UNIT 1

1

MICROPROCESSOR, MICROCOMPUTERS, AND ASSEMBLY LANGUAGE

Unit Structure

1.0 Objectives

1.1 Introduction

1.2 Microprocessor

1.2.1 A Programmable Machine

1.2.2 Advances in Semiconductor Technology

1.2.3 Organization of a Microprocessor-based system

1.2.4 How does the Microprocessor work?

1.3 Microprocessor Instruction Set and Computer Languages

1.3.1 Machine Language

1.3.1.1 8085 Machine language

1.3.2 Assembly language

1.3.2.1 8085 Assembly language

1.3.2.2 Writing and executing an assembly language program

1.3.3 High-level language

1.3.4 Operating systems

1.4 From Large Computers to Single-Chip Microcontrollers

1.4.1 Large computers

1.4.2 Medium-size computers

1.4.3 Microcomputers

1.5 Application:

1.5.1 System hardware

1.5.2 System software

1.6 Summary

1.7 List of references

1.8 Unit End Exercise

1.0 Objectives

After going through this chapter, you will be able to

- Define microprocessor
- Define microprocessor based system and functions of each component
- Differentiate between machine language, assembly language and high-level language
- Classification of computers
- Design of basic microprocessor controlled temperature system (MCTS) application

1.1 Introduction

- Today microprocessor systems are used in every sphere of life with day to day demand increasing for faster and better systems.
- Microprocessors are multipurpose versatile devices that are designed either for generic or specific functionalities.
- Microprocessor is the brain of computer which does all the work.
- Before we start a detailed study about the microprocessors, we need to know the differences between the following:
    - ➢ Microcomputer – A computer with a microprocessor as its CPU and includes memory, I/O
    - ➢ Microprocessor –A silicon chip which includes ALU, register circuits & control circuits
    - ➢ Microcontroller –Also a silicon chip which includes microprocessor, memory & I/O all in a single package.

1.2 Microprocessor

- The term microprocessor is an amalgamation of two words - micro and processor.
- Processor means a device that processes data or information but in this perspective it will process only binary data 0s and 1s.
- The word micro is the latest addition.
- In early 1960s, the processor was built using separate elements and in the 1970s all of the components that make up the processor are fabricated on a single chip i.e. silicon chip.
- So this reduces the size of the processor and increases the computation speed.
- So we can now say that Microprocessor is born.

Definition: *Microprocessor is a multipurpose register-based clock-driven device that takes binary data as input, processes it according to instructions stored in its memory, and provides binary results as output.*

- A microprocessor is the central processing unit of a computer system and is processes the distinctive set of instructions and programs
- The microprocessor is made up of several tiny components namely diodes, transistors and registers that work together.
- The microprocessor is a programmable device that takes in data and performs arithmetic or logical operations on them based on the instructions and program stored in memory and then produces processed results.
- We elaborate underlined words in depth.
- As a **programmable device,** the microprocessor performs different operations on the data based on the instruction given to carry out the task. The microprocessor manipulates it by changing the operation carried out by the instruction.
- The data the microprocessor **takes in** for manipulation comes from the input devices. These devices writes data from the outside world to the microprocessor.
- The microprocessor only understands binary data or numbers. A binary number is called a bit. The microprocessor identifies and manipulates these groups of bits together.

- Every microprocessor has basic **arithmetic operations** such as addition, subtraction, increment and decrement and **logic operations** such as NOT, OR, AND, EXOR, shifting (left or right).
- **Memory** is the location where instructions and programs are **stored**. Each location is capable of storing only one bit. So several bit locations are combined to create registers and several registers are combined to create a memory. Each location in memory is identified by address and capable of storing group of bits.
- After manipulation the microprocessor **produces** results and sends it to the output device. These devices display the results to the outside world reading it from the microprocessor.

1.2.1 A Programmable Machine

- A typical programmable machine has four parts that work collectively.
- Hardware is the physical part of the system.
- Set of instruction is the program that instructs the microprocessor to perform the required operation and set of programs makes the software.

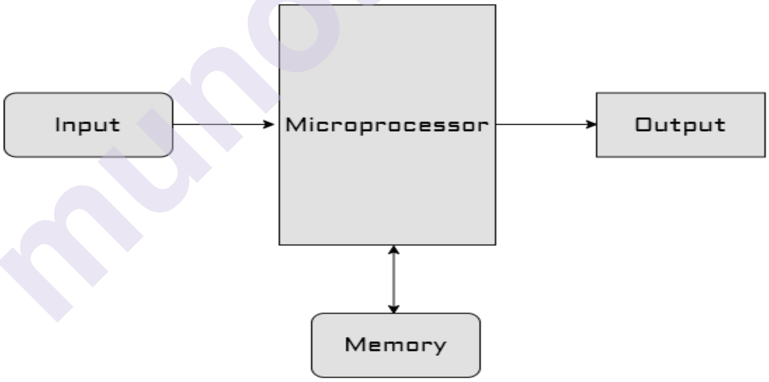
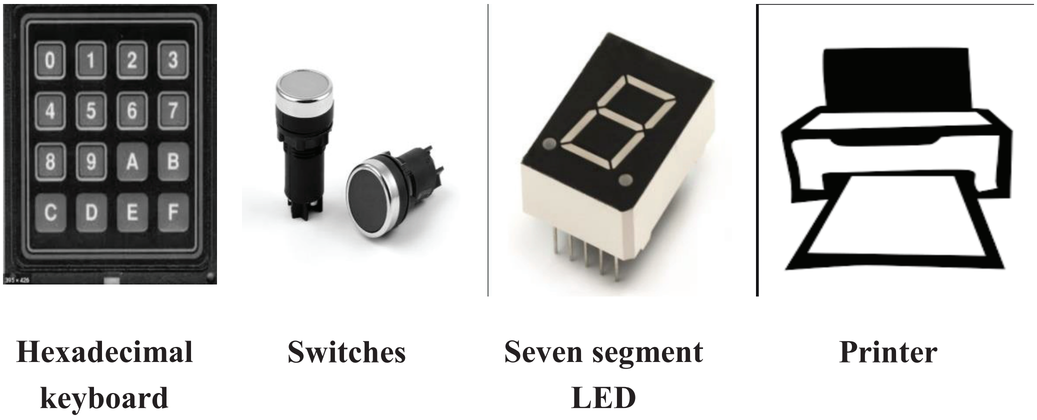


Figure 1.1 – A Programmable Machine

- The microprocessor is the central unit of a computer system that performs basic arithmetic and logic operation.
- A microprocessor works on the instructions stored in memory by accepting binary data as input followed by processing the data and then produces output.

- The microprocessor applications are classified into two categories
  - ➢ Reprogrammable system –The microprocessor computes and processes instructions given by the user or programmer.
  - ➢ Embedded system – The programming is already done and microprocessor is part of the specific system such as washing machine, fridge and air conditioner.
- The microprocessor processes on binary digits (called bits) i.e. 0 and 1 only.
- Several digits i.e. bits are combined to create words and this forms the basis of microprocessor classification based on word length.
- The memory stores instructions and data in group of bits.
- This information is given to the microprocessor whenever it is required..
- Memory is viewed as pages in notebook where each line represents register and is capable of storing a binary word.
- So for example each line is an 8 bit register that stores a 8 bit word and several registers are combined one below the another to create a memory cell.
- The combination of registers to create a memory cell is always in power of two.
- The user enters data into memory through input devices like keyboard and simple switch.
- The microprocessor reads the instruction from the memory and processes the data according to those instructions.
- The results are viewed on output device like seven segment LEDs and printer.



Hexadecimal keyboard **Switches** **Seven segment LED** **Printer**

Figure 1.2 – Input and Output Devices

1.2.2 Advances in Semiconductor Technology

- Semiconductor technology has undergone unprecedented changes from its inception in 1950s
- In early systems the semiconductor devices have replaced vacuum tubes which drastically reduced the size of the system.
- The invention of transistors further brought down the size of the system
- **SSI** (Small Scale Integration) containing upto 10 transistors were used in development of early systems.
- The number of transistors slowly increased to about 100 calling it **MSI** (Medium-Scale Integration).
- By 1970s tens of thousands of transistors were used calling it **LSI** (Large Scale Integration) system.
- Later on hundreds of thousands of transistors on a single chip brought in the **VLSI** (Very Large Scale Integration) era.
- Today we are in the era of **SLSI** (Super Large Scale Integration).

Historical Perspective:

- The world's first recognized microprocessor is Intel4004 a 4-bit microprocessor–programmable controller on a chip.
- There were just 45 instructions for 4004 and widely used in video games and small-scale microprocessor control system.
- The instructions executed at 50KIPs (kilo-instructions per second) and fabricated with P-Channel MOSFET (Metal Oxide Semiconductor Field Effect Transistor).
- In 1970 due to the increase in computational demand, Intel built the 8008 which is extended 8-bit version of the 4004 microprocessor.
- As people begin to use, Intel recognized the limitations and came up with the powerful 8 bit microprocessor 8080 in 1973.
- Several other manufacturers too entered the microprocessor market.
- Table 1.1 lists several of these early microprocessors and their manufacturers.

Table 1.1 – Early Microprocessor

Manufacturer	Part Number
Fairchild	F-8
Intel	8080
MOS Technology	6502
Motorola	MC6800
National Semiconductor	IMP-8
Rockwell International	PPS-8
Zilog	Z-8

- With fifty years since the invention of the 4004, Intel has made the processors that are designed with 15 million transistors that can address 1TB of memory and can operate at 400 Mhz to 1.5 GHz frequency.
- The following table summarizes the historical perspective of Intel microprocessors.

Table 2 – Intel Microprocessor Historical Perspective

Microprocessor	Year of Introduction	Data Bus	Address Bus
4004	1971	4	8
8008	1972	8	8
8080	1974	8	16
8085	1977	8	16
8086	1978	16	20
80186	1982	16	20
80286	1983	16	24
80386	1986	32	32
80486	1989	32	32
Pentium	1993 onwards	32	
Core Solo	2006	32	
Dual Core	2006	32	
Core 2 Duo	2006	32	
Quad Core	2008	32	
i3, i5, i7	2010	64	

1.2.3 Organization of A Microprocessor-Based System

- The organization of a microprocessor based system consist of three basic components: microprocessor, I/O and memory.
- These components are organized around a common communication path called a bus.

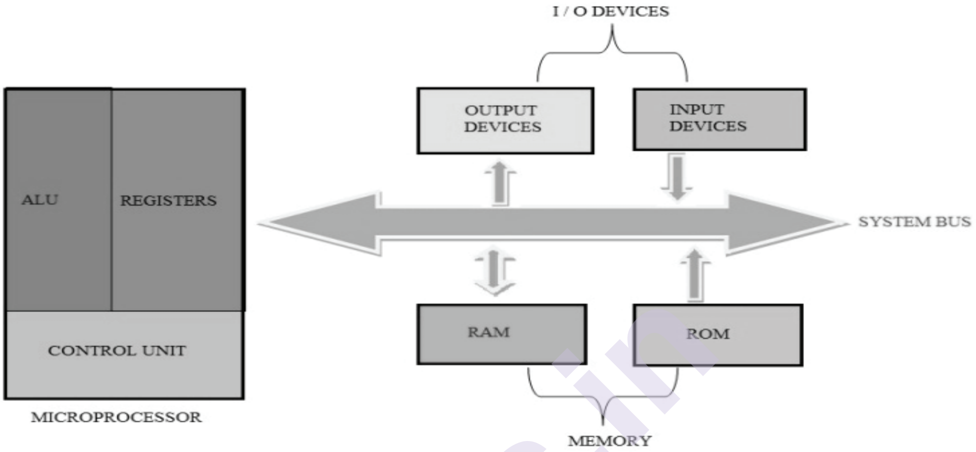


Figure 1.3 – Microprocessor-Based System with Bus Architecture

- **Microprocessor:** A typical microprocessor consists of three sub components namely arithmetic and logic unit (ALU), control unit and register array to process the instructions.
  - **Arithmetic and Logic Unit**: The ALU performs the arithmetic and logical operations such as Addition, Subtraction, AND, OR, XOR etc. The data is taken from memory, accumulator and registers to perform operation. The results of the operations in the microprocessor are usually stored in the accumulator or memory.
  - **Register Array:** The 8085 microprocessor includes six general-purpose registers, one accumulator and one flag register. It also has two 16-bit registers namely stack pointer and program counter.
  - **Control unit:** It provides timing and control signal to the microprocessor to perform operations such as read and write from memory and peripherals.
- **Memory:** It stores information in binary format. The user enters its instructions into the memory. The microprocessor reads these instructions stored in the memory in sequence, interprets and executes one by one and stores the final results in memory or sends it to output device. The microprocessor contains basically two types of memory.

  - ➢ **Read Only Memory (ROM) –** A nonvolatile memory that stores information that does not change.
  - ➢ **Random Access Memory (RAM) or Read/Write Memory –** A volatile memory that stores information supplied by the user such as programs and data.

- **I/O Devices:** I/O (Input/output) devices also referred as peripherals. The input device is a hexadecimal keyboard with some additional functional keys to perform. The output device is a 7 segment LED display which displays the processed results.
- **System bus:** It is a set of wires that establishes communication or connection between the microprocessor and peripherals to exchange data. There are three types of system bus
  - ➢ Address bus
  - ➢ Data bus
  - ➢ Control bus

1.2.4 How Does The Microprocessor Work?

- To execute a program, the microprocessor reads the instruction from the memory, interprets and decodes it, then executes it.
- The instructions are sequentially stored in the memory one after another.
- So the microprocessor fetches the first instruction from the memory, interprets and executes that instruction.
- The series is continued until each and every instructions are done.
- The microprocessor uses system bus to fetch the address, data and binary instructions to and from the memory.
- It uses registers to store data temporarily, performs the operations in ALU and sends the binary result to seven segment LEDs.

1.3 Microprocessor Instruction Set and Computer Languages

- Microprocessor processes binary words.
- However each microprocessor has its own binary words creating the instruction set and the interpretation of the words which is designed based on the microprocessor.

- The word length (expressed in bytes – 8 bits) is defined as the number of bits the microprocessor recognizes and processes in a given time
- To communicate with the computer one must give instruction in binary language or **machine language**.
- It is difficult for most people to write programs in binary format.
- So programmers write program in **assembly language** which contain English like word to represent the binary instruction of a machine.
- But assembly programs are specific to given machine and cannot be transferred from one machine to another.
- So **high level languages** are used which contains English like statements and are machine independent.

1.3.1 Machine Language

- The number of bits in a word for a given language is fixed and the words are formed through various combinations of these bits.
- If a machine language has 'n' bits then the language has 2^n words.
- For example a machine with a word length of eight bits can have 256 (2^8) combinations of eight bits – thus a language have 256 words.
- The design engineer selects only certain combination of bit pattern and not all and then gives a specific interpretation to each combination known as instruction.

1.3.1.1 8085 MACHINE LANGUAGE

- The Intel 8085 is an 8-bit microprocessor with 246 identifiable patterns forming 74 instructions.
- It is difficult to enter the instructions in binary and hence entered in hexadecimal code format.
- For example, the combination 0000 1100 (Hex Code – 0C) is interpreted as Increment the value of Register C by 1.
- It is tedious and error-inducive for people to write instructions in binary language so writing in hexadecimal makes it less error-prone.

1.3.2 Assembly Language

- Entering the instructions using hexadecimal is quite easier than entering the binary combinations.
- But just reading a set of hexadecimal code makes it difficult for the user to interpret the meaning of the program.
- So the designers of the microprocessor give symbolic name for each hexadecimal code.
- This code known as mnemonics are short English words which are machine dependent.
- The assembly language program written for one type of microprocessor is not transferable to a computer with another microprocessor unless the machine codes are compatible with each other.

1.3.2.1 8085 ASSEMBLY LANGUAGE

- The complete set of 8085 mnemonics is called the 8085 assembly language program.
- Using the same example from before, 0000 1100 is 0C in hex and the mnemonic is INR C.
- It is important to remember that a machine language and its associated assembly language are completely machine dependent.
- For example, an 8-bit microprocessor for Motorola i.e. 6800 and Intel i.e. 8085s instruction sets are different from each other.
- So a program written for the 6800 microprocessor cannot be executed on 8085 microprocessor and vice versa.

1.3.2.2 Writing and Executing an Assembly Language Program

- There are two ways to accomplish this task.
- The first procedure is called either **manual** or **hand assembly.** The steps are
  - ➢ From the instruction set given by the manufacturer, write the instructions in assembly language.
  - ➢ Find the corresponding hexadecimal code for each instruction.
  - ➢ Enter the hex code in the memory step by step in the kit using the hexadecimal keyboard.
  - ➢ Press the Exec key to execute the code and check for presence of errors.

  - Correct errors if any and view the result in seven segment LED Display.

- Another technique is the use of **assembler tool.**
  - The assembler is a tool that translates the mnemonics or hex code into the corresponding binary machine codes of the microprocessor.
  - Each microprocessor uses its own assembler because the mnemonics and machine codes are specific to the microprocessor, and each assembler has its own set of conversion rules that must be followed by the user.

1.3.3 High-Level Language

- They are programming language which are machine-independent
- They have English like statements with proper syntax and semantic.
- The machine does not understand high level language and so tools like compiler and interpreter convert them into machine language for processing.
- Example BASIC, C, C++ and Java

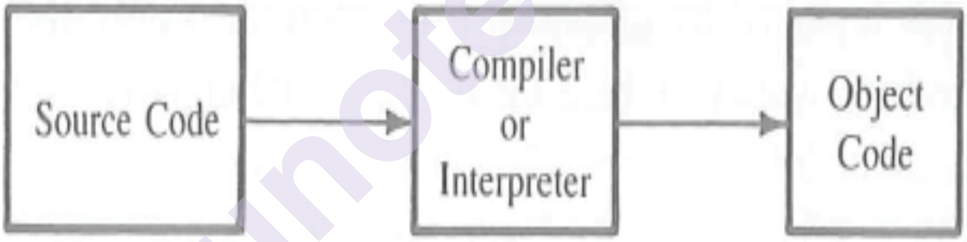


Figure 1.4 – Translation of High-level Language Program into Machine Code

1.3.4 Operating Systems

- The operating system controls the overall operation and manages the interaction between the computer hardware and software.
- It is basically a collection of programs.

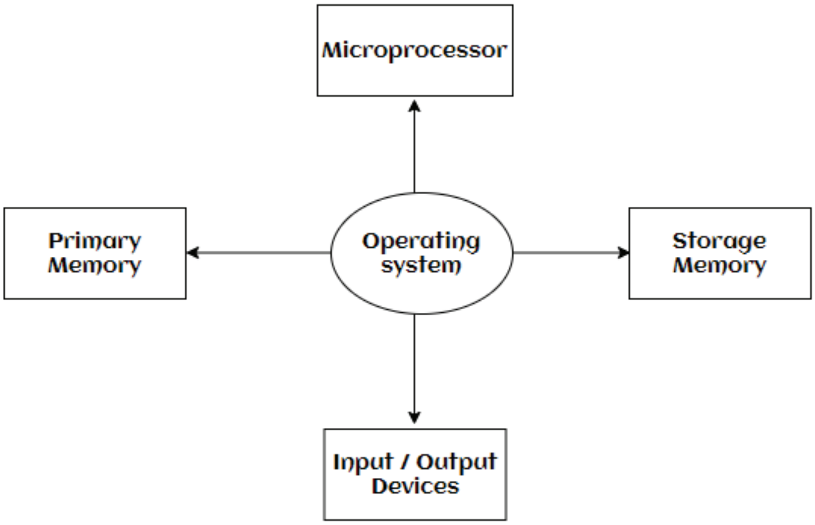


Figure 1.5 – Functional Relationship of Operating System with System Hardware

- The operating system helps in communication between the memory and peripherals and stores the information on the disk.
- The operating system boots when the system is switched on identifying the hardware and handles the application programs running in the background.

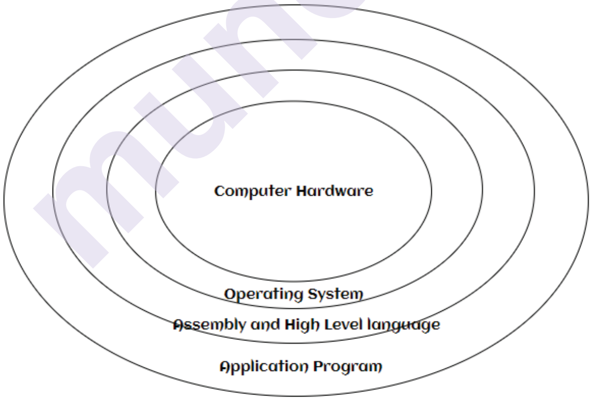


Figure 1.6 – Hierarchical relationship of operating system with hardware & software

- Several operating systems have evolved over the years from control monitor program to graphical user interface operating system.
- Example MS-DOS, Unix, Linux, Os/2, Windows 95/98/2000/ME/7/7/1/10

1.4 From Large Computers to Single-Chip Microcontrollers

- Depending on the needs of the user, computers are classified and designed for different purpose.
- Initial classification of computers was mainframe, mini computers and microcomputers.
- With changes in technology and microprocessor being part of every computer the new classification includes large computer, medium size computer and microcomputers.

1.4.1 Large Computers

- General purpose multitasking multi user computers are called large computers.
- They are capable of solving complex and scientific calculations and handles huge volumes of data and handles hundreds of user.
- Based on size they are classified into
  - ➢ **Mainframe** – High speed computers to handle large count of users. Example IBM System/390 series.
  - ➢ **Supercomputer** – High speed and high performance computer used in research. Example Cray-2 and Y-MP.

Mainframe

Supercomputer

Figure 1.7 - Large Computers

1.4.2 Medium-Size Computers

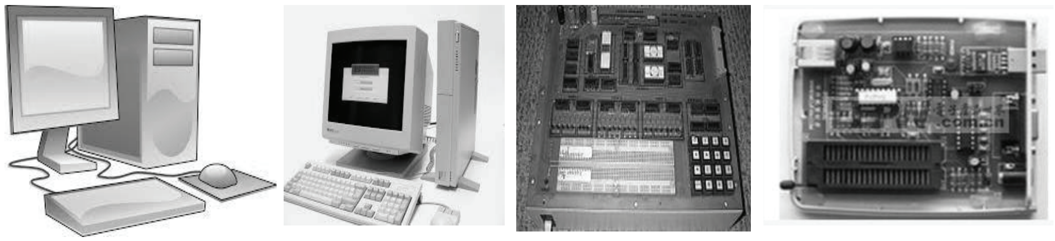
- To meet the needs of small factories and data processing tasks the medium-size or mini computers are introduced.

- Multi processing system capable of supporting hundreds of users but technology is different as these machines are slow and memory size is less than mainframe.

Figure 1.8 - Mini Computer

1.4.3 Microcomputers

- It contains microprocessor as its central processor along with memory and minimal input/output devices
- Most powerful in 1970s and in 1980s with advent of increasingly powerful microprocessors more widely used.
- They are classified into four categories –
  - **Personal computer (PC)** – A small computer designed for single user and is relatively inexpensive. The entire processor is on single chip and used in word processing, accounting, personal finance, desktop publishing, accessing resources on the Internet.
  - **Workstations** - It is similar to PC but a powerful processor that suites for engineering and scientific applications, software development and specific applications seeking average computing power and high-end graphics.
  - **Single-board microcomputer** – It is used in small industries and college labs to understand, evaluate and test the performance of specific microprocessor. All basic components such as memory, keyboard, LED are there with the 8 or 16 bit microprocessor.
  - **Single-chip microcomputers** – Includes a microprocessor, R/W Memory, Read Only Memory and I/O Devices on single chip and hence commonly referred as microcontrollers.



Personal Computer **Workstation** **Single-board microcomputer** **Single-chip microcomputer**

Figure 1.9 – Types of micro computers

1.5 Application – Microprocessor Controlled Temperature System (MCTS)

- We are building a microprocessor controlled temperature system that is capable of reading the temperature in a room and display the recorded temperature in LCD (Liquid Crystal Display) and based on the value turn on the fan if the temperature is high or turn on the heater if temperature is low based on the set point value

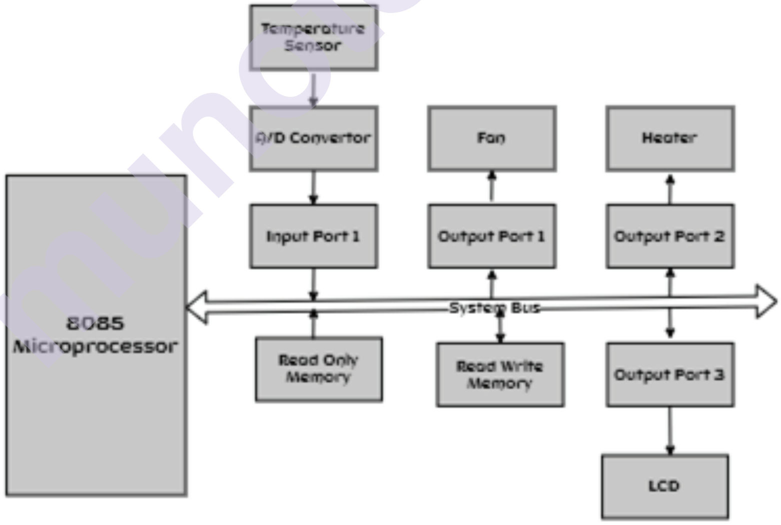


Figure 1.10 – MCTS

1.5.1 System Hardware

- The hardware part of the application comprises of the following components
    - ➢ Microprocessor – The job of the processor is control the overall communication. It reads instructions from memory, interprets and

executes them. It reads temperature from LCD and turn on/off fan or heater depending on the set point value

- Memory – The ROM provides instructions to monitor the system and R/W memory temporary stores the temperature value.
- Input Device – The temperature sensor is the main input device which measures the temperature in signals and for the microprocessor to process the signal A/D (Analog to Digital) Convertor is used. Devices cannot be directly connected to processor and therefore connected via input ports.
- Output Device – LCD panel, fan and heater forms the output device of this application. Just as input device cannot be directly connected, the output devices are also connected via ports.

1.5.2 System Software

- The system is initially reset and microprocessor reads instructions from the memory one by one, decodes it and then executes it.

1.5 Summary

- The microprocessor is an important component of a digital computer.
- The microprocessor along with memory and I/O devices carry out various functionalities.
- The 8085 is a powerful microprocessor.
- Different computer languages are available but the processor understands the machine language and users program in assembly or high level language
- Various classification of systems are available today but microprocessor is an integral part of these systems
- In designing a microprocessor based system, the hardware and software part have to be designed concurrently because of the interdependency.

List of References

- Ramesh Gaonkar "Microprocessor Architecture, Programming and Applications with the 8085", Fifth Edition, Penram International Publishing (I) Private Limited

- B. Ram, “Fundamentals of Microprocessor and Microcomputers”,. Sixth Revised and Enlarged Edition, Dhanpat Rai Publications (P) Limited

- Barry B. Brey, “The Intel Microprocessors - Architecture, Programming, and Interfacing”, Eight Edition, Pearson Prentice Hall

- https://www.tutorialspoint.com/microprocessor/microprocessor_overview.htm

Unit End Exercise

1. What are the essential components of a microprocessor based system? Draw the block diagram and explain the function of each component.
2. What is a microprocessor? What is the difference between a microprocessor and CPU?
3. Explain the various I/O devices used along with a microprocessor.
4. Explain the types of computer languages. What are the advantages of an assembly language over the high level language?
5. What is an operating system? Explain its functional and hierarchical relationship with various hardware components.
6. Differentiate between personal computer, workstation, single-board and single-chip microcomputers.

❖❖❖

UNIT 1

2

MICROPROCESSOR ARCHITECTURE AND MICROCOMPUTER SYSTEMS

Unit Structure

2.0 Objectives

2.1 Introduction

2.2 Microprocessor Architecture and its Operation

2.2.1 Microprocessor-Initiated Operations

2.2.2 Internal Operations

2.2.3 Peripheral Operations

2.3 Memory

2.3.1 Flip-flop or Latch as Storage Element

2.3.2 Memory Map and Address

2.3.3 Word Size of Memory

2.3.4 Instruction Fetch from Memory

2.3.5 Classification of Memory

2.4 I/O Devices

2.4.1 Peripheral-mapped I/O

2.4.2 Memory-mapped I/O

2.5 Microcomputer System Illustration

2.6 Logic Devices

2.6.1 Tri-state Device

2.6.2 Decoder

2.6.3 Encoder

2.6.4 D Flip-flop

2.7 Application: MCTS

2.8 Summary

2.9 List of references

2.10 Unit End Exercise

2.0 Objectives

After going through this chapter, you will be able to

- Understand and identify the operations performed by the microprocessor
- Recognize the memory organization
- Learn to create the memory map and address range
- Understand the types of memory available
- Identify the ways to communicate with I/O Devices
- Know the need of logic devices for interfacing

2.1 Introduction

- The microprocessor is a programmable digital device designed with flip-flop, registers, buffers and several other tiny element all which is integrated on a single chip.
- To establish communication between these elements, each microprocessor has its own instruction set.
- The instruction set of the microprocessor is designed to help perform data manipulation and communication.
- The architectural and logic design of the microprocessor helps in executing these instructions to obtain the desired result.

2.2 Microprocessor Architecture and its Operation

- User can write programs using the instruction set of the microprocessor.
- We can program the microprocessor to do variety of functions.
- So the above functions are categorized into three categories namely Microprocessor initiated operations, Internal operations, Peripheral operations

2.2.1 Microprocessor-Initiated Operations

- They represent the basic communication between the microprocessor with memory and I/O devices and they make up four such operations namely

- Memory Read ~ It indicates reading data or instruction from the memory
- Memory Write ~ It indicates writing data or instruction to the memory
- I/O Read ~ It indicates reading data from the input device
- I/O Write ~ It indicates writing result to the output device

- To perform any of these operation the microprocessor executes the following step
  - **Step 1: Identify location of memory register or I/O device** – This is achieved with the help of the address bus which identifies the location. The 8085 microprocessor has 16 bit unidirectional address bus to perform this step.
  - **Step 2: Transfer data or instruction** – This can be achieved with the help of the data bus. The 8085 microprocessor has 8 bit data bus to carry information from/to memory or I/O device.
  - **Step 3: Provide timing and control signal** – This can be achieved with the help of the control bus. The 8085 microprocessor provides four control signals depending on the nature of operation to be performed.

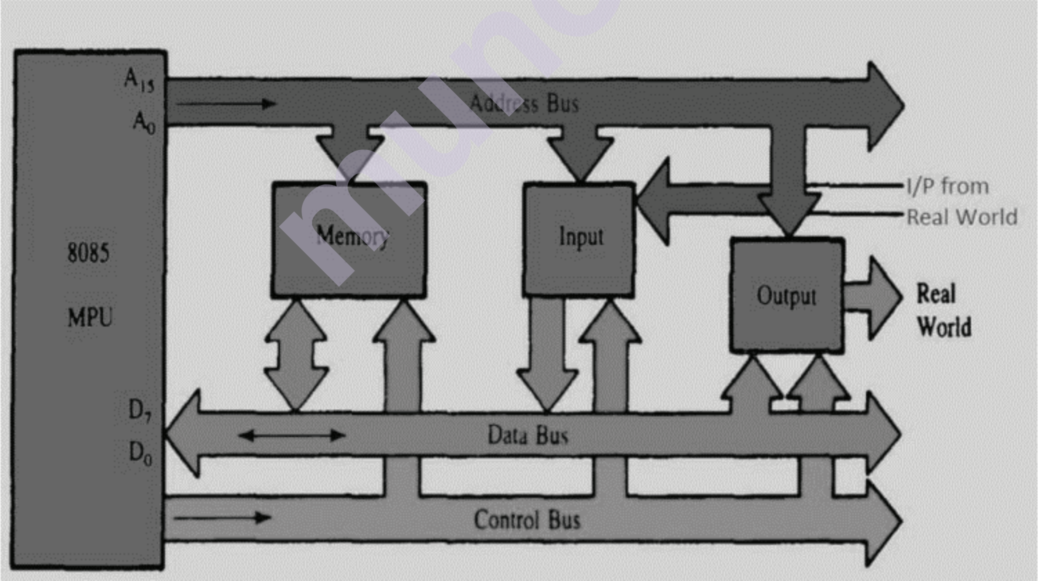
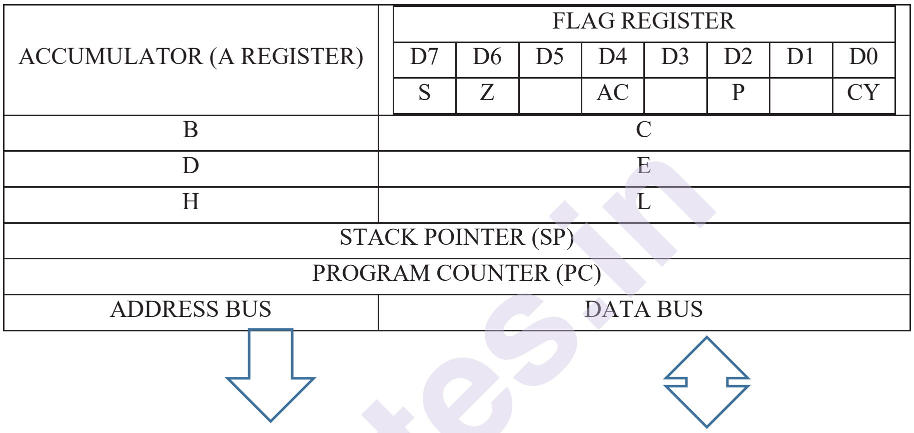


Figure 2.1 – Microprocessor and bus structure

2.2.2 Internal Operations

- The internal architecture of the microprocessor unit (MPU) is solely responsible for the various internal operations.
- The 8085 MPU has ALU, internal bus, general-registers, accumulators, program counter and stack pointer to serve the purpose.
- The operation include storing data in registers, performing arithmetic and logical calculations, checking conditions, executing the instructions in sequence and using stack for temporarily processing the data.



<table>
<tr><td rowspan="3">ACCUMULATOR (A REGISTER)</td><td colspan="8">FLAG REGISTER</td></tr>
<tr><td>D7</td><td>D6</td><td>D5</td><td>D4</td><td>D3</td><td>D2</td><td>D1</td><td>D0</td></tr>
<tr><td>S</td><td>Z</td><td></td><td>AC</td><td></td><td>P</td><td></td><td>CY</td></tr>
<tr><td>B</td><td colspan="8">C</td></tr>
<tr><td>D</td><td colspan="8">E</td></tr>
<tr><td>H</td><td colspan="8">L</td></tr>
<tr><td colspan="9">STACK POINTER (SP)</td></tr>
<tr><td colspan="9">PROGRAM COUNTER (PC)</td></tr>
<tr><td>ADDRESS BUS</td><td colspan="8">DATA BUS</td></tr>
</table>

Figure 2.2 – Microprocessor internal structure

- For example, consider the following assembly program

```
2000    0E    MVI C, 65H
2001    65
2002    0C    INR C
2003    79    MOV A, C
2004    76    HLT
```

- The first instruction is written at address 2000H and this value is loaded in program counter and execution begins.
- The processor decodes and executes the instruction at 2000H, it increments the value in program counter 2001H and fetches the next code and concludes by interpreting that value 65H be written in C register.
- Then program counter is incremented to next address which is 2002H and the instructions increments the value in C register.
- The next address in program counter will be 2003H which is decoded as content of C register is copied to accumulator

- The last address in program counter is 2004H which indicates end od execution
- So for basic internal operations the various registers of the microprocessor are used.

2.2.3 Peripheral Operations

- The peripherals or I/O devices are the external devices that carry out this operation and hence this category is commonly called as externally-initiated operation.
- Certain individual pins of the microprocessor perform this type of operation
- They include reset, interrupt, ready and hold pin.

2.3 Memory

- The integral and important component of the microcomputer is the storage device i.e. the memory.
- The microprocessor reads data and programs (here instructions) from the memory and stores results to the memory.
- The microcomputer system has two types of memory – Read Only Memory and Read/Write Memory
- The Read Only Memory is non-volatile that stores system level programs available to system all time.
- The Read/Write memory holds program and data which can be read and written by the programmer and are volatile in nature.
- To read/write information from any kind of memory the microprocessor performs the following steps
  - ➢ Select the right memory chip
  - ➢ Identify the exact memory location in the selected chip
  - ➢ Access the data available at the identified location

2.3.1 Flip-Flop or Latch as Storage Element

- Memory is defined as a circuit that stores voltage level or electric charge recorded in terms on binary digits 0 and 1.

- So a basic feature of memory that stores the binary bits is the flip-flop or latch.
- D latch or flip-flop is the ideal choice for a memory element.
- The latch includes one input line, one output line, and an enable input that allows the latch to be triggered.
- For securing and controlling the data in and out of latch, we use tristate buffers at input and output line.

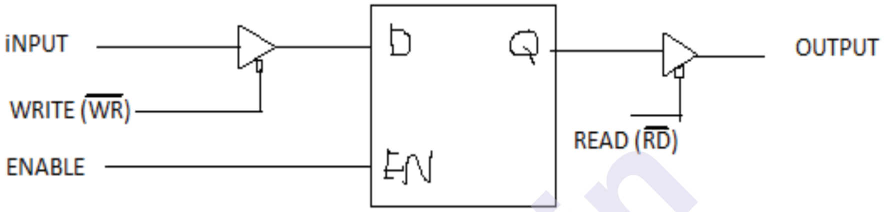


Figure 2.3 – Basic D Latch with Tristate Buffers

- The input buffer in controlled by the active low control signal write ($\overline{WR}$) and the output buffer is controlled by the active low control signal read ($\overline{RD}$)
- This latch is called as a memory cell capable of storing a binary bit or digit.
- In order to create a memory register, several such latches needs to be combined together.
- For example four latches have been connected to create a memory register identified as 1 x 4 where 1 represents count of memory register and 4 represents the number of bits the individual register can hold.

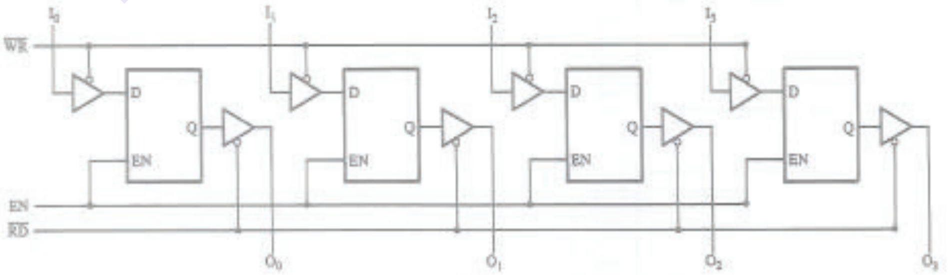


Figure 2.4 – 1 x 4 memory organization

- So a simplified block diagram to represent the above construction.

- The Enable signal (EN) enables the register and the write or read operation can be performed in the register by enabling the input or output buffer respectively with the control signal.

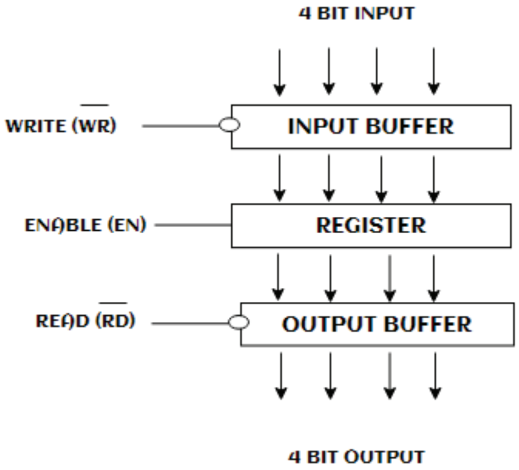


Figure 2.5 – 1 x 4 memory register block diagram

- We can now expand the diagram by adding several memory registers to create a memory chip.
- Using interfacing logic with help of decoder, we can select individual registers and perform write or read operation by enabling the input or output buffer respectively.
- Let's construct as 4 x 4 memory i.e. 4 registers each capable of storing a 4 bit word.
- When count of registers increases then the enable signal is replaced by address lines for register selection.
- Two address line A_1-A_0 are connected to decoder for selecting the register.
- Four combinations (00, 01, 10 and 11) will select the register (0, 1, 2 and 3).

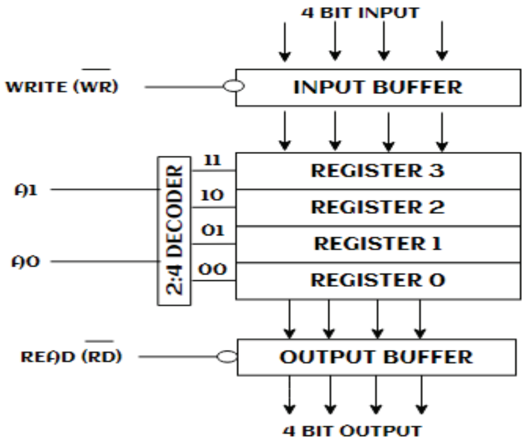


Figure 2.6 – 4 x 4 memory register block diagram

- We shall further expand with eight registers on one chip each capable of storing 8 bits/1 byte of data.
- For this we will require three address lines $A_2 – A_0$ which provides eight combinations (000, 001, 010, 011, 100, 101, 110, 111) for register select.

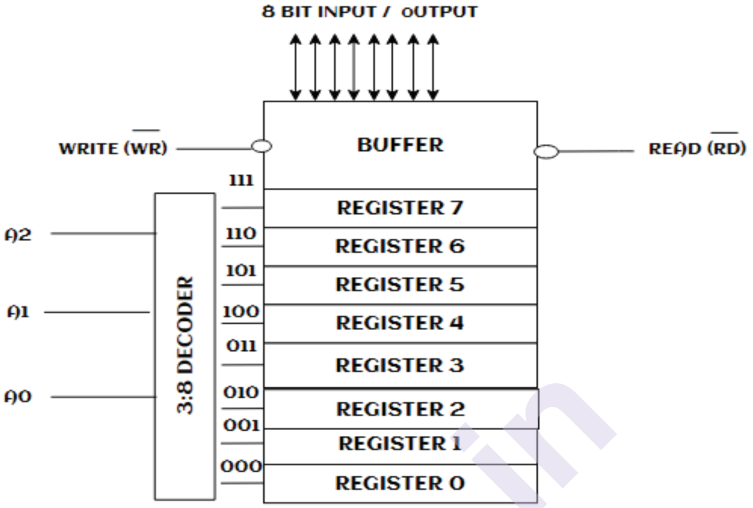


Figure 2.7 – 8 x 8 memory register block diagram

- And if we have 16 registers, we need four address lines and this calculation goes on.

No. of address lines	Size of memory
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024 ≈ 1K
11	2048 ≈ 2K
12	4096 ≈ 4K
13	8192 ≈ 8K
14	16384 ≈ 16K
15	32768 ≈ 32K
16	65536 ≈ 64K

Table 2.1 – Memory size and address line relationship

- We can also build memory on several chips.
- For example the above 8x8 memory organization can also be arranged on two chips with each chip having 4 registers each.
- Same three address lines will be used but $A_1 - A_0$ will be used to select the individual registers and A_2 will be used to select the memory chip.

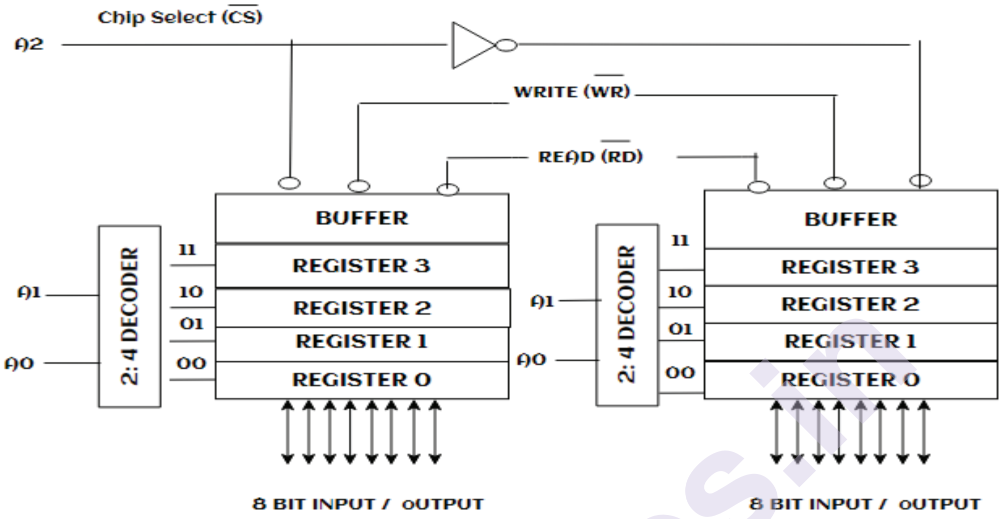


Figure 2.8 – 8 x 8 memory register with chip select

- As the number of registers increase we use a chip select ($\overline{CS}$) logic which is active low signal and acts as master enable pin to decide to perform read or write operation.
- So now we construct a typical read/write and read only memory chip.

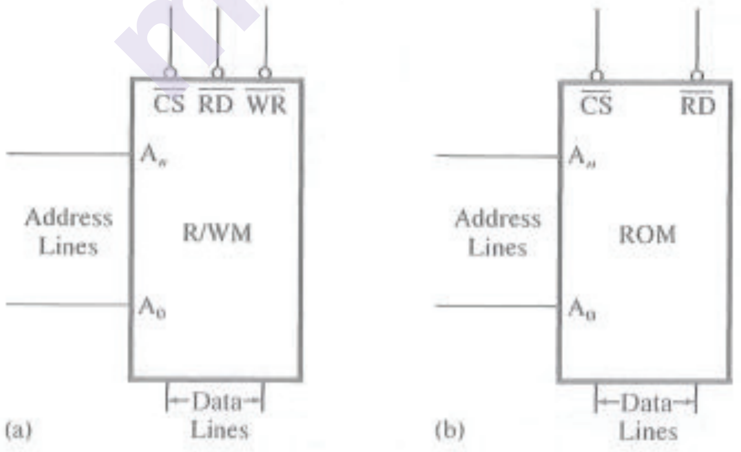


Figure 2.9 – R/W and ROM model

- The difference between the models is concerned with address decoding and most notable difference is that the ROM does not include $\overline{WR}$ signal.

2.3.2 Memory Map and Addresses

- The 8085 microprocessor has 16 address wires or lines and therefore can identify a maximum of 2^{16} = 655536 registers.
- The entire memory address range can be represented as

Table 2.2 – Memory map

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

- The above table is called the memory map which is the pictorial representation of the address range.
- The address range is 0000H to FFFFH.
- We shall now see a memory organization of 256 registers with 8 data lines i.e. this memory called as 256 x 8 memory organization.
- 256 = 2^8. So 8 address lines (A_7—A_0) are used for selecting memory register and remaining 8 address lines (A_{15}—A_8) are used for chip select ($\overline{CS}$) logic and two control signals Write ($\overline{WR}$) and Read ($\overline{RD}$) for data storage and retrieval respectively.
- For register select the address line are connected to internal decoder and for chip select the address line are connected to NAND gate via invertors.
- So when the NAND gate output is low the memory chip is selected.

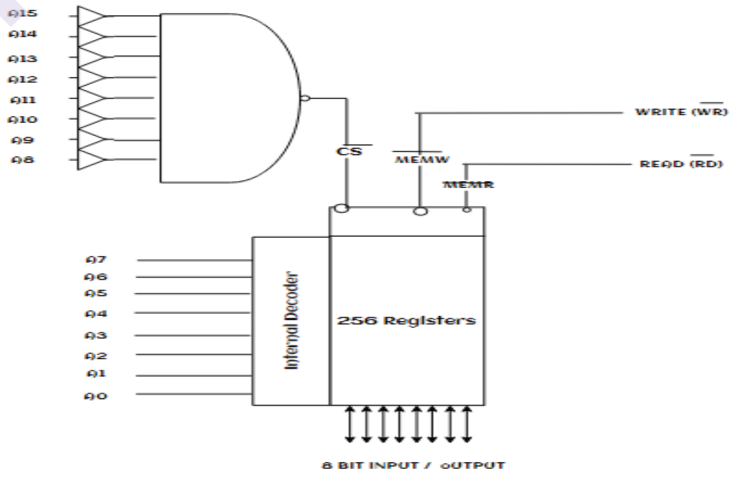


Figure 2.10 – 256 x 8 memory model

- The memory address range for 256 x 8 memory is represented as

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Chip Select Register Select

Table 2.3 – Memory map

- The address range is 0000H to 00FFH.
- We shall now see a memory organization of 8192 registers with 8 data lines i.e. this is referred as a memory size of 8192 x 8.
- 8192 = 2^{13}. So 8 address lines (A_{12}—A_0) are used for selecting memory register and remaining 3 address lines (A_{15}—A_{13}) are used for chip select ($\overline{CS}$) logic and two control signals Write ($\overline{WR}$) and Read ($\overline{RD}$) for data storage and retrieval respectively with same logic applied for interfacing.

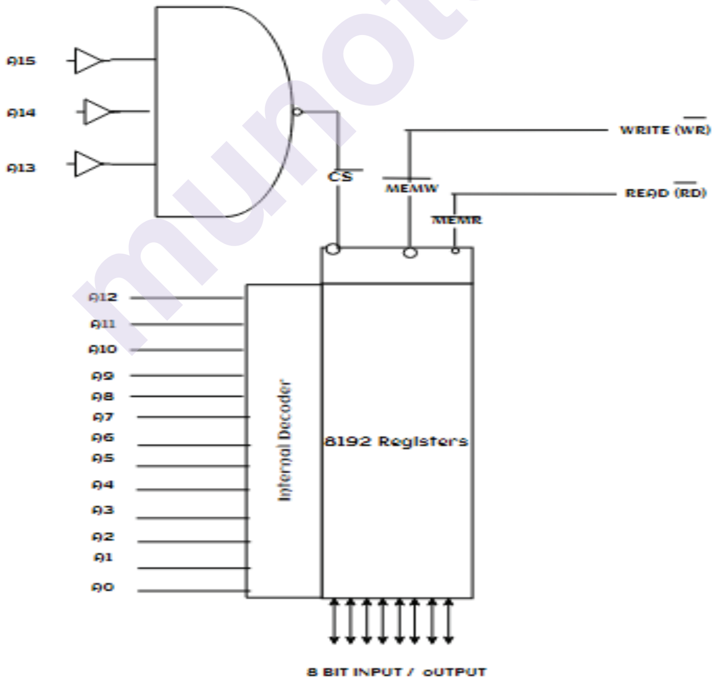


Figure 2.11 – 8192 x 8 memory model

- The memory address range for 8192 x 8 memory is represented as

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Chip Select Register Select

Table 2.4 – Memory map

- The address range is 0000H to 1FFFH.
- We can construct any memory based on above logic and calculate the address range on the memory chip.

2.3.3 Word Size of Memory

- Memory device come in different word size measured in bytes and memory chip comes in number of bits it stores.
- For example, a memory chip of size 512 x 4 means 512 registers individually which stores 4 bits and total memory chip size calculated as 512 x 4 = 2048 bits.

2.3.4 Instruction Fetch from Memory

- The purpose of memory is to store data and instructions, so MPU issues command to access it.
- This is done by sending the address of the specific memory register via the address bus and enable the data flow depending on the control signal issued.
- For example the memory address 2000H contains the instruction MOV E, A

Memory	Address	Hex Code	Binary Code	Mnemonic
2000	5F	0101	1111	MOV E, A

- The sequence of steps are
  - ➢ The instruction is available at address 2000H and this value in loaded in program counter.
  - ➢ Now the control unit issues the active low memory read $(\overline{MEMR})$ control signal to activate the output buffer.

- ➢ The code 5F(0101 1111) for the instruction MOV E, A available at location 200H is now placed in data bus

- ➢ The code is now loaded in instruction register and is decoded where the content of accumulator is copied into E register.

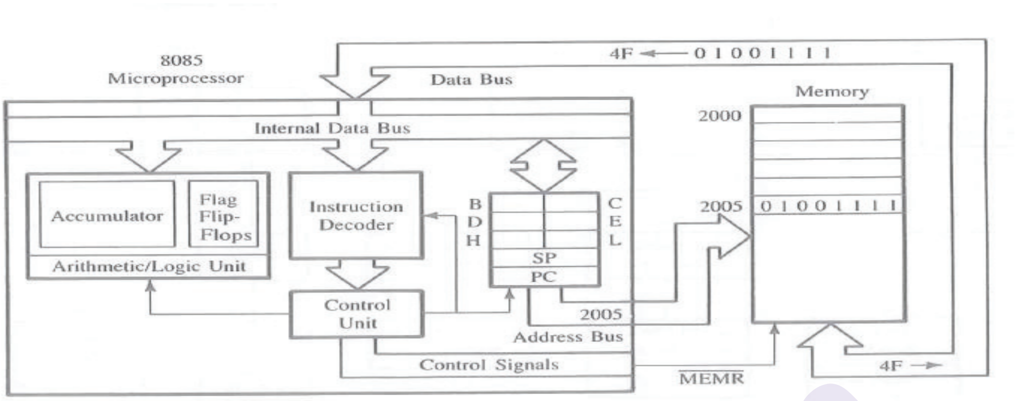
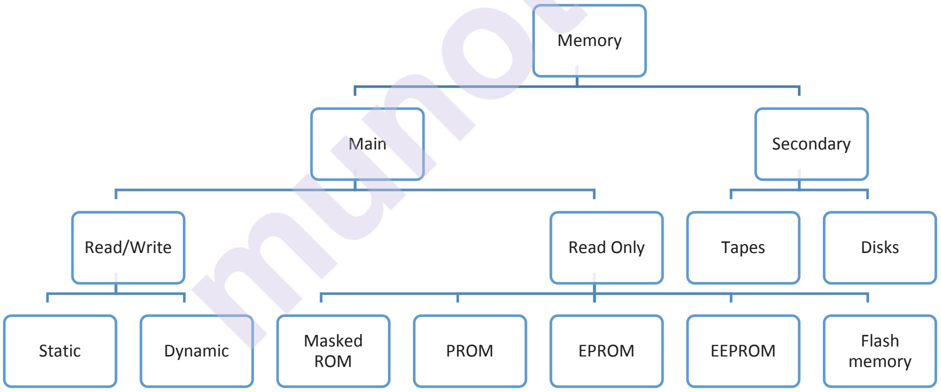


Figure 2.12 – Instruction Fetch Operation

2.3.5 Classification of Memory

- Memory is integral part of microcomputer system and are primarily classified into primary and secondary memory which further have several sub types.



- Main or primary memory contains data and instructions which computer is currently using.

  - ➢ The Read/Write or Random Access Memory (RAM) is volatile memory which stores data, instructions and immediate output.

    - ▪ Static RAM is built of flip-flops and holds data without external refresh and is expensive but high speed memory chip.

    - ▪ Dynamic RAM is built of MOS transistor gates which must be refreshed many times with low power consumption, high density and cheaper than static RAM.

- The Read Only Memory (ROM) is non-volatile memory which stores system level programs that PC want at all times.
  - Masked ROM are hard-wired and bit masking is done by metallization process usually by the manufacturers.
  - Programmable ROM (PROM) is diode-based and programmed using special devices that burn the fuses for storage of the appropriate bit pattern.
  - Erasable PROM (EPROM) can be programmed as it stores bits by charging and erasing the floating gate. Erasing is done using UV rays and hence entire memory content will be erased and can be reprogrammed again.
  - Electrically Erasable PROM (EEPROM) – Similar to EPROM with difference that erasing is done under software control at register level instead of entire chip.
  - Flash Memory are advanced level with erasing done at sector level or entirely and can be reprogrammed million times.
- Secondary memory or storage memory store information and results after execution and are non-volatile and stored for future reference. Microprocessor cannot execute programs from here and hence have to be loaded in main memory for execution.
  - The magnetic tapes are serial access device with high capacity but slow access.
  - The disks are semi-random devices made of metal and plastic scanned by a laser beam.

2.4 I/O DEVICES

- Input/ Output (I/O) devices or peripherals help the microprocessor communicate with the external world.
- Through input devices such as keyboard the system accepts input and the results are displayed via the output devices such as display and printers
- Communication with the I/O device can be established using either an 8-bit address (Peripheral- Mapped I/O) or 16-bit address (Memory-Mapped I/O).
- The steps of communication between microprocessor and peripherals are
  - Identify the I/O device by placing the address on the address bus.
  - Depending on operation to be performed the MPU issues read ($\overline{IOR}$) and write ($\overline{IOW}$) control signal on the control bus.
  - Data is then transferred to/from data bus depending on operation.

2.4.1 Peripheral-Mapped I/O

- It is known as I/O Mapped I/O.
- Eight address lines are used for I/O interface recognition.
- $2^8 = 256$. Thus, 256 input and output devices with a range of 00H to FFH can be identified.
- To access or read from input device active low I/O Read control signal ($\overline{IOR}$) is generated and to access or write to output device active low I/O Write control signal ($\overline{IOW}$) is generated by the MPU.

2.4.2 Memory-Mapped I/O

- The I/O devices are assumed as memory registers and therefore 16 address lines are used for identification.
- The MPU uses the control signal active low Memory Read ($\overline{MEMR}$) and active low Memory Write ($\overline{MEMW}$) to access the memory.

2.5 Microcomputer System Illustration

- We now develop a microcomputer system based on previous study.
- The system includes 8085 MPU, memory (EEPROM and R/W Memory), input and output devices and all interconnected using the various system bus.
- The MPU communicates with only device at a time by issuing the corresponding control signal.
- Address of memory location is identified by A_{15} –A_0 and I/O device are identified by A_7- A_0 using the address bus
- Data bus D_7-D_0 is common for all and bidirectional

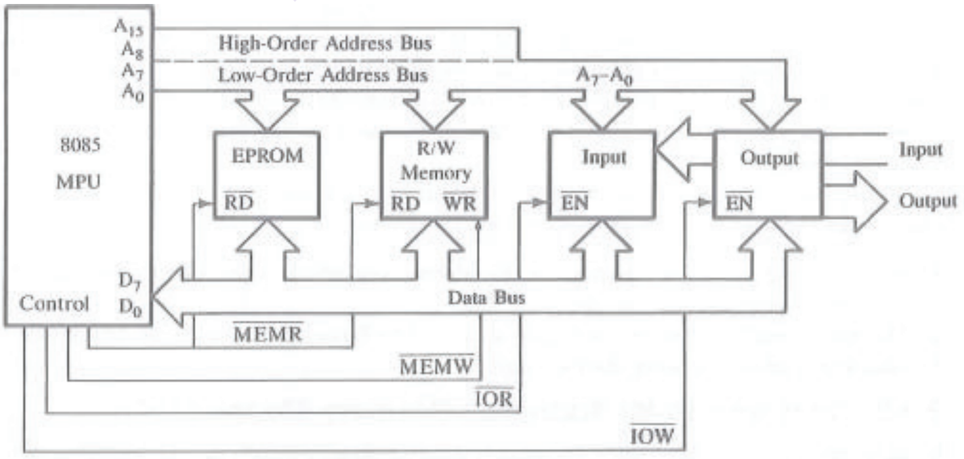


Figure 2.13 – Microcomputer System Illustration

2.6 Logic Devices

- Several types of interfacing devices are necessary to interface the microprocessor with memory and peripherals to establish the communication using the buses.

2.6.1 Tri-State Device

- Normal logic devices have two states – Logic 0 and Logic 1
- As name indicate these devices have a three output states – Logic 0, Logic 1 and high impedance state
- This state is to effectively remove the device influence from rest of the circuit.
- Tristate devices have enable line to confirm whether device works in normal state or high impedance state
- Tristate buffer and tristate invertor are commonly used interfacing devices.
- When enable is high these device act as normal device and when enable line is low, the device enters into high impedance state.

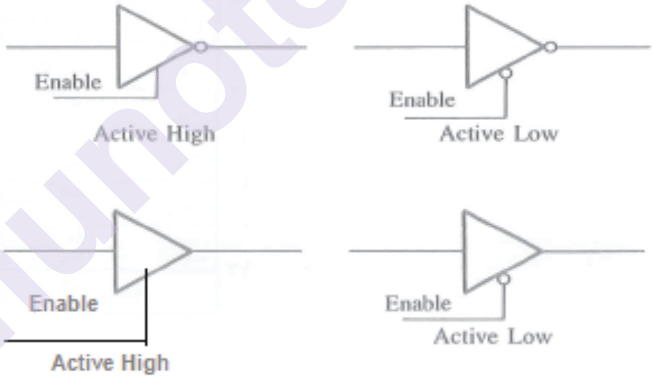


Figure 2.14 – Tristate Devices

2.6.2 Decoder

- Decoder is a logic circuit that converts binary information from n input to 2^n output.
- So in high performance memory system decoders can minimize the effect of memory selection.
- Each combination of input lines helps to select a single unique memory register.
- Example 2:4 Decoder, 3:8 Decoder and so on.

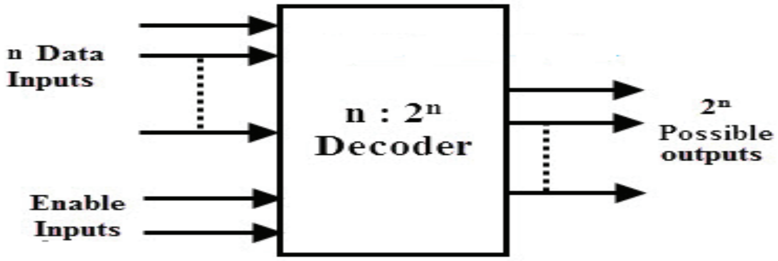


Figure 2.15 – Decoder

2.6.3 Encoder

- Encoder is opposite of decoder where it provides output for each input signal
- Single input line produces a corresponding output, but however if two or more input lines gets activated then appropriate output code cannot be generated
- They are commonly used device for interfacing with keyboards.

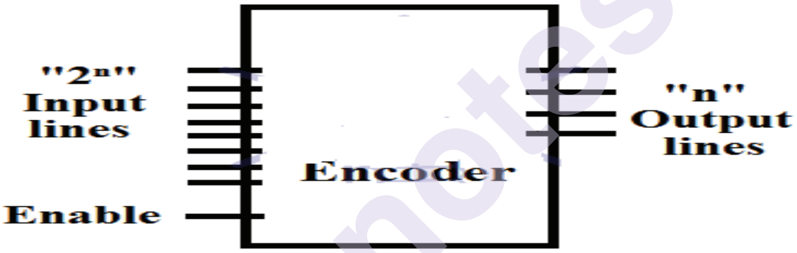


Figure 2.16 – Encoder

2.6.4 D Flip-Flop

- It is most essential device while interfacing the output devices
- The MPU holds data in the data bus only for few microseconds and it is very important to latch the data before it is lost
- So D latch or flip-flop serves this purpose.

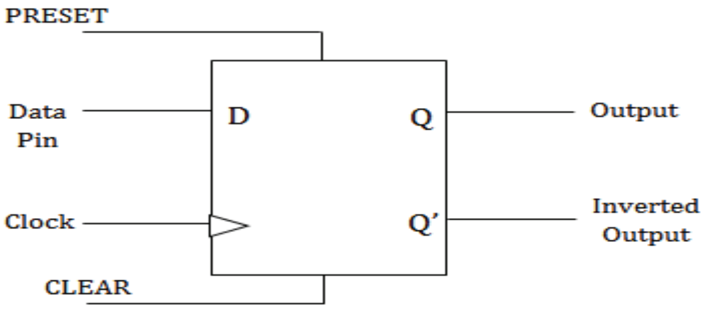


Figure 2.17 – D FF

2.7 Application: MCTS

- The application discussed in the last chapter is now expanded with interfacing devices included.
- Decoder is required to interface each device
- The input device in this case temperature sensor are interfaced using buffer
- The output devices in this case fan, heater and LCD are interfaced using the latch

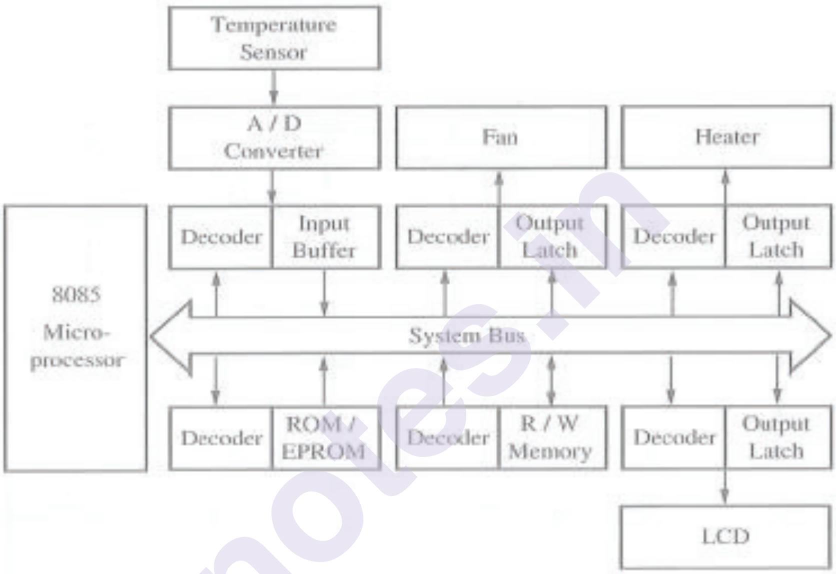


Figure 2.18 – MCTS Application

2.8 Summary

- The microprocessor performs basic four operations such as Memory Read ($\overline{MEMR}$), Memory Write ($\overline{MEMW}$), I/O Read ($\overline{IOR}$), I/O Write ($\overline{IOW}$).
- The address bus, data bus and control bus govern the entire communication between MPU, memory and peripherals
- The architecture of 8085 is robust to handle different kinds of operation.
- Memory is integral part of MPU and different types of memories help in execution of different activities of the microprocessor all identified by 16 bit address.
- I/O devices are interfaced either by 8 bit address or 16 bit address.
- To interconnect peripherals and MPU several interfacing devices are used to ease the operation

2.9 List of References

- Ramesh Gaonkar, “Microprocessor Architecture, Programming and Applications with the 8085”, Fifth Edition, Penram International Publishing (I) Private Limited
- B. Ram, “Fundamentals of Microprocessor and Microcomputers”. Sixth Revised and Enlarged Edition, Dhanpat Rai Publications (P) Limited
- https://www.tutorialsmate.com/2020/04/types-of-computer-memory.html

2.10 Unit End Exercise

1. Explain the operation performed by the 8085 microprocessor?
2. Explain the role of 8085 system bus is communication of MPU with peripherals.
3. What is memory? Draw and explain the memory organization of 16x8 registers on 1 chip and 2 chips with 4 address lines.
4. Illustrate the memory map and address range of the chip with 4096 bytes and explain how the range can be changed by modifying the hardware of the chip select line
5. Explain the following interfacing devices: (A) Tristate device (B) Buffer (C) D-Flip flop

❖❖❖

UNIT 1

3

8085 MICROPROCESSOR ARCHITECTURE AND MEMORY INTERFACING

Unit Structure

3.0 Objectives

3.1 Introduction

3.2 8085 Microprocessor Unit

3.2.1 8085 Microprocessor

3.2.2 Microprocessor communication and bus system

3.2.3 Demultiplexing the address-data bus

3.2.4 Generating Control Signals

3.2.5 The 8085 Microprocessor Architecture

3.3 8085 based Microcomputer Illustration

3.3.1 The 8085 Machine Cycle

3.3.2 Opcode Fetch Cycle

3.3.3 Memory Read Machine Cycle

3.3.4 Memory Write Machine Cycle

3.3.5 How to Recognize Machine Cycle

3.4 Memory Interfacing

3.4.1 Memory structure and its requirements

3.4.2 Basic concepts in memory interfacing

3.4.3 Address Decoding and Memory Address

3.4.3.1 Read Only Memory Chip

3.4.3.2 Read/Write Memory Chip

3.5 Interfacing 8155 Memory Segment

3.6 Designing Memory for MCTS Project

3.7 Testing and Troubleshooting Memory Interfacing Circuit

3.7.1 Testing

3.7.2 Troubleshooting

3.7.3 Diagnostic Routine

3.8 8085-Based Single-Board microcomputer

3.9 Summary

3.10 List of references

3.11 Unit End Exercise

3.0 Objectives

After going through this chapter, you will be able to

- Understand the purpose of each pin
- Know the role of system bus in decoding and executing an instruction
- Know the different techniques available in memory interfacing
- Understand the difference between absolute and partial decoding
- Need for testing and troubleshooting memory interface circuit.

3.0 Introduction

- The 8085 microprocessor is enhanced version of its predecessor 8080A.
- Introduced in the year 1976, the 8085 is the predominantly and widely accepted 8 bit microprocessor.
- The instruction set of 8085 is upward compatible with additional instructions.

3.1 8085 Microprocessor Unit

- The 8085 is 8-bit general purpose microprocessor packaged as dual inline package (DIP) with 40 pins.
- It works on single +5V power supply and operates on 3MHz clock frequency.
- It has 16 address lines with which it can address 64K memory and 8 data lines.

3.1.1 8085 MICROPROCESSOR

- The Pin diagram of 8085 microprocessor is categorized into six groups

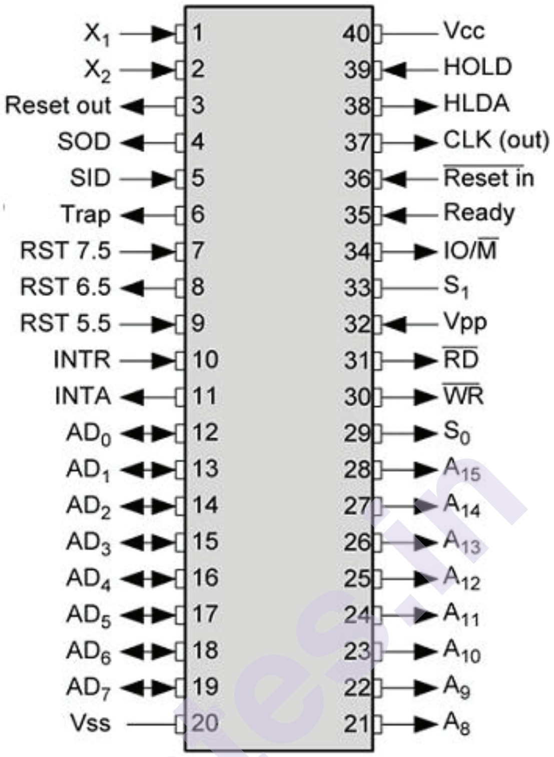


Figure 3.1 – Pin Configuration

- **Address bus**
  - There are 16 address lines which are unidirectional
  - The first eight pins A_{15} - A_8 carry the high order address i.e. the most significant bits and the remaining eight pins A_7 - A_0 are combined with data bus which holds the low order address i.e. the least significant bits and data.

- **Multiplexed Address-Data bus**
  - There are eight pins that serve dual purpose – It holds low order address and data and are identified as AD_7 – AD_0.
  - When they hold the address the lines are unidirectional and when the lines hold data they are bidirectional
  - During instruction execution, initially these pins hold address which is latched and then same pins hold data during the latter part of the execution.

- **Control and status signals**
  - Address Latch Enable (ALE) is output signal which determines the multiplexed bus contains address or data. ALE = 1→AD_7 – AD_0 holds address and ALE = 0 → AD_7 – AD_0 holds data.
  - Read ($\overline{RD}$) is active low output signal to read from memory location or peripherals.
  - Write ($\overline{WR}$) is active low output signal to write to memory location or peripherals.
  - Input-Output/Memory (IO/$\overline{M}$) is output status signal to help the microprocessor differentiate between peripheral and memory related operation. IO/$\overline{M}$ = 1 → peripheral operation and IO/$\overline{M}$ = 0 → memory operation.
  - Status signal (S1 & S0) are rarely used output status signal for status of operation of microprocessor.
- **Power supply and frequency signals**
  - Power Supply (V_{cc}) is +5V
  - Ground (V_{ss}) – 0V
  - Clock Input (X_1 and X_2) are input pins connected to crystal to make the system operate at 3MHz and so crystal should have 6MHz.
  - Clock Output (CLK OUT) is output signal used as system clock to trigger other devices.
- **Externally initiated signals**
  - Interrupt Request (INTR) is active high input signal for general purpose interrupt.
  - Interrupt Acknowledge ($\overline{INTA}$) is active low output signal for response to the INTR signal.
  - Restart Interrupts (RST 7.5, RST 6.5 and RST 5.5) are active high input maskable interrupts with RST 7.5 is edge triggered and other two are level triggered.
  - Trap (TRAP) interrupt is active high input edge and level triggered and non-maskable highest priority hardware interrupt.

  - Hold (HOLD) is active high input signal to allow Direct Memory Access (DMA) between peripherals and memory.
  - Hold Acknowledge (HLDA) is active high output response to HOLD signal.
  - Ready (READY) is active high input control signal to delay the read or write operation as peripherals are slow devices compared to microprocessor
  - Reset Input ($\overline{RESET\ IN}$) is active low signal to reset the microprocessor by setting program counter address to 0000H and buses in tristate mode.
  - Reset Output (RESET OUT) is active high signal by the microprocessor to reset other devices.

- **Serial I/O ports**
  - Serial Input Data (SID) is active high input signal for serial transmission.
  - Serial Output Data (SOD) is active high output signal for serial reception.

3.1.2 Microprocessor Communication and Bus System

- The communication process is four-fold.
- First step involves the microprocessor placing the 16-bit address on the program counter.
- Next the active low read control signal ($\overline{RD}$) on the control bus
- The data in the memory location identified by the address in now on placed on the data bus.
- The data bus carries it to instruction decoder where the action is taken according to the instruction.
- The diagram represents the execution of the instruction MOV C, A stored in address 2005H

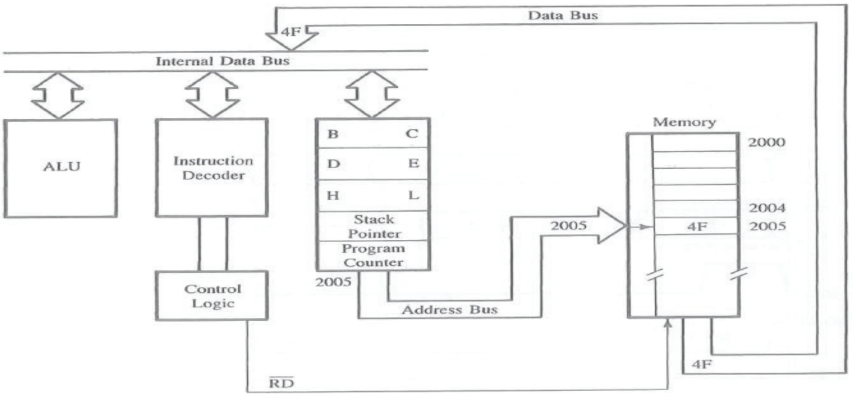


Figure 3.2 – Communication flow of instruction execution

3.1.3 Demultiplexing the Address-Data Bus

- In 8085, the high order address lines A_{15} - A_8 are directly available.
- The low order address lines are multiplexed with data bus as AD_7– AD_0.
- So it necessary to demultiplex (separate) the address and data.

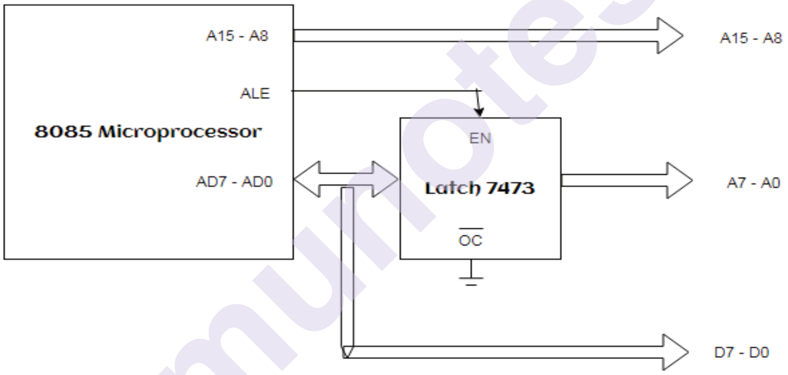
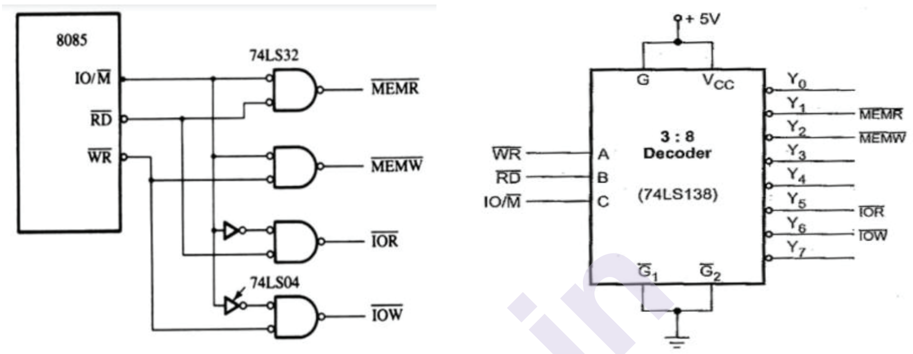


Figure 3.3 – Latching the multiplexed bus

- Execution of any instruction (opcode) requires four clock period.
- The high order address is available for three clock period and low order is available for only one clock period and hence will be lost
- So we save the address in a latch which is enabled by ALE pin.
- When ALE is high, the latch is enabled and the bus contains address which is stored in latch and when ALE is low the latch is disabled and the bus now contains data which is directly used.

3.1.4 Generating Control Signals

- The control bus is responsible for carrying the control signal which tells the operation the microprocessor is to perfom.
- Three pins read $(\overline{RD})$, write $(\overline{WR})$ and Input-Output/Memory (IO/$\overline{M}$) are combined to generate four control signals.
- The signals can be generated using negative NAND gates or a 3:8 decoder.



Using negative NAND gate **Using 3:8 Decoder**

Figure 3.4 – Generating the control signal

3.1.5 The 8085 Microprocessor Architecture

- The architecture is divided into five functional groups.

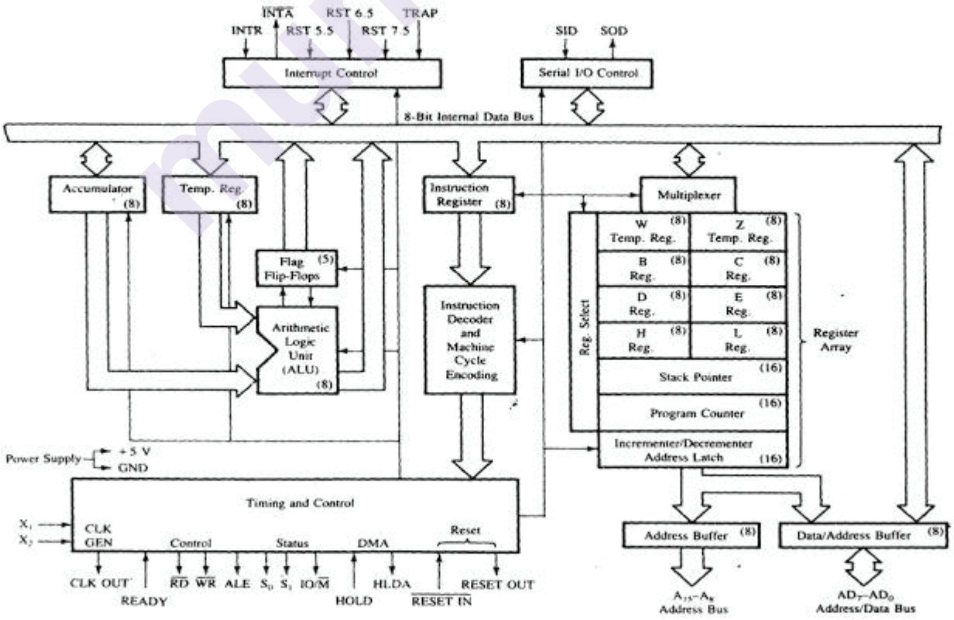


Figure 3.5 – 8085 Architectural Block Diagram

- **Arithmetic and Logical Group**
    - It includes the Arithmetic-Logic Unit which performs the arithmetic operation like addition, subtraction, increment and decrement and logical operation like AND, OR, XOR, complement, compare and rotate.
    - An 8-bit general purpose register called accumulator or ‘A’ register which holds one of the operand and result of the most of the operation.
    - An 8-bit temporary register not available for programmer but used by microprocessor for certain operation
    - A 8 bit flag register (made up of flip-flops) which defines five flags which are set (value = 1) or reset (value =0) depending on the result of the arithmetic and logical operation
    - The various flag flip-flops are are Sign (S), Zero (Z), Auxiliary Carry (AC), Parity (P) and Carry (CY) flag.

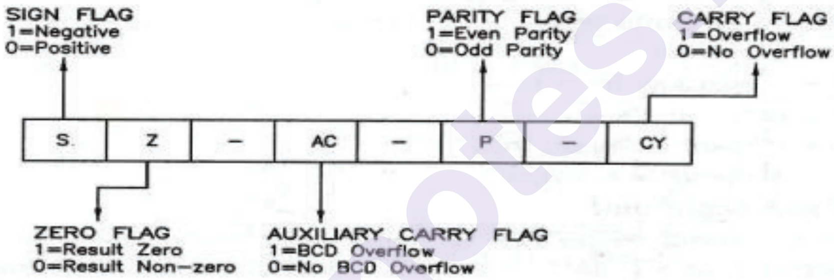


Figure 3.6 – 8085 Flag Register Format

- **Register Group**
    - A pair of temporary register W and Z not available to the programmer but used by the microprocessor during the stack operations.
    - Six general purpose 8-bit register B, C, D, E, H and L to store data.
    - They are combined to form register pairs BC, DE and HL to store 16-bit data.
    - A 16-bit program counter (PC) to hold the address of the instruction to be executed and which keeps incrementing to fetch the next instruction till the end of program.
    - A 16-bit stack pointer register used as pointer to stack location where data is stored temporarily.

- **Interrupt Control Group**
    - Various interrupt pins discussed earlier like TRAP, RST 7.5, RST 6.5 and RST 5.5, are handled by executing the Interrupt Service Routine (ISR) at the corresponding vector address to decide the action to be taken when the interrupt occurs.
    - For the interrupt INTR, it generates $\overline{INTA}$ signal to acknowledge and send opcode of CALL instruction to transfer control to that address mentioned by the interrupting device.

- **Instruction Register, Decoder and Control Unit Group**
    - The Instruction Register (IR) is 8bit register that stores opcode of the current instruction being executed and used only by the microprocessor and not the programmer.
    - The Instruction Decoder accepts opcode from IR then decodes it and sends information to control logic to perform operation specified by the instruction.
    - The timing-control unit generates various signals to sequence and synchronize the microprocessor operation.

- **Serial I/O Group**
    - The 8085 is a parallel device but using the pins SID and SOD it can support serial transmission and reception and also instruction RIM and SIM help in performing serial transfer.

3.2 8085 Based Microcomputer Illustration

- The 8085 microprocessor executes the instruction in terms of instruction cycle, machine cycle and T-states
- The time taken for the complete execution of single instruction is the instruction cycle.
- The time taken for the complete execution of one operation such as accessing memory or peripheral is the machine cycle.
- One sub division of operation performed in synchronization with system clock is called the T-state.
- The 8085 one instruction cycle can have one to six machine cycle and each machine cycle has three to six T-states.

3.3.1 The 8085 Machine Cycle

- There are 74 instructions in 8085 microprocessor and instruction is made of opcode and operand.
- Some instructions are one byte and some multi-byte.
- Hence different machine cycles such as opcode fetch, memory read/write, I/O read/write are required for the decoding and execution.

3.3.2 Opcode Fetch Cycle

- All 8085 instruction must have opcode as compulsory part and operand is optional.
- So the first machine cycle of any instruction is opcode fetch cycle which has four T-states.
- It uses the first three states T1 —T3, to fetch the code and T4 to decode and execute the opcode.
- The high order address A_{15} - A_8 is available for three T-states.
- The multiplexed bus AD_7– AD_0 hold low order address during T1 which is indicated by ALE signal going high and between T2- T3 holds the data indicated as ALE goes low.
- Opcode fetch is a memory operation and indicated by status signals IO/$\overline{M}$, S_1 and S_0 = 011.
- When opcode (data) is placed in the data bus the read signal ($\overline{RD}$) goes low to enable the operation.

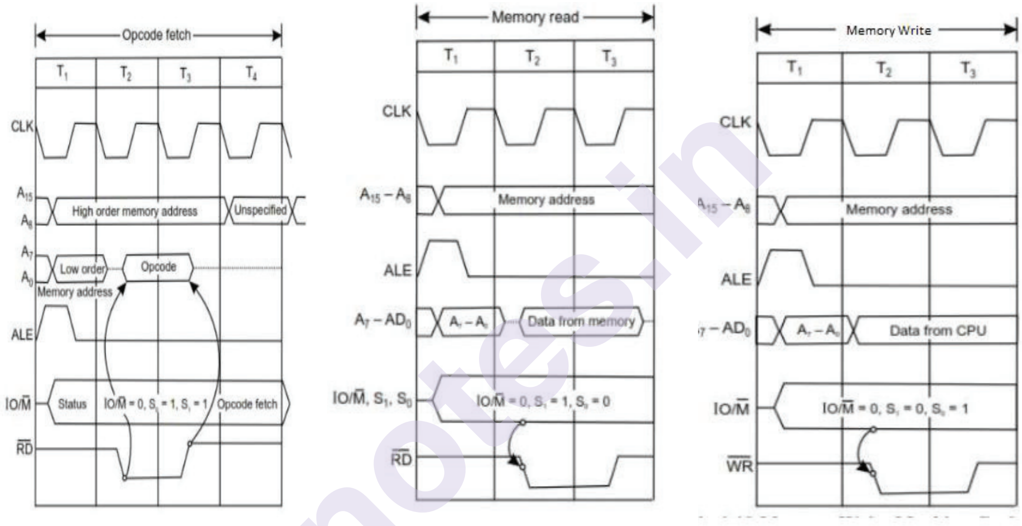
3.3.3 Memory Read Machine Cycle

- The memory read machine cycle requires three T-states to perform the operation.
- The high order address and multiplexed address data bus logic remains same as opcode fetch cycle.is available for three T-states.
- Memory read operation is indicated by status signals IO/$\overline{M}$, S_1 and S_0 = 010.

  When data is placed in the data bus from memory location the read signal ($\overline{RD}$) goes low to enable the operation.

3.3.4 Memory Write Machine Cycle

- The memory read machine cycle requires three T-states to perform the operation.
- The high order address and multiplexed address data bus logic remains same as opcode fetch cycle.is available for three T-states.
- Memory write operation is indicated by status signals IO/$\overline{M}$, S_1 and S_0 = 001.
- When data is placed in the memory location from the data bus now instead of read, the write signal ($\overline{WR}$) goes low to enable the operation



Opcode Fetch Cycle **Memory Read Cycle** **Memory Write Cycle**

Figure 3.7 – 8085 Machine Cycle

3.3.5 How To Recognize Machine Cycle

- The first illustration is execution of the instruction MVI B, 43H stored in address 2000H.
- It is a two byte instruction and it requires two machine cycle to execute the instruction.
- The first machine cycle is opcode fetch which requires four T-states and second machine cycle is memory read which requires three T-states and so a total of seven T-states.

Memory Address	Mnemonic	Hex / Machine Code
2000	MVI B, 43H	06 (0000 0110)
2001		43 (0100 0011)

- In the first machine cycle the microprocessor decodes the opcode as seen in the opcode fetch machine cycle and in the second machine cycle the memory places the data byte on the data bus which is then stored in register B at the end of T_3.
- The execution time of the instruction is calculated as
  - Let us assume the Clock frequency f = 2 MHz
  - T-state = clock period (1/f) = 0.5μs
  - Execution time for 1st Machine Cycle (Opcode Fetch) = 4T x 0.5 = 2μs
  - Execution time for 2nd Machine Cycle (Memory Read) = 3T x 0.5 = 1.5μs
  - Total Execution time for Instruction: 2μs + 1.5μs = 3.5μs

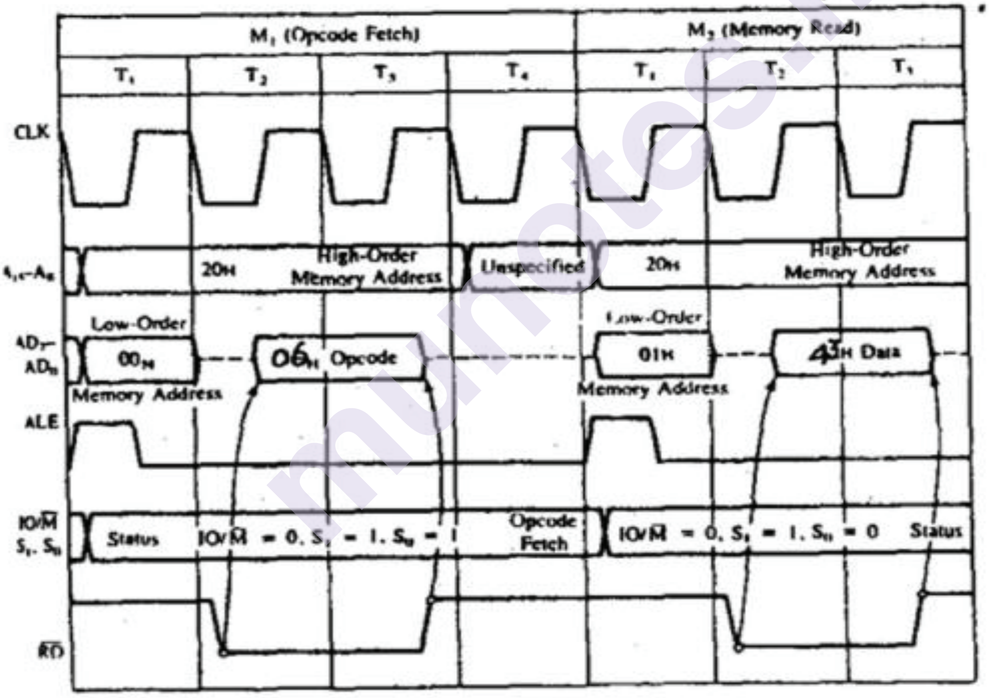


Figure 3.8 – Machine Cycles for the instruction MVI B, 43H

- Consider another illustration of executing the instruction STA 8000H stored in memory address 2050H.
- It is a three byte instruction but it requires four machine cycle to execute the instruction.
- ***There is no direct relationship between the number of bytes of instruction and the number of machine cycles required to decode that instruction.***

- The first machine cycle is opcode fetch which requires four T-states and second and the third machine cycle is memory read with three T-states each followed by final cycle of memory write with three T-states making it a total of thirteen T-states.
- In the opcode fetch cycle, the microprocessor places the address 2050H in address bus and opcode 32H in data bus.
- In the first memory read cycle the microprocessor reads the address 2051H and the data bus reads the low order byte 00H.
- In the second memory read machine cycle the microprocessor reads the address 2052H and the data bus reads the high order byte 80H.
- In the last cycle the address 8000H is placed in the address bus and the byte available in the accumulator is stored in this location via the data bus.
- The execution time of the instruction is calculated as
  - Let us assume the Clock frequency f = 2 MHz
  - T-state = clock period (1/f) = 0.5μs
  - Execution time for 1st Machine Cycle

    (Opcode Fetch) = 4T x 0.5 = 2μs
  - Execution time for 2nd Machine Cycle
    (Memory Read) = 3T x 0.5 = 1.5μs
  - Execution time for 3rd Machine Cycle
    (Memory Read) = 3T x 0.5 = 1.5μs
  - Execution time for 4th Machine Cycle
    (Memory Write) = 3T x 0.5 = 1.5μs
  - Total Execution time for Instruction:
    2μs + 1.5μs + 1.5μs + 1.5μs = 6.5μs

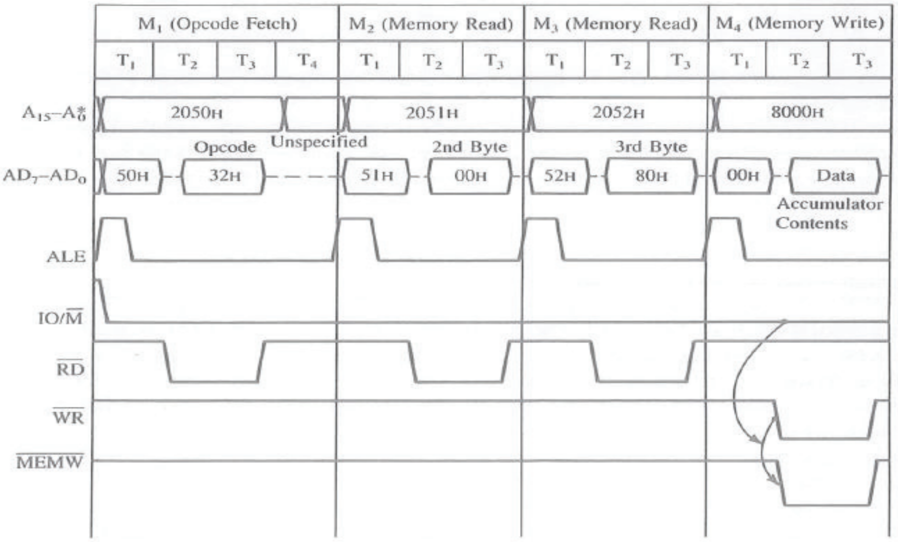


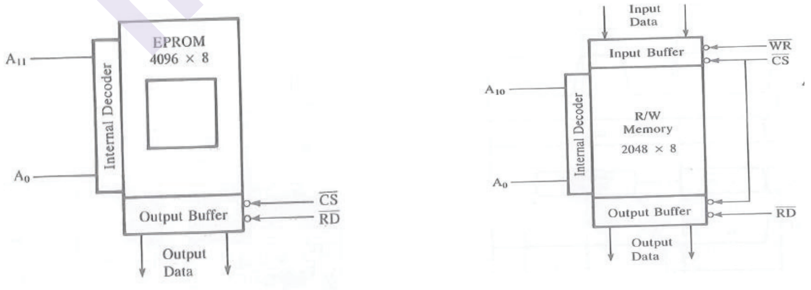
Figure 3.9 – Machine Cycles for the instruction STA 8000H

3.4 Memory Interfacing

- Memory is an important component of the microprocessor based system where we store data, program and results.
- So the memory must be properly interfaced so it can be easily and frequently accessed to read data and instructions and write results to it.

3.4.1 Memory Structure and its Requirements

- The microprocessor based system possess Read Only memory chip with 4096 registers and R/W memory chip with 2048 registers and each register capable of storing 8 bits.



Read Only Memory Chip (4096 x 8)

R/W Memory Chip (2048 x 8)

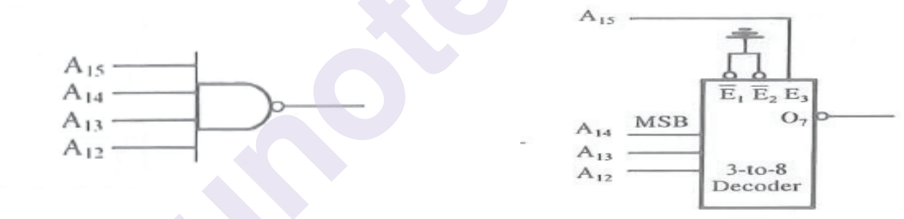
Figure 3.10 – Memory Chip

3.4.2 Basic Concepts in Memory Interfacing

- The basic idea of memory interfacing involves foremost selecting the memory chip, followed by the identifying individual register and enabling the buffer for the right operation.
- The memory read and write cycle explained earlier illustrates the interfacing concept.

3.4.3 Address Decoding and Memory Address

- Address decoding is the process of identifying the memory chip and register for a given address.
- A unique pulse identifies the address and this pulse can be generated using NAND gate and 3:8 Decoder.
- The output of NAND gate selects the chip when address A_{15}- A_{12} will be high.
- The same result can be obtained 3:8 decoder which can decode 8 devices based on the address bits A_{14}- A_{12}.



Using NAND Gate **Using Decoder**

Figure 3.11 – Interfacing Techniques

3.4.3.1 Read Only Memory Chip

- A typical EPROM memory with 4096 registers is used which is identified by 12 address lines A_{11}—A_0, one chip select signal and only one control signal $\overline{RD}$ to activate output buffer.
- The address lines A_{11}—A_0 are connected to memory chip to select register.
- The address lines A_{14}—A_{12} are connected as inputs to 3:8 decoder which when asserted low selects the chip ($\overline{CE}$) and A_{15} is connected to enable pin of the decoder.
- So the value A_{15}—A12 = 0000 enables the decoder to assert output through O_0.

- The control signal $\overline{MEMR}$ is generated by ORing IO/$\overline{M}$ and $\overline{RD}$ which enables the output buffer$\overline{OE}$.

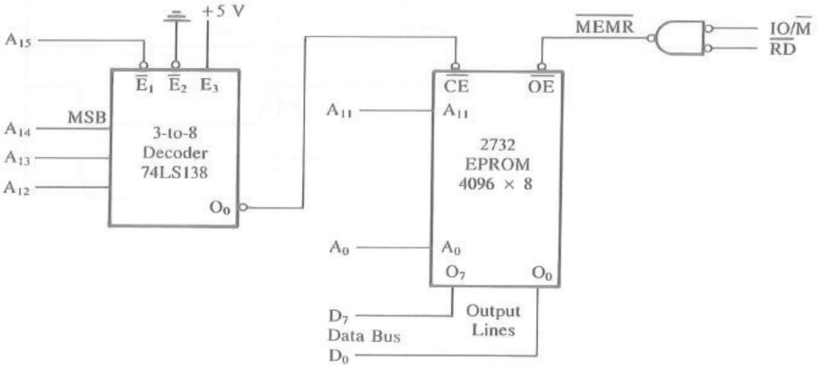


Figure 3.12 – Interfacing 4096 x 8 EPROM Memory

- The address range for the above memory would be

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1

Chip Select

Register Select

Table 3.1 - Memory map for 4096 x 8 EPROM

- The range is 0000H - 0FFFH.

3.4.3.2 Read/ Write Memory Chip

- A typical R/W memory with 2048 registers is used which is identified by 11 address lines A_{10}—A_0, one chip select signal and control signals

- The address lines A_{10}—A_0 are connected to memory chip to select register.

- The address lines A_{13}—A_{11} are connected as inputs to 3:8 decoder which when asserted low selects the chip ($\overline{CE}$) and remaining address lines A_{15} and A_{14} along with IO/$\overline{M}$ is connected to enable pin of the decoder

- So the value A_{15}—A11 = 10001 enables the decoder to assert output through O_1.

- The control lines $\overline{RD}$ and $\overline{WR}$ are directly connected to memory chip to enable output and input buffer respectively and the signals $\overline{MEMR}$ and $\overline{MEMW}$ are not generated.

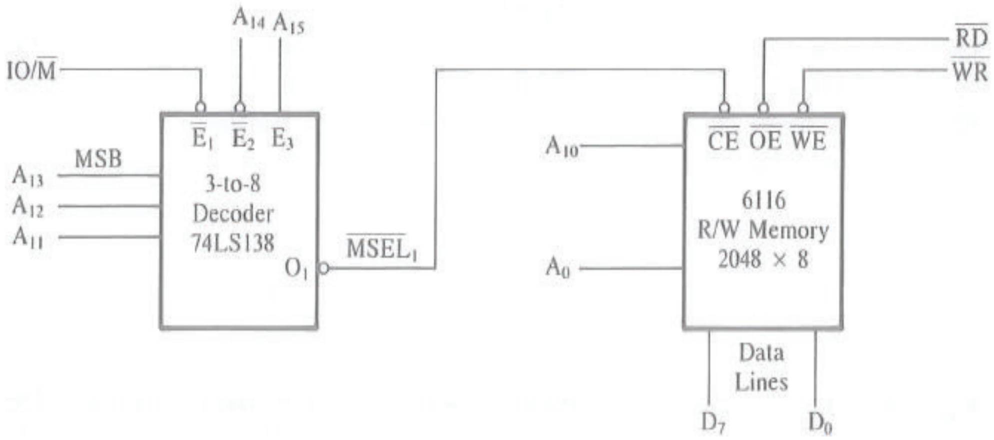


Figure 3.13 – Interfacing 2048 x 8 R/W Memory

- The address range for the above memory would be

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1

Chip Select

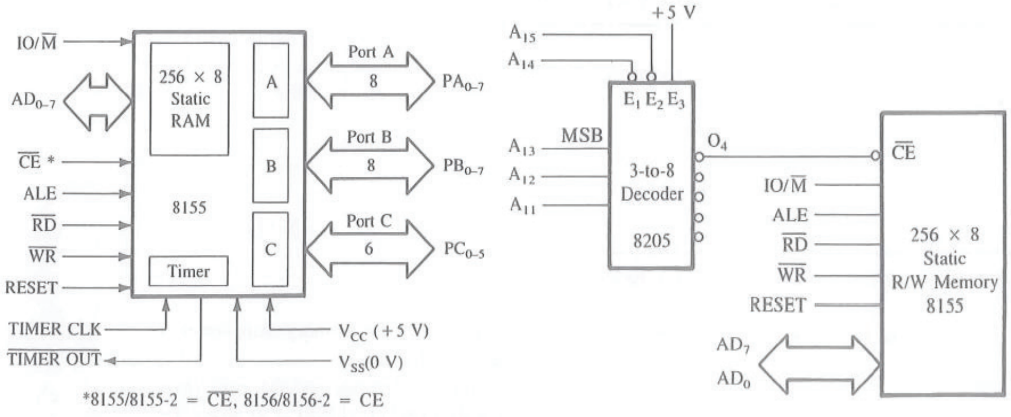
Register Select

Table 3.2 - Memory map for 2048 x 8 R/W Memory

- The address range is 8800H - 8FFFH.
- Both the memory chips used all address lines for decoding and this is known as *absolute decoding*.

3.5 Interfacing 8155 Memory Segment

- The 8155 is multipurpose programmable device interfacing the peripherals the 8085 microprocessor containing memory (256 bytes R/W memory), I/O ports (three ports 8-bit each) and timer.
- The 256 x 8 memory is identified with 8 address lines A_7-A_0 and the remaining address lines A_{15}-A_{11} are connected to decoder with A_{15}-A_{14} connected to enable pin and A_{13}-A_{11} connected to input of the decoder and output O_4 of the decoder is asserted low to enable the chip. So A_{15}-A_{11} = 00100 enables the decoder.
- The control signals IO/$\overline{M}$, $\overline{RD}$ and $\overline{WR}$ are directly connected to the memory chip to enable the buffer.



8155 Memory Section **Interfacing 8155 Memory**

Figure 3.14 – 8155 Memory and SDK-85

- The address lines A_{10}-A_8 are not connected and considered as don't care lines.
- This interfacing technique where all address lines are not used for decoding is called as *partial decoding* and advantageous as cost saving technique.
- The address range for the above memory is called foldback or mirror memory.

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Address
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2000H
0	0	1	0	0	0	0	0	1	1	1	1	1	1	1	1	20FFH
0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	2100H
0	0	1	0	0	0	0	1	1	1	1	1	1	1	1	1	21FFH
0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	2200H
0	0	1	0	0	0	1	0	1	1	1	1	1	1	1	1	22FFH
0	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	2300H
0	0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	23FFH
0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	2400H
0	0	1	0	0	1	0	0	1	1	1	1	1	1	1	1	24FFH
0	0	1	0	0	1	0	1	0	0	0	0	0	0	0	0	2500H
0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	25FFH
0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	2600H
0	0	1	0	0	1	1	0	1	1	1	1	1	1	1	1	26FFH
0	0	1	0	0	1	1	1	0	0	0	0	0	0	0	0	2700H
0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	27FFH

Chip Select Don't Care Register Select

Table 3.3 - Memory map for 256 x 8 R/W Memory

- The primary address range is 2000H -20FFH and foldback memory range is 2100H – 27FFH.

3.6 Designing Memory for MCTS Project

- The interfacing circuit for microprocessor controlled temperature system MCTS studied in the earlier chapter includes 4K x 8 EPROM and 2K x 8 R/W Memory and 3:8 Decoder for interfacing.

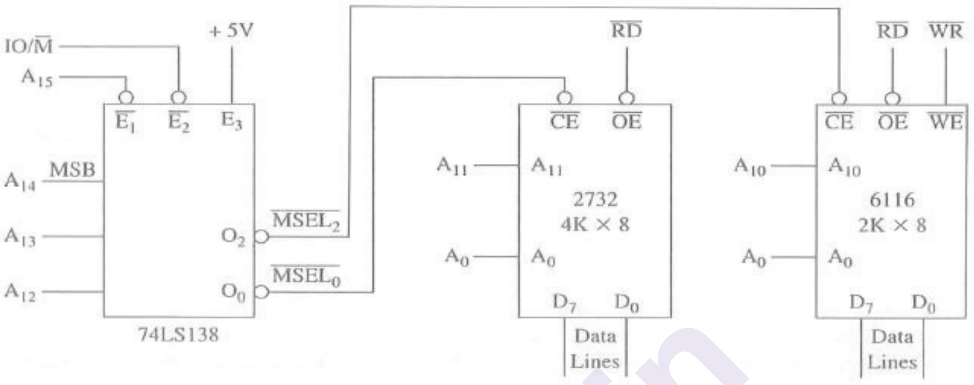


Figure 3.15 – MCTS Project Memory Interfacing

- The EPROM memory with 4K memory is identified by 12 address lines A_{11}—A_0 and remaining address lines A_{15}—A_{12} = 0000 with O_0 of decoder asserted low to enable the chip and $\overline{RD}$ directly connected to enable output buffer.

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1

Chip Select

Register Select

Table 3.4 - Memory map for 4K EPROM

- The range is 0000H - 0FFFH

- The R/W memory with 2K memory is identified by 11 address lines A_{10}—A_0 and remaining address lines A_{15}—A_{12} = 0010 with O_2 of decoder asserted low to enable the chip and $\overline{RD}$ directly connected to enable output buffer and $\overline{WR}$ directly connected to enable the input buffer.

- The address line A_{11} is not connected resulting in foldback memory.

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Address
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2000H
0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	27FFH
0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	2800H
0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	2FFFH

Chip Select Don't Care Register Select

- The primary address range is 2000H -27FFH and foldback memory range is 2800H – 2FFFH.

3.7 Testing and Troubleshooting Memory Interfacing Circuit

- Memory is integral part of microcomputer system as it holds data and instructions
- Hence testing and troubleshooting of memory is fairly important to check the integrity of the contents.

3.7.1 Testing

- Testing is simply done by randomly choosing a memory location and writing the data byte at that location with help of keyboard and then reading the content of the same memory location and displaying the result in display.
- If the content matches then memory is available and we can repeat this test at different memory locations.
- When we cannot load a byte then we need to perform troubleshooting.

3.7.2 Troubleshooting

- The best way to perform troubleshooting is visual inspection.
- We check wires, pin connections which is easy but logic level of buses checking is difficult as it is dynamic.
- Signal injection is useful technique where signal is injected at input and check output as per expected outcome
- To perform signal injection we write diagnostic routine.

3.7.3 Diagnostic Routine

- Diagnostic routine involves writing continuous loop.

  ```
  loop: MVI A, AAH
  STA 2000H
  JMP loop
  ```

- The infinite loop loads value AAH in accumulator and then stores the value in the memory location 2000H and this process is repeated infinitely.
- We can diagnose the following
  - If we cannot read the value AAH we need to check data bus connections.
  - If we cannot read the address 2000H, we need to check the address bus connections
  - If output of decoder is asserted high, then we need to check the address line connections to the decoder chip.
  - The control signals $\overline{RD}$ and $\overline{WR}$ have to be active low to perform the operation.

3.8 8085-Based Single-Board microcomputer

- When we turn on power, a monitor program is executed where program counter is reset to 0000H and hex code is loaded from there.
- The monitor program reads hex keyboard and check for closure.
- It then display the key pressed at display and simultaneously stores binary equivalent memory.
- Finally transfer the program execution to user program as the EXECUTE button is pressed.

3.9 Summary

- The 8085 microprocessor is dual inline package with 40 pins with each pin having a unique functionality.
- The architecture of 8085 has functional groups that helps to carry the operation with ease.

- Each instruction is executed in instruction cycle which comprises of several machine cycle which comprises of several T-states.
- There is no direct relationship between the number of bytes of instruction and the number of machine cycles required to decode that instruction.
- In interfacing the memory, if all address lines are used then it represents absolute decoding and if few address lines are used its partial decoding and this reduces hardware and generates foldback mirror which provides multiple addresses.

3.10 List of references

- Ramesh Gaonkar, "Microprocessor Architecture, Programming and Applications with the 8085", Fifth Edition, Penram International Publishing (I) Private Limited.
- https://tutorialspoint.com
- https://www.brighthubengineering.com
- https://www.javatpoint.com

3.11 Unit End Exercise

1. State the functions of the following pins: (i) X_1 (ii) HLDA (iii) IO/$\overline{M}$ (iv) $\overline{INTA}$ (v) READY
2. Illustrate the steps and timing of data flow for the instruction MOV D, M stored in memory location 2000H.
3. Illustrate the steps and timing of data flow for the instruction LDA 4050H stored in memory location 3000H.
4. How to interface the EPROM. What is the address decoding technique & state the memory address range?
5. How to test and troubleshoot memory interfacing circuit?

❖❖❖

UNIT-2

4

I/O INTERFACING

Unit Structure

4.0 Objectives

4.1 Introduction

4.2 Basic Interfacing Concepts

4.2.1 Peripheral I/O Instructions

4.2.2 I/O Execution

4.2.3 Device Selection And Data Transfer

4.2.4 Absolute vs Partial Decoding

4.2.5 Input Interfacing

4.2.6 Interfacing I/Os Using Decoders

4.3 Interfacing Output Displays

4.3.1 Illustration: LED Display for Binary Data

4.3.2 Illustration: Seven- Segment LED Display as an Output Device

4.4 INTERFACING INPUT DEVICES

4.4.1 Illustration: Data Input from DIP Switches

4.5 Memory –Mapped I /O

4.5.1 Execution of Memory- Related Data Transfer Instructions

4.5.2 Illustration: Safety Control System Using Memory-Mapped I/O Technique

4.5.3 Comparison of Memory-Mapped I/O and Peripheral I/O

4.6 Testing And Troubleshooting I/O Interfacing Circuits

4.6.1 Diagnostic Routine And Machine Cycles

4.7 Summary

4.0 Objectives

- Illustrate the 8085 bus contents and control signals when OUT and IN instructions are executed.
- Recognize the device (port) address of a peripheral-mapped I/O by analyzing the associated logic circuit.
- Illustrate 8085 bus contents and control signal when memory- related instructions (LDA, STA, etc) are executed.
- Recognize the device (port) address of a memory-mapped I/O by analyzing the associated logic circuit.
- Explain the difference between the peripheral-mapped and memory-mapped I/O techniques.
- Interface an I/O device to a microcomputer for a specified device address by using logic gates and MSI chips, such as decoders, latches, and buffers.

4.1 Introduction

Any application of a microprocessor based system requires the transfer of data between external circuitry to the microprocessor and microprocessor to the external circuitry User can give information to the microprocessor based system using keyboard and user can see the result or output information from the microprocessor based system with the help of display device. The transfer of data between keyboard and microprocessor, and microprocessor and display device is called input / output data transfer or I/O data transfer. This data transfer is done with the help of I / O ports.

4.1.1 Input port:-

It is used to read data from the input device such as keyboard. The simplest form of input port is a buffer. The input device is connected to the microprocessor through buffer as shown in the figure. This buffer is a tri-state buffer and its output is available only when enable signal is active. When microprocessor wants to read data from the input device (keyboard), the control signals from the microprocessor activates the buffer by asserting enable input of the buffer. Once the buffer is enabled, data from input device is available on the data bus. Microprocessor reads this data by initiating read command.

4.1.2 Output port : -

It is used to send data to the output device such as display from the microprocessor. The simplest form of output port is a latch. The output device is connected to the microprocessor through latch, as shown in the figure, When microprocessor wants to send data to the output device it puts the data on the data bus and activates the clock signal of the latch, latching the data from the data bus at the output of latch. It is then available at the output of latch for output device.

4.2 Basic Interfacing Concepts

The approach to designing an interfacing circuit for an I/O device is determined primarily by the instructions to be used for data transfer. An I/O device can be interfaced with the 8085 microprocessor either as a peripheral I/O or as a memory-mapped I/O.

In the peripheral I/O, the instructions IN/OUT are used for data transfer, and the device is identified by an 8- bit address. In the memory-mapped I/O, memory-related instructions are used for data transfer, and device is identified by a 16-bit address.

However, the basic concepts in interfacing I/O devices are similar in both methods. Peripheral I/O is described in the following section, and memory- mapped I/O in further section.

4.2.1 Peripheral I/O Instructions

The 8085 microprocessor has two instructions for data transfer between the processor and the I/O device: IN and OUT.

The instruction IN (Code DB) inputs data from an input device (such as a keyboard) into accumulator, and the instruction OUT (Code D3) sends the contents of the accumulator to an output device such as an LED display. These are 2-byte instructions, with the second byte specifying the address or the port number of an I/O device. For example, the OUT instruction is described as follows.

Opcode	Operand	Description
OUT	8- bit Port Address	This is a two-byte instruction with the hexadecimal opcode D3, and the second byte is the port address of an output device. This instruction transfers (copies) data from the accumulator to the output device.

Typically, to display the accumulator at an output device (such as LEDs) with the address, for example, 01H, the instruction will be written and stored in memory as follows:

Memory Address	**Machine Code**	**Mnemonics**		**Memory Contents**
2050	D3	OUT 01H	:2050	11010011=D3H
2051	01		:2051	00000001=01H

(Note: The memory locations 2050H are chosen here arbitrarily for the illustration.)

If the output port with the address 01H is designed as an LED display, the instruction OUT will display the contents of the accumulator at the port. The second byte of this OUT instruction can be any of the 256 combinations of eight bits, from 00H to FFH. Therefore, the 8085 can communicate with 256 different output ports with device addresses ranging from 00H to FFH. Similarly, the instruction IN can be used to accept data from 256 different input ports. Now the question remains: How does one assign a device address or a port number to an I/O device from among 256 combinations? The decision is arbitrary and somewhat dependent on available logic chips. To understand a device address, it is necessary to examine how the microprocessor executes IN/OUT instructions.

4.2.2 I/O Execution

The execution of I/O instructions can best be illustrated using the example of the OUT instruction given in the previous section (4.2.1). the 8085 executes the OUT instruction in three machine cycles, and it takes ten T-states (clock periods) to complete the execution.

Out Instruction (8085)

In the first machine cycle, M_1 (Opcode Fetch, Figure 4.1) the 8085 places the high-order memory address 20H on A_{15}-A_8 and the low-order address 50H on AD-AD_0. At the same time, ALE goes high and IO/ M goes low. The ALE signal indicates the availability of the address on AD_7-AD_0, and it can be used to demultiplex the bus. The IO/M, being low, indicates that it is a memory-related operation. At T_2, the microprocessor sends the $\overline{RD}$ control signal, which is combined with IO/M (externally, to generate the ($\overline{MEMR}$) signal, and the processor fetches the instruction code D3 using the data bus.

When the 8085 decodes the machine code D3, it finds out that the instruction is a 2- byte instruction and that it must read the second byte.

In the second machine cycle, M_2 (Memory Read), the 8085 places the next address, 2051 H, on the address bus and gets the device address 01H via the data bus.

On the low –order (AD_7 - AD_0) as well as high-order (A_{15} - A_8) address bus. The IO/Ṁ signal goes high to indicate that it is an I/O operation. At T_2, the accumulator contents are placed on the data bus (AD_7 - AD_0), followed by the control signal WR

By ANDing the IO/Ṁ and $\overline{WR}$ signals, the $\overline{IOW}$ signal can be generated to enable an output device.

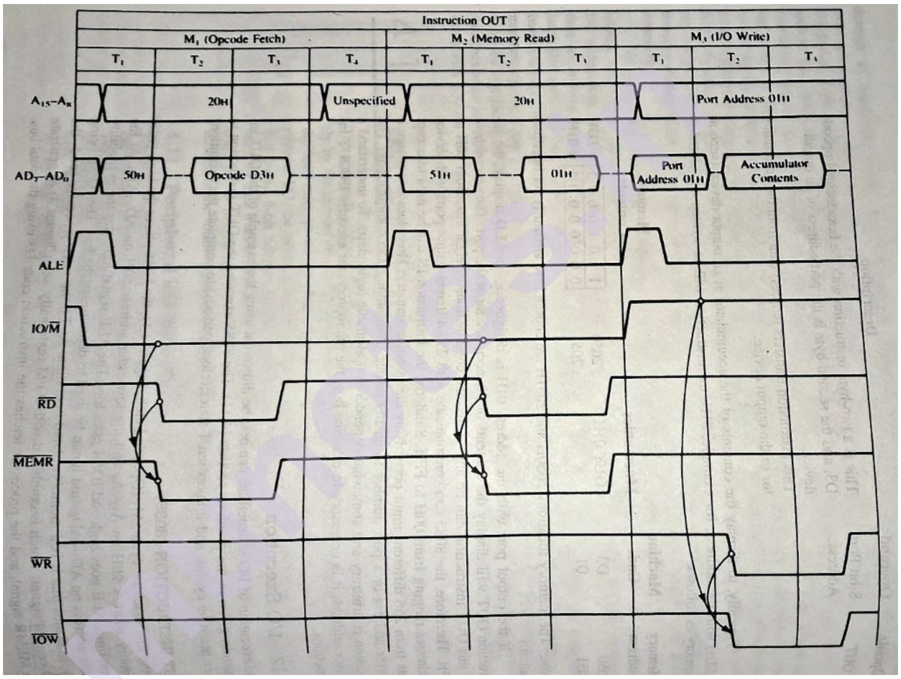


Figure 4.1 : 8085 Timing for Execution of OUT Instruction

Figure 4.1 shows the execution timing of the OUT instruction. The information necessary for interfacing an output device is available during T_2 and T_3 of the M_3 cycle. The data byte to be displayed is on data bus, 8- bit device address is available on the low- order as well as high-order address bus, and availability of the data byte is indicated by the $\overline{WR}$ control signal. The availability of the device address on both segments of the address bus is redundant information; in peripherals I/O, only one segment of the address bus (low or high) is sufficient for interfacing. The data byte remains on the data bus only for two T- states, then the processor goes on to execute the next instruction. Therefore, the data byte must be latched now, before it is lost, using the device address and control signal

In Instruction

The 8085 instruction set includes the instruction IN to read (copy) data from input devices such as switches, keyboard, and A/D data convertors. This is a two- byte instruction that reads an input device and places the data in the accumulator. The first byte is the opcode, and the second byte specifies the port address. Thus, the addresses for input devices can range from 00H to FFH . The instruction is described as

IN 8- bit :- This is a two- byte instruction with the hexadecimal opcode DB, and the second byte is the port address of an input device.

This instruction reads (copies) data from an input device and places the data byte in the accumulator.

To read switch positions, for example, from an input port with the address 84H, the instructions will be written and stored in memory as follows :

Memory Address	**Machine Code**	**Mnemonics**		**Memory Contents**
2065	DB	IN 84H	:2065	11011011=DBH
2066	84		:2066	10000100=84H

(Note: - The memory locations 2065H and 66H are selected arbitrary for the illustration.)

When the microprocessor is asked to execute this instruction, it will first read the machine codes (or bytes) stored at locations 2065H and 2066H, then read the switch positions at port 84H by enabling the interfacing device of the port. The data byte indicating switch positions from the input port will be placed in the accumulator. Figure shows the timing of the IN instruction; M_1 and M_2 cycles are identical to that of the OUT instruction.

In the M_3 cycle, the 8085 microprocessor places the address of the input port (84H) on the low- order address bus AD_7 –AD_0 as well as on the high- order address bus A_{15} – A_8 and asserts the $\overline{RD}$ signal, which is used to generate the I/O Read (IOR) signal.

The $\overline{IOR}$ enables the input port, and the data from input port are placed on the data bus and transferred into the accumulator.

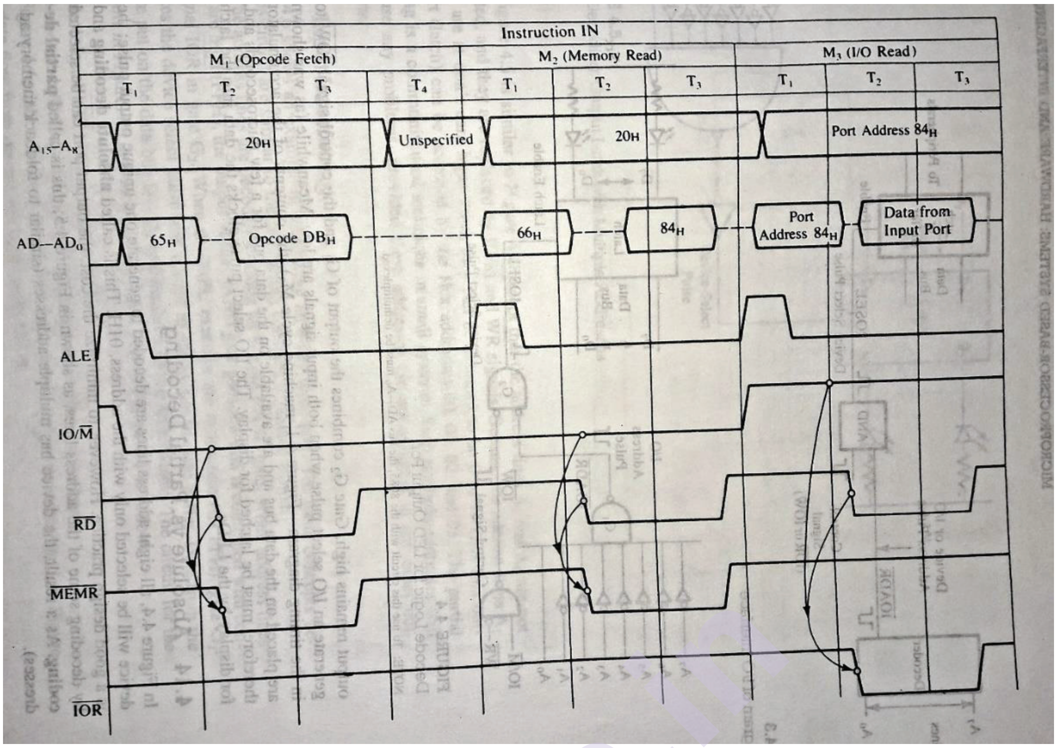


Figure 4.2:- 8085 Timing for Execution of IN Instruction

Machine cycles M_3 (Figure 4.2) is similar to the M_3 cycle of the OUT instruction; the only differences are (1) the control signal is $\overline{RD}$instead of $\overline{WR}$ and (2) data flow from an input port to the accumulator rather than from the accumulator to an output port.

4.2.3 Device Selection And Data Transfer

The objective of interfacing an output device is to get information or a result out of the processor and store it or display it. The OUT instruction serves that purpose; during the M_3 cycle of the OUT instruction the processor places that information (accumulator contents) on the data bus. If we connect the data bus to a latch, we can catch that information and display it via LEDs or a printer. Now the questions are : (1) When should we enable the latch to catch that information ? and (2) What should be the address of that latch ? The answers to both questions can be found in the M_3 cycle (Figure 4.1) .The latch should be enabled when IO/M is high and $\overline{WR}$ is active low. Similarly, the address of an output port is also on the address bus during M_3 (it is 01H in figure 4.1). Now the task is to generate one pulse by decoding the address bus (A_7 - A_0 or A_{15} – A_8) to indicate the presence of the port address we are interested in, generate a timing pulse by combining IO/M and $\overline{WR}$ signals to indicate that the data byte we are looking for is on the data bus, and use these pulses (by combining them) to enable the latch. These steps are summarized as follows. (For all subsequent discussion, the bus A_7- A_0 is assumed to be that demultiplexed bus AD_7 – AD_0).

1. Decode the address bus to generate a unique pulse corresponding to the device address on the bus; this is called device address pulse or I/O address pulse.
2. Combine (AND) the device address pulse with the control signal to generate a device select (I/ O select) pulse that is generated only when both signals are asserted.
3. Use the I/O select to activate the interfacing device (I/O port).

The block diagram (Figure 4.3) illustrate these steps for interfacing an I/O device.

In Figure 4.3, address lines A_7 - A_0 connected to a decoder, which will generate a unique pulse corresponding to each address on the address lines. This pulse is combined with the control signal to generate a device select pulse, which is used to enable an output latch or an input buffer.

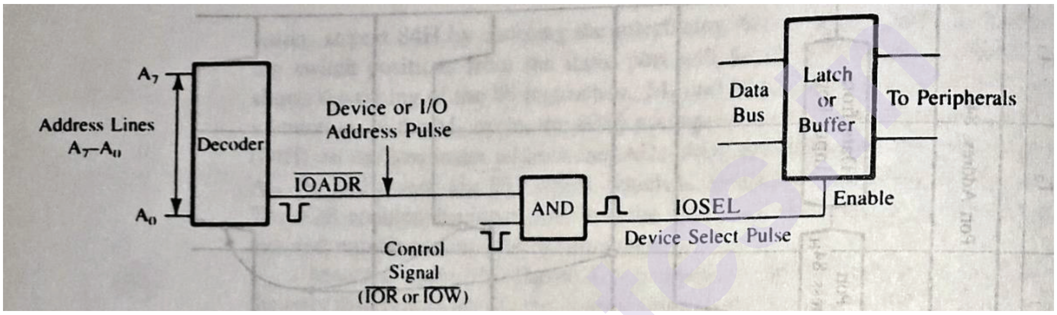


Figure 4.3 :- Block Diagram of I/O Interface

Figure 4.4 shows a practical decoding circuit for the output device with address 01H. Address lines $A_7 – A_0$ are connected to 8- input NAND gate that functions as a decoder. Lines A_0 is connected directly, and lines $A_7 – A_1$ are connected through the inverters. When the address bus carries address 01H, gate G_1 generates a low pulse; otherwise, the output remains high. Gate G_2 and the control signal $\overline{IOW}$ to generate an I/O select pulse when both input signals are low.

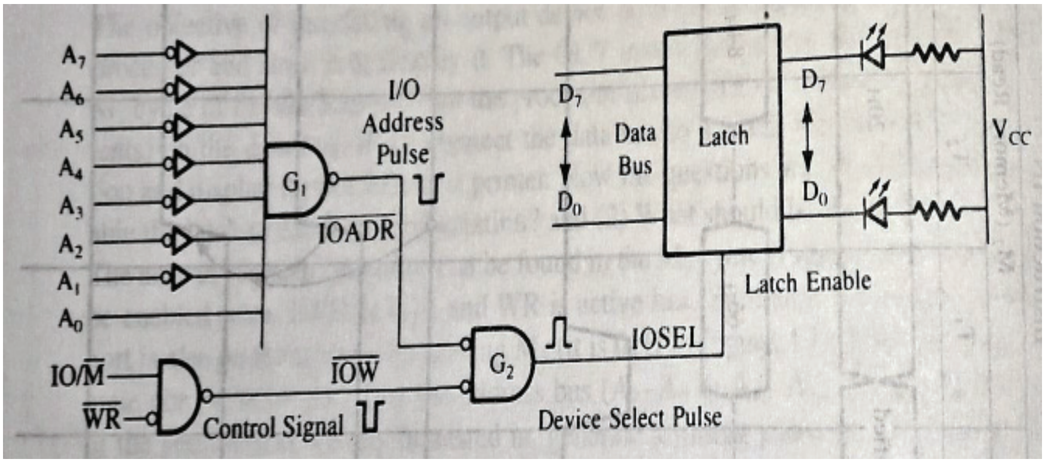


Figure 4.4:- Decoder Logic for LED Output Port

Meanwhile the contents of the accumulator are placed on the data bus and are available on the data bus for a few microseconds and, therefore, must be latched for display. The I/O select pulse clocks the data into the latch for display by the LEDs.

4.2.4 Absolute vs Partial Decoding

In figure 4.4, all eight address lines are decoded to generate one unique output pulse; the device will be selected only with the address, 01H. This is called **absolute decoding** and is a good design practice. However, to minimize the cost, the output port can be selected by decoding some of the address lines, as shown in figure ; this is called **partial decoding.**

As a result, the device has multiple addresses (similar to fold back memory addresses).

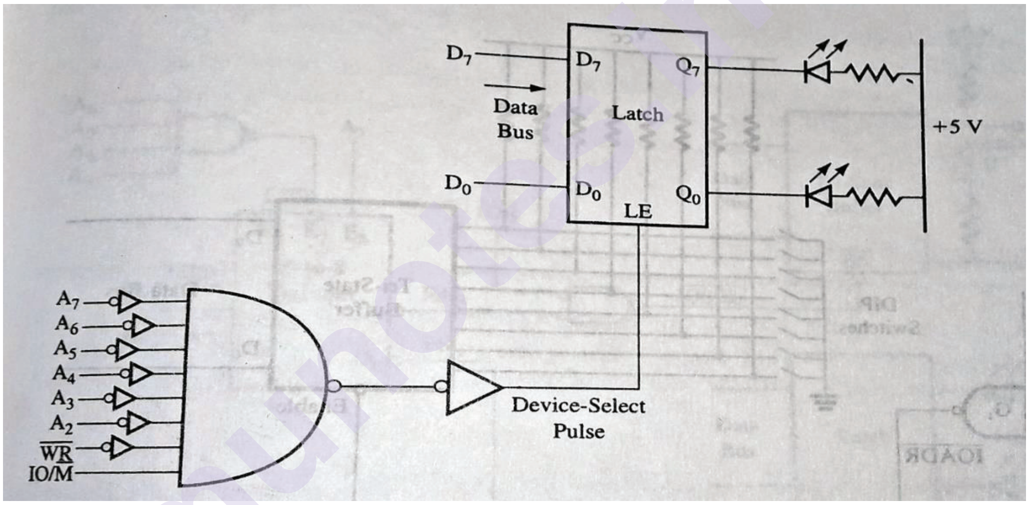


Figure 4.5 :- Partial Decoding : Output Latch with Multiple Addresses

Figure 4.5 is similar to figure 4.4 except that the address lines A_1 and A_0 are not connected, and they are replaced by IO/ M and $\overline{WR}$ signals. Because the address lines A_1 and A_0 are at don't care logic level, they can be assumed to be 0 and 1.

Thus this output port (latch) can be accessed by the Hex addresses 00, 01, 02 and 03. The partial decoding is a commonly used technique in small systems. Such multiple addresses will not cause any problems, provided these addresses are not assigned to any other output ports.

4.2.5 Input Interfacing

Figure 4.6 shows an example off interfacing an 8-key input port. The basic concepts behind this circuit are similar to the interfacing concepts explained earlier.

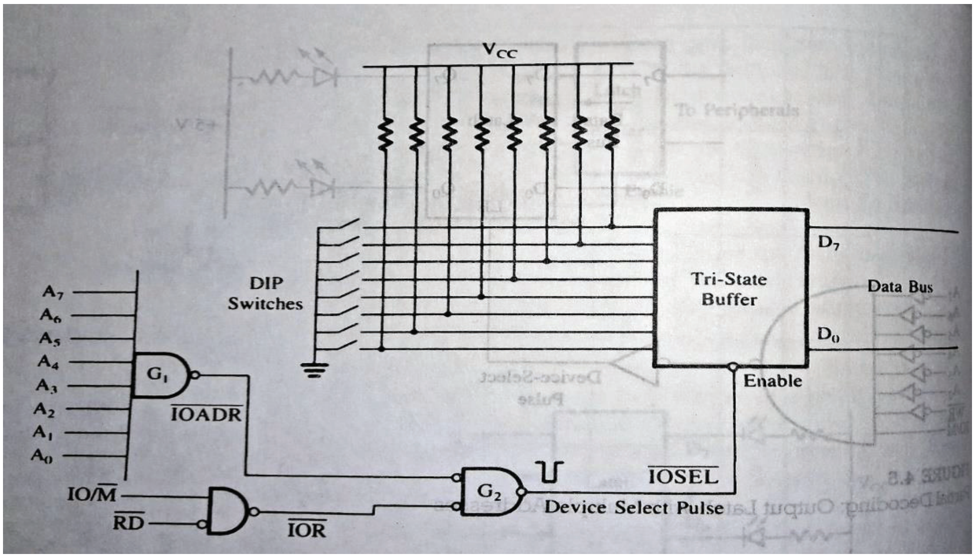


Figure 4.6 :- Decode Logic for a Dip – switch Input Port

The address lines are decoded by using an 8- input NAND gate. When address lines A_7 – A_0 are high(FFH), the output of the NAND gate goes low and is combined with control signal $\overline{IOR}$ in gate G_2. When the MPU executes the instruction (IN FFH), gate G_2 generates the device select pulse that is used to enable the tri- state buffer. Data from the key are put on the data bus D_7 – D_0 and loaded into the accumulator. The circuit for the input port in figure differs from the output port in figure as follows:

1. Control signals $\overline{IOR}$ is used in place of $\overline{IOW}$.
2. The tri-state buffer is used as an interfacing port in place of the latch.
3. In figure 4.6, data flow from the keys to the accumulator on the other hand, in figure, data flow from the accumulator to the LEDs.

4.2.6 Interfacing I/Os Using Decoders

Various techniques and circuits can be used to decode an address and interface an I/O device to the microprocessor. However, all of these techniques should follow the three basic steps suggested in above section.

Figure 4.4 and 4.6 illustrate an approach to device selection using an 8- input NAND gate. Figure 4.5 illustrate a technique using minimum hardware; this technique has the disadvantage of having multiple addresses for same device.

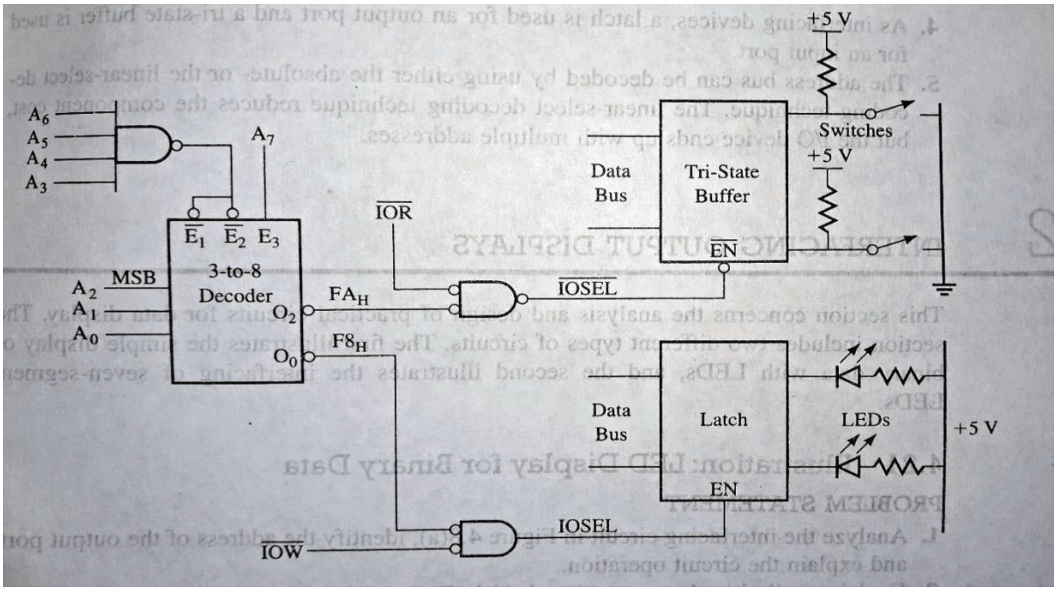


Figure 4.7 :- Address Decoding Using 3-to-8 Decoder

Figure 4.7 illustrate another scheme of address decoding. In this circuit, a 3- to- 8 decoder and a 4- input NAND gate are used to decode the address bus; the decoding of the address bus is the first step in interfacing I/O devices.

The address lines A2, A1 and A0 are used as input to the decoder, and the remaining address lines $A_7 – A_3$ are used to enable the decoder. The address lines A_7 is directly connected to E_3 (active high Enable line), and the address lines $A_6 – A_3$ are connected to E_1 and E_2 (active low enable line) using the NAND gate. The decoder has eight output lines; thus, we can use this circuit to generate eight device address pulses for eight different addresses.

The second step is to combine the decoded address with an appropriate control signal to generate the I/O select pulse. Figure shows that the output O_0 of the decoder is logically ANDed in a negative AND gate with the $\overline{IOW}$ control signal.

The output of the gate is the I/O select pulse for an output port. The third step is to use this pulse to enable the output port. Figure shows that the I/O select pulse enables the LED latch with the output port address F8H, as shown below ($A_7 – A_0$ is the de multiplexed low- order bus).

Similarly, the output O_2 of the decoder is combined with I/O Read (IOR) signal, and the I/O select pulse is used to enable the input buffer with the address FAH.

4.3 Interfacing Output Displays

This section concerns the analysis and design of practical circuits for data display. The section includes two different types of circuits. The first illustrates the simple display of binary data with LEDs, and the second illustrate the interfacing of seven-segment LEDs.

4.3.1 IIIustration: LED Display for Binary Data

Problem Statement

1. Analyze the interfacing circuit in figure, identify the address of the output port, and explain the circuit operation.
2. Explain similarities between (a) and (b) in figure.
3. Write instructions to display binary data at the port.

Circuit Analysis

Address bus $A_7 - A_0$ is decoded by using an 8- input NAND gate. The output of the NAND gate goes low only when the address lines carry the address FFH. The output of the NAND gate is combined with the microprocessor control signal $\overline{IOW}$ in a NOR gate (connected as negative AND).The output of NOR gate 74LS02 goes to generate an I/O select pulse when both inputs are low (or both signals are asserted). Meanwhile, the contents of accumulator have been put on the data bus. The I/O select pulse is used as a clock pulse to activate the D- type latch, and the data are latched and displayed.

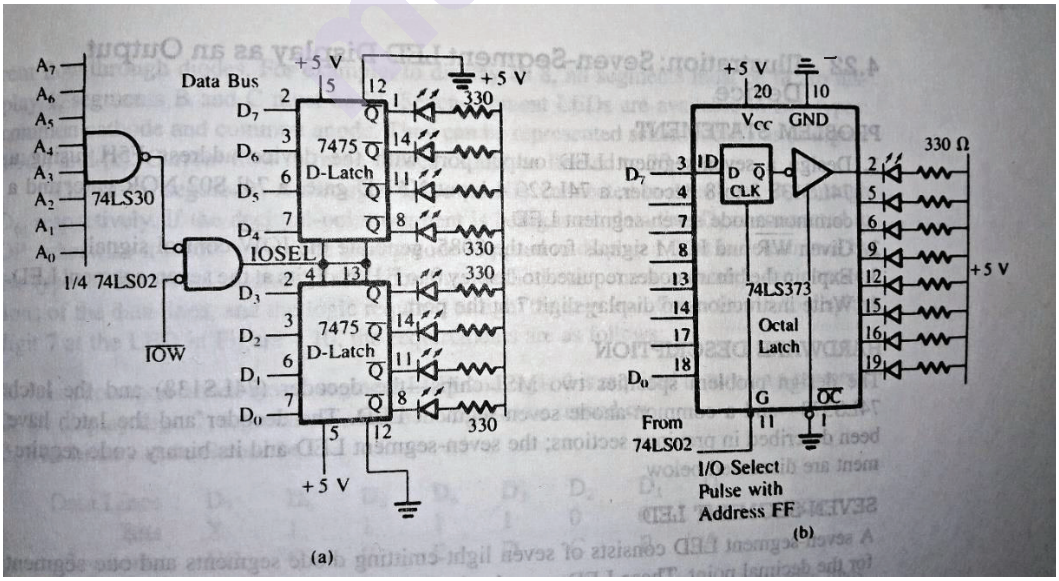


Figure 4.8 :- Interfacing LED Output Port Using the 7475 D-Latch (a) and Using the 74LS373 Octal D-Type Latch (b)

In this circuit, the LED cathodes are connected to the Q output of the latch. The anodes are connected to +5V through resistors to limit the current flow through the diodes. When the data line (for example D_0) has 1, the output Q is 0 and the corresponding LED is turned on. If the LED anode were connected to Q, its cathode would be connected to the ground .In this configuration, the D flip-flop would not be able to supply the necessary ground. In this configuration, the D flip-flop would not be able to supply the necessary current to the LED.

Figure 4.8 uses the 74LS373 octal latch an interfacing device, and both circuit(a) an (b) are functionally similar. The 74LS373 includes D-latches(flip-flops) followed by tri-state buffers. This device has two control signals: Enable (G) to clock data in the flip-flops and Output Control (OC) to enable the buffers. In this circuit, the 74LS373 is used as a latch; therefore the tri- state buffers are enabled by grounding the OC signal.

Program

Address (LO)	Machine Code	Mnemonics	Comments
00	3E	MVI A, DATA	; Load accumulator with data
01	DATA*		
02	D3	OUT FFH	; Output accumulator contents to port FFH
03	FF		
04	76	HLT	; End of program

Program Dscription

Instruction MVI loads the accumulator with the data you enter, and instruction OUT FFH identifies the LED port as the output device and displays the data.

4.3.2 Illustration: Seven- Segment LED Display as an Output Device

Problem Statement

1. Design a seven-segment LED output port with the device address F5H, using a 74LS138 3 –to- 8 decoder, a 74LS20 4-input NAND gate, a 74LS02 NOR gate, and a common anode seven- segment LED.

2. Given $\overline{WR}$ and IO/ M signals from the 8085, generate the $\overline{IOW}$ control signal.

3. Explain the binary codes required to display 0 to F Hex digits at the seven-segment LED.

4. Write instructions to display digit 7 at the port.

Hardware Description

The design problem specifies two MSI chips – decoder (74LS138) and the latch 74LS373 – and a common- anode seven-segment LED. The decoder and the latch have been described in previous sections; the seven-segment LED and its binary code requirement are discussed below.

Seven- Segment Led

A seven- segment LED consists of seven light- emitting diode segments and one segment for the decimal point. These LEDs are physically arranged as shown in Figure 4.9 (a) . To display a number, the necessary segments are lit by sending an appropriate signal for current flow through diodes.

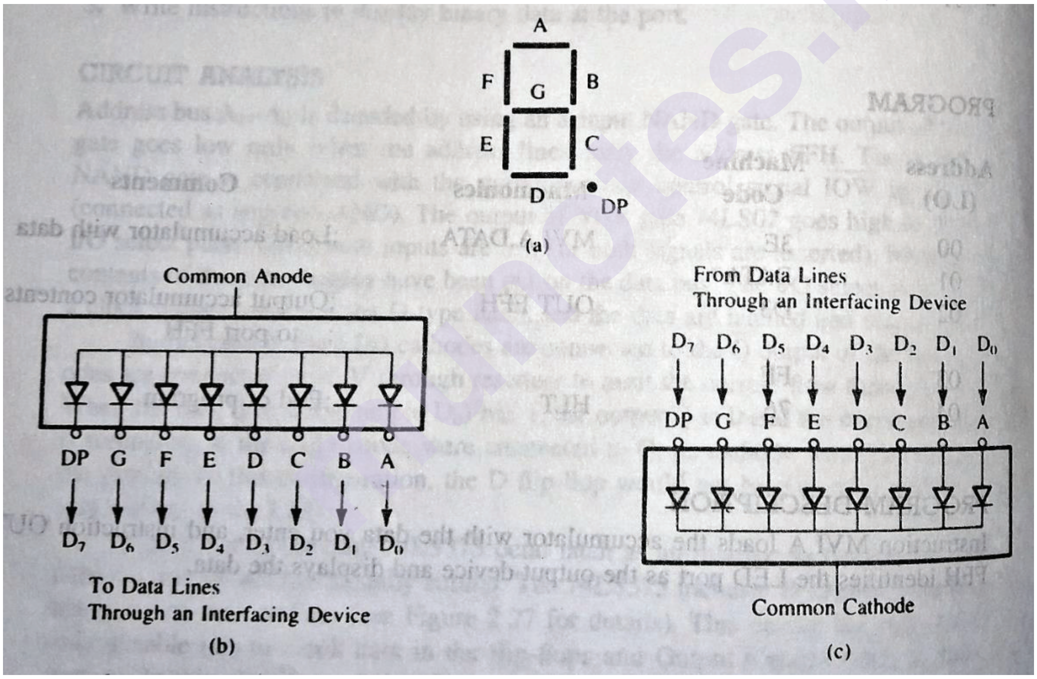

A
F
G
B
E
C
D
DP
(a)
Common Anode
DP G F E D C B A
D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0
To Data Lines
Through an Interfacing Device
(b)
From Data Lines
Through an Interfacing Device
D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0
DP G F E D C B A
Common Cathode
(c)

Figure 4.9 :- Seven-Segment LED : LED Segments(a) ;Common-Anode LED(b); Common- Cathode LED (c)

For example, to display an 8, all segments must be lit. To display 1, segments B and C must be lit. Seven- segment LEDs is available in two types: common cathode and common anode. They can be represented schematically as in Figure 4.9 (b) and (c). Current flow in these diodes should be limited to 20mA.

The seven segments, A through G, are usually connected to data lines D_0 through D_6, respectively. If the decimal-point segment is being used, data line D_7 is connected to DP; otherwise it is left open. The binary code required to display a digit is determined by the type of the seven-segment LED (common cathode or common anode), the connections of the data lines, and the logic required to light the segment. For example, to display digit 7 at the LED in Figure 4.10, the requirements are as follows :

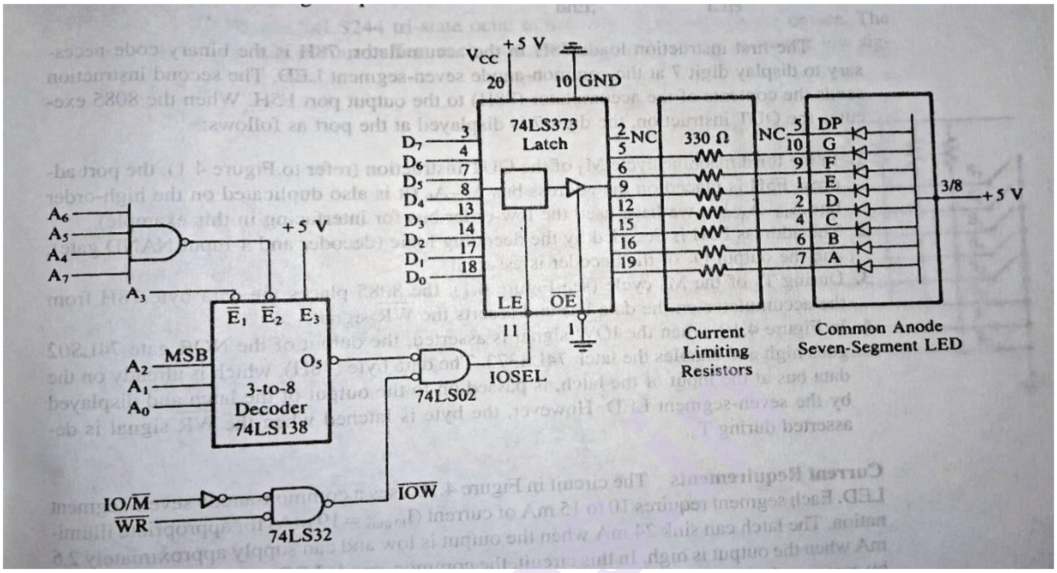


Figure 4.10:- Interfacing Seven-Segment LED

1. It is a common- anode seven- segment LED, and logic 0 is required to turn on a segment.
2. To display digit 7, segments A, B and C should be turned on.
3. The binary code should be

Data Lines	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	=78H
Bits	X	1	1	1	1	0	0	0	
Segments	NC	G	F	E	D	C	B	A	

The code for each digit can be determined by examining the connections of the data lines to the segments and the logic requirements.

Interfacing Circuit and its Analysis

To design an output port with the address F5H, the address lines $A_7 – A_0$ should have the following logic:

A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
1	1	1	1	0	1	0	1	= F5H

This can be accomplished by using A_2, A_1 and A_0 as input lines to the decoder. A_3 can be connected to active low enable E_1, and the remaining address lines can be connected to E_2 through the 4- input NAND gate. Figure shows an output port with the address F5H. The output O_5 of the decoder is logically ANDed with the control signal $\overline{IOW}$ using the NOR gate (74LS02). The output of the NOR gate is the I/O select pulse that is used to enable the latch (74LS373). The control signal is $\overline{IOW}$ is generated by logically ANDing IO / Ṃ and $\overline{WR}$ signals in the negative NAND gate (physically OR gate 74LS32).

Instructions: The following instructions are necessary to display digit 7 at the output port:

MVI A, 78H ; Load seven- segment code in the accumulator

OUT F5H ; Display digit 7 at port F5H

HLT; End

The first instruction loads 78H in the accumulator; 78H is binary code necessary to display digit 7 at the common- anode seven- segment LED. The second instruction sends the contents of the accumulator (78H) to the output port F5H. When the 8085 executes the OUT instruction, the digit 7 is displayed at the port as follows:

1. In the third machine cycle M_3 of the OUT instruction (refer the figure), the port address F5H is placed on the address bus $A_7 – A_0$ (it is also duplicated on the high- order bus $A_{15} – A_8$, but we have used the low- order bus for interfacing in this example).

2. The address F5H is decoded by the decoding logic (decoder and 4- input NAND gate), and the output O_5 of the decoder is asserted.

3. During T_2 of the M_3 cycle (see Figure), the 8085 places the data byte 78H from the accumulator on the data bus and asserts the $\overline{WR}$ signal.

4. In the Figure 4.10, when the $\overline{IOW}$ signal is asserted, the output of the NOR gate 74LS02 goes high and enables the latch 74LS 373.

The data byte (78H), which is already on the data bus at th input of the latch, is passed on the output of the latch and displayed by the seven-segment LED. However, the byte is latched when the $\overline{WR}$ signal is de-asserted during T_3.

Current Requirements: The circuit in Figure uses a common- anode seven-segment LED. Each segment requires 10 to 15 mA of current ($I_{D\ max}$ = 19 mA) for appropriate illumination. The latch can sink 24mA when the output is low and can supply approximately 2.6 mA when the output is high. In this circuit, common-anode LED segments are turned on by zeros on the output of the latch. If common-cathode seven segment LED were used in this circuit, the output of the latch would have to be high to drive the segments. The current supplied would be about 2.6mA, which is insufficient to make the segments visible.

4.4 Interfacing Input Devices

The interfacing of input device is similar to that of interfacing output devices, except with some differences in bus signals and circuit components. We will follow the same basic steps described in Section and timing diagram for the execution of the IN instruction shown in the Figure.

4.4.1 Illustration: Data Input from DIP Switches

In this section, we will analyze the circuit used for interfacing eight DIP switches, as shown in Figure 4.11. The circuit includes the 74LS138 3- to- 8 decoder to decode the low- order bus and the tri-state octal buffer (74LS244) to interface the switches to the data bus. The port can be accessed with the address 84H ; However, it has multiple addresses, as explained below.

4.4.2 Hardware

Figure 4.11 shows the 74LS244 tri- state octal buffer used as an interfacing device. The device has two groups of four buffers each, and they are controlled by active low signal OE. When OE is low, the input data show up on the output lines (connected to the data bus), and when OE is high, the output lines assume the high impedance state.

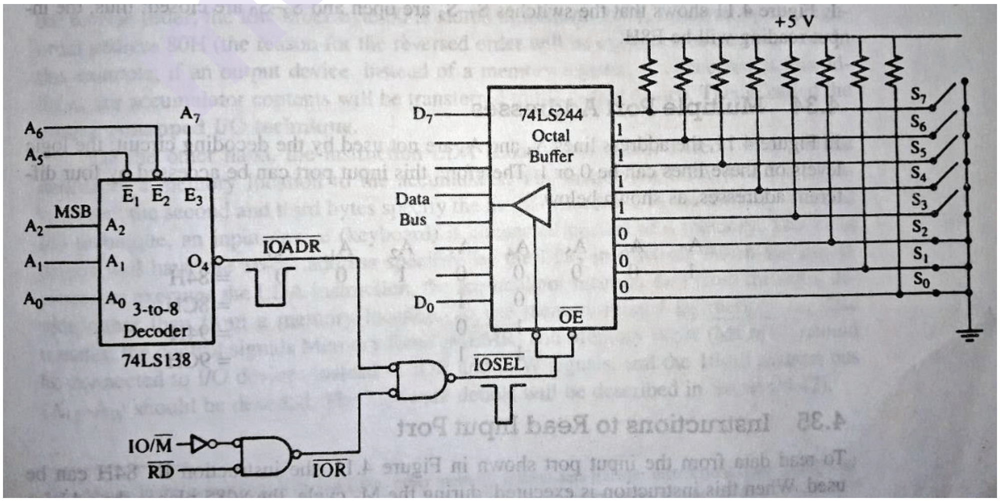


Figure 4.11:- Interfacing DIP Switches

4.4.3 Interfacing Circuit

Figure 4.11 shows that the low order address bus, except the lines A_4 and A_3, is connected to the decoder (the L74S138) ; the address lines A_4 and A_3 are left in the don't care state.

The output line O_4 of the decoder goes low when the address bus has the following address (assume the don't care lines are at logic 0);

A7	A6	A5	A4	A3	A2	A1	A0	
1	0	0	0	0	1	0	0	= 84H
Enables lines			Don't Care		Input			

The control signal I/O Read (IOR) is generated by ANDing the IO / M (through an inverter) and $\overline{RD}$ in a negative NAND gate, and the I/O select pulse is generated by ANDing the output of the decoder and the control signal $\overline{IOR}$. When the address is 84H and the control signal $\overline{IOR}$ is asserted, the I/O select pulse enables the tri-state buffer and the logic levels of the switches are placed on the data bus.

The 8085, then, begins to read switch positions during T_3 (Figure) and places the reading in the accumulator. When a switch is closed, it has logic 0, and when it is open, it is tied to +5V, representing logic 1.

Figure 4.11 shows that the switches S_7 –S_3 are open and S_2 – S_0 are closed; thus, the input reading will be F8H.

4.4.4 Multiple Port Addresses

In Figure 4.11, the address lines A_4 and A_3 are not used by the decoding circuit; the logic levels on these lines can be 0 or 1 .Therefore, this input port can be accessed by four different addresses, as shown below.

A7	A6	A5	A4	A3	A2	A1	A0	
1	0	0	0	0	1	0	0	= 84H
			0	1				=8CH
			1	0				=94H
			1	1				=9CH

4.4.5 Instructions To Read Input Port

To read data from the input port shown in Figure, the instruction IN 84H can be used. When this instruction is executed, during the M_3 cycle, the 8085 places the address 84H on the low- order bus (as well as on the high-order bus), asserts the $\overline{RD}$ control signal, and reads the switch positions.

4.5 Memory –Mapped I /O

In memory-mapped I/O, the input and output devices are assigned and identified by 16- bit addresses. To transfer data between the MPU and I/O devices, memory-related instructions (such as LDA, STA etc) and memory control signals ($\overline{MEMR}$) and $\overline{MEMW}$) are used.

The microprocessor communicates with an I/O device as if it were one of the memory locations. The memory- mapped I/O technique is similar in many ways to the peripheral I/O technique. To understand the similarities, it is necessary to review how a data byte is transferred from the 8085 microprocessor to a memory location or vice- versa. For example, the following instruction will transfer the contents of the accumulator to the memory location 8000H.

Memory Address	Machine Code	Mnemonics	Comments
2050	32	STA 8000H	; Store contents of accumulator in memory location 8000H
2051	00		
2052	80		

(Note : It is assumed here that the instruction is stored in memory locations 2050H, 51H, and 52H).

The STA is a three- byte instruction; the first byte is the opcode, and the second and third bytes specify the memory address. However, the 16- bit address 8000H is entered in the reverse order ; the low- order byte 00 is stored in location 2051, followed by the high-order address 80H (the reason for the reversed order will be explained in Section). In this example, if an output device, instead of a memory register, is connected at this address, the accumulator contents will be transferred to the output device. This is called the **memory- mapped I/O technique.**

On the other hand, the instruction LDA (Load Accumulator Direct) transfers the data from a memory location to the accumulator. The instruction LDA is a 3-byte instruction; the second and third bytes specify the memory location. In the memory-mapped I/O technique, an input device (keyboard) is connected instead of a memory. The input device will have the 16- bit address specified by the LDA instruction.

When the microprocessor executes the LDA instruction, the accumulator receives data from the input device rather than from a memory location. To use memory-related instructions for data transfer, the control signals Memory Read ($\overline{MEMR}$)

and Memory Write (MEMW) should be connected to I/O devices instead of $\overline{IOR}$ and $\overline{IOW}$ signals, and the 16- bit address bus ($A_{15} – A_0$) should be decoded. The hardware details will be described in section).

4.5.1 Execution of Memory- Related Data Transfer Instructions

The execution of memory-related data transfer instructions is similar to the execution of IN or OUT instructions, except that the memory-related instructions have 16-bit addresses.

The microprocessor requires four machine cycles (13 T-states) to execute the instruction STA (Figure 4.12). The machine cycle M_4 for the STA instruction is similar to the machine cycle M_3 for the OUT instruction.

For example, to execute the instruction STA 8000H in the fourth machine cycle (M_4), the microprocessor places memory address 8000H on the entire address bus (A_{15} - A_0) .The accumulator contents are sent on the data bus, followed by the control signal Memory Write $\overline{MEMW}$ (active low).

On the other hand, in executing the OUT instruction (Figure), the 8- bit device address is repeated on the low- order address bus ($A_0 – A_7$) as well as on the high-order bus, and the $\overline{IOW}$ control signal is used. To identify an output device, either the low-order or the high-order bus can be decoded. In the case of the STA instruction, the entire bus must be decoded.

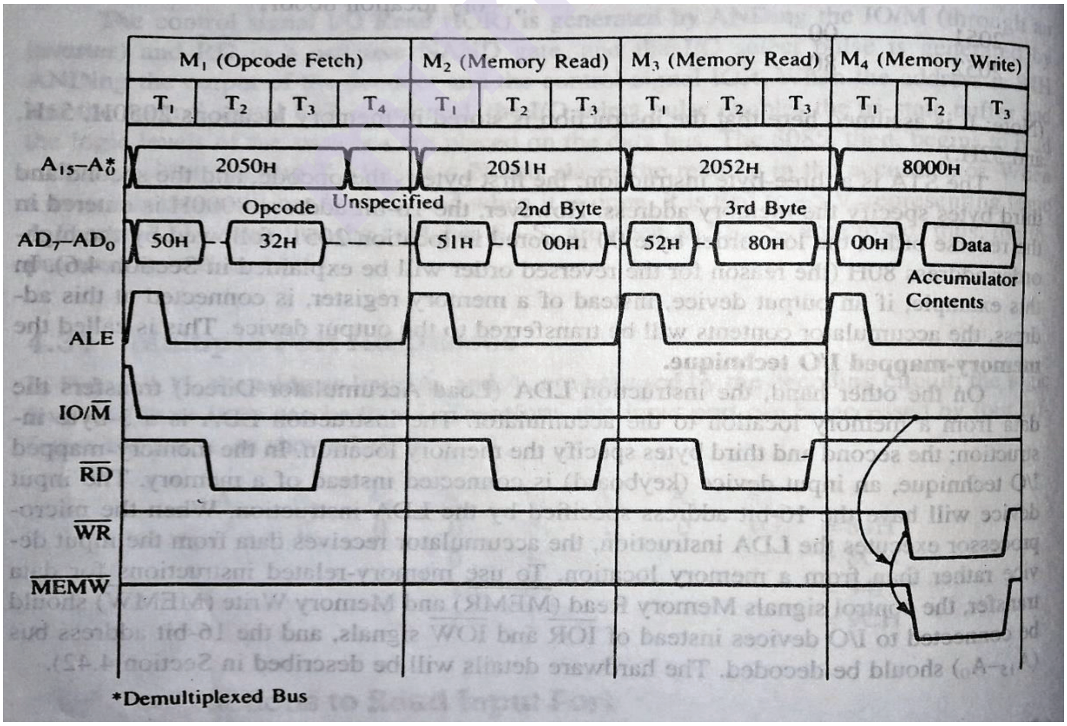


Figure 4.12 :- Timing for Execution of the Instruction STA 8000H

Device selection and data transfer in memory-mapped I/O require three steps that are similar to those required in peripheral I/O:

1. Decode the address bus to generate the device address pulse.
2. AND the control signal with the device address pulse to generate the device select (I / O select) pulse.
3. Use the device select pulse to enable the I/O port.

To interface a memory- mapped input port, we can use the instruction LDA 16- bit, which reads data from an input port with the 16-bit address and places the data in the accumulator.

The instruction has four machine cycles; only the fourth machine cycle differs from M_4 in Figure 4.12. The control signal will be $\overline{RD}$rather than R, the data flow from the input port to microprocessor.

4.5.2 Illustration: Safety Control System Using Memory-Mapped I/O Technique

Figure 4.13 shows a schematic of interfacing I/O device using the memory-mapped I/O technique. The circuit includes one input port with eight DIP switches and one output port to control various processes and gates, which are turned on/off by the microprocessor according to the corresponding switch positions.

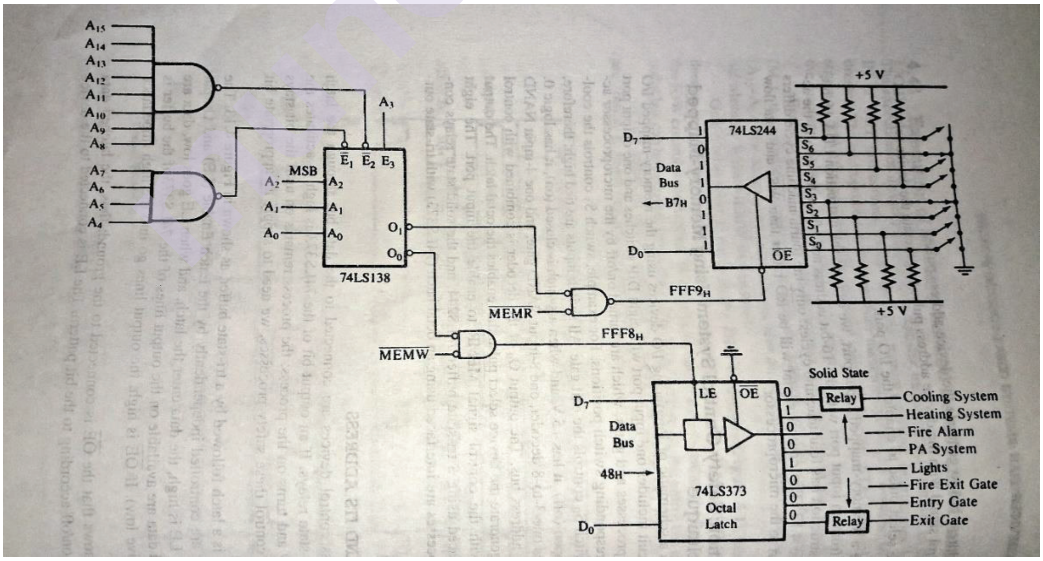


Figure 4.13 :- Memory-Mapped I/O Interfacing

For example, switch S_7 controls the cooling system, and switch S_0 controls the exit gate. All switch inputs are tied high; therefore, when switch is open (off), it has +5V and when a switch is closed (on), it has logic 0.

The circuit includes 3 – to-8 decoder, one 8- input NAND gate, and one 4-input NAND gate to decode the address bus. The output O_0 of decoder is combined with control signal $\overline{MEMW}$ to generate the device select pulse that enables the octal latch. The output O_1 is combined with the control signal ($\overline{MEMR}$) to enable the input port. The eight switches are interfaced using a tri-state buffer 74LS244, and the solid state relays controlling various processes are interfaced using an octal latch (74LS373) with tri-state output.

Output Port and its Address

The various process control devices are connected to the data bus through the latch 74LS373 and solid state relays. If an output bit of the 74Ls373 is high, it activates the corresponding relays and turns on the process; the process remains on until the bit stays high. Therefore, to control these safety processes, we need to supply an appropriate bit pattern to the latch.

The 74LS373 is a latch followed by a tri-state buffer, as shown in Figure. The latch and and the buffer are controlled independently by the Latch Enable (LE) and Output Enable (OE).

When LE is high, the data enter the latch, and when LE goes low, data are latched. The latched data are available on the output lines of the 74LS373 if the buffer is enabled by OE (active low). If OE is high, the output lines go into the high impedance state.

Figure 4.13 shows that the OE is connected to the ground; thus, the latched data will keep the relays on/ off according to the bit pattern. The LE is connected to the device select pulse, which is asserted when the output O_0 of the decoder and the control signal $\overline{MEMW}$ go low .Therefore, to assert the I/O select pulse, the output port address should be FFF8H, as shown below:

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	=FFF8 H
To 8- input NAND gate to Enable E_2								To 4-input NAND gate to Enable E_1				To Enable E_3	Decoder Input			

Input Port and its Address

The DIP switches are interfaced with the 8085 using the tri- state buffer 74LS244. The switches are tied high, and they are turned on by grounding, as shown in Figure. The switch positions can be read by enabling the signal OE, which is asserted when the output O1 of the decoder and the control signal ($\overline{MEMR}$) go low. Therefore, to read the input port, the port address should be

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	=FFF9 H
To 8- input NAND gate to Enable E_2								To 4-input NAND gate to Enable E_1				To Enable E_3	Decoder Input			

Instructions: - To control the processes according to switch positions, the microprocessor should read the bit pattern at the input port and send that bit pattern to the output port.

The following instructions can accomplish this task:

READ : LDA FFF9H ; Read the switches

CMA ; Complement switch reading, convert on- switch (logic 0) into logic 1 to turn on appliances

STA FFF8H ; Send switch positions to output port and turn on/off appliances.

JMP READ ; Go back and read again

When this program is executed, the first instruction reads the bit pattern

1011 0111 (B7H) at the input port FFF9H and places that reading in the accumulator; this bit pattern represents the on- position of switches S_6 and S_3 . The second instruction complements the reading ; this instruction is necessary because the on-position has logic 0, and to turn on solid state relays logic 1 is necessary. The third instruction sends the complemented accumulator contents (0100 1000= 48H) to the output port FFF8H. The 74LS373 latches the data byte 0100 1000 and turns on the heating system and lights. The last instruction, JMP READ, takes the program back to the beginning and repeats the loop continuously .Thus, it monitors the switches continuously.

4.5.3 Comparison of Memory-Mapped I/O and Peripheral I/O

Characteristics	**Memory-Mapped I/O**	**Peripheral I/O**
1. Device address	16- bit	8- bit
2. Control signals for Input / Output	$(\overline{MEMR})$ / MEMW	$\overline{IOR}$ / $\overline{IOW}$
3. Instruction available	Memory-related instructions such as STA; LDA; LDAX; STAX; MOV M, R: ADD M; SUB M; ANA M; etc.	IN and OUT

4. Data transfer	Between any register and I/O	Only between I/O and the accumulator
5. Maximum number of I/Os possible	The memory map (64K) is shared between I/Os and system memory	The I/O map is independent of the memory map; 256 input devices and 256 output devices can be connected.
6. Execution speed	13 T-states (STA, LDA) 7 T-states (MOV M, R)	10 T-states
7. Hardware requirements	More hardware is needed to decode 16-bit address	Less hardware is needed to decode 8-bit address
8. Other features	Arithmetic or logical operations can be directly performed with I/O data	Not available

4.6 Testing and Troubleshooting I/O Interfacing Circuits

In previous sections we illustrated how to interface an I/O device to a working microcomputer system or add an I/O port as an expansion to the existing system. In section we designed the LED output port with the address F5H. The next step is to test and verify that we can display the digit 7 by sending the code 78H as specified in the design problem. In the first attempt, the most probable outcome will be that nothing is displayed or digit 8 is displayed irrespective of the code sent to the port.

Now we need to troubleshoot the interfacing circuit. The obvious first step is to check the wiring and the pin connections. After this preliminary check, we need to generate a constant and identifiable signal and check various points in relation to that signal. We can generate such a signal by asking the processor to execute a continuous loop, called a diagnostic routine.

4.6.1 Diagnostic Routine and Machine Cycles

We can use the same instructions for the diagnostic routine that we used in the design problem; however, to generate a continuous signal, we need to add a Jump instruction, as shown next.

Instruction	Bytes	T- States	Machine Cycles		
			M_1	M_2	M_3
START : MVI A, 78H	2	7 (4, 3)	Opcode Fetch	Memory Read	
OUT F5H	3	10 (4, 3, 3)	Opcode Fetch	Memory Read	I/O Write
JMP START	3	10 (4, 3, 3)	Opcode Fetch	Memory Read	Memory Read

This loop has 27 T- states and eight operations (machine cycles). To execute the loop once, the microprocessor asserts the $\overline{RD}$ signal seven times (the Opcode Fetch is also a loop is executed in 8.9µs, and the $\overline{WR}$ signal is repeated every 8.9µs that can be observed on a scope. If we sync the scope on the $WR\overline{WR}$ pulse from the 8085, we can check the output on a scope. If we sync the scope on the $\overline{WR}$ pulse from the 8085, we can check the output of the decoder, $\overline{IOW}$, and IOSEL signals; some of these signals of a working circuit are shown in Figure 4.14

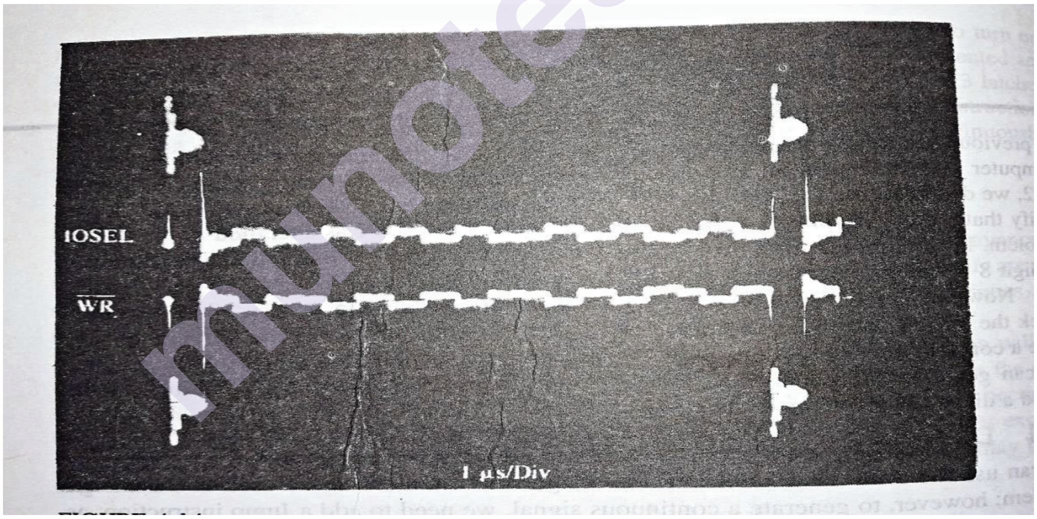


Figure 4.14 :- Timing Signals of Diagnostic Routine

When the 8085 asserts the $\overline{WR}$ signal, the port address F5H must be on the address bus A_7- A_0, and the output O_5 of the decoder in figure must be low. Similarly, the $\overline{IOW}$ must be low and the IOSEL (the output of the 74LS02) must be high. Now if we check the data bus in relation to $\overline{WR}$ signal, one line at a time, we must read the data byte 78H.If the circuit is not properly functioning, we can check various signals in reference to the $\overline{WR}$ signal as suggested below:

1. If IOSEL is low, check $\overline{IOW}$ and O_5 of the decoder .

2. If $\overline{IOW}$ is high, check the input to the OR gate 74LS32 . Both should be low.
3. If O_5 of the decoder is high, check all the output lines O_0 to O_7 of the decoder. If all of them are high, that means the decoder is not enabled. If one of the outputs of the decoder is low, it suggests that the input address lines are improperly connected.
4. If the decoder is not enabled, check the address lines A_4 – A_7 ; all of them must be high and the address line A_3 must be low.
5. Another possibility is that the port is enabled, but the seven-segment display is wrong .

The problem must be with data lines. Try different codes to display other digits. If two data lines are interchanged, you may be able to isolate these two data lines. The final step is to check all the data lines.

4.7 Summary

In this chapter, we examined the machine cycles of the OUT and IN instructions and derived the basic concepts in interfacing peripheral-mapped I/Os. Similarly, we examined the machine cycles of memory-related data transfer instructions and derived the basic concepts in interfacing memory-mapped I/Os. These concepts were illustrated with various examples of interfacing I/O devices.

Peripheral-Mapped I/O

The OUT is a two-byte instruction. It copies (transfers or sends) data from the accumulator to the addressed port.

When the 8085 executes the OUT instruction, in the third machine cycle, it places the output port address on the low-order bus, duplicates the same port address on the high-order bus, places the contents of the accumulator on the data bus, and asserts the control signal $\overline{WR}$

A latch is commonly used to interface output devices.

The IN instruction is a two-byte instruction. It copies (transfer or reads) data from an input port and places the data into the accumulator.

When the 8085 executes the IN instruction, in the third machine cycle, it places the input port address on the low-order bus, as well as on the high-order bus, asserts the control signal $\overline{RD}$, and transfers data from the port to the accumulator.

A tri-state buffer is commonly used to interface input devices.

To interface an output or an input device, the low- order address bus A7 – A0 (or high- order bus $A_{15} – A_8$) needs to be decoded to generate the device address pulse, which must be combined with the control signal $\overline{IOR}$ (or $\overline{IOW}$) to select the device.

Memory-Mapped I/O

Memory-related instructions are used to transfer data.

To interface I/O devices, the entire bus must be decoded to generate the device address pulse, which must be combined with the control signal ($\overline{MEMR}$) (or MEMW) to generate the I/O select pulse. This pulse is used to enable the I/O device and transfer the data.

Questions and Problems

1. Explain why the number of output ports in the peripheral-mapped I/O is restricted to 256 ports.
2. In the peripheral-mapped I/O, can an input port and an output port have the same port address?
3. If an output and input port can have the same 8-bit address, how does the 8085 differentiate between the ports?
4. Specify the two 8085 signals that are used to latch data in an output port.
5. What are control signals necessary in the memory-mapped I/O?
6. Can the microprocessor differentiate whether it is reading from a memory-mapped input port or from memory?
7. Specify the 8085 signals that are used to latch data in an input port.
8. Specify the type of pulse (high or low) required to latch data in the 7475.

Books and References

1. Computer System Architecture by M. Morris Mano, PHI Publication, 1998.
2. Structured Computer Organization by Andrew C. Tanenbaum, PHI Publication.
3. Microprocessors Architecture, Programming and Application with 8085 by Ramesh Gaonker, PENRAM, Fifth Edition, 2012.

❖❖❖

Unit-2

5

INTRODUCTION TO 8085 ASSEMBLY LANGUAGE PROGRAMMING

Unit Structure

5.0 Objectives

5.1 Introduction

5.2 The 8085 Programming Model

5.2.1 Programming Registers

5.3 Instruction Classification

5.3.1 The 8085 Instruction Set

5.4 Instruction and Data Format

5.4.1 Instruction Word Size

5.4.2 Opcode Formats

5.4.3 Data Format

5.5 How to Write Assemble, and Execute a Simple Program

5.5.1 Illustrate Program: Adding Two Hexadecimal Numbers

5.5.2 How Does a Microprocessor Differentiate Between Data and instruction Code?

5.6 Overview of the 8085 Instruction Set

5.7 Summary

5.0 Objectives

- Explain the various functions of the registers in the 8085 programming model.
- Define the term flag and explain how the flags are affected
- Explain the terms operation code (opcode) and operand, and illustrate these terms by writing instructions.

- Classify the instructions in terms of their word size and specify the number of memory registers required to store the instructions in memory
- List the five categories of the 8085 instruction set.
- Define and explain the term addressing mode.
- Write logical steps to solve a simple programming problem.
- Draw a flowchart from the logical steps of a given programming problem.
- Translate the flowchart into mnemonics and convert the mnemonics into Hex code for a given programming problem.

5.1 Introduction

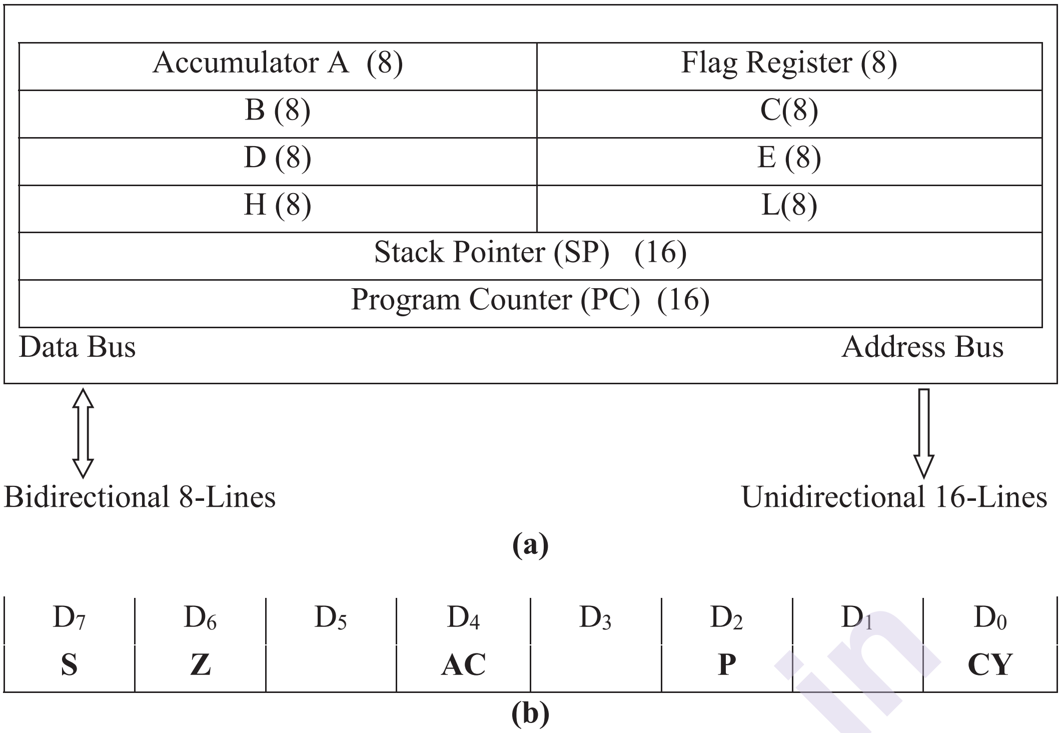
An Assembly program is a set of instructions written in the mnemonics of a given microprocessor. These instructions are commands to the microprocessor to be executed in the given sequence to accomplish a task. To write such programs for the 8085 microprocessor, we should be familiar with the programming model and the instruction set of the microprocessor.

The 8085 instruction set is classified into five different groups : data transfer, arithmetic, logic, branch, and machine control; each of these groups is illustrated with examples. It also discusses the instruction format and various addressing modes. A simple problem of adding to Hex numbers is used to illustrate writing, assembling, and executing a program. The flowcharting technique and symbols are discussed in the context of the problem. It concludes with a list of selected 8085 instructions.

5.2 THE 8085 PROGRAMMING MODEL

5.2.1 Programming Registers

The 8085 programming model includes six registers, one accumulator, and flag register, as shown in Figure 5.1.In addition,it has two 16- bit registers: the stack pointer and program counter. They are described briefly as follows.



Accumulator A (8)	Flag Register (8)
B (8)	C(8)
D (8)	E (8)
H (8)	L(8)
Stack Pointer (SP) (16)	
Program Counter (PC) (16)	

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
S	**Z**		**AC**		**P**		**CY**

Figure 5.1: 8085 Programming Model (a) and Flag Register (b)

Registers

The 8085 has six general-purpose registers to store 8- bit data; these are identified as B,C, D, E, H, and L, as shown in Figure 5.1. They can be combined as register pairs – BC, DE, and HL – to perform some 16- bit operations. The programmer can use these registers to store or copy data into the registers by using data copy instructions.

Accumulator

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

Flags

The ALU includes five flip-flops, which are set or reset after an operation according to data conditions of the result in the accumulator and other registers.

They are called Zero (Z), Carry (CY), Sign (S), Parity (P),and Auxiliary (AC) flags; they are listed in Table 5.1 and their bit positions in the flag register are shown in

Figure 5.1(a) . The most commonly used flags are Zero, Carry and Sign. The microprocessor uses these flags to test data conditions.

For example, after an addition of two numbers, if sum in the accumulator is larger than eight bits, the flip- flop used to indicate a carry – called the Carry flag (CY) is set to one.

Table 5.1: THE 8085 Flags

The following flags are set or reset after execution of an arithmetic or logic operation ; data copy instructions do not affect any flags
Z – Zero; The Zero flag is set to 1 when the result is zero ; otherwise it is reset.
CY – Carry: If an arithmetic operation results in a carry, the CY flag is set ; otherwise it is reset.
S – Sign : The Sign flag is set if bit D_7 of the result =1 ; otherwise it is reset.
P – Parity : If the result has an even number of 1s, the flag is set; for an odd number of 1s, the flag is reset.
AC – Auxiliary Carry : In an arithmetic operation, when a carry is generated by digit D_3 and passed to digit D_4, the AC flag is set. This flag is used internally for BCD (binary- coded decimal) operations ; there is no Jump instruction associated with this flag.

When an arithmetic operation results in zero, the flip- flop called the Zero (Z) flag is set to one. Figure 5.1(a) shows an 8-bit register, called the flag register, adjacent to the accumulator. However, it is not used as a register; five bit positions out of eight are used to store the outputs the five flip-flops. The flags are stored in the 8-bit register so that the programmer can examine these flags (data conditions) by accessing the register through an instruction.

These flags have critical importance in the decision- making process of microprocessor. The conditions (set or reset) of the flag are tested through software instructions.

For example, the instruction JC (Jump on Carry) is implemented to change the sequence of a program when the CY flag is set. The thorough understanding of flags is essential in writing assembly language programs.

Program Counter (PC)

This 16- bit register deals with sequencing the execution of instructions. This register is a memory pointer. Memory locations have 16- bit addresses, and that is why this is a 16- bit register.

The microprocessor uses this register to sequence the execution of the instructions.

The function of the program counter is to point to the memory address from which the next byte is to be fetched. When a byte (machine code) is being fetched, the program counter is incremented by one to point to the next memory location.

Stack Pointer (SP)

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

5.3 Instruction Classification

An instruction is a binary pattern designed inside a microprocessor to perform a specific function. The entire group of instructions called instruction set, determines what functions the microprocessor can perform. The 8085mmicroprocessor includes the instruction set of its predecessor, the 8080A, plus two additional instructions.

5.3.1 The 8085 Instruction Set

The 8085 instruction can be classified into the following five functional categories: data transfer (copy) operations, arithmetic operations, logical operations, branching operations, and machine- control operations.

Data Transfer (Copy) Opertions

This group of instruction copies data from a location called source to another location, called a destination, without modifying the contents of the source. The term data transfer is used for this copying function .However, the term transfer is misleading; it creates the impression that the contents of source are destroyed when, in fact, the contents are retained without any modification. The various types of data transfer (copy) are listed below together with examples of each type:

Table 5.2 : Data Transfer Examples

Types	Examples
Between registers	Copy the contents of register B into register D
Specific data byte to a register or a memory location	Load register B with data byte 32H

Types	Examples
Between a memory location and a register	From the memory location 2000H to register B
Between an I/O device and the accumulator	From an input keyboard to the accumulator

Arithmetic Operations

These instructions perform arithmetic operation such as addition, subtraction, increment, and decrement.

1. **Addition**: - Any 8- bit number, or the contents of a register, or the contents of a memory location can be added to the contents of the accumulator and the sum is stored in the accumulator. No two other 8- bit registers can be added directly (e.g. the contents of register B cannot be added directly to contents of register C).

   The instruction DAD is exception; it adds 16- bit data directly in register pairs.

2. **Subtraction**: - Any 8- bit number, or the contents of a register, or the contents of a memory location can be subtracted from the contents of the accumulator and the result is stored in the accumulator.

   The subtraction is performed in 2's complement, and the results, if negative, are expressed in 2's complement. No two other registers can be subtracted directly.

3. **Increment / Decrement: -** The 8- bit contents of a register or a memory location can be incremented or decremented by 1.Similarly, the 16- bit contents of a register pair (such as BC) can be incremented or decremented by 1. These increment and decrement operations differ from addition and subtraction in an important way i.e. they can be performed in any one of the registers or in a memory location.

Logical Operations

These instructions perform various logical operations with the contents of the accumulator.

1. **AND, OR, Exclusive -OR :-** Any 8- bit number, or contents of a register, or of a memory location can be logically ANDed,ORed, or Exclusive-ORed with the contents of the accumulator. The results are stored in the accumulator.

2. **Rotate: -** Each bit in the accumulator can be shifted either left or right to the next position.
3. **Compare: -** Any 8-bit number, or the contents of a register, or memory location can be compared for equality, greater than, or less than, with the contents of the accumulator.
4. **Complement: -** The contents of the accumulator can be complemented; all 0s are replaced by 1s and all 1s are replaced by 0s.

Branching Opertions

This group of instructions alters the sequence of program execution either conditionally or unconditionally.

1. **Jump:-** Conditional jumps are an important aspects of the decision-making process in programming. These instructions test for a certain condition (e.g. Zero or Carry flag) and alter the program sequence when the condition is met. In addition, the instruction set includes an instruction called unconditional jump.
2. **Call, Return, and Restart:-** These instructions change the sequence of a program either by calling a subroutine or returning from a subroutine. The conditional Call and Return instructions also can test condition flags.

Machine Control Opertions

These instructions control machine functions such as Halt, Interrupt, or do nothing.

Some important aspects of the instruction set are noted below:

1. In data transfer, the contents of the source are not destroyed; only the contents of the destination are changed. The data copy instructions do not affect the flags.
2. Any register including memory can be used for increment or decrement.
3. Arithmetic and logical operations are performed with the contents of the accumulator.
4. A program sequence can be changed either conditionally or by testing for a given data condition.

5.4 Instruction and Data Format

An instruction is a command to the microprocessor to perform a given task on specified data. Each instruction has two parts: one is the task to be performed, called the operation code (opcode) and the second is the data to be operated on, called the operand. The operand (or data) can be specified in various ways. It may include 8-bit (or 16- bit) data, an internal register, a memory location, or 8- bit (or 16- bit) address. In some instructions, the operand is implicit.

5.4.1 Instruction Word Size

The 8085 instruction set is classified into the following three groups according to word size:

1. One-word or 1-byte instructions.
2. Two-word or 2-byte instructions
3. Three-word or 3-byte instructions

In the 8085, "byte" and "word" are synonymous because it is an 8-bit microprocessor. However, instructions are commonly referred to in terms of bytes rather than words.

One- Byte Instructions

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

Table 5.3: One- Byte Instructions

Task	Opcode	Operand	Binary Code	Hex Code
Copy the contents of the accumulator in register C	MOV	C,A	0100 1111	4FH
Add the contents of register B to the accumulator	ADD	B	1000 0000	80H
Invert (complement) each bit in the accumulator	CMA		0010 1111	2FH

These instructions are 1-byte instructions performing three different tasks. In the first instruction, both operand registers are specified.

In the second instruction, the operand B is specified and the accumulator is assumed. Similarly, in the third instruction, the accumulator is assumed to be the implicit operand. These instructions are stored in 8-bit binary format in memory; each requires one memory location.

Two-Byte Instructions

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

Table 5.4 : Two-Byte Instructions

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

Assume the data byte is 32H.The assembly language instruction is written as

Mnemonics	**Hex Code**
MVI A,32H	3E 32H

This instruction would require two memory locations to store in memory.

Three-Byte Instructions

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

Table 5.5: Three-Byte Instructions

Task	Opcode	Operand	Binary Code	Hex Code	
Transfer the program sequence to the memory location 2085H	JMP	2085H	1100 0011 1100 0011 0010 0000	C3 85 20	First byte Second byte Third byte

This instruction would three memory locations to store in memory.

These commands are in many ways similar to our everyday conversation. For example, while eating in a restaurant, we may make the following requests and orders:

1. Pass (the) butter
2. Pass (the) bowl.
3. (Let us) eat.
4. I will have combination 17 (on the menu).
5. I will have what Susie ordered.

The first request specifies the exact item; it is similar to the instruction for loading a specific data byte in a register.

The second request mentions the bowl rather than the contents, even though one is interested in the contents of the bowl. It is similar to the instruction MOV C,A where registers (bowls) are specified rather than data.

The third suggestion (let us eat) assumes that one knows what to eat. It is similar to the instruction Complement, which implicitly assumes that the operand is accumulator.

In the fourth sentence, the location of the item on menu is specified and not the actual item. It is similar to the instruction: transfer the data byte from the location 2050H.

The last order (what Susie ordered) is specified indirectly. It is similar to an instruction that specifies a memory location through the contents of a register pair.

These various ways of specifying data are called the **addressing modes**. Although microprocessor instructions require one or more words to specify the operands, the notations and conventions used in specifying the operands have very little to do with the operation of the microprocessor.

The mnemonic letters used to specify a command are chosen by the manufacturer. When an instruction is stored in memory, it is stored in binary code, the only code the microprocessor is capable of reading and understanding. The conventions used in specifying the instructions are valuable in terms of keeping uniformity in different programs and in writing assemblers. The important point to remember is that the microprocessor neither reads nor understands mnemonics or hexadecimal numbers.

5.4. 2 OPCODE FORMATS

The microprocessor 8085 is an 8 bit microprocessor and has 8 bit opcodes .Each instruction has a unique opcode.

The opcode contains information regarding the operation, type of operation to be performed, registers to be used, flags. The opcode is fixed for each instruction.

Notations used in object code or OPCODE are:

Table: 5.6 (a) Opcode Formats

Notations	Meaning
ddd	Destination register(s)
sss	Source registers, ddd=sss = 111= A register = 000= B register = 001 = C register

Notations	Meaning
	= 010 = D register =011 = E register = 100 = H register = 101 = L register
nnn	Restart number 000 to 111

Table: 5.6 (b) Opcode Formats

Notation	Meaning
yy	An 8 bit binary data
yyyy	A 16 binary data unit
x	Register pair 0= BC 1= DE
xx	Register pair 00= BC 01=DE 10=HL 11=SP (if PUSH/POP)PSW
PPQQ	A 16 bit memory address

Table: 5.6 (c) Information operations

Information operations			
Io	Io	Io	Operation
0	0	0	Read/ set interrupt mask
0	0	1	Immediate operation R_p
0	1	0	Load / store
0	1	1	Increment / Decrement R_p
1	0	0	Increment single register
1	0	1	Decrement single register
1	1	0	Immediate operation on single register
1	1	1	Register shifting / Miscellaneous

Table: 5.6 (d) Arithmetic logic unit operations

Arithmetic logic unit operations			
A_L	A_L	A_L	Operation
0	0	0	ADD
0	0	1	ADD with carry
0	1	0	SUB
0	1	1	SUB with borrow
1	0	0	Logical AND
1	0	1	X-OR
1	1	0	Logical OR
1	1	1	Compare

Table: 5.6 (e) Branch condition

Branch condition			
C_B	C_B	C_B	Operation
0	0	0	JNZ
0	0	1	JZ
0	1	0	JNC
0	1	1	JC
1	0	0	JPO
1	0	1	JPE
1	1	0	JP
1	1	1	JM

Table 5.6 (f) Branch condition

Branch operation			
Bo	Bo	Bo	Operation
0	0	0	Conditional return
0	0	1	Simple return / Miscellaneous
0	1	0	Conditional Jump
0	1	1	Unconditional jump / Miscellaneous
1	0	0	Conditional CALL
1	0	1	Simple CALL / Miscellaneous
1	1	0	Special A / L operations
1	1	1	Special unconditional jump

For all data transfer instructions except MOV instruction format the opcode is,

0	0	d	d	d	Io	Io	Io
		Destination register			Information operation		

5.4.3 Data Format

The 8085 is an 8- bit microprocessor, and it processes (copy, add, subtract, etc) only binary numbers. However, the real world operates in decimal numbers and languages of alphabets and characters. Therefore, we need to code binary numbers into different media.

Let us examine coding. What is letter “A”? It is a symbol representing a certain sound in a visual medium that eyes can recognize. Similarly, we can represent or code groups of bits into different media. In 8- bit processor systems, commonly used codes and data formats are ASCII, BCD, signed integers, and unsigned integers.

ASCII Code – This is a 7 bit alphanumeric code that represents decimal numbers, English alphabets, and nonprintable characters such as carriage return. Extended ASCII is an 8- bit code.

The additional numbers (beyond 7- bit ASCII code) represent graphical characters.

BCD Code – The term BCD stands for binary-coded decimal; it is used for decimal numbers. The decimal numbering system has ten digits, 0 to 9. Therefore, we need only four bits to represent ten digits from 0000 to 1001. The remaining numbers, 1010 (A) to 1111(F), are considered invalid. An 8- bit register in thev8085 can accommodate two BCD numbers.

Signed Integer - A signed integer is either a positive number or a negative number. In an 8- bit processor, the most significant digit D_7, is used for the sign ; 0 represents the positive sign and 1 represent the negative sign. The remaining seven bits $D_6 – D_0$, represent the magnitude of an integer. Therefore, the largest positive integer that can be processed by the 8085 at one time is 0111 1111 (7FH) ; the remaining Hex number in this microprocessor are represented in 2’s complement format.

Unsigned Integers- An integer without a sign can be represented by all the 8 bit in a microprocessor register. Therefore, the largest number that can be processed at one time is FFH. However, this does not imply that the 8085 microprocessor is limited to handling only 8- bit numbers.

Numbers larger than 8 bits (such as 16-bit or 24-bit numbers) are processed by dividing them in groups of 8 bits.

Now let us examine how the microprocessor interrupts any number. Let us assume that after performing some operations the result in the accumulator is 0100 0001 (41H) . This number can have many interpretations:

(1) It is an unsigned number equivalent to 65 in decimal.

(2) It is BCD number representing 41 decimal

(3) It is the ASCII capital letter "A" or

(4) It is group of 8 bits where bits D_6 and D_0 turn on and the remaining bits turn off output devices.

The processor processes binary bits; it is up to the user to interpret the result. In our example, the number 41H can be displayed on a screen as an ASCII "A" or 41 BCD.

5.5 How to Write Assemble, and Execute a Simple Program

A Program is a sequence of instructions written to tell a computer to perform a specific function. The instructions are selected from the instruction set of the microprocessor. To write a program, divide a given problem in small steps in terms of the operations the 8085 can perform, then translate these steps into instructions. Writing a simple program of adding two numbers in the 8085 language is illustrated below.

5.5.1 Illustrate Program: Adding Two Hexadecimal Numbers

Problem Statement

Write instructions to load the two hexadecimal numbers 32H and 48H in registers A and B respectively. Add the numbers, and display the sum at the LED output port PORT1.

Problem Analysis

Even though this is a simple problem, it is necessary to divide the problem into small steps to examine the process of writing programs. The wording of the problem provides sufficient clues for the necessary steps. They are as follows:

1. Load the numbers in the registers.

2. Add the numbers.

3. Display the sum at the output port PORT1.

Flowchart

The steps listed in the problem analysis and the sequence can be represented in a block diagram, called a flowchart. Figure shows such a flowchart representing the above steps. This is a simple flowchart, and the steps are self-explanatory.

Assembly Language Program

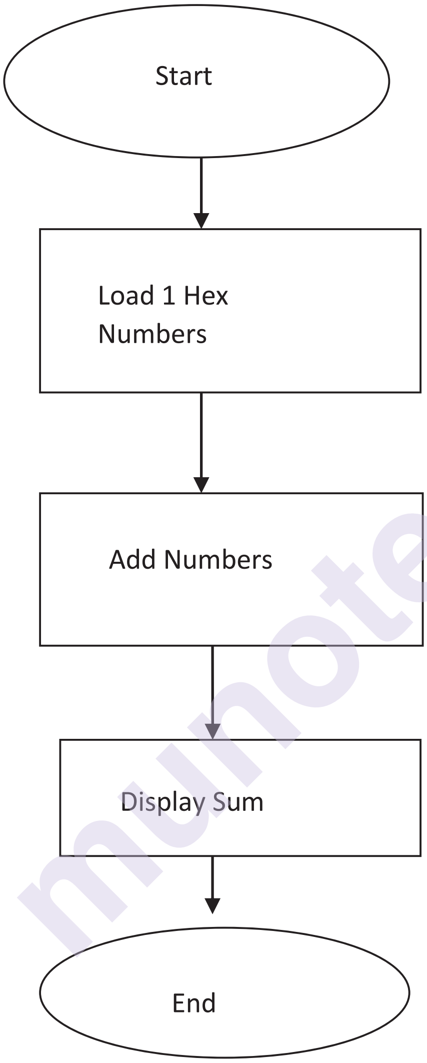


Figure 5.2: Flowchart: Adding Two Numbers

To write an assembly language program, we need to translate the blocks shown in the flowchart into 8085 operations and then, subsequently into mnemonics. By examining the blocks, we can classify them into three types of operations: Block 1 and 3 are copy operations ; Block 2 in an arithmetic operation ; and Block 4 is a machine-control operation. To translate these steps into assembly and machine languages, you should review thee instruction set.

The translation of each block into mnemonics with comments is shown as follows:

Block 1: MVI A, 32H Load register A with 32H
MVI B, 48H Load register B with 48H

Block 2: ADD B Add two bytes and save the sum in A

Block 3: OUT 01H Display accumulator contents at port 01H

Block 4 : HALT End

From Assembly Language to Hex Code

To convert the mnemonics into Hex code, we need to look up the code in the 8085 instruction set; this is called either manual or hand assembly.

Mnemonics	Hex Code	
MVI A,32H	3E 32	2- byte instruction
MVI B, 48H	06 48	2- byte instruction
ADD B	80	1- byte instruction
OUT 01H	D3 01	2- byte instruction
HLT	76	1- byte instruction

Storing In Memory and Converting From Hex Code to Binary Code

To store the program in R/ W memory of a single-board microprocessor and display the output, we need to know the memory addresses and the output port address.

Let us assume that R/W memory ranges from 2000H to 20FFH, and the system has an LED output port with the address 01H. Now, to enter the program:

1. Reset the system by pushing the RESET key.

2. Enter the first memory address using the Hex keys where the program should be stored.

   Let us assume it is 2000H.

3. Enter each machine code by pushing Hex keys. For example, to enter the first machine code, push the 3, E, and STORE keys. (The STORE key may be labelled differently in different systems.) When you push the STORE key, the program will store the machine code in memory location 2000H and upgrade the memory address to 2001H.

4. Repeat Step 3 until the last machine code, 76H.

5. Reset the system.

Now the question is : How does the Hex code get converted into binary code?

The answer lies with the Monitor program stored in Read- Only memory (or EPROM) of the microcomputer system. An important function of the Monitor program is to check the keys and convert Hex code into binary code. The entire process of manual assembly is shown in Figure

In this illustrate example, the program will be stored in memory as follows:

Mnemonics	**Hex Code**	**Memory Contents**	**Memory Address**
MVI A, 32H	3E 32	0011 1110 0011 0010	2000 2001
MVI B,48H	06 48	0000 0110 0100 1000	2002 2003
ADD B	80	1000 0000	2004
OUT 01H	D3 01	1101 0011 0000 0001	2005 2006
HLT	76	0111 1110	2007

This program has eight machine codes and will require eight memory locations to store the program. The critical concept that needs to be emphasized here is that the microprocessor can understand and execute only the binary instructions (or data) everything else (mnemonics, Hex code, comments) is for the convenience of human being.

Executing the Program

To execute the program, we need to tell the microprocessor where the program begins by entering the memory address 2000H. Now we can push the Execute key (or the key with a similar label) to begin the execution. As soon as the Execute function key is pushed, the microprocessor loads 2000H in the program counter and the program control is transferred from the Monitor program to our program.

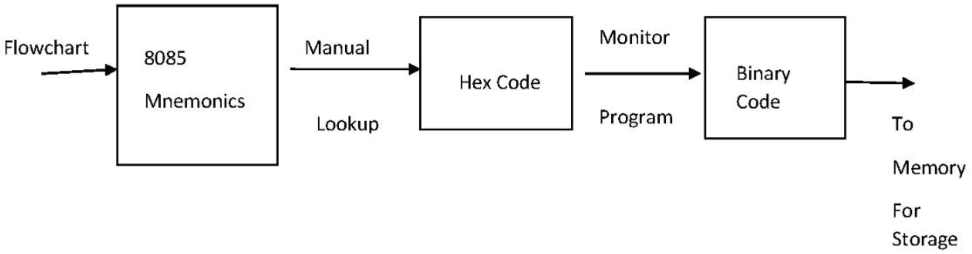


Figure 5.3: Manual Assembly Process

The microprocessor begins to read one machine code at a time, and when it fetches the complete instruction, it executes that instruction. For example, it will fetch the machine codes stored in memory locations 2000H and 2001H and execute the instruction MVI A, 32H; thus it will load 32H in register A. The ADD instruction will add the two numbers, and the OUT instruction will display the answer 7A (32H +48H= 7A) at the LED port. It continues to execute instructions until it fetches the HLT instruction.

Recogning the Number of Bytes in An Instruction

Students who are introduced to an assembly language for the first time should hand assemble at least a few small programs. Such exercises can clarify the relationship among instruction codes, data, memory registers, and memory addressing. One of the stumbling blocks in hand assembly is in recognizing the number of bytes in a given instruction. The following clues can b used to recognize the number of bytes in an instruction of the 8085 microprocessor.

1. One- byte instruction – A mnemonic followed by a letter (or two letters) representing the registers (such as A,B, C, D, E,H,L,M, and SP) is a one-byte instruction.

   Instructions in which registers are implicit are also one- byte instructions.

   Examples: (a) MOV A,B; (b) DCX SP (c) RRC

2. Two- byte instruction – A mnemonic followed by 8-bit (byte) is a two- byte instruction.

   Examples: (a) MVI A, 8-bit; (b) ADI 8-bit

3. Three-byte instruction- A mnemonic followed by 16-bit (also terms such as adr or dble) is a three-byte instruction.

In writing assembly language programs, we can assign memory addresses in a sequence once we know the number of bytes in a given instruction. For example, a three-byte instruction has three Hex codes and requires three memory locations in a sequence.

In hand assembly, omitting a byte inadvertently can have a disastrous effect on program execution, as explained in the next section.

5.5.2 How Does a Microprocessor Differentiate Between Data and Instruction Code?

The microprocessor is a sequential machine. As soon as a microprocessor- based system is turned on, it begins the execution of the code in memory. The execution continues in a sequence, one code after another (one memory location after another) at the speed of its clock until the system is turned off (or the clock stops) . If an unconditional loop is set up in a program, the execution will continue until the system is either reset or turned off.

Now a puzzling questions is : How does the microprocessor differentiate between a code and data when both are binary numbers ?

The answer lies in the fact that the microprocessor interprets the first byte it fetches as an opcode.

When the 8085 is reset, its program counter is cleared to 0000H and it fetches the first code from the location 0000H

In the example of the previous section, we tell the processor that our program begins at location 2000H. The first code it fetches is 3EH. When it decodes that code, it knows that it is a two- byte instruction. Therefore, it assumes that the second code, 32H, is a data byte. If we forgot to enter 32H and enter the next code, 06H, instead, the 8085 will load 06H in the accumulator, interpret the next code, 48H, as an opcode, and continue the execution in sequence. As a consequence, we may encounter a totally unexpected result.

5.6 Overview of the 8085 Instruction Set

The 8085 microprocessor instruction set has 74 operation codes that result in 246 instructions. The set includes all the 8080A instructions plus two additional instructions (SIM an RIM, related to serial I/O) .

The following notations are used in the description of the instructions.

R= 8085 8- bit register (A, B, C, D, E, H, L)
M=Memory register (location)
Rs= Register Source (A, B, C, D, E, H, L)
Rd= Register Destination (A, B, C, D, E, H, L)
Rp= Register Pair (BC, DE, HL, SP)
( ) = Contents of

1. Data transfer (copy) instructions:
From register to register
Load an 8-bit number in a register
Between memory and register
Between I/O and accumulator
Load 16-bit number in a register pair

Table 5.8 : Data transfer (copy) instructions

Mnemonics	Tasks
1. MOV Rd,Rs	Copy data from source register Rs into destination register Rd
2. MVI R, 8-bit	Load 8-bit data in a register
3. OUT 8-bit (port address)	Send (write) data byte from accumulator to an output device
4. IN 8-bit (port address)	Accept (read) data byte from an input device and place it in the accumulator.
5. LXI Rp, 16-bit	Load 16- bit in a register pair
MOV R,M	Copy the data byte from a memory location (source) into a register
LDAX Rp	Copy the data byte into the accumulator from a memory location indicated by a register pair.
LDA 16-bit	Copy the data byte into the accumulator from a memory location specified by 16- bit address.
MOV M,R	Copy the data byte from register into memory location.
STAX Rp	Copy the data byte from the accumulator into the memory location indicated by a register pair.
STA 16-bit	Copy the data byte from the accumulator in the memory location specified by 16- bit address

2. Arithmetic instructions:

Add
Subtract
Increment (Add 1)
Decrement (Subtract 1)

Table 5.9: Arithmetic instructions

Mnemonics	Tasks
ADD R	Add the contents of a register to the contents of the accumulator
ADI 8-bit	Add 8-bit data to the contents of the accumulator
SUB R	Subtract the contents of a register from the contents of the accumulator.
SUI 8-bit	Subtract 8-bit data from the contents of the accumulator.
INR R	Increment the contents of a register

Mnemonics	Tasks
DCR R	Decrement the contents of a register
INX Rp	Increment the contents of a register pair
DCX Rp	Decrement the contents of a register pair
ADD M	Add the contents of a memory location to the contents of the accumulator
SUB M	Subtract the contents of a memory location from the contents of the accumulator.
INR M	Increment the contents of a memory location.
DCR M	Decrement the contents of a memory location.

3. Logical instructions:

AND
OR
X-OR
Compare
Rotate

Table 5.10: Logical instructions

Mnemonics	Tasks
ANA R/M	Logically AND the contents of register/memory with the contents of the accumulator.
ANI 8-bit	Logically AND the 8-bit data with the contents of the accumulator.
ORA 8-bit	Logically OR the contents of register/memory with the contents of the accumulator.
ORI 8-bit	Logically OR the 8-bit data with the contents of the accumulator.
XRA 8-bit	Exclusive-OR the contents of register/memory with the contents of the accumulator.
XRI 8-bit	Exclusive-OR the 8-bit data with the contents of the accumulator.
CMA	Complement the contents of the accumulator.
RLC	Rotate each bit in the accumulator to left position.
RAL	Rotate each bit in the accumulator including the carry to the left position.
RRC	Rotate each bit in the accumulator including the carry to the right position.

Mnemonics	Tasks
RAR	Rotate each bit in the accumulator including the carry to the right position.
CMP R/M	Compare the contents of register/ memory with the contents of the accumulator for less than, equal to, or more than.
CPI 8-bit	Compare 8-bit data with the contents of the accumulator for less than, equal to, or more than.

4. Branch Instructions:

Change the program sequence unconditionally.

Change the program sequence if specified data conditions are met.

Table 5.11: Branch Instructions

Mnemonics	Tasks
JMP 16- bit address	Change the program sequence to the location specified by the 16- bit address.
JZ 16- bit address	Change the program sequence to the location specified by the 16-bit address if the Zero flag is set.
JNZ 16-bit address	Change the program sequence to the location specified by the 16- bit address if Zero flag is reset.
JC 16-bit address	Change the program sequence to the location specified by the 16- bit address if Carry flag is set.
JNC 16-bit address	Change the program sequence to the location specified by the 16- bit address if Carry flag is reset.
CALL 16- bit address	Change the program sequence to the location of a subroutine.
RET	Return to the calling program after completing the subroutine sequence.

5. Machine Control instructions:

Table 5.12: Machine Control instructions

Mnemonics	Tasks
HLT	stop processing and wait.
NOP	Do not perform any operation.

This set of instructions is a representative sample; it does not include various instructions related to 16- bit data operations, additional Jump instructions, and conditional Call and Return instructions.

5.7 Summary

This chapter described the data manipulation functions of the 8085 microprocessor, provided an overview of the instruction set, and illustrated the execution of instructions in relation to the system's clock. The important concepts in this chapter can be summarized as follows.

The 8085 microprocessor operations are classified into five major groups: data transfer (copy), arithmetic, logic, branch, and machine control.

An instruction has two parts: opcode (operation to be performed) and operand (data to be operated on).

The operand can be data (8-bit or 16- bit), address, or register, or it can be implicit. The method of specifying an operand (directly, indirectly, etc.) is called the addressing mode.

The instruction set is classified in three groups according to the word size : 1,2,3-byte instructions.

To write an assembly language program, divide the given problem into small steps in terms of the microprocessor operations, translate these steps into assembly language instructions, and then translate them into the 8085 machine code.

Questions and Programming Assignments

1. List the four categories of 8085 instructions that manipulate data.
2. Define opcode and operand, and specify the opcode and the operand in the instruction MOV H,L.
3. Find the Hex codes for the following instructions, identify the opcodes and operands, and show the order of entering the codes in memory.

   a. STA 2050H b. JNZ 2070H

4. Find the Hex machine code for the following instructions from the instruction set, and identify the number of bytes of each instruction.

```
MVI B, 4FH ; Load the first byte
MVI C,78H; Load the second byte
MOV A,C ; Get ready for addition
ADD B;  Add two bytes
OUT 07H ; Display the result at port 7
HLT : End the program
```

5. Assemble the following program, starting at location 2000H.

   START: IN F2H; Read input switches at port F2H
   CMA; Set ON switches to logic 1.
   JZ START ; Go back and read input port if all switches are off.

6. Write logical steps to add the following two Hex numbers. Both the numbers should be saved for future use. Save the sum in the accumulator.

   Numbers: A2H and 18H

7. Data byte 28H is stored in register B and data byte 97H is stored in the accumulator. Show the contents of register B, C, and the accumulator after the execution of the following two instructions:

   MOV A,B

   MOV C,A

Books and References

1. Computer System Architecture by M. Morris Mano, PHI Publication, 1998.
2. Structured Computer Organization by Andrew C. Tanenbaum, PHI Publication.
3. Microprocessors Architecture, Programming and Application with 8085 by Ramesh Gaonker, PENRAM, Fifth Edition, 2012.

❖❖❖

Unit-2

6

INTRODUCTION TO 8085 INSTRUCTIONS

Unit Structure

6.0 Objectives

6.1 Introduction

6.2 Data Transfer (Copy) Opertions

6.2.1 Addressing Modes

6.2.2 Illustrative Program: Data Transfer – From Register to Output Port

6.2.3 Illustrative Program: Data Transfer to Control Output Devices

6.3 Arithmetic Operations

6.3.1 Addition

6.3.2 Illustrate Program: Arithmetic Operations –Addition and Increment

6.3.3 Subtraction

6.3.4 Illustrative Program: Subtraction of Two Unsigned Numbers

6.4 Logic Operations

6.4.1 Logic and

6.4.2 Illustrative Program: Data Masking with Logic And

6.4.3 Or, Exclusive-Or and Not

6.4.4 Setting and Resetting Specific Bits

6.4.5 Illustrative Program : Oring Data from Two Input Ports

6.5 Branch Operations

6.5.1 Unconditional Jump

6.5.2. Illustrative Program: Unconditional Jump to Set Up A Continuous Loop

6.5.3 Conditional Jumps

6.5.4 Illustrative Program: Testing of the Carry Flag

6.6 Writing Assembly Language Programs

6.6.1 Getting Started

6.6.2 Illustrative Program: Microprocessor – Controlled Manufacturing Process

6.6.3 Documentation

6.7 Debuging A Program

6.7.1 Debugging Machine Code

6.8 Summary

6.0 Objectives

- Explain the functions of data transfer (copy) instructions and how the contents of the source register and destination register are affected.
- Explain the Input/ output instructions and port addresses.
- Explain the functions of the machine control instructions HLT and NOP
- Recognize the addressing modes of the instructions
- Draw a flowchart of a simple program
- Write a program in 8085 mnemonics to illustrate an application of data copy instructions, and translate those mnemonics manually into their Hex code.
- Explain the arithmetic instructions, and recognize the flags that are set or reset for given data conditions.
- List the important steps in writing and troubleshooting a simple program.

6.1 Introduction

A microcomputer performs a task by reading and executing the set of instructions written in its memory. This set of instructions, written in a sequence, is called a **program.** Each instruction in the program is a command, in binary, to the microprocessor to perform an operation. This chapter introduces 8085 basic instructions, their operations, and their applications.

It is concerned with using instructions within the constraints and capabilities of its registers and the bus system. A few instructions are introduced from each of the five groups (Data Transfer, Arithmetic, Logical, Branch, and Machine Control) and are used to write simple programs to perform specific tasks.

The simple illustrative programs given in this chapter can be entered and executed on the single-board microcomputers used commonly in laboratories.

6.2 Data Transfer (Copy) Opertions

One of the primary functions of the microprocessor is copying data, from a register (or I/O or memory) called the source, to another (or I/O or memory) called the destination. The copying function is frequently labelled as the **data transfer function.**

MOV : Move Copy a data byte.

MVI : Move Immediate Load a data byte directly.

OUT : Output to Port Send a data byte to an output device.

IN : Input from Port Read a data byte from an input device.

The term copy is equally valid for input /output functions because the contents of the source are not altered. However, the term data transfer is used so commonly to indicate the data copy function, these terms are used interchangeably when the meaning is not ambiguous.

In addition to data copy instructions, it is necessary to introduce two-machine control operations to execute programs.

HLT : Halt Stop processing and wait.

NOP : No Operation Do not perform any operation.

Instructions

The data transfer instructions copy data from a source into a destination without modifying the contents of the source. The previous contents of the destination are replaced by the contents of the source.

In the 8085 processor, data transfer instructions do not affect the flags.

Table 6.1: Data Transfer (Copy) Operations

Opcode	Operand	Description
MOV	Rd, Rs	Move This is a 1-byte instruction Copies data from source register Rs to destination register Rd

Opcode	Operand	Description
MVI	R, 8- bit	Move Immediate This is a 2-byte instruction Loads the 8 bits of the second byte into the register specified.
OUT	8-bit port address	Output to Port This is a 2-byte instruction Sends (copies) the contents of the accumulator (A) to the output port specified in the second byte.
IN	8-bit port address	Input from Port This is a 2-byte instruction Accepts (reads) data from the input port specified in the second byte, and loads into the accumulator.
HLT		Halt This is a 1-byte instruction The processor stops executing and enters wait state The address bus and data bus are placed in high impedance state. No register contents are affected.
NOP		No Operation This is 1-byte instruction No operation is performed. Generally used to increase processing time or substitute in place of an instruction. When an error occurs in a program and an instruction needs to eliminated, it is more convenient to substitute NOP than to reassemble the whole program.

Example 6.1:- Load the accumulator A with data byte 82H (H as Hexadecimal number) and save the data in register B.

Instructions: MVI A, 82H
MOV B, A

The first instruction is a 2-byte instruction that loads the accumulator with the data byte 82H, and the second instruction MOV B, A copies the contents of the accumulator in register B without changing the contents of the accumulator.

Example 6.2 :- Write instructions to read eight ON/OFF switches connected to the input port with the address 00H, and turn on the devices connected to the output port with the address 01H, as shown in Figure 6.1

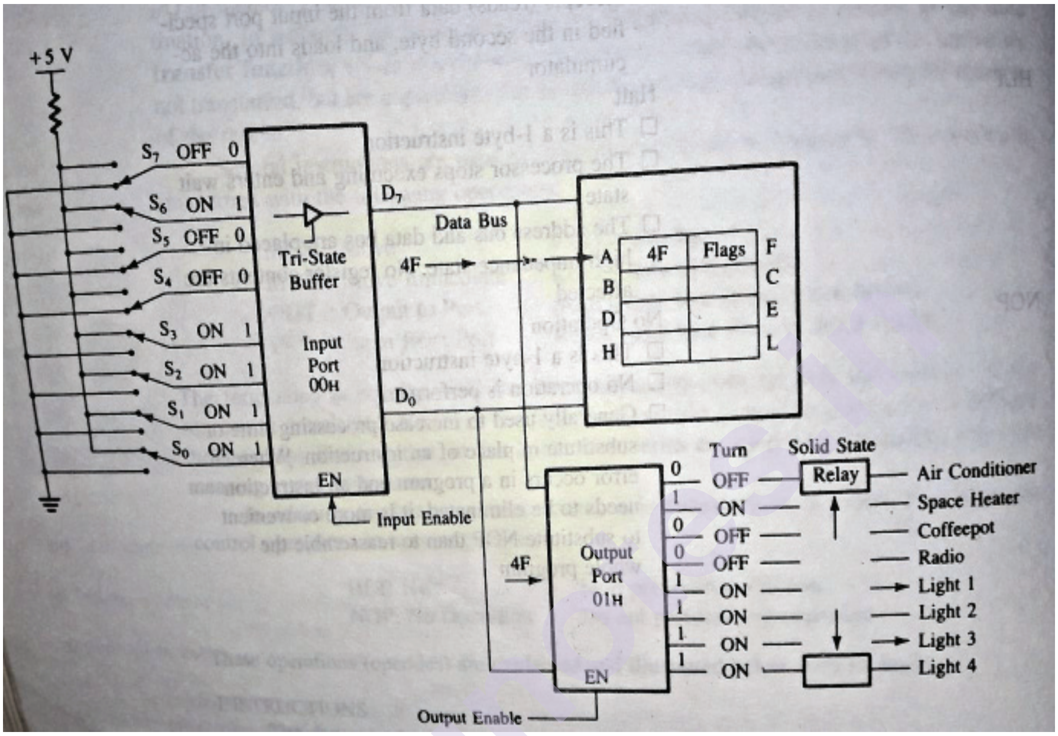


Figure 6.1: Reading Data at Input Port and Sending Data to Output Port

Solution: - The input has eight switches that are connected to the data bus through the tri-state buffer. Any one of the switches can be connected to +5V (logic 1) or to ground (logic 0), and each switch controls the corresponding device at the output port. The microprocessor needs to read the bit pattern on the switches and send the same bit pattern to the output port to turn on the corresponding devices.

Instructions: IN 00H
OUT 01H
HLT

When the microprocessor executes the instruction IN 00H, it enables the tri-state buffer. The bit pattern 4FH formed by the switch positions is placed on the data bus and transferred to the accumulator. This is called reading an input port.

When the microprocessor executes the next instruction, OUT 01H, it places the contents of the accumulator on the data bus and enables the output port 01H.The output port latches the bit pattern and turns ON/ OFF the devices connected to port according to the pattern. In the Figure 6.1, the bit pattern 4FH will turn on the devices connected to the output port data lines D_6, D_3, $D_{2,}$ D_1, and D_0 . The space heater and four light bulbs. To turn off some of the devices and turn off some of the devices and turn on the other devices, the bit pattern can be modified by changing the switch positions.

For example, to turn on the radio and the coffeepot and turn off all other devices, the switches S_4 and S_5 should be on the others should be off. The microprocessor will read the bit pattern 0011 0000, and this bit pattern will turn on the radio and the coffeepot and turn off other devices.

The preceding explanation raises two questions:

1. What are second bytes in the instructions IN and OUT?

2. How are they determined?

In answer to the first question, the second bytes are I/O port addresses. Each I/O port is identified with a number or an address similar to the postal address of a house. The second byte has eight bits, meaning 256 (2^8) combinations; thus 256 input ports and 256 output ports with the addresses from 00H to FFH can be connected to the system.

The answer to the second question depends on the logic circuit used to connect and identify a port by the system designer.

6.2.1 Addressing Modes

The above instructions are commands to the microprocessor to copy 8-bit data from a source to destination. In these instruction source can be a register, an input port, or an 8-bit number (00H to FFH) .Similarly, a destination can be register or an output port.

The various formats of specifying the operands are called the addressing modes.

1. Immediate Addressing – MVI R, Data

2. Register Addressing - MOV Rd, Rs

3. Direct Addressing - IN / OUT Port#

The classification of the addressing modes is unimportant, except that it provides some clues in understanding mnemonics.

For example, in the case of the MVI opcode, the letter I suggests that the second byte is data and not a register. What is important is to become familiar with the instructions.

6.2.2 Illustrative Program: Data Transfer – From Register to Output Port

Problem Statement

Load the hexadecimal number 37H in register B, and display the number at the output port labelled PORT1.

Problem Analysis

Even though this is a very simple problem it is necessary to break the problem into small steps and outline the thinking process in terms of the tasks.

STEPS

Step 1: Load register B with a number.

Step 2: Send the number to the output port.

Questions to be asked

Is there an instruction to load the register B? YES - MVI B.

Is there an instruction to send the data from register B to the output port? No Review the instruction OUT. This instruction sends data from the accumulator to an output port.

The solution appears to be as follows: Copy the number from register B into accumulator A.

Is there an instruction to copy data from one register to another register ? YES – MOV Rd, Rs.

Flowchart

The thinking process described here and the steps necessary to write the program can be represented in a pictorial format, called a flowchart. Figure 6.2 describes the preceding steps in a flowchart.

Flowcharting is an art. The flowchart in Figure 6.2 does not include all the steps described earlier. Although the number of steps that should be represented in a flowchart is ambiguous, not all of them should be included. It should represent a logical approach and sequence of steps in solving the problem.

A flowchart is a similar to the block diagram of a hardware system or to the outline of a chapter. Information in each block of the flowchart should be similar to heading of a paragraph.

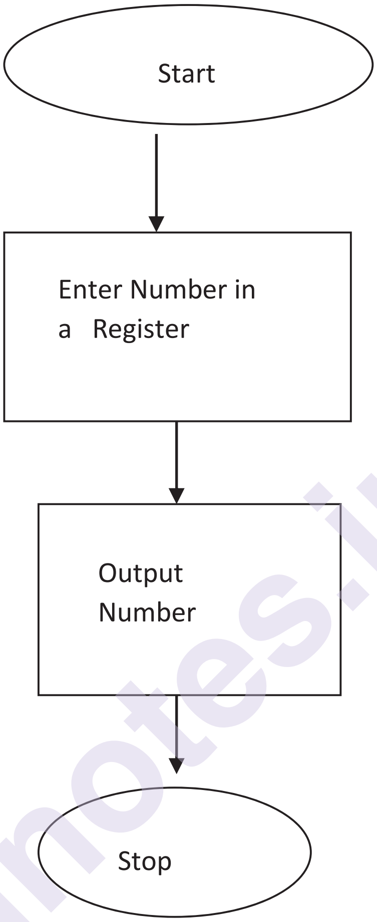
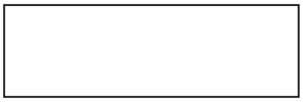
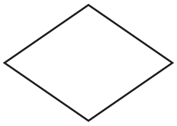
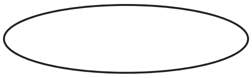


Figure 6.2 : Flowchart

Symbol commonly used in flowcharting are shown in Figure 6.3. Two types of symbols – rectangles and ovals – are already illustrated in Figure 6.2 .

	Arrow: Indicates the direction of the program execution
	Rectangle: Represents a process or an operation
	Diamond : Represents a decision-making block
	Oval : Indicates the beginning or end of a program

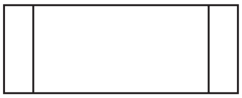
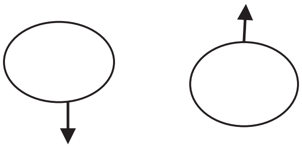
	Double-sided rectangle : Represents a predefined process such as a subroutine
	Circle with an arrow: Represents continuation (an entry or exit) to a different page.

Figure 6.3: Flowcharting Symbols

The diamond is used with Jump instructions for decision making and the double-sided rectangle is used for subroutine.

The flowchart in Figure 6.2 includes what steps to do and in what sequence. As a rule, a general flowchart does not include how to perform these steps or what registers are being used.

Assembly Language Program

Tasks	8085 Mnemonics
1. Load register B with 37H	MVI B, 37H
2.Copy the number from B to A	MOV A, B
3. Send the number to output –port 01H	OUT PORT1
4. End the program	HLT

Translation from Assembly Language to Machine Language

Now, to translate the assembly language program into machine language, look up the hexadecimal machine codes for each instruction in the 8085 instruction set and write each machine code in the sequence, as follows :

8085 Mnemonics	Hex Machine Code
1.MVI B, 37H	06 37
2.MOV A, B	78
3.OUT PORT1	D3 01
4.HLT	76

This program has six machine codes and will require six bytes of memory to enter the program into your system. If your single-board microprocessor has R/W memory starting at the address 2000H, this program can be entered in the memory locations 2000H to 2005H. The format generally used to write an assembly language program is shown below.

Program Format

Memory Address (Hex)	Machine Code (Hex)	Instruction Opcode	Operand	Comments
XX00	06	MVI	B, 37H	Load register B with data 37H
XX01	37			
XX02	78	MOV	A, B	Copy(B) into (A)
XX03 XX04	D3 PORT1	OUT	PORT1	Display accumulator contents (37H) at Port1
XX05	76	HLT		End of the program

This program has five columns: Memory Address, Machine Code, Opcode, Operand and Comments.

Memory Address These are 16-bit addresses of the user (R/W) memory in the system. Where the machine code of the program is stored. The beginning address is shown as XX00; the symbol XX represents the page number of the available R/W memory in the microcomputer, and 00 represents the line number.

Machine Code

The monitor program, which is stored in Read-only memory (ROM) of the microcomputer, translates the Hex number into binary digits and stores the binary digits in the R/W memory.

If the system has R/W memory with the starting address at 2000H and the output port address 01H, the program will be stored as follows :

Memory Address	Memory Contents	Hex Code
2000	0000 0110	06
2001	0011 0111	37
2002	0111 1000	78
2003	1101 0011	D3
2004	0000 0001	01
2005	0111 0110	76

Opcode (Operation Code) :- An instruction is divided into two parts : Opcode and Operand . Opcodes indicate the type of operation or function that will be performed by the machine code.

Operand: - The operand part of an instruction specifies the item to be processed; it can be 8- bit or 16-bit data, a register, or a memory address.

An instruction, called mnemonic is formed by combining an opcode and an operand.

The mnemonics are used to write programs in the 8085 assembly language; and then mnemonics in these programs are translated manually into the binary machine code by looking up in the instruction set.

Comments The comments are written as a part of the proper documentation of a program to explain or elaborate the purpose of the instruction used.

These are separated by a semicolon (;) from an instruction on same line.

How to Enter and Execute The Program

This program assumes that one output port is available on your microcomputer system. The program cannot be executed without modification if your microcomputer has no independent output ports other than the system display of memory address and data or if it has programmable I/O ports.

1. Push the Reset key.
2. Enter the 16-bit memory address of the first machine code of your program.
3. Enter and store all the machine codes sequentially, using the hexadecimal keyboard on your system.
4. Reset the system.
5. Enter the memory address where the program begins and push the Execute key.

If the program is properly entered and executed, the data byte 37H will be displayed.

How to Execute a Program without an Output Port

If your system does not have an output port, either eliminate the instruction OUT PORT1, or substitute NOP (No Operation) in place of the OUT instruction. Assuming your system has R/W memory starting at 2000H, you can enter the program as follows:

Memory Address	**Machine Code**	**Mnemonic Instruction**
2000	06	MVI B, 37H
2001	37	
2002	78	MOV A, B
2003	00	NOP
2004	00	NOP
2005	76	HLT

After you have executed this program, you can find the answer in the accumulator by pushing the Examine Register Key

The program also can be executed by entering the machine code 76 in location 2003H, thus eliminating the OUT instruction.

6.2.3 Illustrative Program: Data Transfer to Control Output Devices

Problem Statement

A microcomputer is designed to control various appliances and lights in your house. The system has an output port with the address 01H, and various units are connected to the D_7 to D_0 as shown in Figure 6.4. On a cool morning you want to turn on the radio, the coffeepot, and space heater. Write appropriate instructions for the microcomputer.

Assume the R/W memory in your system begins at 3000H.

Problem Analysis

The output port in Figure 6.4 is a latch (D flip-flop) .When data bits are sent to the output port they are latched by the D flip-flop. A data bit at logic 1 supply approximately 5V as output and can turn on solid-state relays.

To turn on the radio, the coffeepot and the space heater set D_6, D_5, and D_4 at logic 1, and the other bits at logic 0.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	
0	1	1	1	0	0	0	0	= 70H

The output port requires 70H, and it can be sent to the port by loading the accumulator with 70H.

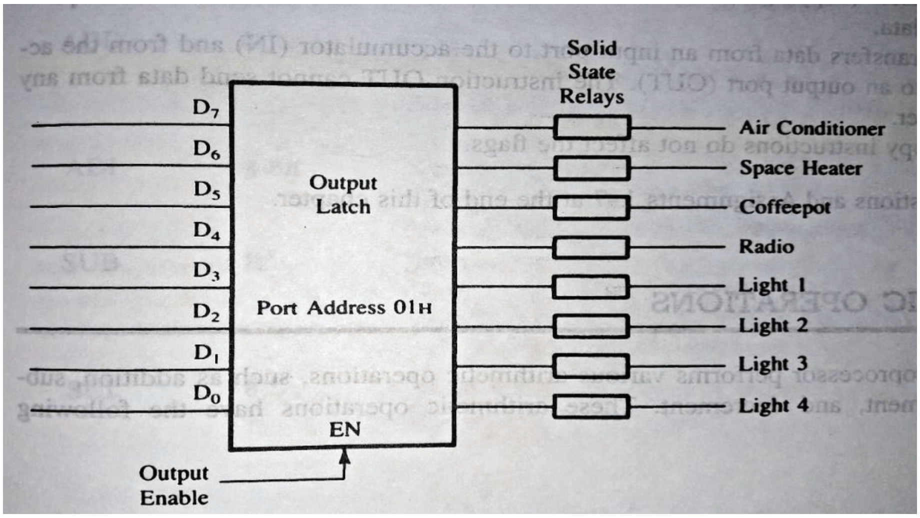


Figure 6.4: Output Port to Control Devices

Program

Memory Address	Machine Code	Mnemonic Instruction	Comments
3000	3E	MVI A, 70H	Load the accumulator with the bit pattern necessary to turn on the devices
3001	70		
3002	D3	OUT 01H	Send the bit pattern to the port 01H, and turn on the devices
3003	01		
3004	76	HLT	End of the program

Program Output

The program simulates controlling of the devices connected to the output port by displaying 70H on a seven-segment LED display. If your system has individual

LEDs, the binary pattern – 0111 0000- will be displayed.

6.3 Arithmetic Operations

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

ADD : Add	Add the contents of a register.
ADI : Add Immediate	Add 8- bit data
SUB : Subtract	Subtract the contents of a register.
SUI : Subtract Immediate	Subtract 8-bit data.
INR : Increment	Increase the contents of a register by 1
DCR : Decrement	Decrease the contents of a register by 1.

The arithmetic operations Add and Subtract are performed in relation to the contents of the accumulator. However, the Increment or Decrement operations can be performed in any register.

Instructions

These arithmetic instructions (except INR and DCR)

1. Assume implicitly that the accumulator is one of the operands.
2. Modify all the flags according to the data conditions of the result.
3. Place the result in the accumulator.
4. Do not affect the contents of the operand register.

The instructions INR and DCR

1. Affect the contents of the specified register.
2. Affect all flags except the CY flag.

Opcode	**Operand**	**Description**
ADD	R^+	Add It is 1- byte instruction Adds the contents of register R to the contents of the accumulator.
ADI	8-bit	Add Immediate It is 2- byte instruction Adds the second byte to the contents of the accumulator.
SUB	R^+	Subtract It is 1- byte instruction Subtracts the contents of register R from the contents of the accumulator.
SUI	8-bit	Subract Immediate It is 2- byte instruction Subtracts the second byte from the contents of the accumulator.
INR	R*	Increment It is a 1-byte instruction Increases the contents of register R by 1 All flags except the CY are affected
DCR	R*	Decrement It is a 1-byte instruction Decreases the contents of register R by 1 All flags except the CY are affected

6.3.1 Addition

The 8085 performs addition with 8-bit binary numbers and stores the sum in the accumulator. If the sum is larger than eight bits (FFH), it sets the Carry flag.

Addition can be performed either by adding the contents of a source register (B, C, D, E, H, L, or memory) to the contents of the accumulator (ADD) or by adding the second byte directly to the contents of the accumulator (ADI).

Example 6.3 :- The contents of the accumulator are 93H and the contents of register C are B7H. Add both contents.

Instruction :- **ADD C**

		CY	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
(A)	93= +		1	0	0	1	0	0	1	1
(C)	B7=		1	0	1	1	0	1	1	1
Carry		1		1	1		1	1	1	
SUM(A)	14A	1	0	1	0	0	1	0	1	0

Flag Status: S= 0, Z=0, CY=1

When the 8085 adds 93H and B7H, the sum is 14AH; it is larger than eight bits, . Therefore, the accumulator will have 4AH in binary, and the CY flag will be set.

The result in the accumulator (4AH) is not 0, and bit D_7 is not 1; therefore the Zero and the Sign flags will be reset.

Example 6.4:- Add the number 35H directly to the sum in the previous example when the CY flag is set.

Instruction: - **ADI 35H**

		CY	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
(A)	4AH +	1	0	1	0	0	1	0	1	0
Data	35H=		0	0	1	1	0	1	0	1
SUM(A)	7FH	0	0	1	1	1	1	1	1	1

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

The addition of 4AH and 35H does not generate a carry and will reset the previous carry flag.

Therefore, in adding numbers, it is necessary to count how many times the CY flag is set by using some other programming techniques.

Example 6.5:- Assume the accumulator holds the data byte FFH. Illustrate the difference in the flags set by adding 01H and by incrementing the accumulator contents.

Instruction:- ADI 01H

		CY	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
(A)	FFH= +		1	1	1	1	1	1	1	1
(Data)	01H=		0	0	0	0	0	0	0	1
Carry		1	1	1	1	1	1	1	1	
SUM(A)	100H	1	0	0	0	0	0	0	0	0

Flag Status: S= 0, Z=1, CY=1

After adding 01H to FFH the sum in the accumulator is 0 with a carry.

Therefore, the CY and Z flags are set. Sign flag is reset because D_7 is 0.

Instruction INR A

The accumulator contents will be 00H, the same as before. However, the instruction INR will not affect the Carry flag; it will remain in its previous status.

Flag Status: S= 0, Z=1, CY=NA

Flag Concepts And Cautions

After an operation, one or more flags may be set, and they can be used to change the direction of the program sequence by using the Jump instructions. The programmer should be alert for them to make a decision. If the flags are not appropriate for the tasks, the programmer can ignore them.

Caution #1 :- In the Example 6.3, CY flag is set, and in Example 6.4, CY flag is reset. If programmer ignores the flag, it can be lost after the subsequent instructions. The flag can be ignored when the programmer is not interested in using it.

Caution #2 :- In Example 6.5, two flags are set. The programmer may use one or more flags to make decisions or may ignore them if they are irrelevant.

Caution #3 :- The CY flag has a dual function ; it is used as a carry in addition and as a borrow in subtraction.

Carry Flag Set to 1 because the answer is larger than eight bits; there is a carry generated out of the last bit D_7. During the addition bits D_0 through D_6 may generate carries but these carries do not affect the CY flag.

Misconception #1:- Bit D_0 in the result (4AH) corresponds to the bit position of the Carry flag D_0 in the flag register, therefore, the Carry flag is reset.

Misconception #2 :- In the addition process, bits D_0 of 93H and B7H generate a carry (or other bit additions generate carries); therefore the Carry flag is set.

Zero Flag:- Reset to 0 because the answer is not zero. The Zero flag is set only when all eight bits in the result are 0.

Misconception #3 :- Bit D_6 in the result (4AH) is 1, and it corresponds to bit D_6 (zero flag position) in the fag register. Therefore, the Z flag is set.

Sign Flag :- Reset to 0 because D_7 in the result is 0. The position of the sign flag in the flag register is also D_7 . But it is just a coincidence. The microprocessor designer could have chosen bit D_6 for the Sign flag and bit D_7 for the Zero flag in the flag register. The Sign flag is relevant only when we are using signed numbers.

Misconception #4 :- If the Sign flag is set, the result must be negative.

6.3.2 Illustrate Program: Arithmetic Operations –Addition and Increment

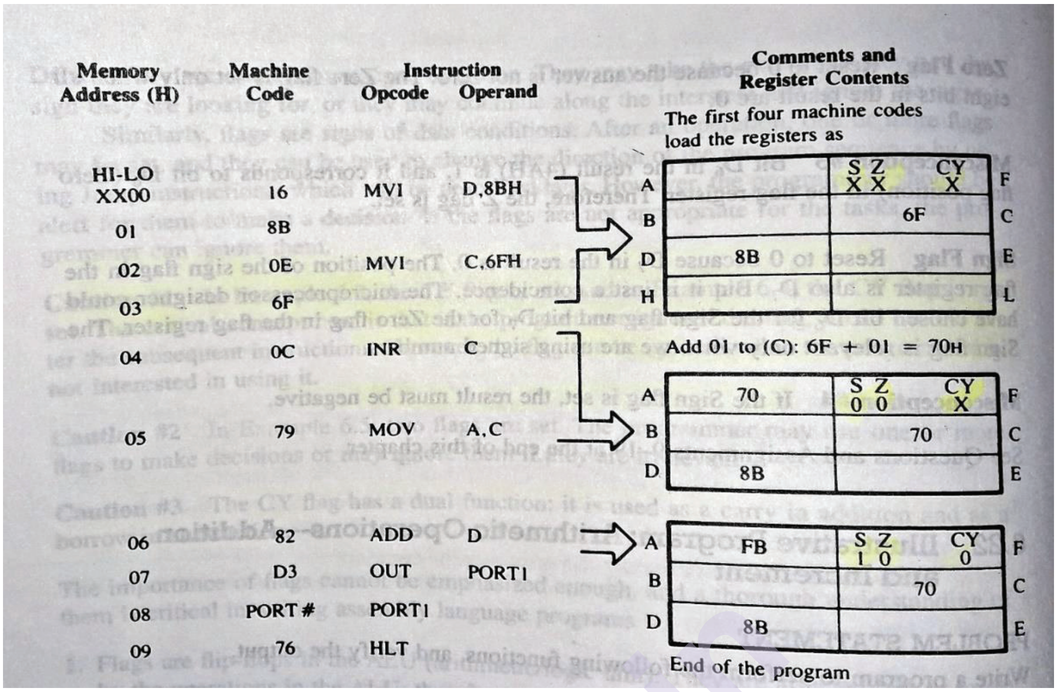
Problem Statement

Write a program to perform the following functions, and verify the output.

1. Load the number 8BH in register D
2. Load the number 6FH in register C.
3. Increment the contents of register C by 1.
4. Add the contents of register C and D and display the sum at the output PORT1.

Program

The illustrative program for arithmetic operations using addition and increment is presented as figure 6.5 to show the register contents during some of the steps.



Memory Address (H)	Machine Code	Instruction Opcode	Instruction Operand	Comments and Register Contents
HI-LO XX00	16	MVI	D,8BH	The first four machine codes load the registers as
01	8B			
02	0E	MVI	C,6FH	
03	6F			
04	0C	INR	C	Add 01 to (C): 6F + 01 = 70H
05	79	MOV	A,C	
06	82	ADD	D	
07	D3	OUT	PORT1	
08	PORT#	PORT1		
09	76	HLT		End of the program

Figure 6.5: Illustrative Program for Arithmetic Operations-Using Addition and Increment

Program Description

1. The first machine cycle codes load 8BH in register D and 6FH in register C . These are data copy instructions, so no flags are affected and they remain in the previous state.

   The status of the flags is shown X to indicate no change in their status.

2. Instruction INR C adds 1 to 6FH and changes the contents of C to 70H . The result is nonzero and bit D_7 is zero; therefore, the S and Z flags are reset. The CY flag is not affected by the INR instruction,

3. To add (C) to (D), the contents of the registers must be transferred to the accumulator because the 8085 cannot add two registers directly. The instruction MOV A, C copies 70Hfrom C register into the accumulator without affecting (C).

4. Instruction ADD D, adds (D) to (A) stores the sum in A, and sets the Sign flag as shown below:

   (A) : 70H = 0 1 1 1 0 0 0 0
   +
   (D) : 8BH = 1 0 0 0 1 0 1 1

   (A) : FBH = 1 1 1 1 1 0 1 1

   Flag Status: S=1, Z=0, CY=0

5. The sum is displayed by the OUT instruction.

Program Output

It will display FBH at the output port. If the output port is not available, the program can be executed by entering the NOP instructions in place of the OUT instruction and the answer FBH can be verified by examining the accumulator A. Similarly the contents of registers C and D and the flags can be verified.

By examining the contents of the registers, following points can be confirmed:

1. The sum is stored in Accumulator
2. The contents of the source registers are not changed.
3. The Sign (S) flag is set.

Even though the Sign(S) flag is set, this is not a negative sum. The microprocessor sets the Sign flag whenever an operation results in D_7 =1. The microprocessor cannot recognize whether FBH is a sum, a negative number, or a bit pattern.

In this example, the addition is not concerned with the signed numbers. With the signed numbers, bit D_7 is reserved for a sign programmer and no number larger than $+127_{10}$ can be entered.

6.3.3 Subtraction

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

The subtraction can be performed by using either the instruction SUB to subtract the contents of a source register or the instruction SUI to subtract an 8- bit number from the contents of the accumulator.

The 8085 performs the following steps internally to execute the instruction SUB (or SUI)

Step1: Converts subtrahend (the number to be subtracted) into its 1's complement.

Step 2 : Adds 1 to 1's complement to obtain 2's complement of the subtrahend.

Step3: Add 2's complement to the minuend (the contents of the accumulator)

Step 4: Complement the Carry flag

Example 6.6

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

Instruction: SUB B
Subtrahend (B) : 65H = 0 1 1 0 0 1 0 1

Step1:

1's complement of 65H=	1 0 0 1 1 0 1 0

Step2:

+	
Add 01 to obtain	0 0 0 0 0 0 0 1
2's complement of 65H=	1 0 0 1 1 0 1 1

To subtract: 97H- 65H

Step 3:

Add 97H to 2's complement of 65H=		1 0 0 1 0 1 1 1
2's complement of 65H=		1 0 0 1 1 0 1 1
Carry		1 1 1 1 1
	CY 1	0 0 1 1 0 0 1 0
	CY 0	0 0 1 1 0 0 1 0

Step 4 : Complement Carry

Result (A) :32H

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

If the answer is negative, it will be shown in the 2's complement of the actual magnitude.

For example, if the above subtraction is performed as 65H – 97H, the answer will be the 2's complement of 32H with the Carry (Borrow) flag set.

6.3.4 Illustrative Program: Subtraction of Two Unsigned Numbers

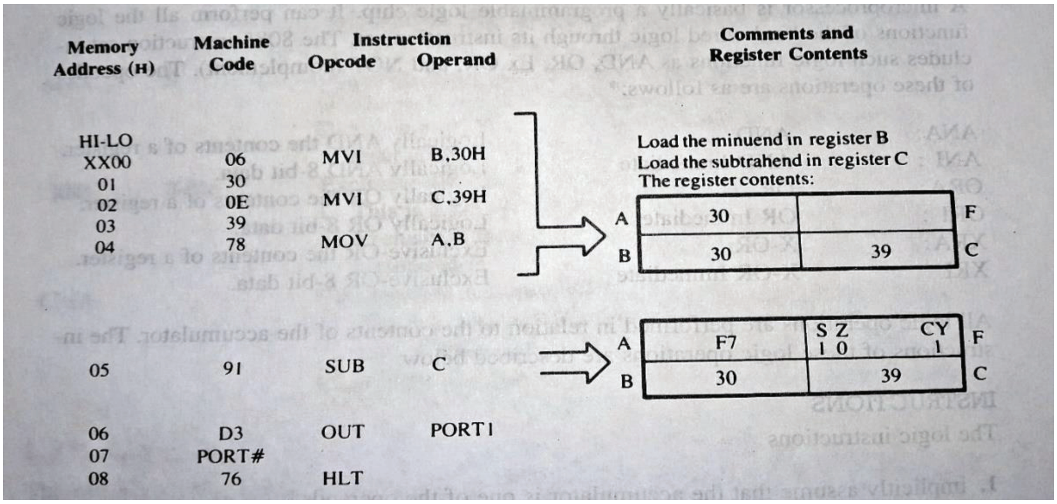
Problem Statement

Write a program to do the following:

1. Load the number 30H in register B and 39H in register C.
2. Subtract 39H from 30H
3. Display the answer at PORT1

Program

The illustrative program for subtraction of two unsigned numbers is presented as Figure 6.6 to show the register contents during the steps.



Memory Address (H)	Machine Code	Instruction Opcode	Operand	Comments and Register Contents
HI-LO				
XX00	06	MVI	B,30H	Load the minuend in register B
01	30			Load the subtrahend in register C
02	0E	MVI	C,39H	The register contents:
03	39			
04	78	MOV	A,B	
05	91	SUB	C	
06	D3	OUT	PORT1	
07	PORT#			
08	76	HLT		

A 30 F
B 30 39 C
A F7 S Z 1 0 CY 1 F
B 30 39 C

Figure 6.6: **Illustrative Program: Subtraction of Two Unsigned Numbers**

Program Description

1. Register B and C are loaded with 30H and 39H, respectively. The instruction MOV A, B copies 30H into the accumulator. The contents of a register can be subtracted only from the contents of the accumulator and not from any other register.

2. To execute the instruction SUB C the microprocessor performs the following steps

Step1:

39H = 0 0 1 1 1 0 0 1
1's complement of 39H= 1 1 0 0 0 1 1 0

Step2:

+
Add 01 to obtain 0 0 0 0 0 0 0 1
2's complement of 39H= 1 1 0 0 0 1 1 1

To subtract: 30H- 39H

Step 3:

Add 30H to 2's complement of 39H= 0 0 1 1 0 0 0 0
2's complement of 39H= 1 1 0 0 0 1 1 1

CY=0 1 1 1 1 0 1 1 1

Step 4: Complement carry
CY=1 1 1 1 1 0 1 1 1 = F7H

3. The number F7H is a 2's complement of the magnitude (39H – 30H) = 09H.

4. The instruction OUT display F7H at PORT1.

Program Output

In this program, the unsigned numbers were used to perform the subtraction.

There is no way to differentiate between a straight binary number and 2's complement by examining the answer at the output port. The flags are internal and not easily displayed. However, a programmer can test the Carry flag by using the instruction Jump On Carry (JC) and can find a way to indicate the answer is in 2's complement.

6.4 Logic Operations

The 8085instruction set includes logic functions such as AND, OR, Ex OR, and NOT (complement) . The opcodes of these operations as follows :

ANA:	AND	Logically AND the contents of a register
ANI:	AND Immediate	Logically AND 8-bit data
ORA:	OR	Logically OR the contents of a register
ORI:	OR Immediate	Logically OR 8-bit data
XRA:	X-OR	Exclusive –OR the contents of a register.
XRI:	X-OR Immediate	Exclusive-OR 8-bit data.

All logic operations are performed in relation to the contents of the accumulator.

Instructions

1. Implicitly assume that the accumulator is one of the operands.

2. Reset (clear) the CY flag .The instruction CMA is an exception; it does not affect any flags.

3. Modify the Z, P, and S flags according to the data conditions of the result.

4. Place the result in the accumulator.

5. Do not affect the contents of the operand register.

Opcode	Operand	Description
ANA	R	Logical AND with Accumulator This is a 1- byte instruction Logically ANDs the contents of the register R with the contents of accumulator. 8085 : CY is reset and AC is set.
ANI	8-bit	AND Immediate with Accumulator This is a 2- byte instruction Logically ORs the second byte with the contents of accumulator 8085 : CY is reset and AC is set.
ORA	R	Logical OR with Accumulator This is a 1- byte instruction Logically ORs the contents of the register R with the contents of accumulator
ORI	8-bit	OR Immediate with Accumulator This is a 2- byte instruction Logically ORs the second byte with the contents of accumulator
XRA	R	Logical Exclusive-OR with Accumulator This is a 1- byte instruction Exclusive-OR the contents of the register R with the contents of accumulator.
XRI	8-bit	Exclusive-OR Immediate with Accumulator This is a 2- byte instruction Exclusive-ORs the second byte with the contents of accumulator
CMA		Complement Accumulator This is a 1-byte instruction that complements the contents of accumulator No flags are affected.

6.4.1 Logic AND

The process of performing logic operations through the software instructions is slightly different from the hardwired logic. The AND gate is shown in Figure 6.7(a) has two inputs and one output

On the other hand, the instruction ANA simulates eight AND gates, as shown in Figure 6.7(b) .

For example, assume that register B holds 77H and the accumulator A holds 81H. The result of the instruction ANA B is 01H and is placed in the accumulator replacing the previous contents as shown in figure 6.7 (b)

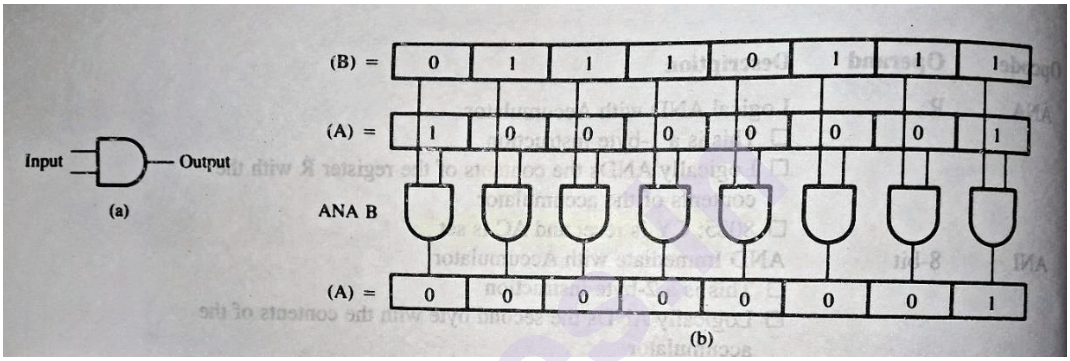


Figure 6.7(a) Gate (b) a simulated ANA Instruction

Figure 6.7(b) shows that each bit of register b is independently ANDed with each bit of the accumulator, thus simulating eight 2- input AND gates.

6.4.2 Illustrative Program: Data Masking with Logic AND

Problem Statement

To conserve energy and to avoid an electrical overload on a hot afternoon, implement the following procedures to control the appliances throughout the house(figure 6.8) . Assume that the control switches are located in the kitchen, and they are available to anyone in the house. Write the instruction to

1. Turn on the air conditioner if switch S_7 of the input port 00H is on.
2. Ignore all other switches of the input port even if someone attempts to turn on other appliances.

Problem Analysis

In this problem we are interested in only one switch positions, S_7, which is connected to data line D_7. Assume that various persons in the family have turned on the switches of the air conditioner (S_7), the radio (S_4), and the lights (S_3, S_2, S_1, S_0).

If the microprocessor reads the input port (IN 00H), the accumulator will have data byte 9FH .This can be simulated by using the instruction MVI A, 9FH .However, if we are interested in knowing only whether switch S_7 is on, we can mask bits D_6 through D_0 by ANDing the input data with a byte that has 0 in bit positions D_6 through D_0 and 1 in the position D_7 .

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

After bits D_6 through D_0 have been masked, the remaining byte can be sent to the output port to simulate turning on the air conditioner.

Program

Memory Address	Machine Code	Instruction Opcode	Operand	Comments
HI-LO				
XX00	3E	MVI	A, Data	This instruction simulates the instruction IN 00H
01	9F			
02	E6	ANI	80H	Mask all the bits except D_7
03	80			
04	D3	OUT	01H	Turn on the air conditioner if S_7 is on.
05	01			
06	76	HLT		End of program

Program Output

The instruction ANI 80H ANDs the accumulator data as follows:

(A)=	1 0 0 1 1 1 1 1	(9FH)
AND		
(Masking Byte =	1 0 0 0 0 0 0 0	(80H)
(A)	1 0 0 0 0 0 0 0	(80H)

Flag Status: S=1, Z= 0, CY= 0

The ANDing operation always reset the CY flag. The result (80H) will be placed in accumulator and then sent to output port, and the logic 1 of data bit D_7 turns on the air conditioner.

The masking is a commonly used technique to eliminate unwanted bits in abyte. The masking byte to be logically ANDed is determined by placing 0s in bit positions that are to be masked and by placing 1s in the remaining bit positions.

6.4.3 OR, Exclusive-OR and NOT

The instruction ORA (and ORI) simulates logic ORing with eight 2- input OR gate ; this process is similar to that of ANDing The instruction XRA (and XRI) performs Exclusive ORing of eight bits, and the instruction CMA inverts bits off the accumulator.

Example 6.7:- Assume register B holds 93H and the accumulator holds 15H. Illustrate the results of the instruction ORA B, XRA B, and CMA.

1. The instruction ORA B will perform the following operation:

(B)	1 0 0 1 0 0 1 1	(93H)
OR		
(A)	0 0 0 1 0 1 0 1	(15H)
(A)	1 0 0 1 0 1 1 1	(97H)

Flag Status: S= 1, Z= 0, CY=0

The result 97H will be placed in the accumulator, the CY flag will be reset, and the other flags will be modified to reflect the data conditions in the accumulator.

2. The instruction XRA B will perform the following operation.

(B)	1 0 0 1 0 0 1 1	(93H)
X-OR		
(A)	0 0 0 1 0 1 0 1	(15H)
(A)	1 0 0 0 0 1 1 0	(86H)

Flag Status: S= 1, Z= 0, CY=0

The result 86H will be placed in the accumulator and flags will be modified as shown.

3. The instruction CMA will result in

(A)	0 0 0 1 0 1 0 1	(15H)
CMA		
(A)	1 1 1 0 1 0 1 0	(EAH)

The result EAH will be placed in the accumulator and no flags will be modified.

6.4.4 Setting And Resetting Specific Bits

At the various times, we may want to set or reset a specific bit without affecting the other bits. OR logic can used to set the bit, and AND logic can be used to reset the bit.

Example 6.8 :- In Figure 6.8, keep the radio on (D_4) continuously without affecting functions of other appliances, even if someone turns off the switch S_4 .

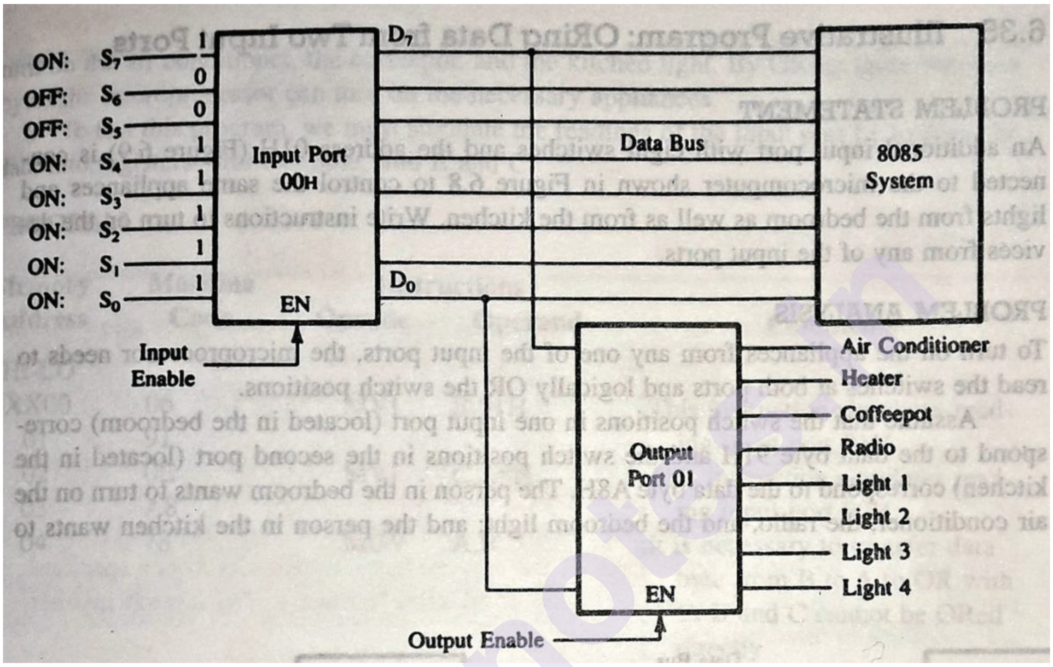


Figure 6.8:- Input Port To Control Appliances

Solution:- To keep the radio on without affecting the other appliances, the bit D_4 should be set by ORing the reading of the input port with the data byte 10H as follows :

IN 00H	(A)=	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
ORI 10H	=	0	0	0	1	0	0	0	0
	(A)	D_7	D_6	D_5	1	D_3	D_2	D_1	D_0

Flag Status: CY=0 ; other will depends on data

The instruction IN reads the switch positions shown as $D_7 – D_0$ and the instruction ORI sets the bit D_4 without affecting any other bits.

Example 6.9:- In the Figure 6.8, assume it is winter and turn off the air conditioner without affecting the other appliances.

Solution :- To turn off the air conditioner, reset bit D7 by ANDing the reading of the input port with the data byte 7FH as follows :

IN 00H	(A)=	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
ANI 7FH	=	0	1	1	1	1	1	1	1
	(A)	0	D_6	D_5	D_4	D_3	D_2	D_1	D_0

Flag Status: CY=0 ; other will depends on data

The ANI instruction resets bit D_7 without affecting the other bits.

6.4.5 Illustrative Program : ORing Data From Two Input Ports

Problem Statement

An additional input port with eight switches and the address 01H (Figure 6.9) is connected to the microcomputer shown in the Figure 6.8 to control the same appliances and lights from the bedroom as well as from the kitchen. Write instructions to turn on the devices from any of the input ports.

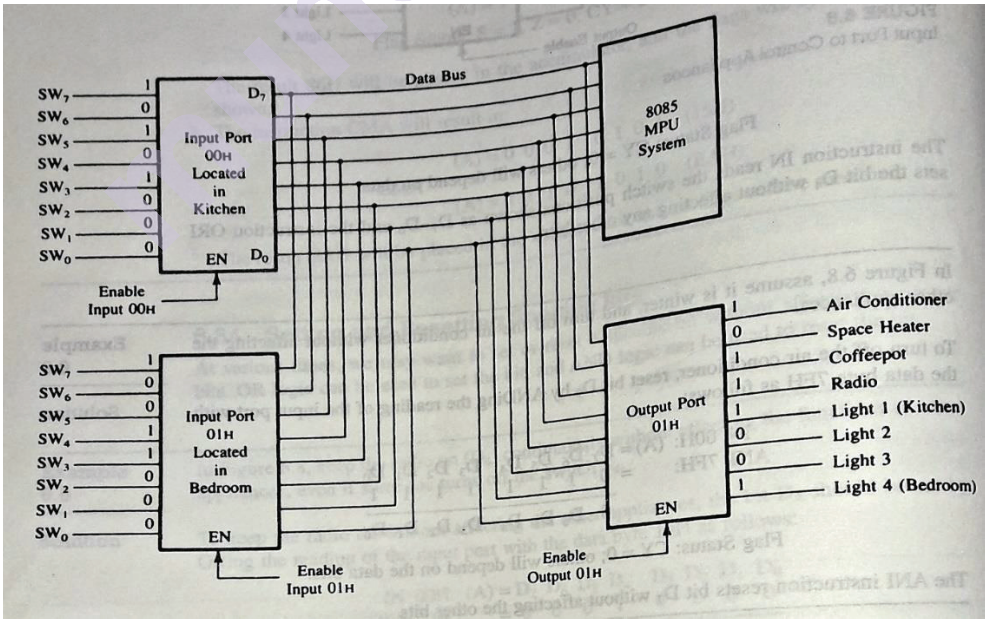


Figure 6.9 Two Input Ports to Control Output Devices

Problem Analysis

To turn on the appliances from any one of the input ports, the microprocessor needs to read the switches at both ports and logically OR the switch positions.

Assume that the switch positions in one input port correspond to the data byte 91H and the switch positions in the second port correspond to the data byte A8H. The person in the bedroom wants to turn on the air conditioner, the radio, and the bedroom light; and the person in the kitchen wants to turn on the air conditioner, the coffeepot, and the kitchen light. By ORing these two data bytes the microprocessor can turn on the necessary appliances.

To test this program, we must simulate the reading of the input port by loading the data into register – for example, into B and C.

Program

Memory	Machine	Instruction		Comments
Address	Code	Opcode	Operand	
HI-LO				
XX00	06	MVI	B, 91H	This instruction simulates reading input port 01H
01	91			
02	0E	MVI	C, A8H	This instruction simulates reading input port 00H
03	A8			
04	78	MOV	A, B	It is necessary to transfer data byte fro B to A to OR with C. Band C cannot be ORed directly.
05	B1	ORA	C	Combine the switch positions from register B and C in the accumulator.
06	D3	OUT	PORT!	Turn on appliances and light
07	PORT1			
08	76	HLT		End of program

PROGRAM OUTPUT

By logically ORing the data bytes in registers B and C

(B) → (A) =	1 0 0 1 0 0 0 1	91H
(C) =	1 0 1 0 1 0 0 0	A8H
(A)=	1 0 1 1 1 0 0 1	(B9H)

Flag Status: S=1, Z=0, CY= 0

Data byteB9H is placed in the accumulator that turns on the air conditioner, radio, coffeepot, and bedroom and kitchen lights.

6.5 BRANCH OPERATIONS

The branch instructions are most powerful instructions because they allow the microprocessor to change the sequence of a program, either unconditionally or under certain test conditions. These instructions are key to the flexibility and versatility of a computer.

Branch instructions instruct the microprocessor to go to a different memory location, and the microprocessor continues executing machine codes from that new location.

The address of the new memory location is either specified explicitly or supplied by the microprocessor or by extra hardware. They are classified in three categories:

1. Jump instructions
2. Call and Return instructions
3. Restart instructions

The Jump instructions specify the memory location explicitly. They are 3-byte instructions: one byte for the operation code; followed by a 16-bit memory address. Jump instructions are classified into two categories Unconditional Jump and Conditional Jump.

6.5.1 Unconditional Jump

The 8085 instruction set includes one unconditional jump instruction. The unconditional Jump instruction enables the programmer to set up continuous loops.

Instruction

Opcode	**Operand**	**Description**
JMP	16-bit	Jump It is 3-byte instruction The second and third bytes specify the 16-bit memory address. Second byte low-order and third-byte high- order memory address.

For example, to instruct the microprocessor to go the memory location 2000H, the mnemonics and the machine code entered will be as follows:

Machine Code	Mnemonics
C3 00 20	JMP 2000H

The 16 –bit memory address of the jump location is entered in the reverse order, the low-order byte(00H) first, followed by the high-order byte(20H)

6.5.2. Illustrative Program: Unconditional Jump to Set Up a Continuous Loop

Problem Statement

Modify the program in Example6.2 to read the switch positions continuously and turn on the appliances accordingly.

Problem Analysis

One of the major drawbacks of the program in Example 6.2 is that the program reads switch positions once and then stops. Therefore, if you want to turn on/off different appliances, you have to reset the system and start all over again. This is impractical in real-life situations. However, the unconditional Jump instruction, in place of the HLT instruction, will allow the microcomputer to monitor the switch positions continuously.

Memory Address	Machine Code	Label	Mnemonics	Comments
2000	DB	START	IN 00H	Read input switches
2001	00			
2002	D3		OUT 01H	Turn on devices according to switch position
2003	01			
2004	C3		JMP START	Go back to beginning and read the switches again
2005	00			
2006	20			

Program Format

The program includes one more column called label . The memory location 2000H is defined with the label START ; therefore, the operand of the jump instruction can be specified by the label START. The program sets up the endless loop, and the microprocessor monitors the input port continuously. The output will reflect any change in the switch positions.

6.5.3 Conditional Jumps

Conditional Jump instructions allow the microprocessor to make decisions based on certain conditions indicated by the flags. After logic and arithmetic operations, flip-flops (flags) are set or reset to reflect data conditions. The conditional Jump instructions check the flag conditions and make decisions to change the sequence of a program.

Flags

The 8085 flag register has five flags, one of which (Auxiliary Carry) is used internally.

The other four flags used by the Jump instructions are

1. Carry flag
2. Zero flag
3. Sign flag
4. Parity flag

Two Jump instructions are associated with each flag. The sequence of a program can be changed either because the condition is present or because the condition is absent.

For example, while adding the numbers we can change the program sequence either because the carry is present (JC= Jump on Carry) or because carry is absent (JNC=Jump On No Carry).

Instructions

All conditional Jump instructions in 8085 are 3-byte instructions; the second byte specifies the low-order (line number) memory address, and the third byte specifies the high-order (page number) memory address.

The following instructions transfer the program sequence to the memory location specified under the given conditions.

Opcode	Operand	Description
JC	16-bit	Jump on Carry (if result generate carry and CY=1)
JNC	16-bit	Jump on No Carry (CY=0)
JZ	16-bit	Jump on Zero (if result is zero and Z=1)
JNZ	16-bit	Jump on No Zero(Z=0)
JP	16-bit	Jump On Plus (if D_7 = 0 and S=0)
JM	16-bit	Jump On Minus (if D_7= 0 and S=0)
JPE	16-bit	Jump On Even Parity (P=1)
JPO	16-bit	Jump On Odd Parity (P=0)

All the Jump instructions are listed here. Zero and Carry flags and related Jump instructions are used frequently.

6.5.4 Illustrative Program: Testing of the Carry Flag

Problem Statement

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

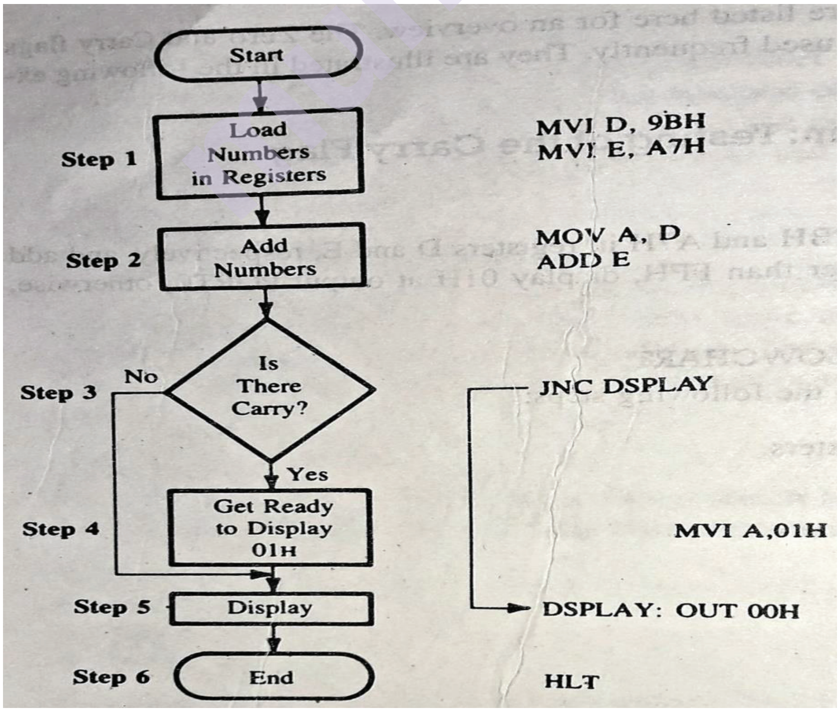
Problem Analysis And Flowchart

The problem can be divided into the following steps:

1. Load the numbers in the registers.
2. Add the numbers
3. Check the sum.
   Is the sum > FFH, go to step 4, else go to step 5
4. Get ready to display 01
5. Display
6. End

Flowchart and Assembly Language Program

The six steps listed above can be converted into a flowchart and assembly language program as shown in Figure 6.10



```
          MVI D, 9BH
          MVI E, A7H

          MOV A, D
          ADD E

          JNC DSPLAY

          MVI A,01H

DSPLAY:   OUT 00H

          HLT
```

Figure 6.10: Flowchart And Assembly Language Program to Test Carry Flag

Step 3 is a decision-making block. In a flowchart, the decision-making process is represented by a diamond shape.

It is important to understand how this block is translated into the assembly language program.

1. Is there a Carry
2. If the answer is no, change the sequence of the program. In the assembly language this is equivalent to JUMP On No Carry –JNC.
3. Now the next question is where to change the sequence – to Step 5.At this point exact location is not known, but it is labelled DSPLAY.
4. The next step in the sequence is 4 .Get ready to display byte 01H.
5. After completing the straight line sequence, translate Step 5 and Step 6 : Display at the port and halt.

Machine Code with Memory Addresses

Assuming R/W memory begins at 2000H, the preceding assembly language program can be translated as follows:

Memory Address	Machine Code	Label	Mnemonics
2000	16	START:	MVI D, 9BH
2001	9B		
2002	1E		MVI E, A7H
2003	A7		
2004	7A		MOV A, D
2005	83		ADD E
2006	D2		JNC DSPLAY
2007	X		
2008	X		
2009	3E		MVI A, 01H
200A	01		
200B	D3	DISPLAY:	OUT 00H
200C	00		
200D	76		HLT

While translating into machine code, we leave memory locations 2007H and 2008H blank because the exact locations of the transfer is not known. What is known is that two bytes should be reserved for the 16-bit address. After completing the straight line sequence, we know the memory address of the label DSPLAY i.e. 200BH. This address must be placed in the reversed order as shown:

2007	0B	Low- order: Line Number
2008	20	High-order : Page Number

Using the Instruction Jump On Carry (JC)

Now the question remains : Can the same problem be solved b using the instruction Jump On Carry (JC) ? To use instruction JC, exchange the places of the answers YES and NO to the question : Is there a Carry ?

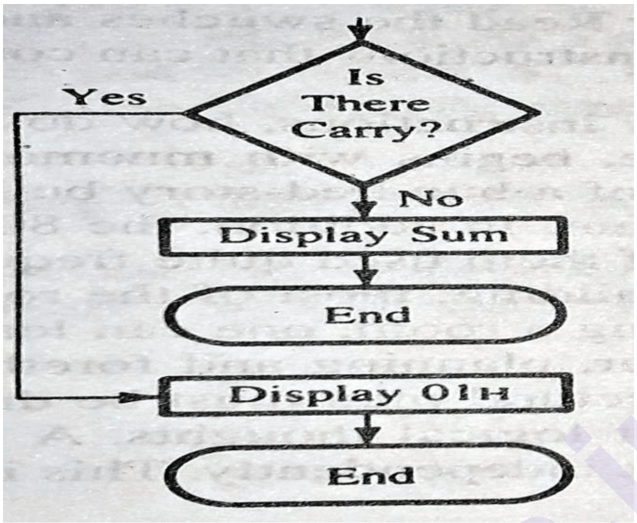


Figure 6.11: Flowcharts for Instruction JUMP ON CARRY

The flowchart will be as be as in Figure 6.11, and it shows that the program sequence is changed if there is a Carry. This flowchart has two end points : thus it will require a few more instructions than that of the Figure 6.10.In this particular example, it is unimportant whether to use instruction JC or JNC, but in most cases the choice is made by the logic of a problem.

6.6 Writing Assembly Language Programs

Writing a program is equivalent to giving commands to the microprocessor in a sequence to perform a task.

6.6.1 Getting Started

Perform a Task. What is the task you are asking to do?

Sequence .What is the sequence you want it to follow?

Commands What are commands (instruction set) it can understand?

These terms can be translated into the steps as follows:

Step 1: Read the problem carefully.

Step 2 : Break it down into small steps.

Step 3: Represent these steps in a possible sequence with a flowchart – a plan of attack.

Step 4: Translate each block of the flowchart into appropriate mnemonic instructions.

Step 5 : Translate mnemonics into the machine code.

Step 6: Enter the machine code in memory and execute. Only on rare occasions is a program successfully executed on the first attempt.

Step 7 : Start troubleshooting (debug a program).

6.6.2 Illustrative Program: Microprocessor –Controlled Manufacturing Process

Problem Statement

A microcomputer is designed to monitor various processes on the floor of a manufacturing plant, presented schematically in Figure 6.12.It has two input ports with the addresses F1H and F2H and output port with address F3H.

Input port F1H has six switches, five of which D_4 –D_0 control the conveyer belts through the output port F3H.

Switch S_7, corresponding to the data line D_7, is reserved to indicate an emergency on the floor. Input port F2H is controlled by the foreman, and its switch S_7’ is used to indicate an emergency. Output line D_6 of port F3H is connected to the emergency alarm.

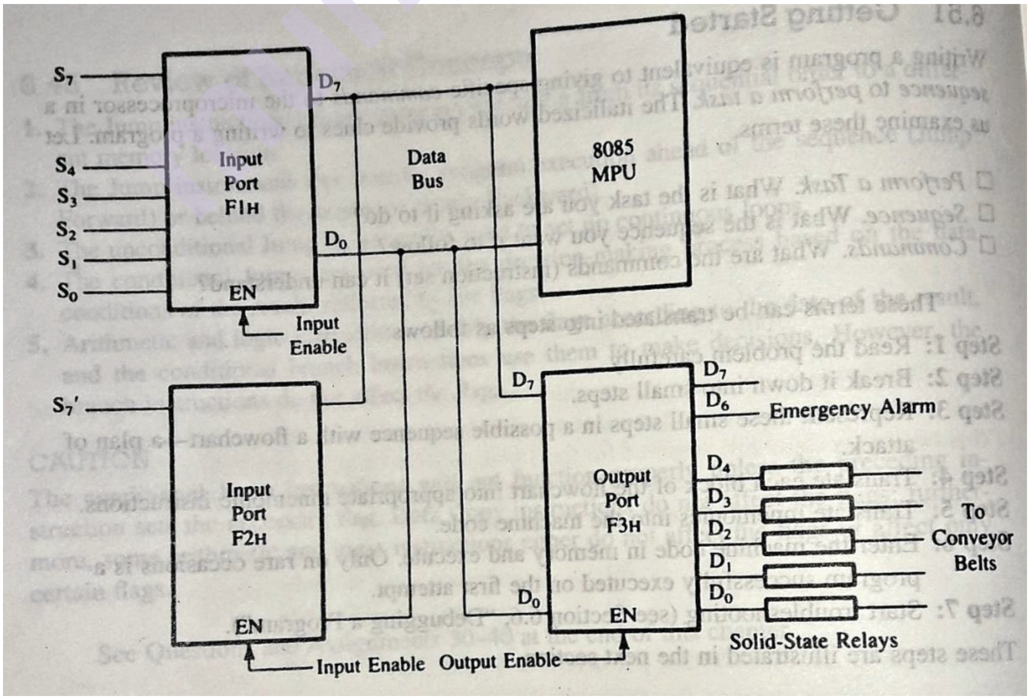


Figure 6.12: Input /Output Ports to control Manufacturing Processes

Write a program to

1. Turn on the five conveyer belts according to the ON/ OFF positions of the switches $S_4 – S_0$ at port F1H.
2. Turn off the conveyer belts and turn on the emergency alarm only when both switches –S7 from port F1H and S_7’ from the port F2H – are triggered.
3. Monitor the switches continuously.

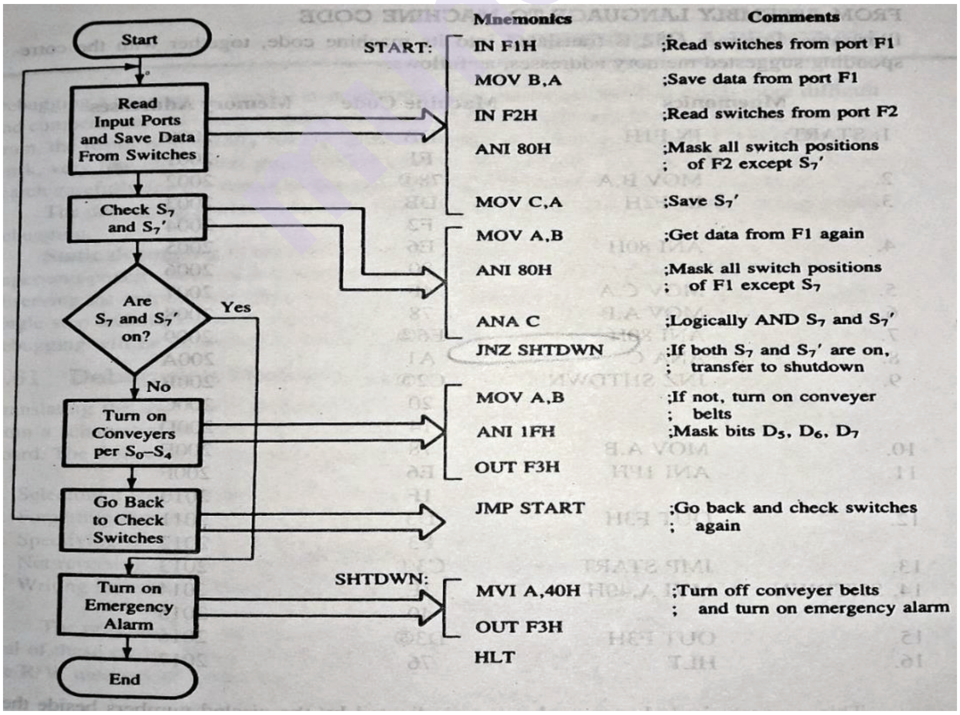
Problem Analysis

To perform the task specified in the problem, the microprocessor needs to

1. Read the switch positions.
2. Check whether switches S_7 and S_7’ from the ports F1H and F2H are on.
3. Turn on the emergency signal if both switches are on, and turn off all the conveyer belts.
4. Turn on the conveyer belts according to the switch positions S_0 through S_4 at input port F1H if both the switches, S_7 and S_7’ are not on simultaneously.
5. Continue checking the switch positions.

The five steps listed above can be translated into a flowchart and an assembly language program as shown in the Figure 6.13

Flowchart and Program


Start
Read Input Ports and Save Data From Switches
Check S₇ and S₇′
Are S₇ and S₇′ on?
Yes
No
Turn on Conveyers per S₀–S₄
Go Back to Check Switches
Turn on Emergency Alarm
End

```
         Mnemonics         Comments
START:   IN F1H            ;Read switches from port F1
         MOV B,A           ;Save data from port F1
         IN F2H            ;Read switches from port F2
         ANI 80H           ;Mask all switch positions
                           ;  of F2 except S7'
         MOV C,A           ;Save S7'
         MOV A,B           ;Get data from F1 again
         ANI 80H           ;Mask all switch positions
                           ;  of F1 except S7
         ANA C             ;Logically AND S7 and S7'
         JNZ SHTDWN        ;If both S7 and S7' are on,
                           ;  transfer to shutdown
         MOV A,B           ;If not, turn on conveyer
                           ;  belts
         ANI 1FH           ;Mask bits D5, D6, D7
         OUT F3H
         JMP START         ;Go back and check switches
                           ;  again
SHTDWN:  MVI A,40H         ;Turn off conveyer belts
                           ;  and turn on emergency alarm
         OUT F3H
         HLT
```

Figure 6.13: Flowchart and Program for Controlling Manufacturing Processes

6.6.3 Documentation

A program is similar to a circuit diagram. Its purpose is to communicate to others what the program does and how it does it. Appropriate comments are critical for conveying the logic behind a program. The program as a whole should be self-documented.

From Assembly Language To Machine Code

Mnemonics	Machine Code	Memory Addresses
1. START: IN F1H	DB F1	2000 2001
2. MOV B, A	78 (1)	2002
3. IN F2H	DB F2	2003 2004
4. ANI 80H	E6 80	2005 2006
5. MOV C, A	4F	2007
6. MOV A, B	78	2008
7. ANI 80H	E6(2)	2009
8. ANA C	A1	200A
9. JNZ SHTDWN	C2(3) 20 14	200B 200C 200D
10. MOV A, B	78	200E
11. ANI 1FH	E6 1F	200F 2010
12. OUT F3H	D3 F3	2011 2012
13. JMP START 14. SHTDWN: MVIA, 40H	C3(4) 3E 40	2013 2014 2015
15. OUT F3H	D3(5)	2016
16. HLT	76	2017

This program includes the several errors, indicated by the () besides the code.

Program Execution

The above machine codes can be loaded in R/W memory, starting with memory address 2000H.The execution of the program can be done in two ways. The first is to execute the entire code by pressing the Execute key, and second is to use the

Single-Step key executes one instruction at a time, and by examining Register key and flags as each instruction is being executed.

6.7 DEBUGING A PROGRAM

Debugging is a program is similar to troubleshooting hardware. It is essential to search carefully for the errors in the program logic, machine code, and execution.

Static Debugging is similar to visual inspection of a circuit board; it is done by a paper- and pencil check off a flowchart and machine code.

Dynamic Debugging involves observing the output, or register contents, following the execution of each instruction(Single-Single technique) or of a group of instructions (the breakpoint technique).

6.7.1 Debugging Machine Code

Translating the assembly language to the machine code is similar to building a circuit from a schematic diagram; the machine code will have errors just as would the circuit.

The following errors are common:

1. Selecting a wrong code.
2. Forgetting the second or third byte of an instruction.
3. Specifying the wrong jump instruction.
4. Not reversing the order of high and low bytes in a Jump instruction
5. Writing memory addresses in decimal, thus specifying wrong jump instructions.

6.8 Summary

The instructions from 8085 instruction set include Data transfer instructions such as MOV, MVI, IN, OUT instruction. These instructions copy the contents of the source into the destination without affecting the source register.

Arithmetic Instructions such as ADD, ADI, SUB, SUI, INR, DCR and Logic Instructions such as ANA, ANI, ORA, ORI, XRA, XRI, CMA. The results of arithmetic and logic operations are usually placed in the accumulator.

The conditional Jump instructions are executed according to the flags set after an operation.

Questions and Programming Assignments

Q1) Explain data transfer operations with examples.

Q2) Write instructions to load the hexadecimal number 65H in register C, and 92H in the accumulator A. Display the number 65 at PORT0 and 92H in PORT1.

Q3) Explain Arithmetic operations with examples.

Q4) Write a program using the ADI instruction to add the two hexadecimal numbers 3AH and 48H and to display the answer at an output port.

Q5) Explain Logic operations with examples.

Q6) Explain branch operations with examples.

Books and References

1. Computer System Architecture by M. Morris Mano, PHI Publication, 1998.
2. Structured Computer Organization by Andrew C. Tanenbaum, PHI Publication.
3. Microprocessors Architecture, Programming and Application with 8085 by Ramesh Gaonker, PENRAM, Fifth Edition, 2012.

❖❖❖

UNIT 3

7

PROGRAMMING TECHNIQUES WITH ADDITIONAL INSTRUCTIONS

Unit Structure

7.1 Objectives

7.2 Introduction

7.3 Looping, Counting And Indexing

7.4 Additional Data Transfer And 16-Bit Arithmetic Instructions&Arithmetic Instruction Related To Memory.

7.5 Logic Operations: Rotate,Logics Operations: Compare, Dynamic Debugging.

7.1 Objectives

At the end of this unit, the student will be able to

- Write the program on Looping
- Write the program on 16 bit arithmetic Instructions
- Illustrate various Data transfer instructions
- Describe the concept of rotate, compare instructions and dynamic debugging

7.2 Introduction

1. The 8085 instruction set includes equivalents of the 8086
2. An instruction is a binary pattern designed inside a microprocessor to perform a specific function.
3. As it is tedious and error inductive to recognize and write instruction in binary languages these instructions are written in hexadecimal code.
4. Each manufacturer of a microprocessor has devised a symbolic code for each instruction called Mnemonic.

5. The mnemonic for a particular instruction consists of letters and suggest the operation to be performed by that instruction.

6. The mnemonic for a particular instruction consists of letters and suggest the operation to be performed by that instruction.

Table 1 Equivalent Binary and Hexadecimal code for Mnemonic

Binary Code	Hexadecimal Code	Mnemonic
00111100	3C	INR A
10000000	80	ADD B

7. Machine Language- The instruction in binary coded form or hexadecimal coded form are called machine code. A program written using machine code (only 0 & 1) is called machine language program

8. Assembly Language- The program can also be written using mnemonic operation codes & symbolic address for writing instructions and the data using different notations I.e binary,decimal,hexadecimal etc. This is called assembly language, program written in assembly language has to be translated in to a machine language. A translator, which translates an assembly program in to a machine language program is known as assembler.

9. Mnemonics can be written by hand on a paper and translated manually in hexadecimal code by looking in to the opcode sheet or table. This is called hand assembly.

10. Then the program can be feed in to the microprocessor kit for execution starting from the first address of RAM(Random Access Memory).

11. Assembly language program (Source Code) -------------> Hand Assembly---------> Machine Language Program

12. The 8085 has 74 instructions and 246 binary patterns.

13. The entire group of instructions that a microprocessor supports is called instruction set.

14. Each instructions is represented by an 8-bit binary value.

15. These 8-bits of binary value is called op-code or instruction.

16. Classification of Instruction set

    16.1 Data Transfer Group - This group of instructions copies data from a location

16.2 Arithmetic Group-This group of instructions perform operations such as addition, substraction, decrement, or data in register or in memory location

16.3 Logical Group-This group of instructions perform logical operations such as AND, OR EX-OR, compare, rotate complement with contents of the accumulator. Then the result is stored in accumulator.

16.4 Branch Control Group- This group of instruction that change the sequence of program execution using conditional and unconditional jumps, subroutine call, Return and Restart

16.5 Stack I/O and Machine Control Group- This group of instruction includes set of those instruction which can be able to perform function on stack.

7.3 Looping, Counting And Indexing

1. Looping- In this technique, the program is instructed to execute certain set of instructions repeatedly to execute a particular task number of times.

2. Counting- This technique allows programmer to count how many times the instruction/set of instructions are executed.

3. Indexing- This technique allows programmer to point or refer the data stored in sequential memory location one by one.

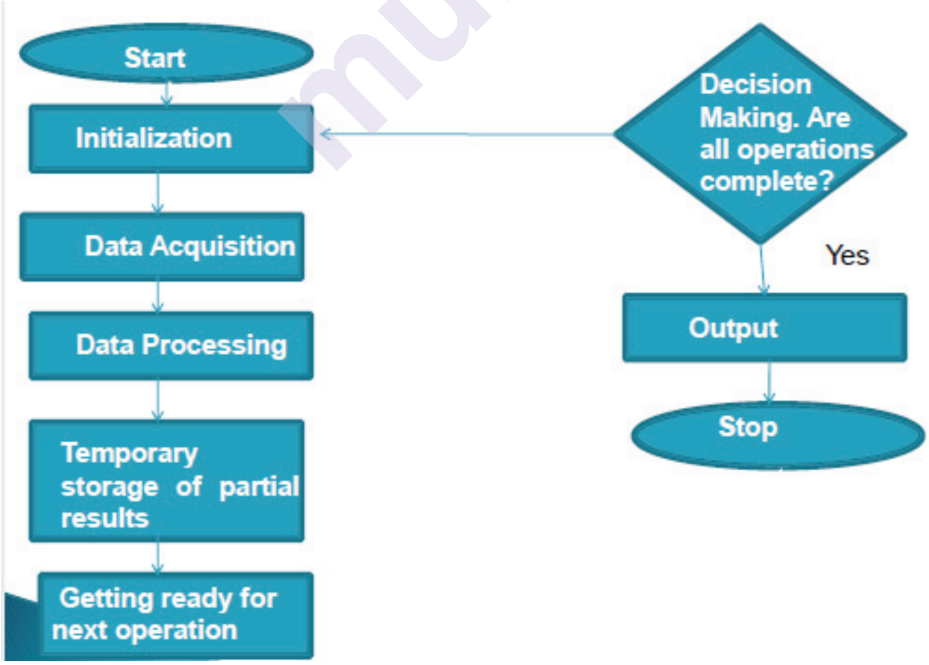

Start
Initialization
Data Acquisition
Data Processing
Temporary storage of partial results
Getting ready for next operation
Decision Making. Are all operations complete?
Yes
Output
Stop

Fig 1 Generalized Programming Flowchart

4. Example 1 Write a program to store 'FFH' in 20 continuous memory locations starting at 4500H.

Instruction
LXI H,4500H
MVI C, 14h
UP: MVI M, FFh
INX H
DCR C
JNZ UP
HLT

The program listed above will continuously add data FF h in memory location starting at 4500H.

First Instruction- LXI H will load the address 4500 H address in H & L

Second Instruction- MVI C will load data 14h in Register C

Third Instruction- Will Copy data FFH in M(H & L) Register continuously with labelled loop UP

Fourth instruction- Increment Memory location.

Fifth Instruction- will decrement the counter register over here is C

Sixth Instruction- Will continuously Jump if no zero.

Seventh Instruction- Will Halt the program

Example 2 Write a program on Bubble Sort

Instruction
Start: LXI B, OFF5H
MVI D, 00H
MVI C,04H
Check: Mov A,M
INX H
CMP M
JC Nxtbt
DCX H
MOV M,A
DCX H
MOV B,M
INX H
MVI D,01H
Nxtbt: DCR C
JNZ Check
Mov A,D
RRC
JC Start
HLT

The above program will compare two numbers and sort the number in ascending order

Example 3 To find the largest number in an array of data using 8085 instruction set

Algorithm

1. Load the address of the first element of the array in HL pair
2. Move the count to B register
3. Increment the pointer
4. Get the first data in A register
5. Decrement the counter
6. Increment the pointer
7. Compare the content of memory addressed by HL pair with that of A register
8. If carry=0, go to step 10 or if carry=1 go to step 9
9. Move the content of memory addressed by HL to A register
10. Decrement the counter
11. Check for Zero of the counter. If ZF=0, go to step 6, or if ZF=1 go to next step
12. Store the largest data in memory
13. Terminate the Program

Instruction	Explanation
LXI H,4200	Set pointer for array
MOV B,M	Load the Count
LOOP: INX H	Increment the Memory location
CMP M	If A register > M go to Head
JNC AHEAD	Jump to AHEAD label if carry=0
MOV A,M	Set new value as largest
AHEAD:DCR B	Decrement the B counter
JNZ LOOP	Repeat comparisons till count=0
STA 4300	Store the largest value at 4300
HLT	Terminate the program

Input 05(4200)
0A(4201)
F1(4202)
1F(4203)
26(4204)
FE(4205)

Output FE(4300)

Example 5 To find the smallest number in an array of data using 8085 instruction set

Algorithm

1. Load the address of the first element of the array in HL pair
2. Move the counttoB–reg.
3. Incrementthe pointer
4. Getthe firstdata inA –reg.
5. Decrementthe count.
6. Incrementthepointer
7. Compare the contentofmemory addressed by HLpairwiththatofA -reg.
8. Ifcarry = 1,go tostep10orifCarry = 0goto step9
9. Move the contentofmemory addressedby HLtoA –reg.
10. Decrementthecount
11. CheckforZero ofthe count. IfZF=0,goto step6,orifZF= 1go to next step.
12. Storethe smallestdata inmemory.
13. Terminatetheprogram.

Program

	LXIMOV INX	H,4200B,M H	Set pointer for array Loadthe Count
	MOV	A,M	Set1stelementaslargestdata
	DCR	B	Decrement the count
LOOP:	INX	H	
	CMP	M	IfA-reg < Mgo toAHEAD
	JC	AHEAD	Jump if carry=1 to AHEAD Label

	MOV	A,M	Set the new value as smallest
AHEAD:	DCR	B	
	JNZ	LOOP	Repeatcomparisonstillcount= 0
	STA	4300	Storethe largestvalue at4300
	HLT		Terminate the program

Input: 05(4200) Array Size
0A(4201)
F1(4202)
1F(4203)
26(4204)
FE(4205)

Output: 0A(4300)

Example 6 To write a program to arrange anarray of descending order

Algorithm

1. Initialize HL pairas memory pointer
2. Get the countat4200intoC–register
3. Copy it in D–register(for bubble sort(N-1)times required)
4. Get the first value in A –register
5. Compare it with the value at next location.
6. If they are out of order, exchange the contents of A–register and Memory
7. Decrement D–register content by 1
8. Repeat steps 5 and 7 till the value in D-register become zero
9. Decrement C –register content by 1
10. Repeat steps 3 to 9 till the value in C –register becomes zero

REPEAT:	LXIMOVD CR MOV	H,4200C,M C D,C
	LXI	H,4201
LOOP:	MOV	A,M
	INX	H
	CMP	M
	JNC	SKIP

	MOV	B,M
	MOV	M,A
	DCX	H
	MOV	M,B
	INX	H
SKIP:	DCR	D
	JNZ	LOOP
	DCR	C
	JNZ	REPEAT
	HLT	

Input:	4200	05 (Array Size)
	4201	01
	4202	02
	4203	03
	4204	04
	4205	05
Output:	4200	05 (ArraySize)
	4201	05
	4202	04
	4203	03
	4204	02
	4205	01

7.4 Additional Data Transfer and 16-Bit Arithmetic Instructions

1. Load Register Pair Immediate- LXI Reg pair, 16-bit data. The instruction loads 16 bit data in the register pair designated in the operand. Eg LXI H,2034H.

2. Load H and L registers direct-LHLD 16-bit address. The instruction copies the contents of the memory location pointed out by the 16-bit address in to register L and Copies the contents of the next memory location in to register H. The contents of source memory locations are not altered. Eg LHLD 2040 H.

3. Mov R.M-R.M copies data byte from memory to register. Memory location, its location is specified by the contents of the HL registers. Eg MOV B,M.

4. LDAX B/D Register pair- The contents of the designated register pair point to a memory location. This location in to the accumulator. The contents of either the register pair or the memory location are not altered. Eg LDAX B

5. Load Accumulator- LDA 16-bit address , the contents of a memory location, specified by a 16-bit address in the operand, are copied to the accumulator. The contents of the source are not altered. Eg LDA 2034H

6. MOV M,R-This instruction copies the contents of the source. The source register are not altered. As one of the operands is a memory location, its location is specified by the contents of the HL registers. Eg MOV M,B

7. STA 16-bit address- The contents of the accumulator are copied in to the memory location specified by the operand. This is a 3-byte instruction, the second byte specifies the low-order address and the third byte. Eg MOV M,B

8. Store Accumulator Indirect- STAX register pair the contents of the accumulator are copied in to the memory location specified by the contents of the operand(register pair). The contents of the accumulator are not altered. Eg STAX B

9. Store H and L registers indirect- SHLD 16-bit address, the contents of register L are stored in to the memory location specified by the 16-bit address in the operand and the contents of the H register are stored in to the next memory location by incrementing the operand. The contents of register HL are not altered. This is a 3 byte instruction, the second byte specifies the low-order address and the third byte specifies the high-order address. Eg SHLD 2470H

10. Increment register pair by 1-INX R, the contents of the designated register pair are incremented by 1 and the result is stored in the same place.
Eg INX H

11. Decrement register pair by 1- DCX R, the contents of the designated register pair are decremented by 1 and the result is stored in the same place.
Eg DCX H

12. Add memory(ADD M)- The contents of the operand (memory) are added to the contents of the accumulator and the result is stored in the accumulator. The operand is a memory location, its location is specified by the contents of the HL registers. All flags are modified to reflect the result of the addition.

13. Substract Memory (SUB M)- The contents of the operand (memory) are substracted to the contents of the accumulator and the result is stored in the accumulator. The operand is a memory location, its location is specified by the contents of the HL registers. All flags are modified to reflect the result of the substraction.

14. Increment memory by 1/ Decrement memory by 1(INR M/DCR M)- The contents of the memory are incremented by 1 using INR and decremented by 1 using DCR and the result is stored in the same place. The operand is a memory location, its location is specified by the contents of the HL registers.

Eg Toperformadditionoftwo8bitnumbersusing8085.

Algorithm

1. Start the program by loading the first data into Accumulator.
2. Move the data to a register (B register).
3. Get the second data and load into Accumulator.
4. Add the two register contents.
5. Check for carry.
6. Store the value of sum and carry in memory location.
7. Terminate the program.

Program

MVI	C, 00	InitializeC registerto00
LDA	4150	Loadthevalue toAccumulator.
MOV	B,A	Movethe contentofAccumulatortoBregister.
LDA	4151	Loadthevalue toAccumulator.
ADD	B	Addthe value ofregisterBto A
JNC	LOOP	Jumponno carry.
INR	C	Incrementvalue ofregisterC
LOOP:STA	4152	StorethevalueofAccumulator(SUM).
MOV	A, C	Move contentofregisterC toAcc.
STA	4153	Storethevalue ofAccumulator(CARRY)
HLT		Haltthe program.

Eg 2 Toper form the subtraction of two 8bit numbers using 8085.

Input:	80(4150)
	80(4251)
Output:	00(4152)
	01(4153)

Algorithm

1. Start the program by loading the first data into Accumulator.
2. Move the data to a register(B register).
3. Get the second data and load in to Accumulator.
4. Subtract the two register contents.
5. Check for carry.
6. If carry is present take 2's complement of Accumulator.
7. Store the value of borrow in memory location.
8. Store the difference value (present in Accumulator) to a memory
9. location and terminate the program.

Program

MVI	C, 00	Initialize C to 00
LDA	4150	Loadthe value to Acc.
MOV	B,A	Move the contentof Acc to Bregister.
LDA	4151	Loadthe value to Acc.
SUB	B	
JNC	LOOP	Jumpon no carry.
CMA		Complement Accumulator contents.
INR	A	Increment value in Accumulator.
INR	C	Increment value in register C
LOOP:STA	4152	Storethe value of A-reg to memory address.
MOV	A, C	Move contents of register C to Accumulator.
STA	4153	Storethe value of Accumulator memory address.
HLT		Terminate the program.

Input: 06(4150)
02(4251)

*Output:*04(4152)
01(4153)

Toper form the multiplication of two 8 bit numbers using 8085.

ALGORITHM:

1. Start the program by loading HL register pair with address of memory location.
2. Move the data to a register (B register).
3. Get the second data and load in to Accumulator.
4. Add the two register contents.
5. Check for carry.
6. Increment the value of carry.
7. Check whether repeated addition is over and store the value of product and carry in memory location.
8. Terminate the program.

PROGRAM:			
	MVI	D,00	InitializeregisterDto00
	MVI	A, 00	InitializeAccumulatorcontentto00
	LXI	H,4150	
	MOV	B,M	Get the first number in B-reg
	INX	H	
	MOV	C, M	Get the second number in C-reg.
LOOP:	ADD	B	Add content of A- reg to register B.
	JNC	NEXT	Jump on no carry to NEXT.
	INR	D	Increment content of register D
NEXT:	DCR	C	Decrement content of register C.
	JNZ	LOOP	Jump on no zero to address
	STA	4152	Store the result in Memory
	MOV	A, D	
	STA	4153	Store the MSB of result in Memory
	HLT		Terminate the program.

Eg 4 Toper form the division of two 8bit numbers using 8085.

Algorithm:

1. Start the program by loading HL register pair with address of memory location.
2. Move the data to a register (B register).
3. Get the second data and load in to Accumulator.
4. Compare the two numbers to check for carry.
5. Subtract the two numbers.
6. Increment the value of carry .
7. Check whether repeated subtraction is over and store the value of product and carry in memory location.
8. Terminate the program.

PROGRAM:			
	LXI	H,4150	
	MOV	B,M	Getthedividendin B– reg.
	MVI	C, 00	ClearC –regforqoutient
	INX	H	
	MOV	A, M	Getthe divisorin A –reg.
NEXT:	CMP	B	CompareA -regwithregisterB.
	JC	LOOP	Jumponcarry toLOOP
	SUB	B	SubtractA–regfromB-reg.
	INR	C	IncrementcontentofregisterC.
	JMP	NEXT	JumptoNEXT
LOOP:	STA	4152	Storethe remainderinMemory
	MOV	A, C	
	STA	4153	Storethe quotientinmemory
	HLT		Terminatetheprogram.

Input: FF(4150)
FF(4251)

Output: 01(4152) Remainder
FE(4153) Quotient

7.5 Logic Operations: Rotate,Logics Operations: Compare, Dynamic Debugging

1. Logic instructions of a microprocessor are simply the instructions that carry out basic logical operations such as OR, AND, XOR and So on. In intel's 8085 microprocessor, the destination operand for the instructions is always the accumulator register. Here, the logical operations work on a bitwise level. The corresponding result is also stored in the accumulator register.

2. Following is the table showing the list of logical instruction

Sr. No	OP Code	Operand	Destination	Explanation
1	ANA	R	A=A AND R	ANA B
2	ANA	M	A=A AND MC (Memory Content)	ANA 2050
3	ANI	8-bit Data	A=A AND 8-bit data	ANI 50
4	ORA	R	A=A OR R	ORA B
5	ORA	M	A=A OR MC	ORA 2050
6	ORI	8-bit data	A=A OR 8-bit data	ORI 50
7	XRA	R	A=A XOR R	XRA B
8	XRA	M	A= A XOR MC	XRA 2050
9	XRI	8 bit data	A=A XOR 8-bit data	XRI 50
10	CMA	None	A=1's	CMA
11	CMP	R	Compares R with A and triggers	CMP B
12	CMP	M	Compares MC with A and triggers the flag register	CMP 2050
13	CPI	8-bit data	Compares 8-bit data with A and triggers the flag register	CPI 50
14	RRC	none	Rotate accumulator right without carry	RRC
15	RLC	None	Rotate accumulator left without carry	RLC
16	RAR	none	Rotate accumulator right with carry	RAR
17	RAL	none	Rotate accumulator left with carry	RAL
18	CMC	none	Compliments the carry flag	CMC
19	STC	none	Sets the carry flag	STC

3. Debugging is the process of identifying and removing bug from software or program. It refers to identification of errors in the program logic, machine codes and execution. It gives step by step information about the execution of code to identify the fault in the program.

   3.1 Debugging of Machine code-Translating the assembly language to machine code is similar to building a circuit from a schematic diagram. Debugging can help in determining

   3.1.1 Value of register

   3.1.2 Flow of Program

   3.1.3 Entry and exit point of a function

   3.1.4 Entry in to if or else statement

   31.5 Logging of code

   3.1.6 Calculation check

   3.2 Common Sources of Error

   1. Selecting a wrong code
   2. Forgetting second or third byte of instruction
   3. Specifying wrong jump locations
   4. Not reversing the order of high and low bytes in a jump instruction
   5. Writing memory addresses in decimal instead of hexadecimal
   6. Failure to clear accumulator when adding two numbers
   7. Failure to clear carry registers
   8. Failure to set flag before jump instruction
   9. Specifying wrong memory address on Jump instruction
   10. Use of improper combination of rotate instructions

   3.3 The debugging process is divided in to two parts

   1 Static Debugging- It is similar to visual inspection of circuit board, it is done by a paper and pencil to check the flow chart and machine codes. It is used to the understanding of code logic and structure of program

   2. Dynamic Debugging- It involves observing the contents of register or output after execution of each instruction (in single step technique) or a group of instructions(in breakpoint technique).

3.4 In single board microprocessor, techniques and tools commonly used in dynamic debugging are-

1. Single Step- This technique allows to execute one instruction at a time and observe the results of each instruction. Generally, this is build using hard-wired logic circuit. As we press the single step run key we will be able to observe the contents of register and memory location. However, if there is large loop then single step debugging can be very tiring and time-consuming. So instead of running the loop n times, we can reduce the number of iteration to check the effectiveness of the loop. The single step technique is very useful for short programs. This helps to spot: A)incorrect addresses B)incorrect jump location in loops C)incorrect data or missing codes.

2. Break Point- The breakpoint facility is usually a software routine that allows users to execute a program in sections. The breakpoints can be set using RST instruction. When we push the Execute key, the program will be executed till the breakpoint. The registers can be examined for the expected result. With the breakpoint facility, isolate the segment of program with errors. Then that segment can be debugged using the single-step facility. It is usually used to check: a) Timing loop b) I/O section c) Interrupts

3. Register Examine- The register examine key allows you to examine the contents of the microprocessor register. This technique is used in conjunction with either single-step or breakpoint facility.

Miscellaneous Questions

Q1. Write down some arithmetic Instruction with illustrative program

Q2. Write Logical instructions with program

Q3. Describe the concept about dynamic debugging

Q4. Describe the types of dynamic debugging

❖❖❖

UNIT 3

8

COUNTER & TIME DELAYS

Unit Structure

8.1 Objectives

8.2 Introduction

8.3 Counters and Time Delays, Illustrative Program: Hexadecimal Counter

8.4 Illustrative Program: zero-to-nine (Modulo Ten) Counter

8.5 Generating Pulse Waveforms

8.6 Debugging Counter and Time-Delay Programs

8.1 Objectives

At the end of this chapter, the student will be able to

- Describe about Counter and Delays
- Write the program Hexadecimal counter
- Write the program for modulo ten counter
- Illustrate the concept of Pulse waveform
- Elaborate on concept of debugging and Time-Delay Programs

8.2 Introduction

1. The delay will be used in different places to simulate clocks, or counters or some other area.

2. When the delay subroutine is executed, the microprocessor does not execute other tasks. For the delay we are using the instruction execution times, executing some instructions in a loop, the delay is generated.

3. So in this chapter we are going to study more about counters, time delays and various other programs of 8085 for hexadecimal counter, modulo ten counter, pulse wave form.

8.3 Counters and Time Delays, Illustrative Program: Hexadecimal Counter

1. Counters are used to keep track of events

2. Time delays are important in setting up reasonably accurate timing between two events

3. A Counter is designed simply by loading an appropriate number in to one of the registers and using the INR(Increment by one) or the DCR(Decrement by one) instructions. A loop is established to update the count and each count is checked to determine whether it has reached the final number, if not, the loop is repeated.

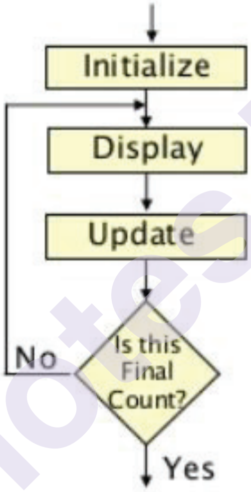


Fig. 1 Flow chart of Counter

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

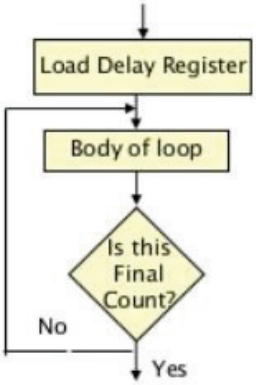


Fig 2 Flow chart of Time Delay

5. Calculating Time Delays- Each instruction passes through different combinations of opcode fetch, memory read and memory write cycles

6. Knowing the combination of cycles, one can calculate how long such an instruction would require to complete with respect to number of bytes, number of machine cycles, number of T-State.

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

   Time Delay=No of T-States * Clock Period

8. For example - MVI instruction uses 7 T States. Therefore, if the microprocessor is running at 2 MHZ, the instruction would require 3.5 microsecond to complete.

9. We can design time delay using following three techniques

   9.1 Using One Register

   9.2 Using a Register Pair

   9.3 Using a Loop with in a Loop

10 Using One Register- A Count is loaded in a register, and we can use a loop to produce a certain amount of time delay in a program.

   10.1 The following is an example of a delay using one register

```
MVI C, FFH  7T 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 7T-States

   The following two instructions form a loop that requires 14T States to execute and is repeated 255 times until C becomes 0.

   10.2 We need to keep in mind though that in the last iteration of the loop, the JNZ instruction will fail and require only 7T States rather than the 10.

   10.3 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

   Tdelay= TO + TL

Whereas Tdelay= Total delay, TO=delay outside the loop and TL=delay of the loop

TO is the sum of all delays outside the loop and TL is calculated using the formula

TL=T*Loop T-States*N(No of iterations)

10.4 Using a single register, one can repeat a loop for a maximum count of 255 times.

10.5 It is possible to increase this count by using a register pair for the loop counter instead of the single register.

10.5 A minor problem arises in how to test for the final count since DCX and INX do not modify the flags.

10.6 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

11 Using a Register Pair- The Following is an example of a delay loop set up with a register pair as the loop counter.

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

11.1 Using the same formula from before, we can calculate To=10 T Sates(The delay for the LXI instruction)

TL=(24 * 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 sates for the JNZ in the last iteration).

TDelay=(10 +98301) * 0.5 m sec=49.155 msec

12 Using a Loop with in a loop- Nested loops can be easily set up in assembly language by using two registers for the two loop counters and updating the right register in the right loop.

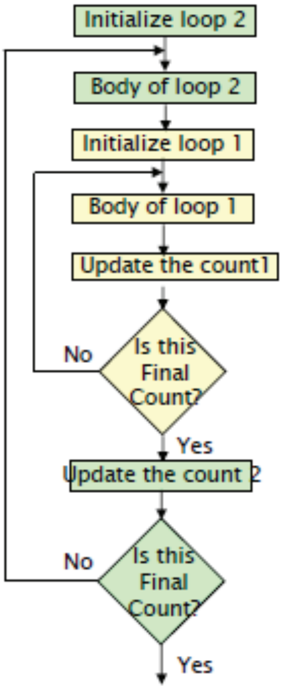


Fig 3 Flow chart for time delay with two loops

12.1 Instead (or in conjunction with) Register Pairs, anested loop structure can be used to increase thetotal 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
```

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

12.3 Delay of inner loop

TO1=7 T States (MVI C, FFH)

TL1=(255*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)
TLoop1= 7+3567=3574 T States

Delay of the outer loop

TO2=7 T States

(MVI B, 10H)

TL1= (16 *(14+3574))-3=57405 T-States (14 T Sates for the DCR B and JNZ instructions and 3574 T States for loop1 repeated 16 times minus 3 for the final JNZ

T Delay=7+57405 = 57412 T Sates

Total Delay(T Delay)= 57412*0.5 micro Sec=28.706 Msec

13. Increasing the Time Delay- The Delay can be further increased by using register pairs for each of the loop counters in the nested loops set up. It can also be increased by adding dummy instructions(like NOP) in the body of the Loop.

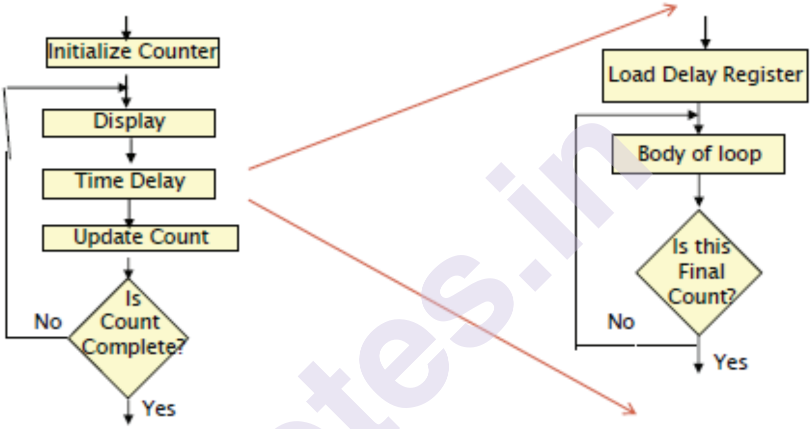


Fig 4 Counter Design with Time Delay

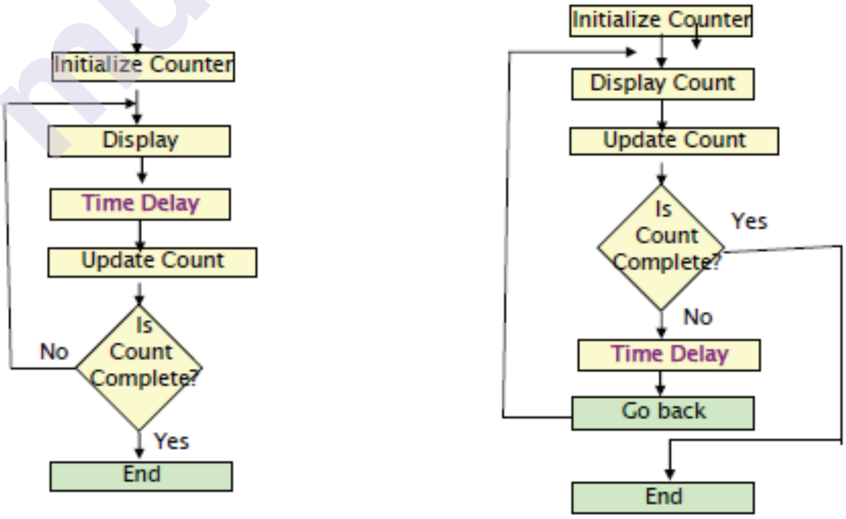


Fig 5 Variations of Counter Flow chart

14 Illustrative Program on Hexadecimal Counter

Write a program to count continuously in hexadecimal from ffh to 00h in a system with a 0.5 micro sec clock period. Use register c to set up a one millisecond delay between each count and display the numbers at one of the Output ports.

14.1 This Problem has two parts, the first is to set up a continuous down-counter and the second is to design a given delay between two counts. The hexadecimal counter is set up by loading a register with an appropriate starting number and decrementing it until it becomes zero. After zero count, the register goes back to FF because decrementing zero results in a (-1), which is FF in 2's Complement. The 1 ms delay between each count is set up by using delay techniques.

	MVI B,00H	Store the 00 H in B register and initialize a counter
NEXT:	DCR B	Decrement the B counter
	MVI C, COUNT	Load register C with Delay count
DELAY:	JNZ DELAY	
	MOV A,B	Copy the Contents of B to A
	OUT PORT #(number of port)	Display the output at port
	JMP NEXT	

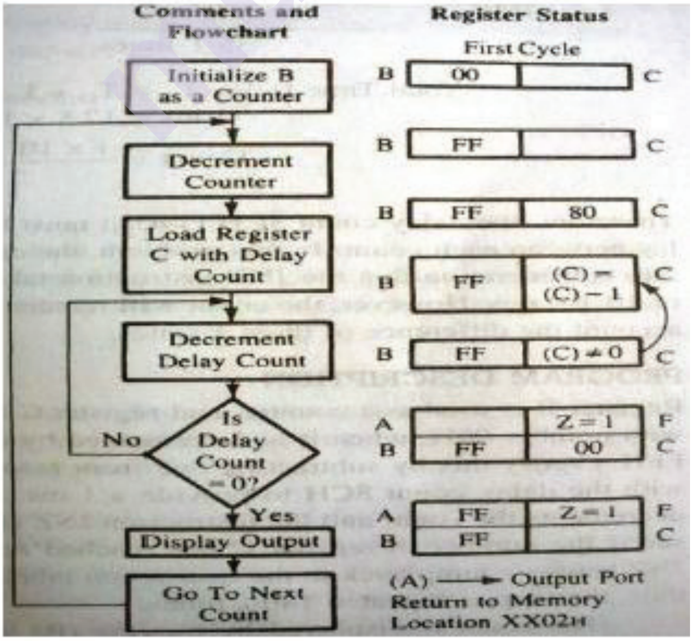


Fig 6 Flow chart for Hexadecimal Counter

Time Delay Calculation - Delay loop includes two instructions: DCR C andJNZ with 14 T-states. Therefore the time delay TL inthe loop (without accounting for the fact that JNZrequires 7 T-States in the last cycle, because countwill remain same even if the calculations take intoaccount the difference of 3 T-States) is:

TL = 14 T-states X T (Clock period) X Count
= 14 X (0.5 x 10-6) x Count
= (7.0 X 10-6) X Count
The Delay outside the loop includes the following instructions:
DCR B 4T Delay outside the loop:
MVI C,COUNT 7T To = 35 T-States * T
MOV A,B 4T = 35 * (0.5* 10-6)
OUT PORT 10T   = 17.5 µs
JMP 10T
    35 T-States
Total Time Delay TD = To + TL
1ms= 17.5 * 10-6 + (7.0 *10-6) *Count
Count= 1x 10-3 – 17.5 x 10-6 ≈14010 ms
7.0 x 10-6

Hence, a delay count 8CH(14010) must be loaded in C.

8.4 Illustrative Program: Zero-to-Nine (Modulo Ten) Counter

Write A Program To Count From0-9 With A One-Second DelayBetween Each Count. At Count Of9, The Counter Should ResetItself To 0 And Repeat The Sequence Continuously. UseRegister Pair Hl To Set Up TheDelay, And Display Each Count AtOne Of The Output Ports. AssumeThe Clock Frequency Of TheMicrocomputer Is 1 Mhz.

START:	MVI B,OOH	7 T(no of states)	Initialize the B as a counter
	MOV A,B	4 T	Copy the contents of B to A
DISPLAY:	OUT PORT #	10 T	Output will be displayed at particular port
	LXI H, 16 bit	10 T	Load 16 bit data in H and L pair
LOOP:	DCX H	6T	Decrement the H

	MOV A,L	4T	Copy the contents of L to A
	ORA H	4T	Oring the data of H
	JNZ LOOP	10/7	Jump if no zero
	INR B	4	Increment the B counter
	CPI 0AH	7	Compare the data
	JNZ DISPLAY	10/7	
	JZ START	10/7	

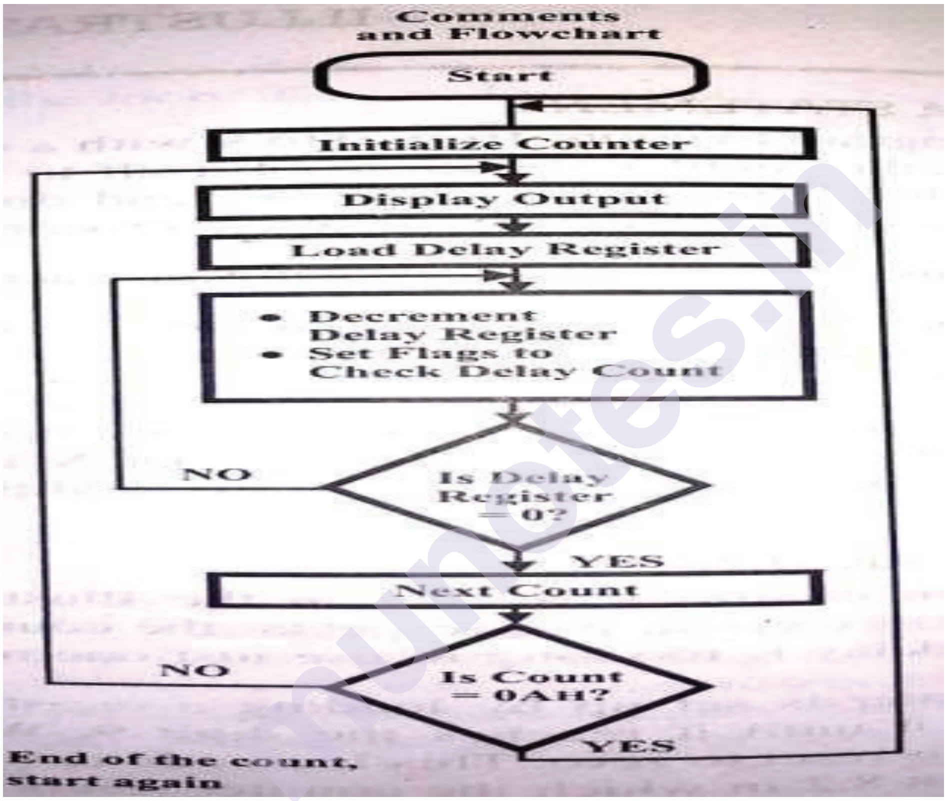


Fig 6 Flow chart Modulo Ten Counter

Time Delay Calculation- The major delay between two counts is provided by the 16-bit number in the delay register HL(inner loop in flow chart). This delay is set up by using a register pair.

Loop Delay TL= 24 T-states *T * Count

1 second= 24*1.0 *10-6 *Count

Count= 1 = 41666 = A2C2H

24 x 10-6

A2C2H would provide approx 1 sec delay between two counts. To achieve higher accuracy in the delay, the instructions outside the loop must be accounted for delay calculation. (will be 41665).

8.5 Generating pulse Wave forms

Write A Program To Generate A Continuous Square Wave With the Period Of 500Micro Sec. Assume The System Clock Period is 325 Ns, And Use Bit D0 To Output The Square Wave.

	MVI D,AA	7T	Move immediate AA data to D register
ROTATE:	MOV A,D	4T	Copy the data of accumulator to register
	RLC	4T	Rotate left with Carry
	MOV D,A	4T	Move the content from A to D
	ANI 01H	7T	And the contents of A register with 01 H
	OUT PORT 1	10 T	Display the output at port 1
	MVI B, COUNT		Keep the count in B
DELAY:	DCR B	4T	Decrement B
	JNZ DELAY	10/7T	
	JMP ROTATE	10T	

A 1 0 1 0 1 0 1 0
After RLC 0 1 0 1 0 1 0 1
A AND 01H 0 0 0 0 0 0 0 1
COUNT= 52.410 = 34H

8.6 Debugging Counter and Time-Delay Programs

1. Errors in counting T-States in a delay loop. Typically, the firstinstruction – to load a delay register – is mistakenly includedin the loop.
2. Errors in recognizing how many times a loop is repeated.
3. Failure to convert a delay count from a decimal number intoits hexadecimal equivalent.

4. Conversion error in converting a delay count from decimal tohexadecimal number or vice versa.
5. Specifying a wrong jump location.
6. Failure to set a flag, especially with 16-bitdecrement/increment instructions.
7. Using a wrong jump instruction.
8. Failure to display either the first or the last count.
9. Failure to provide a delay between the last and the last-but one count.

Miscellaneous Questions

Q1. Write in short about Counter & Time Delays in 8085

Q2. Write a program for hexadecimal counter

Q3. How to calculate time delay for a particular program

Q4. Write a Program for Modulo ten counter

Q5. How to write a program for generating pulse wave form

Q6. Write the steps for debugging counter and Timer delay programs

UNIT 3

9

STACKS AND SUB-ROUTINES

Unit Structure

9.1 Objectives

9.2 Introduction

9.3 Stack, Subroutine, Restart, Conditional Call, Return Instructions

9.4 Advanced Subroutine concepts

9.1 Objectives

At the end of this unit, the student will be able to

- Describe the concept of stack and its instruction
- Illustrate more about subroutines

9.2 Introduction

1. The stack is an area of memory identified by the programmer for temporary storage of information.
2. The stack is a LIFO (Last In First Out.) structure.
3. The stack normally grows backwards into memory.
4. In other words, the programmer defines the bottom of the stack and the stack grows up into reducing address range.

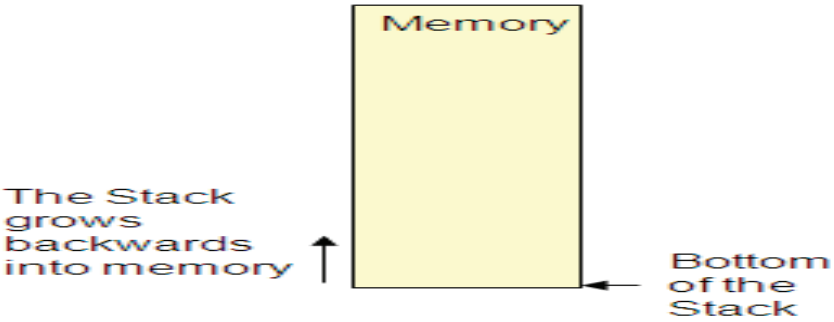


Fig 1 Stack Structure

5. 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).

6. Saving Information on the Stack

   1. Information is saved on the stack by Pushing it on.
   2. It is retrieved from the stack by Poping it off.
   3. The 8085 provides two instructions: PUSH and POP for storing information on the stack and retrieving it back.
   4. Both PUSH and POP work with register pairs ONLY.

7. The PUSH Instruction

PUSH B/D/H/PSW

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

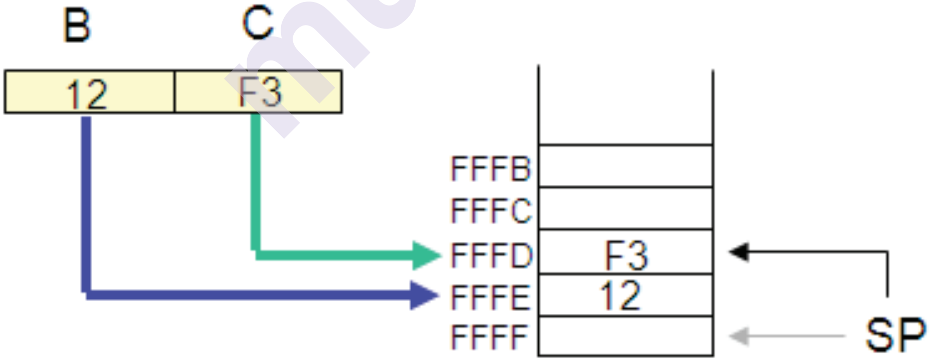


Fig 2. Push Operation of Stack

8. The POP Instruction

POP B/D/H/PSW

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

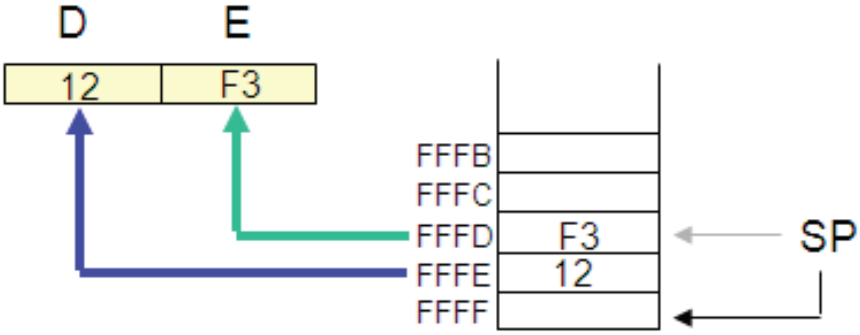


Fig.3 - POP operation on Stack

9. Operation of the Stack

1. 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".

2. 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.

3. 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.

4. 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 andthe program will fail.It is not advisable to place PUSH or POP inside a loop.

5. Program to Reset and display Flags

Clear all Flags.

Load 00H in the accumulator, and demonstrate that the zero flag is not affected by data transfer instruction.

Logically OR the accumulator with itself to set the Zero flag, and display the flag at PORT1 or store all flags on the stack.

6. Program to Reset and display Flags

XX00 LXI SP,XX99H Initialize the stack
03 MVI L,00H Clear L
05 PUSH H Place (L) on stack
06 POP PSW Clear Flags
07 MVI A, 00H Load 00H
09 PUSH PSW Save Flags on stack
0A POP H Retrieve flags in L
0B MOV A, L
0C OUT PORT0 Display Flags (00H)
0E MVI A, 00H Load 00H Again

7. Program to Reset and display Flags

XX10 ORA A Set Flags and reset(CY, AC)
11 PUSH PSW Save Flags on Stack
12 POP H Retrieve Flags in L
13 MOV A, L
14 ANI 40H Mask all Flags except Z
16 OUT PORT1 Displays 40H
18 HLT End of Program

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
S	Z		AC		P		CY

Fig. 4 - Flags affected after executing above program

9.3 Stack, Subroutine, Restart, Conditional Call, Return Instructions

1. Subroutines

1. A subroutine is a group of instructions that will be used repeatedly in different locations of the program.
2. Rather than repeat the same instructions several times, they can be grouped into a subroutine that iscalled from the different locations.
3. In Assembly language, a subroutine can exist anywhere in the code.
4. However, it is customary to place subroutines separately from the main program.
5. The 8085 has two instructions for dealing with subroutines.
6. The CALL instruction is used to redirect program execution to the subroutine.
7. The RTE instruction is used to return the execution to the calling routine.

2. The CALL Instruction

1. CALL 4000H

   1.1 3-byte instruction.

   1.2 Push the address of the instruction immediately following theCALL onto the stack and decrement the stack pointer register by two.Load the program counter with the 16-bit address supplied with the CALL instruction.

   1.3 Jump Unconditionally to memory location.

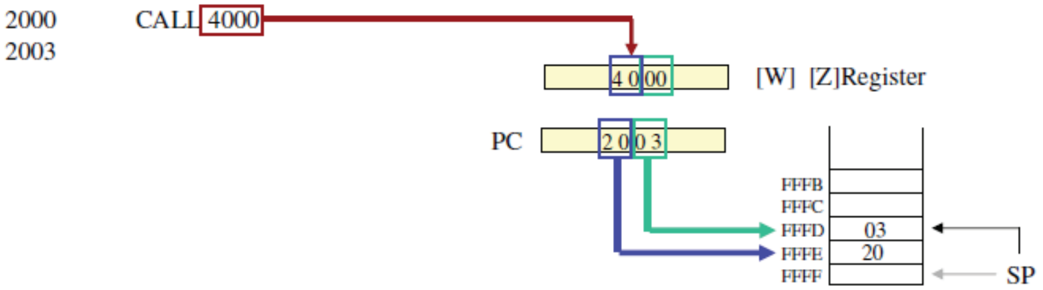


Fig 5 CALL Instruction

2. MP reads the subroutine address from the next two memory location and stores the higher order 8 bit of the address in the W register and stores the lower order 8 bit of the address in the Z register.

3. PUSH the address of the instruction immediately following the CALL on to the stack (Return address).

4. Loads the program counter with the 16-bit address supplied with the CALL instruction from WZ register.

3. The RTE Instruction

1. It is 1-byte instruction

2. Retrieve the return address from the top of the stack and increments stack pointer register by two.

3. Load the program counter with the return address.

4. Unconditionally returns from a subroutine.

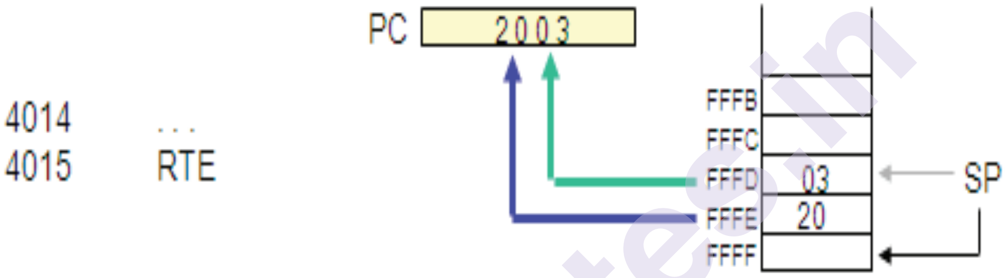
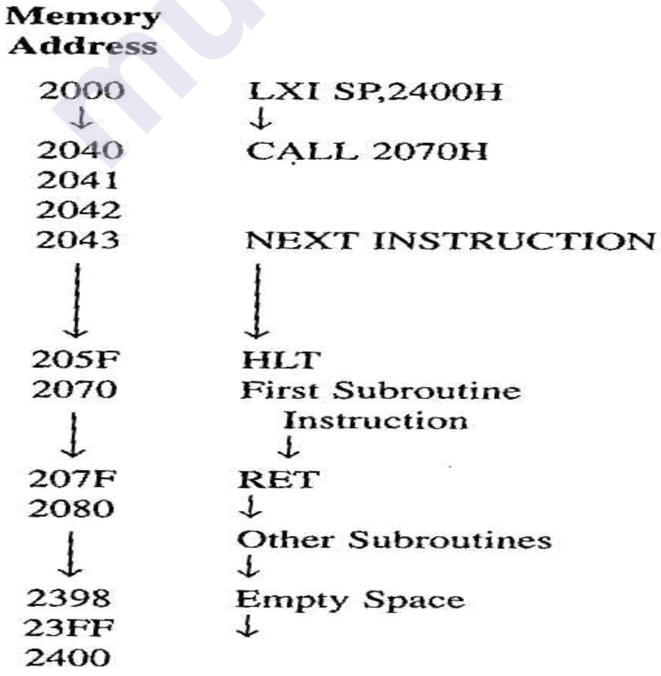


Fig. 6 - RTE Instruction

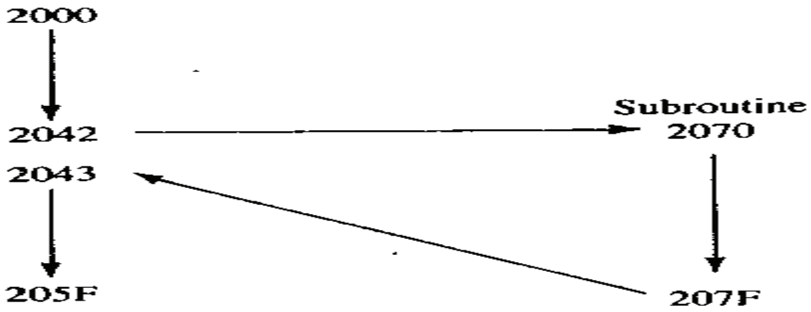
4. Illustrates the exchange of information between stack and Program Counter



```
Memory
Address
 2000        LXI SP,2400H
  ↓          ↓
 2040        CALL 2070H
 2041
 2042
 2043        NEXT INSTRUCTION
  ↓          ↓
 205F        HLT
 2070        First Subroutine
               Instruction
  ↓             ↓
 207F        RET
 2080        ↓
  ↓          Other Subroutines
             ↓
 2398        Empty Space
 23FF        ↓
 2400
```

Fig. 7 - Program

Program Execution



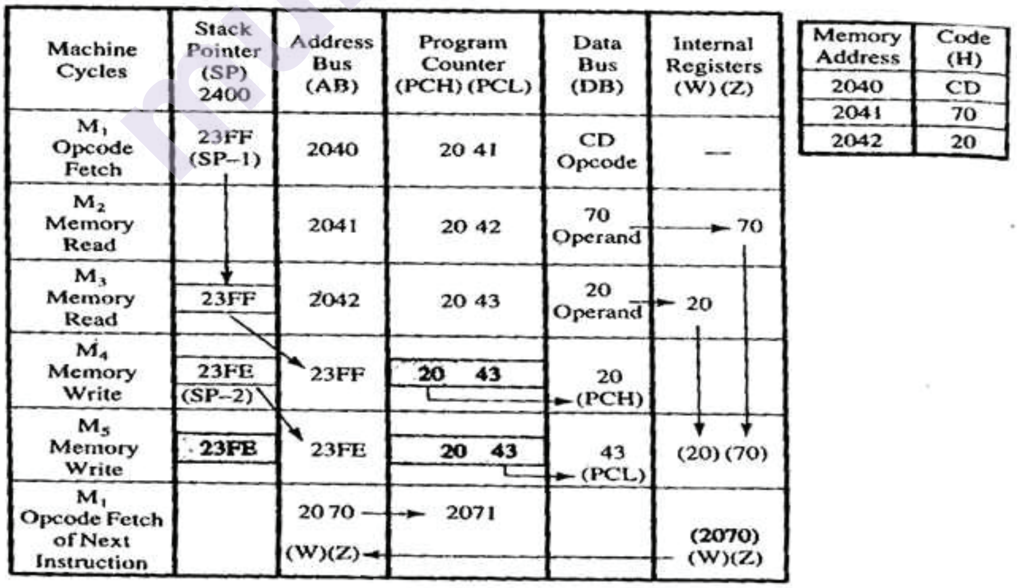
Memory Address	Machine Code	Mnemonics	Comments
2040	CD	CALL 2070H	;Call subroutine located at the memory
2041	70		; location 2070H
2042	20		
2043	NEXT	INSTRUCTION	

Fig. 8 - Program execution with CALL instruction

5. CALL Execution

Instruction requires five machine cycles and eighteen Tstates:Call instruction is fetched, 16-bit address is read during M2 and M3 and stored temporarily in W/Zregisters. In next two cycles content of program counter are stored on the stack (address from where microprocessor continue it execution of program after completion of the subroutine)

Instruction: CALL 2070H



Machine Cycles	Stack Pointer (SP) 2400	Address Bus (AB)	Program Counter (PCH) (PCL)	Data Bus (DB)	Internal Registers (W) (Z)
M$_1$ Opcode Fetch	23FF (SP–1)	2040	20 41	CD Opcode	—
M$_2$ Memory Read		2041	20 42	70 Operand	70
M$_3$ Memory Read	23FF	2042	20 43	20 Operand	20
M$_4$ Memory Write	23FE (SP–2)	23FF	20 43	20 (PCH)	
M$_5$ Memory Write	23FE	23FE	20 43	43 (PCL)	(20) (70)
M$_1$ Opcode Fetch of Next Instruction		20 70 (W)(Z)	2071		(2070) (W)(Z)

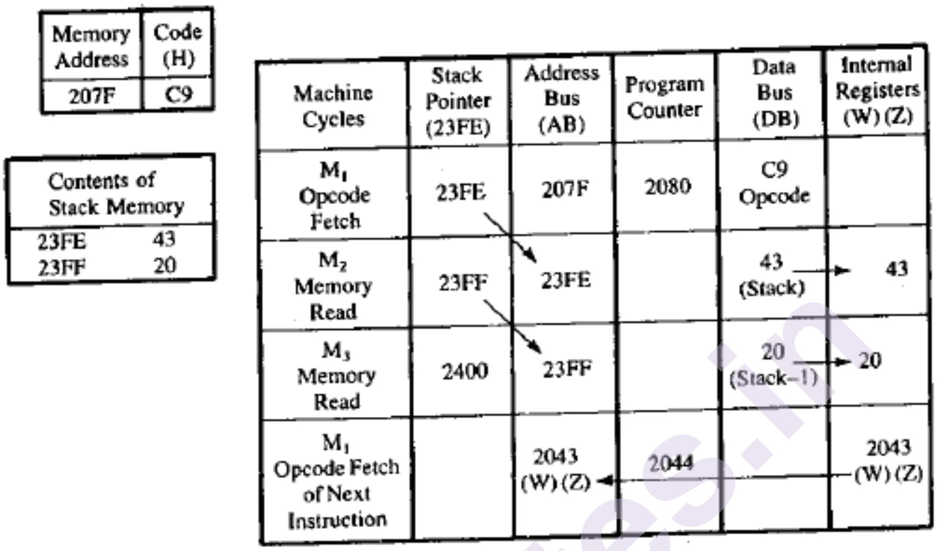
Memory Address	Code (H)
2040	CD
2041	70
2042	20

Data Transfer During the Execution of the CALL Instruction

Fig. 9 - Data transfer during execution of CALL instruction

6. RET Execution

Program execution sequence is transferred to the memory location 2043H location.M1 is normal fetch cycle during M2 contents of stack pointer are placed on address bus so 43H data is fetched and stored on Z register and SP is upgraded.Similarly for M3. Program sequence is transfered to2043H by placing contents of memory and stack



Memory Address	Code (H)
207F	C9

Contents of Stack Memory	
23FE	43
23FF	20

Machine Cycles	Stack Pointer (23FE)	Address Bus (AB)	Program Counter	Data Bus (DB)	Internal Registers (W) (Z)
M₁ Opcode Fetch	23FE	207F	2080	C9 Opcode	
M₂ Memory Read	23FF	23FE		43 (Stack)	43
M₃ Memory Read	2400	23FF		20 (Stack–1)	20
M₁ Opcode Fetch of Next Instruction		2043 (W) (Z)	2044		2043 (W) (Z)

Data Transfer During the Execution of the RET Instruction

Fig. 10 - Data transfer during execution of RET instruction

7. Passing Data to a Subroutine

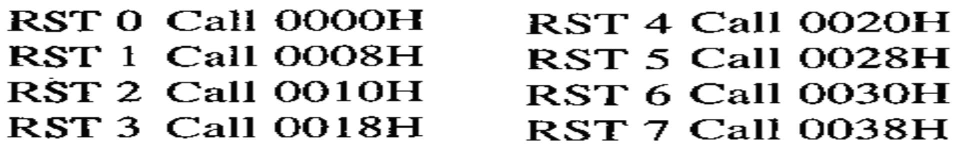
1. In Assembly Language data is passed to a subroutine through registers.
2. The data is stored in one of the registers by the calling program and the subroutine uses the value from the register.
3. The other possibility is to use agreed upon memory locations.
4. The calling program stores the data in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

8. Restart, Conditional Call & Return Instructions

1. In addition to unconditional CALL and RET instructions, the 8085 instruction set includes eight restart instructions and eight conditional CALL and Return instructions.
2. RST Instruction

   2.1 RST instruction are 1-byte call instructions that transfer the program execution to a specific location on page 00H

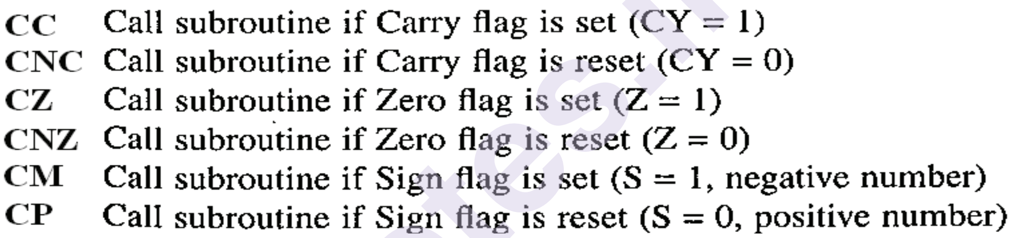
2.2 Executed the same way as call instructions, the 8085 stores the contents of the program counter(the address of the next instruction) on the top of the stack and transfers the program to the restart location.



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

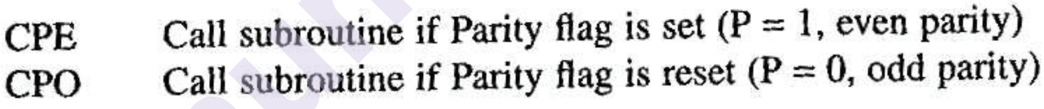
Fig. 11 - 7 types of RST

3. The conditional Call and Return instructions are based on four data conditions(flags): Carry, Sign and Parity. In case of a conditional call the program is transferred to the subroutine if condition is met. In case of a conditional return instruction, the sequence returns to the main program if the condition is met.



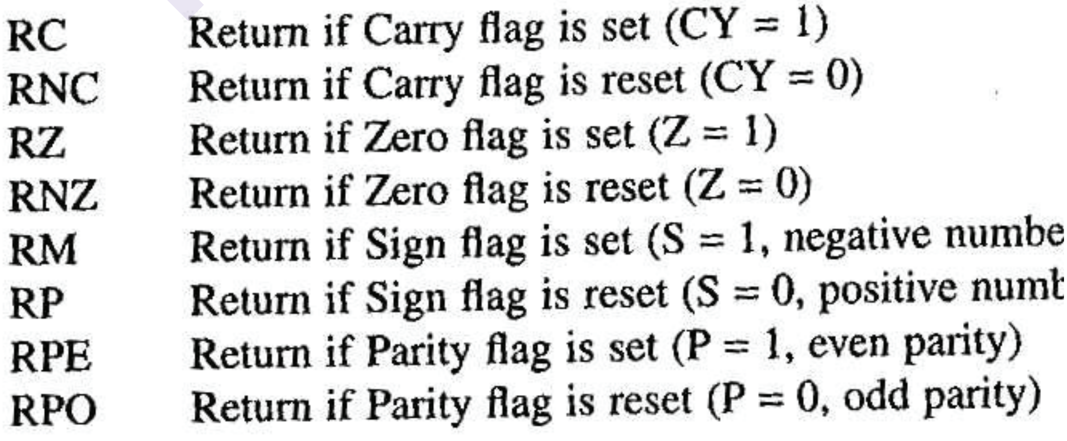
CC	Call subroutine if Carry flag is set (CY = 1)
CNC	Call subroutine if Carry flag is reset (CY = 0)
CZ	Call subroutine if Zero flag is set (Z = 1)
CNZ	Call subroutine if Zero flag is reset (Z = 0)
CM	Call subroutine if Sign flag is set (S = 1, negative number)
CP	Call subroutine if Sign flag is reset (S = 0, positive number)

Fig. 12 - Conditional CALL



CPE	Call subroutine if Parity flag is set (P = 1, even parity)
CPO	Call subroutine if Parity flag is reset (P = 0, odd parity)

Fig. 13 - CALL instruction for subroutine



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

Fig. 14 - Conditional Return Instructions

9.4 Advanced Subroutine Concepts

1. According to Software Engineering practices, a proper subroutine:Is only entered with a CALL and exited with an RTE

2. Has a single entry point

3. Do not use a CALL statement to jump into different points of the same subroutine.

4. For eg Writing Subroutines

   Write a Program that will display FF and 11 repeatedly on the seven segment display. Write a 'delay' subroutine and Call it as necessary.

C000: LXI SP, FFFF
C003: MVI A, FF
C005: OUT 00
C007: CALL C014
C00A: MVI A, 11
C00C: OUT 00
C00E: CALL 1420
C011: JMP C003

4.1 Writing Subroutines

DELAY: C014: MVIB, FF
C016: MVIC, FF
C018: DCR C
C019: JNZ C018
C01C: DCR B
C01D: JNZ C016
C020: RET

5. Nesting Subroutines

   1. The Programming technique of a subroutine calling another subroutine.

   2. This process is limited only by the number of available stack locations.

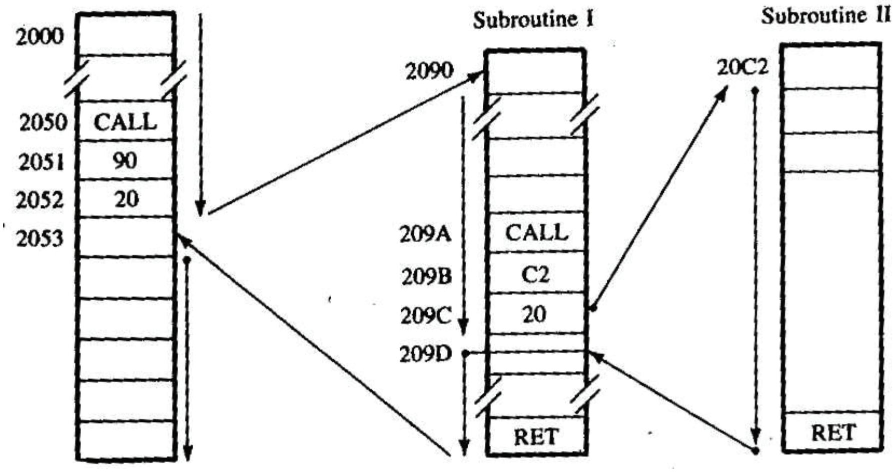
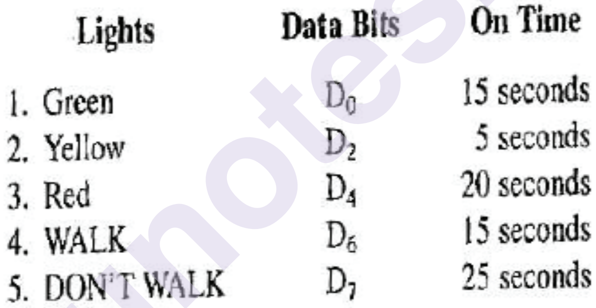


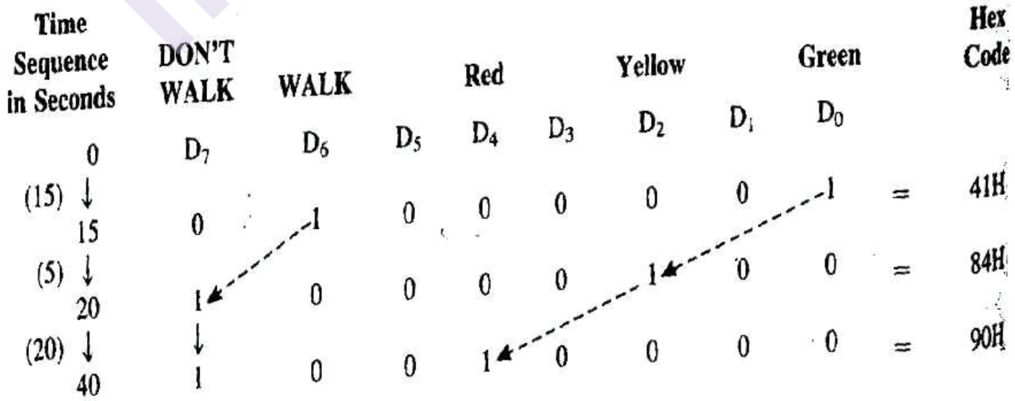
Fig. 15 - Nested Subroutines

6. Write a program to provide the given on /off time to three traffic lights (Green, Yellow and Red) and two pedestrian signs(Walk and Don't Walk). The signal lights and signs are turned on /off by the data bits of an output port



Lights	Data Bits	On Time
1. Green	D_0	15 seconds
2. Yellow	D_2	5 seconds
3. Red	D_4	20 seconds
4. WALK	D_6	15 seconds
5. DON'T WALK	D_7	25 seconds

Fig. 16 - Input to a program



Time Sequence in Seconds	DON'T WALK	WALK		Red		Yellow		Green		Hex Code
	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0		
0 (15) ↓ 15	0	1	0	0	0	0	0	1	=	41H
(5) ↓ 20	1	0	0	0	0	1	0	0	=	84H
(20) ↓ 40	↓ 1	0	0	1	0	0	0	0	=	90H

Fig. 17 - Schematic execution of a program

Label	Mnemonics	Comments
	LXI SP, 0099H	Initalize stack pointer at location 0099H
START	MVI A, 41H	Load accumulator with the bit pattern for green light and Walk Sign
	OUT PORT #	Turn on Green light and Walk sign, that is displayed on out port number
	MVI B,0FH	Use B as a counter to count 15 seconds, B is decremented in the subroutine
	CALL DELAY	Call delay subroutine located at 50H
	MVI A, 84H	Load accumulator with the bit for Yellow light and Don’t Walk
	OUT PORT #	Turn on Yellow light and Don’t Walk
	MVI B,05	Set up 5 second delay counter
	CALL DELAY	Call of subroutine
	MVI A, 90H	Load accumulator with the bit pattern of Red light and Don’t Talk
	OUT PORT #	Turn on Red light, kepp Don’t walk on and turn off yellow light
	CALL DELAY	
	JMP START	Go back to location START to repeat the sequence
DELAY	PUSH D	Save the contents of DE and accumulator
	PUSH PSW	Push the contents on program status word
SECOND	LXI D, COUNT	Load register pair DE with a count for 1 second delay
LOOP	DCX D	Decrement register pair DE
	MOV A,D	
	ORA E	OR(D) and (E) to set Zero flag
	JNZ LOOP	Jump to loop if delay decrement the counter
	JNZ SECOND	Go back to repeat 1-second delay
	POP PSW	Retrieve contents of saved registers
	POP D	
	RET	Return to main program

Miscellaneous Questions

Q1. Explain the concept of Stack, subroutines, Return, Restart and conditional Call

Q2. Explain about nesting of subroutines

References

To refer opcodes for program in simulator use the link Here

https://electricalvoice.com/opcodes-8085-microprocessor/

❖❖❖

UNIT 4

10

CODE CONVERSION WITH BCD

Unit Structure

10.0 Objectives

10.1 Introduction

10.2 BCD –TO- Binary Conversion

10.2.1 Illustrative Program: 2- Digit BCD – to- Binary Conversion

10.3 Binary –TO-BCD Conversion

10.3.1 Illustrative Program: Binary – To-Unpacked-BCD Conversion

10.4 BCD-TO-Seven-Segment-LED Code Conversion

10.4.1IllustrativeProgram:BCD-TO-Common-Cathode-LEDCode Conversion

10.5 Binary –TO-ASCII AND ASCII–TO-Binary Code Conversion

10.5.1 Illustrative Program: Binary-To-ASCII Hex Code Conversion

10.5.2 Illustrative Program: ASCII Hex-to-Binary Conversion

10.6 Summary

Questions and Programming Assignments

10.0 Objectives

Write programs and subroutine to

- Convert a packed BCD number (0-99) into its binary equivalent.
- Convert a binary digit (0 to F) into its ASCII Hex code and vice versa.
- Select an appropriate seven-segment code for a given binary number using the table look-up technique.
- Convert a binary digit (0 to F) into its ASCII Hex code and vice versa.
- Decimal- adjust 8-bit BCD addition and subtraction.
- Perform such arithmetic operation as multiplication and subtraction using 16-bit data related instructions.
- Demonstrate uses of instructions such as DAD, PCHL, XTHL, and XCHG.

10.1 Introduction

In microprocessor applications, various number systems and codes are used to input data or to display results. The ASCII (American Standard Code for Information Interchange) keyboard is a commonly used input device for disk- based microcomputer systems. Similarly, alphanumeric characters (letters and numbers) are displayed on a CRT(cathode ray tube) terminal using the ASCII code. However , inside the microprocessor , data processing is usually performed in binary. In some instances, arithmetic operations are performed in BCD numbers. Therefore, data must be converted from one code to another code. The programming techniques used for code conversion fall into four general categories.

1. Conversion based on position of a digit in a number (BCD to binary and vice versa) .
2. Conversion based on hardware consideration (binary to seven-segment code using table look-up procedure).
3. Conversion based on sequential order of digits (binary to ASCII and vice versa).
4. Decimal adjustment in BCD arithmetic operations. (This is an adjustment rather than a code conversion).

   This chapter discusses these techniques with various examples as subroutines. The subroutines are written to demonstrate industrial practices in writing software, and can be verified on single- board microcomputers. In addition, instructions related to 16- bit data operations are introduced and illustrated.

10.2 BCD –to- Binary Conversion

In most microprocessor –based products, data are entered and displayed in decimal numbers.

For example, in an instruction laboratory , readings such as voltage and current are maintained in decimal numbers, and data are entered through a decimal keyboard. The system-monitor program of the instrument converts each key into equivalent 4- bit binary number and stores two BCD number in an 8-bit register or a memory location. These numbers are called **packed BCD.** Even if data are entered in decimal digits, it is inefficient to process data in BCD numbers because , in each 4-bit combination, digit A though F are unused. Therefore, BCD numbers are generally converted into binary numbers for data processing.

The conversion of a BCD number into its binary equivalent employs the principle of positional weighting in a given number.

For example: 72_{10} = 7 * 10+ 2

The digit 7 represents 70, based on its second position from the right. Therefore , converting 72_{BCD} into its binary equivalent requires multiplying the second digit by 10 and adding the first digit.

Converting a 2- digit BCD number into its binary equivalent requires the following steps:

1. Separate an 8- bit packed BCD number into two 4- bit unpacked BCD digits: BCD_1 and BCD_2
2. Convert each digit into its binary value according to its position.
3. Add both binary numbers to obtain the binary equivalent of the BCD number.

Example 10.1: - Convert 72_{BCD} into its binary equivalent

72_{10} = 0111 0010 $_{BCD}$

Step 1 : 0111 0010 → 0000 0010 Unpacked BCD_1

→ 0000 0111 Unpacked BCD_2

Step 2 : Multiply BCD_2 by 10 (7 * 10)

Step 3 : Add BCD_1 to the answer in step 2

The multiplication of BCD_2 by 10 can be performed by various methods. One method is multiplication with repeated addition: add 10 seven times. This technique is illustrated in the next program.

10.2.1 Illustrative Program: 2- Digit BCD – to- Binary Conversion

Problem Statement

A BCD number between 0 and 99 is stored in an R/W memory location called the Input Buffer (INBUF) . Write a main program and a conversion subroutine (BCDBIN) to convert the BCD number into its equivalent binary number. Store the result in a memory location defined as the Output Buffer (OUTBUF).

Program

START : LXI SP, STACK;	Initialize stack pointer
LXI H, INBUF	Point HL index to the Input Buffer Memory location where BCD number is stored.
LXI B,OUTBUF	Point BC index to the Output Buffer memory where binary number will be stored.
MOV A, M	Get BCD number

CALL BCDBIN	Call BCD to binary conversion routine
STAX B	Store binary number in the Output Buffer
HLT	End of program

BCDBIN : Function : This subroutine converts a BCD number into its binary equivalent

Input : A 2-digit packed BCD number in the accumulator

Output : A binary number in the accumulator

No other register contents are destroyed.

Example : Assume BCD number is 72.

PUSH B ; Save BC registers

PUSH D ; Save DE registers   A   0111 0010 → 72_{10}

MOV B,A ; Save BCD number   B   0111 0010 → 72_{10}

ANI 0FH ; Mask most significant four bits   A   0000 0010 → 02_{10}

MOV C,A ; Save unpacked BCD_1 in C   C 0000 0010 → 02_{10}

MOV A,B ; Get BCD again   A   0111 0010 → 72_{10}

ANI F0H ; Mask least significant four bits   A   0111 0000 → 70_{10}

RRC : Convert most significant four

RRC; bits into unpacked BCD_2

RRC

RRC

MOV D,A; Save BCD_2 in D   A   0000 0111 → 07_{10}

XRA A; Clear accumulator   D   0000 0111 → 07_{10}

MVI E,0AH ; Set E as multiplier of 10   E 0000 10 10 → 0AH

SUM : ADD E ; Add 10 until (D) =0 Add E as many times as (D)

DCR D; Reduce BCD_2 by one

JNZ SUM ; Is multiplication complete ?   After adding E seven times A

;If not ,go back and add again   contains : 0100 0110

ADD C ; Add BCD_1 C   +0000 0010

A   0100   1000 → 48H

POP D ; retrieve previous contents

RET

Program Description

1. In writing assembly language programs, the use of labels is a common practice. Rather than writing a specific memory location or a port number a programmer uses such labels as INBUF (Input Buffer) and OUTBUF (Output Buffer). Using labels give flexibility and ease of documentation.

2. The main program initializes the stack pointer and two memory indexes. It brings the BCD number into the accumulator and passes that parameter to the subroutine.

3. After returning from the subroutine , the main program stores the binary equivalent in the Output Buffer memory.

4. The subroutine saves the contents of the BC and DE registers because these registers are used in the subroutine. Even if this particular main program does not use the DE registers, the subroutine may be called by some other program in which the DE registers are being used. Therefore, it is good practice to save the registers that are used in the subroutine, unless parameters are passed to the subroutine. The accumulator contents are not saved because that information is passed to the subroutine.

5. The conversion from BCD to binary is illustrated in the subroutine with the example of 72_{BCD} converted to binary.

Program Execution

To execute the program on a single-board computer, complete the following steps:

1. Assign memory addresses to the instructions in the main program and in the subroutine .Both can be assigned consecutive memory addresses.

2. Define STACK: the stack location with a 16- bit address in the R/W memory (such as 2099H).

3. Define INBUF (Input Buffer) and OUTBUF (Output Buffer): two memory locations in the R/W memory (e.g. 2050H and 2060H).

4. Enter a BCD byte in the Input Buffer (e.g. 2050H).

5. Enter and execute the program.

6. Check the contents of the Output Buffer memory location(2060H) and verify the answer.

10.3 Binarys–to-BCD Conversion

In most microprocessor –based products, numbers are displayed in decimal. However, if data processing inside the microprocessor is performed in binary, it is necessary to convert the binary results into their equivalent BCD numbers just before they are displayed. Results are quite often stored in R/W memory locations called the **Output Buffer**.

The conversion of binary to BCD is performed by dividing the number by the powers of ten; the division is performed by the subtraction method.

For example, assume the binary number is

$1\,1\,1\,1 \quad 1\,1\,1\,1_2$ (FFH) $=255_{10}$

To represent this number in BCD requires twelve bits or three BCD digits, labelled here as BCD_3 (MSB) , and BCD_1(LSB)

0010	0101	0101
BCD_3	BCD_2	BCD_1

The conversion can be performed as follows :

Step 1: If the number is less than 100 , go to Step 2; otherwise , divide by 100 or subtract 100 repeatedly until the remainder is less than 100.The quotient is the most significant BCD digit , BCD_3

Step 2 : If the number is less than 10 , go to step 3; otherwise divide by 10 repeatedly until the remainder is less than 10. The quotient is BCD_2

Step 3: The remainder from Step 2 is BCD_1

Example		Quotient
255 -100 -100	 =155 =55 BCD_3	 1 1 2
55 -10 -10 -10 -10 -10	 =45 =35 =25 =15 =05	 1 1 1 1 1
BCD_2 = 5 BCD_1= 5		

10.3.1 Illustrative Program: Binary – To-Unpacked-BCD Conversion

Problem Statement

A binary number is stored in memory location BINBYT. Convert the number into BCD; store each BCD as two unpacked BCD digits in the output Buffer. To perform this task, write a main program and two subroutines: one to supply the powers of ten and the other to perform the conversion.

Program

This program converts an 8- bit binary number into a BCD number; thus it requires 12 bits to represent three BCD digits. The result is stored as three unpacked BCD digits in three Output-Buffer memory locations.

	LXI SP,STACK	;Intialize stack pointer
	LXI H, BINBYT	;Point HL index where binary number is stored
	MOV A,M	;Transfer byte
	CALL PWRTEN	;call subroutine to load power of 10
	HLT	
PWRTEN:	; this subroutine loads the power of 10 in register B and calls the binary – to- BCD conversion routine. ; Input : Binary number in accumulator ;Output : Powers of ten and stores BCD1 in the first Output-Buffer memory ; Calls BINBCD routine and modifies register B	
	LXI H,OUTBUF	;Point HL index to Output-Buffer memory
	MVI B,64H	;Load 100 in register B
	CALL BINBCD	; Call conversion
	MVI B,0AH	; Load 10 in register B
	CALL BINBCD	
	MOV M,A	Store BCD_1
	RET	
BINBCD	; This subroutine coverts a binary number into BCD and stores BCD2 and BCD_3 in the Output Buffer ; Input : Binary number in accumulator and powers of 10 in B ; Output : BCD_2 and BCD_3 in Output Buffer ;Modifies accumulator contents	

	MVI M,FFH	;Load buffer with (0-1)
NXTBUF:	INR M	; Clear buffer and increment for each subtraction
	SUB B	; Subtract power of 10 from binary number
	JNC NXTBUF	; Is number > power of 10? If yes, add 1 to buffer memory
	ADD B	; If no , add power of 10 to get back remainder
	INX H	;Go to next buffer location
	RET	

Program Description

This program illustrates the concepts of the nested subroutine and the multiple-calll subroutine. The main program calls the PWRTEN subroutine ; in turn the PWRTEN calls the BINBCD subroutine twice.

1. The main program transfers the byte to be converted to the accumulator and calls the PWRTEN subroutine.

2. The subroutine PWRTEN supplies the powers of ten by loading register B and the address of the first Output-Buffer memory location , and calls the conversion routine BINBCD.

3. In the BINBCD conversion routine , the Output-Buffer memory is used as a register. It is incremented for each subtraction loop. This step also can be achieved by using a register in the microprocessor. The BINBCD subroutine is called twice, once after loading register B with 64H (100_{10}) and again after loading register B with 0AH (10_{10}).

4. During the first call of BINBCD, the subroutine clears the Output Buffer , stores BCD_3, and points the HL registers to the next Output-Buffer location. The instruction ADD B is necessary to restore the remainder because one extra subtraction is performed to check the borrow.

5. During the second call of BINBCD , the subroutine again clears the output buffer, stores BCD_2 , and points to the next buffer location. BCD_3 is already in the accumulator after the ADD instruction, which is stored in the third Output- Buffer memory by the instruction MOV M,A in the PWRTEN subroutine.

   This is an efficient subroutine; it combines the functions of storing the answer and finding a quotient. However, two subroutines are required, and the second subroutine is called twice for a conversion.

10.4 BCD-to-Seven-Segment-Led Code Conversion

When a BCD number is to be displayed by a seven-segment LED, it is necessary to convert the BCD number to its seven-segment code. The code is determined by hardware considerations such as common-cathode or common-anode LED; the cod has no direct relationship to binary numbers. Therefore , to display a BCD digit at a seven-segment LED , the table look-up technique is used.

In the look-up technique the codes of the digits to be displayed are stored sequentially in memory. The conversion program locates the code of a digit based on its magnitude and transfers the code to the MPU to send out to a display port. The table look-up technique is illustrated in the next program.

10.4.1 IllustrativeProgram:BCD-TO-COMMON-CATHODE-LED CODE CONVERSION

Problem Statement

A set of three packed BCD numbers (six digits) representing time and temperature are stored in memory locations starting at XX50H. The seven-segment codes of the digit 0 to 9 for a common-cathode LED are stored in memory locations starting at XX70H, and the Output-Buffer memory is reserved at XX90H.

Write a main program and two subroutines, called UNPAK and LEDCOD , to unpack the BCD numbers and select an appropriate seven-segment code for each digit. The codes should be stored in the Output-Buffer memory.

Program

	LXI SP,STACK	; Initialize stack pointer
	LXI H, XX50H	; Point HL where BCD digits are stored.
	MVI D,03H	; Number of digits to be converted is placed in D
	CALL UNPAK	Call subroutine to unpack BCD numbers
	HLT	; End of conversion
UNPAK:	; This subroutine unpacks the BCD number in two single digits ;Input : Starting memory address of the packed BCD numbers in HL registers ;Number of BCDs to be converted in register D Output : Unpacked BCD into accumulator and output ;Buffer address in BC ;Calls subroutine LEDCOD	
	LXI B,BUFFER	;Point BC index to the buffer memory

NXTBCD:	MOV A,M	Get packed BCD number
	ANI F0H	Masked BCD_1
	RRC RRC	Rotate four times to place BCD_2 as unpacked single-digit BCD
	RRC RRC	
	CALL LEDCOD	; Find seven-segment code
	INX B	;Point to next buffer location
	MOV A,M	Get BCD number again
	ANI 0FH	;Separate BCD_1
	CALL LEDCOD	
	INX B	
	INX H	; Point to next BCD
	DCR D	; One conversion complete , reduce BCD count
	JNZ NXTBCD	;If all BCDs are not yet converted , go back to convert next BCD
	RET	
LEDCOD:	; This subroutine converts an unpacked BCD into its seven-segment-LED code ;Input :An unpacked BCD in accumulator ;Memory address of the buffer in BC register ;Output : Stores seven-segment code in the output buffer	
	PUSH H	;Save HL contents of the caller
	LXI H,CODE	; Point index to beginning of seven-segment code
	ADD L	; Add BCD digit to starting address of the code
	MOV L,A	; Point HL to appropriate code
	MOV A,M	; Get seven- segment code
	STAX B	;Store code in buffer
	POP H	
	RET	

CODE:	3F	; Digit 0: Common- cathode codes
	06	; Digit 1
	5B	; Digit 2
	4F	; Digit 3
	66	; Digit 4
	6D	; Digit 5
	7D	; Digit 6
	07	; Digit 7
	7F	; Digit 8
	6F	; Digit 9
	00	;Invalid Digit

Program Description / Output

1. The main program initializes the stack pointer, the HL register as a pointer for BCD digits , and the counter for the number off digits; then it calls the UNPAK subroutine.

2. The UNPAK subroutine transfers a BCD number into the accumulator and unpacks it into two BCD digits by using the instruction ANI and RR. This subroutine also supplies the address of the buffer memory to the next subroutine, LEDCOD. The subroutine is repeated until counter D becomes zero.

3. The LEDCOD subroutine saves the memory address of the BCD number and points the HL register to the beginning address of the code.

4. The instruction ADD L adds the BCD digit in the accumulator to the starting address of the code. After storing the sum in register L, the HL register points to the seven-segment code of that BCD digit.

5. The code is transferred to the accumulator and stored in the buffer.

   This illustrative program uses the technique of the nested subroutine (one-subroutine calling another).

   Parameters are passed from one subroutine to another; therefore, we should be careful in using Push instructions to store register contents on the stack. In addition , the LEDCOD does not account for a situation if by adding the register L a carry is generated.

10.5 Binary –To-ASCII and ASCII–to-Binary Code Conversion

The American Standard Code for Information Interchange (known as ASCII) is used commonly in data communication. It is a seven-bit code, and its 128(2^7) combinations are assigned different alphanumeric characters. For example, the hexadecimal capital letters 30H to 39H represent 0 to 9 ASCII decimal numbers. and 41H to 5AH represent capital letters A though Z ; in this code , bit D_7 is zero. In serial data communication, bit D_7 is used for parity checking.

The ASCII keyboard is a standard input device for entering programs in a microcomputer. When an ASCII character is entered, the microprocessor receives the binary equivalent of the ASCII Hex number. For example, when the ASCII key for digit 9 is pressed, the microprocessor receives the binary equivalent of 39H, which must be converted to the binary 1001 for arithmetic operations. Similarly, to display digit 9 at the terminal, the microprocessor must send out the ASCII Hex code (39H) .

These conversions are done through software ,as in the following illustrative program.

10.5.1 Illustrative Program: Binary-To-ASCII Hex Cod Conversion

Problem Statement

An 8-bit binary number (e.g. 9FH) is stored in memory location XX50H .

1. Write a program to
   a. Transfer the byte to the accumulator.
   b. Separate the two nibbles.
   c. Call the subroutine to convert each nibble into ASCII Hex code.
   d. Store the codes in memory locations XX60H and XX61H.
2. Write a subroutine to convert a binary digit (0 to F) into ASCII Hex Code.

Main Program

	LXI SP,STACK	; Initialize the stack pointer
	LXI H, XX50H	; Point index where binary number is stored.
	LXI D,XX60H	; Point index where ASCII code is stored.
	MOV A,M	;Transfer byte
	MOV B,A	;Save byte

	RRC RRC RRC RRC	;Shift high-order nibble to the position of low-order nibble
	CALL ASCII	; Call conversion routine
	STAX D	;Store first ASCII Hex in XX60H
	INX D	;Point to next memory location ,get ready to store next byte
	MOV A,B	;Get number again for second digit.
	CALL ASCII	
	STAX D	
	HLT	
ASCII :	;This subroutine converts a binary digit between 0 to F to ASCII Hex Code ;Input : Single binary number 0 to F in the accumulator. ; Output : ASCII Hex code in the accumulator	
	ANI 0FH	;Mask high-order nibble
	CPI 0AH	;Is digit less than 10_{10} ?
	JC CODE	;If digit is less than 10_{10} , go to CODE to add 30H
	ADI 07H	;Add 7H to obtain code for digits from A to F
CODE:	ADI 30H	;Add base number 30H
	RET	

Program Description

1. The main program transfers the binary data byte from the memory locations to the accumulator.

2. It shifts the high-order nibble into the low-order nibble, calls the conversion subroutine, and stores the converted value in the memory.

3. It retrieves the byte again and repeats the conversion process for the low-order nibble.

In this program, the masking instruction ANI is used once in the subroutine rather than twice in the main program as illustrated in the program for BCD –To –Common-Cathode Code Conversion.

10.5.2 Illustrative Program :ASCII Hex-to-Binary Conversion

Problem Statement

Write a subroutine to convert an ASCII Hex number into its binary equivalent. A calling program places the ASCII number in the accumulator , and the subroutine should pass the conversion to the accumulator.

Subroutine

ASCBIN :	This subroutine converts an ASCII Hex number into its binary equivalent ;Input: ASCII Hex number in the accumulator ;Output : Binary equivalent in the accumulator	
	SUI 30H	; Subtract 0 bias from the number
	CPI 0AH	; Check whether number is between 0 to 9
	RC	; If yes , return to main program
	SUI 07H	; If not , subtract 07H to find number between A and F
	RET	

Program Description:

This program subtracts the ASCII weighting digits from the number. This process is exactly opposite to that of the Illustrative Program that converted binary into ASCII Hex .However, this program uses two return instructions , an illustration of the multiple-ending subroutine.

10.6 Summary

The system-monitor program of the instrument converts each key into equivalent 4- bit binary number and stores two BCD number in an 8-bit register or a memory location. These numbers are called **packed BCD.**

Even if data are entered in decimal digits, it is inefficient to process data in BCD numbers because, in each 4- bit combination, digit A though F are unused. Therefore, BCD numbers are generally converted into binary numbers for data processing.

The conversion of a BCD number into its binary equivalent employs the principle of positional weighting in a given number.

In most microprocessor –based products, numbers are displayed in decimal. However, if data processing inside the microprocessor is performed in binary, it is necessary to convert the binary results into their equivalent BCD numbers just before they are displayed. Results are quite often stored in R/W memory locations called the **Output Buffer**.

The conversion of binary to BCD is performed by dividing the number by the powers of ten; the division is performed by the subtraction method.

When a BCD number is to be displayed by a seven-segment LED, it is necessary to convert the BCD number to its seven-segment code. The code is determined by hardware considerations such as common-cathode or common-anode LED; the cod has no direct relationship to binary numbers. Therefore, to display a BCD digit at a seven-segment LED, the table look-up technique is used.

In the look-up technique the codes of the digits to be displayed are stored sequentially in memory. The conversion program locates the code of a digit based on its magnitude and transfers the code to the MPU to send out to a display port.

The American Standard Code for Information Interchange (known as ASCII) is used commonly in data communication. It is a seven-bit code, and its 128(2^7) combinations are assigned different alphanumeric characters. For example, the hexadecimal capital letters 30H to 39H represent 0 to 9 ASCII decimal numbers. and 41H to 5AH represent capital letters A though Z ; in this code , bit D_7 is zero. In serial data communication, bit D_7 is used for parity checking.

The ASCII keyboard is a standard input device for entering programs in a microcomputer. When an ASCII character is entered, the microprocessor receives the binary equivalent of the ASCII Hex number. For example, when the ASCII key for digit 9 is pressed, the microprocessor receives the binary equivalent of 39H, which must be converted to the binary 1001 for arithmetic operations. Similarly, to display digit 9 at the terminal, the microprocessor must send out the ASCII Hex code (39H) .

QUESTIONS AND PROGRAMMING ASSIGNMENTS

Q1) Explain BCD –to –binary conversion with examples.

Q2) Write a program for 2-digit BCD to binary conversion

Q3) Explain Binary –to –BCD conversion with examples.

Q4) Write a program for binary to unpacked BCD conversion.

Q5) Rewrite the BCDBIN subroutine to include storing results in the Output Buffer . Eliminate unnecessary PUSH and POP instructions.

Q6) Write a program for BCD to common cathode LED code conversion.

Q7) Write a program for binary to ASCII Hex code conversion

Q8) Write a program for ASCII Hex to binary conversion.

Books and References

1. Computer System Architecture by M. Morris Mano , PHI Publication, 1998.

2. Structured Computer Organization by Andrew C. Tanenbaum , PHI Publication.

3. Microprocessors Architecture, Programming and Application with 8085 by Ramesh Gaonker ,PENRAM , Fifth Edition ,2012.

❖❖❖

UNIT 4

11

BCD ARITHMETIC AND 16-BIT DATA OPERATION

11.0 Objectives

11.1 Introduction

11.2 BCD Addition

11.2 .1 Illustrative Program: Addition of Unsigned BCD Numbers

11.3 BCD Subtraction

11.3.1 Illustrative Problem: Subtraction of Two Packed BCD Numbers

11.4 Introduction To Advanced Instructions And Application

11.4.1:- 16- Bit Data Transfer (Copy) and Data Exchange Group

11.4.2 Arithmetic Group

11.4.3 Instruction Related to the Stack Pointer and the Program Counter

11.4.4 Miscellaneous Instruction

11.5 Multiplication

11.5.1 Illustrative Program: Multiplication of Two 8-Bit Unsigned Numbers

11.6 Subtraction With Carry

11.6.1 Illustrative Program: 16-Bit Subtraction

11.7 Summary

Questions and Programming Assignments

11.0 Objectives

Write programs and subroutine to

- Convert a packed BCD number (0-99) into its binary equivalent.
- Convert a binary digit (0 to F) into its ASCII Hex code and vice versa.
- Select an appropriate seven-segment code for a given binary number using the table look-up technique.
- Convert a binary digit (0 to F) into its ASCII Hex code and vice versa.
- Decimal- adjust 8-bit BCD addition and subtraction.
- Perform such arithmetic operation as multiplication and subtraction using 16-bit data related instructions.
- Demonstrate uses of instructions such as DAD, PCHL,XTHL, and XCHG.

11.1 Introduction

In microprocessor applications, various number systems and codes are used to input data or to display results. The ASCII (American Standard Code for Information Interchange) keyboard is a commonly used input device for disk-based microcomputer systems. Similarly, alphanumeric characters (letters and numbers) are displayed on a CRT(cathode ray tube) terminal using the ASCII code. However, inside the microprocessor, data processing is usually performed in binary. In some instances, arithmetic operations are performed in BCD numbers. Therefore, data must be converted from one code to another code. The programming techniques used for code conversion fall into four general categories.

1. Conversion based on position of adigit in a number (BCD to binary and vice versa) .
2. Conversion based on hardware consideration (binary to seven-segment code using table look-up procedure).
3. Conversion based on sequential order of digits (binary to ASCII and vice versa).
4. Decimal adjustment in BCD arithmetic operations. (This is an adjustment rather than a code conversion).

This chapter discusses these techniques with various examples as subroutines. The subroutines are written to demonstrate industrial practices in writing software, and can be verified on single- board microcomputers. In addition, instructions related to 16- bit data operations are introduced and illustrated.

11.2 BCD Addition

In some applications, input /output data are presented in decimal numbers, and the speed of data processing is unimportant. In such applications, it may be convenient to perform arithmetic operations directly in BCD numbers. However, the addition of two BCD numbers may not represent an appropriate BCD value. For example, the addition of 34_{BCD} and 26_{BCD} results in 5AH as shown below:

34_{10} =	$0011\ 0100_{BCD}$
$+26_{10}$ =	$0010\ \ 0110_{BCD}$
60_{10} =	$0101\ \ 1010_{BCD}$

The microprocessor cannot recognize BCD numbers; it adds any two numbers in binary.

In BCD addition, any number larger than 9 (from A to F) is invalid and needs to be adjusted by adding 6 in binary.

For example, after 9, the next BCD number is 10; However, in Hex it is A. The Hex number A can be adjusted as a BCD number by adding 6 in binary .The BCD adjustment in 8-bit binary register can be shown as follows:

A=	0 0 0 0 1 0 1 0
+6=	0 0 0 0 0 1 1 0
	0 0 0 1 0 0 0 0 $\longrightarrow 10_{BCD}$

Any BCD sum can be adjusted to proper BCD value by adding 6 when the sum exceeds 9 . In case of packed BCD, both BCD_1 and BCD_2 need to be adjusted; if a carry is generated b adding 6 to BCD_1, the carry should be added to BCD_2, as shown in the following example.

Example 11.1:- Add two packed BCD number: 77 and 48**.**

77 =	0 1 1 1	0 1 1 1
+48=	0 1 0 0	1 0 0 0
125=	1 0 1 1	1 1 1 1
		+ 0 1 1 0
	CY 1	0 1 0 1
	+0 1 1 0	
CY 1	0 0 1 0	0 1 0 1

The value of the least significant four bits is larger than 9. Add 6.

The value of the most significant four bits is larger than 9, Add 6 and the carry from the previous adjustment.

In this example, the carry is generated after the adjustment of the least significant four bits for the BCD digit and is again added to the adjustment of the most significant four bits.

A special instruction called DAA (decimal Adjust Accumulator) performs the function of adjusting a BCD sum in 8085 instruction set. This instruction uses the Auxiliary Carry flip-flop (AC) to sense that the value of the least four bits is larger than 9 and adjusts the bits to the BCD value. Similarly, it uses the Carry flag (CY) to adjust the most significant four bits. However, the AC flag is used internally by the microprocessor; this flag is not available to the programmer through any Jump instruction.

Instruction

DAA : Decimal Adjust Accumulator

This is a 1- byte instruction

It adjusts an 8- bit number in the accumulator to form two BCD numbers by using the process described above.

It uses the AC and the CY flags to perform the adjustment.

All flags are affected.

It must be emphasized that instruction DAA

Adjust a BCD sum.

It does not convert a binary number into BCD numbers.

It works only with addition when BCD numbers are used ; does not work with subtraction.

11.2 .1 Illustrative Program: Addition of Unsigned BCD Numbers

Problem Statement

A set of ten packed BCD numbers is stored in the memory location starting at XX50H.

1. Write a program with a subroutine to add these numbers in BCD. If a carry is generated, save it in register B, and adjust it for BCD. The final sum will be less than 9999_{BCD} .
2. Write a second subroutine to unpack the BCD sum stored in registers A and B, and store them in the output-buffer memory starting at XX60H. The most significant digit (BCD_4) should be stored at XX60H and the least significant digit (BCD_1) at XX63H.

START :	LXI SP, STACK	;Initialize stack pointer
	LXI H, XX50H	; Point index to XX50H
	MVI C,COUNT	;Load register C with the count of BCD numbers to be added
	XRA A	; Clear accumulator
	MOV B,A	; Clear register B to save carry
NXTBCD	CALL BCDADD	; Call subroutine to add BCD numbers
	INX H	; Point to next memory location
	DCR C	; One addition of BCD number is complete, decrement the counter
	JNZ / NXTBCD	;If all numbers are added go to next step, otherwise go back
	LXI H,XX63H	;Point index to store BCD_1 first

	CALL UNPAK	; Unpack the BCD stored in the accumulator
	MOV A,B	;Get ready to store high-order BCD- BCD_3 and BCD_4
	CALL UNPAK	; Unpack and store BCD_3 and BCD_4 at XX61H and XX60H
	HLT	
BCDADD:	; This subroutine adds the BCD number from the memory to the accumulator and decimal adjusts it. If sum is larger than eight bits, it saves the carry and decimal-adjusts the carry sum. ; Input : The memory address in HL register where the BCD number is stored ;Output : Decimal-adjusted BCD number in the accumulator and the carry in register B	
	ADD M	; Add packed BCD byte and adjust it for BCD sum.
	DAA	
	RNC	;If no carry, go back to next BCD
	MOV D,A	; If carry is generated, save the sum from the accumulator
	MOV A,B	; Transfer CY sum from register B and add 01
	ADI 01H	
	DAA	; Decimal-adjust BCD from B
	MOV B,A	; Save adjusted BCD in B
	MOV A,D	;Place BCD_1 and BCD_2 in the accumulator
	RET	
UNPAK:	; This subroutine unpacks the BCD in the accumulator and the carry register and stores them in the output buffer.	

	; Input : BCD number in the accumulator, and the buffer address in HL registers ;Output : Unpacked BCD in the output buffer	
	MOV D,A	; Save BCD number
	ANI 0FH	;Mask high-order BCD
	MOV M,A	; Store low-order BCD
	DCX H	; Point to next memory location
	MOV A,D	; Get BCD again
	ANI F0H	; Mask low-order BCD
	RRC RRC RRC RRC	; Convert the most significant four bits into unpacked BCD
	MOV M,A	; Store high-order BCD
	DCX H	; Point to the next memory location
	RET	

Program Description

1. The expected maximum sum is 9090, which requires two registers. The main program clears the accumulator to save BCD$_1$ and BCD$_2$, clears register B to save BCD$_3$ and BCD$_4$, and call the subroutine to add the numbers. The BCD bytes are added until the counter becomes zero.

2. The BCDADD subroutine is an illustration of the multiple ending subroutines. It adds a byte, decimal –adjusts the accumulator and, if there is no carry, returns the program execution to the main program. If there is a carry, it adds 01 to the carry register B by transferring the contents to the accumulator and decimal-adjusting the contents. The final sum is stored in register A and B.

3. The main program calls the UNPAK subroutine,which takes the BCD number from the accumulator (e.g. 57$_{BCD}$), unpacks it into two separate BCD (e.g. 05$_{BCD}$ and 07$_{BCD}$), stores them in the output buffer. When a subroutine stores a BCD number in memory, it decrements the index because BCD$_1$ is stored first.

11.3 BCD Subtraction

When subtracting two BCD numbers, the instruction DAA cannot be used to decimal adjust the result of two packed BCD numbers; the instruction applies only to addition. Therefore, it is necessary to devise a procedure to subtract two BCD numbers. Two BCD numbers can be subtracted by using the procedure of 100's complement (also known as 10's complement), similar to 2's complement. The 100's complement of a subtrahend can be added to a minuend as illustrated:

For example, 82-48(=34) can be performed as follows:

```
100's complement of subtrahend 52  (100-48=52)
Add minuend                      +82
                                ________
                                1 /  34
```

The sum is 34 if the carry is ignored. This is similar to subtraction by 2's complement .However in an 8-bit microprocessor; it is not a simple process to find 100's complement of a subtrahend (100_{BCD} requires twelve bits). Therefore, in writing a program 100's complement is obtained by finding 99's complement and adding 01.

11.3.1 Illustrative Problem: Subtraction of Two Packed BCD Numbers

Problem Statement

Write a subroutine one packed BCD number from another BCD number. The minuend is placed in register B, and subtrahend is placed is register C by the calling program. Return the answer into the accumulator.

Subroutine

SUBBCD	; This subroutine subtracts two BCD numbers and adjusts the result to BCD values by using the 100's complement method. ; Input : A minuend in register B and a subtrahend in register C ; Output : The result is placed in the accumulator	
	MVI A,99H	
	SUB C	; Find 99's complement of subtrahend
	INR A	;Find 100's complement of subtrahend
	ADD B	; Add minuend to 100's complement of subtrahend
	DAA	; Adjust for BCD
	RET	

11.4 Introduction to Advanced Instructions and Application

The instructions deal primarily with 8-bit data (except LXI). However, in some instances data larger than eight bits must be manipulated, especially in arithmetic manipulations and stack operations. Even if the 8085 is an 8- bit microprocessor, its architecture allows specific combinations of two 8-bit registers to form 16-bit registers. Several instructions in the instruction set are available to manipulate 16 bit data.

11.4.1:- 16- Bit Data Transfer (Copy) and Data Exchange Group

LHLD:-	Load HL registers direct This is a 3-byte instruction. The second and third bytes specify a memory location (the second byte is a line number and the third byte is a page number) . Transfers the contents of the specified memory location to L register. Transfers the contents of the next memory location to H register
SHLD:-	Store HL registers direct This is a 3-byte instruction. The second and third bytes specify a memory location (the second byte is a line number and the third byte is a page number) . Stores the contents of L register in the specified memory location Stores the contents of H register in the next memory location.
XCHG:-	Exchange the contents of HL and DE This is a 1-byte instruction The contents of H register are exchanged with the contents of D register, and the contents of L register are exchanged with the contents of E register

Example 11.2:- Memory locations 2050H and 2051H contain 3FH and 42H, respectively, and register pair DE contains 856FH. Write instructions to exchange the contents of DE with the contents of the memory locations.

Before Instructions: D $\boxed{85\ 6F}$ E Memory
$\boxed{3F}$ 2050
$\boxed{42}$ 2051

Instructions:

Machine Code	**Mnemonics**			
2A 50 20	LHLD 2050	H 42 3F L	3F 42	2050 2051
EB	XCHG	D 42 3F E H 85 6F L	3F 42	2050 2051
22 50 20	SHLD 2050H	H 85 6F L	6F 85	2050 2051

11.4.2 Arithmetic Group

Operation:	: Addition with Carry
ADC R ADC M ACI 8-bit	These instructions add the contents of the operand, the carry, and the accumulator. All flags are affected.

Example 11.3 Registers BC contain 2793H and register DE contain 3182H . Write instructions to add these two16- bit numbers, and place the sum in memory locations 2050H and 2051H.

Before instructions:	B 27 93 C
	D 31 82 E

Instructions

MOV A,C	A 93 F	93
ADD E	A 15 CY=1 F	+82
MOV L,A	H 15 L	1 /15H
MOVA,B		27
ADC D		+31
MOV H,A	H 59 15 L	59H
SHLD 2050H		

Operation: Subtraction with Carry

SBB R
SBB M
SBI 8-bit

These instructions subtract the contents of the operand and borrow from the contents of the accumulator.

Example 11.4 :- Register BC contains 8538H and registers DE contain 62A5H . Write instructions to subtract the contents of DE from the contents of BC, and place the result in BC.

Instructions

MOV A,C	(B) 85	38 (C)
SUB E		------
MOV C,A	(D) 62	A5 (E)
MOV A,B	-1	1 / 93
SBB D	(B) 22	93 (C)
MOV B,A		

Operation : Double Register ADD

DAD Rp
DAD B
DAD D
DAD H
DAD SP

Add register pair to HL

It is a 1-byt instruction

Adds the contents of the operand (register pair or stack pointer) to the contents of HL registers

The result is placed in HL registers

The Carry flag is altered to reflect the result of the 16-bit addition. No other flags are affected.

The instruction set includes four instructions.

Example 11.5 Write instructions of the stack pointer register at output ports

Instructions

LXI H, 0000H	; Clear HL
DAD SP	; Place the stack pointer contents in HL
MOV A,H	; Place high-order address of the stack pointer in the accumulator
OUT PORT1	
MOV A,L	; Place low-order address of the stack pointer in the accumulator
OUT PORT2	

The instruction DAD SP adds the contents of the stack pointer register to the HL register pair, which is already cleared. This is only instruction in the 8085 that enables the programmer to examine the contents of the stack pointer register.

11.4.3 Instruction Related to the Stack pointer and the Program Counter

XTHL : Exchange Top of the Stack with H and L

The contents of L are exchanged with the contents of the memory location shown by the stack pointer, and the contents of H are exchanged with the contents of memory location of the stack pointer +1.

Example 11.6 :- Write a subroutine to set the Zero flag and check whether the instruction JZ (Jump on Zero) functions properly, without modifying any register contents other than flags.

Subroutine

CHECK:	PUSH H	
	MVI L,FFH	; Set all bits in L to logic 1
	PUSH PSW	;Save flags on the top of the stack
	XTHL	; Set all bits in the top stack location
	POP PSW	; Set Zero flag
	JZ NOERROR	
	JMP EEROR	
NOERROR:	POP H	
	RET	

The instruction PUSHPSW places the flags in the top location of the stack,and the instruction XTHL changes all the bits in that location to logic 1 . The instruction POPPSW sets all the flags.

If the instruction JZ is functioning properly, the routine returns to the calling program; otherwise, it goes to the ERROR routine (not shown).

This example shows that the flags can be examined, and can be set or reset to check malfunctions in the instructions.

SPHL :- Copy H and L registers into the Stack Pointer Register

The contents of H specify the high-order byte and contents of L specify the low-order byte.

The contents of HL registers are not affected.

This instruction can be used to load a new address in the stack pointer register.

PCHL :- Copy H and L registers into the Program Counter

The contents of H specify the high-order byte and the contents of L specify the low-order byte.

11.4.4 Miscellaneous Instruction

CMC : Complement the Carry flag (CY)

If the Carry flag is 1, it is reset ; if it is 0, it is set.

STC: Set the Carry Flag (CY=1)

11.5 Multiplication

Multiplication can be performed by repeated addition; this technique is used in BCD –to –binary conversion. It is however, an inefficient technique for a large multiplier.

A more efficient technique can be devised by following the model of manual multiplication of decimal numbers.

For example,

	108
	X 15 ___________
Step 1 : (108 X 5) =	540
Step 2 : shift left and add (108 X 1) =	+108
	___________ 1620

In this example, the multiplier multiplies each digit of the multiplicand, starting from the farthest right, and adds the product by shifting to the left. The same process can be applied in binary multiplication.

11.5.1 Illustrative Program: Multiplication of Two 8-Bit Unsigned Numbers

Problem Statement

A multiplicand is stored in memory location XX50H and a multiplier is stored in location XX51H. Write a main program to

1. Transfer the two numbers placed in registers H and L.

2. Store the product in the Output Buffer at XX90H.

Write a subroutine to

1. Multiply two unsigned numbers placed in registers H and L.

2. Return the result into the HL pair.

Main Program

LXI SP, STACK	
LHLD XX50H	; Place contents of XX50 in L register and contents of XX51 in H register
XCHG	; Place multiplier in D and multiplicand in E
CALL MLTPLY	; Multiply the two numbers
SHLD XX90H	; Store the product in locations XX90 and 91H
HLT	

Subroutine

MLTPLY : This subroutine multiplies two 8-bit unsigned numbers ; Input : Multiplicand in register E and multiplier in register D ; Output : Results in HL register		
MLTPLY:	MOV A,D	; Transfer multiplier to accumulator
	MVI D,00H	; Clear D to use in DAD instruction
	LXI H,0000H	; Clear HL
	MVI B,08H	;Set up register B to count eight rotations
NXTBIT:	RAR	;Check if multiplier bit is 1
	JNC NOADD	; If not, skip adding multiplicand
	DAD D	; If multiplier is 1, add multiplicand to HL and place partial result in HL.
NOADD:	XCHG	; Place multiplicand in HL
	DAD H	; And shift left
	XCHG	; Retrieve shifted multiplicand
	DCR B	; One operation is complete, decrement counter
	JNZ NXTBIT	; Go back to next bit
	RET	

Program Description

1. The objective of the main program is to demonstrate use of the instruction LHLD, SHLD, and XCHG. The main program transfers the two bytes (multiplier and multiplicand) from memory locations to the HL registers by using the instruction LHLD, places them in DE register by the instruction XCHG, and places the result in the Output Buffer by the instruction SHLD.

2. The multiplier routine follows the format –add and shift to the left. The routine places the multiplier in the accumulator and rotates it eight times until the counter (B) becomes zero. The reason for clearing D is to use the instruction DAD to add register pairs.

3. After each rotation, when a multiplier bit is 1, the instruction DAD D performs the addition,and DAD H, shifts bits to the left. When a bit is 0, the subroutine skips the instruction DAD D and just shifts the bits.

11.6 Subraction with Carry

The instruction set includes several instructions specifying arithmetic operations with carry. Description of these instructions convey an impression that these instructions can be used to add (or subtract) 8- bit numbers hen the addition generates carries.

In fact, in these instructions when a carry is generated, it is added to bit D_0 of the accumulator in the next operation . Therefore, these instructions are used primarily in 16- bit addition and subtraction, as shown in the next program.

11.6.1 Illustrative Program: 16-Bit Subtraction

Problem Statement

A set of five 16-bit readings of the current consumption of industrial control units is monitored by meters and stored at memory locations starting at XX50H . The low- order byte is stored first (e.g., at XX50H), followed by the high-order byte(e.g. at XXX51H) . The corresponding maximum limits for each control unit are stored starting at XX90H.

Subtract each reading from its specified limit, and store the difference in place of the readings. If any reading exceeds the maximum limit, call the indicator routine and continue checking.

Main Program

	LXI D, 2050H	; Point index to reading
	LXI H,2080H	; Point index to maximum limits
	MVI B,05H	; Set up B as a counter
NEXT:	CALL SBTRAC	;Point to next location
	INX D	;Point to next location
	INX H	
	DCR B	
	JNZ NEXT	
	HLT	

Subroutine

SBTRAC : This subroutine subtracts two 6- bit numbers
; Input : The contents of registers DE point to reading locations
; The contents of registers HL point to maximum limits

; Output : The results are placed in reading locations, thus destroying the initial readings;		
Memory Contents		
The first current reading =6790H	2050H=90H 2051H= 67H	LSB MSB
Maximum limit =7000H	2090H=00H 2091H= 70H	LSB MSB
		; Illustrative Example
SBTRAC:	MOV A,M	;(A) =00H LSB of maximum limit
	XCHG	; (HL) =2050H
	SUB M	; (A) = 0000 0000 2's complement of 90H ;(M) = 0111 0000 Borrow flag is set to 1 0111 0000 indicate the result is in 2's complement.
	MOV M,A	;Store at 2050H
	XCHG	; (HL) =2090H
	INX H	; (HL) =2091H
	INX D	; (DE) =2051H
	MOV A,M	; (A) =70H MSB of the maximum limit
	XCHG	; (HL) =2051H
	SBB M	; (A) = 0111 0000 (70H) ; (M) = 10011001 2's complement of 67H ; (CY)= 1 Borrow flag
	CC INDIKET	;Call Indicate subroutine if reading is higher than the maximum limit
	MOV M,A	
	RET	

Program Description

This is a 16- bit subtraction routine that subtracts one byte at a time. The low-order bytes are subtracted by using the instruction SUB M . If a borrow is generated, it is accounted for by using the instruction SBB M (subtract with Carry) for the high-order bytes. In the illustrative example, the first subtraction (00H -90H) generates a borrow that is subtracted from the high-order bytes. The instruction XCHG changes the index pointer alternately between the set of readings and the maximum limits.

11.7 Summary

16- bit Data transfer (copy) and data exchange instructions

LHLD, SHLD, XCHG,XTHL, PCHL,SPHL

Arithmetic Instructions used in 16- bit operations

The following instructions add the contents of the operand, the carry, and the accumulator.

ADC R, ADC M, ACI 8- bit

Subtraction: The following instructions subtract the contents of the operand and the borrow from the contents of the accumulator.

SBB R, SBB M, SBI 8-bit

Questions and Programming Assignments

Q1) Explain BCD Addition with examples.

Q2) Write a counter program to count continuously from a 0 to 99 in BCD with a delay of 750ms between each count. Display the count at an output port.

Q3) Explain BCD Subtraction with examples.

Q4) Write a program to subtract a 2-digit BCD number from another 2-digit BCD number .

Q5) Explain Multiplication with examples.

Q6) A set of 16- bit readings is stored n memory locations starting at 2050H .Each reading occupies two memory locations: the low- order byte is stored first, followed by the high- order byte. The number of readings stored is specified by the contents of B register. Write a program to add all the reading and store the sum in the Output-Buffer memory. (The maximum limit of a sum is 24 bits).

Books and References

1. Computer System Architecture by M. Morris Mano, PHI Publication, 1998.

2. Structured Computer Organization by Andrew C. Tanenbaum, PHI Publication.

3. Microprocessors Architecture, Programming and Application with 8085 by Ramesh Gaonker,PENRAM, Fifth Edition,2012.

Books and References

1. Computer System Architecture by M. Morris Mano, PHI Publication, 1998.

2. Structured Computer Organization by Andrew C. Tanenbaum, PHI Publication.

3. Microprocessors Architecture, Programming and Application with 8085 by Ramesh Gaonker, PENRAM, Fifth Edition, 2012.

❖❖❖

UNIT 4

12

SOFTWARE DEVELOPMENT SYSTEM AND ASSEMBLERS

Unit Structure

12.1 Objectives

12.2 Introduction

12.3 Microprocessors-Based Software Development system

12.4 Operating System and Programming Tools

12.5 Assemblers and Cross-Assemblers, Writing Program Using Cross Assemblers.

12.1 Objectives

At the end of this chapter, the student will be able to

- Illustrate the concept of Microprocessor-Based Software Development System
- Describe the concept of operating system and programming tools
- Explain the assemblers and Cross-Assemblers
- Write the program using cross Assemblers

12.2 Introduction

1. The assembly language level differs in a significant respect from the micro-architecture, ISA and operating system machine levels- it is implemented by translation rather than by interpretation.
2. Programs that convert a user's program written in some language to another language are called translators.
3. The language in which the original program is written is called the source language and the language to which it is converted is called the target language.

4. Translation is used when a processor (either hardware or an interpreter) is available for the target language but not for the source language. If the translation has been performed correctly, running the translated program will give precisely the same results as the execution of the source program would have given had a processor for it been available.

5. In translation, the original program in the source language is not directly executed. Instead it is converted to an equivalent program called an object program or executable binary program whose execution is carried out only after the translation has been completed.

6. While the object program is being executed, only three levels are in evidence, the micro-architecture level, the ISA level and the operating system machine level and these programs are found on computer's memory during their execution, this is possible due to advancement in microprocessor in software level with respect to operating system and assemblers.

7. So in this chapter we will going to study more about operating system and its programming tools along with assemblers , cross assemblers.

12.3 Microprocessors-Based Software Development System

1. It is simply a computer that enables the user to write, modify, debug and test programs.

2. In a microprocessor-based development system, a microprocessor is used to develop software for a particular microprocessor.

3. Generally, the microprocessor has a large R/W memory(typically 8M to 64 M), disk storage, and a video terminal with an ASCII keyboard.

4. The System(I/Os, files, programs etc) is managed by a program called the operating system.

5. Software development system includes an ASCII keyboard, a CRT terminal, an MPU board with 8M to 64M R/W memory and disc controllers and disk drivers.

6. The disk controller is an interfacing circuit through which the microprocessor unit can access a disk and provide Read/Write control signals.

7. The disk drives have Read/Write elements, which are responsible for reading and writing data on the disk.

8. At present, most systems are equipped a 3.5-inch disk stores 1.44M bytes of information.

9. The Storage capacity of a typical hard disk in a PC is 2.2G(giga) bytes or higher.

10. Floppy disk- It is made up of a thin magnetic material(iron oxide) that can store logic 0s and 1s in the form of magnetic direction. The surface on the disk is divided in to a number of concentric tracks, and each track is divided in to sectors.Data are stored on concentric circular tracks on both sides(known as doubled-sided). At the edge of the disk there is a 'notch' called write protected notch.

Fig. 1 - Image of Floppy Disk

11. Hard Disk- Another type of storage memory used with computers called a hard disk. In general the disk is fastened in a dust free drive mechanism. It is highly precise and reliable. It requires sophisticated controller circuitry. It is more stable than the floppy disk. They are available in various sizes and their storage capacity is quite large in the order of gigabytes.

Fig. 2 - Image of Hard Disk

12. CD-ROM- A CD-ROM is a optical disk that uses a laser beam to store digital information that can be read with a laser diode. The disk is immune to dust and mechanical wear because of its optical nature.It comes in various size(3.5-14 inch) and stores huge amount of data from 100 mb to several gb.

Fig. 3 - Image of CD-ROM

12.4 Operating System and Programming Tools

1. The operating system of a computer is a group of programs that manages or oversees all the operations of the computer.

2. The Computer transfers information constantly among peripherals such as a floppy disk, printer, keyboard and video monitor.

3. It also stores user programs under file name on a disk. Each computer has its own operating system like MS-DOS(microsoft Disk Operating System), OS/2(Operating system 2), Windows 95, Windows xp, Windows 7,10 etc and Unix.

4. Older version of operating system like MS-DOS is being replaced by newer version such as microsoft windows newer versions, IBM OS/2 and unix/linux.

5. MS-DOS(Microsoft Disk Operating System)- It is a single user operating system used normaly in PC's.

   5.1 History of DOS- a) In 1979, a small company called seattle computer products wrote its own OS names as ODOS. IBM purchased ODOS and then took the help of microsoft to develop a new product. This product was announced with IBM-PC in 1981 with names as MS-DOS version 1.0. b)In 1983 MS-DOS ver 2.0 appeared with many advances in design. It was made for PC announced by IBM. It introduced fixed disk formatting, back up utilities and filter commands for redirection of input/output operations.

   5.2 Features- a) MS-DOS has enhanced version MS-DOS 7.0 in market which has GUI(Graphical user interface) facility. b)In 1991 tie up between IBM and Microsoft ended and microsoft started new series which name is window operating system.c)1981- DOS1.1 Renamed version of QDOS(Quick and Dirty operating system) which was purchased by Microsoft from Seattle Computer products. d)1982- supported use of double sided disks. e)1983-DOS 2.0 supported IBM's 10 mb hard disk and some other additional features. f)1984- DOS 3.0 support for high density 3.5 inch floppy disk. Allowed partition on hard disk. g)1991- DOS 5.0, much upgraded version included text editor and improved BASIC interpreter etc. h) 1993- DOS 6.0- Added a disk compression utility antivirus program and disk defragmenter. i)1995- DOS 7.0, this version is part of MS windows 95, supports long file names but remove a large number of utilities. j)1997-DOS 7.1, support for FAT 32 hard disks and is part of later versions of MS windows 95.

5.3 MS-DOS parts- The DOS OS is a set of programs which are stored on some secondary storage device, normally on floppy disk. It is then loaded in to RAM when required. The DOS software is divided in to three parts stored in three different files on disk. The disk which contains these three files is called a Bootable disk.

5.4 The three files loaded on this disk are IO.SYS, MSDOS.SYS and COMMAND.COM. The IO.SYS file contains the device drivers for the standard devices such as keyboard, disk, floppy, printer and monitor are present. All these device drivers are often called BIOS(Basic Input Output System). The MSDOS.SYS file is also called DOS Kernel. It contains all the modules for process management. These modules are written in machine independent manner so that they could be easily ported.

5.5 File in DOS-A file is a organized collection of data stored on storage device such as magnetic tape, disk. The file is used to store only one kind of information. Different types of files are used in computer such as text file, batch file, database file etc. Some file extensions and their meanings

- ✓ .CC Source program file
- ✓ .ftn Fortran source program file
- ✓ .pas Pascal Source program file
- ✓ .obj object file
- ✓ .exe Executable File

- ✓ .bak Backup File
- ✓ .dat Data File
- ✓ .Wav Microsoft windows sound file

5.6 File name in DOS- The file name can have up to 8 alphanumeric characters and an optional extension. The extension name can be up to 3 characters long. The periods(.) separates the primary name and optional extension. DOS permits following characters in a file name (A to Z, a to z, 0 to 9, $,&,##,%,(),@,!)

5.7 Directory- Files on the hard disk are divided in to various segments called directories. A directory can store any number of files. It helps to organize our files in an efficient manner. Just like filenames, directory name can also have up to eight alphanumeric characters. Directory is further classified in to three parts (Current directory, Sub directory, root directory) for eg C:\abc\xyz.

5.8 DOS provides two wildcard characters (*) and (?) , whereas (*) means single character replacement and (?) means any number of characters. For eg C:\>del**, C:\>dir??am.c

5.9 Piping- DOS also supports piping technique which permits us to combine multiple commands on a single command. The symbol is |

6 Window OS- Windows is a very user friendly and popular operating system developed by microsoft corporation company in 1985. Windows was single user OD initially but after windows 98 it was turned as multiuser Multitasking OS. S. Versions of Windows

- ➢ Windows 1.0
- ➢ Windows 2.0
- ➢ Windows 3.0
- ➢ Windows 95
- ➢ Windows 98
- ➢ Windows XP
- ➢ Windows 2000
- ➢ Windows 7
- ➢ Windows Vista
- ➢ Windows 10

6.1 The microsoft windows XP operating system is a 32 / 64 bit preemptive multi tasking operating system for AMD K6/K7, Intel IA32/IA64 and later multiprocessor.

6.2 The Successor to windows NT and windows 2000, windows XP is also intended to replace the windows 95/98 operating system.

6.3 In october 2001, windows XP was released as both an update to the windows 2000 desktop operating system and an replacement for windows 95/98. In 2002, the server versions of window xp become available called windows.Net Server.

6.4 Window XP provides better networking and device experience (including instant messaging, streaming media and digital photography/video), dramatic performance improvements for both the desktop and large multiprocessors and better reliability and security then earlier windows operating systems.

6.5 Window XP is a multi-user operating system, supporting simultaneous access through distributed services or through multiple instances of the GUI via the window terminal server.

6.6 Window XP was the first version of windows to ship a 64 bit version and hence all the higher version of window like 7-10 are available with 64 bit OS.

7. Unix- It is a multi user, multitasking OS. It was designed for mini computers but it is now used on various machines ranging from Microcomputers to supercomputers. It is widely used in Engineering, Scientific and Research Environment as it open source software.

8. Programming Tools/ Paradigm

    1. Operating System provide a lot of services to the user and in today's OS, will execute number of processes at a same time. The role of user for executing processes is to just create that process and wait for the output, because the execution of that process is the responsibility of operating system. For that OS having its own tools of performing smooth execution of all the process. Following are the tools described below as

        - User Management
        - Security Policy
        - Device Management
        - Performance Monitor
        - Task Scheduler

1. User Management- It describes the ability for administrators to manage user access to various IT resources like systems, devices, applications, storage systems, networks and more. It is a core part to any directory service and is a basic security essential for any organization. It enables admins to control user access and on-board and off-board users to and from IT resources. There are 3 types of accounts present in Unix OS

2. Root- This account belongs to system administrator. This is also called as super user. Super user has permission to run any command.

3. System Accounts- This account is required to system related components eg email

4. User Accounts- These accounts belongs to users and group of users. General user have these accounts and these user have limited to system files and directories.

2. Security Policy- There are many types of OS security policies and procedures that can be implemented based on the industry we work in. In general definition, an OS security policy is one that contains information of processes ensuring that the OS maintains a certain level of integrity, confidentiality and availability. OS security protects systems and data from threats, viruses, worms, malware, backdoor intrusions and more. Security policies cover all preventive measures and techniques to ensure the safeguarding of an OS, the network it connects to and the data which can be stolen, edited or deleted. Since OS Security policies and procedures cover a broad area there are many ways to address them. Some of these areas include

✓ Ensuring Systems are updated regularly

✓ Installing and updating anti-virus software

✓ Installing a firewall and ensuring it is configured properly to monitor all incoming and outgoing traffic

✓ Implementing user management procedures secure user accounts and privileges.

3. Device Management- Device management in OS implies the management of the I/O devices such as keyboard, magnetic tape, disk, printer, microphone, USB ports, scanner, camcorder etc as well as the

supporting units like control channels. The basic I/O devices can fall in to 3 categories

- ✓ Block Device- It stores information in fixed size block each one with its own address. For eg Disks.
- ✓ Character Device- It's delivers or accepts a stream of characters. The individual characters are not addressable. For eg Printers, Keyboard etc.
- ✓ Linux treats each device as file. Like file, it open the device, write data to it and read from it. After using the device, OS then closes it.
- ✓ Device drivers are responsible for treating every device as file. Device drivers is a program that controls particular device. When OS kernel writes data to a particular device then device drivers of that device carries out appropriate action for that device.

4. Performance Monitor- It can be used to display real time performance information as well as collect performance data using data collector sets and by saving the information in log files. We can also generate performance alerts based on specific thresholds for performance objects such as the processor, the hard disk, memory, networking interfaces and protocols and so on. It can be used to compare the performance information stored in two or more log files. We must use the performance monitor as a stand alone utility to use this feature.

5. Task Scheduler- is a component of microsoft windows that provides the ability to schedule the launch of programs or scripts at pre-defined times or after specified time intervals: Job scheduling (task scheduling)

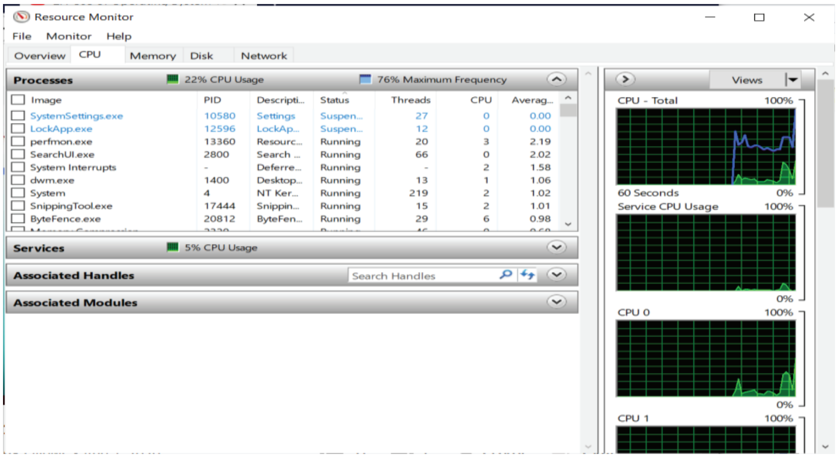


Image	PID	Descripti...	Status	Threads	CPU	Averag...
□ SystemSettings.exe	10580	Settings	Suspen...	27	0	0.00
□ LockApp.exe	12596	LockAp...	Suspen...	12	0	0.00
□ perfmon.exe	13360	Resourc...	Running	20	3	2.19
□ SearchUI.exe	2800	Search ...	Running	66	0	2.02
□ System Interrupts	-	Deferre...	Running	-	2	1.58
□ dwm.exe	1400	Desktop...	Running	13	1	1.06
□ System	4	NT Ker...	Running	219	2	1.02
□ SnippingTool.exe	17444	Snippin...	Running	15	2	1.01
□ ByteFence.exe	20812	ByteFen...	Running	29	6	0.98

Fig. 4 - Resource Monitor tool of OS

12.5 Assemblers and Cross-Assemblers, Writing Program Using Cross Assemblers

1. Assemblers

1. Assembler is a program for converting instructions written in low-level assembly code in to relocatable machine code and generating along information for the loader.

Assembly code
Assembler
Machine code

Fig1 Assembler

2. It generates instructions by evaluating the mnemonics (symbols) in operation field and find the value of symbol and literals to produce machine code. Now, if assembler do all this work then in one scan then it is called single pass assembler, otherwise if it does in multiple scans then called multiple pass assembler. Here assembler divide these tasks in two passes.

3. Pass-1-

   3.1.1 Define symbols and literals and remember them in symbol table and literal table respectively.

   3.1.2 Keep track of location counter.

   3.1.3 Process Pseudo-operations

4. Pass-2-

   4.1 Generate object code by converting symbolic op-code in to respective numeric op-code

   4.2 Generate data for literals and look for values of symbol.

5 For eg we will take a small assembly language program to understand the working in their respective passes. Assembly language statement format (Label, opcode and operand).

ADD R1='3'

Where M=label Add=Opcode an R1 is register operand

Label	Opcode	Operand	LC Value(Locat
J1	START	200	
	MOVE R	R1=3	200
	MOVE M	R1,X	201
L1	MOVE R	R2 =2	202
	LTORG		203
X	DS	1	204
	END		

Explanation of above code

- ✓ Start- This instructions starts the execution of program from location 200 and label with start provides name for the program (J1).
- ✓ MOVE R-It moves the content 3 in to register operand R1.
- ✓ MOVE M- It moves the content of register in to memory operand(X).
- ✓ MOVE R- It again moves the content 2 in to register operand R2 and its label is specified as L1
- ✓ LTORG- It assigns address to literals (current LC value).
- ✓ DS(Data Space)- It assigns a data space of 1 to symbol X.
- ✓ END-It finishes the program execution

6. Working of Pass-1- Define symbol and literal table with their addresses. Literal address is specified by LTORG or END.

6.1 START 200(here no symbol or literal is found so both table would be empty).

6.2 MOVER R1=3 200(=3 is a literal so literal table is made).

6.3 MOVEM R1, X201- X is a symbol prior to its declaration so it is stored in symbol table with blank address field.

6.4 L1 MOVER R2=2 202- L1 is a label and ='2' is a literal so store them in respective tables

6.5 LTORG 203-Assign address to first literal specified by LC value,
I.e 203

6.6 X DS 1 204- It is a data declaration statement I.e X is assigned data space of 1. But X is a symbol which was referred earlier in step 3 and defined in step 6. This condition is called Forward Reference Problem where variable is referred prior to its declaration and can be solved by back-patching. So now assembler will assign X the address specified by LC value of current step.

7 END 205 - Program finishes execution and remaining literal will get address specified by LC value of END instruction. Here is the complete symbol and literal table made by pass 1 assembler.

8. Now tables generated by pass 1 along with their LC value will go to pass-2 of assembler for further processing of pseudo-opcodes and machine op-codes.

9. Working of Pass-2- Pass-2 of assembler generates machine code by converting symbolic machine-opcodes in to their respective bit configuration(machine understandable form). It stores all machine-opcodes in MOT table(op-code table) with symbolic code, their length and their bit configuration. It will also process pseudo-ops and will store them in POT table(Pseudo-OP table).

Various Data bases required by pass-2

1. MOT table(machine opcode table)

2. POT table(Pseudo opcode table)

3. Base table(Storing value of base register)

4. LC(Location Counter)

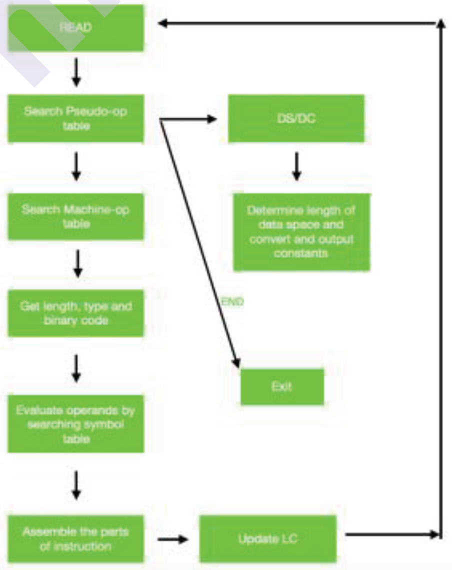

READ
Search Pseudo-op table
DS/DC
Search Machine-op table
Determine length of data space and convert and output constants
Get length, type and binary code
END
Exit
Evaluate operands by searching symbol table
Assemble the parts of instruction
Update LC

Fig. 2 - Flow chart of how assembly work

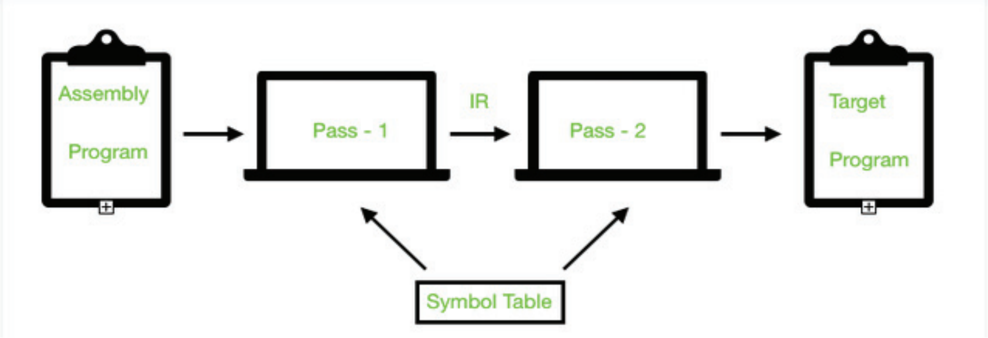


Fig. 3 - As a whole working of assembler

2. **Cross-Assembler**

A cross-assembler that runs on a computer with one type of processor but generates machine code for a different type of processor.For eg, if we use a PC with the 8086 compatible machine language to generate a machine code for the 8085 processor, we need a cross-assembler program that runs on the PC compatible machine but generates the machine code for 8085 mnemonics. It takes assembly language as input and give machine language as output.

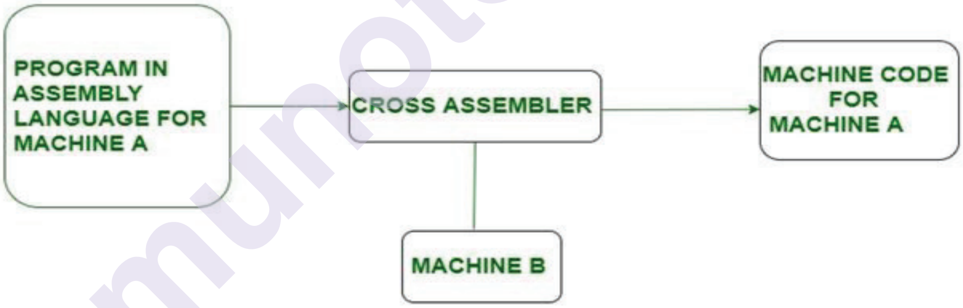

PROGRAM IN ASSEMBLY LANGUAGE FOR MACHINE A
CROSS ASSEMBLER
MACHINE CODE FOR MACHINE A
MACHINE B

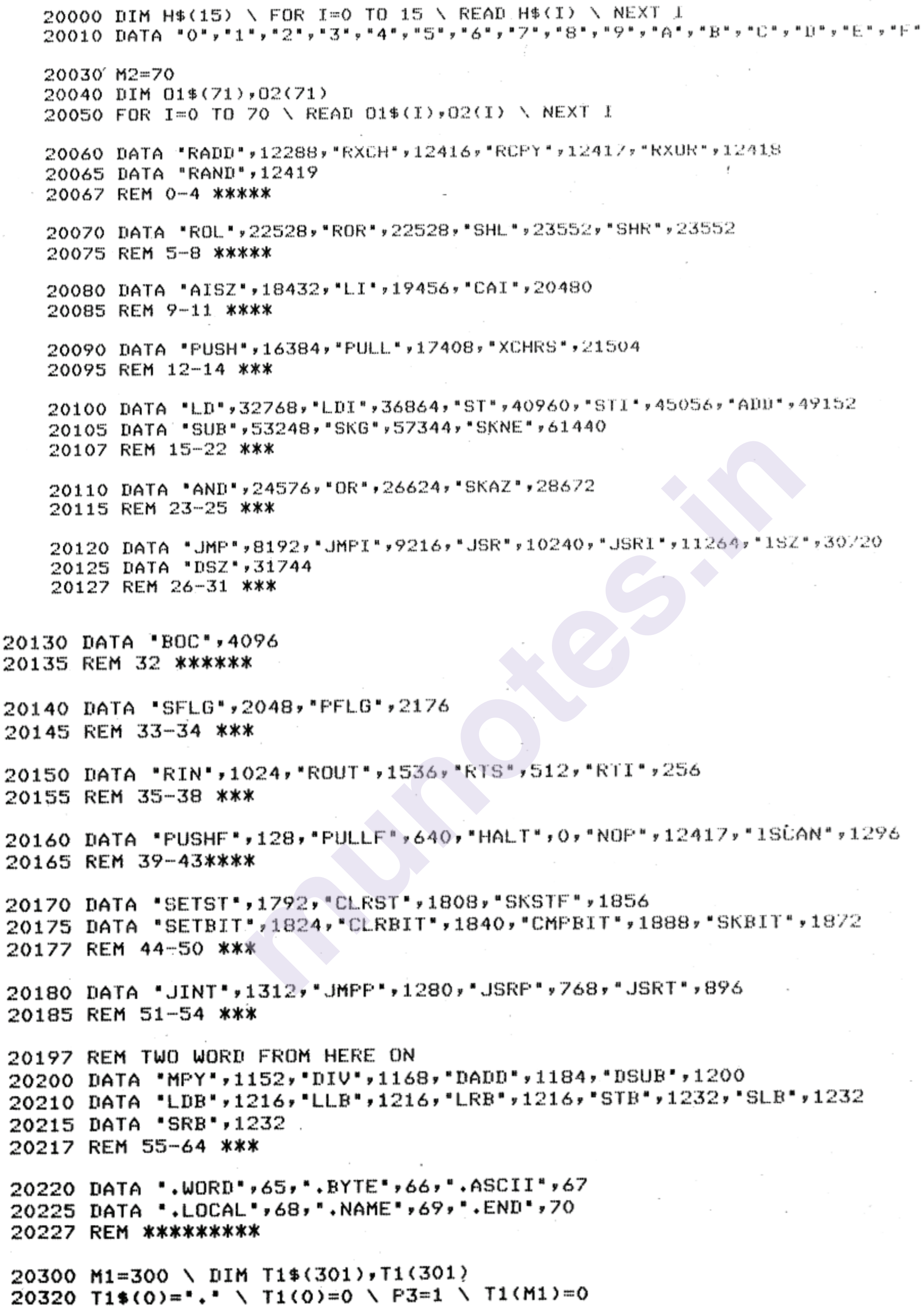
Fig. 4 - Cross Assembler

The above figure explains that there is an assembler which is running on machine B but converting assembly code of Machine A to machine code, this assembler is cross-assembler.

2.1 Features of Cross-Assembler

- ✓ It is used to convert assembly language in to binary machine code.
- ✓ They are also used to develop program which will run on game console and other small electronic system which are not able to run development environment on their own.
- ✓ It can be used to give speed development on low powered system

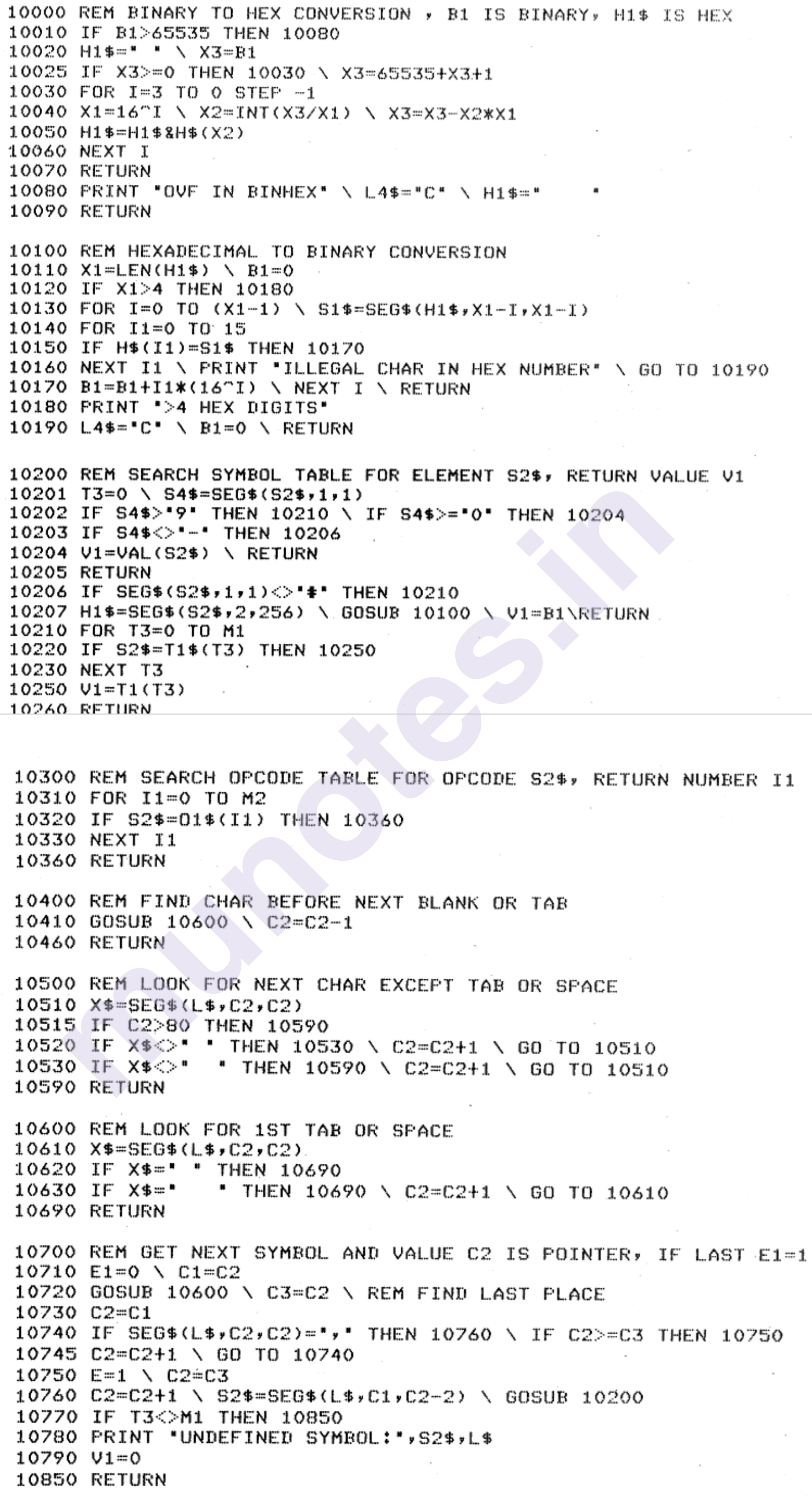
3. Writing program through Cross Assembler



```
20000 DIM H$(15) \ FOR I=0 TO 15 \ READ H$(I) \ NEXT I
20010 DATA "0","1","2","3","4","5","6","7","8","9","A","B","C","D","E","F"

20030 M2=70
20040 DIM O1$(71),O2(71)
20050 FOR I=0 TO 70 \ READ O1$(I),O2(I) \ NEXT I

20060 DATA "RADD",12288,"RXCH",12416,"RCPY",12417,"RXOR",12418
20065 DATA "RAND",12419
20067 REM 0-4 *****

20070 DATA "ROL",22528,"ROR",22528,"SHL",23552,"SHR",23552
20075 REM 5-8 *****

20080 DATA "AISZ",18432,"LI",19456,"CAI",20480
20085 REM 9-11 ****

20090 DATA "PUSH",16384,"PULL",17408,"XCHRS",21504
20095 REM 12-14 ***

20100 DATA "LD",32768,"LDI",36864,"ST",40960,"STI",45056,"ADD",49152
20105 DATA "SUB",53248,"SKG",57344,"SKNE",61440
20107 REM 15-22 ***

20110 DATA "AND",24576,"OR",26624,"SKAZ",28672
20115 REM 23-25 ***

20120 DATA "JMP",8192,"JMPI",9216,"JSR",10240,"JSRI",11264,"ISZ",30720
20125 DATA "DSZ",31744
20127 REM 26-31 ***

20130 DATA "BOC",4096
20135 REM 32 ******

20140 DATA "SFLG",2048,"PFLG",2176
20145 REM 33-34 ***

20150 DATA "RIN",1024,"ROUT",1536,"RTS",512,"RTI",256
20155 REM 35-38 ***

20160 DATA "PUSHF",128,"PULLF",640,"HALT",0,"NOP",12417,"ISCAN",1296
20165 REM 39-43****

20170 DATA "SETST",1792,"CLRST",1808,"SKSTF",1856
20175 DATA "SETBIT",1824,"CLRBIT",1840,"CMPBIT",1888,"SKBIT",1872
20177 REM 44-50 ***

20180 DATA "JINT",1312,"JMPP",1280,"JSRP",768,"JSRT",896
20185 REM 51-54 ***

20197 REM TWO WORD FROM HERE ON
20200 DATA "MPY",1152,"DIV",1168,"DADD",1184,"DSUB",1200
20210 DATA "LDB",1216,"LLB",1216,"LRB",1216,"STB",1232,"SLB",1232
20215 DATA "SRB",1232
20217 REM 55-64 ***

20220 DATA ".WORD",65,".BYTE",66,".ASCII",67
20225 DATA ".LOCAL",68,".NAME",69,".END",70
20227 REM *********

20300 M1=300 \ DIM T1$(301),T1(301)
20320 T1$(0)="." \ T1(0)=0 \ P3=1 \ T1(M1)=0
```

Fig 5. - Initialization of Cross Assembler

For eg Program for Binary to Hex conversion using Cross Assembler



```
10000 REM BINARY TO HEX CONVERSION , B1 IS BINARY, H1$ IS HEX
10010 IF B1>65535 THEN 10080
10020 H1$=" " \ X3=B1
10025 IF X3>=0 THEN 10030 \ X3=65535+X3+1
10030 FOR I=3 TO 0 STEP -1
10040 X1=16^I \ X2=INT(X3/X1) \ X3=X3-X2*X1
10050 H1$=H1$&H$(X2)
10060 NEXT I
10070 RETURN
10080 PRINT "OVF IN BINHEX" \ L4$="C" \ H1$="     "
10090 RETURN

10100 REM HEXADECIMAL TO BINARY CONVERSION
10110 X1=LEN(H1$) \ B1=0
10120 IF X1>4 THEN 10180
10130 FOR I=0 TO (X1-1) \ S1$=SEG$(H1$,X1-I,X1-I)
10140 FOR I1=0 TO 15
10150 IF H$(I1)=S1$ THEN 10170
10160 NEXT I1 \ PRINT "ILLEGAL CHAR IN HEX NUMBER" \ GO TO 10190
10170 B1=B1+I1*(16^I) \ NEXT I \ RETURN
10180 PRINT ">4 HEX DIGITS"
10190 L4$="C" \ B1=0 \ RETURN

10200 REM SEARCH SYMBOL TABLE FOR ELEMENT S2$, RETURN VALUE V1
10201 T3=0 \ S4$=SEG$(S2$,1,1)
10202 IF S4$>"9" THEN 10210 \ IF S4$>="0" THEN 10204
10203 IF S4$<>"-" THEN 10206
10204 V1=VAL(S2$) \ RETURN
10205 RETURN
10206 IF SEG$(S2$,1,1)<>"$" THEN 10210
10207 H1$=SEG$(S2$,2,256) \ GOSUB 10100 \ V1=B1\RETURN
10210 FOR T3=0 TO M1
10220 IF S2$=T1$(T3) THEN 10250
10230 NEXT T3
10250 V1=T1(T3)
10260 RETURN

10300 REM SEARCH OPCODE TABLE FOR OPCODE S2$, RETURN NUMBER I1
10310 FOR I1=0 TO M2
10320 IF S2$=O1$(I1) THEN 10360
10330 NEXT I1
10360 RETURN

10400 REM FIND CHAR BEFORE NEXT BLANK OR TAB
10410 GOSUB 10600 \ C2=C2-1
10460 RETURN

10500 REM LOOK FOR NEXT CHAR EXCEPT TAB OR SPACE
10510 X$=SEG$(L$,C2,C2)
10515 IF C2>80 THEN 10590
10520 IF X$<>" " THEN 10530 \ C2=C2+1 \ GO TO 10510
10530 IF X$<>"  " THEN 10590 \ C2=C2+1 \ GO TO 10510
10590 RETURN

10600 REM LOOK FOR 1ST TAB OR SPACE
10610 X$=SEG$(L$,C2,C2)
10620 IF X$=" " THEN 10690
10630 IF X$="   " THEN 10690 \ C2=C2+1 \ GO TO 10610
10690 RETURN

10700 REM GET NEXT SYMBOL AND VALUE C2 IS POINTER, IF LAST E1=1
10710 E1=0 \ C1=C2
10720 GOSUB 10600 \ C3=C2 \ REM FIND LAST PLACE
10730 C2=C1
10740 IF SEG$(L$,C2,C2)="," THEN 10760 \ IF C2>=C3 THEN 10750
10745 C2=C2+1 \ GO TO 10740
10750 E=1 \ C2=C3
10760 C2=C2+1 \ S2$=SEG$(L$,C1,C2-2) \ GOSUB 10200
10770 IF T3<>M1 THEN 10850
10780 PRINT "UNDEFINED SYMBOL:",S2$,L$
10790 V1=0
10850 RETURN
```

Fig. 6 - Program in Cross Assembler with sub routines

Miscellaneous Questions

Q1. Describe about software development of micro processor

Q2. Elaborate about uses of operating tools

Q3. Illustrate the concept of Operating System

Q4. Describe about Cross Assembler

❖❖❖

Unit 4

13

INTERRUPTS

Unit Structure

13.0 Objectives

13.1 Introduction

13.2 8085 Interrupt

13.2.1 Restart Instructions

13.2.2 Implementation of Interrupt Process

13.2.3 Multiple Interrupts & Priorities

13.3 8085 Vectored Interrupts

13.3.1 TRAP

13.3.2 RST 7.5, RST 6.5 and RST 5.5

13.3.3 Interrupt Driven Clock Illustration

13.4 Restart as Software Instructions

13.4.1 Breakpoint Technique Illustration

13.5 Additional I/O Concepts

13.5.1 Programmable Interrupt Controller

13.5.2 Direct Memory Access

13.6 Summary

13.7 List of References

13.8 Unit End Exercise

13.0 Objectives

After going through this chapter, you will be able to

- Understand the 8085 interrupt process
- Know the difference between the types of interrupt and how to handle them
- Interpret the working of the interrupt instructions

- Understand how to handle multiple interrupts and set their priorities
- Understand the use of Restart Instructions
- Learn the features of Programmable Interrupt Controller and Direct Memory Access transfer

13.1 Introduction

- Interrupt is defined as signal which suspends the normal execution of the microprocessor and gets itself serviced.
- It could be simple task of data transfer by peripheral or any external device which informs the processor that it requires immediate attention and suspend the ongoing activity.
- So a particular task is assigned to each interrupt signal that the microprocessor must handle.
- The process is asynchronous as the peripheral can interrupt any time but the response is fully under the control of the microprocessor.

13.2 8085 Interrupt

- The interrupt processor of the 8085 microprocessor is controlled by the Interrupt Enable (IE) flip-flop which can be set (logic 1) or reset (logic 0) with help of the instructions.
- When the flip-flop is enabled, the microprocessor is interrupted with the INTR (Interrupt Request) pin which is maskable and is disabled.
- The interrupt process of 8085 can be handled in a similar way as receiving and responding a telephone call while we are sipping a coffee and enjoying reading the novel
- The interrupt process is as follows:
    - o The 8085 microprocessor should be ready. This is done by enabling the interrupt process by writing the instruction EI (Enable Interrupt) in the main program. The telephone system in a similar way is enabled meaning the receiver is on the hook.
    - o The microprocessor checks the INTR signal during program execution. It is similar to glancing the telephone to see if lights are on and it is functioning.

  - If INTR is found high, microprocessor finishes current execution by disabling the IE flip-flop and sends active low $\overline{INTA}$ (Interrupt Acknowledge) telling the microprocessor not to accept any request till the flip-flop is enabled again. It is similar to picking up the receiver on seeing blinking light and until the person places the receiver back, no phone calls can be received as the line is busy,
  - With the support of external hardware, the signal $\overline{INTA}$ inserts RST (Restart) instructions which transfers control to specific location and restarts execution at that location in next step. It is similar to receiving a call to shut the window as there is sandstorm.
  - On receiving the RST instruction, the address of the program counter (next instruction) is saved onto stack. This is similar to inserting a bookmark in the page we were reading so that we can get back to it when we finish attending the call.
  - The code written at the new location is called the ISR (Interrupt Service Routine) which the processor performs. It is similar to closing the window as phone call instructed it to do so.
  - The ISR includes the instruction EI to enable the interrupt process again like the person hooking up the telephone as the task is done.
  - Finally the RET instruction in the ISR returns the control to the main program where the microprocessor was interrupted to continue the execution. This is similar to going back to the book, picking up the bookmark and continuing the reading operation.

- The important instructions to carry the above process are
  - EI – Enable Interrupt. It is 1-byte instruction that sets the IE flip-flop to handle the interrupt process.
  - DI – Disable Interrupts. It is 1-byte instruction that resets the IE flip-flop and disables the interrupt process.

13.2.1 Restart Instructions

- The 8085 microprocessor has eight RST (Restart) instructions.
- They are 1 byte instructions that transfer the program control to specific address and executed in similar way as CALL instruction.
- The address in the program counter is stored in stack and control is transferred to the RST defined vector address.

- The microprocessor when encounters RET (Return) instruction in the subroutine, is pops the address from the stack and begins execution of the main program
- The hex code and address of restart instruction is tabulated as follows

Table 13.1 – RST Instruction

Mnemonics	Binary Code								Hex	Address
	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Code	In Hex
RST 0	1	1	0	0	0	1	1	1	C7	0000
RST 1	1	1	0	0	1	1	1	1	CF	0008
RST 2	1	1	0	1	0	1	1	1	D7	0010
RST 3	1	1	0	1	1	1	1	1	DF	0018
RST 4	1	1	1	0	0	1	1	1	E7	0020
RST 5	1	1	1	0	1	1	1	1	EF	0028
RST 6	1	1	1	1	0	1	1	1	F7	0030
RST 7	1	1	1	1	1	1	1	1	FF	0038

- The RST instructions are widely used in hardware interrupt.
- It is inserted in the microprocessor by external hardware and the signal $\overline{INTA}$.
- For example consider the instruction RST 5 which is built using resistors and a tri-state buffer as shown in figure.

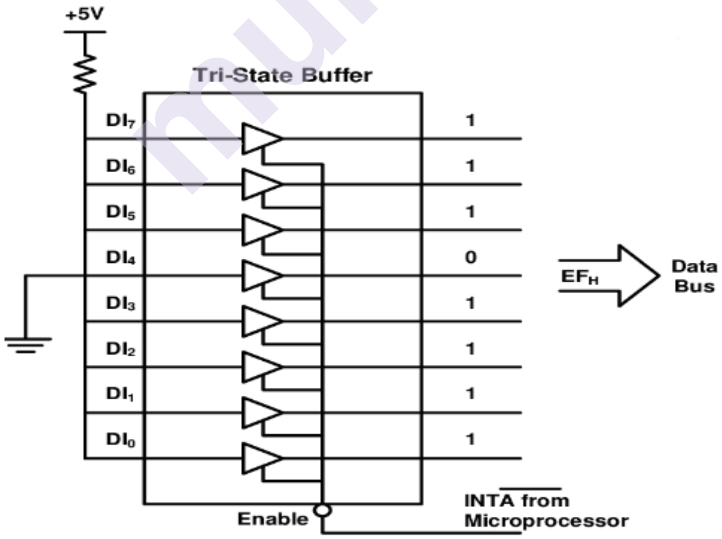


Figure 13.1 – RST 5 Circuit

- The timing diagram of $\overline{INTA}$ signal consist of three machine cycle.

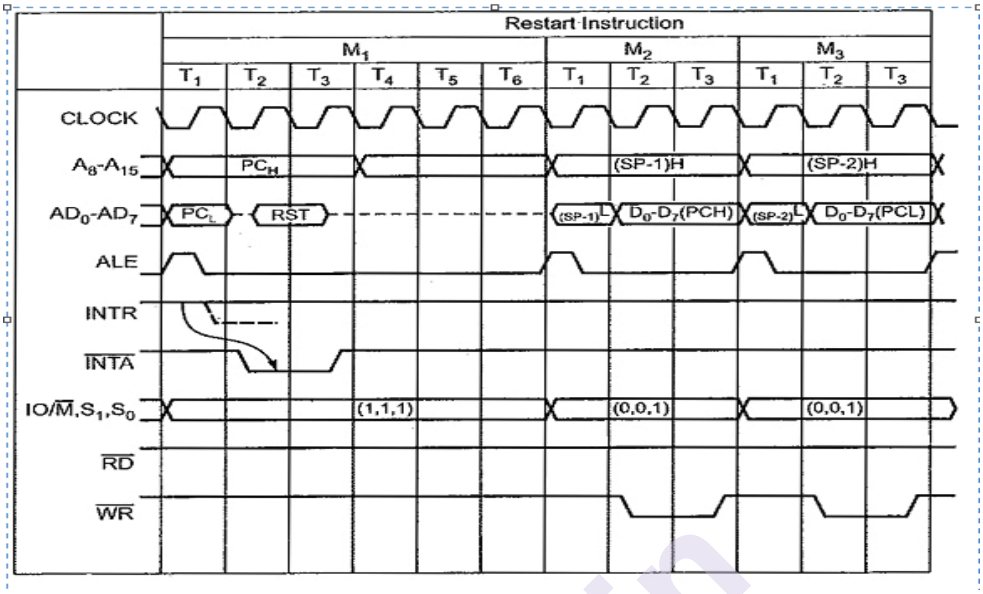


Figure 13.2 – $\overline{INTA}$ and RST 5 Timing Diagram

- During the M1 cycle, $\overline{INTA}$ is used to enable buffer and RST code is placed on data bus and address in the program counter is stored in stack to be retrieved later.
- It is similar to Opcode Fetch Cycle, with $\overline{INTA}$ signal instead of $\overline{RD}$ and status signals IO/$\overline{M}$, S_0 and S_1 = 111 instead of 011.
- During the machine cycle M2, the high order address of the program counter is stored on the stack and during the machine cycle M3, the low order address of the program counter is stored on the stack.
- The Machine cycle M2 and M3 are memory write cycle that stores content of program counter on stack.

13.2.2 Implementation Of Interrupt Process

- **Problem Statement**
  - A program to count continuously in binary with a one second delay between each count. A service routine to flash FFH five times when the program is interrupted, with some appropriate delay between each count.

- **Main Program**

Address	Label	Mnemonics
XX00		LXI SP, XX99H
XX03		EI
XX04		MVI A,00H
XX06	loop:	OUT 01H
XX08		MVI C, 01H
XX0A		CALL delay
XX0D		INR A
XX0E		JMP loop

- **Interrupt Service Routine**

Address	Label	Mnemonics
XX50	IS:	PUSH B
XX51		PUSH PSW
XX52		MVI B, 0AH
XX54		MVI A, 00H
XX56	loop:	OUT 00H
XX58		MVI C, 00H
XX5A		CALL delay
XX5D		CMA
XX5E		DCR B
XX5F		JNZ loop
XX62		POP PSW
XX63		POP B
XX64		EI
XX65		RET

- **Description Of The Interrupt Process**
  - The main program is at memory address XX00H, the delay subroutine at XX30H, interrupt service routine at address XX50H and stack pointer at XX99H.
  - The program counts from 00H to FFH continuously with delay of one second
  - Let the INTR line be pulled high in order to interrupt the processor while PC hold address XX06H.

- The microprocessor completes execution of instruction at that address (OUT 01H).
- It then sends $\overline{INTA}$ signal, disables other interrupt, enables the tri-state buffer and places RST 5 (EF) code on data bus.
- The address XX08H (next instruction – MVI C, 01H) is placed on stack and program control is transferred to location 0028H where the instruction **JMP XX50H** redirects it to the subroutine definition at XX50H.
- The subroutine is executed which loads ten in register B to output five count and five blank and RET statement retrieves the address from stack (XX08H) and return control to main program.

- **Testing Interrupt on Single-Board Computer System**
  - As discussed above, the program control is transferred to address 0028H.
  - But this location is not accessible to users and system designers prefix the code at this location to be redirected to another address such as

    0028 JMP 20D0H
  - RST 5 transfers control to 0028H, which redirects to 20D0 and where we write the instruction to transfer to out interrupt service routine definition at XX50H as 20D0 JMP IS

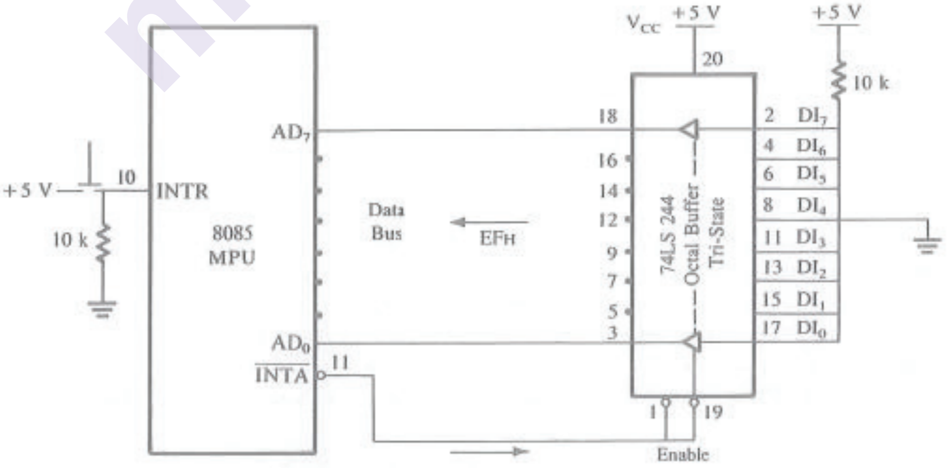


Figure 13.3 – Interrupt Implementation

13.2.3 Multiple Interrupts and Priorities

- When we handle single interrupt, we can invoke the interrupt process using the INTR signal.
- But to handle multiple interrupts we need a 8:3 priority encoder that helps in determining the priority among multiple devices.
- The address line A2- A0 are connected to data lines D5-D3 through tri-state buffer and 8 devices can be connected.
- The interrupting device request for service, then that input line goes low which makes Enable line E0 go high to interrupt the processor.
- $\overline{INTA}$ Acknowledges and enables buffer and corresponding code for example EF (RST 5) is placed on data line.
- If there are simultaneous interrupts from multiple devices then priority is determined by higher-level input.
- So device connected to Pin I7 has always the highest priority and this is drawback of this technique.

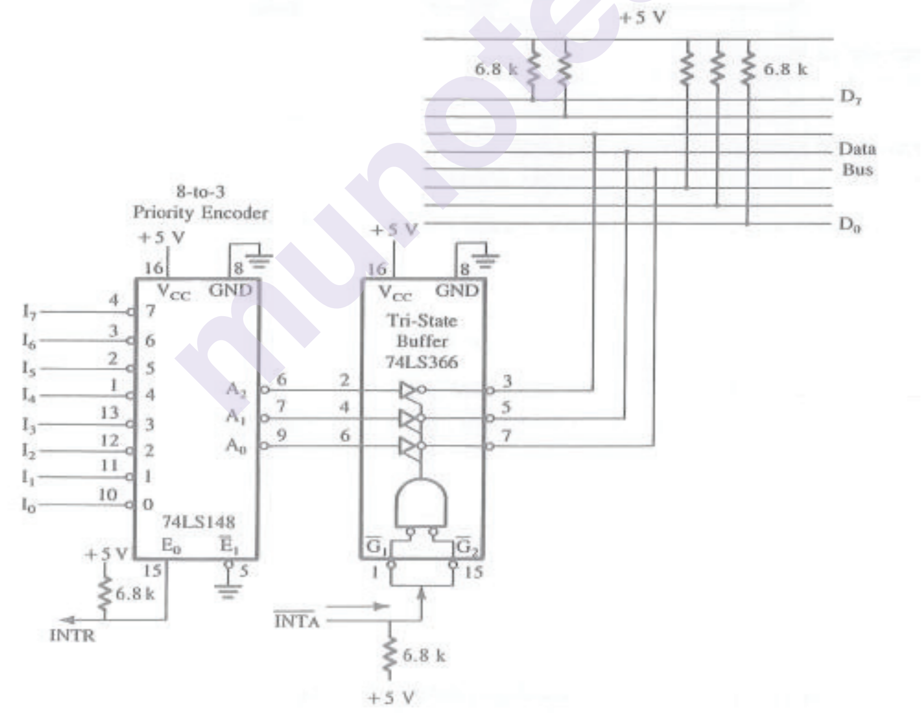


Figure 13.3 – 8:3 Priority Encoder to handle multiple interrupts

13.3 8085 Vectored Interrupts

- The 8085 microprocessor has five interrupts - INTR, RST 7.5, RST 6.5, RST 5.5 and TRAP.
- Last four are vectored to specific location on memory without support of external hardware as it is implemented inside 8085 microprocessor.

Table 13.2 – Interrupts and their call locations

Interrupt	Call Location	Priority
TRAP	0024H	Highest
RST 7.5	003CH	↓
RST 6.5	0034H	
RST 5.5	002CH	Lowest

- INTR has the lowest priority among all.

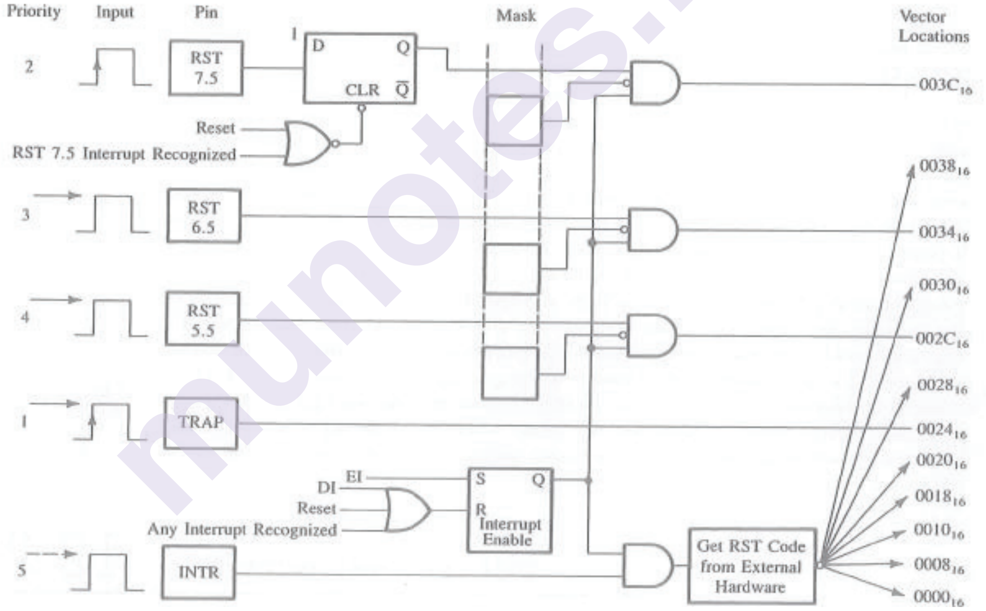


Figure 13.4 – 8085 Interrupts and Vector Location

13.3.1 Trap

- TRAP is non-maskable interrupt highest priority interrupt which cannot be disabled and used in critical events.
- It is both level and edge sensitive and stays high to be acknowledged.
- When this interrupt occurs, the control is transferred to location 0024H.
- It requires no external hardware or EI instruction support.

13.3.2 RST 7.5, 6.5, AND 5.5

- These are maskable interrupts under the supervision of two instructions – EI and SIM
- RST 7.5 is positive edge triggered with short pulse and cleared by Reset or bit D4 in SIM Instruction.
- RST 6.5 and 5.5 are level-sensitive and microprocessor is unable to service it immediately then it is stored externally.
- The entire interrupt process except TRAP is disabled by resetting the IE flip-flop.
- The important instructions are - SIM (Set Interrupt Mask) and RIM (Read Interrupt Mask)
- **SIM** is 1 byte instruction and depending on the value in the accumulator and has three functions - One function is to set mask for RST 7.5, 6.5 and 5.5, the second function is to reset RST 7.5 and the third function is for transmit serial data

D7	D6	D5	D4	D3	D2	D1	D0
SOD	SDE	xxx	R7.5	MSE	M7.5	M6.5	M5.5

- Bit D_2-D_0 sets the mask. 0 → available and 1 → masked.
- Bit D_3 is Mask Set Enable 0 → bits 0-2 are ignored and 1 → mask is set
- Bit D_4 resets RST 7.5. 1 → reset.
- Bit D_5 is ignored.
- Bit D_6 if 1 is output to Serial Output Data Latch
- Bit D_7 is Serial Output Data and ignored if bit 6=0
- **RIM** is 1 byte instruction and depending on the value in the accumulator and has three functions - One function is to read current status of interrupt masks, the second function is to identify pending interrupts and the third function is to receive serial data.

D7	D6	D5	D4	D3	D2	D1	D0
SID	I7.5	I6.5	I5.5	IE	M7.5	M6.5	M5.5

- Bit D_2-D_0 reads the mask. 0 → available and 1 → masked.
- Bit D_3 is Interrupt Enable Flag 1 → enabled
- Bit D_6 – D_4 represent pending interrupts 1 → pending
- Bit D_7 is Serial Input Data if any

13.3.3 Interrupt Driven Clock Illustration

- **Problem Statement**
  - Design a 1-minute timer using a 60 Hz power line as an interrupting source. The output ports should display minutes and seconds in BCD. At the end of the minute, the output ports should continue displaying one minute and zero seconds.

- **Hardware Description**
  - This timer is designed with a 60 Hz AC line with frequency of 16.6 ms
  - The circuit uses a step down transformer, the 74121 monostable multivibrator to provide appropriate pulse width and interrupt pin RST 6.5 which will transfer control to address 0034H when interrupted.
  - The interrupt flip-flop is enabled again within 6 μs in the timer service routine;

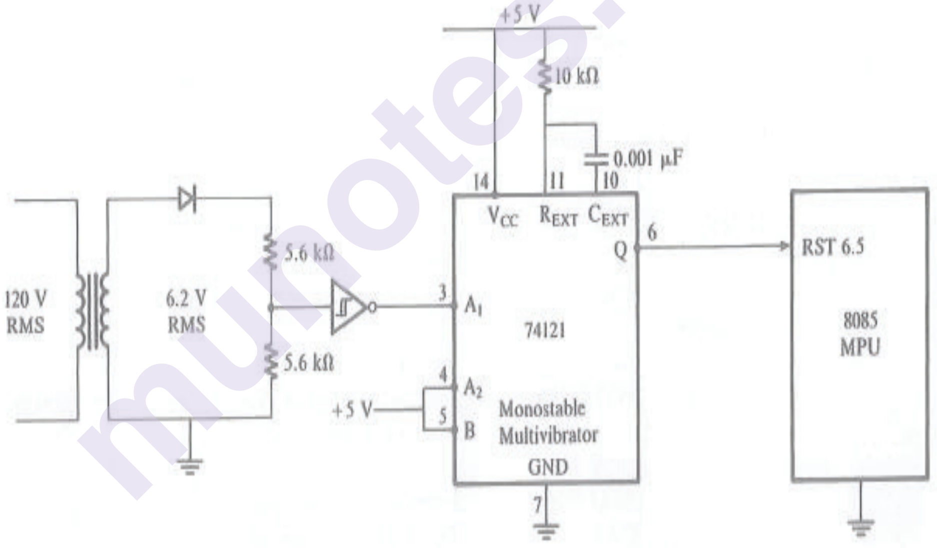


Figure 13.5 – Interrupt Driven Clock

- **Monitor Program**

0034 JMP RWM

- **Main Program**

Address	Label	Mnemonics
XX00		LXI SP, XX99H
XX03		RIM
XX04		ORI 08H
XX06		SIM
XX07		LXI B, 0000H
XX0A		MVI D, 3CH
XX0C		EI
XX0D	loop:	MOV A, B
XX0E		OUT 01H
XX10		MOV A, C
XX11		OUT 02H
XX13		JMP loop
XX16	RWM:	JMP rout

- **Interrupt Service Routine**

Address	Label	Mnemonics
XX50	rout:	DCR C
XX51		EI
XX52		RNZ
XX53		DI
XX54		MVI D, 3CH
XX56		MOV A,C
XX57		ADI 01H
XX59		DAA
XX5A		MOV C,A
XX5B		CPI 60H
XX5D		EI
XX5E		RNZ
XX5F		DI
XX60		MVI C, 00H
XX62		INR B
XX63		RET

- **Program Description**
  - The main program is at memory address XX00H, interrupt service routine at address XX50H and stack pointer at XX99H.

- The main program clears B register to store minutes and C registers to store seconds with initial values 00H in both and loads D register with 60H to count and enables the interrupt as well.
- SIM instruction triggers RST 6.5 which transfer control to vector address 0034H.
- The service routine defines three sections, where in the first section the D register is decremented every second and interrupt is enabled and control is transferred to main program and in the second section it increments the seconds counter and returns control to main program and in the final section it increments the minute counter and returns back to the main program.

13.4 Restart as Software Instructions

- Usually external hardware inserts RST instruction when requested to INTR.
- But RST are software instructions and used to set breakpoints which is a useful debugging technique.
- A breakpoint is RST instruction where the execution of program is stopped temporarily and control is transferred to RST defined vector address.
- During this the user examines register and memory content on key press.
- Once the routine is executed, the control is again transferred back to the main program where the breakpoint was set.

13.4.1 Breakpoint Technique Illustration

- **Problem Statement**
  - Write a subroutine to implement breakpoint at RST 5 and display accumulator and flag content when key A is pressed and exit the routine when key 0 is pressed.
- **Problem Analysis**
  - The accumulator and flag contents is displayed when RST instruction is encountered.
  - The register contents are stored on stack.
  - When A key is pressed the content of accumulator is displayed and wait for key pressed and retrieve content from stack
  - When 0 key is pressed it should return to main program.

- **Breakpoint Subroutine**

Address	Label	Mnemonics
XX50	brkpt:	PUSH PSW
XX51		PUSH B
XX52		PUSH D
XX53		PUSH H
XX54	kychk:	CALL kbrd
XX57		CPI 0AH
XX59		JNZ loop
XX5C		LXI H, 0007h
XX5F		DAD SP
XX60		MOV A.M
XX61		OUT 01H
XX63		DCX H
XX64		MOV A, M
XX65		OUT 02H
XX67		JMP kychk
XX6A	loop:	CPI 00H
XX6C		JNZ kychk
XX6F		POP H
XX70		POP D
XX71		POP B
XX72		POP PSW
XX73		RET

- **Program Description**
  - The breakpoint subroutine is located at the address XX50H.
  - Initially all registers are stored on the stack
  - When key press is detected, A key in our case, the HL register adds SP content without modifying SP data.
  - This is of due importance as if stack contents are altered then data will not be retrieved correctly with POP and RET instruction.
  - The accumulator content is displayed at output port 01H and flag register content is displayed at output port 02H.

13.5 Additional I/O Concepts and Processes

- There is single interrupt pin in 8085 microprocessor and this limits the performance to determine interrupt priorities.
- These limitations are overcome by using programmable interrupt controller which extends the capability of 8085 microprocessor.
- Also Direct Memory Access is another interrupt technique which facilitates high speed data transfer.

13.5.1 Programmable Interrupt Controller

- The 8259A is the programmable interrupt controller managing device using the signal INTR/INT for its operation.
- It can handle eight interrupt request and can transfer control to any vector address in the memory with no additional hardware support and restart instructions.
- However the request are spaced at interval of four or eight locations.
- The eight levels can be resolved in several modes and biggest advantage is that it can be expanded up to sixty four levels with additional 8259A device.
- The shortcoming of 8085 interrupt process is that all interrupt request are redirected to vector address 00H which is reserved in ROM and accessing this location after the system is designed is difficult.
- Also additional hardware support to insert and execute restart instructions make things complex.
- But these are easily overwhelmed by the 8259A.
- The 8259A consist of control logic, registers to manage the interrupt request, priority resolver to determine the priority, cascade logic to connect additional 8259A device and internal bus for communication
- The instructions are written in device register.
- Then interrupt request lines go high requesting the service (multiple request can occur).
- The 8259A resolves priorities and sends INT signal which is acknowledged by $\overline{INTA}$.
- After acknowledgment is received, the CALL instruction is executed and twice $\overline{INTA}$ signal is issued to read the 8 bit low order address and then 8-bit high order address of the interrupt vector address.

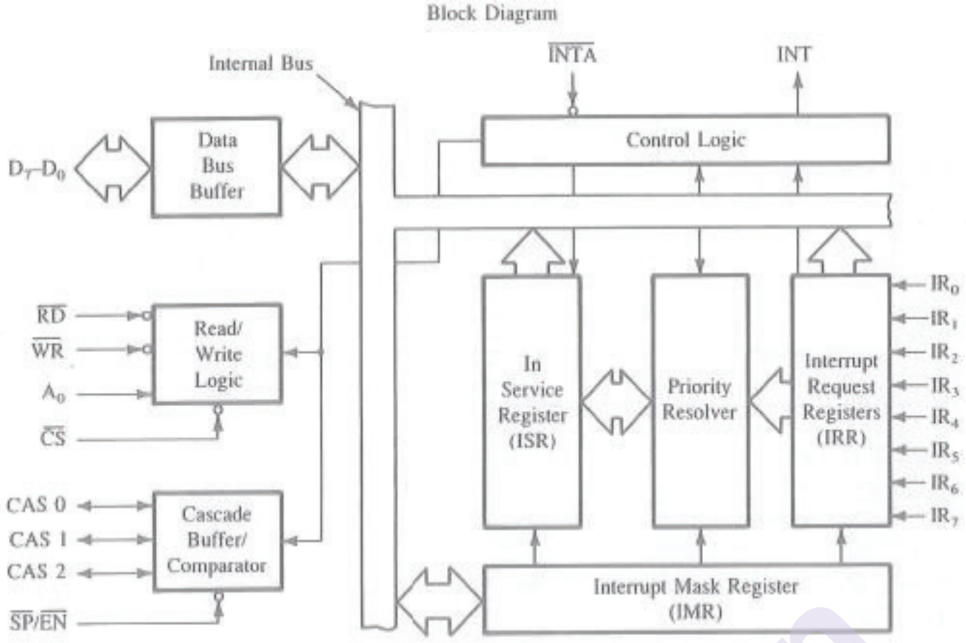


Figure 13.6 – 8259A Architecture Diagram

- The control then transfers to the address specified by CALL instruction.
- 8259A also can read status and change interrupt mode during program execution.

13.5.2 Direct Memory Access

- Direct Memory Access (DMA) is communication or data transfer between memory and peripherals without intervention of processor.
- This is done because the peripherals are slow devices and processor cycles are wasted waiting for the slow responding devices
- The 8085 microprocessor has two pins to support the DMA communication.
  - HOLD (Hold) – It is active high input signal to the 8085 from requesting device to use the system buses.
  - HLDA (Hold Acknowledge) – It is active high output signal indicating microprocessor is relinquishing the control of the system bus
- The 8257 DMA controller is commonly used.
- The controller sends the request to the HOLD pin of the processor.
- The processor completes the current execution and floats the system bus in high impedance state and sends HLDA signal.

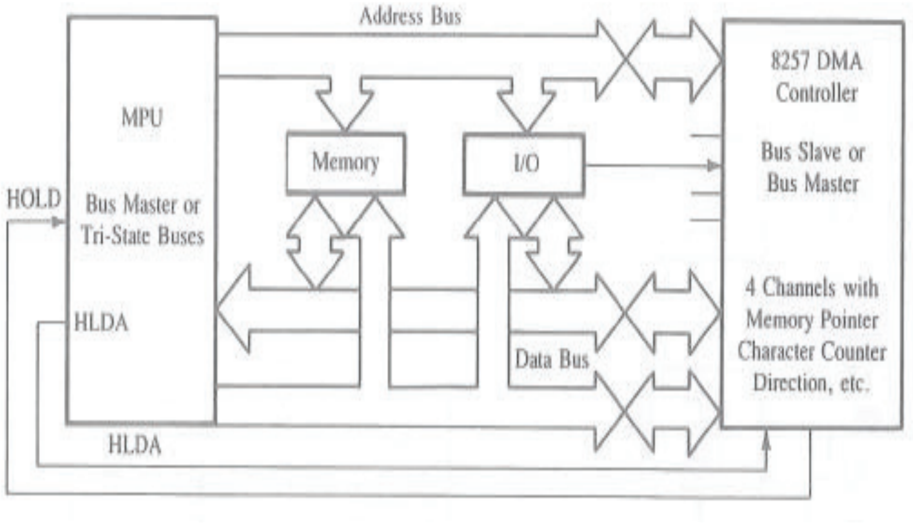


Figure 13.7 – 8257 DMA Communication

- The DMA controller now have control over the buses and transfer data between memory and peripherals
- Once exchange is over a low signal is sent to HOLD pin and microprocessor gain regains the control of the buses.

13.6 Summary

- The interrupt is asynchronous process of communication between microprocessor and peripherals or external device
- 8085 has maskable and non maskable interrupts
- Instruction EI and DI are used to enable and disable the interrupt mask for the 8085 microprocessor.
- Instruction SIM and RIM are used to implement and read the status of the various interrupts
- The restart instruction are software instructions and the transfer the control to vectored location
- Multiple interrupts and priorities can be handled using a priority encoder.
- A better way to achieve and improve the interrupt process of 8085 microprocessor is by using the Programmable Interrupt Controller 8259A.
- Using the 8257 DMA controller, high speed data transfer under control of external devices can be easily achieved.

13.7 List Of References

- Ramesh Gaonkar, “Microprocessor Architecture, Programming and Applications with the 8085”, Fifth Edition, Penram International Publishing (I) Private Limited.
- https://tutorialspoint.com
- https://www.brighthubengineering.com
- https://www.javatpoint.com

13.8 Unit End Exercise

1. Explain the following instructions (i) EI (ii) DI (iii) RST 5 (iv) SIM (v) RIM
2. Explain the working of an interrupt in 8085 microprocessor.
3. Illustrate the timing and data flows for 8085 Interrupt acknowledge machine cycle and execution of RST instruction.
4. Explain the working of 8259A Programmable Interrupt Controller.
5. Write a short note on Direct Memory Access (DMA).

❖❖❖

UNIT 5

14

THE PENTIUM AND PENTIUM PRO MICROPROCESSORS

Unit Structure

14.0 Objectives:

14.1 Introduction to Pentium Processors

14.2 Special Pentium Registers

14.2.1 The Programming Model

14.2.1.1 Multipurpose Registers

14.2.1.2 Special-purpose Registers

14.2.1.3 Segment Registers

14.3 Memory Management

14.3.1 Real Mode Memory Addressing

14.3.1.1 Segments and Offsets

14.3.1.2 Default Segment and Offset Registers

14.3.2 Segment and Offset Addressing Scheme Allows Relocation

14.3.2 Memory Paging

14.3.2.1 Paging Registers

14.3.3.2. The Page Directory and Page Table

14.4. Pentium Instructions

14.4.1 Instruction Set

14.4.2 Data Movement

14.4.3 Integer Arithmetic

14.4.4 Logical

14.4.5 Floating Point Arithmetic

14.4.6 I/O

14.4.7 Control Instructions

14.5. Pentium Pro Microprocessors

14.5.1 Modes

14.5.2 Register Set

14.5.3 Addressing

14.5.4 Processor Reset

14.5.5 Assembly Programming

14.5.5.1 Memory operands

14.5.5.2 Instruction Syntax

14.5.5.3 Assembler Directives

14.5.5.4 Inline Assembly

14.6. Special Pentium Pro Features

14.7. Summary

14.8. Review Your Learnings:

14.9. Sample Questions:

14.10. References for further reading

14.0 Objectives

1. Explain Pentium Processors
2. Explain features of Pentium Processors
3. Understand and explain Assembly Programs
4. Explain difference between Pentium and Pro-Pentium Processors
5. Explain various registers used in Pentium
6. Explain Pentium Instructions sets

14.1 Introduction to Pentium Processors

Pentium is a brand used for a series of x86 architecture-compatible microprocessors produced by Intel since 1993. In their form as of November 2011, Pentium processors are considered entry-level products that Intel rates as "two stars", meaning that they are above the low-end Atom and Celeron series, but below the faster Intel Core line-up, and workstation Xeon series.

They are based on both the architecture used in Atom and that of Core processors. In the case of Atom architectures, Pentiums are the highest performance implementations of the architecture. Pentium processors with Core architectures prior to 2017 were distinguished from the faster, higher-end *i-series* processors by lower clock rates and disabling some features, such as hyper-threading, virtualization and sometimes L3 cache.

The name Pentium is originally derived from the Greek word *pente* meaning "five", a reference to the prior numeric naming convention of Intel's 80x86 processors (8086–80486), with the Latin ending -*ium* since the processor would otherwise have been named 80586 using that convention.

The Pentium family of processors originated from the 80486 microprocessor. The term "Pentium processor" refers to a family of microprocessors that share a common architecture and instruction set. It runs at a clock frequency of either 60 or 66 MHz and has 3.1 million transistors.

Some of the features of Pentium architecture are:

1. Complex Instruction Set Computer (CISC) architecture with Reduced Instruction Set Computer (RISC) performance.
2. 64-Bit Bus
3. Upward code compatibility.
4. Pentium processor uses Superscalar architecture and hence can issue multiple instructions per cycle.
5. Multiple Instruction Issue (MII) capability.
6. Pentium processor executes instructions in five stages. This staging, or pipelining, allows the processor to overlap multiple instructions so that it takes less time to execute two instructions in a row.
7. The Pentium processor fetches the branch target instruction before it executes the branch instruction.
8. The Pentium processor has two separate 8-kilobyte (KB) caches on chip, one for instructions and one for data. It allows the Pentium processor to fetch data and instructions from the cache simultaneously.
9. When data is modified, only the data in the cache is changed. Memory data is changed only when the Pentium processor replaces the modified data in the cache with a different set of data

10. The Pentium processor has been optimized to run critical instructions in fewer clock cycles than the 80486 processor.

11. 8 bytes of data information can be transferred to and from memory in a single bus cycle.

12. Supports burst read and burst write back cycles.

13. Supports pipelining.

14. Instruction cache.

15. 8 KB of dedicated instruction cache.

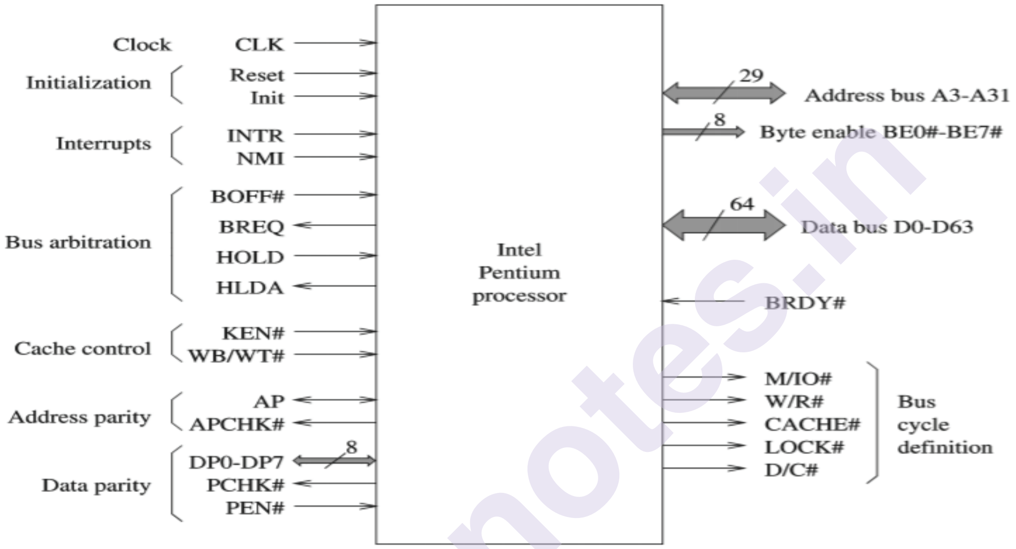


Figure 1.1: Pentium Processor

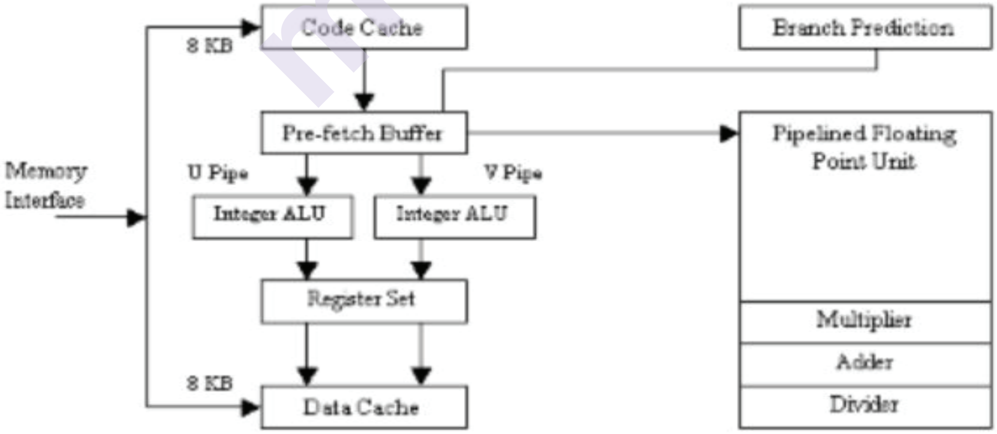


Figure 1.2: Architecture of Pentium Processor

The Pentium processor has two primary operating modes -

1. **Protected Mode** - In this mode all instructions and architectural features are available, providing the highest performance and capability. This is the recommended mode that all new applications and operating systems should target.

2. **Real-Address Mode** - This mode provides the programming environment of the Intel 8086 processor, with a few extensions. Reset initialization places the processor in real mode where, with a single instruction, it can switch to protected mode.

The Pentium's basic integer pipeline is five stages long, with the stages broken down as follows:

1. **Pre-fetch/Fetch:** Instructions are fetched from the instruction cache and aligned in pre-fetch buffers for decoding.

2. **Decode1:** Instructions are decoded into the Pentium's internal instruction format. Branch prediction also takes place at this stage.

3. **Decode2:** Same as above, and microcode ROM kicks in here, if necessary. Also, address computations take place at this stage.

4. **Execute**: The integer hardware executes the instruction.

5. **Write-back:** The results of the computation are written back to the register file.

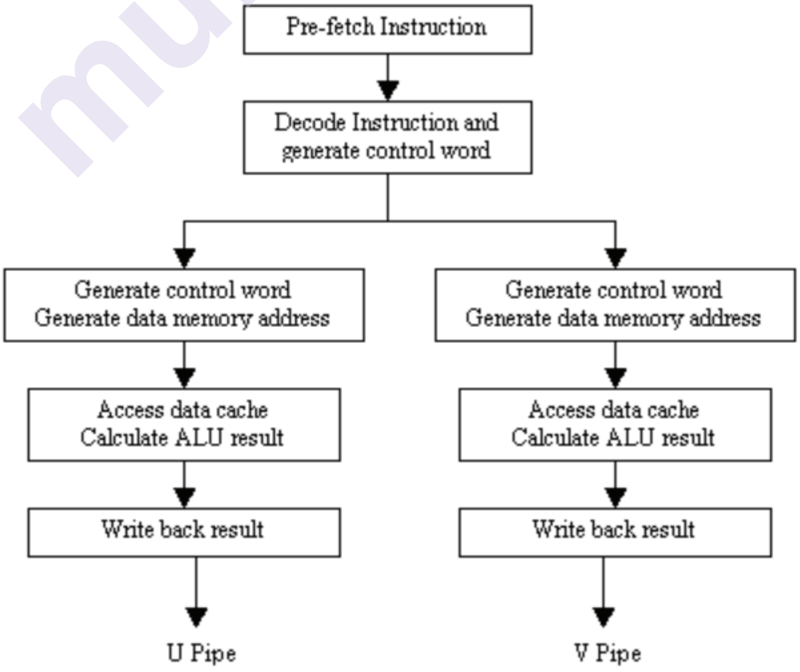

Pre-fetch Instruction
Decode Instruction and generate control word
Generate control word Generate data memory address
Generate control word Generate data memory address
Access data cache Calculate ALU result
Access data cache Calculate ALU result
Write back result
Write back result
U Pipe
V Pipe

Figure 1.3: Pentium Pipeline Stages

Floating Point Unit:

There are 8 general-purpose 80-bit Floating point registers. Floating point unit has 8 stages of pipelining. First five are similar to integer unit. Since the possibility of error is more in Floating Point unit (FPU) than in integer unit, additional error checking stage is there in FPU. The floating-point unit is shown as below:

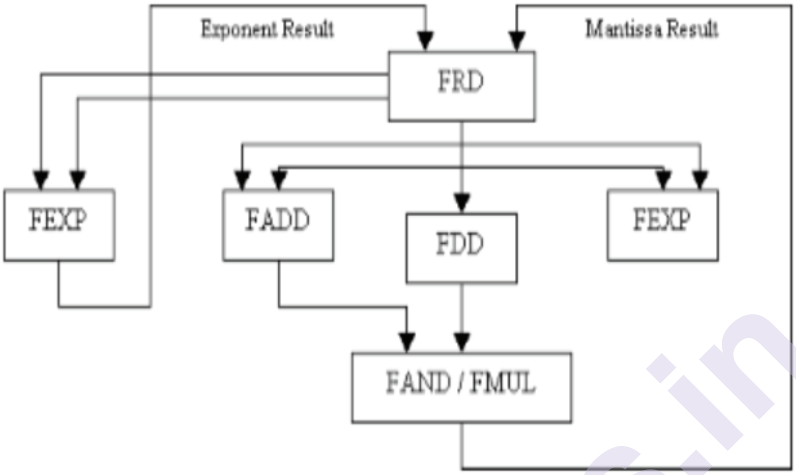


Figure 1.4: Floating Point Unit

where,

FRD - Floating Point Rounding

FDD - Floating Point Division

FADD - Floating Point Addition

FEXP - Floating Point Exponent

FAND - Floating Point And

FMUL - Floating Point Multiply

14.2 Special Pentium Registers

14.2.1 The Programming Model

The programming model of the 8086 through the Pentium II's considered to be **program visible** because its registers are used during application programming and are specified by the instructions. Other registers, detailed later in this chapter, are **program invisible** because they are not addressable directly during applications programming, but may be used indirectly during system programming. Only the 80286 and above contain the program-invisible registers used to control and operate the protected memory system.

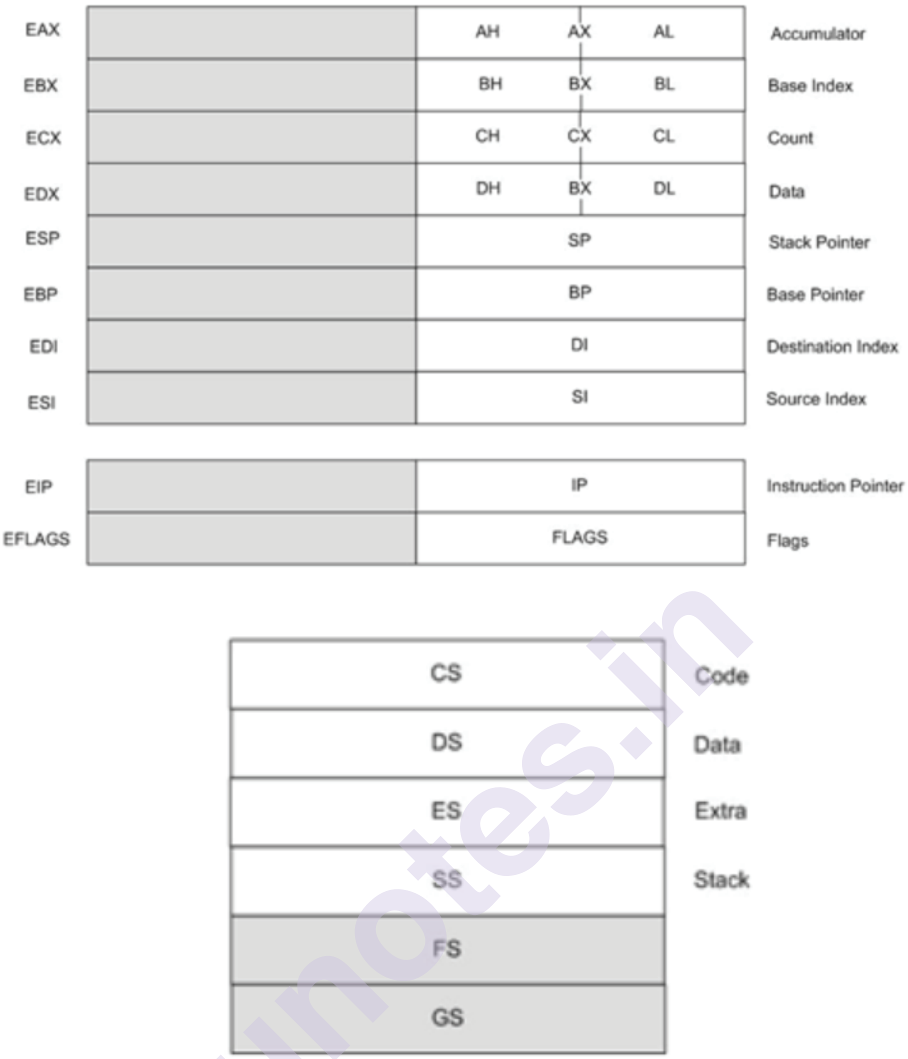


Figure 2.1: The Programming Model of Microprocessor

Figure 2.1 illustrates the programming model of the 8086 through the Pentium II microprocessor. The earlier 8086, 8088, and 80286 contain **16-bit** internal architectures, a subset of the registers. The 80386, 80486, Pentium, Pentium Pro, and Pentium II microprocessors contain full 32-bit internal architectures. The architectures of the earlier 8086 through the 80286 are fully forward-compatible to the 80386 through the Pentium II. The shaded areas in this illustration represent registers that are not found in the 8086, 8088, or 80286 microprocessors.

The programming model contains 8-, 16-, and 32-bit registers. The 8-bit registers are **AH**, **AL**, **BH**, **BL**, **CH**, **CL**, **DH**, and **DL** and are referred to when an instruction is formed using these two-letter designations. The 16-bit registers are **AX**, **BX**, **CX**, **DX**, **SP**, **BP**, **DI**, **SI**, **IP**, **FLAGS**, **CS**, **DS**, **ES**, **SS**, **FS**, and **GS**.

The **extended 32-bit registers are** **EAX**, **EBX**, **ECX**, **EDX**, **ESP**, **EBP**, **EDI**, **ESI**, **EIP**, and **EFLAGS**. These 32-bit extended registers, and 16-bit registers ES and GS are available only in the 80386 and above.

Some registers are general-purpose or multipurpose registers, while some have special purposes. The **multipurpose** registers include EAX, EBX, ECX, EDX, EBP, EDI, and ESI. These registers hold various data sizes (bytes, words, or doublewords) and are used for almost any purpose, as dictated by a program.

14.2.1.1 Multipurpose Registers

EAX (accumulator)

EAX is referenced as a 32-bit register (EAX), as a 16-bit register (AX), or as either of two 8-bit registers (AH and AL). Note that if an 8- or 16-bit register is addressed, only that portion of the 32-bit register changes without affecting the remaining bits. The accumulator is used for instructions such as multiplication, division, and some of the adjustment instructions. For these instructions, the accumulator has a special purpose, but is generally considered to be a multipurpose register. In the 80386 and above, the EAX register may also hold the offset address of a location in the memory system.

EBX (base index)

EBX is addressable as EBX, BX, BH, or BL. The BX register sometimes holds the offset address of a location in the memory system in all versions of the microprocessor. In the 80386 and above, EBX also can address memory data.

ECX (count)

ECX is a general-purpose register that also holds the count for various instructions. In the 80386 and above, the ECX register also can hold the offset address of memory data. Instructions that use a count are the repeated string instructions (REP/REPE/REPNE); and shift, rotate, and LOOP/LOOPD instructions. The shift and rotate instructions use CL as the count, the repeated string instructions use CX, and the LOOP/LOOPD instructions use either CX or ECX.

EDX (data)

EDX is a general-purpose register that holds a part of the result from a multiplication or part of the dividend before a division. In the 80386 and above, this register can also address memory data.

EBP (base pointer)

EBP points to a memory location in all versions of the microprocessor for memory data transfers. This register is addressed as either BP or EBP.

EDI (destination index)

EDI often addresses string destination data for the string instructions. It also functions as either a 32-bit (EDI) or 16-bit (DI) general-purpose register.

ESI (source index)

ESI is used as either ESI or SI. The source index register often addresses source string data for the string instructions. Like EDI, ESI also functions as a general-purpose register. As a 16-bit register, it is addressed as SI; as a 32-bit register, it is addressed as ESI.

14.2.1.2 Special-purpose Registers

The special-purpose registers include EIP, ESP, EFLAGS; and the segment registers CS, DS, ES, SS, FS, and GS.

EIP (instruction pointer)

EIP addresses the next instruction in a section of memory defined as a code segment. This register is IP (16 bits) when the microprocessor operates in the real mode and EIP (32 bits) when the 80386 and above operate in the protected mode. Note that the 8086, 8088, and 80286 do contain EIP, and only the 80286 and above operate in the protected mode. The instruction pointer, which points to the next instruction in a program, is used by the microprocessor to find the next sequential instruction in a program located within the code segment. The instruction pointer can be modified with a jump or a call instruction.

ESP (stack pointer)

ESP addresses an area of memory called the stack. The stack memory stores data through this pointer. This register is referred to as SP if used as a 16-hit register and ESP if referred to as a 32-bit register.

EFLAGS

EFLAGS indicate the condition of the microprocessor and control its operation. Figure 2-2 shows the flag registers of all versions of the microprocessor. Note that the flags are upward-compatible from the 8086/8088 to the Pentium II microprocessor. The 8086-80286 contain a FLAG register (16 bits) and the 80386 and above contain an EFLAG register (32-bit extended flag register).

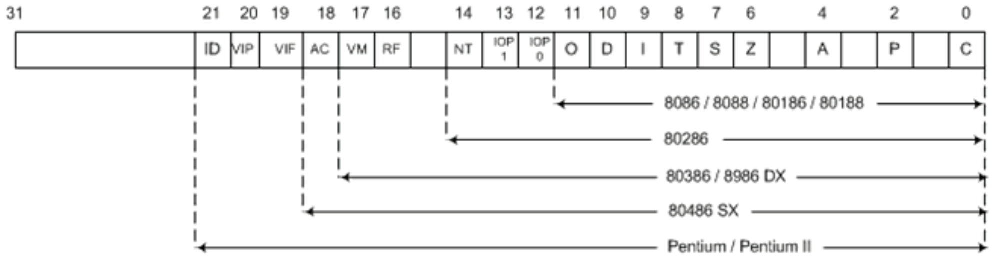


Figure 2.2: EFLAG Register

The rightmost five flag bits and the overflow flag change after many arithmetic and logic instructions execute. The flags never change for any data transfer or program control operation. Some of the flags are also used to control features found in the microprocessor. Following is a list of each flag bit, with a brief description of their function.

C (Carry)

Carry holds the carry after addition or the borrow after subtraction. The carry flag also indicates error conditions, as dictated by some programs and procedures. This is especially true of the DOS function calls.

P (Parity)

Parity is a logic 0 for odd parity and a logic 1 for even parity. Parity is a count of ones in a number expressed as even or odd.

If a number contains zero one bits, it has even parity. The parity flag finds little application in modern programming and was implemented in early Intel microprocessors for checking data in data communications environments. Today parity checking is often accomplished by the data communications equipment instead of the microprocessor.

A (Auxiliary Carry)

The auxiliary carry holds the carry (half-carry) after addition or the borrow after subtraction between bits positions 3 and 4 of the result. This highly specialized flag bit is tested by the DAA and DAS instructions to adjust the value of AL after a BCD addition or subtraction. Otherwise, the A flag bit is not used by the microprocessor or any other instructions.

Z (Zero)

The zero flag shows that the result of an arithmetic or logic operation is zero. If Z=1, the result is zero; if Z= 0, the result is not zero.

S (Sign)

The sign flag holds the arithmetic sign of the result after an arithmetic or logic instruction executes. If S=1, the sign bit (leftmost hit of a number) is set or negative; if S=0, the sign bit is cleared or positive.

T (Trap)

The trap flag enables trapping through an on-chip debugging feature. (A program is debugged to find an error or bug.) If the T flag is enabled (1), the microprocessor interrupts the flow of the program on conditions as indicated by the debug registers and control registers. lf the T flag is a logic 0, the trapping (debugging) feature is disabled.

I (Interrupt)

The interrupt flag controls the operation of the INTR (interrupt request) input pin. If I=1. the INTR pin is enabled: if I= 0, the INTR pin is disabled. The state of the I flag bit is controlled by the STI (set I flag) and CLI (clear I flag) instructions.

D (Direction)

The direction flag selects either the increment or decrement mode for the Dl and/or SI registers during string instructions. If D=1, the registers are automatically decremented: if D=1, the registers are automatically incremented. The D flag is set with the STD (set direction) and cleared with the CLD (clear direction) instructions.

0 (Overflow)

Overflows occurs when signed numbers are added or subtracted. An overflow indicates that the result has exceeded the capacity of the machine. For unsigned operations, the overflow flag is ignored.

IOPL (I/0 Privilege Level)

IOPL is used in protected mode operation to select the privilege level for I/O devices. If the current privilege level is higher or more trusted than the IOPL, I/O executes without hindrance. If the IOPL is lower than the current privilege level, an interrupt occurs, causing execution to suspend. Note that an IOPL of 00 is the highest or most trusted: if IOPL is 11, it is the lowest or least trusted.

NT (Nested Task)

The nested task flag indicates that the current task is nested within another task in protected mode operation. This line is set when the task is nested by software.

RF (Resume)

The resume flag is used with debugging to control the resumption of execution after the next instruction.

VM (Virtual Mode)

The VM flag bit selects virtual mode operation in a protected mode system. A virtual mode system allows multiple DOS memory partitions that are 1M byte in length to coexist in the memory system. Essentially, this allows the system program to execute multiple DOS programs.

AC (Alignment Check)

The alignment check flag bit activates if a word or douhleword is addressed on a non-word or non-douhleword boundary. Only the 80486SX microprocessor contains the alignment check hit that is primarily used by its companion numeric coprocessor, the 80487SX, for synchronization.

VIF (Virtual Interrupt Flag)

The VIF is a copy of the interrupt flag bit available to the Pentium-Pentium II microprocessors.

VIP (Virtual Interrupt Pending)

VIP provides information about a virtual mode interrupt for the Pentium—Pentium II microprocessors. This is used in multitasking environments to provide the operating system with virtual interrupt flags and interrupt pending information.

ID (Identification)

The ID flag indicates that the Pentium—Pentium II microprocessors support the CPUID instruction. The CPUID instruction provides the system with information about the Pentium microprocessor, such as its version number and manufacturer.

14.2.1.3 Segment Registers

Additional registers, called segment registers, generate memory addresses when combined with other registers in the microprocessor. There are either four or six segment registers in various versions of the microprocessor. A segment register functions differently in the real mode when compared to the protected mode operation of the microprocessor. Following is a list of each segment register, along with its function in the system:

CS (Code)

The code segment is a section of memory that holds the code (programs and procedures) used by the microprocessor. The code segment register defines the starting address of the section of memory holding code. In real mode operation, it defines the start of a 64K-byte section of memory; in protected mode, it selects a descriptor that describes the starting address and length of a section of memory holding code. The code segment is limited to 64K bytes in the 8088-80286, and 4G bytes in the 80386 and above when these microprocessors operate in the protected mode.

DS (Data)

The data segment is a section of memory that contains most data used by a program. Data are accessed in the data segment by an offset address or the contents of other registers that hold the offset address. As with the code segment and other segments, the length is limited to 64K bytes in the 8086-80286, and 4G bytes in the 80386 and above.

ES (Extra)

The extra segment is an additional data segment that is used by some of the string instructions to hold destination data.

SS (Stack)

The stack segment defines the area of memory used for the stack. The stack entry point is determined by the stack segment and stack pointer registers. The BP register also addresses data within the stack segment.

FS and GS

The FS and GS segments are supplemental segment registers available in the 80386, 80486, Pentium. and Pentium Pro microprocessors to allow two additional memory segments for access by programs.

14.3 Memory Management

14.3.1 Real Mode Memory Addressing

The 80286 and above operate in either the real or protected mode. Only the 8086 and 8088 operate exclusively in the real mode. **Real mode operation** allows the microprocessor to address only the first 1M byte of memory space-even if it is the Pentium II microprocessor. Note that the first 1 M byte of memory is called either the **real memory** or **conventional memory** system. The DOS operating system requires the microprocessor to operate in the real mode. Real mode operation allows application software written for the 8086/8088, which contain only 1 M byte of memory, to function in the 80286 and above without changing the software. The upward compatibility of software is partially responsible for the continuing success of the Intel family of microprocessors. In all cases, each of these microprocessors begins operation in the real mode by default whenever power is applied or the microprocessor is reset.

14.3.1.1 Segments and Offsets

A combination of a segment address and an offset address access a memory location in the real mode. All real mode memory addresses must consist of a segment address plus an offset address. The **segment address,** located within one of the segment registers, defines the beginning address of any 64K-byte memory segment. The **offset address** selects any location within the 64K byte memory segment. Segments in the real mode always have a length of 64K bytes. Figure 2-3 shows how the segment plus offset addressing scheme selects a memory location. This illustration shows a memory segment that begins at location 1 0000H and ends at location 1 FFFEH 64K bytes in length. It also shows how an offset address, sometimes called a displacement, of F000H selects location 1F000H in the memory system. Note that the offset or displacement is the distance above the start of the segment, as shown in Figure 3.1.

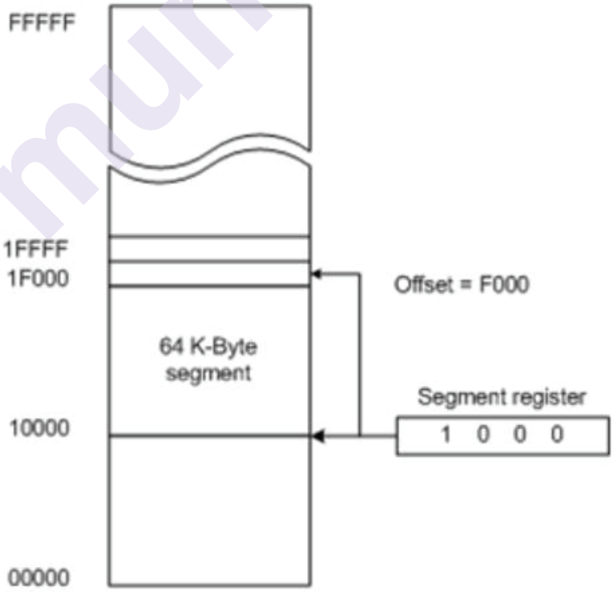


Figure 3.1: The real mode memory addressing scheme

The segment register in Figure3.1 contains a 1000H, yet it addresses a starting segment at location 10000H. In the real mode, each segment register is internally appended with a 0H on its rightmost end. This forms a 20-bit memory address, allowing it to access the start of a segment. The microprocessor must generate a 20-hit memory address to access a location within the first 1 M of memory. For example, when a segment register contains a 1200H, it addresses a 64K-byte memory segment beginning at location 12000H. Likewise, if a segment register contains a 1201H, it addresses a memory segment beginning at location 12010H. Because of the internally appended 0H, real mode segments can begin only at a 16-byte boundary in the memory system. This 16-byte boundary is often called a **paragraph.**

Because a real mode segment of memory is 64K in length, once the beginning address is known, **the ending address** is found by adding FFFFH.

The offset address, which is a part of the address, is added to the start of the segment to address a memory location within the memory segment. For example, if the segment address is 1000H and the offset address is 2000H, the microprocessor addresses memory location 12000H. The offset address is always added to the starting address of the segment to locate the data. The segment and offset address is sometimes written as 1000:2000 for a segment address of 1000H with an offset of 2000H.

In the 80286 (with special external circuitry), and the 80386 through the Pentium II, an extra 64K minus 16 bytes of memory is addressable when the segment address is FFFFH and the HIMEM.SYS driver is installed in the system. This area of memory (0FFFF0H-10FFEFH) is referred to as **high memory.**

Some addressing modes combine more than one register and an offset value to form an offset address. When this occurs, the sum of these values may exceed FFFFH. For example, the address accessed in a segment whose segment address is 4000H, and whose offset address is specified as the sum of F000H plus 3000H, will access memory location 42000H instead of location 52000H. When the F000H and 3000H are added, they form a 16-bit **(modulo 16)** sum of 2000H used as the offset address; not 12000H, the true sum. Note that the carry of 1 (F000H + 3000H=12000H) is dropped for this addition to form the offset address of 2000H. This means that the address is generated as 4000:2000 or 42000H.

14.3.1.2 Default Segment and Offset Registers

The microprocessor has a set of rules that apply to segments whenever memory is addressed. These rules, which apply in the real and protected mode, define the segment register and offset register combination. For example, the code segment register is always used with the instruction pointer to address the next instruction in a program. This combination is **CS:IP** or **CS:EIP**, depending upon the microprocessor's mode of operation. The **code segment** register defines the start of the code segment and the **instruction pointer** locates the next instruction within the code segment. This combination (CS:IP or CS:EIP) locates the next instruction executed by the microprocessor.

Another of the default combinations is the **stack.** Stack data are referenced through the stack segment at the memory location addressed by either the stack pointer (SP/ESP) or the base pointer (BP/EBP). These combinations are referred to as SS:SP (SS:ESP) or SS:BP (SS:EBP). Note that in real mode, only the rightmost 16 bits of the extended register address a location within the memory segment. In the 80386—Pentium II, never place a number larger than FFFFH into an offset register if the microprocessor is operated in the real mode. This causes the system to halt and indicate an addressing error.

Table 3.1: Default 16-bit segment and offset combinations

Segment	**Offset**	**Special Purpose**
CS	IP	Instruction Address
SS	SP or BP	Stack Address
DS	BX, DI, SI, an 8-bit number, or a 16-bit number	Data Address
ES	DI for string instructions	String Destination Address

Table 3.2: Default 32-bit segment and offset combinations

Segment	**Offset**	**Special Purpose**
CS	EIP	Instruction Address
SS	ESP or EBP	Stack Address
DS	EBX, EDI, ESI, EAX, ECX, EDX, on 8-bit number, or an 16-bit number	Data Address
ES	EDI for string instructions	String Destination Address
FS	No Default	General Address
GS	No Default	General Address

Other defaults are shown in Table 3.1 for addressing memory using any Intel microprocessor with 16-bit registers. Table 3.2 shows the defaults assumed in the 80386 and above when using 32-bit registers. Note that the 80386 and above have a far greater selection of segment offset address combinations than do the 8086 through the 80286 microprocessors.

The 8086-80286 microprocessors allow four memory segments and the 80386 and above allow six memory segments. Figure 3.2 shows a system that contains four memory segments. Note that a memory segment can touch or even overlap if 64K bytes of memory are not required for a segment. Think of segments as windows that can be moved over any area of memory to access data or code. Also note that a program can have more than four or six segments but can only access four or six segments at a time.

Suppose that an application program requires 1000H bytes of memory for its code, 190H bytes of memory for its data, and 200H bytes of memory for its stack. This application does not require an extra segment. When this program is placed in the memory system by DOS, it is loaded in the TPA at the first available area of memory above the drivers and other TPA programs. This area is indicated by a **free pointer** that is maintained by DOS. Program loading is handled automatically by the **program loader** located within DOS. Figure 3.3 shows how this application is stored in the memory system. The segments show an overlap because the amount of data in them does not require 64K bytes of memory. The side view of the segments clearly shows the overlap. It also shows how segments can be moved over any area of memory by changing the segment starting address. Fortunately, the DOS program loader calculates and assigns segment starting addresses.

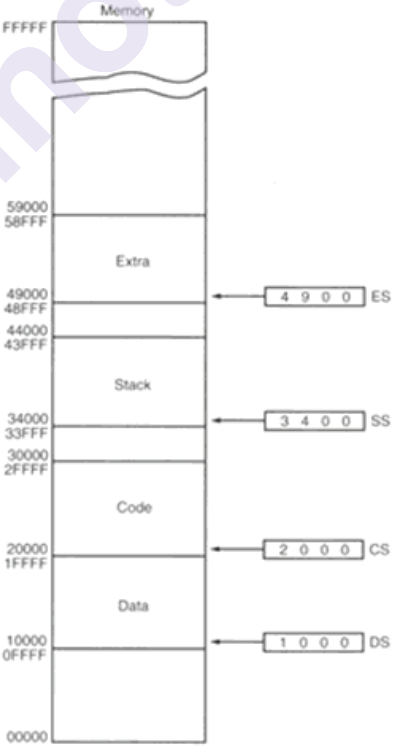


Figure 3.2: A memory system showing the placement of four memory segments

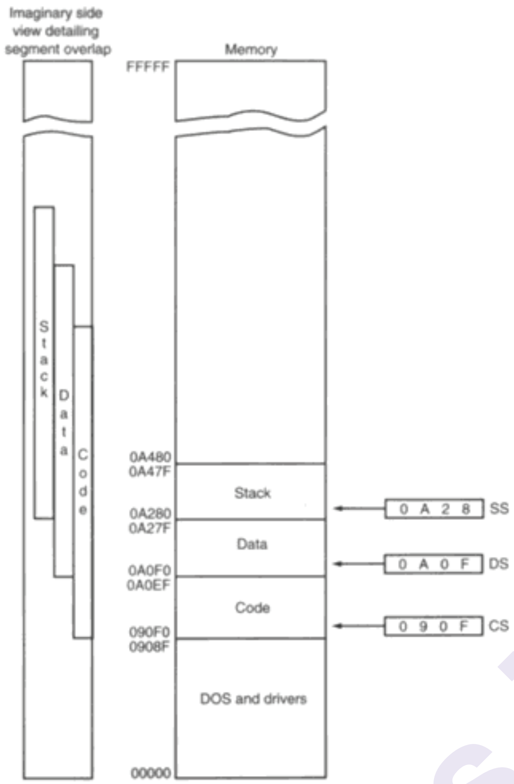


Figure 3.3: An Application program containing code, data and stack segment loaded into DOS system memory

14.3.2 Segment and Offset Addressing Scheme Allows Relocation

The segment and offset addressing scheme seems unduly complicated. It is complicated, but it also affords an advantage to the system. This complicated scheme of segment plus offset addressing allows programs to be relocated in the memory system. It also allows programs written to function in the real mode to operate in a protected mode system. A relocatable program is one that can be placed into any area of memory and executed without change. **Relocatable data** are data that can be placed in any area of memory and used without any change to the program. The segment and offset addressing scheme allows both programs and data to be relocated without changing a thing in a program or data. This is ideal for use in a general-purpose computer system in which not all machines contain the same memory areas. The personal computer memory structure is different from machine to machine, requiring relocatable software and data.

Because memory is addressed within a segment by an offset address, the memory segment can be moved to any place in the memory system without changing any of the offset addresses. This is accomplished by moving the entire program, as a block, to a new area and then changing only the contents of the segment registers. If an instruction is 4 bytes above the start of the segment, its offset address is 4. If the entire program is moved to a new area of memory, this offset address of 4 still points to 4 bytes above the start of the segment. Only the contents of the segment register must be changed to address the program in the new area of memory. Without this feature, a program would have to be extensively rewritten or altered before it is moved. This would require additional time or many versions of a program for the many different configurations of computer systems.

14.3.2 Memory Paging

The memory paging mechanism located within the 80386 and above allows any physical memory location to be assigned to any linear address. The **linear address** is defined as the address generated by a program. With the memory paging unit, the linear address is invisibly translated into any physical address, which allows an application written to function at a specific address to be relocated through the paging mechanism. It also allows memory to be placed into areas where no memory exists. An example is the upper memory blocks provided by EMM386.EXE.

The EMM386.EXE program reassigns extended memory, in 4K blocks, to the system memory between the video BIOS and the system BIOS ROMS for upper memory blocks. Without the paging mechanism, the use of this area of memory is impossible.

14.3.2.1 Paging Registers

The paging unit is controlled by the contents of the microprocessor's control registers. See Figure 2-11 for the contents of control registers CR0 through CR3. Note that these registers are only available to the 80386 through the Pentium microprocessors. Beginning with the Pentium, an additional control register labeled CR4 controls extensions to the basic architecture provided in the Pentium and above microprocessors. One of these features is a 4M-byte page that is enabled by setting bit position 4, or CR4.

The registers important to the paging unit are CR0 and CR3. The leftmost bit (PG) position of CR0 selects paging when placed at a logic 1 level. If the PG bit is cleared (0), the linear address generated by the program becomes the physical address used to access memory. If the PG bit is set (1), the linear address is converted to a physical address through the paging mechanism. *The paging mechanism functions in both the real and protected modes.*

CR3 contains the page directory base address, and the PCD and PWT bits. The PCD and PWT bits control the operation of the PCD and PWT pins on the microprocessor. If PCD is set (1), the PCD pin becomes a logic one during bus cycles that are not pages. This allows the external hardware to control the level 2 cache memory. (Note that the level 2 cache memory is an external high-speed memory that functions as a buffer between the microprocessor and the main DRAM memory system.) The PWT bit also appears on the PWT pin, during bus cycles that are not pages, to control the write-through cache in the system. The page directory base address locates the page directory for the page translation unit. Note that this address locates the page directory at any 4K boundary in the memory system because it is appended internally with a 000H. The page directory contains 1024 directory entries of 4 bytes each.

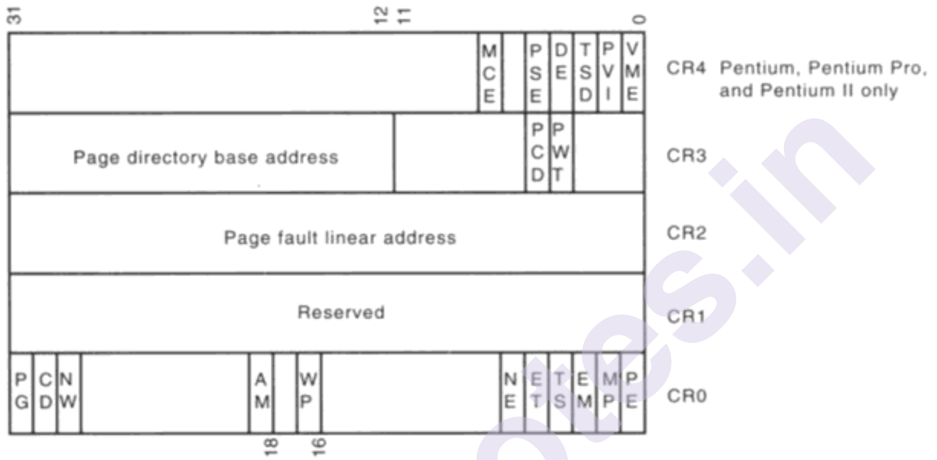

31
12
11
0
MCE
PSE
DE
TSD
PVI
VME
CR4 Pentium, Pentium Pro, and Pentium II only
Page directory base address
PCD
PWT
CR3
Page fault linear address
CR2
Reserved
CR1
PG
CD
NW
AM
WP
NE
ET
TS
EM
MP
PE
CR0
18
16

Figure 3.4: The control register structure of the microprocessor

Each page directory entry addresses a page table that contains 1024 entries.

The linear address, as it is generated by the software, is broken into three sections that are used to access the **page directory entry, page table entry, and page offset address.** Figure 3.4 shows the linear address and its makeup for paging. Notice how the leftmost 10 bits address an entry in the page directory. For linear address 00000000H—003FFFFFH, the first entry of the page directory is accessed. Each page directory entry represents or repages a 4M-byte section of the memory system. The contents of the page directory select a page table that is indexed by the next 10 bits of the linear address (bit positions 12-21). This means that address 00000000H— 00000FFFH selects page directory entry 0 and page table entry 0. Notice this is a 4K-byte address range. The offset part of the linear address (bit positions 0-11) next selects a byte in the 4K-byte memory page. In Figure 2-12, if the page table 0 entry contains address 00100000H, then the physical address is 00100000H-00100FFFH for linear address 00000000H-00000FFFH. This means that when the program accesses a location between 00000000H and 00000FFFH, the microprocessor physically addresses location 00100000H—00100FFFH.

Because the act of re-paging a 4K-byte section of memory requires access to the page directory and a page table, which are both located in memory, Intel has incorporated a cache called the TLB **(translation look-aside buffer).** In the 80486 micro-processor, the cache holds the 32 most recent page translation addresses. This means that the last 32-page table translations are stored in the TLB, so if the same area of memory is accessed, the address is already present in the TLB, and access to the page directory and page tables is not required. This speeds program execution. If a translation is not in the TLB, the page directory and page table must be accessed, which requires additional execution time. The Pentium, Pentium Pro, and Pentium II contain separate TLBs for each of their instruction and data caches.

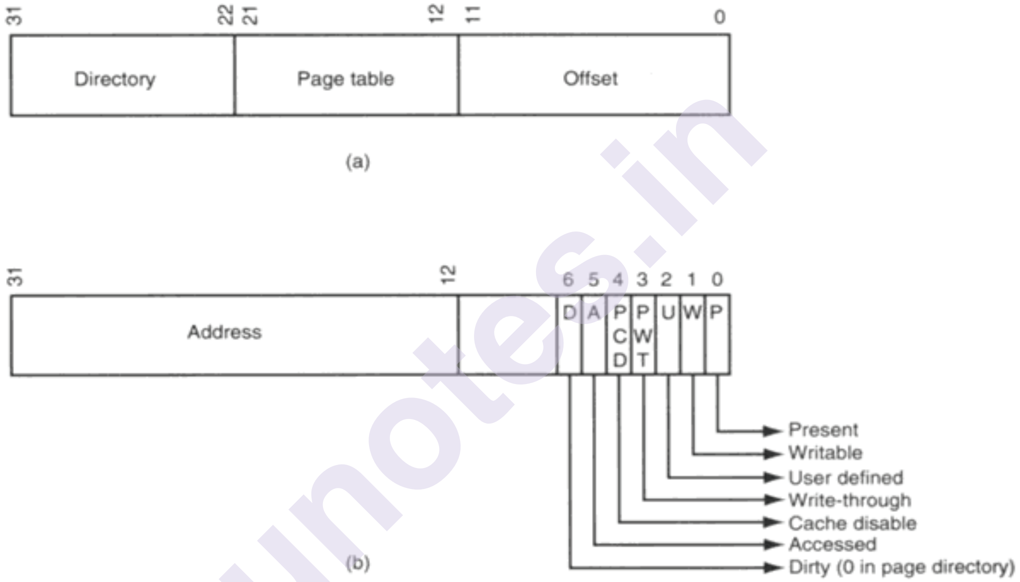


Figure 3.5: The format for the linear address (a) and a page directory or page table entry (b)

14.3.3.2. The Page Directory and Page Table

Figure 3.6 shows the page directory, a few page tables, and some memory pages. There is only one page directory in the system. The page directory contains 1024 double word addresses that locate up to 1024 page tables. The page directory and each page table are 4K bytes in length. If the entire 4G byte of memory is paged, the system must allocate 4K bytes of memory for the page directory, and 4K times 1024 or 4M bytes for the 1024 page tables. This represents a considerable investment in memory resources.

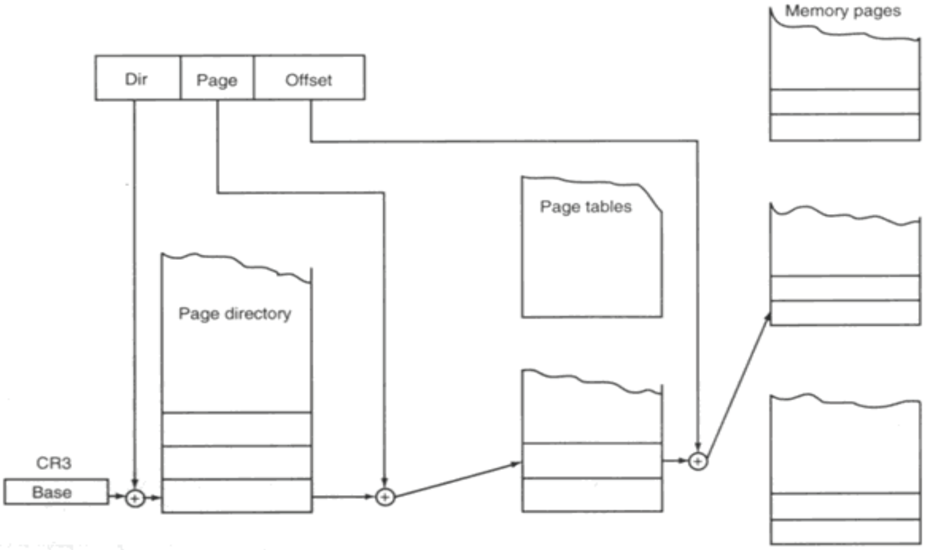


Figure 3.6: The paging mechanism the 80386, 80486, Pentium, Pentium Pro, and Pentium II microprocessor

The DOS system and EMM386.EXE use page tables to redefine the area of memory between locations C8000H—EFFFFH as upper memory blocks. It does this by repaging extended memory to back-fill this part of the conventional memory system to allow DOS access to additional memory. Suppose that the EMM386.EXE program allows access to 16M bytes of extended and conventional memory through paging and locations C8000H—EFFFFH must be repaged to locations 110000—138000H, with all other areas of memory paged to their normal locations. Such a scheme is depicted in Figure 3.7

Here, the page directory contains four entries. Recall that each entry in the page directory corresponds to 4M bytes of physical memory. The system also contains four page tables with 1024 entries each. Recall that each entry in the page table repages 4K bytes of physical memory. This scheme requires a total of 16K of memory for the four page tables and 16 bytes of memory for the page directory.

As with DOS, the Windows program also repages the memory system. At present, Windows version 3.11 supports paging for only l6M bytes of memory because of the amount of memory required to store the page tables. On the Pentium and Pentium Pro microprocessors, pages can be either 4K bytes in length or 4M bytes in length. Although no software currently supports the 4M-byte pages, as the Pentium II and more advanced versions pervade the personal computer, operating systems of the future will undoubtedly begin to support 4M-byte memory pages.

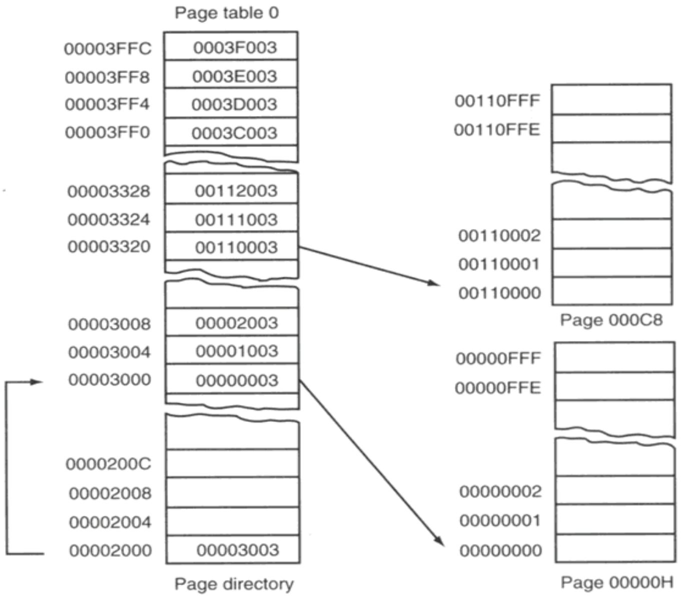


Figure 3.7: The page directory, page table 0, and two memory pages

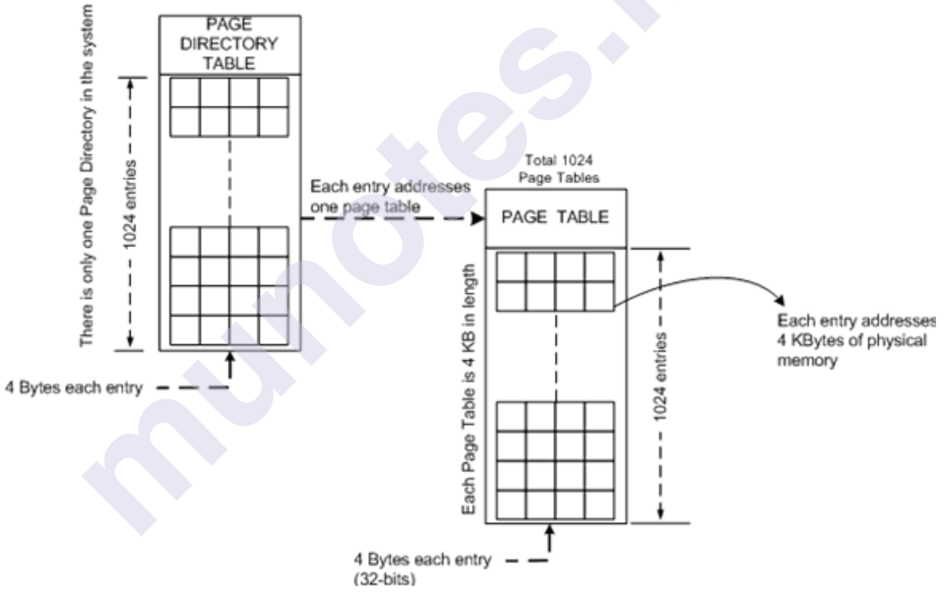

PAGE DIRECTORY TABLE
There is only one Page Directory in the system
1024 entries
4 Bytes each entry
Each entry addresses one page table
Total 1024 Page Tables
PAGE TABLE
Each Page Table is 4 KB in length
1024 entries
Each entry addresses 4 KBytes of physical memory
4 Bytes each entry (32-bits)

Figure 3.8: The page directory, and page table

14.4 Pentium Instructions

14.4.1 Instruction Set

These instructions set have exactly 2 operands. If there are 2 operands, then one of them will be required to use register mode, and the other will have no restrictions on its addressing mode.

There are most often ways of specifying the same instruction for 8-, 16-, or 32-bit operands.

I left out the 16-bit ones to reduce presentation of the instruction set.

Note that on a 32-bit machine, with newly written code, the 16-bit form will never be used.

Meanings of the operand specifications:

reg - register mode operand, 32-bit register
reg8 - register mode operand, 8-bit register
r/m - general addressing mode, 32-bit
r/m8 - general addressing mode, 8-bit
immed - 32-bit immediate is in the instruction
immed8 - 8-bit immediate is in the instruction
m - symbol (label) in the instruction is the effective address

14.4.2 Data Movement

```
mov   reg, r/m              ; copy data
r/m, reg
reg, immed
r/m, immed
movsx reg, r/m8              ; sign extend and copy data
movzx reg, r/m8              ; zero extend and copy data
lea   reg, m                 ; get effective address
(A newer instruction, so its format is much restricted
over the other ones.)
```

Examples:

```
mov EAX, 23  ; places 32-bit 2's complement immediate 23
             ; into register EAX

movsx ECX, AL  ; sign extends the 8-bit quantity in register
               ; AL to 32 bits, and places it in ECX

mov [esp], -1  ; places value -1 into memory, address given
               ; by contents of esp

lea EBX, loop_top ; put the address assigned (by the assembler)
                  ; to label loop_top into register EBX
```

14.4.3 Integer Arithmetic

```
add   reg, r/m                 ; two's complement addition
      r/m, reg
      reg, immed
      r/m, immed
      inc   reg                ; add 1 to operand
      r/m
      sub   reg, r/m           ; two's complement subtraction
      r/m, reg
      reg, immed
      r/m, immed
      dec   reg               ; subtract 1 from operand
      r/m
      neg   r/m                ; get additive inverse of operand
      mul   eax, r/m           ; unsigned multiplication
                               ; edx||eax <- eax * r/m
      imul   r/m               ; 2's comp. multiplication
                               ; edx||eax <- eax * r/m
      reg, r/m                 ; reg <- reg * r/m
      reg, immed               ; reg <- reg * immed
      div   r/m                ; unsigned division
                               ; does edx||eax / r/m
                               ; eax <- quotient
                               ; edx <- remainder
      idiv   r/m               ; 2's complement division
                               ; does edx||eax / r/m
                               ; eax <- quotient
                               ; edx <- remainder
      cmp   reg, r/m           ; sets EFLAGS based on
      r/m, immed               ; second operand - first operand
      r/m8, immed8
      r/m, immed8              ; sign extends immed8 before subtract
```

Examples:

```
       neg [eax + 4]           ; takes doubleword at address eax+4
                               ; and finds its additive inverse, then places
                               ; the additive inverse back at that address
                               ; the instruction should probably be
                               ; neg  dword ptr [eax + 4]
       inc ecx                 ; adds one to contents of register ecx, and
                               ; result goes back to ecx
```

14.4.4 Logical

```
 not    r/m               ; logical not
and     reg, r/m          ; logical and
        reg8, r/m8
        r/m, reg
        r/m8, reg8
        r/m, immed
        r/m8, immed8
or      reg, r/m          ; logical or
        reg8, r/m8
        r/m, reg
        r/m8, reg8
        r/m, immed
        r/m8, immed8
xor     reg, r/m          ; logical exclusive or
        reg8, r/m8
        r/m, reg
        r/m8, reg8
        r/m, immed
        r/m8, immed8
test    r/m, reg          ; logical and to set EFLAGS
        r/m8, reg8
        r/m, immed
        r/m8, immed8
```

Examples:

```
and edx, 00330000h        ; logical and of contents of register
                          ; edx (bitwise) with 0x00330000,
                          ; result goes back to edx
```

14.4.5 Floating Point Arithmetic

Since the newer architectures have room for floating point hardware on chip, Intel defined a simple-to-implement extension to the architecture to do floating point arithmetic. In their usual zeal, they have included MANY instructions to do floating point operations.

The mechanism is simple. A set of 8 registers are organized and maintained (by hardware) as a stack of floating-point values. ST refers to the stack top. ST(1) refers to the register within the stack that is next to ST. ST and ST(0) are synonyms.

There are separate instructions to test and compare the values of floating-point variables.

```
finit                    ; initialize the FPU
fld      m32            ; load floating point value
         m64
         ST(i)
fldz                     ; load floating point value 0.0
fst      m32            ; store floating point value
         m64
         ST(i)
fstp     m32             ; store floating point value
         m64             ;   and pop ST
         ST(i)
fadd     m32             ; floating point addition
         m64
         ST, ST(i)
         ST(i), ST
faddp    ST(i), ST       ; floating point addition
                         ; and pop ST
ETC. (see p.201-202)
```

14.4.6 I/O

The only instructions which actually allow the reading and writing of I/O devices are priviledged. The OS must handle these things. But, in writing programs that do something

useful, we need input and output. Therefore, there are some simple macros defined to help us do I/O.

These are used just like instructions.

```
put_ch  r/m              ; print character in the least significant
                         ;   byte of 32-bit operand
get_ch  r/m              ; character will be in AL
put_str m                ; print null terminated string given
                         ; by label m
```

14.4.7 Control Instructions

These are the same control instructions that all started with the character 'b' in SASM.

```
jmp   m          ; unconditional jump
jg    m          ; jump if greater than 0
jge   m          ; jump if greater than or equal to 0
jl    m          ; jump if less than 0
jle   m          ; jump if less than or equal to 0
```

14.5 PENTIUM PRO MICROPROCESSORS

14.5.1 Modes

The Pentium and Pentium Pro processor has three operating modes:

1. **Real-address mode:** This mode lets the processor to address "real" memory address. It can address up to 1Mbytes of memory (20-bit of address). It can also be called "unprotected" mode since operating system (such as DOS) code runs in the same mode as the user applications. Pentium and Prentium Pro processors have this mode to be compatible with early Intel processors such as 8086. The processor is set to this mode following by a power-up or a reset and can be switched to protected mode using a single instruction.

2. **Protected mode:** This is the preferred mode for a modern operating system. It allows applications to use virtual memory addressing and supports multiple programming environment and protections.

3. **System management mode:** This mode is designed for fast state snapshot and resumption. It is useful for power management.

   There is also a virtual-8086 mode that allows the processor to execute 8086 code software in the protected, multi-tasking environment.

14.5.2 Register Set

There are three types of registers: general-purpose data registers, segment registers, and status and control registers. The following figure shows these registers:

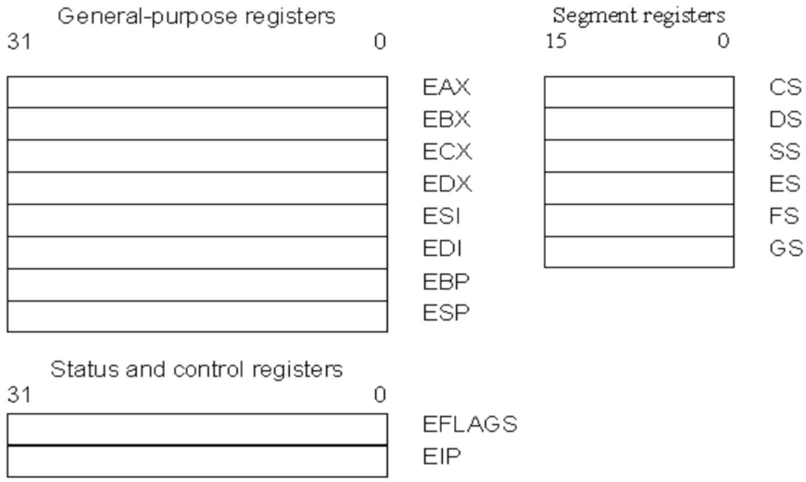


Figure 5.1: Register Set

General-purpose Registers

The eight 32-bit general-purpose data registers are used to hold operands for logical and arithmetic operations, operands for address calculations and memory pointers. The following shows what they are used for:

EAX→Accumulator for operands and results data.

EBX→Pointer to data in the DS segment.

ECX→Counter for string and loop operations.

EDX→I/O pointer.

ESI→Pointer to data in the segment pointed to by the DS register; source pointer for string operations.

EDI→Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations.

ESP→Stack pointer (in the SS segment).

EBP→Pointer to data on the stack (in the SS segment).

The following figure shows the lower 16 bits of the general-purpose registers can be used with the names AX, BX, CX, DX, BP, SP, SI, and DI (the names for the corresponding 32-bit ones have a prefix "E" for "extended"). Each of the lower two bytes of the EAX, EBX, ECX, and EDX registers can be referenced by the names AH, BH, CH, and DH (high bytes) and AL, BL, CL, and DL (low bytes).

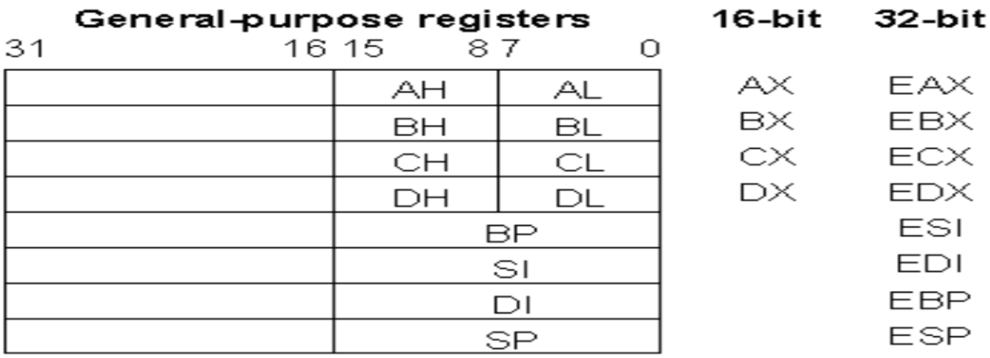


Figure 5.2: Registers with lower bytes

Segment Registers

There are six segment registers that hold 16-bit segment selectors. A segment selector is a special pointer that identifies a segment in memory. The six segment registers are:

CS: Code Segment Register

SS: Stack Segment Register

DS, ES, FS, GS: Data Segment Registers

Four data segment registers provide programs with flexible and efficient ways to access data.

Modern operating system and applications use the (unsegmented) memory model - all the segment registers are loaded with the same segment selector so that all memory references a program makes are to a single linear-address space.

When writing application code, you generally create segment selectors with assembler directives and symbols. The assembler and/or linker then creates the actual segment selectors associated with these directives and symbols. If you are writing system code, you may need to create segment selectors directly.

EFLAGS Register

The 32-bit EFLAGS register contains a group of status flags, a control flag, and a group of system flags. The following shows the function of EFLAGS register bits:

Table 5.1: EFLAGS Register

Function	EFLAG Register bit or bits
ID Flag (ID)	21 (system)
Virtual Interrupt Pending (VIP)	20 (system)
Virtual Interrupt Flag (VIF)	19 (system)

Function	EFLAG Register bit or bits
Alignment check (AC)	18 (system)
Virtual 8086 Mode (VM)	17 (system)
Resume Flag (RF)	16 (system)
Nested Task (NT)	14 (system)
I/O Privilege Level (IOPL)	13 to 12 (system)
Overflow Flag (OF)	11 (system)
Direction Flag (DF)	10 (system)
Interrupt Enable Flag (IF)	9 (system)
Trap Flag (TF)	8 (system)
Sign Flag (SF)	7 (status)
Zero Flag (ZF)	6 (status)
Auxiliary Carry Flag (AF)	4 (status)
Parity Flag (PF)	2 (status)
Carry Flag (CF)	0 (status)

Bits 1, 3, 5, 15, and 22 through 31 of this register are reserved.

EIP Register (Instruction Pointer)

The EIP register (or instruction pointer) can also be called "program counter." It contains the offset in the current code segment for the next instruction to be executed. It is advanced from one instruction boundary to the next in straight-line code or it is moved ahead or backwards by a number of instructions when executing JMP, Jcc, CALL, RET, and IRET instructions. The EIP cannot be accessed directly by software; it is controlled implicitly by control-transfer instructions (such as JMP, Jcc, CALL, and RET), interrupts, and exceptions. The EIP register can be loaded indirectly by modifying the value of a return instruction pointer on the procedure stack and executing a return instruction (RET or IRET).

Note that the value of the EIP may not match with the current instruction because of instruction prefetching. The only way to read the EIP is to execute a CALL instruction and then read the value of the return instruction pointer from the procedure stack.

14.5.3 Addressing

Bit and Byte Order

Pentium and Pentium-Pro processors use "little endian" as their byte order. This means that the bytes of a word are numbered starting from the least significant byte and that the least significant bit starts of a word starts in the least significant byte.

Data Types

The Pentium/Pentium Pro provide four data types: a byte (8 bits), a word (16 bits), a doubleword (32 bits), and a quadword (64 bits). Note that a doubleword is equivalent to "long" in Gnu assembler.

Memory Addressing

One can use either flat memory model or segmented memory mode. With the flat memory model, memory appears to a program as a single, continuous address space, called a linear address space. Code (a programs instructions), data, and the procedure stack are all contained in this address space. The linear address space is byte addressable, with addresses running contiguously from 0 to 2^{32-1}.

With the segmented memory mode, memory appears to a program as a group of independent address spaces called segments. When using this model, code, data, and stacks are typically contained in separate segments. To address a byte in a segment, a program must issue a logical address, which consists of a segment selector and an offset. (A logical address is often referred to as a far pointer.) The segment selector identifies the segment to be accessed and the offset identifies a byte in the address space of the segment. The programs running on a Pentium Pro

processor can address up to 16,383 segments of different sizes and types. Internally, all the segments that are defined for a system are mapped into the processors linear address space. So, the processor translates each logical address into a linear address to access a memory location. This translation is transparent to the application program.

14.5.4 Processor Reset

A cold boot or a warm boot can reset the CPU. A cold boot is powering up a system whereas a warm boot means that when three keys CTRL-ALT-DEL are all pressed together, the keyboard BIOS will set a special flag and resets the CPU.

Upon reset, the processor sets itself to real mode with interrupts disabled and key registers set to a known state. For example, the state of the EFLAGS register is 00000002H and the memory is unchanged. Thus, the memory will contain garbage upon a cold boot. The CPU will jump to the BIOS (Basic Input Output Services) to load the bootstrap loader program from the diskette drive or the hard disk and begins execution of the loader. The BIOS loads the bootstrap loader into the fixed address 0:7C00 and jumps to the starting address.

14.5.5 Assembly Programming

It often takes a while to master the techniques to program in assembly language for a particular machine. On the other hand, it should not take much time to assembly programming for Pentium or Pentium Pro processors if you are familiar with another processor.

This section assumes that you are already familiar with Gnu assembly syntax. The simplest way to learn assembly programming is to compile a simple C program into its assembly source code as a template. For example, gcc -S -c foo.c will compile foo.c its assembly source foo.s. The source code will tell you common opcodes, directives and addressing syntax.

The goal of this section is to answer some frequently encountered questions and provide pointers to related documents.

14.5.5.1 Memory operands

Pentium and Pentium Pro processors use segmented memory architecture. It means that the memory locations are referenced by means of a segment selector and an offset:

- The segment selector specifies the segment containing the operand, and
- The offset (the number of bytes from the beginning of the segment to the first byte of the operand) specifies the linear or effective address of the operand.

  The segment selector can be specified either implicitly or explicitly. The most common method of specifying a segment selector is to load it in a segment register and then allow the processor to select the register implicitly, depending on the type of operation being performed. The processor automatically chooses a segment according to the following rules:

- Code segment register CS for instruction fetches
- Stack segment register SS for stack pushes and pops as well as references using ESP or EBP as a base register
- Data segment register DS for all data references except when relative to stack or string destination
- Data segment register ES for the destinations of string instructions

The offset part of the memory address can be specified either directly as a static value (called a *displacement*) or through an address computation made up of one or more of the following components:

- Displacement→An 8-, 16-, or 32-bit value.
- Base→The value in a general-purpose register.
- Index→The value in a general-purpose register except EBP.
- Scale Factor→A value of 2, 4, or 8 that is multiplied by the index value.

An effective address is computed by:

Offset = Base + (Index ´ Scale) + displacement

The offset which results from adding these components is called an *effective address* of the selected segment. Each of these components can have either a positive or negative (2's complement) value, with the exception of the scaling factor.

14.5.5.2 Instruction Syntax

There are two conventions about their syntax and representations: Intel and AT&T. Most documents including those at http://www.x86.org use the Intel convention, whereas the Gnu assembler uses the AT&T convention. The main differences are:

Table 5.2: Difference between Intel and AT&T

	Intel	**AT&T (Gnu Syntax)**
Immediate operands	Undelimited e.g.: push 4 mov ebx, d00ah	Preceded by "$" e.g.: push $4 movl $0xd00a, %eax
Register operands	Undelimited e.g.: eax	Preceded by "%" e.g.: %eax
Argument order (e.g. adds the address of C variable "foo" to register EAX)	Dest, source [, source2] e.g.: add eax, _foo	Source, [source,] dest e.g.: addl $_foo, %eax
Single-size operands	Implicit with register name, **byte ptr**, **word ptr**, or **dword ptr** e.g.: mov al, foo	opcode{b,w,l} e.g.: movb foo, %al
Address a C variable "foo"	[_foo]	_foo

	Intel	AT&T (Gnu Syntax)
Address memory pointed by a register (e.g. EAX)	[eax]	(%eax)
Address a variable offset by a value in the register	[eax + _foo]	_foo(%eax)
Address a value in an array "foo" of 32-bit integers	[eax*4+foo]	_foo(,%eax,4)
Equivalent to C code *(p+1)	If EAX holds the value of p, then [eax+1]	1(%eax)

In addition, with the AT&T syntax, the name for a long JUMP is ljmp and long CALL is lcall.

14.5.5.3 Assembler Directives

The GNU assembler directives are machine independent, so your knowledge about assembly programming applies. All directive names begin with a period "." and the rest are letters in lower case. Here are some examples of commonly used directives:

.ascii "string" defines an ASCII string "string"

.byte 10, 13, 0 defines three bytes

.word 0x0456, 0x1234 defines two words

.long 0x001234, 0x12345 defines two long words

.equ STACK_SEGMENT, 0x9000 sets symbol STACK_SEGMENT the
value 0x9000

.globl symbol makes "symbol" global (useful for defining global labels and procedure names)

.code16 tells the assembler to insert the appropriate override prefixes so the code will run in real mode.

When using directives to define a string, bytes or a word, you often want to make sure that they are aligned to 32-bit long word by padding additional bytes.

14.5.5.4 Inline Assembly

The most basic format of inline assembly code into your the assembly code generated by the gcc compiler is to use asm("assembly-instruction");

where assembly-instruction will be inlined into where the asm statement is. This is a very convenient way to inline assembly instructions that require no registers. For example, you can use

asm("cli");
to clear interrupts and
asm("sti");
to enable interrupts.

The general format to write inline assembly code in C is:
asm("statements": output_regs: input_regs: used_regs);

where statements are the assembly instructions. If there are more than one instruction, you can use "\n\t" to separate them to make them look pretty. "input_regs" tells gcc compiler which C variables move to which registers. For example, if you would like to load variable "foo" into register EAX and "bar" into register ECX, you would say

: "a" (foo), "c" (bar)
gcc uses single letters to represent all registers:

Table 5.3: Register Representations

Single Letters	Reigsters
a	eax
b	ebx
c	ecx
d	edx
S	esi
D	edi
I	constant value (0 to 31)
q	allocate a register from EAX, EBX, ECX, EDX
r	allocate a register from EAX, EBX, ECX, EDX, ESI, EDI

Note that you cannot specify register AH or AL this way. You need to get to EAX first and then go from there.

"output_regs" provides output registers. A convenient way to do this is to let gcc compiler to pick the registers for you. You need to say "=q" or "=r" to let gcc compiler pick registers for you. You can refer to the first allocated register with "%0", second with "%1", and so on, in the assembly instructions. If you refer to the registers in the input register list, you simply say "0" or "1" without the "%" prefix.

"used_regs" lists the registers that are used (or clobbered) in the assembly code.

To understand exactly how to do this, please try to use gcc to compile a piece of C code containing the following inline assembly:

```
asm ("leal (%1,%1,4), %0"
    : "=r" (x_times_5)
    : "r" (x) );
```

and

```
asm ("leal (%0,%0,4), %0"
   : "=r" (x)
   : "0" (x) );
```

Also, to avoid the gcc compiler's optimizer to remove the assembly code, you can put in keyword volitale to ensure your inline. Here are some macro code examples:

```
#define disable() __asm__ __volatile__ ("cli");

#define enable() __asm__ __volatile__ ("sti");
```

to disable and enable interrupts.

14.6 Special Pentium Pro Features

Silent features of Pentium Pro Architecture:

- 64-bit data bus
- 8 bytes of data information can be transferred to and from memory in a single bus cycle
- Supports burst read and burst write back cycles
- Supports pipelining
- Instruction cache

Core specifications

- Pentium Pro
- L1 cache: 8, 8 KB (data, instructions)
- L2 cache: 256, 512 KB (one die) or 1024 KB (two 512 KB dies) in a multi-chip module clocked at CPU-speed

- Socket: Socket 8
- Front side bus: 60 and 66 MHz
- VCore: 3.1–3.3 V
- Fabrication: 0.50 µm or 0.35 BiCMOS[18]
- Clockrate: 150, 166, 180, 200 MHz, (capable of 233 MHz on some motherboards)
- First release: November 1995

14.7 Summary

In this chapter we have studied about processor architecture. The various registers used in processors and their uses. We have seen the page tables for memory accessing methods. We have seen various types of instructions sets used in assembly language programming. We have also seen the features of Pentium Pro Microprocessors.

14.8 Review Your Learnings

1. Are you able to explain Pentium Processors?
2. Are you able to recognize the Pentium Instructions Set?
3. Are you able to understand the features of Pentium and Pentium pro Processor?
4. Do you feel capable to explain the architecture of Pentium Processor?
5. Will you be able to explain the page tables and memory management concept in processor?

14.9 Sample Questions:

1. Explain Pentium Processors and its acrhitectures?
2. Explain the any two Pentium Instructions Set with examples?
3. Are you able to understand the features of Pentium and Pentium pro Processor?
4. Explain the architecture of Pentium Processor.
5. Explain the page tables and memory management concept in processor.
6. Explain control instructions set.
7. Explain various addressing modes used in processors for accessing memories.
8. Explain Real Mode Architecture in processors.

14.10 References for further reading

- Pentium Pro Family Developers Manual, Volume 2: Programmer's Reference Manual, Intel Corporation, 1996
- Pentium Pro Family Developers Manual, Volume 3: Operating System Writer's Manual, Intel Corporation, 1996
- http://www.x86.org/intel.doc/intelDocs.html
- https://www.byclb.com/TR/Tutorials/microprocessors/ch2_1.htm#:~:text=The%2016%2Dbit%20registers%20are,in%20the%2080386%20and%20above
- https://eun.github.io/Intel-Pentium-Instruction-Set-Reference/data/index.html

❖❖❖

UNIT 5

15

CORE 2 AND LATER MICROPROCESSORS

Unit Structure

15.0 Objectives

15.1 Introduction

15.2 Pentium II Software Changes

15.3 Pentium IV

15.4 Core 2, i3, i5 and i7

15.4.1 Core i2

15.4.2 The Microarchitectures of Nehalem

15.4.3 Core i3, i5 and i7

15.5 Summary

15.6 Review Your Learnings:

15.7 Sample Questions:

15.8 References for further reading

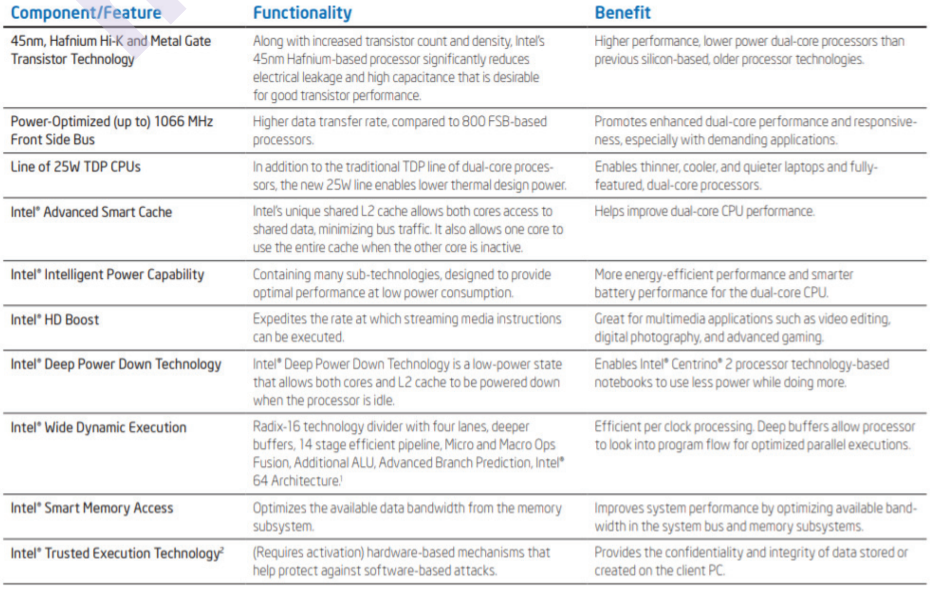
15.0 Objectives

1. Explain Microprocessor properties
2. Analyse performance between i2, i3, i5 and i7 processors
3. Explain multicore processors
4. Differentiate between properties of i2, i3, i5 and i7 processors
5. Explain concept of Turbo Boost, Pipelining etc

15.1 Introduction

Performance analysis is a more efficient method of improving processor performance. We need to learn various already invented and newly emerging processor architectures. With the evolution of Intel processor architecture over time, most customer (buyers) of Intel architecture do not really have time to test and analyse the architecture before they purchase it for their various day to day use. In a nutshell, people have not been able to analyse the differences in the architectures before purchase. Testing performance of computer system is very necessary because it helps consumers decide what type and configurations of products to purchase for a particular nature of computing job. However, the performance is strongly dependent on several factors which include the system architecture, processor microarchitecture, operating systems, type of compiler, and program implementation etc. Many processor manufacturers including Intel has performance analysis tools which can be used to determine the performance of their architecture. Intel Corporation produces different processors with different numbers of cores for different nature of jobs, however, it is the important for users of processor machines to acquire the right processor specifications that would efficiently process target applications based on the workloads characteristics of the application program. For instance, some specification of machine works better on graphics while others perform best on computation. With the evolution of Intel processor architectures over time, testing performance is necessary. The aim of this study is to measure the performance of different cores using different applications (both Single and Multithreaded). The objectives are 1) compare architecture performance on applications (Single and Multithreaded), 2) measure performance counters on representative processors and, 3) show methods for exploring processor architectures. One of the goals of this work is to highlight the advantages of each feature in a system and to study how the hardware makes use of CPU resources.

Table 1.1 Functionality and benefit comparison between various Intel Processors



Component/Feature	Functionality	Benefit
45nm, Hafnium Hi-K and Metal Gate Transistor Technology	Along with increased transistor count and density, Intel's 45nm Hafnium-based processor significantly reduces electrical leakage and high capacitance that is desirable for good transistor performance.	Higher performance, lower power dual-core processors than previous silicon-based, older processor technologies.
Power-Optimized (up to) 1066 MHz Front Side Bus	Higher data transfer rate, compared to 800 FSB-based processors.	Promotes enhanced dual-core performance and responsiveness, especially with demanding applications.
Line of 25W TDP CPUs	In addition to the traditional TDP line of dual-core processors, the new 25W line enables lower thermal design power.	Enables thinner, cooler, and quieter laptops and fully-featured, dual-core processors.
Intel® Advanced Smart Cache	Intel's unique shared L2 cache allows both cores access to shared data, minimizing bus traffic. It also allows one core to use the entire cache when the other core is inactive.	Helps improve dual-core CPU performance.
Intel® Intelligent Power Capability	Containing many sub-technologies, designed to provide optimal performance at low power consumption.	More energy-efficient performance and smarter battery performance for the dual-core CPU.
Intel® HD Boost	Expedites the rate at which streaming media instructions can be executed.	Great for multimedia applications such as video editing, digital photography, and advanced gaming.
Intel® Deep Power Down Technology	Intel® Deep Power Down Technology is a low-power state that allows both cores and L2 cache to be powered down when the processor is idle.	Enables Intel® Centrino® 2 processor technology-based notebooks to use less power while doing more.
Intel® Wide Dynamic Execution	Radix-16 technology divider with four lanes, deeper buffers, 14 stage efficient pipeline, Micro and Macro Ops Fusion, Additional ALU, Advanced Branch Prediction, Intel® 64 Architecture.[1]	Efficient per clock processing. Deep buffers allow processor to look into program flow for optimized parallel executions.
Intel® Smart Memory Access	Optimizes the available data bandwidth from the memory subsystem.	Improves system performance by optimizing available bandwidth in the system bus and memory subsystems.
Intel® Trusted Execution Technology[2]	(Requires activation) hardware-based mechanisms that help protect against software-based attacks.	Provides the confidentiality and integrity of data stored or created on the client PC.

15.2 Pentium II Software Changes

Pentium II

The Pentium II made a number of subtle changes to the Pentium Pro's design and one big honkin' shift. It re-added the segment register cache previous x86 CPUs had used but the Pentium Pro hadn't to improve 16-bit performance, doubled the L1 cache size to 32K while splitting the L1 into instructions and data caches, widened the execution core by adding MMX support, and, of course, moved from a socket configuration to Intel's Slot 1. The Pentium Pro used an onboard L2 cache that was connected to the primary CPU by a dedicated bus, but the cache itself only ran at half clock. The Pentium Pro's cache, in contrast, had run at full CPU clock. This design was a huge success for Intel overall -- most of the company’s last x86 competitors were on their last legs by this time.

To trace the history of Intel CPU cores is to trace the history of various epochs in the evolution of CPU performance. In the 1980s and 1990s, clock speed improvements and architectural enhancements went hand in hand. From 2005 forward, it was the era of multi-core chips and higher efficiency parts. Since 2011, Intel has focused on improving the performance of its low power CPUs more than other capabilities. This focus has paid real dividends — laptops today have far better battery life and overall performance than they did a decade ago.

Unlike previous Pentium and Pentium Pro processors, the Pentium II CPU was packaged in a slot-based module rather than a CPU socket. The processor and associated components...

Max. CPU clock rate: 233 MHz to 450 MHz

Min. feature size: 0.35 µm to 0.18 µm

FSB speeds: 66 MHz to 100 MHz

Socket(s): Slot 1; MMC-1; MMC-2; Mini-Cartridge; PPGA-B615 (µPGA1)

Intel improved 16-bit code execution performance on the Pentium II, an area in which the Pentium Pro was at a notable handicap, by adding segment register caches. Most consumer software of the day was still using at least some 16-bit code, because of a variety of factors. The issues with partial registers was also addressed by adding an internal flag to skip pipeline flushes whenever possible. To compensate for the slower L2 cache, the Pentium II featured 32 KB of L1 cache, double that of the Pentium Pro, as well as 4 write buffers (vs. 2 on the Pentium Pro); these can also be used by either pipeline, instead of each one being fixed to one pipeline. The Pentium II was also the first P6-based CPU to implement the Intel MMX integer SIMD instruction set which had already been introduced on the Pentium MMX.

The Pentium II was basically a more consumer-oriented version of the Pentium Pro. It was cheaper to manufacture because of the separate, slower L2 cache memory. The improved 16-bit performance and MMX support made it a better choice for consumer-level operating systems, such as Windows 9x, and multimedia applications. The slower and cheaper L2 cache's performance penalty was mitigated by the doubled L1 cache and architectural improvements for legacy code. General processor performance was increased while costs were cut.

Pentium II Software Updates

Pentium II processor system bus agents can also be configured with some additional software

Configuration options. These options can be changed by writing to a power-on configuration

Register which all bus agents must implement. These options should be changed only after

Considering synchronization between multiple Pentium II processor system bus agents.

Pentium II processor system bus agents have the following configuration options:

- Output tristate {Hardware}
- Execution of the processor’s built-in self test (BIST) {Hardware}
- Data bus error-checking policy: enabled or disabled {Software}
- Response signal error-checking policy: parity disabled or parity enabled {Software}
- AERR# driving policy: enabled or disabled {Software}
- AERR# observation policy: enabled or disabled {Hardware}
- BERR# driving policy for initiator bus errors: enabled or disabled {Software}
- BERR# driving policy for target bus errors: enabled or disabled {Software}
- BERR# driving policy for initiator internal errors: enabled or disabled {Software}
- BINIT# error-driving policy: enabled or disabled {Software}
- BINIT# error-observation policy: enabled or disabled {Hardware}

- In-order Queue depth: 1 or 8 {Hardware}
- Power-on reset vector: 1M-16 or 4G-16 {Hardware}
- FRC mode: enabled or disabled {Hardware}
- APIC cluster ID: 0 or 1 {Hardware}
- APIC mode: enabled or disabled {Software}
- Symmetric agent arbitration ID: 0, 1, 2, or 3 {Hardware}
- Clock frequencies and ratios {Hardware}

15.3 Pentium IV

The term "Pentium Processor" refers to a family of microprocessors that share a common architecture and instruction set. The first Pentium processors were introduced in 1993. It runs ata clock frequency of either 60 or 66 MHz and has 3.1 million transistors. Some of the features of Pentium Architectures are listed below:

- Complex Instruction Set Computer (CISC) architecture with Reduced Instruction Set Computer (RISC) performance.
- 64-bit Bus
- Upward code compatibility
- Pentium Processor uses Superscalar Architecture and hence can issue multiple instructions per cycle.
- Multiple Instructions Issue (MII) capability
- Pentium Processor executes instructions in five stages. This staging or pipelining allows the processor to overlap multiple instructions so that it takes less time to execute two instructions in a row.
- The Pentium Processor fetches the branch target instruction before it executes the branch instructions.
- The Pentium processors have two separate 8 KB caches on chip, one for instruction and one for data. It allows the Pentium processor to fetch data and instructions from cache simultaneously.
- When data is modified, only the data in cache is changed. Memory data is changed only when Pentium Processor replaces modified data in cache with a different set of data.

- The Pentium Processor has been optimized to run critical instructions in fewer clock cycle than 80486 processor.

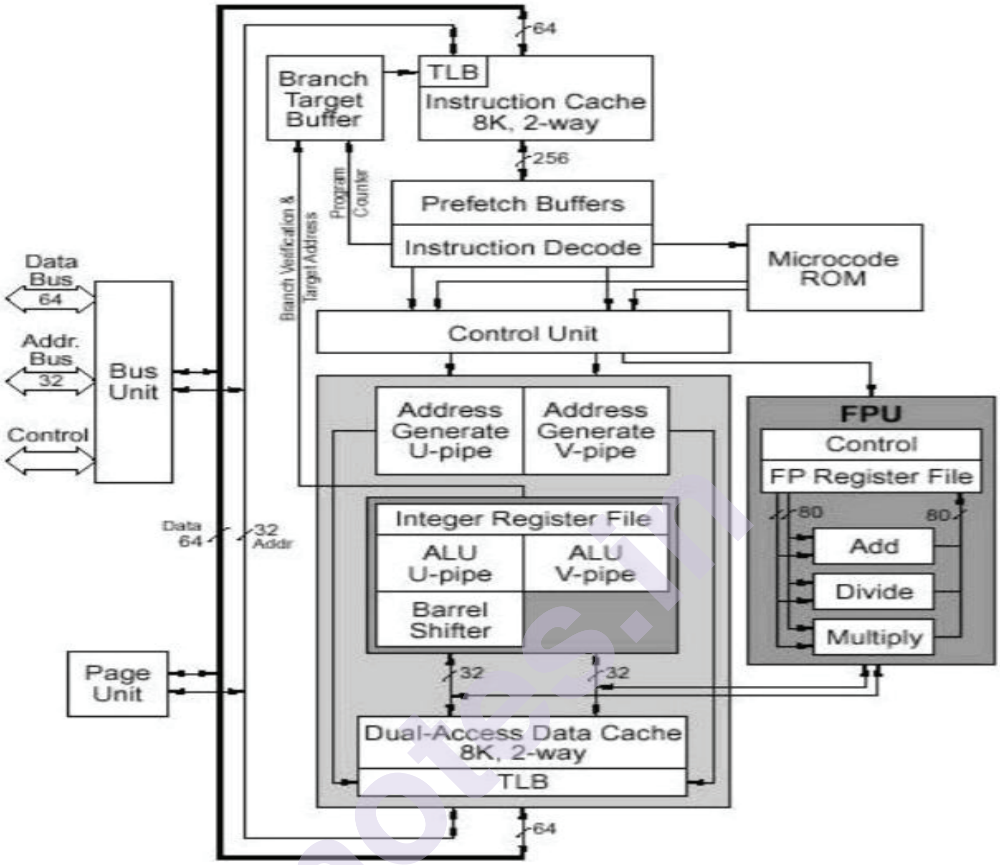


Figure 3.1: The Pentium Architecture

15.4 Core 2, i3, i5 and i7

15.4.1 Core i2

The Intel Core 2 Duo processor belongs to the Intel's mobile core family. It is implemented by using two Intel's Core architecture on a single die. The design of Intel Core 2 Duo is chosen to maximize performance and minimize power consumption. It emphasizes mainly on cache efficiency and does not stress on the clock frequency for high power efficiency. Although clocking at a slower rate than most of its competitors, shorter stages and wider issuing pipeline compensates the performance with higher IPC's. In addition, the Core 2 Duo processor has more ALU units. Core 2 Duo employs Intel's Advanced Smart Cache which is a shared L2 cache to increase the effective on-chip cache capacity. Upon a miss from the core's L1 cache, the shared L2 and the L1 of the other core are looked up in parallel before sending the request to the memory. The cache block located in the other L1 cache can be fetched without off-chip traffic. Both memory controller and FSB are still located off-chip. The off-chip memory controller can adapt the new DRAM technology with the cost of longer memory access latency. Intel Advanced Smart Cache provides a peak transfer rate of 96 GB/sec (at 3 GHz frequency).

The microarchitectures of Intel Core and Intel Core 2 are shown below in Figure 4.1 and Figure 4.2.

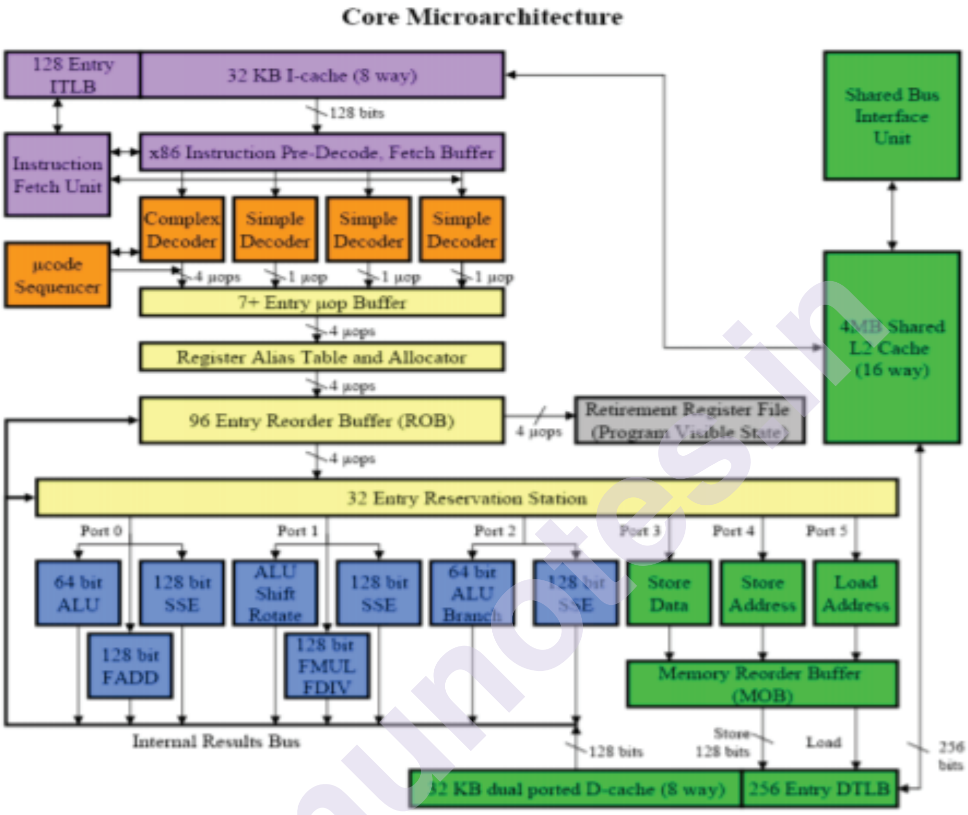


Figure 4.1: Intel Core Microarchitecture

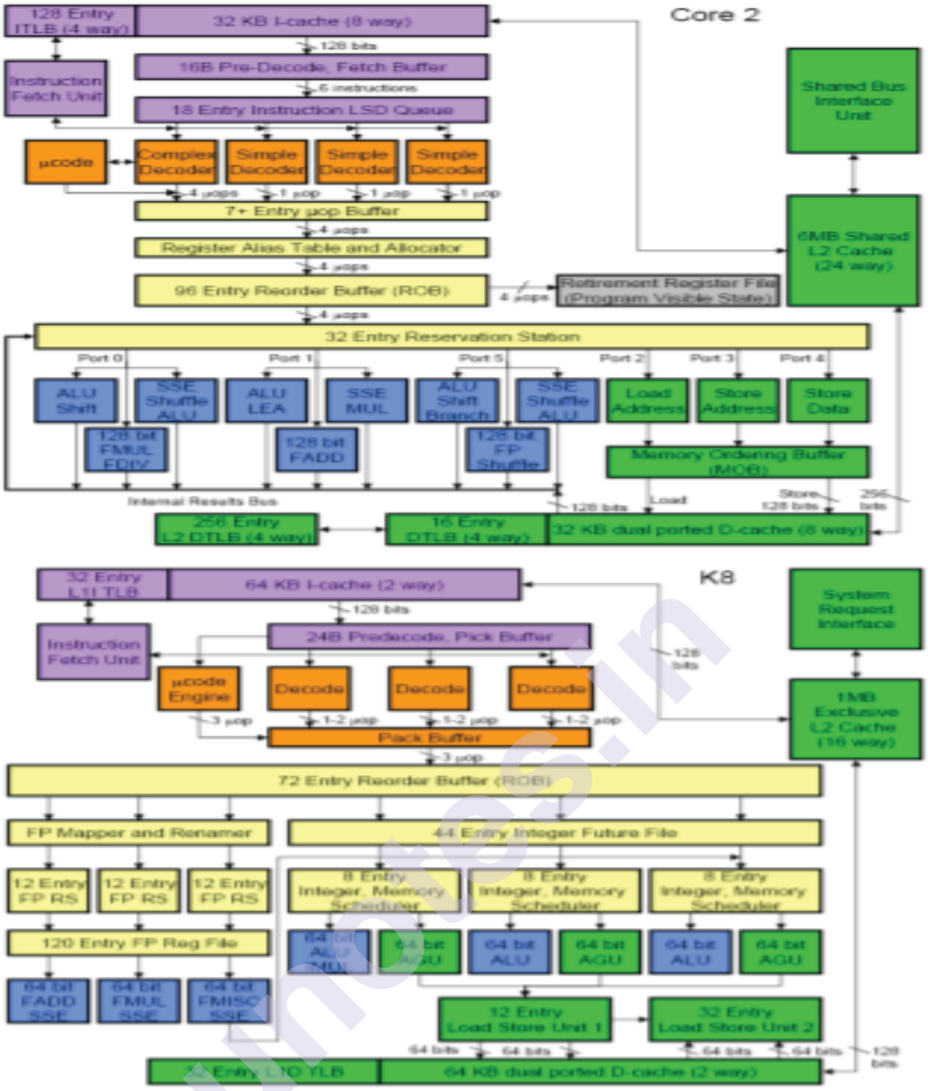


Figure 4.2: Intel Core 2 Microarchitecture

14.4.2 The Microarchitectures of Nehalem

The Microarchitectures of Nehalem Nehalem architecture is more modular than the Core architecture which makes it much more flexible and customizable to the application. The architecture is shown in Figure 4.3. The architecture really only consists of a few basic building blocks. The main blocks are a microprocessor core (with its own L2 cache), a shared L3 cache, a Quick Path Interconnect (QPI) bus controller, an integrated memory controller (IMC), and graphics core. With this flexible architecture, the blocks can be configured to meet what the market demands. For example, the Bloomfield model, which is intended for a performance desktop application, has four cores, an L3 cache, one memory controller, and one QPI bus controller. Another significant improvement in the Nehalem microarchitecture involves branch prediction. For the Core architecture, Intel designed what they call a “Loop Stream Detector,” which detects loops in code execution and saves the instructions in a special buffer so they do not need to be continually fetched from cache. This increased branch prediction success for loops in the code and improved performance. Intel engineers took the concept even further with the Nehalem architecture by placing the Loop Stream Detector after the decode stage eliminating the instruction decode from a loop iteration and saving CPU cycles.

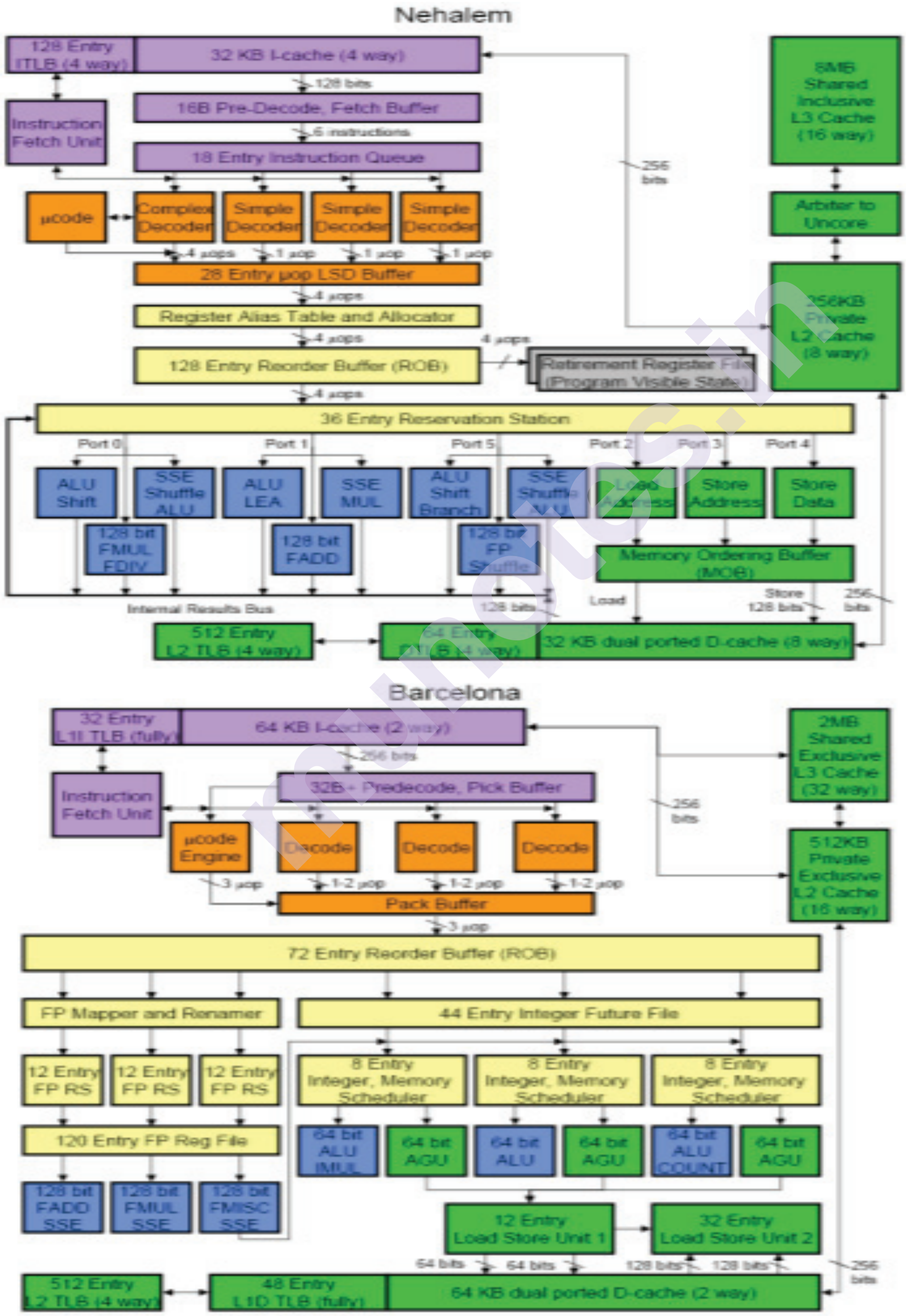
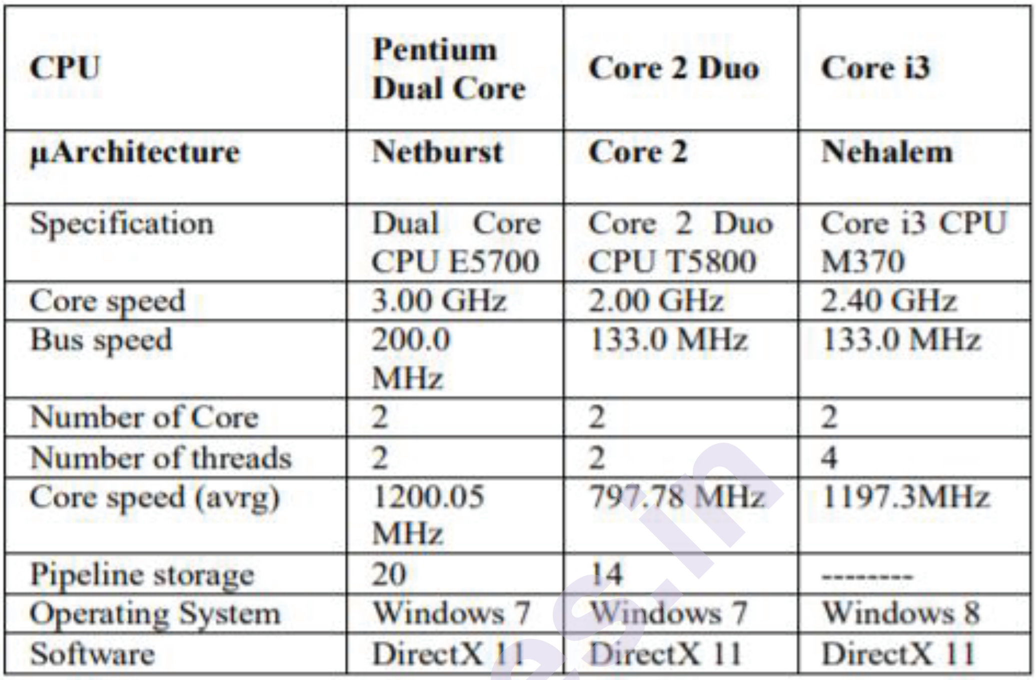


Figure 4.3: Nehalem Microarchitecture

The comparison between configurations of system architectures of Pentium Dual Core, Core i2 Duo and Core i3 is mentioned below in Table 4.1.

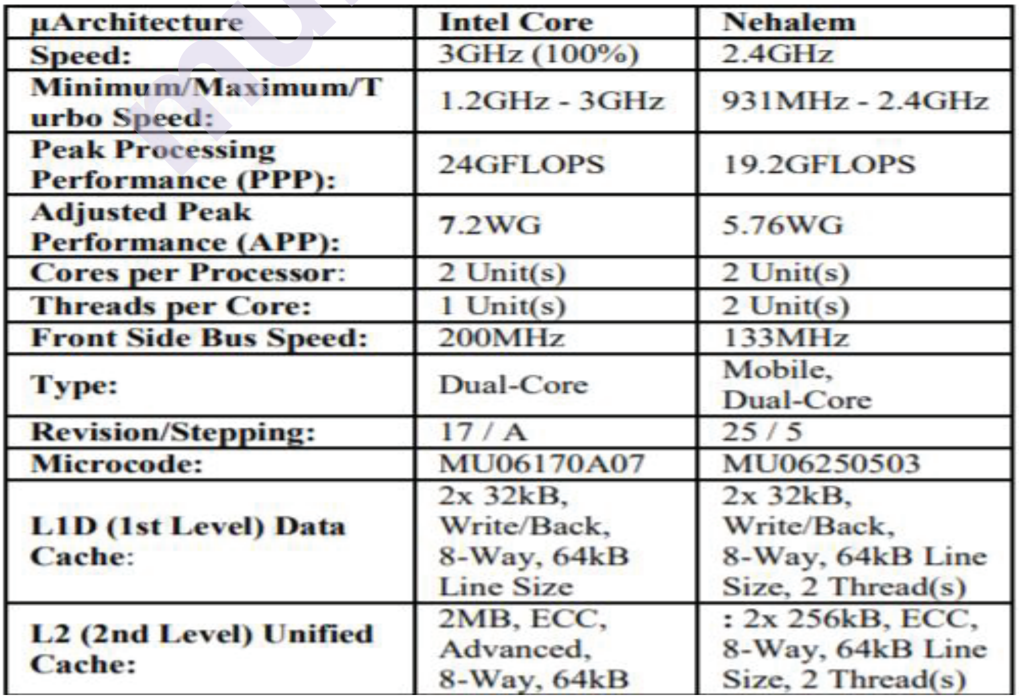
Table 4.1 Configuration of System Architecture



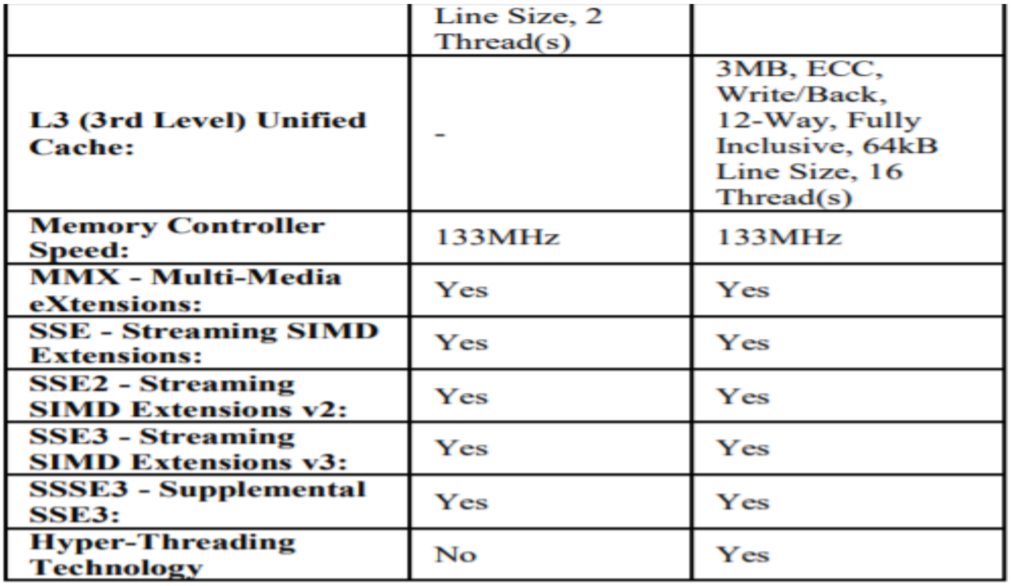
CPU	Pentium Dual Core	Core 2 Duo	Core i3
µArchitecture	**Netburst**	**Core 2**	**Nehalem**
Specification	Dual Core CPU E5700	Core 2 Duo CPU T5800	Core i3 CPU M370
Core speed	3.00 GHz	2.00 GHz	2.40 GHz
Bus speed	200.0 MHz	133.0 MHz	133.0 MHz
Number of Core	2	2	2
Number of threads	2	2	4
Core speed (avrg)	1200.05 MHz	797.78 MHz	1197.3MHz
Pipeline storage	20	14	--------
Operating System	Windows 7	Windows 7	Windows 8
Software	DirectX 11	DirectX 11	DirectX 11

The comparisons between Intel Core and Nehalem processors are mentioned below in Table 4.2.

Table 4.2: Processors Microarchitecture Features



µArchitecture	Intel Core	Nehalem
Speed:	3GHz (100%)	2.4GHz
Minimum/Maximum/Turbo Speed:	1.2GHz - 3GHz	931MHz - 2.4GHz
Peak Processing Performance (PPP):	24GFLOPS	19.2GFLOPS
Adjusted Peak Performance (APP):	7.2WG	5.76WG
Cores per Processor:	2 Unit(s)	2 Unit(s)
Threads per Core:	1 Unit(s)	2 Unit(s)
Front Side Bus Speed:	200MHz	133MHz
Type:	Dual-Core	Mobile, Dual-Core
Revision/Stepping:	17 / A	25 / 5
Microcode:	MU06170A07	MU06250503
L1D (1st Level) Data Cache:	2x 32kB, Write/Back, 8-Way, 64kB Line Size	2x 32kB, Write/Back, 8-Way, 64kB Line Size, 2 Thread(s)
L2 (2nd Level) Unified Cache:	2MB, ECC, Advanced, 8-Way, 64kB	: 2x 256kB, ECC, 8-Way, 64kB Line Size, 2 Thread(s)



	Line Size, 2 Thread(s)	
L3 (3rd Level) Unified Cache:	-	3MB, ECC, Write/Back, 12-Way, Fully Inclusive, 64kB Line Size, 16 Thread(s)
Memory Controller Speed:	133MHz	133MHz
MMX - Multi-Media eXtensions:	Yes	Yes
SSE - Streaming SIMD Extensions:	Yes	Yes
SSE2 - Streaming SIMD Extensions v2:	Yes	Yes
SSE3 - Streaming SIMD Extensions v3:	Yes	Yes
SSSE3 - Supplemental SSE3:	Yes	Yes
Hyper-Threading Technology	No	Yes

15.4.3 Core i3, i5 and i7

Intel's core processors are divided into three ranges (Core i3, Core i5 and Core i7), with several models in each range. The differences between these ranges aren't same on laptop chips as on desktops. Desktop chips follow a more logical pattern as compared to laptop chips, but many of the technologies and terms, we are about to discuss, such as cache memory, the number of cores, Turbo boost and Hyper-Threading concepts is same. Laptop processors have to balance power efficiency with performance – a constraint that doesn't really apply to desktop chips. Similar is the case with the Mobile processors.

Let's start differentiating the processors on the basis of the concepts discussed below!

Concepts and Technologies

Total number of cores present: Out of all differences between the intel processor ranges, this is one that will affect performance the most.

Having several cores can also drastically increase the speed at which certain programs run. The Core i3 range is entirely dual core, while Core i5 and i7 processors have four cores.It is difficult for an application to take advantage of the multicore system. Each core is effectively its own processor – your PC would still work (slowly) with just one core enabled. Having multiple cores means that the computer can work on more than one task at a time more efficiently.

Personal Computer	Intel Core i3	Intel Core i5	Intel Core i7
Number of Cores	2	4	4

What is Turbo Boost in processors?

This may be interesting, the slowest Core i3 chips runs at a faster speed than the base Core i5 and Core i7. This is where clock speed comes into the scenario. Let's first define, What is Clock speed?

The GHz represents the number of clock cycles (calculations) a processor can manage in a second. Putting simply, a bigger number means a faster processor.

Examples:

2.4GHz means 2,400,000,000 clock cycles.

Personal Computer	Intel Core i3	Intel Core i5	Intel Core i7
Clock Speed Range (Several Models)	3.4GHz – 4.2GHz	2.4GHz – 3.8GHz	2.9GHz – 4.2GHz

Turbo Boost has nothing to do with fans or forced induction but is Intel's marketing name for the technology that allows a processor to increase its core clock speed dynamically whenever the need arises. Core i3 processors don't have Turbo Boost, but Core i5 and Core i7s do. Turbo Boost dynamically increases the clock speed of Core i5 and i7 processors when more power is required. This means that the chip can draw less power, produce less heat and only boost when it needs to. For example, although a Core i3-7300 runs at 4GHz compared to 3.5GHz for the Core i5-7600, the Core i5 chip can boost up to 4.1GHz when required, so will end up being quicker. A processor can only Turbo Boost for a limited amount of time. It is a significant part of the reason why Core i5 and Core i7 processors outperform Core i3 models in single-core-optimised tasks, even though they have lower base clock speeds.

Personal Computer	Intel Core i3	Intel Core i5	Intel Core i7
Turbo Boost	No	Yes	Yes

Note:

If a processor model ends with a K, it means it is unlocked and can be 'overclocked'. This means you can force the CPU to run at a higher speed than its base speed all the time for better performance.

Cache memory: A processor's performance isn't only determined by clock speed and number of cores, though. Other factors such as cache memory size also play a part. When a CPU finds it is using the same data over and over, it stores that data in its cache. Cache is even faster than RAM because it's part of the processor itself.

Here, bigger is better. Core i3 chips have 3- or 4MB, while i5s have 6MB and the Core i7s have 8MB.

Personal Computer	Intel Core i3	Intel Core i5	Intel Core i7
Cache Memory	3 – 4MB	4 – 6MB	8MB

What is Hyper-Threading?

It's one of the concepts which is a little confusing to explain, but also confuses as it's available on Core i7 and Core i3, but not on the mid-range core i5. A little shocking, right? Normally we assume that we get more features as we go higher towards the processor range, but not here. Back to the concept, A thread in computing terms is a sequence of programmed instructions that the CPU has to process. For example: If a CPU consists of one core, it can process only one thread at once, so can only do one thing at once.

Hyper-Threading is a clever way to let a single core handle multiple thread. It essentially tricks operating system into thinking that each physical processor core is, in fact, two virtual (logical) cores. A two-core Core i3 processor will appear as four virtual cores in Task Manager, and a four-core i7 chip will appear as eight cores. Whereas the current Core i5 range doesn't have Hyper-Threading so can also only process four cores. Due to Hyper-Threading operating system can share processing tasks between these virtual cores in order to help certain applications run more quickly, and to maintain system performance when more than one application is running at once.

Personal Computer	Intel Core i3	Intel Core i5	Intel Core i7
Hyper-Threading	Yes	No	Yes

From these, we conclude why Core i7 processors are the creme de la creme. Not only are they quad cores, they also support Hyper-Threading. Thus, a total of eight threads can run on them at the same time. Combine that with 8MB of cache and Intel Turbo Boost Technology, which all of them have, and you'll see what sets the Core i7 apart from its siblings.

On the other side, it totally depends on the requirements, to choose a processor.

Table 4.3: Comparative study if i3, i5 and i7 Processors

Sr No.	Parameter	Multicore i3	Multicore i5	Multicore i7
1	clock rate	2.933 GHz to 3.2 GHz	2.4 GHz to 3.33 GHz	2.4 GHz to 3.33 GHz
2	Launch Year	2010	2009	-
3	Cores	two cores	two or four cores	two, four, or six cores
4	Turbo Boost	No	Yes	Yes
5	cache memory	3- or 4MB	6MB	8MB.

15.5 Summary

In this chapter we have studied various microprocessor architectures like Pentium IV, i2, i3, i5, i7 etc. We have seen their comparison using various properties like speed, pipelining, cache size, hyper threading, clock speed etc.

15.6 Review Your Learnings:

1. Are you able to analyse the properties of microprocessors?
2. Are you able to recognize various clock speeds of microprocessors?
3. Are you able to explain microarchitecture of processor cores?
4. Are you able to do performance evaluation of various microprocessors?
5. Are you able to figure out the impact of cache size and hyper threading on performance of microprocessor?

15.7 Sample Questions:

1. Enlist various microprocessor names and its core types.
2. Compare the performance of Pentium IV and i5 microprocessor.
3. What do you mean by Hyper-Threading in microprocessors?
4. Explain the impact of cache size, Hyper threading and pipelining in efficiency of microprocessor.
5. Explain architecture of Pentium IV

15.8 References for further reading

- https://www.pdfdrive.com/computer-system-architecture-morris-mano-third-edition-e51589001.html
- https://www.pdfdrive.com/microprocessor-architecture-programming-and-applications-with-the-8085-d176171206.html
- Pentium Pro Family Developers Manual, Volume 2: Programmer's Reference Manual, Intel Corporation, 1996
- Pentium Pro Family Developers Manual, Volume 3: Operating System Writer's Manual, Intel Corporation, 1996
- https://www.researchgate.net/publication/278912508_Performance_Analysis_of_Dual_Core_Core_2_Duo_and_Core_i3_Intel_Processor/link/55afa4da08aeb0ab4668933e/download
- http://www.x86.org/intel.doc/intelDocs.html
- https://www.byclb.com/TR/Tutorials/microprocessors/ch2_1.htm#:~:text=The%2016%2Dbit%20registers%20are,in%20the%2080386%20and%20above
- https://eun.github.io/Intel-Pentium-Instruction-Set-Reference/data/index.html
- https://www.lpthe.jussieu.fr/~talon/pentiumII.pdf
- http://www.darshan.ac.in/Upload/DIET/Documents/CE/2150707-MPI-Study-Material_04112017_033410AM.pdf

❖❖❖

UNIT 5

16

SUN SPARC MICROPROCESSOR

Unit Structure

16.0 Objectives:

16.1 SUN SPARC Architecture

16.1.1 Integer Unit:

16.1.2 Integer Unit: Register Window

16.1.3 Floating-point Unit (FPU)

16.1.4 Coprocessor Unit (CU)

16.2. Register File

16.2.1 Integer Unit: Register Window

16.2.2 Advantage: Register Window

3. Data Types

4. Instruction Format

16.4.1 Arithmetic/Logical/Shift instructions

16.4.2 Load/Store Instructions

16.4.3 Branch Instructions

16.4.4 SPARC Fundamental Instructions

16.4.4.1 Load/Store Instructions

16.4.4.2 Arithmetic/Logical Instructions

16.4.4.3 Call Instruction

16.4.4.4 Synthetic Instructions 13

16.5. Summary

16.6. Review Your Learnings:

16.7. Questions

16.8. References for further reading

16.0 Objectives

1. Explain SPARC architecture.
2. Explain the various Registers used in SPARC architecture
3. Explain the instruction set of SPARC
4. Explain advantages of Register Window of SPARC architecture.
5. Explain Instruction set categories with example

16.1 SUN SPARC Architecture

SPARC is an acronym for Scalable Processor ARChitecture.

Its specifications are listed below:

- Engineered at Sun Microsystems in 1985
- Designed to optimize compilers and pipelined hardware implementations
- Offers fast execution rates
- SPARCs are load/store RISC processors.
- Load/store means only loads and stores access memory directly.
- RISC (Reduced Instruction Set Computer) means the architecture is simplified with a limited number of instruction formats and addressing modes.
- Simple, uniform instruction set allowing fast cycle times.
- Goal — “One instruction per cycle.”(RISC)
- Up to 128 general-purpose registers
- All arithmetic operations are register-to-register
- Simplified instruction set
- Higher number of instructions with fewer transistors
- Flexible integration of cache, memory and FPUs
- 64-bit addressing and 64-bit data bus
- Increased bandwidth
- Fault tolerance
- Nine stage pipeline; can do up to 4 instructions per cycle

- On-chip 16Kb data and instruction caches with 2Mb external cache
- A large "windowed" register file — at any one instant, a program sees 8 global integer registers plus a 24-register window into a larger register file.

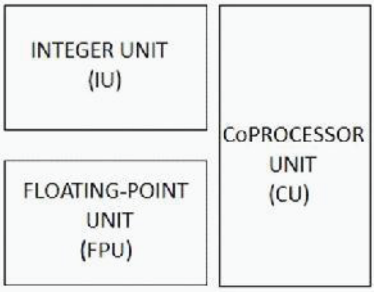


Figure 1.1: SPARC Architecture

16.1.1 Integer Unit:

- Contains the general purpose registers and controls the overall operation of the processor.
- Executes the integer arithmetic instructions and computes memory addresses for loads and stores.
- Maintains the program counters and controls instruction execution for the FPU.

16.1.2 Integer Unit: Register Window

When a procedure is called, the register window shifts by sixteen registers, hiding the old input registers and old local registers and making the old output registers the new input registers.

- Input registers : arguments are passed to a function
- Local registers: to store any local data.
- Output registers: When calling a function, the programmer puts his argument in these registers.

16.1.3 Floating-point Unit (FPU)

- The FPU has
- 32 Registers (32-bit single-precision floating-point registers)
- 32 Registers (64-bit double-precision floating-point registers)
- 16 Registers (128-bit quad-precision floating-point registers)
- Floating-point load/store instructions are used to move data between the FPU and memory.

- The memory address is calculated by the IU.
- Floating-Point operate (FPop) instructions perform the floating-point arithmetic operations and comparisons.

16.1.4 Coprocessor Unit (CU)

- The instruction set includes support for a single, implementation-dependent coprocessor.
- The coprocessor has its own set of registers.
- Coprocessor load/store instructions are used to move data between the coprocessor registers and memory.

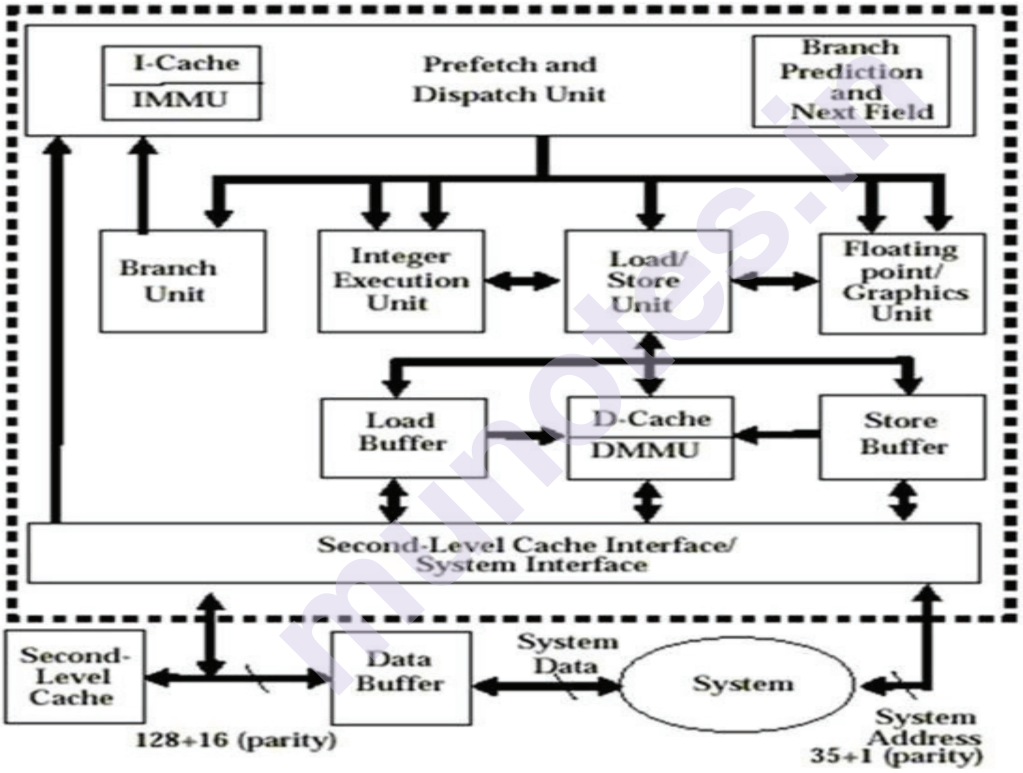


Figure 1.2: Block Diagram of Ultra SPARC

16.2 Register File

The SPARC architecture's definition includes the IU (Integer Unit) which is the CPU, the FPU (Floating Point Unit) and the CP (Co-Processor) which is optional for the user. Other options are the memory management unit and cache.

An important concept of the SPARC architecture is borrowed from the Berkeley RISC chips, the TMS 9900 mainly. This is register windowing. When a program is running it has access to 32 32-bit processor registers which include eight global registers plus 24 registers that belong to the current register window. The first 8 registers in the window are called the in registers (i0-i7). When a function is called, these registers may contain arguments that can be used. The next 8 are the local registers which are scratch registers that can be used for anything while the function executes. The last 8 registers are the out registers which the function uses to pass arguments to functions that it calls.

When one function calls another, the callee can choose to execute a SAVE instruction. This instruction decrements an internal counter, the current workspace pointer, shifting the register window downward. The caller's out registers then become the callee' s in registers, and the callee gets a new set of local and out registers for its own use. Only the pointer changes because the registers and return address do not need to be stored on a stack. The CALL instruction automatically saves its own address in 07 (output register 7) which becomes input register 7 if the CWP is decremented. Therefore, the callee can access the return address whether or not it has decremented the CWP.

Register windows are also used to save the processor contexts when traps, or interrupts occur. The SPARC OS's always ensure that there is a register window not being used below the current one. If a trap occurs, then the CWP is decremented and the new window saves the processor context.

The chip that was implemented by Sun had seven overlapping windows which brought the total of registers to (7*16) + 7 (without counting g0) which is 119 registers. If six levels are not enough due to recursive or deeply nested function calls, then the program attempts to decrement the CWP to the last unused window and it discovers that the window has been marked invalid in a register called the window invalid mask register. This causes a trap and the processor has an opportunity to "spill" register s in order to make more room. It writes some of the contents out to memory.

A long series of subroutine returns can cause a window underflow, which consequently causes the processor to call in a trap handler that fills registers from memory. All the spilling and filling is hidden from an executing user program usually. Spilling and filling registers is an essential part of Unix multitasking on SPARC.

Sparc has 32 general purpose integer registers visible to the program at any given time. Of these, 8 registers are global registers, and 24 registers are in a register window. A window consists of three groups of 8 registers, the out, local, and in registers. See table 1. A Sparc implementation can have from 2 to 32 windows, thus varying the number of registers from 40 to 520. Most implementation have 7 or 8 windows. The variable number of registers is the principal reason for the Sparc being "scalable".

At any given time, only one window is visible, as determined by the current window pointer (CWP) which is part of the processor status register (PSR). This is a five bit value that can be decremented or incremented by the SAVE and RESTORE instructions, respectively. These instructions are generally executed on procedure call and return (respectively). The idea is that the in registers contain incoming parameters, the local register constitute scratch registers, the out registers contain outgoing parameters, and the global registers contain values that vary little between executions. The register windows overlap partially, thus the out registers become renamed by SAVE to become the in registers of the called procedure. Thus, the memory traffic is reduced when going up and down the procedure call. Since this is a frequent operation, performance is improved.

(That was the idea, anyway. The drawback is that upon interactions with the system the registers need to be flushed to the stack, necessitating a long sequence of writes to memory of data that is often mostly garbage. Register windows was a bad idea that was caused by simulation studies that considered only programs in isolation, as opposed to multitasking workloads, and by considering compilers with poor optimization. It also caused considerable problems in implementing high-end Sparc processors such as the SuperSparc, although more recent implementations have dealt effectively with the obstacles. Register windows is now part of the compatibility legacy and not easily removed from the architecture.)

The overlap of the registers is illustrated in Figure 2.2. The figure shows an implementation with 8 windows, numbered 0 to 7 (labelled w0 to w7 in the figure). Each window corresponds to 24 registers, 16 of which are shared with "neighbouring" windows. The windows are arranged in a wrap-around manner, thus window number 0 borders window number 7. The common cause of changing the current window, as pointed to by CWP, is the RESTORE and SAVE instructions, shown in the middle. Less common is the supervisor RETT instruction (return from trap) and the trap event (interrupt, exception, or TRAP instruction).

The "WIM" register is also indicated in the top left of Figure 2.2. The *window invalid mask* is a bit map of valid windows. It is generally used as a pointer, i.e., exactly one bit is set in the WIM register indicating which window is invalid (in the figure it is window 7). Register windows are generally used to support procedure calls, so they can be viewed as a cache of the stack contents. The WIM "pointer" indicates how many procedures calls in a row can be taken without writing out data to memory. In the figure, the capacity of the register windows is fully utilized. An additional call will thus exceed capacity, triggering a *window overflow* trap. At the other end, a *window underflow* trap occurs when the register window "cache" if empty and more data needs to be fetched from memory.

16.2.1 Integer Unit: Register Window

The SPARC register windows are, naturally, intimately related to the stack. In particular, the stack pointer (%sp or %o6) must always point to a free block of 64 bytes. This area is used by the operating system (Solaris, SunOS, and Linux at least) to save the current local and in registers upon a system interrupt, exception, or trap instruction. (Note that this can occur at any time.)

Other aspects of register relations with memory are programming convention. The typical, and recommended, layout of the stack is shown in Figure 2.3. The figure shows a stack frame.

Note that the top boxes of Figure 2.3 are addressed via the stack pointer (%sp), as positive offsets (including zero), and the bottom boxes are accessed over the frame pointer using negative offsets (excluding zero), and that the frame pointer is the old stack pointer. This scheme allows the separation of information known at compile time (number and size of local parameters, etc) from run-time information (size of blocks allocated by alloca()).

"addressable scalar automatics" is a fancy name for local variables.

The clever nature of the stack and frame pointers are that they are always 16 registers apart in the register windows. Thus, a SAVE instruction will make the current stack pointer into the frame pointer and, since the SAVE instruction also doubles as an ADD, create a new stack pointer.

- When a procedure is called, the register window shifts by sixteen registers, hiding the old input registers and old local registers and making the old output registers the new input registers.
- Input registers: arguments are passed to a function
- Local registers: to store any local data.
- Output registers: When calling a function, the programmer puts his argument in these registers.

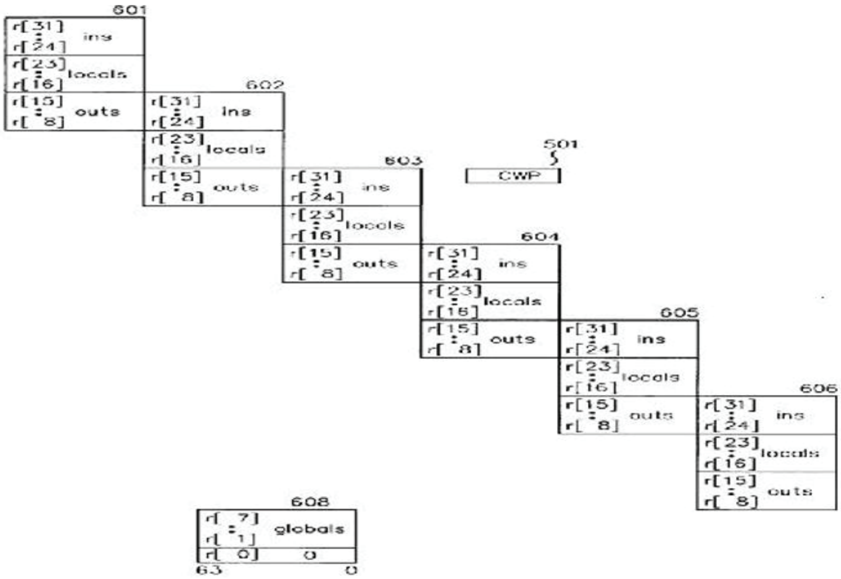


Figure 2.1: Register Window

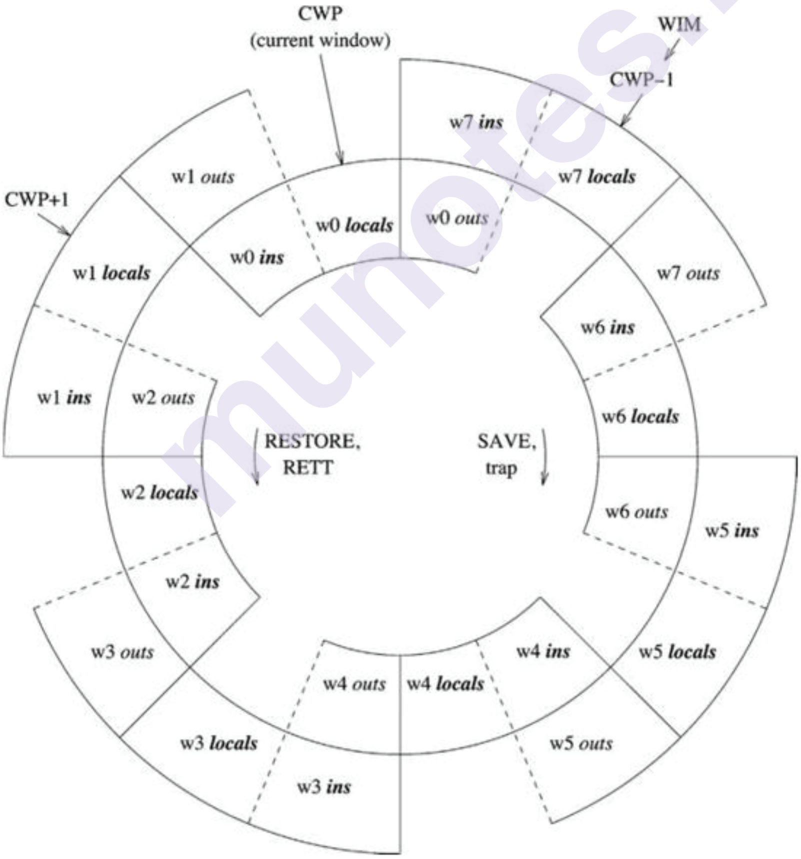
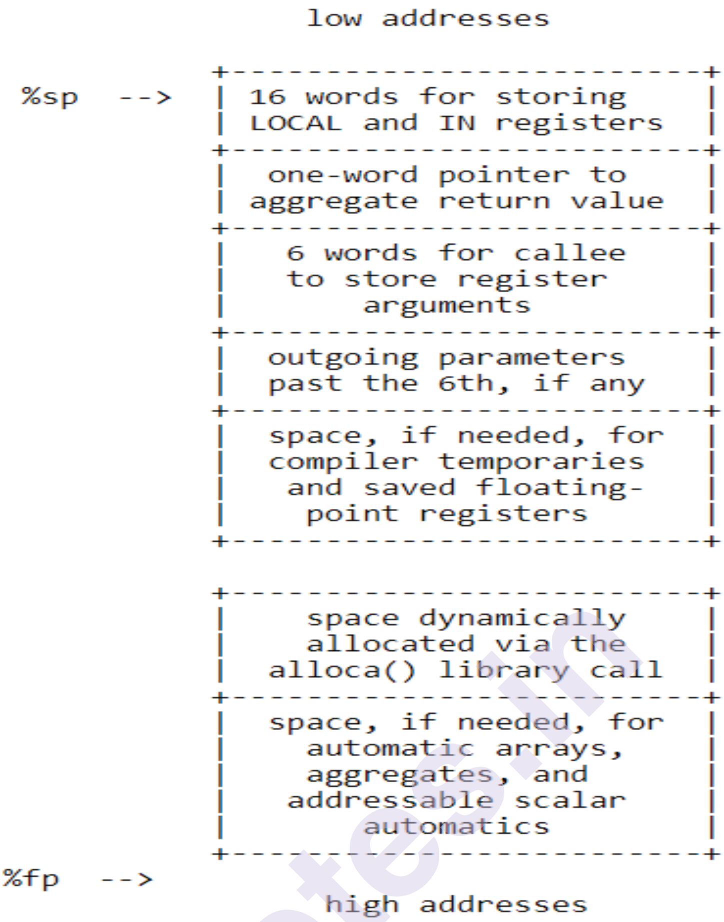


Figure 2.2: Circular Arrangement of Register Window



```
                  low addresses

           +--------------------------+
%sp  -->   | 16 words for storing     |
           | LOCAL and IN registers   |
           +--------------------------+
           |  one-word pointer to     |
           | aggregate return value   |
           +--------------------------+
           |   6 words for callee     |
           |   to store register      |
           |       arguments          |
           +--------------------------+
           |  outgoing parameters     |
           |  past the 6th, if any    |
           +--------------------------+
           |  space, if needed, for   |
           |  compiler temporaries    |
           |   and saved floating-    |
           |    point registers       |
           +--------------------------+

           +--------------------------+
           |    space dynamically     |
           |    allocated via the     |
           |  alloca() library call   |
           +--------------------------+
           |  space, if needed, for   |
           |    automatic arrays,     |
           |    aggregates, and       |
           |   addressable scalar     |
           |       automatics         |
           +--------------------------+
%fp  -->
                  high addresses
```

Figure 2.3: Stack Frame Contents

16.2.2 Advantage: Register Window

- Make very fast procedure calls as they avoid the need to save a processor's current in memory, further reducing off-chip traffic.
- Instead, the state variables are held in the current window, and the next window is opened for the new procedure.
- A refinement on this idea in that the input and output registers of adjacent windows overlap, allowing variables and parameters to be passed to the next process without physically moving data.
- The additional registers are hidden from view until you call a subroutine or other function. Where other processors would push parameters on a stack for the called routine to pop off, SPARC processors just "rotate" the register window to give the called routine a fresh set of registers.
- The old window and the new window overlap, so that some registers are shared.

16.3 Data Types

The SPARC architecture recognizes three fundamental data types:

- Signed Integer— 8, 16, 32, and 64 bits
- Unsigned Integer— 8, 16, 32, and 64 bits
- Floating-Point — 32, 64, and 128 bits

The format widths are defined as:

- Byte — 8 bits
- Half word— 16 bits
- Word/Single word — 32 bits
- Tagged Word— 32 bits (30-bit value plus 2 tag bits)
- Double word— 64 bits
- Quad word— 128 bits

SPARC is "big-endian"- it stores multiple byte objects in memory with the most significant byte at the lowest address.

16.4 Instruction Format

There are very few addressing modes on the SPARC, and they may be used only in certain very restricted combinations. The three main types of SPARC instructions are given below, along with the valid combinations of addressing modes. There are only a few unusual instructions which do not fall into these categories.

16.4.1 Arithmetic/Logical/Shift instructions

```
opcode reg1, reg2, reg3        !reg1 op reg2 -> reg3
opcode reg1, const13, reg3     !reg1 op const13 -> reg3
```

- All "action" instructions (add, sub, and, or, etc.) take three operands.
- The destination is always the third operand.
- The middle operand may be a 13-bit signed constant (-4096...+4095).
- Otherwise, all operands are registers.
- To do the above things in the 680x0, 6 different opcodes would be needed (move, add, addi, clr, neg, cmp)

Examples:

```
add %L1, %L2, %L3              !%L1+%L2->%L3
add %L1,1,%L1                  !increment L1
sub %g0,%i3,%i3                !negate i3
sub %L1,10,%G0                 !compare %L1 to 10 (discard result)
add %L1,%G0,%L2                !move %L1 to %L2 (add 0 to it)
add %G0,%G0,%L4                !clear L4 (0+0 ->%L4)
```

16.4.2 Load/Store Instructions

opcode [reg1+reg2], reg3 opcode [reg1+const13], reg3

- Only load and store instructions can access memory.
- The contents of reg3 is read/written from/to the address in memory formed by adding reg1+reg2, or else reg1+const13 (a 13- bit signed constant as above).
- The operands are written in the reverse direction for store instructions, so that the destination is always last.
- One of reg1 or const13 can be omitted. The assembler will supply $g0 or 0. (This is a shorthand provided by the assembler. Both are always there in machine language.)

Examples:

```
ld [%L1+%L2], %L3        !word at address [%L1+%L2]->%L3
ld [%L1+8],%L2           !word at address [%L1+8]->%L2
ld [%L1],%L2             !word at address [%L1]->%L2
st %g0,[%i2+0]           !0 -> word at address in %i2
st %g0,[%i2]             !same as above
```

16.4.3 Branch Instructions

opcode address

- Branch to (or otherwise use) the address given.
- There are actually 2 types of addresses, but they look the same.

Examples:

```
call printf
be    Loop
```

That's it. Period. No other modes or combinations of modes are possible. This is a RISC machine and R stands for "Reduced".

add %L1,[%L2],%L3 !Invalid. No memory access allowed.
ld 5,%L4 !Invalid. Must be a memory access.

16.4.4 SPARC Fundamental Instructions

16.4.4.1 Load/Store Instructions

- Only these instructions access memory.
- All 32 bits of the register are always affected by a load. If a shorter data item is loaded, it is padded by either adding zeroes (for unsigned data), or by sign extension (for signed data).
- In effect, data in memory may be 1, 2, or 4 bytes long, but data in registers is always 4 bytes long.

ld - load (load a word into a register)

st - store (store a word into memory)

ldub - load unsigned byte (fetch a byte, pad with 0's)

ldsb - load signed byte (fetch a byte, sign extend it)

lduh - load unsigned halfword (fetch 2 bytes, pad)

ldsh - load signed halfword (fetch 2 bytes, sign extend)

stb - store byte (store only the LSB)

sth - store halfword (store only the 2 LSB's)

There are also two instructions for double words. The register number must be even, and 8 bytes are loaded or stored. The MSW goes to the even register and the LSW to the odd register that follows it.

ldd - load double (load 2 words into 2 registers)

std - store double (store 2 words from 2 registers)

4.4.2 Arithmetic/Logical Instructions

- All 32 bits of every register is used.
- Setting the condition code is always optional. Add "cc" to the opcode to set the condition code. By default, it is **not** set.

add - a+b

sub - a-b

and - a&b (bitwise AND)

andn - a&~b (bitwise and - second operand complemented)

or - a|b (bitwise OR)

orn - a|~b (bitwise or - second operand complemented)

xor - a^b (bitwise exclusive or)

xnor - a^~b (bitwise exor - second operand complemented)

Examples:

```
add   %L1,%L2,%L3  ;add %L1+%L2 -> %L3
subcc %L4,10,%G0   ;sub %L4-10, set cc, discard result
or    %o3,0xFF,%o3 ;set lowest 8 bits of %o3 to 1's
xnor  %L6,%G0,%L6  ;complement %L6 (same as NOT in 680x0)
```

16.4.4.3 Call Instruction

This instruction is used to call subprograms. As for the 680x0, we will leave the details for later. For now, it will be used only to call library routines.

call printf

16.4.4.4 Synthetic Instructions

Synthetic Instruction	**Assembled As**
clr %reg	or %g0,%g0,%reg
cmp %reg,%reg	subcc %reg,%reg,%g0
cmp %reg,const	subcc %reg,const,%g0
mov %reg,%reg	or %g0,%reg,%reg
mov const,%reg	or %g0,const,%reg
set const,%reg	sethi %hi(const),%reg
	or %reg,%lo(const22),%reg

And here are some others that may be useful:

Synthetic Instruction	**Assembled As**
clr [address]	st %g0,[address]
clrh [address]	sth %g0,[address]
clrb [address]	stb %g0,[address]
dec %reg	sub %reg,1,%reg
deccc %reg	subcc %reg,1,%reg
inc %reg	add %reg,1,%reg

inccc %reg	addcc %reg,1,%reg
not %reg	xnor %reg,%g0,%reg
neg %reg	sub %g0,%reg,%reg
tst %reg	orcc %reg,%g0,%g0

Here are two that will be used for subprograms later:

Synthetic Instruction	**Assembled As**
restore	restore %g0,%g0,%g0
ret	jmpl %i7+8,%g0

16.5 Summary

In this chapter we have studied about Scalable Processor Architecture (SPARC). The datatypes associated with it. WE have seen the block diagram of architecture of ULTRA SPARC architecture. We have also seen the circular arrangement of Register Window.

16.6 Review Your Learnings:

1. Are you able to understand SPARC Architecture?
2. Can you explain the various instructions format?
3. Can you explain the Registers Windows?
4. Can you explain the Data types associated with SPARC Architecture?

16.7 Questions

1. Explain SPARC architecture.
2. Explain the various Registers used in SPARC architecture
3. Explain the instruction set of SPARC
4. Explain advantages of Register Window of SPARC architecture.
5. Explain Instruction set categories with example.
6. Draw the block diagram and architecture of ULTRA SPARC.
7. Explain the advantages of SPARC over other processors.

16.8 References for further reading

- http://datasheets.chipdb.org/Intel/x86/Pentium%20II/SpecUpdate/24333745.pdf
- https://en.wikipedia.org/wiki/Pentium_II
- https://www.lpthe.jussieu.fr/~talon/pentiumII.pdf
- https://eun.github.io/Intel-Pentium-Instruction-Set-Reference/data/index.html
- https://www.pdfdrive.com/computer-system-architecture-morris-mano-third-edition-e51589001.html
- https://www.pdfdrive.com/microprocessor-architecture-programming-and-applications-with-the-8085-d176171206.html
- Pentium Pro Family Developers Manual, Volume 2: Programmer's Reference Manual, Intel Corporation, 1996

❖❖❖