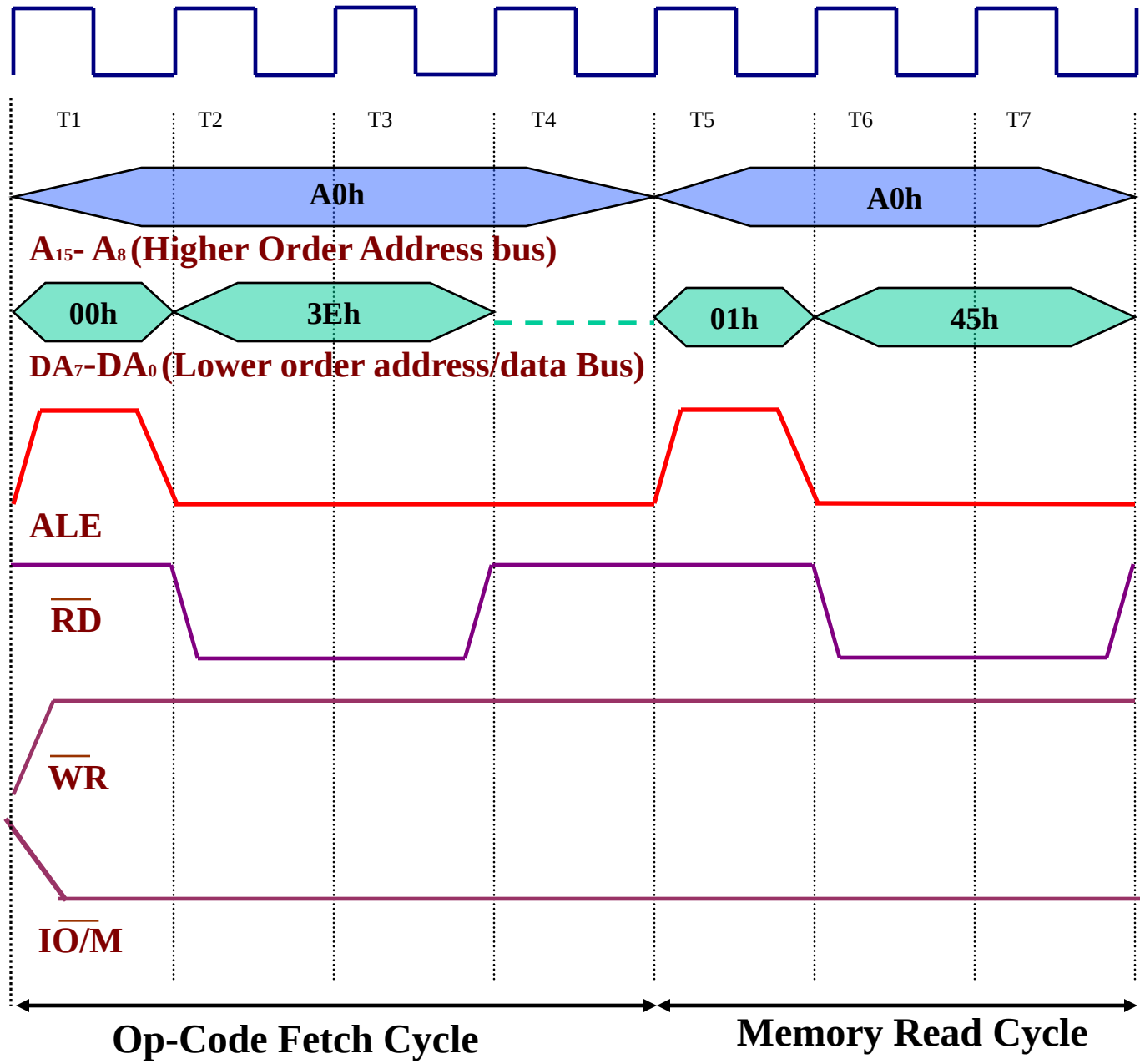
Instruction:

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

Instruction:

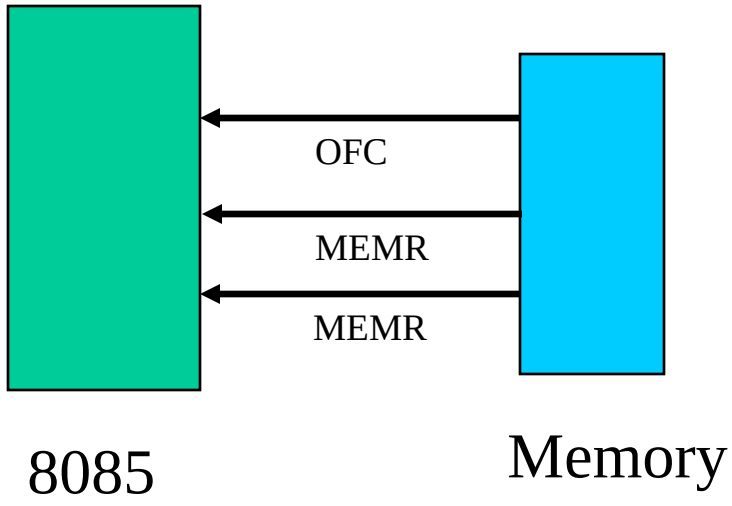
A000h LXI A,FO45h

Corresponding Coding:

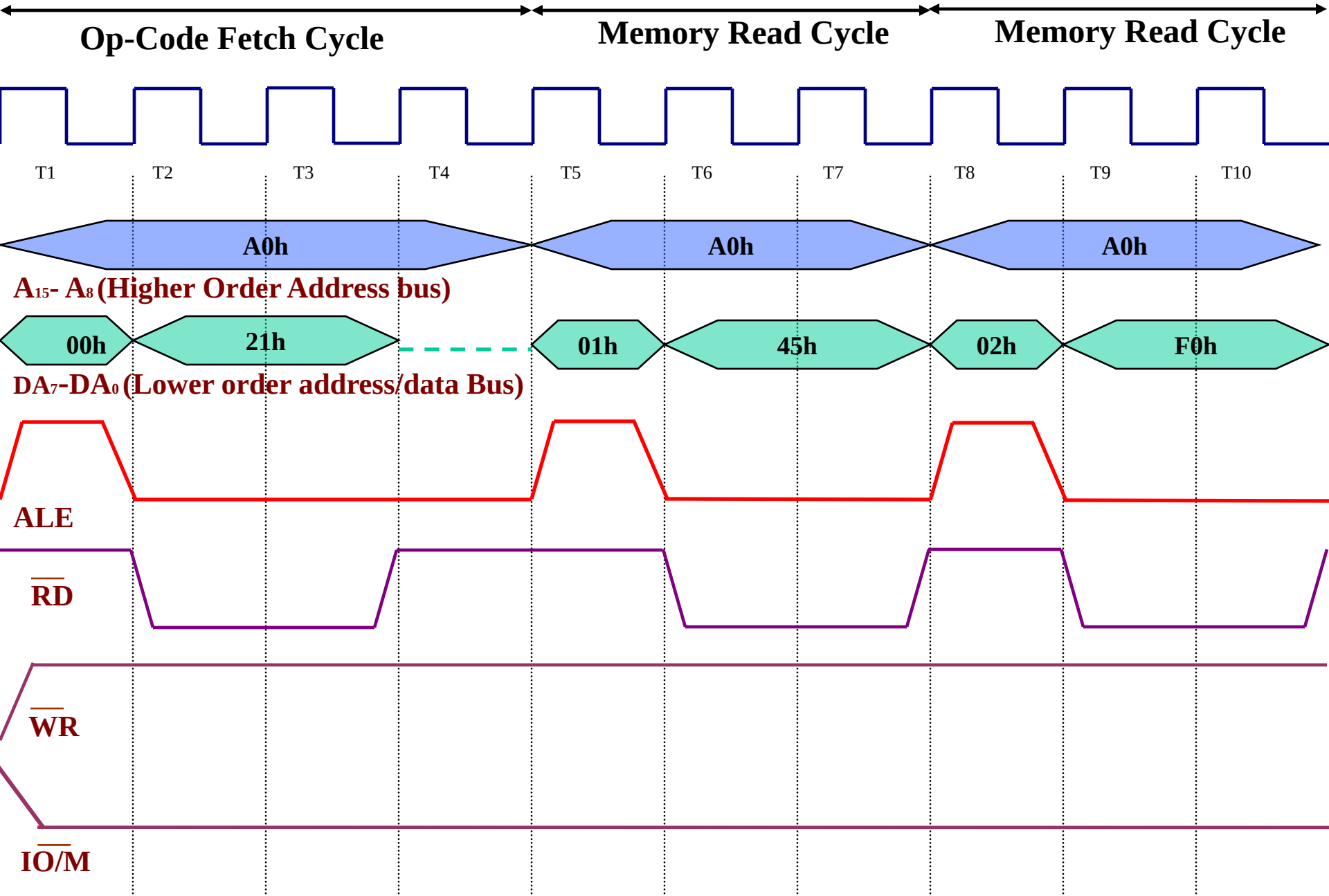
A000h 21

A001h 45

A002h F0



Timing Diagram



Timing Diagram

Instruction:

A000h MOV A,M

Corresponding Coding:

A000h 7E

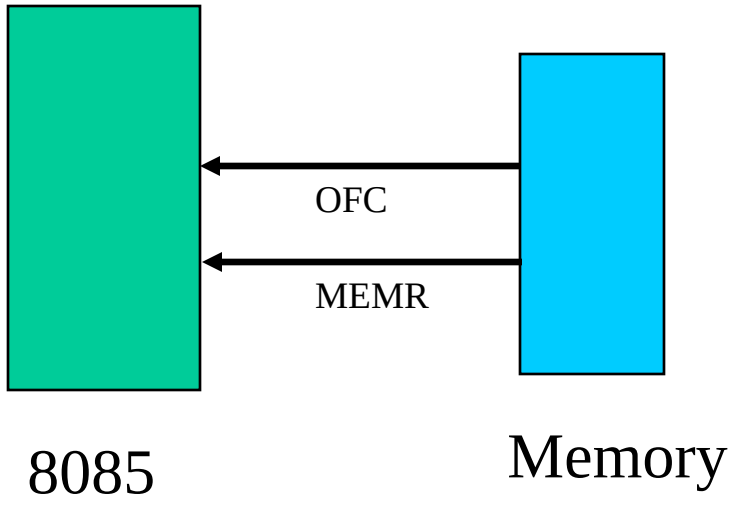
Timing Diagram

Instruction:

A000h MOV A,M

Corresponding Coding:

A000h 7E



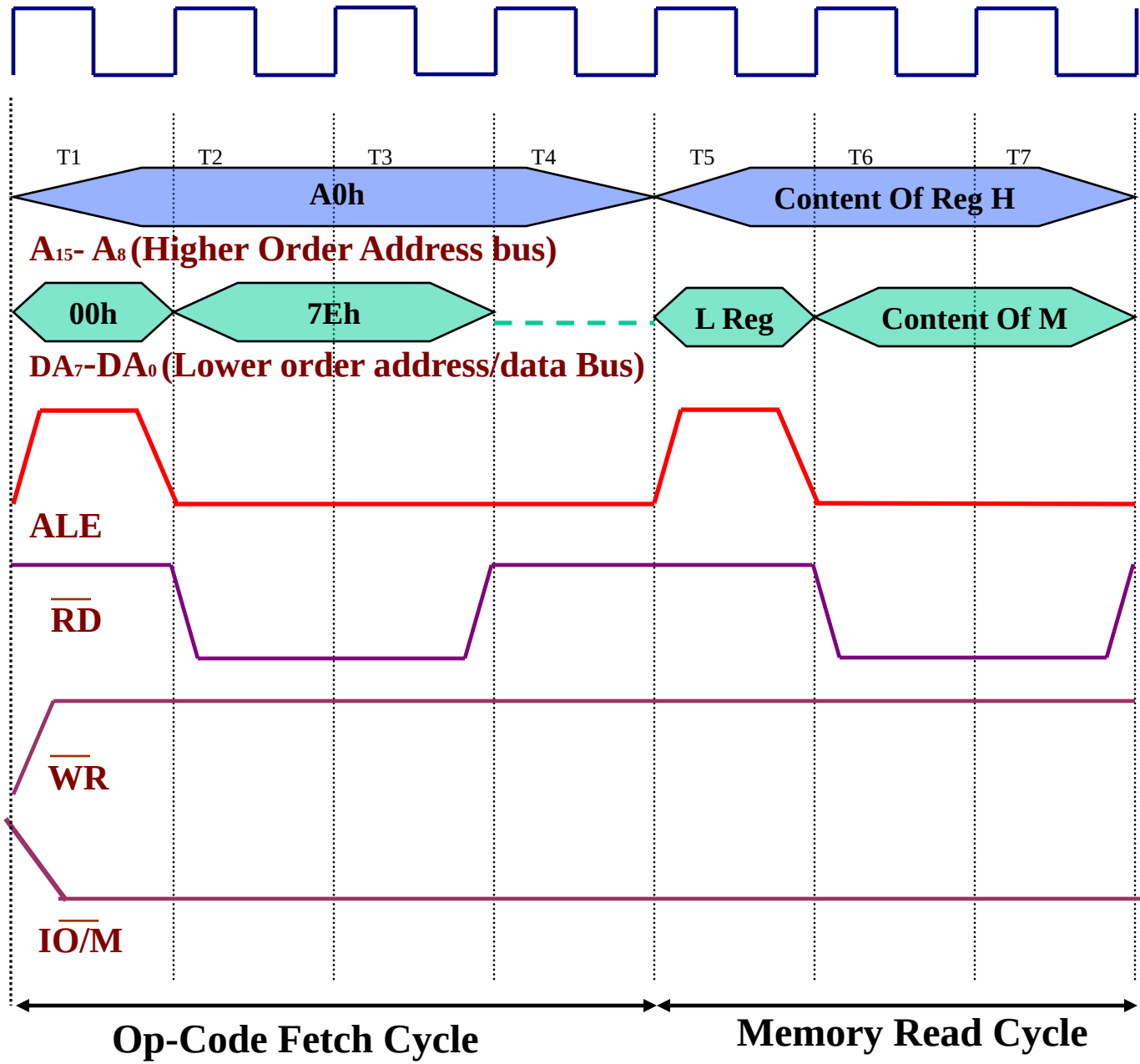
Timing Diagram

Instruction:

A000h MOV A,M

Corresponding Coding:

A000h 7E



Timing Diagram

Instruction:

A000h MOV M,A

Corresponding Coding:

A000h 77

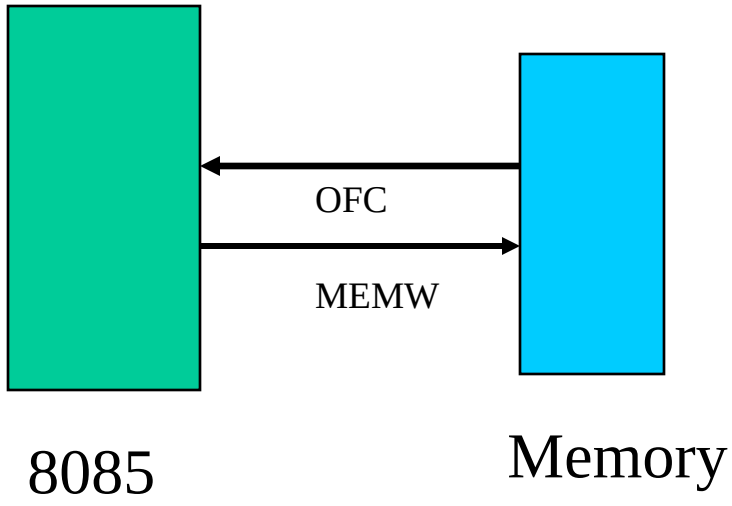
Timing Diagram

Instruction:

A000h MOV M,A

Corresponding Coding:

A000h 77



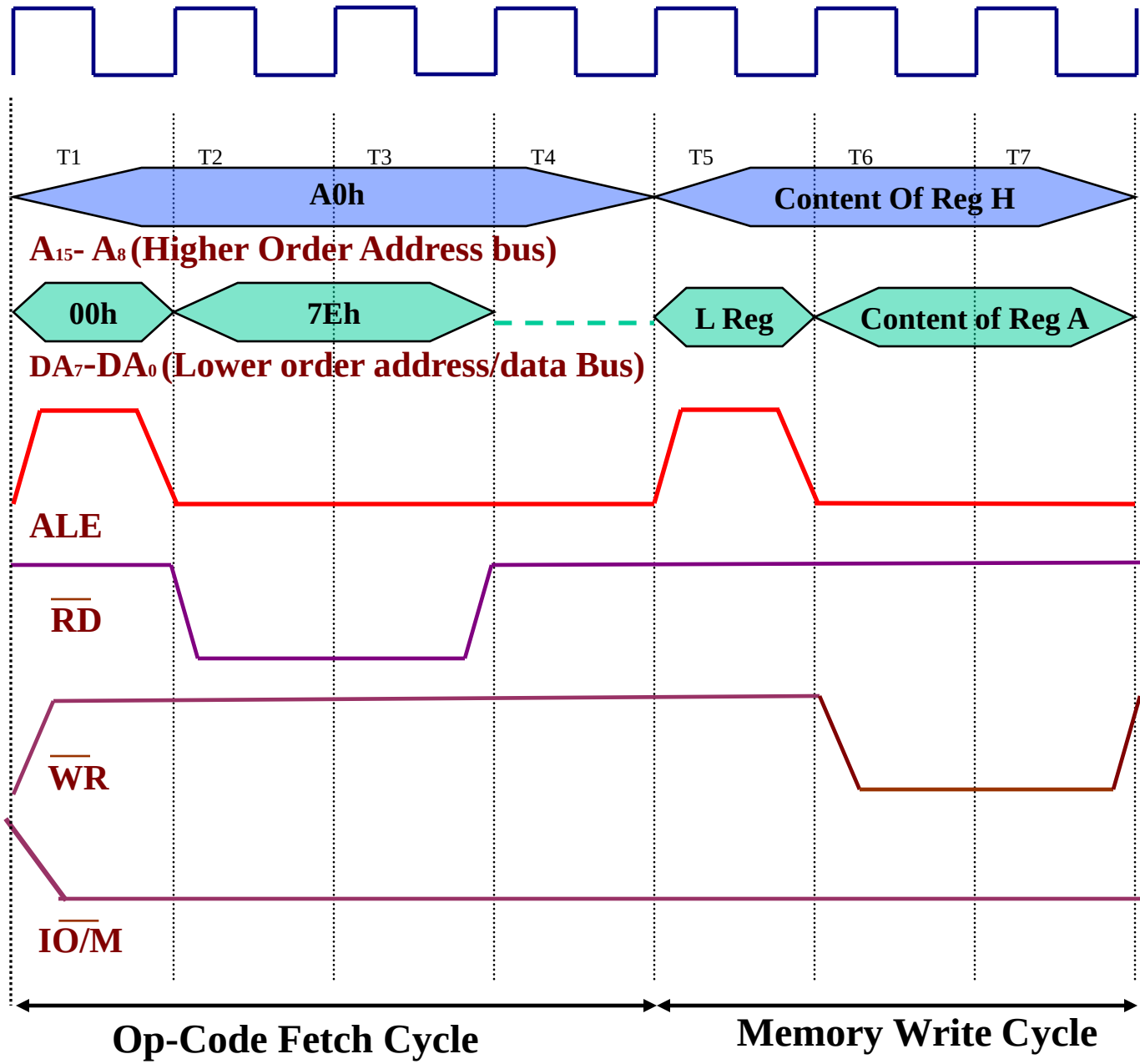
Timing Diagram

Instruction:

A000h MOV M,A

Corresponding Coding:

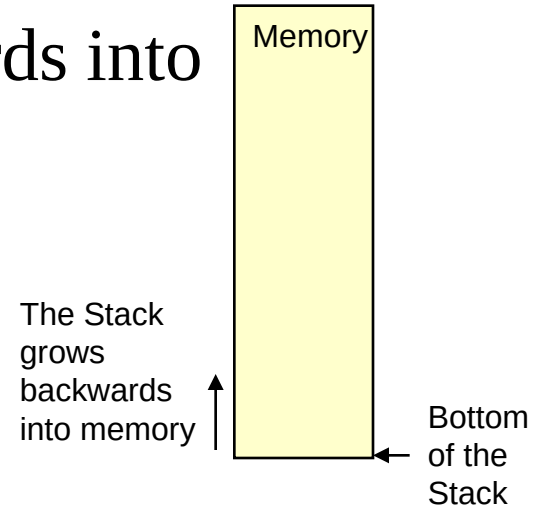
A000h 77



Chapter 9
Stack and Subroutines

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).

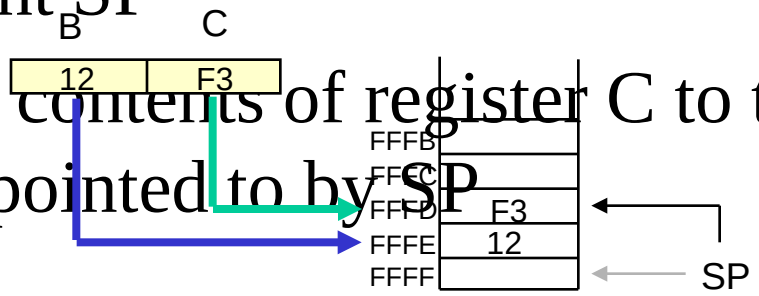
Saving Information on the Stack

- 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.

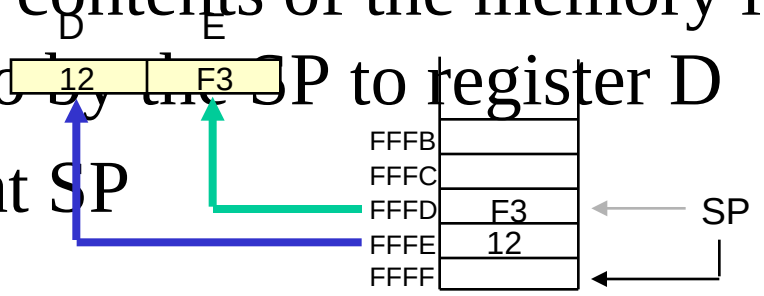
The PUSH Instruction

- 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

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.

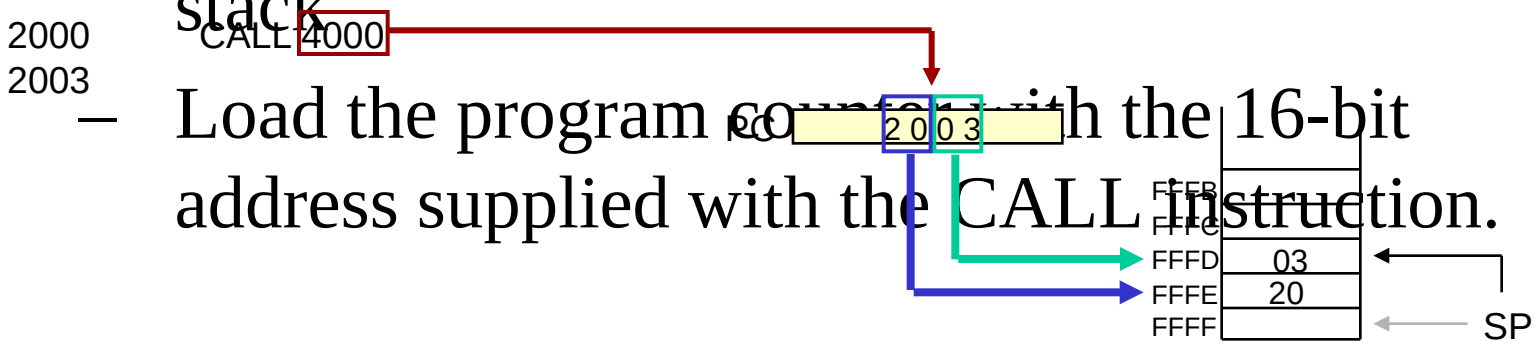
- In Assembly language, a subroutine can exist anywhere in the code.
  - However, it is customary to place subroutines separately from the main program.

Subroutines

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

The CALL Instruction

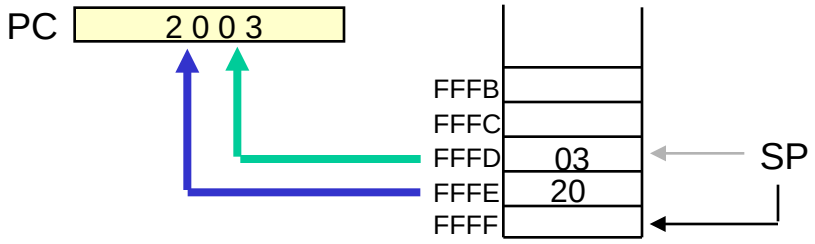
- CALL 4000H
  - Push the address of the instruction immediately following the CALL onto the stack.
  - Load the program counter with the 16-bit address supplied with the CALL instruction.



The RTE Instruction

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

```
4014        . . .
4015        RTE
```



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.

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 RTE Instructions

- The 8085 supports conditional CALL and conditional RTE 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 RTE
  - 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.