Insights on MICROPROCESSORS

TU, PU, PoU, KU, BScCSIT, BIM, BIT, BCA

Features

- A textbook with INSTANT NOTES
- Solution to questions of past exams

Er. Hari Prasad Aryal

Er. Shyam Dahal

परवाजिमा नै मर्दा रहेछन् साँच्चो गाथा जानेहरु

Insights on

MICROPROCESSORS

Bought By
Mr. Bikash Pandey
when he was heavily bored.

(Haha)

By:

Er. Hari Prasad Aryal
(Sr. Technical Officer, SCT Pvt. Ltd.)

Er. Shyam Dahal
(Lecturer, Kathmandu Engineering College)

feel time

feel the taste taste the feel

SYSTEM INCEPTION

PREFACE TO THE FIRST EDITION

The textbook "***Insights on* Microprocessors**" is an outcome of our teaching experience and never ending efforts in making the subject easier. This book presents all topics in a very simple and lucid way so that the students will feel comfortable to prepare for the exams and implement their knowledge in real-world applications.

We are highly indebted to **Er. Anmol Ratna Bajracharya** (Associate Professor, KEC, Kalimati) who paved the way towards accomplishing this fine project. We are also very thankful to all others who directly or indirectly helped us bringing the book in the most understandable format.

Before I pen down, we want to request "Always be helpful and never hurt others". Please make these feelings go viral. Let's together pray for a prosperous and peaceful Nepal.

Er. Hari Prasad Aryal
Er. Shyam Dahal
July, 2017

CONTENTS

Chapter - **1**

INTRODUCTION

A microprocessor is a multipurpose programmable, clock driven, register based electronic device that reads binary instructions from a storage device called memory, accepts binary data as input, processes data according to those instructions and provide results as output. The microprocessor operates in binary 0 and 1 known as bits and are represented in terms of electrical voltages in the machine. That means, 0 represents low voltage level and 1 represents high voltage level. Each microprocessor recognizes and processes a group of bits called the word and microprocessors are classified according to their word length such as 8 bits microprocessor with 8 bit word and 32 bit microprocessor with 32 bit word etc.

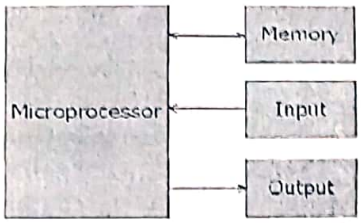


Fig.: A programmable machine

Terms used

- CPU: Central processing unit which consists of ALU and control unit.
- Microprocessor: Single chip containing all units of CPU.
- Microcomputer: Computer having microprocessor as CPU.
- Microcontroller: Single chip consisting of MPU, memory, I/O and interfacing circuits.
- MPU (Microprocessing Unit): Complete processing unit with the necessary control signals.

Basic Block Diagram of a Computer

Traditionally, the computer is represented with four components such as memory, input, output and central processing unit (CPU) which consists of arithmetic logic unit (ALU) and control unit (CU).

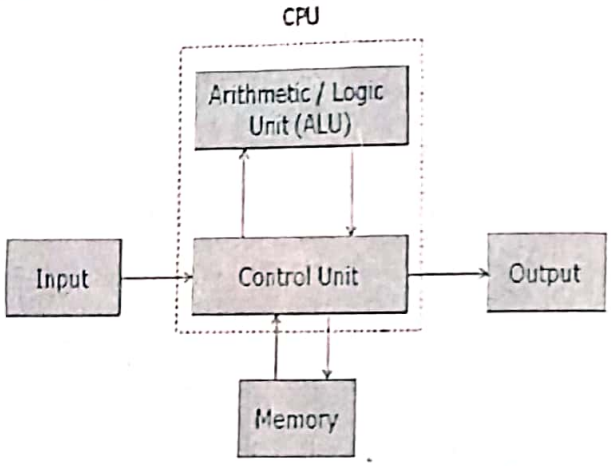


Fig.: Traditional block diagram of a computer

The CPU contains various registers to store data, the ALU to perform arithmetic and logical operations, instruction decoders, counters and control lines. The CPU reads instructions from memory and performs the tasks specified. It communicates with input/output (I/O) devices either to accept or to send data, the I/O devices are known as peripherals.

Around late 1960's, traditional block diagram was replaced with computer having microprocessor as CPU which is known as microcomputer. Here CPU was designed using integrated circuit technology (ICs) which provided the possibility to build the CPU on a single chip.

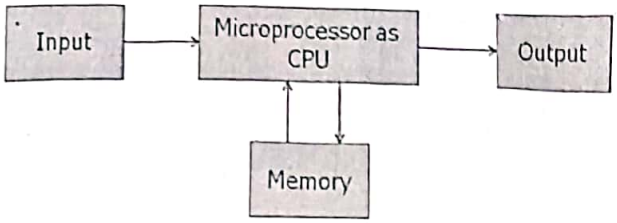


Fig.: Block diagram of a computer with the microprocessor as CPU

Later on, semiconductor fabrication technology became more advanced. Manufacturers were able to place not only MPU but also memory and I/O interfacing circuits on a single chip known as microcontroller, which also includes additional devices such as A/D converter, serial I/O, timer etc.

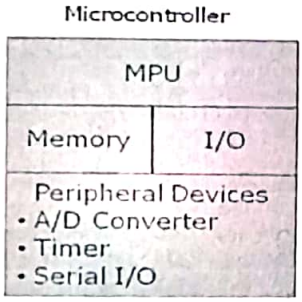


Fig.: Block diagram of a microcontroller

Organization of a Microprocessor Based System

Microprocessor based system includes three components: microprocessor, input/output and memory (read only and read/write). These components are organized around a common communication path called a bus.

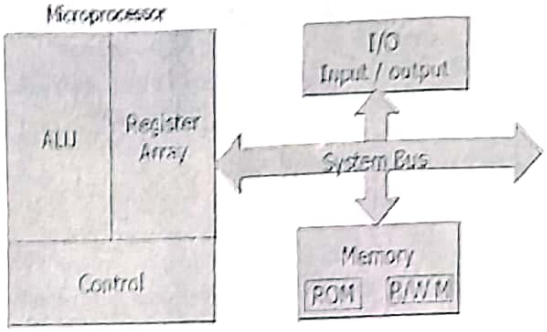


Fig.: Microprocessor based system with bus architecture

Microprocessor:

It is a clock driven semiconductor device consisting of electronic logic circuits manufactured by using either a large scale integration (LSI) or very large scale integration (VLSI) technique. It is capable of performing various computing functions and making decisions to change the sequence of a program execution. It can be divided into three segments.

a. Arithmetic/logic unit: It performs arithmetic operations as addition, subtraction and logic operations as AND, OR, XOR.

b. Register array: The registers are primarily used to store data temporarily during the execution of a program and are accessible to the user through instruction. The registers can be identified by letters such as B, C, D, E, H, and L.

c. Control unit: It provides the necessary timing and control signals to all the operations in the microcomputer. It controls the flow of data between the microprocessor and memory & peripherals.

Memory:

Memory stores binary information such as instructions and data, and provides that information to the microprocessor whenever necessary. To execute programs, the microprocessor reads instructions and data from memory and performs the

computing operations in its ALU. Results are either transferred to the output section for display or stored in memory for later use. Memory has two sections.

a. Read only memory (ROM): Used to store programs that do not need alterations and can only be read.

b. Read/write memory (RAM): Also known as user memory which is used to store user programs and data. The information stored in this memory can be easily read and altered.

Input/Output:

- It communicates with the outside world using two devices input and output which are also known as peripherals.
- The input devices such as keyboard, switches, and analog to digital converter transfer binary information from outside world to the microprocessor.
- The output devices transfer data from the microprocessor to the outside world. They include the devices such as LED, CRT, digital to analog converter, printer etc.

System Bus:

It is a communication path between the microprocessor and peripherals; it is nothing but a group of wires to carry bits.

Bus Organization

Bus is a common channel through which bits from any sources can be transferred to the destination. A typical digital computer has many registers and paths must be provided to transfer instructions from one register to another. The number of wires will be excessive if separate lines are used between each register and all other registers in the system. A more efficient scheme for transferring information between registers in a multiple register configuration is a common bus system. A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time. Control signals determine which register is selected by the bus during each particular register transfer.

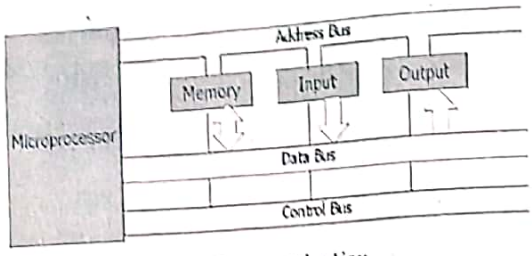


Fig.: Bus organization

A very easy way of constructing a common bus system is with multiplexers. The multiplexers select the source register whose binary information is then placed on the bus.

A system bus consists of about 50 to 100 of separate lines each assigned a particular meaning or function. Although there are many different bus designers, on any bus, the lines can be classified into three functional groups; data, address and control lines. In addition, there may be power distribution lines as well.

- The data lines provide a path for moving data between system modules. These lines are collectively called data bus.
- The address lines are used to designate the source/ destination of data on data bus.
- The control lines are used to control the access to and the use of the data and address lines. Because data and address lines are shared by all components, there must be a means of controlling their use. Control signals transmit both command and timing signals. Timing signals indicate the validity of data and address information. Command signals specify operations to be performed. Control lines include memory read/write, I/O read/write, bus request/grant, clock, reset, interrupt request/acknowledge etc.

Historical Background of the Development of Computers:

The most efficient and versatile electronic machine computer is basically a development of a calculator which leads to the development of the computer. The older computer were mechanical and newer are digital. The mechanical computer namely difference engine and analytical engine developed by

Charles Babbage, the father of the computer can be considered as the forerunners of modern digital computers.

The difference engine was a mechanical device that could add and subtract, and could only run a single algorithm. It's output system was incompatible to write on punched cards and early optical disks. The 'analytical engine' provided more advanced features. It consisted mainly four components: the store (memory), the mill (computation unit) , input section (punched card reader), and output section (punched and printed output). The store consisted of 1000s of words of 50 decimal digits used to hold variables and results. The mill could accept operands from the store, add, subtract, multiply or divide them and return a result to the store.

The evolution of the vacuum tubes led the development of computer into a new era. The world's first general purpose electronic digital computer was ENIAC (Electronic Numerical Integrator and Calculator) built by using vacuum tubes was enormous in size and consumed very high power. However, it was faster than mechanical computers. The ENIAC was decimal machine and performed only decimal numbers. Its memory consisted of 20 'accumulators' each capable of holding 10 digits decimal numbers. Each digit was represented by a ring of 10 vacuum tubes. ENIAC had to be programmed manually by setting switches and plugging and unplug a cable which was the main drawback of it.

Automated calculator:

It is a data processing device that carries out logic and arithmetic operations but has limited programming capability for the user. It accepts data from a small keyboard one digit at a time, performs required arithmetic and logical calculations, and stores the result on visual display like LCD or LED. The calculator's programs are stored in ROMs while the data is stored in RAM.

Some important features of automated calculators:

- The ability to interface easily with keyboards and displays.
- The ability to handle decimal digits - the device is able to handle more than 4 bits at a time.

- Ability to execute the standard programs stored in read only memory.
- Extendibility, so that mathematical functions such as %, √, trigonometric, statistical etc. can be easily executed.
- Flexibility, so it can be used in engineering business or programming without a complete new design.
- Low cost, small size and low power consumptions.

Stored Program Concept and Von-Neumann Machine:

The simplest way to organize a computer is to have one processor, register and instruction code format with two parts op-code and address/operand. The memory address tells the control where to find an operand in memory. This operand is read from memory and used as data to be operated on together with the data stored in the processor register. Instructions are stored in one section of same memory. It is called stored program concept.

The task of entering and altering the programs for ENIAC was tedious. It could be facilitated if the program could be represented in a form suitable for storing in memory alongside the data. So, the computer could get its instructions by reading from the memory and program could be set or altered by setting the values of a portion of memory. This approach, known as 'stored-program concept' was first adopted by John Von-Neumann and such architecture is named as Von-Neumann architecture.

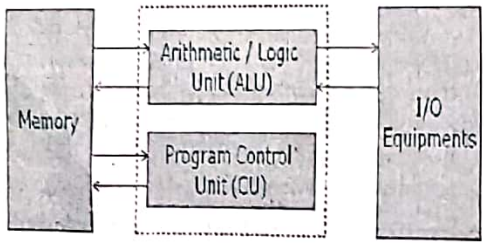


Fig.: Von-Neumann architecture

The main memory is used to store both data and instructions. The arithmetic and logic unit is capable of performing

Data can execute instructions

arithmetic and logical operation on binary data. The program control unit interprets the instruction in memory and causes them to be executed. The I/O unit gets operated from the control unit.

The Von-Neumann architecture is the fundamental basis for the architecture of modern digital computers. It consisted of 1000 storage locations which can hold words of 40 binary digits and both instructions as well as data are stored in it. The storage location of control unit and ALU are called registers and the various models of registers are:

- **MAR (Memory Address Register)** – contains the address in memory of the word to be written into or read from MBR.
- **MBR (Memory Buffer Register)** – consists of a word to be stored in or received from memory.
- **IR (Instruction Register)** – contains the 8-bit op-code to be executed.
- **IBR (Instruction Buffer Register)** – used to temporarily hold the instruction from a word in memory.
- **PC (Program Counter)** - contains the address of the next instruction to be fetched from memory.
- **AC & MQ (Accumulator and Multiplier Quotient)** - holds the operands and results of ALU after processing.

Harvard Architecture

In Von-Neumann architecture, the same memory is used for storing instructions and data. Similarly, a single bus called data bus or address bus is used for reading data and instructions from or writing to memory. This architecture limits the processing speed of computers.

The Harvard architecture based computer consists of separate memory spaces for the programs (instructions) and data. Each space has its own address and data buses. So, instructions and data can be fetched from memory concurrently and provides significance processing speed improvement.

In figure, there are two data and two address buses multipleved for data bus and address bus. Hence, there are two blocks of RAM chips: one for program memory and another for data memory addresses.

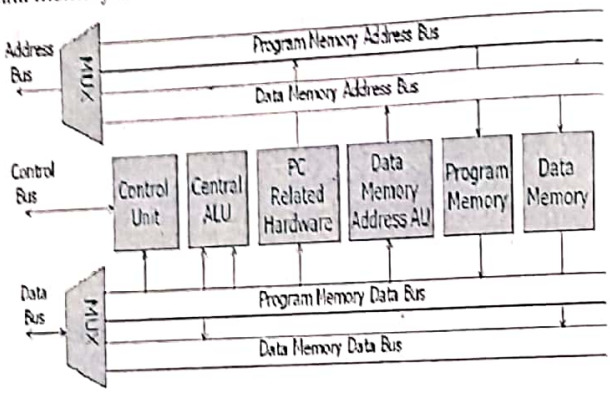


Fig.: Harvard architecture based microprocessor

The control unit controls the sequence of operations. Central ALU consists of ALU, multiplier, accumulator, and scaling chief register. The PC is used to address program memory and always contains the address of next instruction to be executed. Here, data and control buses are bidirectional and address bus is unidirectional.

Evolution of Microprocessors (Intel Series)

The CPU of a computer consists of ALU, CU, and memory. If all these components can be organized on a single chip by means of SSI, MSI, LSI, VLSI, ULSI, ELSI technology, then such chip is called microprocessor. It can fetch instructions from memory, decode and execute them, perform logical and arithmetic functions, accept data from input devices, and send results to the output devices. The evolution of microprocessor is dependent on the development of integrated circuit technology from single scale integration (SSI) to giga scale integration (GSI).

Date	Microprocessor	Data Bus	Address Bus	Memory
1971	4004	4 bit	10 bit	640 Bytes
1972	8008	8 bit	14 bit	16k
1974	8080	8 bit	16 bit	64k
1976	8085	8 bit	16 bit	64k
1978	8086	16 bit	20 bit	1M
1979	8088	8 bit	20 bit	1M
1982	80286	16 bit	24 bit	16M
1985	80386	32 bit	32 bit	4G
1989	80486	32 bit	32 bit	4G
1993	Pentium	32/64 bit	32 bit	4G
1995	Pentium pro	32/64 bit	36 bit	64G
1997	Pentium II	64 bit	36 bit	64G
1998	Celeron	64 bit	36 bit	64G
1999	Pentium III	64 bit	36 bit	64G
2000	Pentium IV	64 bit	36 bit	64G
2001	Itanium	128 bit	64 bit	64G
2002	Itanium 2	128 bit	64 bit	64G
2003	Pentium M/Centrino (wireless capability) for Mobile version e.g. Laptop			
2006	Pentium Dual Core (32 bit or 64 bit processor)			
2006	Core 2 Series (64 bit processor)- Dual and Quad Core Processor			
2008	Atom (32 bit or 64 bit processor) – Single or Dual Core Processor			
2010-present	Core i3 (Dual Core Processor)			
2009-present	Core i5 (Dual and Quad Core Processor)			
2008- present	Core i7 (Dual and Quad Core Processor)			
2011- present	Core i7 Extreme Edition			

Processing Cycle of a Stored Program Computer

- Fetch
- Identify
- Fetch Data
- Process
- Write Back

Microinstructions and Hardwired/Microprogrammed Control Unit

Micro-Operations:

- A computer executes a program consisting instructions. Each instruction is made up of shorter sub-cycles as fetch, indirect, execute cycle, and interrupt.
- Performance of each cycle has a number of shorter operations called micro-operations.
- It is called so because each step is very simple and does very little.
- Thus, micro-operations are functional atomic operation of CPU.
- Hence, events of any instruction cycle can be described as a sequence of micro-operations.

Microinstructions:

Each instruction is characterized with many machine cycles and each cycle is characterized with many T-states. The lower instruction level patterns which are the numerous sequences for a single instruction are known as microinstructions. We can visualize the microinstruction with the help of fetch cycle, or read cycle or write cycle.

Fetch Registers

- Memory Address Register (MAR)
  - connected to address bus
  - specifies address for read or write op-code
- Memory Buffer Register (MBR)
  - connected to data bus
  - holds data to write
- Program Counter (PC)
  - holds address of next instruction to be fetched
- Instruction Register (IR)
  - holds last instruction fetched

Fetch Sequence

- Address of next instruction is in PC
- Address (MAR) is placed on address bus
- Control unit issues READ command
- Result (data from memory) appears on data bus
- Data from data bus copied into MBR
- PC incremented by 1 (in parallel with data fetch from memory)
- Data (instruction) moved from MBR to IR
- MBR is now free for further data fetches

Fetch Sequence (symbolic)

(tx = time unit/clock cycle)

t1: MAR ← PC
t2: MBR ← (memory or MAR)
t3: PC ← PC +1
IR ←MBR

OR

t1: MAR ← PC
t2: MBR ← (memory or MAR)
PC ← PC +1
t3: IR ←MBR

Control Unit

The control unit is the heart of CPU. It gets instruction from memory. The control unit decides what the instructions mean and directs the necessary data to be moved from memory to ALU. It must communicate with both ALU and main memory. It coordinates all activities of processor unit, peripheral devices and storage devices. Two types of control unit can be implemented in computing systems.

1. **Hardwired control unit**
   - This control unit is essentially a combinatorial circuit. It's input logic signals are transformed into set of output logic signals which are control signals.
   - It performs different operations on the basis of op-codes.
   - We have to derive the Boolean expression for each control signal as a function of input.
   - Since modern processor needs a Boolean equation, it is very difficult to build a combinational circuit that satisfies all these operations.
   - It has faster mode of operation.
   - A hardwired control unit needs rewiring if design has to be modified.
2. **Micro-programmed control unit**
   - An alternative to hardwired CU.
   - In micro-programmed control unit, the control information is stored in control memory.
   - The control memory is programmed to initiate required sequence of operations.
   - Use sequences of instructions to perform control operations performed by micro operations.
   - Control address register contains the address of the next microinstruction to be read

- As it is read, it is transferred to control buffer register.
- Sequencing unit loads the control address register and issues a read command.
- It is cheaper and simple than hardwired CU.
- It is slower than hardwired CU.

Introduction to Register Transfer Language (RTL)

The symbolic notation used to describe the micro operation transfers among register is called register transfer language. It is one of the forms of hardware description language (HDL). The term 'register transfer' implies the availability of hardware logic circuits that can perform a stated instruction and transfer the data. It also transfers result of the operation to the same or another register. The term 'language' is borrowed from programmers, who apply this term to programming language.

RTL is the convenient tool for describing the internal organization of digital computers in concise and precise manner. It can also be used to facilitate the design process of digital systems such as microprocessors.

Fetch and Execute Cycle of MOV A, B in terms of RTL Specification:

Within the fetch cycle, the operations performed during execution of instruction MOV A, B are:

i. The program counter contains the address of the next instruction to be executed. If the next instruction to be executed is MOV A, B; the program counter contains the address of the memory location where the instruction code for MOV A, B resides.

   In the first operation of fetch cycle, the contents of program counter will be transferred to the memory address register (MAR). The memory address register then uses the address bus to transmit its contents that specifies the address of memory location from where that instruction code of MOV

A, B is to be fetched. Let t1 indicates the period of first operation

t_1 : MAR ← PC

ii. When the control unit issues the memory read signal, the contents of the address memory location specified by MAR will be transferred to the memory buffer register (MBR). Suppose t2 is the time period for this operation.

t_2 : MBR ← Memory or [MAR]

iii. Finally the contents of MBR will be transferred to the instruction register and then the program counter gets incremented. Let t3 be the time required by the CPU to complete these operations.

t_3 : IR ← (MBR)

PC ← PC + 1

After the fetch cycle completed, the execution starts. The execute cycle steps are described as follows:

i. At the start of execution cycle, the instruction register (IR) consists of instruction code for instruction MOV A, B. The address field of instructions specifies the addresses of the two memory locations A & B. The first step needed is to obtain the data from the location B. For this, the address field of IR indicating the address of memory location will be transferred to address bus through the MAR. Let t1 be this time taken.

t_1 : MAR ← (IR(Address of B))

ii. When the control unit issues a memory read signal, the contents of location B will be output (written) to the memory buffer register (MBR). Now the content of B which is to be written to memory location A is contained in MBR. Let t2 be the time taken for that operation.

t_2 : MBR ← (B)

iii. Now, we need the memory location of A because it is being written with the data of location B. For this the address field

of IR indicating the address of memory location A. A will be transferred to MAR in time t3.

t_3 : MAR ← (IR(Address of A))

iv. When the control unit issues the memory write signal, the contents of MBR will be written to the memory location indicated by the contents of MAR in time t4.

t_4 : A ← MBR or t_4 : [MAR] ← MBR

Note: [MAR] = A

Program consists of instructions which contains different cycles like fetch and execute. These cycles in turn are made up of the smaller operation called micro operations.

Some RTL Examples

1. MVI A, 02H

Fetch:

t1: MAR ← PC

t2: MBR ← [MAR]

t3: IR ← MBR

PC ← PC + 1

Execute:

t4: MBR ← IR [address of immediate data]

t5: MAR ← IR [address of A]

t6: A ← MBR

2. LXI B, 0210H

Fetch:

t1: MAR ← PC

t2: MBR ← [MAR]

t3: IR ← MBR

PC ← PC + 1

Execute:

t4: MBR ← IR [address of immediate data]

t5: MAR ← IR [address of C]

t6: C ← MBR

t7: MBR ← IR [address of immediate data (MSB)]
t8: MAR ← IR [address of (B)
t9: B ← MBR

3. **LDA 2030H**

Fetch:
t1: MAR ← PC
t2: MBR ← [MAR]
t3: IR ← MBR
PC ← PC + 1

Execute:
t4: MAR ← IR [address of immediate data]
t5: MBR ← IR [address of C]
t6: MAR ← IR [address of A]
t7: A ← MBR

4. **STA 2030H**

Fetch:
t1: MAR ← PC
t2: MBR ← [MAR]
t3: IR ← MBR
PC ← PC + 1

Execute:
t4: MAR ← IR [address of immediate A]
t5: MBR ← A
t6: MAR ← IR [address of immediate data]
t7: [MAR] ← MBR

Advantages of Microprocessors:

- Computational/processing speed is high
- Intelligence has been brought to systems
- Automation of industrial process and office automation
- Flexible
- Compact in size
- Maintenance is easier

Applications of Microprocessors:

- Microcomputer: Microprocessor is the CPU of the microcomputer.
- Embedded system: Used in microcontrollers.
- Measurements and testing equipment: used in signal generators, oscilloscopes, counters, digital voltmeters, x-ray analyzer, blood group analyzers, baby incubator, frequency synthesizers, data acquisition systems, spectrum analyzers, etc.
- Scientific and engineering research.
- Industry: used in data monitoring system, automatic weighting, batching systems, etc.
- Security systems: smart cameras, CCTV, smart doors, etc.
- Automatic system
- Communication system
- Games machine
- Accounting system
- Complex industrial controllers
- Data acquisition system
- Military applications system

STA
T_1: M
T_2: unspecified
Execute:
M_1: T_5: MAR ← PC
T_6: MBR ← [MAR]
T_7: Z ← MBR, PC ← PC+1
M_2: T_8: MAR ← PC
T_9: MBR ← [MAR]
T_{10}: W ← MBR, PC ← PC+1
M_3: T_{11}: MAR ← W Z
T_{12}: MAR ← A
T_{13}: [MAR] ← MBR
PC = PC+1

ADDITIONAL QUESTIONS

1. **Write fetch and execute cycles for LXI B, 8400 in terms of RTL specifications.** *[2062 Bhadra]*

⇒ **LXI B, 8400 H**

Assume the given instruction resides at memory location C050H.

C050: op-code (LXI B)
C051: 00 H
C052: 84 H

Fetch:
t_1: MAR ← PC
t_2: MBR ← [MAR]
t_3: IR ← MBR, PC←PC+1
t_4:Unspecified

Execute1:
t_5: MAR ← PC
t_6: MBR ← [MAR]
t_7: C ← MBR, PC ← PC+1

Execute2:
t_8: MAR ← PC
t_9: MBR ← [MAR]
t_{10}: B ← MBR, PC ← PC+1

2. **What could be the register transfer statements for ADD B?** *[2067 Shrawan]*

⇒ **RTL for ADD B**

Fetch:
t1: MAR ← PC
t2: MBR ← [MAR]
t3: IR ← MBR, PC ← PC + 1

Execute:
t4: MAR ← (IR (Address of B))
t5: MBR ← (B)

t6: MAR ← (IR (Address of A))
t7: (A) ← (A) + MBR

3. **Differentiate microprocessors and microcontrollers.** *[2065 Chaitra]*

⇒

Microprocessor	Microcontroller
1. It is an IC which has only the CPU (ALU, RU, and CU). It doesn't have RAM, ROM, and other peripherals on the chip.	1. It is an IC which has microprocessor, memory, and I/O signal lines on a single chip.
2. It is comparatively larger in size and is costlier.	2. It is of small size and costs cheaper.
3. Microprocessor is used in personal computers and other general purpose applications.	3. Microcontroller is designed for specific embedded applications.

❖ ❖ ❖

Chapter – 2

PROGRAMMING WITH 8085 MICROPROCESSOR

Internal Architecture of an 8-bit Microprocessor and its Registers

The Intel 8085A is a complete 8-bit parallel central processing unit. The main components of 8085A are array of registers, the arithmetic logic unit, the encoder/decoder, and timing and control circuits linked by an internal data bus.

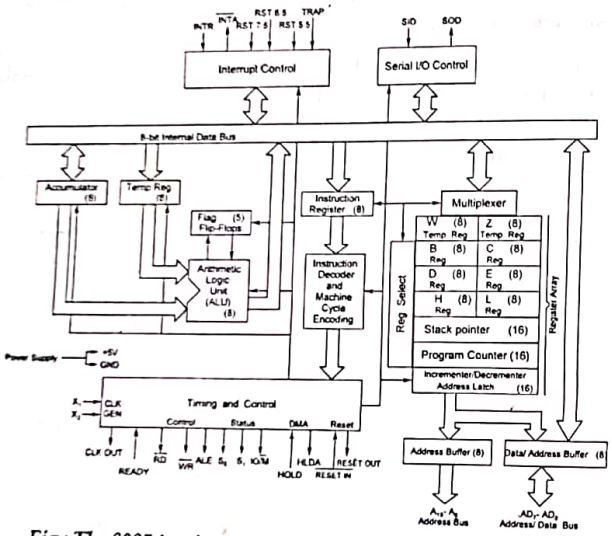


Fig.: The 8085A microprocessor functional block diagram

Source: Intel Corporation. *Embedded Microprocessors (Santa* Clara. Calif: Author.1994), pp 1-11

1. ***ALU:*** The arithmetic logic unit performs the computing functions. It includes the accumulator, the temporary register, the arithmetic and logic circuits and five flags. The temporary register is used to hold data during an arithmetic/logic operation. The result is stored in the accumulator; the flags (flip-flops) are set or reset according to the result of the operation.

2. **Accumulator (Register A):** It is an 8 bit register that is the part of ALU. This register is used to store the 8-bit data and to perform arithmetic and logic operations. 8085 microprocessor is called accumulator based microprocessor. When data is read from input port, it is first moved to accumulator and when data is sent to output port, it must be first placed in accumulator.

3. **Temporary Registers (W and Z):** They are 8 bit registers not accessible to the programmer. During program execution, 8085A places the data into it for a brief period.

4. **Instruction Register (IR):** It is an 8 bit register not accessible to the programmer. It receives the operation codes of instruction from internal data bus and passes to the instruction decoder which decodes so that microprocessor knows which type of operation is to be performed.

5. **Register Array (Scratch pad registers B, C, D, E):** Each one (B, C, D, E) is an 8 bit register accessible to the programmers. Data can be stored upon it during program execution. These can be used individually as 8-bit registers or in pair BC, DE as 16 bit registers. The data can be directly added or transferred from one to another. Their contents may be incremented or decremented and combined logically with the content of the accumulator.

6. **Register H & L:** They are 8 bit registers that can be used in same manner as scratch pad registers.

7. **Stack Pointer (SP):** It is 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.

8. **Program Counter (PC):** Microprocessor uses the PC register to sequence the execution of the instructions. The function of

PC is to point to the memory address from which the next byte is to be fetched. When a byte is being fetched, the PC is incremented by one to point to the next memory location.

9. **Flags:**

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

Register consists of five flip flops, each holding the status of different states separately is known as flag register and each flip flop are called flags. 8085A can set or reset one or more of the flags. The flags are sign (S), Zero (Z), Auxiliary Carry (AC), Parity (P), and Carry (CY). The state of flags indicates the result of arithmetic and logical operations, which in turn can be used for decision making processes. The different flags are described as:

- **Carry:** It stores the carry or borrow from one byte to another. If the last operation generates a carry or borrow, its status will be 1 otherwise 0.
- **Zero:** If the result of last operation is zero, its status will be 1 otherwise 0. It is often used in loop control and in searching for particular data value.
- **Sign:** If the most significant bit (MSB) of the result of the last operation is 1 (negative), then its status will be 1 otherwise 0.
- **Parity:** If the result of the last operation has even number of 1's (even parity), its status will be 1 otherwise 0.
- **Auxiliary Carry:** If the last operation generates a carry from the lower half word (lower nibble), its status will be 1 otherwise 0.

10. **Timing and Control Unit:** This unit synchronizes all the microprocessor operations with the clock and generates the control signals necessary for communication between the microprocessor and peripherals. The control signals are similar to the sync pulse in an oscilloscope. The $\overline{RD}$ and

$\overline{WR}$ signals are sync pulses indicating the availability of data on the data bus.

11. **Interrupt Controls:** The various interrupt controls signals (INTR, RST 5.5, RST 6.5, RST 7.5, and TRAP) are used to interrupt a microprocessor.

12. **Serial I/O Controls:** Two serial I/O control signals (SID and SOD) are used to implement the serial data transmission.

Characteristics (Features) of 8085A Microprocessor and its Signals

The 8085A (commonly known as 8085) is an 8-bit general purpose microprocessor capable of addressing 64K of memory. The device has 40 pins, requires a +5V single power supply, and can operate with a 3-MHZ single phase clock.

1. **Address Bus:** The 8085 has 16 signal lines that are used as the address bus; however, these lines are split into two segments A_{15}–A_8 and AD_7–AD_0. The eight signals A_{15}–A_8 are unidirectional and used as higher order bus.

2. **Data Bus:** The signal lines AD_7– AD_0 are bidirectional; they serve a dual purpose. They are used as lower order address bus as well as data bus.

3. **Control and Status Signals:** This group of signals includes two control signals ($\overline{RD}$ and $\overline{WR}$), three status signals (IO/ $\overline{M}$, S_1 and S_0) to identify the nature of the operation, and one special signal (ALE) to indicate the beginning of the operation.
   - **ALE (Address Latch Enable):** This is a positive going pulse generated every time 8085 begins an operation (machine cycle). It indicates that the bits AD_7–AD_0 are address bits. This signal is used primarily to latch the low-order address from the multiplexed bus and generate a separate set of eight address lines A_7–A_0.
   - $\overline{RD}$ **(Read):** This is a read control signal (active low). This signal indicates that the selected I/O or memory

device is to be read and data are available on the data bus.

- $\overline{WR}$ (Write): This is a write control signal (active low). This signal indicates that the data on the data bus are to be written into a selected memory or I/O location.
- IO/$\overline{M}$: This is a status signal used to differentiate between I/O and memory operations. When it is high, it indicates an I/O operation; when it is low, indicates a memory operation. This signal is combined with $\overline{RD}$ (Read) and $\overline{WR}$ (Write) to generate I/O and memory signals.
- S_1 and S_0 : These status signals, similar to IO/$\overline{M}$, can identify various operations, but they are rarely used in small systems.

4. Power Supply and Clock Frequency:
   - VCC: +5V power supply
   - VSS: Ground reference
   - X1 and X2: A crystal (RC or LC network) is connected at these two pins for frequency.
   - CLK OUT: It can be used as the system clock for other devices.
5. Externally Initiated Signals:
   - INTR (Input): Interrupt request, used as a general purpose interrupt.
   - $\overline{INTA}$ (Output): This is used to acknowledge an interrupt.
   - RST 7.5, 6.5, 5.5 (Inputs): These are vectored interrupts that transfer the program control to specific memory locations. They have higher priorities than INTR

interrupt. Among these three, the priority order is 7.5, 6.5, and 5.5.

- TRAP (Input): This is a non-maskable interrupt with highest priority.
- HOLD (Input): This signal indicates that a peripheral such as a DMA (Direct Memory Access) controller is requesting use of address and data bus.
- HLDA (Output): HLDA stands for Hold Acknowledge. This signal acknowledges the HOLD request
- READY (Input) : This signal is used to delay the microprocessor Read or Write cycles until a slow-responding peripheral is ready to send or accept data. When this signal goes low, the microprocessor waits for an integral number of clock cycles until it goes high.
- $\overline{RESET\ IN}$: When the signal on this pin goes low, the program counter is set to zero, the buses are tri-stated, and MPU is reset.
- RESET OUT: This signal indicates that the MPU is being reset. The signal can be used to reset other devices.
- Serial I/O Ports: The 8085 has two signals to implement the serial transmission: SID (Serial Input Data), and SOD (Serial Output Data). In serial transmission, data bits are sent over a single line, one bit at a time, such as the transmission over telephone lines.

Instruction Description and Format

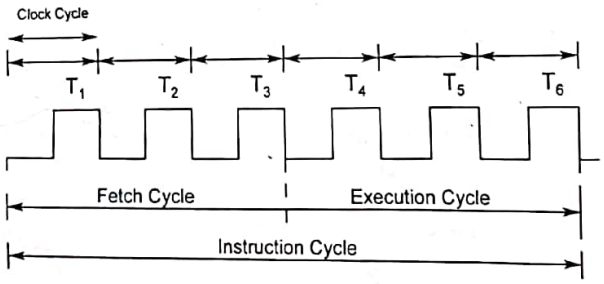
The computer can be used to perform a specific task, only by specifying the necessary steps to complete the task. The collection of such ordered steps forms a 'program' of a computer. These ordered steps are the instructions. Computer instructions are stored in central memory locations and are executed sequentially, one at a time. The control reads an instruction from a specific address in memory and executes it. It then continues by reading

the next instruction in sequence and executes it until the completion of the program.

Instruction cycle

Instruction contained in the program is pointed by the program counter. Instruction is first moved to the instruction register and is decoded in binary form and stored as an instruction in the memory. The computer takes a certain period to complete this task i.e., instruction fetching, decoding and executing on the basis of clock speed. Such time period is called 'instruction cycle' and consists two cycles namely fetch and decode, and execute cycle.

In the fetch cycle, the central processing unit obtains the instruction code from the memory for its execution. Once the instruction code is fetched from memory, it is then executed. The execution cycle consists the calculating the address of the operands, fetching them, performing operations on them and finally outputting the result to a specified location.



Instruction format:

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

The op-code field specifies how data is to be manipulated and address field indicates the address of a data item. For example,

ADD R_1, R_0
op-code *address*

Here, R_0 is the source register and R_1 is the destination register. The instruction adds the contents of R_0 with the content of R_1 and stores result in R_1.

8085A can handle at the maximum of 256 ($=2^8$) instructions; however, only 246 instructions are used in 8085A. The sheet which contains all these instructions with their hex code, mnemonics, descriptions and function is called an instruction sheet. Depending on the number of address specified in instruction sheet, the instruction format can be classified into the categories.

- **One address format (1 byte instruction):** Here, 1 byte will be op-code and operand will be default.

  E.g. ADD B, MOV A,B

- **Two address format (2 byte instruction):** Here, first byte will be op-code and second byte will be the operand/data.

  E.g. IN 40H; MVI A, 8-bit data

- **Three address format (3 byte instruction):** Here, first byte will be op-code, second and third byte will be operands/data. That is,

  2nd byte- lower order data.
  3rd byte – higher order data
  E.g. LXI B, 4050 H
  ↳ load register pair with 16 bit data address.

Classification of an Instruction

An instruction is a binary pattern designed inside a microprocessor to perform a specific function (task). The entire group of instructions called the instruction set. The 8085 instruction set can be classified in to 5- different groups.

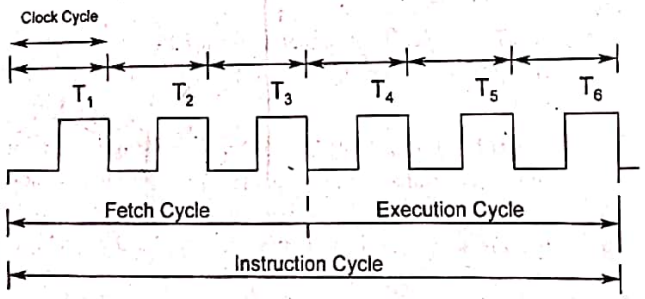
- **Data transfer group:** The instructions which are used to transfer data from one register to another register or register to memory.

the next instruction in sequence and executes it until the completion of the program.

Instruction cycle

Instruction contained in the program is pointed by the program counter. Instruction is first moved to the instruction register and is decoded in binary form and stored as an instruction in the memory. The computer takes a certain period to complete this task i.e., instruction fetching, decoding and executing on the basis of clock speed. Such time period is called 'instruction cycle' and consists two cycles namely fetch and decode, and execute cycle.

In the fetch cycle, the central processing unit obtains the instruction code from the memory for its execution. Once the instruction code is fetched from memory, it is then executed. The execution cycle consists the calculating the address of the operands, fetching them, performing operations on them and finally outputting the result to a specified location.



Instruction format:

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

The op-code field specifies how data is to be manipulated and address field indicates the address of a data item. For example,

ADD R_1, R_0
op-code *address*

Here, R_0 is the source register and R_1 is the destination register. The instruction adds the contents of R_0 with the content of R_1 and stores result in R_1.

8085A can handle at the maximum of 256 ($=2^8$) instructions; however, only 246 instructions are used in 8085A. The sheet which contains all these instructions with their hex code, mnemonics, descriptions and function is called an instruction sheet. Depending on the number of address specified in instruction sheet, the instruction format can be classified into the categories.

- **One address format (1 byte instruction):** Here, 1 byte will be op-code and operand will be default.

  E.g. ADD B, MOV A,B

- **Two address format (2 byte instruction):** Here, first byte will be op-code and second byte will be the operand/data.

  E.g. IN 40H; MVI A, 8-bit data

- **Three address format (3 byte instruction):** Here, first byte will be op-code, second and third byte will be operands/data. That is,

  2nd byte- lower order data.
  3rd byte – higher order data
  E.g. LXI B, 4050 H

Classification of an Instruction

An instruction is a binary pattern designed inside a microprocessor to perform a specific function (task). The entire group of instructions called the instruction set. The 8085 instruction set can be classified in to 5- different groups.

- **Data transfer group:** The instructions which are used to transfer data from one register to another register or register to memory.

- **Arithmetic group:** The instructions which perform arithmetic operations such as addition, subtraction, increment, decrement, etc.
- **Logical group:** The instructions which perform logical operations such as AND, OR, XOR, COMPARE, etc.
- **Branching group:** The instructions which are used for looping and branching are called branching instructions like jump, call, etc.
- **Miscellaneous group:** The instructions relating to stack operation, controlling purposes such as interrupt operations fall under miscellaneous group including machine control instructions like HLT, NOP.

Data transfer group instructions

It is the longest group of instructions in 8085. This group of instructions copy data from a source location to destination location without modifying the contents of the source. The transfer of data may be between the registers or between register and memory or between an I/O device and accumulator. None of these instructions changes the flag. The instructions of this group are:

1. **MOV R_d, R_s (Move register instruction)**
   - 1 byte instruction
   - Copies data from source register to destination register.
   - R_d & R_s may be A, B, C, D, E, H, and L

   E.g. MOV A, B

2. **MVI R, 8 bit data (Move immediate instruction)**
   - 2 byte instruction
   - Loads the second byte (8 bit immediate data) into the register specified.
   - R may be A, B, C, D, E, H, and L

   E.g. MVI C, 53H ; C ← 53H

3. **MOV M, R (Move to memory from register)**
   - Copies the contents of the specified register to memory. Here, memory is the location specified by the contents of HL register pair.

E.g. MOV M, B

4. **MOV R, M (Move to register from memory)**
   - Copies the contents of memory location as specified by HL register pair to a register.

   E. g. MOV B, M

6 **Write a program to load memory locations 7090 H and 7080 H with data 40H and 50H and then swap these data.**

```
MVI H, 70H
MVI L, 90H
MVI A, 40H
MOV M, A
MOV C, M
MVI L, 80H
MVI B, 50H
MOV M, B
MOV D, M
MOV M, C
MVI L, 90H
MOV M, D
HLT
```

5. **LXI R_P, 2 bytes data (Load register pair)**
   - 3-byte instruction
   - Load immediate data to register pair
   - Register pair may be BC, DE, HL, and SP

   1st byte: op-code
   2nd byte: lower order data
   3rd byte: higher order data

   E.g. LXI B, 4532H; B ← 45, C ← 32H

6. **MVI M, data (Load memory immediate)**
   - 2 byte instruction.
   - Loads the 8-bit data to the memory location whose address is specified by the contents of HL pair.

   E.g. MVI M , 35H; [HL] ← 35H

7. **LDA 16-bit address (Load accumulator direct)**
   - 3-byte instruction
   - Loads the accumulator with the contents of memory location whose address is specified by 16 bit address.

   E.g. LDA 4035H; A ← [4035H]

8. **LDAX R_P (Load accumulator indirect)**
   - 1 byte instruction
   - Loads the contents of memory location pointed by the contents of register pair to accumulator.

   E. g. LDAX B; [A] ← [[BC]]
   LXI B, 9000H; B= 90, C= 00
   LDAX B; A ← [9000]

9. **STA 16-bit address (Store accumulator contents direct)**
   - 3 byte instruction
   - Stores the contents of accumulator to specified address

   E.g. STA FA00H; [FA00] ← [A]

10. **STAX R_P (Store accumulator contents indirect)**
    - 1 byte instruction
    - Stores the contents of accumulator to memory location specified by the contents of register pair.

    E. g. STAX B ; [BC] ← A

11. **IN 8-bit address**
    - 2-byte instruction
    - Reads data from the input port address specified in the second byte and loads data into the accumulator i.e. input port to accumulator.

    E. g. IN 40H ; A ← [40H]

12. **OUT 8-bit address**
    - 2 byte instruction
    - Copies the contents of the accumulator to the output port address specified in the 2nd byte i.e. accumulator to output port.

    E. g. OUT 40H ; [40] ← A

13. **LHLD 16-bit address (Load HL directly)**
    - 3 byte instruction
    - Loads the contents of specified memory location to L-register and contents of next higher location to H-register.

    E.g. LXI H, 9500H
    MVI M, 32H; 9500
    MVI L, 01H; 9501
    MVI M, 7AH
    LHLD 9500H; H=7A, L=32

32
7A

14. **SHLD 16-bit address (Store HL directly)**
    - Opposite to LHLD.
    - Stores the contents of L-register to specified memory location and contents of H-register to next higher memory location.

    E .g. LXI H, 9500H
    SHLD 8500H; [8500] = 00, [8501] = 95

15. **XCHG (Exchange)**
    - Exchanges DE pair with HL pair.

    E. g. LXI H, 7500H; H= 75, L=00
    LXI D, 9532H; D=95, E=32
    XCHG ; H=95, L=32 ; D=75, E=00

Addressing Modes

Instructions are command to perform a certain task in microprocessor. The instruction consists of op-code and data called operand. The operand may be the source only, destination only or both of them. In these instructions, the source can be a register, a memory, or an input port. Similarly, destination can be a register, a memory location, or an output port. The various formats (ways) of specifying the operands are called addressing modes. So,

addressing mode specifies where the operands are located rather than their nature. The 8085 has 5 addressing modes.

1. **Direct addressing mode:** The instruction using this mode specifies the effective address as part of instruction. The instruction size either 2-bytes or 3-bytes with first byte op-code followed by 1 or 2 bytes of address of data.

   E. g. LDA 9500H; A ← [9500]
   IN 80H ; A ← [80]

   This type of addressing is also called absolute addressing.

2. **Register direct addressing mode:** This mode specifies the register or register pair that contains the data.

   E g. MOV A, B

   Here, register B contains data rather than address of the data.

   Other examples are ADD, XCHG, etc.

3. **Register indirect addressing mode:** In this mode, the address part of the instruction specifies the memory whose contents are the address of the operand. So, in this type of addressing mode, it is the address of the address rather than address itself.

   E. g. STAX B, LDAX D etc.

4. **Immediate addressing mode:** In this mode, the operand position is the immediate data. For 8-bit data, instruction size is 2 bytes and for 16 bit data, instruction size is 3 bytes.

   E.g. MVI A, 32H
   LXI B, 4567H

5. **Implied or inherent addressing mode:** The instructions of this mode do not have operands.

   E.g. NOP ; No operation
   HLT ; Halt
   EI ; Enable interrupt
   DI ; Disable interrupt

Arithmetic group instructions:

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

1. **ADD R/M**
   - 1 byte instruction
   - Adds the contents of register/memory to the contents of the accumulator and stores the result in accumulator.

   E. g. Add B; A ← [A] + [B]

2. **ADI 8-bit data**
   - 2 byte instruction
   - Adds the 8 bit data with the contents of accumulator and stores result in accumulator.

   E g. ADI 9BH ; A← A+9BH

3. **SUB R/M**
   - 1 byte instruction
   - Subtracts the contents of specified register /memory with the contents of accumulator and stores the result in accumulator.

   E. g. SUB D ; A ← A – D

4. **SUI 8-bit data**
   - 2 byte immediate instruction.
   - Subtracts the 8 bit data from the contents of accumulator stores result in accumulator.

   E. g. SUI D3H; A ← A – D3H

5. **INR R/M, DCR R/M**
   - 1 byte instructions.
   - INR R/M and DCR R/M increase and decrease the contents of R(register) or M(memory) by 1 respectively.

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

- All flags are affected except carry.

6. **INX Rp, DCX RP**

- 1 byte instructions
- Increase and decrease the register pair by 1.
- Acts as 16 bit counter made from the contents of 2 registers

E.g. INX B ; BC=BC+1
DCX D ; DE=DE+1

- No flag is affected

7. **ADC R/M and ACI 8-bit data**

- **ADC R/M** adds the contents of register/memory with the content of accumulator using previous carry. It is 1 byte instruction.

E.g. ADC B ; A ←A+B+CY

- **ACI 8-bit data** adds the 8-bit data with the content of accumulator using previous carry. It is 2 byte instruction.

E.g. ACI 70H ; A ←A + 70+CY

8. **SBB R/M and SBI 8-bit data**

- **SBB R/M** subtracts the contents of register/memory from the content of accumulator using previous borrow. It is 1 byte instruction.

E. g. SBB D ; A ←A-D-Borrow

- **SBI 8-bit data** subtracts the 8-bit data from the content of accumulator using previous borrow. It is 2 byte instruction.

E.g. SBI 70H ; A ←A-70-Borrow

9. **DAD Rp (Double addition)**

- 1 byte instruction
- Adds register pair with HL pair and stores the 16 bit result in HL pair.

E. g. LXI H, 7320H
LXI B, 4220H
DAD B; HL = HL + BC
7320 + 4220 = B540

10. **DAA (Decimal adjustment accumulator)**

- Used only after addition
- 1 byte instruction
- The content of accumulator is changed from binary to two 4-bit BCD digits.

E. g MVI A, 78H ; A=78
MVI B, 42H ; B=42
ADD B ; A=A+B = BA
DAA ; A=20, CY=1

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

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

But the INR and DCR operations can be performed in any register or memory. These instructions

1. Affect the contents of specified register or memory.
2. Affect the flag except carry flag.

Addition operation in 8085:

8085 performs addition with 8-bit binary numbers and stores the result in accumulator. If the sum is greater than 8-bits (FFH), it sets the carry flag.

E.g. MVI A, 93H
MVI C, B7H
ADD C

B7H:	1011	0111
+93H:	1001	0011
[1] 4A	[1] 0100	1010
CY	CY	4AH

Subtraction operation in 8085:

8085 performs subtraction operation by using 2's complement and the steps used are:

1. Converts the subtrahend (the number to be subtracted) into its 1's complement.
2. Adds 1 to 1's complement to obtain 2's complement of the subtrahend.
3. Adds 2's complement to the minuend (the contents of the accumulator).
4. Complements the carry flag.

e.g MVI A, 97 H:
MVI B, 65 H:
SUB B

	0110 0101
	1001 1010
2's comp.:	1001 1011
97 :	+1001 0111
	[1] 0011 0010

Complement carry = 0
Accumulator = 32 H

The memory location 2050H holds the data byte F7H. Write instructions to transfer the data byte to accumulator using different op-codes: MOV, LDAX and LDA.

Using MOV

```
LXI H, 2050H
MOV A, M
```


Using LDAX

```
LXI B, 2050H
LDAX B
```

Using LDA

```
LDA 2050H
```

Register B contains 32H, Use MOV and STAX to copy the contents of register B in memory location 8000H.

Using MOV

```
LXI H, 8000H
MOV M, B
```

Using STAX

```
LXI D, 8000H
MOV A, B
STAX D
```

The accumulator contains F2H, Copy A into memory 8000H. Also copy F2H directly into 8000H.

```
STA 8000H
LXI H, 8000H
MVI M, F2H
```

The data 20H and 30H are stored in 2050H and 2051H. WAP to transfer the data to 3000H and 3001H using LHLD and SHLD instructions.

```
MVI A, 20H
STA 2050H
MVI A, 30H
STA 2051H
LHLD 2050H
SHLD 3000H
HLT
```

Pair B contains 1122H and pair D contains 3344H. WAP to exchange the contents of B and D pair using XCHG instruction.

```
LXI B, 1122H ;     B=11, C=22   } loading in B & D register pair
LXI D, 3344H ;     D=33, E=44
MOV H, B
MOV L, C           } moving to HL pair, because XCHG works only with HL & DE pair
XCHG               ; Exchange DE pair with HL pair
MOV B, H
MOV C, L
HLT
```

WAP to add two 4 digit BCD numbers equals 7342 and 1989 and store result in BC register.

```
LXI H, 7342H
LXI B, 1989H
MOV A, L
ADD C
DAA
MOV C, A
MOV A, H
ADC B
DAA
MOV B, A
```

Register BC contain 2793H and register DE contain 3182H. Write instruction to add these two 16 bit numbers and place the sum in memory locations 2050H and 2051H. What is DAA instruction? Explain its purpose with an example.

[Back Paper 2062]

```
MOV A, C
ADD E
MOV L, A
MOV A, B
ADC D
MOV H, A
SHLD 2050H
HLT
```

Register BC contains 8538H and register DE contain 62A5H. Write instructions to subtract the contents of DE from the contents of BC and place the result in BC.

```
MOV A, C
SUB E
MOV C, A
MOV A, B
SBB D
MOV B, A
HLT
```

BCD Addition:

In many applications, data are presented in decimal number. In such applications, it may be convenient to perform arithmetic operations directly in BCD numbers.

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.

E.g.

```
A: 0000  1010
+ 0000   0110
  0001   0000  →  10BCD
```

A special instruction called DAA performs the function of adjusting a BCD sum in 8085. It uses the AC flag to sense that the value of the least four bits is larger than 9 and adjusts the bits to

BCD value. Similarly, it uses CY flag to adjust the most significant four bits.

E.g. Add BCD 77 and 48

```
77H     0111 0111
+48H   +0100 1000
125H    1011 1111
            +0110
          1 0101
              5
       +0110
        0010
     1   2
```

1 0010 0101 → 125_{BCD}

Logical group instructions:

A microprocessor is basically a programmable logic chip. It can perform all the logic functions of the hardwired logic through its instruction set. The 8085 instruction set includes such logic functions as AND, OR, XOR and NOT (Complement):

The following features hold true for all logic instructions:

1. The instructions implicitly assume that the accumulator is one of the operands.
2. All instructions reset (clear) carry flag except for complement where flag remain unchanged.
3. They modify Z, P, and S flags according to the data conditions of the result.
4. Place the result in the accumulator.

They do not affect the contents of the operand register.

The logical operations have the following instructions.

1. **ANA R/M**
   - Logically AND the contents of register/memory with the contents of accumulator.

   - 1 byte instruction.
   - CY flag is reset, AC is set and others as per result.
2. **ANI 8-bit data**
   - Logically AND 8-bit immediate data with the contents of accumulator.
   - 2 byte instruction.
   - CY flag is reset, AC is set and others as per result.
3. **ORA R/M**
   - Logically OR the contents of register/memory with the contents of accumulator.
   - 1 byte instruction.
   - CY and AC are reset and others as per result.
4. **ORI 8-bit data**
   - Logically OR 8 bit immediate data with the contents of the accumulator.
   - 2 byte instruction.
   - CY and AC are reset and others as per result.
5. **XRA R/M**
   - Logically exclusive OR the contents of register memory with the contents of accumulator.
   - 1 byte instruction.
   - CY and AC are reset and others as per result.
6. **XRI 8-bit data**
   - Logically exclusive OR 8 bit data immediate with the content of accumulator.
   - 2 byte instruction.
   - CY and AC are reset and others as per result.
7. **CMA (Complement accumulator)**
   - 1 byte instruction.
   - Complements the contents of the accumulator.
   - No flags are affected.

8. **Logically compare instructions**

CMP R/M (1 byte instruction)

CPI 8 bit data (2 byte instruction)

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

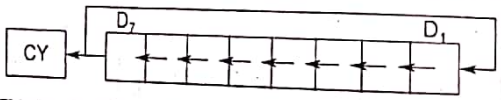
Case	CY	Z
[A]<[R/M] or 8 bit	1	0
[A]=[R/M] or 8 bit	0	1
[A]>[R/M] or 8 bit	0	0

9. **Logically rotate instructions**

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

i. **RLC (Rotate accumulator left)**

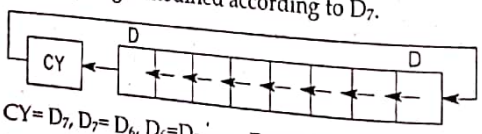
Each bit is shifted to the adjacent left position. Bit D_7 becomes D_0. The carry flag is modified according to D_7.



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

ii. **RAL (Rotate accumulator left through carry)**

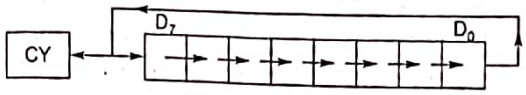
Each bit is shifted to the adjacent left position. Bit D_7 becomes the carry bit and the carry bit is shifted into D_0. The carry flag is modified according to D_7.



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

iii. **RRC (Rotate accumulator right)**

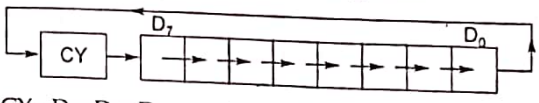
Each bit is shifted right to the adjacent position. Bit D_0 becomes D_7. The carry flag is modified according to D_0.



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

iv. **RAR (Rotate accumulator right through carry)**

Each bit is shifted right to the adjacent position. Bit D_0 becomes the carry bit and the carry bit is shifted into D_7. The carry flag is modified according to D_0.



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

10. **Additional logical instructions**

- CMC (Complement carry): It complements the carry flag.
- STC (Set carry flag): It sets the carry flag.
- CMA (Complement accumulator): It complements the content of accumulator.

Branching group instructions

The microprocessor is a sequential machine; it executes machine codes from one memory location to the next. The branching instructions instruct the microprocessor to go to a different memory location and the microprocessor continues executing machine codes from that new location.

The branching instructions are the most powerful instructions because they allow the microprocessor to change the sequence of a program, either unconditionally or under certain test conditions. The branching instruction code categorized in following three groups:

- Jump instructions
- Call and return instruction
- Restart instruction

i. Jump instructions

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

a. Unconditional jump

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

E.g. JMP 4000H

Here, 40H is higher order address and 00H is lower order address. The lower order byte enters first and then higher order.

The jump location can also be specified using a label or name.

E.g.

Using Label	Using Name
MVI A, 80H	START: IN 00H
OUT 43H	OUT 01H
MVI A, 00H	JMP START
L1: INR A	OUT 40H
JMP L1	HLT
HLT	

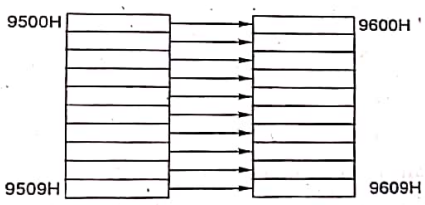
b. Conditional jump

The conditional jump instructions allow the microprocessor to make decisions based on certain conditions indicated by the flags. After logic and arithmetic operations, flags are set or reset to reflect the condition of data. These instructions check the flag

conditions and make decisions to change or not to change the sequence of program. The four flags namely carry, zero, sign and parity used by the jump instruction.

Mnemonics	Description
JC 16 bit	Jump on carry (if CY=1)
JNC 16 bit	Jump on if no carry (if CY=0)
JZ 16bit	Jump on zero (if Z=1)
JNZ 16bit	jump on if no zero (if Z=0)
JP 16bit	jump on positive (if S=0)
JM 16bit	jump on negative (if S=1)
JPE 16bit	Jump on parity even (if P=1)
JPO 16bit	Jump on parity odd (if P=0)

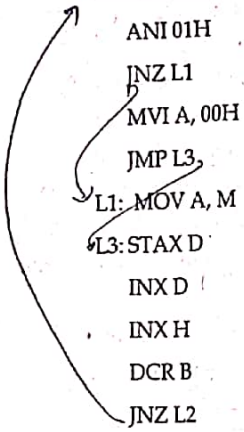
E.g. WAP to move 10 bytes of data from starting address 9500 H to 9600H



2000	MVI B, 0AH
2002	LXI H, 9500H
2005	LXI D, 9600H
2008	MOV A, M
2009	STAX D (1 byte)
200A	INX H
200 B	INX D
200 C	DCR B
200D	JNZ 2008
2010	HLT

Write to transfer 30 data starting from 8500 to 9500H if data is odd else store 00H.

```
    MVI B, 1EH
    LXI H, 8500H
    LXI D, 9500H
L2: MOV A, M
    ANI 01H
    JNZ L1          ; If data is odd, then go to L1.
    MVI A, 00H
    JMP L3
L1: MOV A, M
L3: STAX D
    INX D
    INX H
    DCR B
    JNZ L2
    HLT
```



ii. Call and return instructions (subroutine)

Call and return instructions are associated with subroutine technique. A subroutine is a group of instructions that perform a subtask. A subroutine is written as a separate unit apart from the main program and the microprocessor transfers the program execution sequence from main program to subroutine whenever it is called to perform a task. After the completion of subroutine task microprocessor returns to main program. The subroutine technique eliminates the need to write a subtask repeatedly, thus it uses memory efficiently. Before implementing the subroutine, the stack must be defined; the stack is used to store the memory address of the instruction in the main program that follows the subroutines call.

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

1. **CALL 16-bit address**
   - Call subroutine unconditionally.
   - 3 byte instruction.
   - Saves the contents of program counter on the stack pointer. - Loads the PC by jump address (16 bit memory) and executes the subroutine.
2. **RET**
   - Returns from the subroutine unconditionally
   - 1 byte instruction
   - Inserts the contents of stack pointer to program counter.
3. **CC/CNC/CZ/CNZ/CP/CM/CPE/CPO 16-bit address**
   - Call subroutine conditionally.
   - Same as CALL except that it executes on the basis of flag conditions.
4. **RC/RNC/RZ/RNZ/RP/RM/RPE/RPO**
   - Return subroutine conditionally.
   - Same as RET except that if executes on the basis of flag conditions.

Write an ALP to add two numbers using subroutines.

```
2000    MVI B, 4AH
2002    MVI C, A0H
2004    CALL 3000H    ; SP ← 2007H (PC), PC ← 3000H
2007    MOV B, A
2008    HLT
3000    MOV A, B
3001    ADD C
3002    RET           ; PC ← 2007H (SP)
```

- WAP to sort in ascending order for 10 bytes from 1120H.

```
START: LXI H, 1120H
       MVI D, 00H
       MVI C, 09H
L2:    MOV A, M
       INX H
       CMP M
       JC L1      ; if A<M
       MOV B, M
       MOV M, A
       DCX H
       MOV M, B
       INX H
       MVI D, 01H
L1:    DCR C
       JNZ L2
       MOV A, D
       RRC
       JC START
       HLT
```

iii. **Restart instruction**

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

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

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

Miscellaneous Group Instructions

STACK

The stack is defined as a set of memory location in R/W memory, specified by a programmer in a main memory. These memory locations are used to store binary information temporarily during the execution of a program.

The beginning of the stack is defined in the program by using the instruction LXI SP, 16 bit address. Once the stack location is defined, it loads 16 bit address in the stack pointer register. Storing of data bytes for this operation takes place at the memory location that is one less than the address e.g. LXI SP, 2099H

Here the storing of data bytes begins at 2098H and continuous in reverse order i.e 2097H. Therefore, the stack is initialized at the highest available memory location to prevent the program from being destroyed by the stack information. The stack instructions are:

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

   1 byte instruction.

   Copies the contents of specified register pair or program status word (accumulator and flag) on the stack.

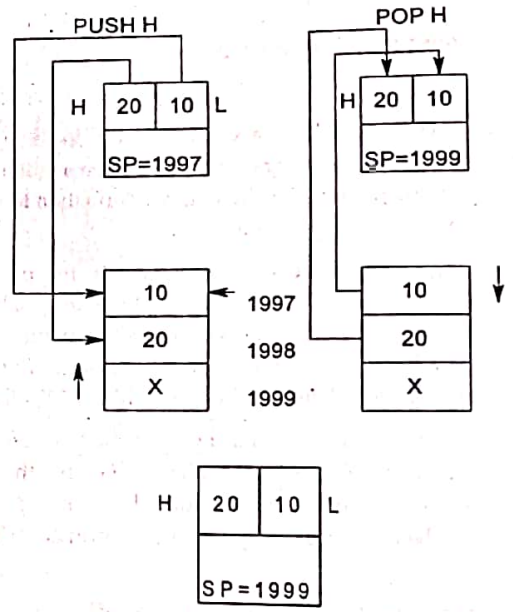
   Stack pointer is decremented and content of high order register is copied. Then it is again decremented and content of low order register is copied.

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

1 byte instruction.

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

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



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

Some instructions related to interrupt

DI – disable interrupt
EI – Enable interrupt
SIM – set interrupt mask
RIM – read interrupt mask

E.g.

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

Beore execution

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

After execution

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

Note: STACK Works in LIFO (Last In First Out) manner.

Counter:

It is designed simply by loading an appropriate number into one of the registers and using the INR or DCR instructions. A loop is established to update a count, and each count is checked to determine whether it has reached the final number, if not the loop is repeated.

Time Delay

When we use loop by counter, the loop causes the delay. Depending upon the clock period of the system, the time delay occurs during looping. The instructions within the loop use their own T-states. So, they need certain time to execute resulting delay.

Suppose, we have an 8085 microprocessor with 2 MHz clock frequency. Let's use the instruction MVI which takes 7 T-states.

Clock frequency of system (f) = 2 MHz

Clock period (T) = $1/f = ½ * 10^{-6} = 0.5\ \mu s$

Time to execute MVI = 7 T-states * 0.5 = 3.5 μs

E.g.	MVI C, FFH	7
	LOOP: DCR C	4
	JNZ LOOP	10/7

Here, register C is loaded with count FFH (255_{10}) by using MVI which takes 7 T-states.

Next, 2 instructions DCR and JNZ form a loop with a total of 14 (4+10) T-states. The loop is repeated 255 times until C=0. The time delay in loop (TC) with 2 MHz frequency is

T_l= (T * loop T- states * count)

Where,

T_l = time delay in loop

T= system clock period

Count = decimal value for counter

$T_l = 0.5 * 10^{-6} * 14 * 255 = 1785$ ms

But JNZ takes only 7 T-states when exited from loop i.e., last count = 0.50.

Adjusted loop delay is calculated as

$T_{la} = T_l - (3$ T-states) = 1785 ms – 3 ×0.5 μs = 1783.5 ms

Total delay loop of program is expressed as

T_D= Time to execute outside code + T_{la} inside loop

= 7 * 0.5 μs + 1783.5 ms = 1787 ms ≈ 1.8 ms

To increase the time delay beyond 1.8 ms for 2 MHz microprocessor, we need to use counter for register pair or loop within a loop.

		T-States	Clocks
	MVI B, 40H; 64	7	7×1
L2:	MVI C, 80H; 128	7	7×64
L1:	DCR C	4	4×128×64
	JNZ L1	10/7	(10×127+7×1) ×64
	DCR B	4	4×64
	JNZ L2	10/7	10×63+7×1
	RET	10	10×1

Total clocks =115854

For 2 MHz microprocessor, total time taken to execute above subroutine = 1158 ×0.5 μs = 57.927 ms

BCD to Binary Conversion

In most microprocessor-based products, data are entered and displayed in decimal numbers. For example, in an instrumentation laboratory, readings such as voltage and current are maintained in decimal numbers, and data are entered through decimal keyboard. The system monitors program of the instrument, converts each key into an equivalent 4-bit binary

number, and stores two BCD numbers in an 8-bit register or a memory location. These numbers are called packed BCD.

Conversion of BCD number into binary number employs the principle of positional weighting in a given number.

E.g. $72_{10} = 7 \times 10 + 2$

Converting an 8-bit BCD number into its binary equivalent requires the following steps:

- Separate an 8- bit packed BCD number into two 4 bit unpacked BCD digits i.e., BCD_1 and BCD_2.

  0111 0010 → 0000 0010 (02H) Unpacked BCD_1

  → 0000 0111 (07H) Unpacked BCD_2

- Convert each digit into its binary value according to its position.

  $BCD_1 = 02H$

  Multiply BCD_2 by $10 = 7 \times 10 = 70 = 46H$

- Add both binary numbers to obtain the binary equivalent of the BCD number.

  $02H + 46H = 48H$

WAP to read BCD number (Suppose 7010: 0111 0000BCD) stored at memory location 2020H and converts it into binary equivalent and finally stores that binary pattern into memory location 2030H.

```
LXI H, 2020H
MVI E, 0A H
MOV A, M        ; 0111 0010
ANI F0H         ; 0111 0000
RRC
RRC
RRC
RRC
```



```
MOV B,A
XRA A
L1: ADD B       ; 7× 10+2
DCR E
JNZ L1
MOV C, A
MOV A, M
ANI 0FH
ADD C
STA 2030H
HLT
```

Binary to BCD Conversion

If we need to convert a binary number into its equivalent BCD number, following steps can be sought.

Step 1: If binary number < 100 (64H), goto step 2

Else subtract 100 (64H) repetitively.

Quotient is BCD_1 (Divide by 100)

Step 2: If remainder from step 1 < 10 (0AH), goto step 3

Else subtract 10 (0AH) repetitively.

Quotient is BCD_2 (Divide by 10)

Step 3: Remainder from step 2 is BCD_3

E.g. $1111\ 1111_2$ *(FFH)* $= 255_{10} = 0010\ 0101\ 0101_{BCD}$

A binary number (Suppose FFH: 1111 11112) is stored in memory location 2020H. Convert the number into BCD and store each BCD as two unpacked BCD digits in memory location from 2030H.

```
LXI SP, 1999H
LXI H, 2020H;   Source
MOV A, M
```

```
        CALL PWRTEN
        HLT
PWRTEN:
        LXI H, 2030H;       Destination
        MVI B, 64H
        CALL BINBCD
        MVI B, 0AH
        CALL BINBCD
        MOV M, A
        RET
BINBCD:
        MVI M, FFH
NEXT:
        INR M
        SUB B
        JNC NEXT
        ADD B
        INX H
        RET
```

Binary to ASCII Conversion

Alphanumeric codes

A computer is a binary machine, to communicate with the computer in alphanumeric letters and decimal numbers, translation codes are necessary. The commonly used code is known as ASCII (American Standard Codes for Information Interchange). It is a 7-bit code with 128 combinations and each combination from 01H to 7FH is organized to a letter, decimal number, symbol or machine command. For example, 30H to 39H represents 0 to 9, 41H to 5AH represents A to Z, 21H to 2FH represents various symbols, and 61H to 7AH represents a to z.

General Letters/Numbers	ASCII (Hex)	ASCII (Decimal)
0 – 9	30 – 39	48 – 57
A – Z	41 – 5A	65 – 90
a – z	61 – 7A	97 – 122

The following simple algorithm can be implemented if we need to convert 8-bit binary number to ASCII

If number < 10, then add 30H

Else add 37H (30H + 07H)

For example: A = A + 30H + 07H = 41H

 # An 8 bit binary number is stored in memory location 1120H. WAP to store the ASCII codes of the binary digits in location 1160H and 1161H.

```
        LXI SP, 1999H
        LXI H, 1120H    ; Source
        LXI D, 1160H    ; Destination
        MOV A, M
        ANI F0H
        RRC
        RRC
        RRC
        RRC
        CALL ASCII
        STAX D
        INX H
        MOV A, M
        ANI 0FH
        CALL ASCII
        STAX D
        HLT
```

```
ASCII:   CPI 0AH
         JC BELOW
         ADI 07H
BELOW:   ADI 30H
         RET
```

ASCII to Binary Conversion

The following algorithm can be implemented for converting a number from ASCII to 8-bit binary.

Step 1: Subtract 30H

Step 2: If < 0AH, then binary as it is
Else subtract 07H
For example:
If ASCII is 41H, then 41H – 30H = 11H; 11H – 07H = 0AH

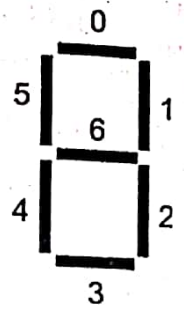
\# **WAP to convert ASCII code stored at memory location 1040H to binary equivalent and store the result at location 1050H.**

```
         LXI SP, 1999H
         LXI H, 1040H;  Source
         LXI D, 1050H;  Destination
         MOV A, M
         CALL ASCBIN
         STAX D
         HLT
ASCBIN:  SUI 30H
         CPI 0AH
         RC
         SUI 07H
         RET
```

BCD to 7 – Segment LED code Conversion:

- Each segment in a seven-segment display is identified by an index from 0 to 6, with the positions given in figure.
- For this conversion, Table Lookup Technique (TLT) is used.

- In TLT, the codes of digits to be displayed are stored sequentially in memory so that these codes can be used effectively and efficiently.



BCD Number	7-Segment Code
0	3FH
1	06H
2	5BH
3	4FH
4	66H
5	6DH
6	7DH
7	07H
8	7FH
9	6FH
Invalid	00H

\# **A set of three packed BCD numbers are stored in memory locations starting at 1150H. The seven segment codes of digits 0 to 9 for a common cathode LED are stored in memory locations starting at 1170H and the output buffer memory is reserved at 1190H. WAP to unpack the BCD number and select an appropriate seven segment code for each digit. The codes should be stored in output buffer memory.**

```
        LXI SP, 1999H
        LXI H, 1150H
        MVI D, 03H
        LXI B, 1190H
NEXT:   MOV A, M
        ANI F0H
        RRC
        RRC
        RRC
        RRC
        CALL CODE
        INX B
        MOV A, M
        ANI 0FH
        CALL CODE
        INX B
        INX H
        DCR D
JNZ NEXT
HLT
CODE:   PUSH H
        LXI H, 1170H
        ADD L
        MOV L, A
        MOV A, M
        STAX B
        POP H
        RET
```


A multiplicand is stored in memory location 1150H and a multiplier is stored in location 1151H. WAP to multiply these numbers and store result from 1160H.

```
        LXI SP, 1999H
        LHLD 1150H;     Two numbers
        XCHG;           Multiplier on D, multiplicand on E
        CALL MULT
        SHLD 1160H;     Store the product
        HLT
MULT:   MOV A, D
        MVI D, 00H
        LXI H, 0000H
        MVI B, 08H;     Counter
NEXT:   RAR;            Check multiplier bit is 1 or 0
        JNC BELOW
        DAD D;          Partial result in HL
BELOW:  XCHG
        DAD H;          Shift left multiplicand
        XCHG;           Retrieve shifted multiplicand
        DCR B
        JNZ NEXT
        RET
```

ADDITIONAL QUESTIONS

1. **Write a program for 8085 to change the bit D_5 of ten numbers stored at address 7600H if the numbers are larger than or equal to 80H.** *[2061 Ashwin]*

⇒

```
        LXI H, 7600H
        MVI C,0AH       ; COUNTER
LOOP1:  MOV A, M        ; A←[HL]
        CPI 80H
        JC NEXT
        XRI 20H
        MOV M, A
NEXT:   INX H
        DCR C
        JNZ LOOP1
        HLT
```

2. **Registers BC contain 2793H and registers DE contain 3182H. Write instructions to add these two 16 bit numbers and place the sum in memory locations 2050H and 2051H. What is DAA instruction? Explain its purpose with an example.** *[2062 Baishakh]*

⇒

```
XCHG    ; exchange H and L with D and E
DAD B   ; contents of BC are added to the contents of the
          HL register and the sum is saved in the HL
          register
SHLD 2050 ; the content of register L are stored in memory
          ;location specified by the 16- bit address in the
          ;operand and the contents of H register are
          stored
          ;in the next memory location by incrementing the
          ;operand
```

DAA (decimal – adjustment accumulator)

Description: the content of the accumulator are changed from a binary value to two 4 bit binary coded Decimal (BCD) digits

Example:

Add decimal 12_{BCD} to accumulator, which contains 39_{BCD}.

$$(A) = 39_{BCD} = 0\ 0\ 1\ 1\ \ 1\ 0\ 0\ 1$$
$$+\ 12_{BCD} = 0\ 0\ 0\ 1\ \ 0\ 0\ 1\ 0$$
$$51_{BCD}\quad 0\ 1\ 0\ 0\ \ 1\ 0\ 1\ 1$$
$$4 \qquad\qquad B$$

The binary sum is 4B. the value of the low- order four bits is larger than 9. Add 06 to the low order four bits

$$4B = 0\ 1\ 0\ 0\ \ 1\ 0\ 1\ 1$$
$$+06 = 0\ 0\ 0\ 0\ \ 0\ 1\ 1\ 0$$
$$1\ \ 1\ 1$$
$$51 = 0\ 1\ 0\ 1\ \ 0\ 0\ 0\ 1$$

3. **Write a program for 8085 to add ten 16-bit BCD numbers and store 24-bit BCD result at the end of the ten given numbers.** *[2062 Bhadra]*

⇒

```
        LXI H, 9000H ;Let 16 bit number starts from 9000H
        MVI C, 09H   ;set counter
        MVI B, 00H   ;to save carry over 16 bits
        MOV E, M     ;get LS byte of 16 bits data to E
        INX H        ;increment the memory pointer
        MOV D, M     ;get MS byte of the 16 bit data to D
REPEAT: INX H        ;increment the memory pointer
        MOV A,M      ;get LS Byte of NEXT data to A
        ADD E        ;add it with PREVIOUS data in E
        DAA
        MOV E,A
        INX H        ;increment the memory pointer
        MOV A,M      ;get MS Byte of NEXT data to A
        ADC D        ;add it with PREVIOUS data in D
        DAA
        MOV D,A
        JNC PASS
        MOV A,B
        ADI 01H      ;increment the carry by 1
```

```
        DAA
PASS:   DCR C           ;update counter
        JNZ REPEAT      ;continue for 10 words
        INX H           ;24 bit results in B, D, E respectively
        MOV M,E         ;storing final 24 bit sum at end of
table
        INX H
        MOV M,D
        INX H
        MOV M,B
        HLT
```

4. Write a program for 8085 to convert to convert and copy the lowercase ASCII codes to uppercase from memory location 9050H if any, otherwise copy as they are. Assume there are fifty codes in the source memory. [Note: ASCII Code for A=65....Z=90, a=97....z=122] *[2062 Bhadra]*

⇒

```
        LXI H, 9650H    ; loads memory in HL
        MVI C, OAH      ; counter.
LOOP1:  MOV A, M
        CPI  AO         ; CY→set→  data < AO → swap
                        ; notset data > AO → set
        JNC SETRSET
        CPI 71H
        JC SETRESET

        MOV D, A
        ANI 48H         ; A ←all bits zero except bit D3,D6
        CPI 48H         ; zero flag if D3 &D6 bits are 1
        JZ  LEAVE
        CPI 00H         ; zero flag if D3 &D6 bits are 1
        JZ LEAVE
        MOV A, D
        XRI 48 H        ; swaps bits of D3 & D6
        JMP  LEAVE
```



```
SETRESET: ORI 08H
          ANI BFH
LEAVE:    MOV M, A
          INX H
          DCR C
          JNZ LOOP1
          HLT
```

5. Write a program to transfer eight-bit numbers from 9080H to 9090H if bit D_5 is 1 and D_3 is 0. Otherwise transfer data by changing bit D_2 and D_6 from 1 to 0 or 0 to 1.Assume there are ten numbers. *[2064 Shrawan]*

⇒

```
          LXI B, 9080H   ; source
          LXI D, 9090H   ;destination
          MVI L, OAH     ; counter
LOOP1:    LDAX, B        ; load A with content of [BE]
          MOV H, A       ; H←A
          ANI 28H
          CPI 20H
          JNZ CHANGE
          MOV A, H       ; A←H
          JMP STORE
CHANGE:   MOV A,H        ; A←H
          XRI 44H        ; A←data with charged bit of
                         ; D2 & D6
STORE:    STAX D
          INX B
          INX D
          DCR L
          JNZ LOOP1      ; continues loop for 10 times.
```

6. There are two tables T1, T2 in memory having ten eight bit data in each. Write a program for 8085 to find the difference of corresponding element of these two tables.

Store the result of each operation on the corresponding element of the third table. Remember that the result should not be negative; it should be |T1-T2|.

[2064 Poush]

⇒

T1 table- 2050H
T2 table- 2060H
T3 table- 2070H

```
        LXI H, 2050H   ; Source 1st table
        LXI D, 2060H   ; Source 2nd table
        LXI B, 2070H   ; Destination

LOOP1:  PUSH B         ; Stores content of BC contained
                       ; in stack
        LDAX D         ; load accumulator with data
                       ; of [DE]
        MOV B, A       ; T2 table content at B
        MOV A, M       ; A←[HL] 1st table content
        CMP B          ;carry occurs when B>A
        JC SWAPSUB
        SUB B
        JMP END

SWAPSUB: MOV C,A
        MOV A,B
        SUB C

END:    POP B
        STAX B
        INX H
        INX D
        INX B
```



```
        MOV A, L       ; counter for 10
        CPI 5A
        JNZ LOOP1
        HLT
```

7. Write an assembly language program to count no. of -ve element in a data block containing 16 bytes of data; store the count at the end of the block if the count is greater than 8 otherwise store 0. *[2065 Chaitra]*

⇒

Assume: The data are located from XXXXH

```
        LXI H,XXXXH
        MOV C, 10H     ; counter
        MOV B, 00H     ; count-ve element
LOOP1:  MOV A, M
        RLC
        JNC DOWN
        INR B
DOWN:   INX H
        DCR C
        JNZ LOOP1
        MOV A, B
        CPI 09H
        JNC STORE
        MVI A, 00H
STORE:  MOV M, A
        HLT
```

8. Write a program for 8085 to swap nibbles (upper four bits and lower four bits) of ten eight bit number stores at 8000H and transfer to new location 8050H if the number have D_5=1 else store FFH in the destination.

[2066 Shrawan]

⇒

```
        LXI H, 8000H
        LXI D, 8050 H
        MVI B, 0A H
LOOP1:  MOV A, M
```

```
        MOV  C, A
        CPI 20 H
        JNZ STOREFF
        MOV A, C
        RLC
        RLC
        RLC
        RLC
        JMP STORE
STOREFF:MVI A, FF
  STORE: STAX D
        INX  D
        INX H
        DCR  B
        JNZ LOOP1
        HLT
```

9. **Write a program for 8085 to add corresponding data from two table if the data from first table is smaller than the second table else subtract data of second table from first table. Store the result of each operation in the corresponding location if the third table? Assume each table has ten eight bit data.** *[2066 Magh]*

⇒ Let,

1st table is at 2050H

2nd table is at 2060 H

3rd table is at 2070 H

```
        LXI H, 2050H    ; Source 1st table
        LXI D, 2060H    ; Source 2nd table
        LXI B, 2070H    ; Destination
LOOP1:  PUSH B          ; Stores the B.C content in stack
        LDAX D          ; load accumulator with data of [DE]
        MOV B,A         ; 2nd table content B
        MOV A,M         ; A←[HL] 1st table content A
```



```
        CMP B           ; carry occurs when B>A
                        ; 2nd table is greater than 1st table
        JC ADDITION     ; 1st table content<2nd table
        SUB B           ; subtract 2nd table from 1st table
        JMP END
ADDITION:. ADD B        ; add 1st table and 2nd table
END:  POP B
        STAX B          ; stores addition or subtraction
        INX H
        INX D
        INX B
        MOV A,L         ; counter for 10
        CPI 5A
        JNZ LOOP1
        HLT
```

10. **Write a program in 8085 to add all the numbers from a table of 8-bit numbers whose higher nibble value is greater than 6 and store the 16 bit result just after the table.** *[2067 Shrawan]*

⇒

Assume there are two numbers in a table

```
        LXI H, XX50H
        MVI C, 0AH      ; counter
        MVI D, 00H      ; carry
                        ;
  D7D6D5D4D3D2D1D0
        MVI E, 00H      ; LSB sum Anding 11110000
  UP:   MOV A, M
        ANI F0H
        CPI 70H
        JC SKIP
        MOV A, M
```

```
        ADD D
        MOV D, A
        JNC SKIP
        INR D
SKIP:   INX H
        DCR C
        JNZ UP
        MOV M, E
        INX H
        MOV M, D
        HLT
```

11. **A set of three reading is stored in memory starting at 9040H. Write an assembly language program to sort the readings in ascending order. Store the smallest value in address 9054H and so on in higher addresses.**

[2067 Mangsir]

⇒

```
START:  LXI H, 9040H    ; set up HL as a memory
                        ; pointer for bytes
        MVI D, 00       ; clear register D to set up a
                        ; flag
        MVI C, 02       ; set register C for comparison
                        ; count
CHECK:  MOV A, M        ; get data byte
        INX H           ; point to next byte
        CMP M           ; compare bytes
        JC NXTBYT       ; if (A) <second byte for
                        ; exchange
        MOV B, M        ; get second byte for
                        ; exchange
        MOV M, A        ; store the first byte in second
                        ; location
        DCX H           ; point to first location
```



```
        MOV M, B        ; store second byte in first
                        ; location
        INX H           ; get ready for next comparison
        MVI D, 01       ; load 1 in D as a reminder for
                        ; exchange
NXTBYT: DCR C           ; decrement comparison count
        JNZ CHECK       ; if comparison count ≠0, go
                        ; back
        MOV A, D        ; get flag bit in A
        RRC             ; place flag bit D0 in carry
        JC START        ; if flag is 1, exchange occurred
                        ; start the next pass
        LXI H, 9040H    ; set up HL as a memory pointer for
                        ; bytes
        LXI D, 9054H    ; set up DE as a memory pointer
                        ; for bytes
        MVI C, 03H      ; set register C for count
AGAIN:  MOV A, M        ; get data byte
        STAX D          ; store data byte at DE register
        INX H           ; point to next byte
        INX D           ; point to next byte
        DCR C           ; decrease the counter
        JNZ AGAIN       ; transfer data till 3 data byte are
                        ; transferred
        HLT             ; end of sorting
```

12. **There is a table in memory which has ten eight bit numbers starting at 9350H. Write a program for 8085 to transfer the numbers from this table to another table that starts at location 9540H by swapping bit D_6 and bit D_2 if the number is greater than 90H else transfer by adding 48H.** *[2068 Jestha]*

⇒

```
        LXI H, 9350H    ; source
        LXI D, 9450 H   ; destination
```

```
        MVI C, 09H      ; counter
LOOP1: MOV A, M
        CPI, 91H        ;CY flag set for data<90
        JNC SWAP
        ADI 48H
        JMP NEXT
SWAP: MOV B, A
        ANI 44H         ; A←all bits zero except
                        ;D6 & D2
        CPI 44H         ; D6 & D2 are 1, 1
        JZ NEXT
        CPI 00H         ; D6 & D2 are 0, 0
        JZ NEXT
        MOV A, B
        XRI 44H
NEXT: STAX D
        INX D
        INX H
        DCR C
        JNZ LOOP1
        HLT
```

13. Ten no. of 8-bit data is started in memory at A000H. Write a program for 8085 microprocessor to copy the data to next table at A030H if the data is less than 70H and greater than 24H. *[2068 Magh]*

⇒

```
        LXI H, A000H
        LXI D, A030H
        MVI C, 0AH
UP:  MOV A, M
        CPI 24H
        JC NEXT
        CPI 70H
        JNC NEXT
        STAX D
        INX D
NEXT:  INX H
        DCR C
        JNZ UP
        HLT
```


14. Write a program in 8085 to transfer 8-bit number from one table to other by setting bit D_5 if the number is less than 80H else transfer the number by resetting bit D_6. *[2068 Bhadra]*

⇒

```
        LXI H, A000H
        LXI D, A030H
        MOV A, M
        CPI 80H
        JNC NEXT
        ORI 20H
        STAX D
        JMP LAST
NEXT:
        ANI BFH
        STAX D
LAST:
        HLT
```

15. Write an assembly language program for 8085 to exchange the bits D_6 and D_2 of every byte of a program. Suppose there are 200 bytes in the program starting from memory location 8090H. *[2070 Bhadra]*

⇒

```
        LXI H,8090H
        MVI C,05H
REPEAT:
        MOV A,M
        ANI 44H
```

```
        CPI 44H
        JZ NO_CHG
        CPI 00H
        JZ NO_CHG
        MOV A,M
        XRI 44H
        MOV M,A
NO_CHG:
        INX H
        DCR C
        JNZ REPEAT
        HLT
```

16. Write a program for 8085 to add the upper and lower nibble of ten 8 bit words stored in a table that starts from location 8B20H. Store the separate results in locations just after the table. *[2071 Bhadra]*

⇒

```
        LXI D,8B20H
        LXI H,8B2AH
        MVI C,0AH
NEXT:   LDAX D
        ANI F0H
        RRC
        RRC
        RRC
        RRC
        MOV B,A
        LDAX D
        ANI 0FH
        ADD B
        MOV M,A
        INX H
        INX D
        DCR C
        JNZ NEXT
        HLT
```

17. Write an assembly language program for 8085 to find the square of ten 8-bit numbers which are ≤ 0FH, stored from memory location C090H. Store the result from the end of the source table. *[2072 Magh]*

⇒

```
        LXI H, C090H  ; Source Address
        LXI D, C09AH  ; Destination Address
        MVI C, 0A     ; Counter
NEXT:   MOV B, M
        MVI A, 00H
SQUARE: ADD M
        DCR B
        JNZ SQUARE
        STAX D
        INX H
        INX D
        DCR C
        JNZ NEXT
        HLT
```

18. Explain briefly the programmer's model of a 8085 microprocessor. *[2071 Magh]*

⇒

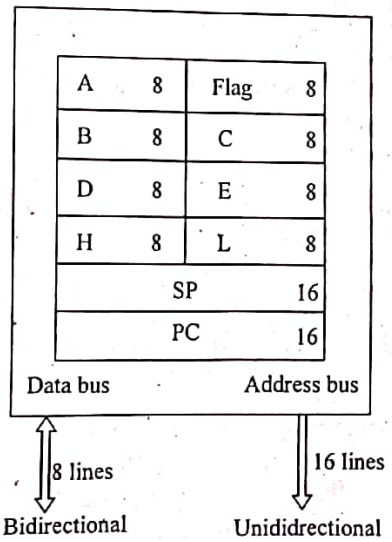


Fig.: Programmer's model of a 8085 microprocessor

The programmer's model of a 8085 microprocessor is explained under the following headings:

Accumulator

It is an 8-bit register accessible to programmer. Almost all arithmetic, logical, and I/O operations are performed on the accumulator.

Flags

Flags are 8-bit register that shows the status of last ALU operations. There are five flip flops, each flip flop is called flag, each holding the status of different states separately.

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

Stack pointer (SP)

It is 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.

Program counter (PC)

It is a 16-bit register that holds the address of next instruction to be executed.

B, C, D, E, H, L

These are six general purpose 8-bit registers. The register pairs (B-C, D-E, H-L) can handle 16-bit of data.

19. What do you understand by looping? How can we perform looping in 8085 microprocessor, explain with example. Explain the mechanism of creating loops in creating delay loops of specified time. *[2064 Poush]*

The programming technique used to instruct the microprocessor to repeat task is called looping. A loop is set up by instructing the microprocessor to change the sequence of execution and perform the task again. This process is accomplished by using jump instruction.

Loops can be classified in to two groups:

- **Unconditional Loop:** An unconditional loop is set up by using the unconditional jump instruction. A program with an unconditional loop does not stop repeating the tasks until the system is reset.
- **Conditional Loop:** A conditional loop is set up by the conditional jump instruction. These instructions checks flags (Zero, Carry, etc.) and repeat the specified the task if the conditioned are satisfied. This loop usually includes counting and indexing.

The looping in 8085 microprocessor is performed by following steps:

- Counter is set up by loading an appropriate count in a register.
- Counting is performed by either incrementing or decrementing the counter.
- Loop is set up by conditional jump instruction.
- End of counting is indicated by a flag.

It is easier to count down to zero than to count up because the zero flag is set when the register becomes zero.

The time delay can be achieved by using two loops; one loop inside the other loop , in the following example register C is used in the inner loop (LOOP1) and register

Time Delay Using a Loop within a Loop Technique

A time delay can be achieved by using two loops; one loop inside the other loop, register C is used in the inner loop (LOOP1) and register B is used for the outer loop (LOOP2). The following instructions

```
        MVI B, 38H      7T
LOOP2:  MVIC, FFH       7T
LOOP1:  DCR C           4T
        JNZ LOOP1       10/7T
```

DCR B 4T

JNZ LOOP2 10/7T

Delay calculations

The delay in LOOP1 is T_{L1}=17 83.5 micro second. We can replace LOOP1 by T_{L1}. Now we can calculate the delay in LOOP2 as if it is one. This loop is executed 56 times because of the count (38h) in register.

T_{L2}= 56(T_{L1} + 21 T-states×0.5 µs)

= 56(1783.5 µs + 10.5 µs)

= 100.46 µs

The total delay should include the execution time of the first instruction (MVI B, 7T). The time delay can be increased by using register pairs.

Similarly, the time delay within a loop can be increased by using instructions that will not affect the program except to increase the delay. For example, the instruction NOP (No Operation) can add four T-stares in the delay loop.

❖ ❖ ❖

Chapter - 3

PROGRAMMING WITH 8086 MICROPROCESSOR

Internal Architecture and Features of 8086 Microprocessor

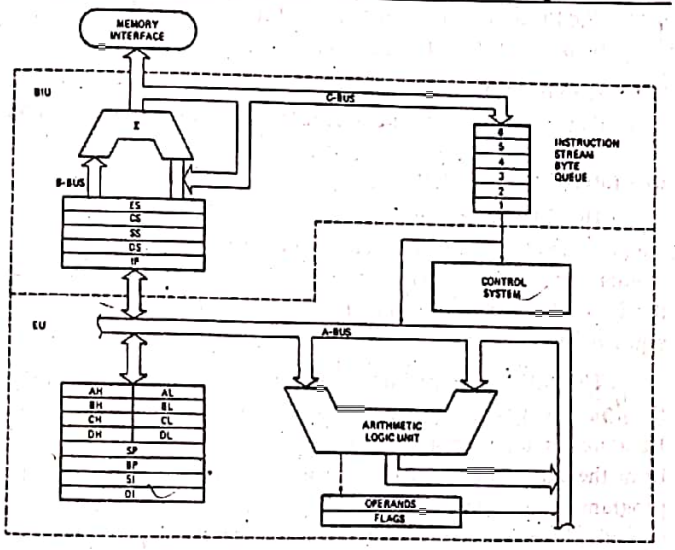


Fig.: Internal block diagram of 8086 microprocessor

Features of 8086 microprocessor

1. Intel 8086 is a widely used 16 bit microprocessor.
2. The 8086 can directly address 1MB of memory.
3. The internal architecture of the 8086 microprocessor is an example of register based microprocessor and it uses segmented memory.

4. It pre-fetches up to 6 instruction bytes from the memory and queues them in order to speed up the instruction execution.
5. It has data bus of width 16 bits and address bus of width 20 bits. So it always accesses a 16 bit word to or from memory.
6. The 8086 microprocessor is divided internally into two separate units which are Bus interface unit (BIU) and the execution unit (EU).
7. The BIU fetches instructions, reads operands and write results.
8. The EU executes instructions that have already been fetched by BIU so that instructions fetch overlaps with execution.
9. A 16 bit ALU in the EU maintains the MP status and control flags, manipulates general register and instruction operands.

Bus interface unit (BIU)

The BIU sends out addresses, fetches instructions from memory, reads data from memory or ports, and writes data to memory or ports. So, it handles all transfers of data and address on the buses for EU. It has mainly 2 parts: instruction queue and segment registers.

The BIU can store up to 6 bytes of instructions with FIFO (First in First Out) manner in a register set called a queue. When EU is ready for next instruction, it simply reads the instruction from the queue in the BIU. This is done in order to speed up program execution by overlapping instruction fetch with execution. This mechanism is known as pipelining.

The BIU contains a dedicated address, which is used to produce 20 bit address. Four segment registers in the BIU are used to hold the upper 16 bits of the starting address of four memory segments that the 8086 is working at a particular time. These are code segment, data segment, stack segment and extra segment. The 8086's 1 MB memory is divided into segments up to 64KB each.

- **Code segment register and instruction pointer (IP):** The CS contains the base or start of the current code segment. The IP

contains the distance or offset from this address to the next instruction byte to be fetched. Code segment address plus an offset value in the IP indicates the address of an instruction to be fetched for execution.

- **Data segment register:** Data segment contains the starting address of a program's data segment. Instructions use this address to locate data. This address plus an offset value in an instruction, causes a reference to a specific byte location in the data segment.
- **Stack segment (SS) register and stack pointer (SP):** Stack segment contains the starting address of a program's stack segment. This segment address plus an offset value in the stack pointer indicates the current word in the stack being addressed.
- **Extra Segment (ES) register:** It is used by some string (character data) to handle memory addressing. The string instructions always use the ES and destination index (DI) to determine 20 bit physical address.

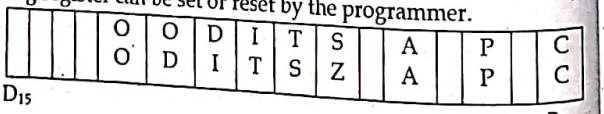
Execution unit (EU)

The EU decodes and executes the instructions. The EU contains arithmetic and logic (ALU), a control unit, and a number of registers. These features provide for execution of instructions and arithmetic and logical operations. It has nine 16 bit registers which are AX, BX, CX, DX, SP, BP, SI, DI and flag. First four can be used as 8 bit register (AH, AL, BH, BL, CH, CL, DH, DL)

- **AX register:** AX register is called 16 bit accumulator and AL is called 8 bit accumulator. The I/O (IN or OUT) instructions always use the AX or AL for inputting/outputting 16 or 8 bit data from or to I/O port.
- **BX register:** BX is known as the base register since it is the only general purpose register that can be used as an index to extend addressing. The BX register is similar to the 8085's H, L register. BX can also be combined with DI or SI as a base register for special addressing.

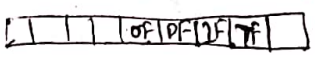
- **CX register:** The CX register is known as the counter register because some instructions such as SHIFT, ROTATE and LOOP use the contents of CX as a Counter.
- **DX register:** The DX register is known as data register. Some I/O operations require its use and multiply and divide operations that involve large values assume the use of DX and AX together as a pair. DX comprises the rightmost 16 bits of the 32-bit EDX.
- **Stack pointer (SP) and base pointer (BP):** Both are used to access data in the stack segment. The SP is used as an offset from the current stack segment during execution of instructions. The SP's contents are automatically updated (increment/decrement) during execution of a POP and/ PUSH instructions.

  The BP contains the offset address in the current stack segment. This offset is used by instructions utilizing the based addressing mode.
- **Index register:** The two index registers SI (Source index) and DI (Destination Index) are used in indexed addressing. The instructions that process data strings use the SI and DI index register together with DS and ES respectively, in order to distinguish between the source and destination address.
- **Flag register:** The 8086 has nine 1 bit flags. Out of nine, six are status and three are control flags. The control bits in the flag register can be set or reset by the programmer.



O (Overflow flag): This flag is set if an arithmetic overflow occurs i.e., if the result of a signed operation is large enough to be accommodated in a destination register.

D (Direction flag): This is used by string manipulation instructions. If this flag bit is '0' , the string is processed beginning from the lowest address to the higher address, i.e. auto incrementing mode otherwise the string is processed



from the highest address towards the lowest address, i.e. auto decrementing mode.

I (Interrupt flag): If this flag is set, the maskable interrupts are recognized by the CPU, otherwise they are ignored.

T (Trap flag): If this flag is set, the processor enters the single step execution mode. In other words, a trap interrupt is generated after execution of each instruction. The processor executes the current instruction and the control is transferred to the Trap interrupt service routine.

S (Sign flag): This flag is set when the result of any computation is negative. For signed computations, the sign flag equals the MSB of the result.

Z (Zero flag): This flag is set when the result of the computation is or comparison performed by the previous instruction is zero. 1 for zero result, 0 fir nonzero result

A (Auxiliary carry): This is set if there is a carry from the lowest nibble i.e., bit three during the addition or borrow for the lowest nibble i.e., bit three, during subtraction.

P (Parity flag): This flag is set to 1 if the lower byte of the result contains even number of 1s otherwise reset.

C (Carry flag): This flag is set when there is a carry out of MSB in case of addition or a borrow in case of subtraction.

Segment and Offset Address

Segments are special areas defined in a program for containing the code, data, and stack. A segment begins on a paragraph boundary. A segment register is of 16 bits in size and contains the starting address of a segment.

A segment begins on a paragraph boundary, which is an address divisible by decimal 16 or hex 10. Consider a DS that begins at location 038EOH. In all cases, the rightmost hex digit is zero, the computer designers decided that it would be unnecessary to store the zero the zero digit in the segment register. Thus 038E0H is stores in register as 038EH.

The distance in bytes from the segment address to another location within the segment is expressed as an offset or displacement. Suppose the offset of 0032H for above example of data segment. Processor combines the address of the data segment with the offset as:

SA: OA (segment address: offset address)

038E: 0032 H = 038E * 10 +0032 = 038EO + 0032

Physical address = 03912H

Instructions in 8086

1. **Arithmetic Instructions**

   a. ADD (Addition)

   - ADD reg_8 /mem_8, reg_8/mem_8/ $Immediate_8$

   - ADD reg_{16}/mem_{16}, reg_{16}/ mem_{16}/ $Immediate_{16}$

   E.g. ADD AH, 15 ; It adds binary number

   ADD AH, NUM1

   ADD AL, [BX]

   ADD [BX], CH

   ADD AX, [BX]

   b. ADC (Addition with carry)

   - ADC reg/ mem, reg/mem/Immediate data

   c. SUB (Subtraction)

   - SUB reg/mem, reg/mem/Immediate

   d. SBB (Subtraction with borrow)

   - SBB reg/mem, reg/mem/Immediate

   e. MUL (Multiplication)

   - MUL reg_8/mem_8 (8 bit accumulator, AL)

   - MUL reg_{16}/ mem_{16} (16 bit accumulator, AX)

E.g.

MUL R_8 (multiplier) ≡ $R_8 \times AL \rightarrow AX$ (16 bit result)

MUL R_{16} (multiplier) ≡ $R_{16} \times AL \rightarrow DX:AX$ (32 bit result)

f. IMUL (Signed multiplication)

- Same operation as MUL but takes sign into account

g. DIV (Division)

- DIV $R_8 \Rightarrow AX/R_8 \Rightarrow$ (Remainder $\rightarrow$ AH) and (Quotient $\rightarrow$ AL)

- DIV $R_{16} \Rightarrow DX:AX/R_{16} \Rightarrow$ (Remainder $\rightarrow$ DX) and (Quotient $\rightarrow$ AX)

h. IDIV (Signed division)

- Same operation as DIV but takes sign into account.

i. INC/DEC (Increment/Decrement by 1)

- INC/DEC reg./mem. (8 bit or 16 bit)

E.g. INC AL , DEC BX, INC NUM1

j. NEG- Negate (2's complement)

k. ASCII-BCD conversion

- AAA (ASCII adjust after addition)
- AAS (ASCII adjust after subtraction)
- AAM (Adjust after multiplication)
- AAD (Adjust after division)
- DAA (Decimal adjust after addition)
- DAS (Decimal adjust after subtraction)

2. **Logical/shifting/comparison instructions**

   a. Logical

   - AND/OR/XOR reg/mem, reg/mem/immediate
   - NOT reg/mem

   E. g. AND AL, AH

   XOR [BX], CL

b. Rotation

- ROL/ROR/RCL/RCR reg/mem, 1/CL

ROL (Rotate left)

ROR (Rotate right)

RCL (Rotate left through carry)

RCR (Rotate right through carry)

E.g. ROL AX, 1 ; rotate by 1

ROL AX, CL ; if we need to rotate more than one bit

RCL CX, 1

RCL [BX], CL ; Only CL can be used

c. Shifting

- SHL/SHR/SAL/SAR reg/mem, 1/CL

SHL (Logical shift left)

SHR (logical shift right)

-Shifts bit in true direction and fills zero in vacant place

SAL (Arithmetic shift left)

SAR (Arithmetic shift right)

-Shifts bit/word in true direction, in former case place zero in vacant place and in later case place previous sign in vacant place.

E.g. SHL AX, 1 ; rotate by 1

SHL AX, CL ; if we need to rotate more than one bit

SAR DX, 1

SAR [BX], CL ; Only CL can be used

d. Comparison

- CMP reg/mem, reg/mem/immediate

CMP (Compare)

E.g. CMP BH, AL

Operand1		Operand 2	CF	SF	ZF
	>		0	0	0
	=		0	0	1
	<		1	1	0

TEST: test bits (AND operation)

TEST reg/mem, reg/mem/immediate

3. **Data transfer instructions**

- MOV reg./mem , reg./mem./immediate
- LEA BX, ARR = MOV BX, OFFSET ARR
- LDS BX, NUM1

LDS (Load data segment register)

LEA (Load effective address)

LES (Load extra segment register)

LSS (Load stack segment register)

- XCHG reg/mem, reg/mem

E.g. XCHG AX, BX

XCHG AL, BL

XCHG CL, [BX]

IN AL, DX ; DX: Port address, AH also in AL

OUT DX, AL/AH

4. **Flag Operation**

- CLC (Clear carry flag)
- CLD (Clear direction flag)
- CLI (Clear interrupt flag)
- STC (Set carry flag)
- STD (Set direction flag)
- STI (Set interrupt flag

- CMC (Complement carry flag)
- LAHF (Load AH from flags (lower byte))
- SAHF (Store AH to flags)
- PUSHF (Push flags into stack)
- POPF (Pop flags off stack)

5. **STACK Operations**
   - PUSH reg_{16}
   - POP reg_{16}
6. **Looping instruction (CX is automatically used as a counter)**
   - LOO P (Loop until complete)
   - LOOPE (Loop while equal)
   - LOOPZ (Loop while zero)
   - LOOPNE (loop while not equal)
   - LOOPNZ (Loop while not zero)
7. **Branching instruction**
   a. **Conditional**
      - JA (Jump if above)
      - JAE (Jump if above/equal)
      - JB (Jump if below)
      - JBE (Jump if below/equal)
      - JC (Jump if carry)
      - JNC (Jump if no carry)
      - JE (Jump if equal)
      - JNE (Jump if no equal)
      - JZ (Jump if zero)
      - JNZ (Jump if no zero)
      - JG (Jump if greater)
      - JNG (Jump if no greater)

      - JL (Jump if less)
      - JNL (Jump if no less)
      - JO (Jump if overflow)
      - JS (Jump if sign)
      - JNS (Jump if no sign)
      - JP (Jump if plus)
      - JPE (Jump if parity even)
      - JNP (Jump if no parity)
      - JPO (Jump if parity odd)

   b. **Unconditional**
      - CALL (Call a procedure)
      - INT (Interrupt)
      - JMP (Unconditional jump)
      - RETN/RETF (Return near/far)
      - RET (Return)
      - IRET (Interrupt return)
8. **Type conversion**
   - CBW (Convert byte to word)
   - CWD (Convert word to double word)
9. **String instructions**
   - MOVS/ MOVSB/MOVSW ; Move string
     DS: SI (Source)
     DS: DI (Destination)
     CX (String length)
   - CMPS/ CMPSB/CMPW ; Compare string
   - LODS /LODSB/LODW ; Load string
   - REP ; Repeat string

Operators in 8086

An operator provides a facility for changing or analyzing operands during an assembly. Operator can be applied in the operand which uses the immediate data/address. Operators are active during assembling but no machine language code will be generated.

i. **Calculation operators**

   a. **Arithmetic operators**

      +Addition, +Positive, -Subtraction, -Negation, *Multiplication, / Division.

   b. **Index**

      MOV [BX + SI+ 4]

   c. **Logical**

      AND, OR, XOR, NOT

   d. **Shift**

      SHR (Shift Logical Right), SAR (Shift Arithmetic Right), SHL (Shift Logical Left), SAL (Shift Arithmetic Left), etc.

ii. **Macro operators**

Basic format of a macro definition:-

macro name	MACRO [parameter-list]	; Define macro
	[instructions]	; Body of macro
	ENDM	; End of macro

iii. **Record operators**

MASK, and WIDTH.

iv. **Relational operators**

EQU, GE, GT, LE, LT, and NE.

v. **Segment operators**

OFFSET, SEG, and Segment override

vi. **Type (or attribute) operators**

HIGH, HIGHWORD, LENGTH, LOW, LOWWORD, PTR, SHORT, SIZE, THIS, and TYPE.

Coding in Assembly Language

Assembly language programming has taken its place in between the machine language (low level) and the high level language.

- High level language's one statement may generate many machine instructions.
- Low level language consists of either binary or hexadecimal operation. One symbolic statement generates one machine level instructions.

Advantage of ALP

- They generate small and compact execution module.
- They have more control over hardware.
- They generate executable module and run faster.

Disadvantages of assembly language programming:

- Machine dependent.
- Lengthy code
- Error prone (likely to generate errors).

Assembly language features:

The main features of ALP are program comments, reserved words, identifies, statements and directives which provide the basic rules and framework for the language.

Program comments:

- The use of comments throughout a program can improve its clarity.
- It starts with semicolon (;) and terminates with a new line.
  E.g. ADD AX, BX ; Adds AX & BX

Reserved words:

- Certain names in assembly language are reserved for their own purpose to be used only under special conditions and includes
- **Instructions:** Such as MOV and ADD (operations to execute)

- **Directives:** Such as END, SEGMENT (information to assembler)
- **Operators:** Such as FAR, SIZE
- Predefined symbols: such as @DATA, @ MODEL

Identifiers:

- An identifier (or symbol) is a name that applies to an item in the program that expects to reference.
- Two types of identifiers are Name and Label.
- Name refers to the address of a data item such as NUM1 DB 5, COUNT DB 0
- Label refers to the address of an instruction.

E. g: MAIN PROC FAR

L1: ADD BL, 73

Statements:

- ALP consists of a set of statements with two types
- Instructions, e. g. MOV, ADD
- Directives, e. g. define a data item

Identifiers	operation	operand	comment
Directive:	COUNT	DB	1
; initialize count			
Instruction:	L30:	MOV	AX, 0
; assign AX with 0			

Directives:

The directives are the number of statements that enables us to control the way in which the source program assembles and lists. These statements called directives act only during the assembly of program and generate no machine-executable code. The different types of directives are:

1. **The page and title listing directives:**

The page and title directives help to control the format of a listing of an assembled program. This is their only purpose

and they have no effect on subsequent execution of the program.

The page directive defines the maximum number of lines to list as a page and the maximum number of characters as a line.

PAGE [Length] [Width]

Default : Page [50][80]

TITLE gives title and place the title on second line of each page of the program.

TITLE text [comment]

2. **SEGMENT directive**

It gives the start of a segment for stack, data and code.

Seg-nameSegment [align][combine]['class']

Seg-nameENDS

Segment name must be present, must be unique and must follow assembly language naming conventions.

An ENDS statement indicates the end of the segment.

Align option indicates the boundary on which the segment is to begin; PARA is used to align the segment on paragraph boundary.

Combine option indicates whether to combine the segment with other segments when they are linked after assembly. STACK, COMMON, PUBLIC, etc are combine types.

Class option is used to group related segments when linking. The class code for code segment, stack for stack segment and data for data segment.

3. **PROC Directives**

The code segment contains the executable code for a program, which consists of one or more procedures, defined initially with the PROC directives and ended with the ENDP directive.

PROC - name PROC [FAR/NEAR]

.....

.....

.....

PROC - name ENDP

FAR is used for the first executing procedure and rest procedures call will be NEAR.

Procedure should be within segment.

4. **END Directive**

An END directive ends the entire program and appears as the last statement.

ENDS directive ends a segment and ENDP directive ends a procedure. END PROC-Name

5. **ASSUME Directive**

An .EXE program uses the SS register to address the stack, DS to address the data segment and CS to address the code segment.

Used in conventional full segment directives only.

Assume directive is used to tell the assembler the purpose of each segment in the program.

Assume SS: Stack name, DS: Data segname, CS: code segname

6. **Processor directive**

Most assemblers assume that the source program is to run on a basic 8086 level computer.

Processor directive is used to notify the assembler that the instructions or features introduced by the other processors are used in the program.

E.g. .386 - program for 386 protected mode.

7. **Dn Directive (Defining data types)**

Assembly language has directives to define data syntax [name] Dn expression

The Dn directive can be any one of the following:

DB	Define byte	1 byte
DW	Define word	2 bytes
DD	Define double	4 bytes
DF	defined farword	6 bytes
DQ	Define quadword	8 bytes
DT	Define 10 bytes	10 bytes

VAL1 DB 25

ARR	DB	21, 23, 27, 53
MOV	AL, ARR [2] or	

MOV AL, ARR + 2 ; Moves 27 to AL register

8. **The EQU directive**

It can be used to assign a name to constants.

E.g. FACTOR EQU 12

MOV BX, FACTOR ; MOV BX, 12

It is short form of equivalent.

Do not generate any data storage; instead the assembler uses the defined value to substitute in.

9. **DUP Directive**

It can be used to initialize several locations to zero.

e. g. SUM DW 4 DUP(0)

Reserves four words starting at the offset sum in DS and initializes them to Zero.

Also used to reserve several locations that need not be initialized. In this case (?) is used with DUP directives.

E. g. PRICE DB 100 DUP(?)

Reserves 100 bytes of uninitialized data space to an offset PRICE.

Program written in Conventional full segment directive

```
Page 60,132
TITLE SUM program to add two numbers
;-----------------------------------
STACK SEGMENT PARA STACK 'Stack'
DW 32 DUP(0)
STACK ENDS
;-----------------------------------
DATA SEG SEGMENT PARA 'Data'
NUM1 DW 3291
NUM 2 DW 582
SUM DW?
DATA SEG ENDS
;-----------------------------------
CODE SEG SEGMENT PARA 'Code'
MAIN PROC FAR
 ASSUME SS: STACK, DS:DATASEG, CS:CODESEG
 MOV AX, @DATA
 MOV DS, AX
 MOV AX, NUM1
 ADD AX, NUM2
 MOV AX, 4C00H
 INT 21H
MAIN ENDP
CODESEG ENDS
END MAIN
```

Description for conventional program:

- STACK contains one entry, DW (define word), that defines 32 words initialized to zero, an adequate size for small programs.

- DATASEG defines 3 words NUM1, NUM2 initialized with 3291 and 582 and sum uninitialized.
- CODESEG contains the executable instructions for the program, PROC and ASSUME generate no executable code.
- The ASSUME directive tells the assembler to perform these tasks.
- Assign STACK to SS register so that the processor uses the address in SS for addressing STACK.
- Assign DATASEG to DS register so that the processor uses the address in DS for addressing DATASEG.
- Assign CODESEG to the CS register so that the processor uses the address in CS for addressing CODESEG.
- When the loading a program for disk into memory for execution, the program loader sets the correct segment addresses in SS and CS.

Program written using simplified segment directives:

```
.Model memory model
Memory model can be
TINY, SMALL, MEDIUM, COMPACT, LARGE, HUGE or FLAT
TINY for .com program
FLAT for program up to 4 GB
```

- Assume is automatically generated

```
.STACK [size in bytes]
Creates stack segment
.DATA: start of data segment
.CODE: start of code segment
```

- DS register can be initialized as

```
MOV AX, @DATA
MOV DS, AX
```

ALP written in simplified segment directives:

```
Page 60, 132
TITLE Sum program to add two numbers.
.MODEL SMALL
.STACK 64
.DATA
NUM1 DW 3241
NUM 2 DW 572
SUM DW?
.CODE
MAIN PROC FAR
MOV AX, @ DATA        ; set address of data segment in DS
MOV DS, AX
MOV AX, NUM1
ADD AX, NUM2
MOV SUM, AX
MOV AX, 4C00H         ; End processing
INT 21H
MAIN ENDP             ; End of procedure
END MAIN              ; End of program
```

DOS Debug (TASM)

1. Save the code text in .ASM format and save it to the same folder where masm and link files are stored.
2. Open dos mode and reach within that folder.
3. \> tasm filename.asm → makes.obj
4. \> tlink filename → makes .exe
5. \> filename.exe → run the code.
6. \> td filename.exe → debug the code [use F7 and F8]

Assembling, Linking, and Executing

1. **Assembling:**

- Assembling converts source program into object program if syntactically correct and generates an intermediate **.obj** file or module.
- It calculates the offset address for every data item in data segment and every instruction in code segment.
- A header is created which contains the incomplete address in front of the generated **obj** module during the assembling.
- Assembler complains about the syntax error if any and does not generate the object module.
- Assembler creates **.obj .lst** and **.crf** files and last two are optional files that can be created at run time.
- For short programs, assembling can be done manually where the programmer translates each mnemonic into the machine language using lookup table.
- Assembler reads each assembly instruction of a program as ASCII character and translates them into respective machine code.

Assembler Types:

There are two types of assemblers:

A. **One pass assembler:**

This assembler scans the assembly language program once and converts to object code at the same time.

This assembler has the program of defining forward references only.

The jump instruction uses an address that appears later in the program during scan, for that case the programmer defines such addresses after the program is assembled.

B. **Two pass assembler**

This type of assembler scans the assembly language twice.

First pass generates symbol table of names and labels used in the program and calculates their relative address.

This table can be seen at the end of the list file and here user need not define anything.

Second pass uses the table constructed in first pass and completes the object code creation.

This assembler is more efficient and easier than earlier.

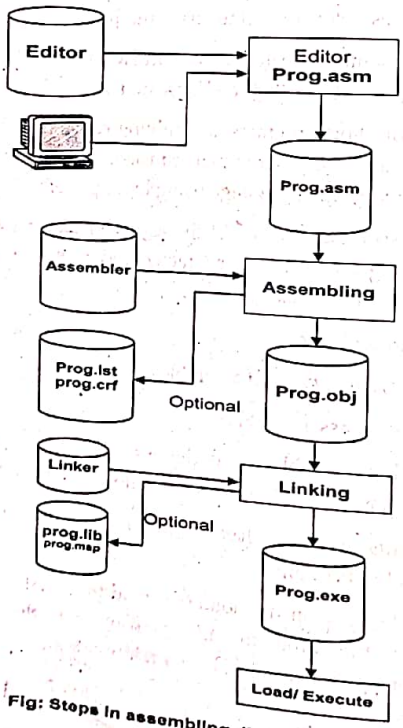


Fig: Steps in assembling, linking & Executing

2. **Linking:**
   - This involves the converting of **.OBJ** module into **.EXE** (executable) module i.e. executable machine code.
   - It completes the address left by the assembler.
   - It combines separately assembled object files.
   - Linking creates **.EXE, .LIB, .MAP** files among which last two are optional files.
3. **Loading and Executing:**
   - It loads the program in memory for execution.
   - It resolves remaining address.
   - This process creates the program segment prefix (PSP) before loading.
   - It executes to generate the result.
   - Sample program $\xrightarrow{\text{assembling}}$ object Program $\xrightarrow{\text{linking}}$ executable program

.COM Programs and .EXE Programs

Writing .COM programs:

- It fits for memory resident programs.
- Code size limited to 64K.
- .com combines PSP, CS, DS in the same segment
- SP is kept at the end of the segment (FFFF), if 64k is not enough, DOS Places stack at the end of the memory.
- The advantage of .com program is that they are smaller than .exe program.
- A program written as .com requires ORG 100H immediately following the code segment's SEGMENT statement. The statement sets the offset address to the beginning of execution following the PSP.

```
.MODEL TINY
.CODE
```

```
ORG 100H                    ; start at end of PSP
BEGIN: JMP MAIN             ;Jump Past data
       VAL1 DW 5491
       VAL2 DW 372
       SUM DW ?
MAIN:  PROC NEAR
       MOV AX, VALL
       ADD AX, VAL2
       MOV SUM, AX
       MOV AX, 4C00H
       INT 21H
MAIN ENDP
END BEGIN
```

Differences between .EXE and .COM programs:

. .EXE and .COM both are executable file. .EXE program consists of separate code, data, and stack segments while .COM program consists of one segment that contains code, data, and stack.

.EXE program

Program size: It uses individual segment of 64 KB maximum for each logical segment. Size of program is larger than .COM program.

Segmentation: A data segment is defined and DS is initialized with the address of the segment. Stack segment must be defined for .EXE programs in higher memory (> 64 KB).

Initialization: The segment registers must be initialized properly by the programmer.

Template of a .EXE program:

```
. data
  ---------
. stack
  ---------
. code
    mov ax, @ data
    mov ds, ax
      '
      '
      '
    end
```


.COM program

Program size: It uses one segment for both instruction and data, restricted to maximum of 64 KB, including program segment prefix (PSP). Size of program is smaller than comparable .EXE program.

Segmentation: It combines the PSP, stack, data, and code segment into one code segment. Assembler automatically generates a stack for .COM programs at the end of segments (< 64 KB).

Initialization: When program loader loads .COM for execution, then CS, DS, SS, and ES are automatically initialized with the address of PSP.

Template of a .COM program:

```
. code
    org 100h
    ------
    ------
```

Macro assembler:

- A macro is an instruction sequence that appears repeatedly in a program assigned with a specific name.
- The macro assembler replaces a macro name with the appropriate instruction sequence each time it encounters a macro name.

- When same instruction sequence is to be executed repeatedly, macro assemblers allow the macro name to be typed instead of all instructions provided the macro is defined.
- Macro are useful for the following purposes:
  - To simplify and reduce the amount of repetitive coding.
  - To reduce errors caused by repetitive coding.
  - To make an assembly language program more readable.
  - Macro executes faster because there is no need to call and return.

Basic format of macro definition:

```
Macro name    MACRO [Parameter list] ; Define macro
              .....
              .....
              [Instructions]          ; Macro body
              .....
              .....
              ENDM                    ; End of macro
```

```
E.g. Addition  MACRO
               IN AX, PORT
               ADD AX, BX
               OUT PORT, AX
               ENDM
```

Passing argument to MACRO:

- To make a macro more flexible, we can define parameters as dummy argument

```
ADDITION    MACRO VALL1, VAL2
            MOV   AX, VAL1
            ADD AX, VAL2
            MOV SUM, AX
            ENDM
.MODEL SMALL
.STACK 64
.DATA
      VAL1  DW    3241
```



```
      VAL2  DW    571
      SUM   DW    ?
.CODE
MAIN PROC FAR
      MOV AX, @ DATA
      MOV DS, AX
      ADDITION VAL1, VAL 2
      MOV AX, 4C00H
      INT 21H
MAIN ENDP
END MAIN
```

Addressing Modes in 8086

Addressing modes describe types of operands and the way in which they are accessed for executing an instruction. An operand address provides source of data for an instruction to process an instruction to process. An instruction may have from zero to two operands. For two operands first is destination and second is source operand. The basic modes of addressing are register, immediate and memory which are described below.

1. **Register addressing:**

   For this mode, a register may contain source operand, destination operand or both.

   E.g. MOV AH, BL

   MOV DX, CX

2. **Immediate addressing**

   In this type of addressing, immediate data is a part of instruction, and appears in the form of successive byte or bytes. This mode contains a constant value or an expression.

   E.g. MOV AH, 35H

   MOV BX, 7A25H

3. **Direct memory addressing:**

   In this type of addressing mode, a 16-bit memory address (offset) is directly specified in the instruction as a part of it.

One of the operand is the direct memory and other operand is the register.

E.g. ADD AX, [5000H]

Note: Here data resides in a memory location in the data segment, whose effective address may be computed using 5000H as the Offset address and content of DS as segment address. The effective address, here, is 10H*DS + 5000H.

4. **Direct offset addressing**

In this addressing, a variation of direct addressing uses arithmetic operators to modify an address.

```
E.g.    ARR DB 15, 17, 18, 21
        MOV AL, ARR [2]     ; MOV AL, 18
        ADD BH, ARR+3       ; ADD BH, 21
```

5. **Indirect memory addressing:**

Indirect addressing takes advantage of computer's capability for segment: offset addressing. The registers used for this purpose are base register (BX and BP) and index register (DI and SI)

```
E.g.    MOV [BX], AL
        ADD CX, [SI]
```

6. **Base displacement addressing:**

This addressing mode also uses base registers (BX and BP) and index register (SI and DI), but combined with a displacement (a number or offset value) to form an effective address.

```
E.g.    MOV BX, OFFSET ARR
        ≡ LEA BX, ARR
        MOV AL, [BX +2]
        ADD TBL [BX], CL
        TBL [BX]  →  [BX + TBL] e.g. [BX + 4]
```


7. **Base index addressing:**

This addressing mode combines a base registers (BX or BP) with an index register (SI or DI) to form an effective address.

```
E.g.    MOV AX, [BX +SI]
        ADD [BX+DI], CL
```

8. **Base index with displacement addressing**

This addressing mode, a variation on base- index combines a base register, an index register, and a displacement to form an effective address.

```
E.g.    MOV AL, [Bx+SI+2]
        ADD TBL [BX +SI], CH
```

9. **String addressing:**

This mode uses index registers, where SI is used to point to the first byte or word of the source string and DI is used to point to the first byte or word of the destination string, when string instruction is executed. The SI or DI is automatically incremented or decremented to point to the next byte or word depending on the direction flag (DF).

E.g. MOVS, MOVSB, MOVSW

Examples:

```
TITLE Program to add ten numbers
        .MODEL SMALL
        .STACK 64
        .DATA
ARR DB 73, 91, 12, 15, 79, 94, 55, 89
SUM DW ?
        .CODE
MAIN PROC FAR
    MOV AX, @DATA
    MOV DS, AX
    MOV CX, 10
    MOV AX, 0
    LEA BX, ARR
L2: ADD AL, [BX]
    JNC L1
```

```
    INC AH
L1: INC BX
    LOOP L2
    MOV SUM, AX
    MOV AX, 4C00H
    INT 21H
MAIN ENDP
END MAIN
```

DOS Functions and Interrupts (Keyboard and Video Processing)

The Intel CPU recognizes two types of interrupts namely hardware interrupt when a peripheral devices needs attention from the CPU and software interrupt that is call to a subroutine located in the operating system. The common software interrupts used here are INT 10H for video services and INT 21H for DOS services.

INT 21H:

It is called the DOS function call for keyboard operations follow the function number. The service functions are listed below:

- # 00H- It terminates the current program.

  Generally not used, function 4CH is used instead.
- # 01H- Read a character with echo

  Wait for a character if buffer is empty

  Character read is returned in AL in ASCII value
- # 02H- Display single character

  Sends the characters in DL to display

  MOV AH, 02H

  MOV DL, 'A' ; move Dl, 65

  INT 21H
- # 03H and 04H – Auxiliary input/output

  INT 14H is preferred.

- # 05H – Printer service

  Sends the character in DL to printer
- # 06H- Direct keyboard and display

  Displays the character in DL.
- # 07H- waits for a character from standard input

  does not echo
- # 08H- keyboard input without echo

  Same as function 01H but not echoed.
- # 09H- string display

  Displays string until '$' is reached.

  DX should have the address of the string to be displayed.
- # 0AH – Read string
- # OBH- Check keyboard status

  Returns FF in AL if input character is available in keyboard buffer.

  Returns 00 if not.
- # 0CH- Clear keyboard buffer and invoke input functions such as 01, 06, 07, 08 or 0A.

  AL will contain the input function.

INT 21H Detailed for Useful Functions

- # 01H

  MOV, AH 01H; request keyboard input INT 21H

  Returns character in AL. IF AL= nonzero value, operation echoes on the screen. If Al= zero means that user has pressed an extended function key such as F1 OR home.
- # 02H

  MOV AH, 02H; request display character

  MOV DL, CHAR; character to display

  INT 21H

Display character in D2 at current cursor position. The tab, carriage return and line feed characters act normally and the operation automatically advances the cursor.

- # 09H

MOV Ah, 09H; request display

LEA DX, CUST_MSG; local address of prompt

INNT 21H

CUST_MSG DB "Hello world", '$'

Displays string in the data area, immediately followed by a dollar sign ($ or 24H), which uses to end the display.

- # OAH

```
MOV AH, 0AH          ; request keyboard input
LEA DX, PARA_ LIST   ; load address of parameter list
INT 21H
```

Parameter list for keyboard input area :

PARA_LIST LABEL BYTE; start of parameter list

MAX_LEN DB 20; max. no. of input character

ACT _ LEN DB ? ; actual no of input characters

KB-DATA DB 20 DUP ('); characters entered from keyboard

LABEL directive tells the assembler to align on a byte boundary and gives location the name PARA _LIST.

PARA_LIST & MAX_LEN refer same memory location, MAX_LEN defines the maximum no of defined characters.

ACT_LEN provides a space for the operation to insert the actual no of characters entered..

KB_DATA reserves spaces (here 20) for the characters.

Example:

```
TITLE to display a string
.MODEL SMALL
.STACK 64
```



```
.DATA
        STR DB 'programming is fun', '$'
.CODE
MAIN PROC FAR
        MOV AX, @DATA
        MOV DS, AX
        MOV AH, 09H          ;display string
        LEA DX, STR
        INT 21H
        MOV AX, 4C00H
        INT 21H
MAIN ENDP
END MAIN
```

INT 10H

It is called video display control. It controls the screen format, color, text style, making windows, scrolling etc. The control functions are:

 # 00H – set video mode

```
MOV AH, 00H          ; set mode
MOV AL, 03H          ; standard color text
INT 10H              ; call interrupt service
```

 # 01H- set cursor size

```
MOV AH, 01H
MOV CH, 00H          ; Start scan line
MOV CL, 14H          ; End scan line
INT 10H              ; (Default size 13:14)
```

 # 02H – Set cursor position:

```
MOV AH, 02H
MOV BH, 00H          ; page no
MOV DH, 12H          ; row/y (12)
MOV DL, 30H          ; column/x (30)
INT 10H
```

 # 03H – return cursor status

```
MOV AH, 03H
```

MOV BH, 00H;
INT 10H
Returns: CH- starting scan line, CL-end scan line, DH- row, DL-column

 # 04H- light pen function

 # 05H- select active page

```
MOV AH, 05H
MOV AL,page-no.      ; page number
INT 10H
```

 # 06H- scroll up screen

```
MOV AX, 060FH   ; request scroll up one line (text)
MOV BH, 61H     ; brown background, blue foreground
MOV CX, 0000H   ; from 00:00 through
MOV DX, 184F H  ; to 24:79 (full screen)
INT 10H
```

AL= number of rows (00 for full screen)
BH= Attribute or pixel value
CX= starting row: column
DX= ending row: column

 # 07H-Scroll down screen

Same as 06H except for down scroll

 # 08H (Read character and Attribute at cursor)

```
MOV AH, 08H
MOV BH, 00H          ; page number 0(normal)
INT 10H
```

AL= character
BH= Attribute

 # 09H -display character and attribute at cursor

```
MOV AH, 09H
MOV AL, 01H  ; ASCII for happy face display
MOV BH, 00H  ; page number
MOV BL, 16H  ; Blue background, brown foreground
MOV CX, 60   ; No of repeated character
INT 10H
```

 # 0AH-display character at cursor

```
MOV AH, 0AH
```



```
MOV Al, Char
MOV BH, page _no
MOV BL, value
MOV CX, repetition
INT 10H
```

 # 0BH- Set color palette

- ✓ Sets the color palette in graphics mode
- ✓ Value in BH (00 or 01) determines purpose of BL
- ✓ BH= 00H, select background color, BL contains 00 to 0FH (16 colors)
- ✓ BH = 01H , select palette, Bl, contains palette

```
MOV AH, 0BH
MOV AH, 0BH
MOV BH, 00H; background      MOV BH, 01H
; select palette
MOV BL, 04H; red             MOV BL, 00H
; black
INT 21H
INT 21H
```

#0CH- write pixel Dot

- Display a selected color

```
AL=color of the pixel        CX= column
BH=page number               DX= row
MOV AH, 0CH
MOV Al, 03
MOV BH,0
MOV CX, 200
MOV DX, 50
INT 10H
```

It sets pixel at column 200, row 50

#0DH- Read pixel dot

- Reads a dot to determine its color value which returns in AL

```
MOV AH, 0DH
MOV BH, 0      ; page no
```

```
MOV CX, 80    ; column
MOV DX, 110   ; row
INT 10H
```

#OEH- Display in teletype mode

Use the monitor as a terminal for simple display

```
MOV AH, 0EH
MOV AL, char
MOV BL, color; foreground color
INT 10H
```

#OF H- Get current video mode

Returns values from the BIOS video .

AL= current video mode
AH= no of screen columns
BH = active video page

```
MOV AH, 0FH
INT 10H
```

TITLE To convert letters into lower case

```
.MODEL SMALL
.STACK 99H
.CODE
MAIN PROC
        MOV AX, @ DATA
        MOV DS, AX
        MOV SI, OFFSER STR
M:      MOV DL, [SI]
        MOV CL, DL
        CMP DL, '$'
        JE N
        CMP DL, 60H
        JL L
K:      MOV DL, CL
        MOV AH, 02H
        INT 21H
        INC SI
        JMP M
```



```
L:      MOV DL, CL
        ADD DL, 20H
        MOV AH, 02H
        INT 21H
        INC SI
        JMP M
N:      MOV AX, 4C00H
        INT 21H
        MAIN ENDP
.DATA
        STR DB 'I am MR Rahul ", '$'
END MAIN
```

TITLE to reverse the string

```
.MODEL SMALL
.STACK 100H
.DATA
    STR1 DB " My name is Rahul" , '$'
    STR2 db 50 dup ('$')
        .CODE
MAIN PROC FAR
        MOV BL,00H
        MOV AX, @ DATA
        MOV DS, AX
        MOV SI, OFFSER STR1
        MOV DI, OFFSET STR2
L2:     MOV DL, [SI]
        CMP DI, '$'
        JE L1
        INC SI
        INC BL
        JMP L2
L1:     MOV CL, BL
        MOV CH, 00H
        DEC SI
L3:     MOV AL, [SI]
```

```
        MOV [DI], AL
        DEC SI
        INC DI
        LOOP L3
        MOV AH,09H
        MOV DX, OFFSET STR2
        INT 21H
        MOV AX, 4C00H
        INT 21H
    MAIN ENDP
    END MAIN
```

TITLE to input characters until 'q' and display

```
    .MODEL SMALL
    .STACK 100H
    .DATA
        STR db 50 DUP ('$')
     .CODE
    MAIN PROC FAR
        MOV AX, @ DATA
        MOV DS, AX
        MOV SI, OFFSET STR
L2:     MOV AH, 01H
        INT 21H
        CMP AL, 'q'
        JE L1
        MOV [SI] , AL
        INC SI
        JMP L2
L1:     MOV AH, 09H
        MOV DX, OFFSET STR
        INT 21H
        MOV AX, 4C00H
        INT 21H
    MAIN ENDP
    END MAIN
```


Calling procedure/subroutine

```
    Procname PROC FAR
    .....
    .....
    Procname ENDP
```

Here, the code segment consists only one procedure. The FAR operand in this case informs the assembler and linker that the defined procedure name is the entry point for program execution, whereas the ENDP directive defines the end of the procedure. A code segment however, may contain any number of procedures, each distinguished by its own PROC and ENDP directives.

A called procedure is a section of code that performs a clearly defined task known as subroutine which provides following benefits.

- Reduces the amount of code because a common procedure can be called from any number of places in the code segment.
- Encourage better program organization.
- Facilitates debugging of a program because defects can be more clearly isolated.
- Helps in the ongoing maintenance of programs because procedures are readily identified for modification.

A CALL to a procedure within the same code segment is NEAR CALL<. A FAR CALL calls a procedure labeled FAR, possibly in another code segment.

```
DISPLAY PROC NEAR
        MOV AH, 09H
        MOV DX, OFFSET STR
        INT 21H
        RET
DISPLAY ENDP
```

✓ To display number contained in [BX]

```
DISPLAY PROC NEAR
        MOV DL, [BX]
        ADD DL, 30
        MOV AH, 02H
        INT 21H
        RET
DISPLAY ENDP
```

INT 10H Video service:

<Video -modes>

Text mode	Row × column	Color	No.of Pages	Resolution	colors
00	25×40	Color	8	360×400	16 colors
01	25×40	Color	8	360×400	16 colors
02	25×80	Color	4	720×400	16 colors
03(by default)	25×80	color	4	720×400	16 colors
07	25×80	Mono-hrome	0	720×400	16 colors

Graphic mode	Color	Pages	Resolution	No of colors
04	Color	8	320×200	4
05	Color	8	320×200	4
06	Color	8	640×200	2
0D	Color	8	320×200	16
0E	Color	4	640×200	16
0F	Mono chrome	2	640×350	1
10	Color	2	640×350	16
11	Color	1	640×480	2
12	Color	1	640×480	16
13	Color	1	320×200	256

Attribute

	Background	Foreground
Attribute:	BL R G B	I R G B
Bit number:	7 6 5 4	3 2 1 0

I – Intensity, BL - Blink

Color	Hex Value
Black	0
Blue	1
Green	2
Cyan	3
Red	4
Magnet	5
Brown	6
White	7
Gray	8
Light Blue	9
Light Green	A
Light cyan	B
Light red	C
Light magenta	D
Yellow	E
Bright white	F

TITLE sorting the numbers - descending order

```
        DOSSEG
                .MODEL SMALL
                .STACK 100H
                .CODE
        MAIN PROC FAR
                        MOV AX, @ DATA
                        MOV DS, AX
                        MOV DX, 4H
        DOPASS:         MOV CX, 4H
                        MOV SI, 00H
```

```
CHECK:   MOV AL, ARR [SI]
         CMP ARP [SI+1] , AL
         JC NOSWAP
         MOV BL, ARR [SI + 1]
         MOV ARR[SI +1] , AL
         MOV ARR [SI], BL
NOSWAP:  INC SI
         LOOP CHECK
         DEC DX
         JNZ DOPASS
         MOV AX, 4C00H
         INT 21H
         MAIN ENDP
    .DATA
         ARR DB 8,2,9,4,7
    END  MAIN
```

Note: Display if numbers are with 1 digit

```
         MOV CX, 05H
         MOV SI, 00H
L:       MOV DL, ARR[SI]
         ADD DL, 30H
         MOV AH, 02H
         INT 21H
         MOV DL,' '
         MOV AH, 02H
         INT 21H
         INC SI
         LOOP L
         MOV AX, 4C00H
         INT 21H
MAIN ENDP
```

TITLE addition of 100 natural even numbers

```
    .MODEL SMALL
    .STACK 100H
    .DATA
         TEN DW 10
    .CODE
```



```
MAIN     PROC FAR
         MOV AX, @ DATA
         MOV DS, AX
         MOV CX, 63H
         MOV AX, 02H
         MOV DX, 04H
L1:      ADD AX, DX
         ADD DX, 02H
         LOOP L1
L2:      MOV DX, 0000H
         DIV TEN        ; DX: AX /10
         INC CX
         ADD DX, 30H    ; remainder
         PUSH DX
         CMP AX, 00H    ; quotient
         JE L3
         JMP L2
L3:      POP DX
         MOV AH, 02H
         INT 21H
         LOOP L3
         MOV AX, 4C00H
         INT 21H
MAIN ENDP
END MAIN
```

TITLE to display string at (10,40) with green background and red foreground

```
         dosseg
         .Model small
         .Stack 100H
         .Code
    MAIN PROC FAR
         MOV AX, @ DATA
         MOV DX, AX
         MOV SI, OFFSET VAR1
L2:      MOV AH, 02H  ; Set cursor position
         MOV DH, ROW
```

```
        MOV DL, COL
        INT 10H
        MOV AL, [SI]
        CMP AL, '$'
        JE L1
        MOV AH, 09H
        MOV DH, ROW
        MOV DL, COL
        MOV BL, 24H ;background & foreground
        MOV BH, 00h  ; page
        MOV CX, 01H  ;no. of repeated characters
        INT 10H
        INC SI
        INC COL
        JMP L2
L1:     MOV AX, 4C00H
        INT 21H
MAIN ENDP
        .DATA
        ROW DB 10
        COL DB 40
        VAR1 DB "video model", '$'
END MAIN
```

TITLE TO GENERATE MULTIPLICATION TABLE

```
        .MODEL SMALL
        .STACK 32
        .DATA
            NUM1 DB 5
            NUM2 DB 1
            TAB DB 10 DUP (?)
        .CODE
MAIN PROC FAR
        MOV AX, @ DATA
        MOV DS, AX
        MOV BX, 0
        MOV CX, 10
```



```
L1:     MOV AL, NUM1
        MUL NUM2
        MOV TAB [BX], AL
        INC BX
        INC NUM2
        LOOP L1
        MOV AX, 4C00H
        INT 21H
MAIN ENDP
END MAIN
```

TITLE to add 10 sixteen bit Numbers in memory table

```
.MODEL SMALL
.STACK 32
.DATA
        NUM DW DUP (2)
        NUM DW DUP (3)
        SUMH DW 0
        SUML DW 0
.CODE
            MAIN PROC FAR
            MOV AX, @ DATA
            MOV DS, AX
            MOV CX, 10
            MOV AX, 0
            MOV BX, 0
            L1:   ADD AX, NUM [BX]
                  MOV SUML, AX
                  JNC L2
                  INC SUMH
            L2:   ADD BX, 2
            LOOP L1
            MOV AX, 4C00H
            INT 21H
MAIN ENDP
END MAIN
```

Note: To access the data of the memory i.e. table. We use e.g. NUM[BX] Increasing the BX register by 2

SUBROUTINE TO CLEAR THE SCREEN

```
SCR_CLEAR PROC NEAR
    MOV AX, 0600H ; Request scroll
    MOV BH, 61H  ; blue on brown for attribute on
```

```
                ;pixel(generally (07H) white on black
MOV CX, 0000    ; Full screen
MOV DX, 184FH
INT 10H
RET
SCR_CLEAR ENDP
```

AH-06H scroll upward of lines in a specified area of the screen.

AL- 00H causes entire screen to scroll up, effectively clearing it. Setting a nonzero value in AL causes the number of lines to scroll up.

ADDITIONAL QUESTIONS

1. Write an assembly language program to read a string from the user, convert it to upper case, count the number of words and display each word in each line. *[2061 Ashwin]*

⇒

```
.model small
.stack
.data
  msg db 'Total count of words is','$'
  maxchar db 255
  nchars db ?
  actstr db 255 dup (?)

.code
newline macro
    push ax
    mov ah,02h
    mov dl,0ah
    int 21h
    mov dl,0dh
    int 21h
    pop ax
newline endm
main proc far
     mov ax,@data
     mov ds,ax
     mov ah,0ah
     mov dx,offset maxchar
     int 21h

 mov si,offset actstr
 mov ah,02h
 mov cx,0
```

```
mov bx,1
mov cl,nchars
newline
again:
mov dl,[si]
sub dl,20h
jne down
newline
inc bl
down: int 21h
inc si
loop again
newline
lea dx,msg
mov ah,09h
mov al,0
int 21h
newline
mov ah,02h
mov dl,bl
or dl,30h
int 21h
mov ah,4ch
int 21h
main endp
end main
```

2. Write assembly language programs for 8086 to sort five numbers in ascending and descending order.

[2062 Baishakh]

⇒ Title sorting five numbers in ascending order

```
.model small
.stack
.data
```



```
array db 4,37,13,50,2,'$'
.code
main proc far
mov ax,@data
mov ds,ax
next:
mov si,offset array
mov bl,0
mov cx,4
again:
mov al,[si]
inc si
cmp al,[si]
jc down
mov dl,[si]
mov [si],al
dec si
mov [si],dl
inc si
mov bl,01
down:
loop again
dec bl
jz next
mov ax,4c00h
int 21h
main endp
end main
```

Title arranging in decending order

```
.model small
.stack
.data
array db 4,37,13,50,2,'$'
```

```
.code
main proc far
mov ax,@data
mov ds,ax
next:
mov si,offset array
mov bl,0
mov cx,4
again:
mov al,[si]
inc si
cmp al,[si]
ja down
mov dl,[si]
mov [si],al
dec si
mov [si],dl
inc si
mov bl,01
down:
loop again
dec bl
jz next
mov si,offset array
mov cl,05
nxtbyte:mov al,[si]
    aam
    or ax,3030h
  mov bx,ax
  mov ah,02
  mov dl,bh
  int 21h
    mov dl,bl
```



```
    int 21h
    mov dl,0
    int 21h
  inc si
  loop nxtbyte
  mov ax,4c00h
  int 21h
  main endp
  end main
```

3. Write a program to read a string and separate the words from the string. Display each word at the center of each line of a clear screen with blue background and cyan foreground. *[2062 Bhadra]*

⇒

```
.model small
.stack 100h
.data
count db 00h
prompt1 db ' enter your string: ' , '$'
pkey db ' press any key.....' , '$'
temp db 50 dup ( "$" )
paralist label byte      ; table of the byte for keyboard input
maxchar db 79            ; maximum limit to enter characters
actlen db ?              ; gives how many Character are entered by
user
kb_buff db 80 dup ('$')  ; this is actually the place to hold entered
                         ;charactered
.code
mainproc
print macro msg          ;defining macro to displaystring
      mov dx, offset msg
      mov ah, 09h
      int 21h
```

```
endm
nextline macro          ; defining macro toadvance to nextline
        mov dl, 0dh     ; control character for from feed
        mov ah, 02h
        int 21h
        mov dl, 0ah     ; control character for for next line
        mov ah, 02h
        int 21h
endm
space macro             ; macro defined print space gap
        mov dl, 09h
        mov ah, 02h
        nt 21h
endm
input macro prlst               ; macro defined to input
                                ;keyboard input
        mov dx, offset prlst
        mov ah, 0ah
        int 21h
endm
setwindow macro
        mov ah, 06h
        mov al, 00h             ; for entire screen
        mov bh, 13h             ; blue background and cyan foreground
        mov cx, 00h
        mov dh, 24
        mov dl, 79
        int 10h
endm
setmode macro
        mov ah, 00h
        mov al, 03h
        int 10h                 ; mode change and screen cleared
```



```
endm
displaycenter macro msg
        mov dh, 12
        mov dl, 40      ; initial row : column
        mov bl, 15
        mov si, offset msg
nxtdsply:mov ah, 02h    ; function code for set cursor
        int 10h         ; set cursor in the beginning
        mov cx, 1       ; cx contains the no of characters to
                        ;display at a time
        mov ah, 0ah             ; display character at cursor
        mov al , [si]
        mov bh, 00h             ; page 0
        int 10h
        inc si                  ; for next pass
        inc dl
        dec bl
        mov bh, [si]
        cmp bh, ' '
        jnz nxtdsply
        inc dh
        mov dl, 39      ; change row and column to nextline middle
        dec bl
        jnz nxtdsply
endm
        movax,@data
        mov ds, ax
        mov es, ax
        print prompt1
        input paralist
        set mode
        setwindow
        displaycenter kb_buff
```

```
        nextline
        print pkey
        mov ah, 1          ; wait for any key…….. hold screen……
        int 21h
        mov ax, 4c00h      ; exit to operating system
        int 21h
main endp
end main
```

4. Write an assembly language program for 8086 to sort an array of ten numbers stored in memory. Display the numbers in the screen after sorting. *[2062 Bhadra]*

⇒

```
.model small
.stack
.data
array db 79,4,37,13,56,63,50,44,23,2,'$'
.code
main proc far
mov ax,@data
mov ds,ax
next:
mov si,offset array
mov bl,0
mov cx,9
again:
mov al,[si]
inc si
cmp al,[si]
jc down
mov dl,[si]
mov [si],al
dec si
mov [si],dl
inc si
```



```
mov bl,01
down:
loop again
dec bl
jz next
mov si,offset array
mov cl,0ah
nxtbyte:mov al,[si]
    aam
    or ax,3030h
  mov bx,ax
  mov ah,02
  mov dl,bh
  int 21h
    mov dl,bl
    int 21h
    mov dl,0
    int 21h
inc si
loop nxtbyte
mov ax,4c00h
int 21h
main endp
end main
```

5. Write an assembly language program for 8086 to read a string count the number of vowels in the string and display the string and its vowels count in a clear screen. *[2063 Kartik]*

⇒

```
.model small
.stack
.data
 vowels db 'a','e','i','o','u','A','E','I','O','U'
 str db 20 dub (?)
```

```
instr db 50
nchar db ?
actstr db 50 dup (?)
count db 0
.code
main proc far
   mov ax,@data
   mov ds,ax
mov ah,0ah
mov dx,offset instr
int 21h
mov bx,offset actstr
mov cx,0
mov cl,nchar
next:
push cx
mov si,offset vowels
mov cl,0ah
mov al,[bx]
up:
cmp al,[si]
 jne down
 inc count
down:
   inc si
   loop up
   inc bx
pop cx
loop next
mov [bx],'$'
mov ah,00h
mov al,03
int 10h
```



```
mov ah,09
mov dx,offset actstr
int 21h
mov al,count
aam
or ax,3030h
mov bx,ax
mov ah,02
mov dl,bh
int 21h
mov dl,bl
int 21h

mov ax,4c00h
int 21h
main endp
end main
```

6. Write a program to generate multiplication table of five numbers stored in memory as array, store the result and display in following format.

5 10 15 20 25 30 35 40 45 50

3 6 9 12 15 18 21 24 27 30... ... ... *[2064 Shrawan]*

⇒

```
.model small
.stack
.data
arrnum db 5, 4, 3, 6, 7
.code
main proc far
mov ax, @data
mov ds,ax

mov ax,0
mov cx,5
```

```
mov bx,0                ; index

total_no_of_table:
mov dl, 1               ; runs for 1 to 10 for each no.
push cx                 ; storing for nested loop
mov cx,10               ; for every no, ten times

label1:
mov al, arrnum [bx]
mul dl                  ; ax=al*dl

;for display
push bx
push cx
push dx
mov bx,0
mov cx,10
no_of_digits:
mov dx,0
div cx                  ;[ dx: ax/cx= ax dx/cx)
add dx,30h
push dx
inc bx
cmp ax,0
ja no_of_digits
mov ah, 02
mov cx, bx

popping:
pop dx
Int 21h
loop popping
pop dx                  ; popping respective register which were
```



```
pop cx
pop bx
; display ends here

inc dl
loop label1
inc bx
pop cx
mov ah, 02h
mov dl, 0ah
int 21h
mov dl, 0dh
int 21h
loop total_no_of_table
mov ax, 4c00h
int 21h
main endp
end main
```

7. **Write down an assembly language program to read a string and count the no of vowels in the string. Display the no. of vowels in the string and the string without the vowels in it in a clear screen with reverse attribute.** *[2064 Poush]*

⇒

```
.model small
.stack
.data
maxchar db 30
nchars db ?
actstr db 30 DUP (0)
msg1 db 'Enter any text: $'
msg2 db  10,13, 'No. of Vowels=$'
msg3 db 10,13,'Here is string without vowels:$'
count db 0
newstr db  30 dup ('$')
```

```
.code
main proc far
mov ax,@data
   mov ds,ax
print macro msg
  mov ah,09h
  mov dx,offset msg
  int  21h
print endm

print msg1
mov ah,0ah
mov dx,offset maxchar
int 21h

mov bx,0
mov cx,0
mov cl,nchars
mov si,offset actstr
mov di,offset newstr

again:
mov ah,[si]
cmp ah,'a'
jl down
cmp ah,'z'
jg down
sub ah,20h
down:
cmp ah,'A'
je vowel
cmp ah,'E'
je vowel
```



```
cmp ah, 'I'
je vowel
cmp ah,'O'
je vowel
cmp ah,'U'
je vowel

mov ah,[si]
mov [di],ah
inc di
jmp NEXT
vowel:inc bl
NEXT:inc si
loop again
mov count,bl
mov bh,70h          ; reverse attribute
mov ax,0            ; code for clear screen
mov dx,184Fh
int 10h
print msg2
mov ah,02h
mov dl,count
or dl,30h
int 21h
print msg3
print newstr
mov ax , 4c00h
int 21h
main endp
end main
```

8. **Write an assembly language program to get a string input; count no. of vowels and display message 'even vowels' on the screen if the count is even otherwise display 'odd vowels'.** *[2065 Chaitra]*

⇒

```
# include io.h
data SEGMENT
```

```
                text db 30, ? , 30 DUP($)
                msg1 db 10 , 13 , 'enter the text:$'
                msgo db 10,13, 'odd  vowels: $'
                msge db 10,13, ' even vowels: $'
data ENDS
code SEGMENTS
                ASSUME ds: data ,cs: code
                mov ax , SEG data
                mov ds , ax

print macro msg
                mov ah , 09h
                mov dx , offset msg
                int 21h
print endm
                print msg1     ; output string using macro
                mov ah , 09h
                mov dx , offset text
                int 21
                mov bx , 0000h     ; to count no. of vowels
                mov cx,  LENGTH text
                mov si, offset text

again : mov ah,[si]
                cmp an , 'a'
                jl down
                cmp ah , 'z'
                jg down
                sub an , 20h

down:  cmp ah , 'A'
                je vowel
                cmp ah, 'E'
```



```
                je vowel
                cmp ah , 'I'
                je vowel
                cmp ah, 'O'
                je vowel
                cmp ah , 'U'
                je vowel
                jmp NEXT
vowel:            inc bx
NEXT:             loop again

                mov ax, bx
                div  2
                cmp ah, 'O'
                je even
                print msgo
                jmp last
                even : print msge
last:       mov ax, 4c00h
                int 21h
code ENDS
```

9. **Write an assembly language program for 8086 to find the sum of the following series, x+2x+3x+4x+..... To ten terms Where x is a two digit number entered by the user. Display the result.** *[2066 Shrawan]*

⇒

```
.model small
.stack
.data
  str1 db 'Enter two digit number:$'
.code
main proc far
   mov ax,@data
```

```
mov ds,ax
mov ah,09h
mov dx,offset str1
int 21h
mov ah,01h
int 21h
and al,0fh
mov dh,al
int 21h
and al,0fh
mov dl,al
mov al,dh
mov bl,0ah
mul bl
add al,dl
  mov dx,0
  mov cx,0
  mov cl,0ah
  mov bl,1
  mov bh,al
again:
mov al,bh
mul bl
add dl,al
inc bl
loop again
mov al,dl
aam
or ax,3030h
mov bx,ax
mov ah,02h
mov dl,bh
int 21h
```



```
mov dl,bl
int 21h
mov ax,4c00h
int 21h
main  endp
end main
```

10. Write an assembly language program to calculate sum of the series $1^2+2^2+3^2+4^2+$.....upto ten terms and display the result. *[2066 Shrawan]*

⇒

```
# include io.h
data SEGMENT
        sum db 4 DUB('$')
        m1db 10 , 13, ' the sum of square : $'
data ENDS
code SEGMENT
    ASSUME ds : data , cs : code
        mov ax , SEG data
        mov ds, ax
        mov dx , 0000          stores sum
        mov cx , 0010          counter
        mov al , 01
again :   mov bh , al
        mul bh                 ax← al * an
        add ax ,dx
        mov dx , ax
        inc al
        loop again
        itoa sum , dx

        mov ah , 09 h
        mov dx , offset m1     ; display msg  m1
        int 21h
```

```
                mov ax , 09h            ; display sum
                mov dx  , offset sum
                int 21h
                mov ax  , 4c00h
                int 21h
code ENDS
```

10. Write a program in 8086 to read a single digit number and display the multiplication table of that number as 2 4 6 8 10 12 14 16 18 20 if the user enters digit 2. *[2067 Shrawan]*

⇒

```
.model small
.stack
.data
.code
main proc far
mov ax, @data
mov dx, ax

mov ah, 07h         ; console input without echo
int 21h
mov cx, 10
mov dl, 1
sub al, 30h         ; to decimal value
label1:
mov ah, 0
push ax
mul dl              ; ax=al*dl

; for display
push cx
push dx
mov bx, 0
mov cx, 10
```



```
no_of_digits:
mov dx, 0
div cx              ; (dx;ax/cx=ax dx/cx)
add dx, 30h
push dx
inc bx
cmp ax, 0
jano_of_digits
mov ah, 02
mov cx, bx

popping:

pop dx
int 21h
loop popping

mov dl, 32
mov ah, 02h
int 21h

pop dx
pop cx

; display ends here

inc dl
pop ax
loop label1

mov ax, 4c00h
int 21h
main endp
end main
```

11. Write an assembly language program to read a string from memory in data segment. Change all the upper case letters to lower case and vice versa. Display the result on the screen. [Note: ASCII code for A=65......Z=90, a=97......z=122]

[2067 Mangsir]

⇒

```
.model small
.stack 100h
.data
count db 00h
prompt1 db 'enter your string: ',', $'
prompt2 db ' you entered : ' , '$'
prompt3 db ' converted string:','$'
pkey db 'press any key','$'
paralist label byte    ; table of 3 bytes for keyboard input
 maxchar db 79
 actlen db ?
 kb_buff db 80 dup ( '$')
.code
main proc
print macro msg  ; defining macro to display string
mov dx, offset msg
mov ah, 09h
 int 21h
print endm
nextline macro       ; defining macro to advance to next line
 mov dl, 0dh         ; control character for form feed
 mov ah, 02h         ; control character for nextLine
 mov ah, 02h
 int 21h
nextline endm
space macro          ; macro defined print spacegap
 mov dl, 09h
 mov ah, 02h
 int 21h
space endm
input macro prlst   ; macro defined to input keyboard input
mov dx, offset prlst
mov ah, 0ah
int 21h
input endm
mov ax,@data
mov ds,ax
mov es, ax
print prompt1
input paralist
nextline
print prompt2
print kb_buff
nextline
call uplow
print prompt3
print kb_buff
nextline
print pkey
mov ah, 1          ;wait for any key.......hold screen...
int 21h
mov ax, 4c00h      ; exit to operating system.
int 21h
main endp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;input: supplied with read string from keyboard
;output: converted to uppercase
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
toupper proc near
```

```
mov si, offset kb_buff
mov ch, 00h
mov cl, actlen      ; cx contain the count of characters by user
chkupr:
  mov al, [si]
  cmp al, 61h
  jc uplast
  cmp al, 7ah
  jnc uplast
  sub al, 20h      ; convert to upper
  mov [si], al
uplast:
  inc si
  loop chkupr
  ret
toupper endp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; to lower
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
tolower proc near
mov si, offset kb_buff
mov ch, 00h
mov cl, actlen    ; cx contains the count of characters entered
                  by user
chkdn:
  mov al, [si]
  cmp al, 41h
  jc dwnlast
  cmp al, 60h
  jnc dwnlast
  add al, 20h          ; convert to lower
```



```
  mov [si] , al
dwnlast:
  inc si
  loop chkdn
  ret
tolower endp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; upper to lower and vice vrsa
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
uplow proc
mov si, offset Kb_buff
  mov ch, 00h
  mov cl, actlen       ; cx contains the count of characters
                       entered by user
again:
  mov al, [si]
xor al, 00100000b    ; you can alter D5 by xoring al with
                     ;00100000 binary value
  mov [si], al
  inc si
  loop again
  ret
uplow endp
end main
;
```

12. **Write a program in 8086 to sort the numbers stored in an array.** *[2068 Jestha]*

⇒
```
.model small
.stack
.data
array db 4,37,13,50,2,'$'
.code
main proc far
```

```
mov ax,@data
mov ds,ax
next:
mov si,offset array
mov bl,0
mov cx,4
again:
mov al,[si]
inc si
cmp al,[si]
jc down
mov dl,[si]
mov [si],al
dec si
mov [si],dl
inc si
mov bl,01
down:
loop again
dec bl
jz next
mov ax,4c00h
int 21h
main endp
end main
```

13. Write a program in 8086 to read a number and display the multiplication table of that number. *[2068 Magh]*

⇒

```
.model small
.stack
.data
.code
main proc far
mov ax, @data
mov dx, ax
```



```
mov ah, 07h        ; console input without echo
int 21h
mov cx, 10
mov dl, 1
sub al, 30h        ; to decimal value
label1:
mov ah, 0
push ax
mul dl             ; ax=al*dl
                   ; for display
push cx
push dx
mov bx, 0
mov cx, 10

no_of_digits:
mov dx, 0
div cx             ; (dx;ax/cx=ax dx/cx)
add dx, 30h
push dx
inc bx
cmp ax, 0
jano_of_digits
mov ah, 02
mov cx, bx

popping:

pop dx
int 21h
loop popping

mov dl, 32
mov ah, 02h
int 21h

pop dx
```

```
pop cx
        ; display ends here
inc dl
pop ax
loop label1

mov ax, 4c00h
int 21h
main endp
end main
```

14. Write a program in 8086 to read a string and count the number of vowels, consonants, numerals and other characters and display the count. *[2068 Bhadra]*

⇒

```
        .model small
        .stack 100h
        .data
        s1 db 'Enter a string $'
        vowl db 'AEIOUaeiou $'
        S3 db 10, 13, 'No of vowels are  $'
        S2 db 50 dup ('$')
        count dw ?
.code
        main proc
        mov ax, @data
        mov ds, ax
        mov es, ax          ; display 'enter the string'
        lea dx, s1
        mov ah, 09h
        int  21h            ;read the string
        lea di, s2
        xor cl, cl
        cld
loop1: mov ah, 01h
        cmp al, 0dh
        jz endinpt
```



```
        stosb
        inc cl
        jmp loop1
endinpt:                    ;checking vowel
        mov  bl, cl
        cmp cl, 00h
        jnz jumpl
        xor ax, ax
        jmp pr
        jumpl:
        cld
        lea si,s2
        next: mov cx, 000bh
        lodsb
        lea di,vowl
        repne scasb
        cmp cx,00
        jz novowl
        inc cl
novowl: dec bl
jnz next                    ; jump to next line
mov dl, 0ah
mov ah, 02h
int 21h                     ; print count of vowels
mov ax, cx
pr:     mov bx,000ah
xor cx, cx; printing multidigit number;  push into stack
print: xor dx, dx
div bx
push dx
inc cx
inc cx
cmp ax, 0000h
jne print;display 'No of vowels are '
lea dx, s3
```

```
mov ah, 09h
int 21h; pop from stack.
        display:
        pop dx
add dl, 30h
mov ah, 02h
int 21h
loop display
mov ah, 4ch
int 21h
main endp
end main
```

15. **Write an assembly language program for 8086 to read a string. Display each word in separate lines in a cleared screen, count how many words there are, and display the count.** *[2070 Bhadra]*

⇒

```
.model small
.stack
.data
   maxchar db 50
   nchars db ?
   actstr db 50 dup (0)
   wcount db 1
.code
main proc far
   mov ax,@data
   mov ds,ax
mov ah,0ah
mov dx,offset maxchar
int 21h
 mov ax,0
 int 10h
mov cx,0
 mov cl,nchars
 mov si,offset actstr
 mov ah,02h
```



```
mov dl,0ah
int 21h
mov dl,0dh
int 21h
again:
mov al,[si]
cmp al,20h
mov dl,al
jne down
      inc wcount
      mov dl,0ah
      int 21h
      mov dl,0dh
down: int 21h
inc si
loop again
  mov dl,0ah
  int 21h
  mov dl,0dh
  int 21h
mov dl,wcount
or dl,30h
int 21h
mov ax,4c00h
int 21h
main endp
end main
```

16. **Write an assembly language to read a text from keyboard, convert the text into uppercase and display on cleared screen.** *[2071 Bhadra]*

⇒

```
.model small
.stack
.data
 maxchar db 50
 nchars db ?
 actstr db 50 dup (0)
```

```
.code
main proc far
  mov ax,@data
  mov ds,ax
mov ah,0ah
mov dx,offset maxchar
int 21h
  mov cl,nchars
  mov si,offset actstr
mov ax,0
int 10h
mov ah,02
again:
  mov al,[si]
  mov dl,al
  cmp al,41h
  jb reject
  cmp al,7ah
  ja reject
  cmp al,51h
  ja check
check:cmp al,61h
    jb reject
  next:int 21h
  reject:inc si
    loop again
mov ax,4c00h
int 21h
main endp
end main
```

17. Write a program in 8086 to read a string and display each word in a separate line in the center of the screen.

[2072 Magh]

⇒

```
.model small
.stack 64h
```



```
.data
    srdb "enter the string: ",'$'
        maxldb254
        actldb ?
        strdb255 dup(?)
        rowdb 12
        coldb 40
.code
main proc
    movax, @ data
    mov ds, ax
    mov dx, offset sr
    mov ah, 09h
    int 21h
        mov ah, 0ah
        lea dx, maxl
        int 21h
        callnextline
        movbx, 00
        mov cl, actl
        movch, 00
l1: cmpstr [bx], 32
      jnz s1
      inc row
      mov col, 40
          callnextline
          jmp s2
s1: mov al,str [bx]
    call display
    inc col
    s2:     inc bx
            loop l1
            call nextline
            mov ax, 4c00h
            int 21h
```

```
main endp

nextline proc near
mov ah, 02h
    mov dl, 0ah
int 21h
mov dl, 0dh
int 21h

    mov ah, 02h
movbh, 00h
mov dh, row
mov dl, col
    int 10h
nextline endp

display proc near
        mov ah, 09h
        mov al, dl
        mov bh, 00h
        mov bl, 07h
        mov cx, 00h
        int 01h
display endp

end main
```

18. **Two tables contain ten 16-bit data each. Write an assembly language program to generate the 3rd table which contains 1FFFH if the corresponding data in the 1st table is less than that of 2nd table, else store 0000H.** *[2073 Magh]*

⇒

```
.model small
.stack 32
.data
        arr1 dw
1145h,7898h,5224h,3969h,8422h,4598h,3574
h,9526h,5893h,6587h
```



```
        arr3 dw
8263h,9200h,2301h,1234h,9156h,3468h,0034
h,9265h,5213h,6157h
        arr3 dw 10 dup(0)
.code
main proc far
        movax, @data
        mov ds, ax
        mov cx, 10
        movbx, 0000h
next:   movax, arr1[bx]
        cmpax, arr2[bx]
        jnc bel
        mov arr3[bx], 1fffh
        jmp bel1
bel:    mov arr3[bx], 0000h
bel1:   inc bx
        loop next
        mov ax, 4c00h
        int 21h
main endp
end main
```

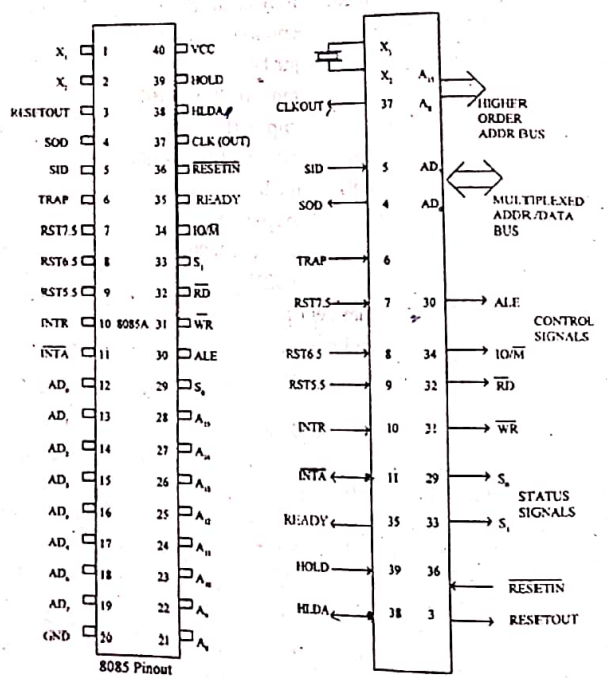
❖ ❖ ❖

Chapter - 4

MICROPROCESSOR SYSTEM

A microcomputer consists of a set of components or modules of three basic types CPU memory and I/O units which communicate with each other.

PIN Configuration of 8085



(a) (b)

Fig.: (a) Pin diagram of 8085 (b) Logical schematic of pin diagram

- The microprocessor is a clock-driven semiconductor device consisting of electronic logic circuits manufactured by using either a large-scale integration (LSI) or very-large-scale integration (VLSI) technique.
- The microprocessor is capable of performing various computing functions and making decisions to change the sequence of program execution.
- In large computers, a CPU implemented on one or more circuit boards performs these computing functions.
- The microprocessor is in many ways similar to the CPU, but includes the logic circuitry, including the control unit, on one chip.
- The microprocessor can be divided into three segments for the sake clarity, arithmetic/logic unit (ALU), register array, and control unit.
- 8085 is a 40 pin IC, DIP package. The signals from the pins can be grouped as follows

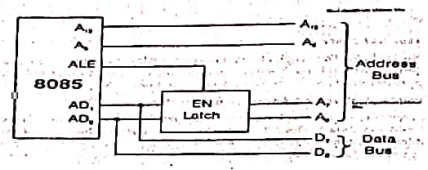
1. Power supply and clock signals
2. Address bus
3. Data bus
4. Control and status signals
5. Interrupts and externally initiated signals
6. Serial I/O ports

1. **Power supply and clock frequency signals:**

- Vcc : + 5 volt power supply
- Vss : Ground
- X1, X2 : Crystal or R/C network or LC network connections to set the frequency of internal clock generator.
- The frequency is internally divided by two. Since the basic operating timing frequency is 3 MHz, a 6 MHz crystal is connected externally.
- CLK (output)-Clock Output is used as the system clock for peripheral and devices interfaced with the microprocessor.

2. **Address bus:**
   - A8 - A15
   - It carries the most significant 8 bits of the memory address or the 8 bits of the I/O address.

3. **Multiplexed address / data bus:**
   - AD0 - AD7
   - These multiplexed set of lines used to carry the lower order 8 bit address as well as data bus.
   - During the opcode fetch operation, in the first clock cycle, the lines deliver the lower order address A0 - A7.
   - In the subsequent IO / memory, read / write clock cycle the lines are used as data bus.
   - The CPU may read or write out data through these lines.



4. **Control and status signals:**

These signals include two control signals ($\overline{RD}$ & $\overline{WR}$) three status signals (IO/ $\overline{M}$, S_1 and S_0) to identify the nature of the operation and one special signal (ALE) to indicate the beginning of the operations.

- **ALE (output) - Address Latch Enable.**
  - This signal helps to capture the lower order address presented on the multiplexed address/ data bus. When it is the pulse, 8085 begins an operation. It generates AD_0 - AD_7 as the separate set of address lines A_0 –A_7.
- $\overline{RD}$ **(active low) - Read memory or I/O device.**
  - This indicates that the selected memory location or I/O device is to be read and that the data bus is

ready for accepting data from the memory or I/O device.

- $\overline{WR}$ **(active low) - Write memory or I/O device.**
  - This indicates that the data on the data bus is to be written into the selected memory location or I/O device.
- IO/ $\overline{M}$ **(output) - Select memory or an I/O device.**
  - This status signal indicates that the read / write operation relates to whether the memory or I/O device.
  - It goes high to indicate an I/O operation.
  - It goes low for memory operations.

5. **Status Signals:**

It is used to know the type of current operation of the microprocessor.

IO/M (Active Low)	S_1	S_0	Data Bus Status (Output)
0	0	0	Halt
0	0	1	Memory WRITE
0	1	0	Memory READ
1	0	1	IO WRITE
1	1	0	IO READ
0	1	1	Opcode fetch
1	1	1	Interrupt acknowledge

6. **Interrupts and Externally initiated operations:**
   - They are the signals initiated by an external device to request the microprocessor to do a particular task or work.
   - There are five hardware interrupts called,

TRAP
RST 7.5
RST 6.5
RST 5.5
INTR
(inputs)

INTA (active low output)

- On receipt of an interrupt, the microprocessor acknowledges the interrupt by the active low, $\overline{INTA}$ (Interrupt Acknowledge) signal.

Hold (Input)

- This indicates peripheral controller requesting the bus.

HLDA (Output)

- This indicates the acknowledgement for the Hold request.

READY (Input)

- It is used to delay the microprocessor read and write cycles until a slow responding peripheral is ready to send or accept data.
- Memory and I/O devices will have slower response compared to microprocessors.
- Before completing the present job such a slow peripheral may not be able to handle further data or control signal from CPU.
- The processor sets the READY signal after completing the present job to access the data.
- The microprocessor enters into WAIT state while the READY pin is disabled.

$\overline{RESETIN}$ (Input, active low)

- This signal is used to reset the microprocessor.
- The program counter inside the microprocessor is set to zero.
- The buses are tri-stated.

RESETOUT (Output)

- It indicates CPU is being reset.
- Used to reset all the connected devices when the microprocessor is reset.

7. Single Bit Serial I/O ports:

- SID (input) - Serial input data line
- SOD (output) - Serial output data line
- These signals are used for serial communication.

Pin Configuration of 8086

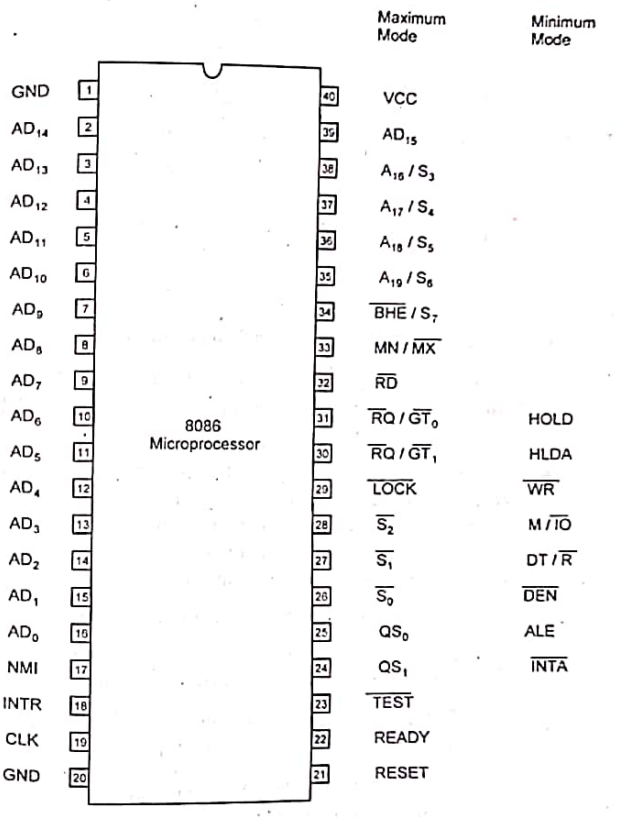


Figure: Pin Configuration for 8086 Microprocessor

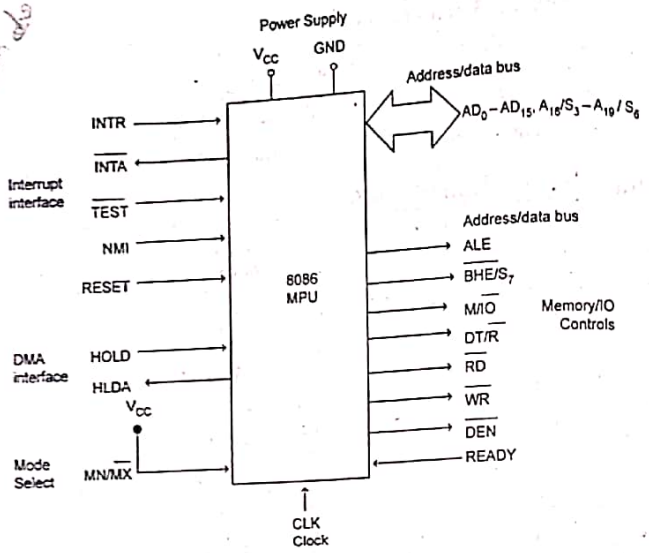


Fig.. Pin details with signal groups fir 8086 microprocessor

- The Microprocessor 8086 is a 16-bit CPU available in different clock rates and packaged in a 40 pin CERDIP or plastic package.
- The 8086 operates in single processor or multiprocessor configuration to achieve high performance. The pins serve a particular function in minimum mode (single processor mode) and other function in maximum mode configuration (multiprocessor mode).
- The 8086 signals can be categorized in three groups.
  - The first are the signal having common functions in minimum as well as maximum mode.
  - The second are the signals which have special functions for minimum mode
  - The third are the signals having special functions for maximum mode.

The following signal descriptions are common for both modes.

- **AD15-AD0:** These are the time multiplexed memory I/O address and data lines.
- **A19/S6,A18/S5,A17/S4,A16/S3:** These are the time multiplexed address and status lines. The address bits are separated from the status bit using latches controlled by the ALE signal.
- $\overline{BHE}$ **/S7:** The bus high enable is used to indicate the transfer of data over the higher order (D15-D8) data bus.
- $\overline{RD}$ **(Read):** This signal on low indicates the peripheral that the processor is performing memory or I/O read operation.
- $\overline{WR}$ **(Write):** This signal on low indicates the peripheral that the processor is performing memory or I/O write operation.
- **READY:** This is the acknowledgement from the slow device or memory that they have completed the data transfer.
- **INTR (Interrupt request):** This is to determine the availability of the request from external devices. If any interrupt request is pending, the processor enters the interrupt acknowledge cycle.
- $\overline{TEST}$ **:** This input is examined by a 'WAIT' instruction. If the $\overline{TEST}$ pin goes low, execution will continue, else the processor remains in an idle state.
- **CLK (Clock input):** The clock input provides the basic timing for processor operation and bus control activity.

The following pin functions are for the minimum mode operation of 8086.

- **M/**$\overline{IO}$ **(Memory/IO) :** When it is low, it indicates the CPU is having an I/O operation, and when it is high, it indicates that the CPU is having a memory operation.

- $\overline{INTA}$ (Interrupt acknowledge): This signal is used for interrupt acknowledge. When it goes low, the processor has accepted the interrupt.
- ALE (Address latch enable): This output signal indicates the availability of the valid address on the address/data lines.
- DT/$\overline{R}$ (Data transmit/receive): This output is used to decide the direction of data flow through the trans-receivers (bidirectional buffers). When the processor sends out data, this signal is high and when the processor is receiving data, this signal is low.
- $\overline{DEN}$ (Data enable): This signal indicates the availability of valid data over the address/data lines. It is used to enable the trans-receivers (bidirectional buffers) to separate the data from the multiplexed address/data signal.
- HOLD, HLDA-Acknowledge: When the HOLD line goes high, it indicates to the processor that another master is requesting the bus access. The processor, after receiving the HOLD request, issues the hold acknowledge signal on.

The following pin functions are applicable for maximum mode operation of 8086.

- S_2, S_1, S_0 (Status lines): These are the status lines which reflect the type of operation, being carried out by the processor.

S_2	S_1	S_0	Indication
0	0	0	Interrupt Acknowledge
0	0	1	Read I/O port
0	1	0	Write I/O port
0	1	1	Halt
1	0	0	Code Access
1	0	1	Read Memory
1	1	0	Write Memory
1	1	1	Passive

- $\overline{LOCK}$: This output pin indicates that other system bus master will be prevented from gaining the system bus, while the $\overline{LOCK}$ signal is low.
- $\overline{RQ}$ / $\overline{GT_0}$, $\overline{RQ}$ / $\overline{GT_1}$ (Request/grant): These pins are used by the other local bus master in maximum mode, to force the processor to release the local bus at the end of the processor current bus cycle.

Bus Structure

A microcomputer consists of a set of components or modules of three basic types CPU memory and I/O units which communicate with each other. A bus is a communication pathway between two or more such components. A bus actually consists of multiple communication pathway or lines. Each line is capable of transmitting signals representing binary 1 and 0. Several lines of the bus can be used to transmit binary data simultaneously. The bus that connects major microcomputer components such as CPU, memory or I/O is called the system bus. System bus consists of number of separate lines. Each line assigned a particular function. Fundamentally in any system bus the lines can be classified into three group buses.

1. Data bus: Data bus provides the path for monitoring data between the system modules. The bus has various numbers of separate lines like 8, 16, 32, or 64. Which referred as the width of data bus .These number represents the no. of bits they can carry because each carry 1 bit.
2. Address bus: Each Lines of address bus are used to designate the source or destination of the data on data bus. For example, if the CPU requires reading a word (8, 16, 32) bits of data from memory, it puts the address of desired word on address bus. The address bus is also used to address I/O ports. Bus width determines the total memory the up can handle.

3. **Control bus:** The control bus is a group of lines used to control the access to control signals and the use of the data and address bus. The control signals transmit both command and timing information between the system modules. The timing signals indicate the validity of data and address information, where as command signals specify operations to be performed. Some of the control signals are:

- Memory Write ($\overline{\text{MEMW}}$): It causes data on the bus to be loaded in to the address location.
- Memory Read ($\overline{\text{MEMR}}$): It causes data from the addressed location to be placed on the data bus.
- I/O Write ($\overline{\text{IOW}}$): It causes the data on the bus to be output to the addressed I/O port.
- I/O Read $\overline{\text{IOR}}$ It causes the data from the addressed I/O port to be placed on the bus.
- Transfer Acknowledge: This signal indicates that data have been accepted from or placed on the bus.
- Bus Request: It is used to indicate that a module wants to gain control of the bus.
- Bus Grant: It indicates that a requesting module has been granted for the control of bus.
- Interrupt Request: It indicates that an interrupt has been pending.
  Interrupt Acknowledge: it indicates that the pending interrupt has been recognized.
- The types of bus are explained as follows:

1. **Synchronous bus:** In a synchronous bus, the occurrence of the events on the bus is determined by a clock . The clock transmits a regular sequence of 0s & 1s of equal duration. All the events start at beginning of the clock cycle.

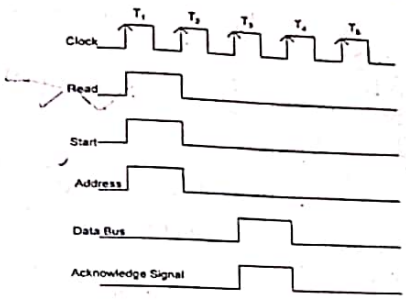


Fig: Synchronous Read Operation

Here, the CPU issues a START signal to indicate the presence of address and control information on the bus.

Then it issues the memory read signal and places the memory address on the address bus.

The addressed memory module recognizes the address and after a delay of one clock cycle it places the data and acknowledgment signal on the buses. In synchronous bus, all devices are tied to a fixed rate, and hence the system cannot take advantage of device performance but it is easy to implement.

2. **Asynchronous bus:** In an asynchronous bus, the timing is maintained in such way that occurrence of one event on the bus follows and depends on the occurrence of previous event.

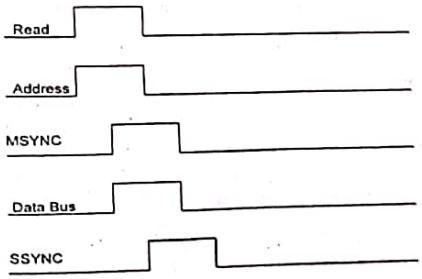


Fig: Asynchronous Read Operation

- Here, the CPU places Memory Read (Control) and address signals on the bus.
- Then it issues master synchronous signal (MSYNC) to indicate the presence of valid address and control signals on the bus.
- The addressed memory module responds with the data and the slave synchronous signal (SSYNC)

Machine Cycles and Bus Timing Diagrams

Operation of a microprocessor can be classified in to following four groups according to their nature.

- Op- Code fetch
- Memory Read /Write
- I/O Read/ Write
- Request acknowledgement

Here Op-Code fetch is an internal operation and other three are external operations. During three operations, microprocessor generates and receives different signals. These all operations are terms as machine cycle.

- Clock Cycle (T state): It is defined as one subdivision of the operation performed in one clock period.
- Machine Cycle: It is defined as the time required to complete one operation of accessing memory, I/O, or acknowledging an external request. This cycle may consist of three to six T-states.
- Instruction Cycle: It is defined as the time required completing the execution of an instruction. The 8085 instruction cycle consists of one to six machine cycles or one to six operations.

Op-code fetch machine cycle

The first operation in any instruction is op-code fetch. The microprocessor needs to get (fetch) this machine code from the memory register where it is stored before the microprocessor can begin to execute the instruction.

Let's consider the instruction MOV C, A stored at memory location 2005H. The Op-Code for the instruction is 4FH and Op-Code fetch cycle is of 4 clock cycles.

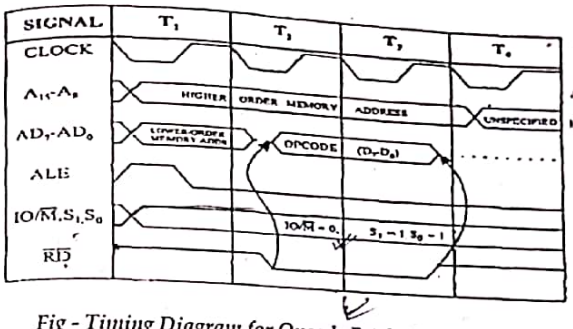


Fig - Timing Diagram for Opcode Fetch Machine Cycle

- Step1: Microprocessor places the 16 bit memory address from Program Counter on the address bus. At T_1, high order address (20) is placed at A_8-A_{15} and lower order address (05) is placed at AD_0- AD_7 ALE signal goes high. IO/$\overline{M}$ goes low and both S_0 and S_1 goes high for Op-Code fetch.
- Step 2: The control unit sends the control signal $\overline{RD}$ to enable the memory chip and active during T_2 and T_3.
- Step 3: The byte from the memory location is placed on the data bus that is 4f into D_0-D_7 and RD goes high impedance.
- Step4: The instruction 4FH is decoded and content of accumulator will be copied into register C during clock cycle T_4.

Memory read machine cycle :

Let's consider the instruction MVI A, 32 H stored at memory location 2000H.

2000	3EH	MVI	A,	32	H
2001	32H				

Here two machine cycles are presented, first is Op-Code fetch which consists of 4 clock cycles and second is memory read consist of 4 clock cycle.

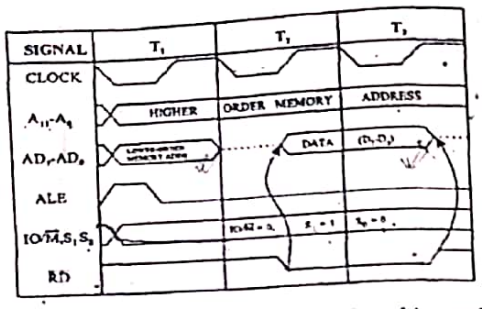


Fig.: Timing diagram for memory read machine cycle

- **Step 1 :** First machine cycle (Op-Code fetch) is identical for timing diagram of Op-Code fetch cycle.
- **Step 2:** After completion of Op-Code fetch cycle, 8085 places the address 2001 on the address bus and increments PC to 2002H. ALE is asserted high, IO/$\overline{M}$ =0, S_1=1, S_0=0 for memory read cycle. When RD =0, memory places the data byte 32H on the dada bus.

Memory write machine cycle:

The memory write machine cycle is executed by the processor to write a data byte in a memory location. The processor takes, 3T states to execute this machine cycle.

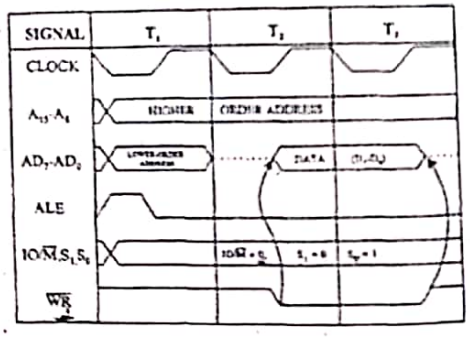


Fig.: Timing diagram for memory write machine cycle

I/O read machine cycle:

- The I/O Read cycle is executed by the processor to read a data byte from I/O port or from the peripheral, which is I/O, mapped in the system.
- The processor takes 3T states to execute this machine cycle.
- The IN instruction uses this machine cycle during the execution.

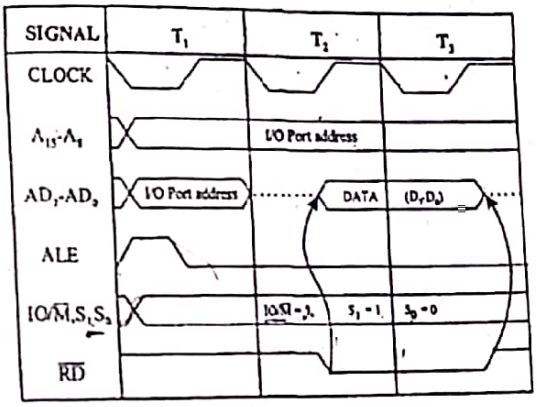


Fig - Timing Diagram for I/O Read Machine Cycle

I/O Write Cycle:

Let's consider the instruction OUT 01H stored at memory location 2050H.

2050	D3	OUT 01H
2051	01	

Op-Code Fetch Cycle	4T
Memory read Cycle	3T
I/O Write Cycle	3T

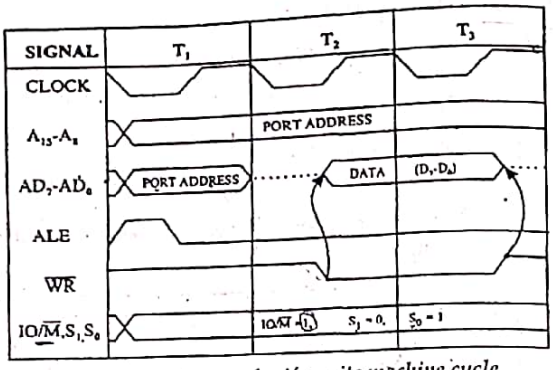


Fig.: Timing diagram for i/o write machine cycle

- **Step 1:** In Machine Cycle M_1, the microprocessor sends $\overline{RD}$ control signal which is combined with IO/ M to generate the MEMR signal and processor fetches instruction code D3 using the data bus.
- **Step 2:** In 2nd Machine Cycle M_2, the 8085 microprocessor places the next address 2051 on the address bus and gets the device address 01H via data bus.
- **Step 3:** In machine Cycle M_3, the 8085 places device address 01H on low order as well as high order address bus. IO/ M goes high for IO and accumulator content are placed on Data bus which are to be written into the selected output port.

Timing diagram for STA 526AH

- STA means Store Accumulator -The contents of the accumulator is stored in the specified address(526A).
- The opcode of the STA instruction is said to be 32H. It is fetched from the memory 41FFH(see fig). - *OF machine cycle*
- Then the lower order memory address is read(6A). - *Memory Read Machine Cycle*
- Read the higher order memory address (52).- *Memory Read Machine Cycle*
- The combination of both the addresses are considered and the content from accumulator is written in 526A. - *Memory Write Machine Cycle*

- Assume the memory address for the instruction and let the content of accumulator is C7H. So, C7H from accumulator is now stored in 526A.

Address	Mnemonics	Opcode
41FF	STA 526AH	32_H
4200		$6A_H$
4201		52_H

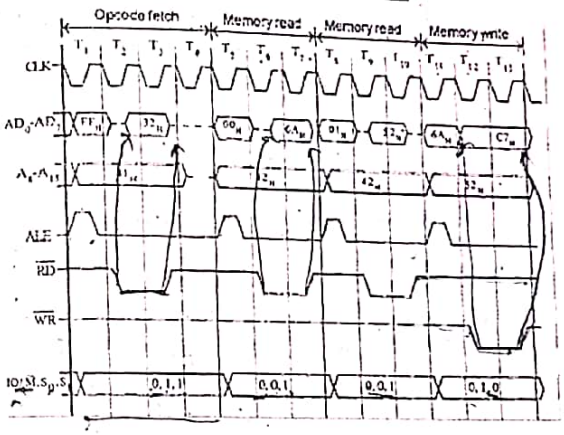


Fig.: Timing diagram for STA 526AH

Timing diagram for IN C0H.

- Fetching the Opcode DBH from the memory 4125H.
- Read the port address C0H from 4126H.
- Read the content of port C0H and send it to the accumulator.
- Let the content of port is 5EH.

Address	Mnemonics	Opcode
4125	IN $C0_H$	DB_H
4126		$C0_H$

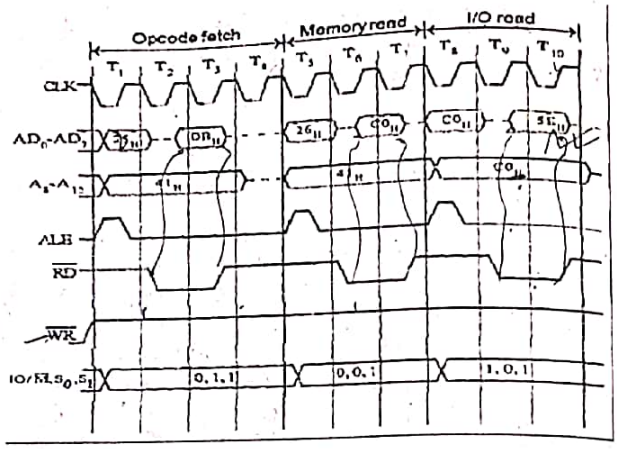
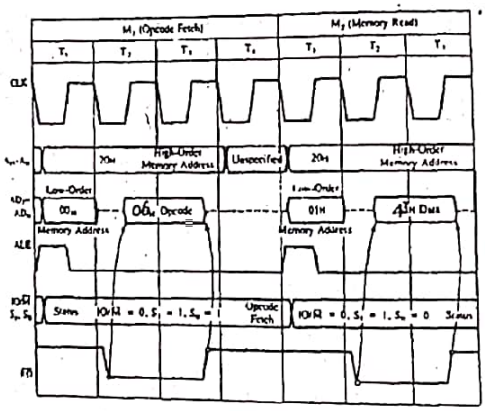


Fig.: Timing diagram for IN C0H.

Timing diagram for MVI B, 43H.



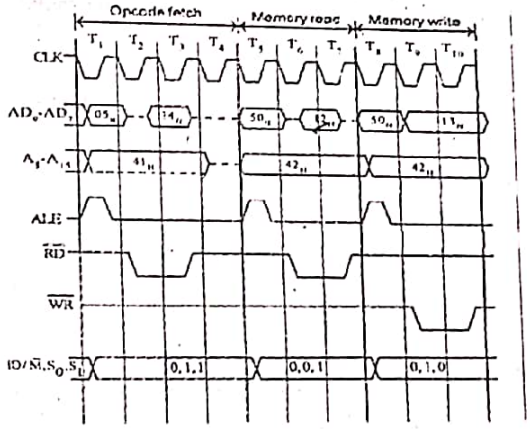
- Fetching the Opcode 06H from the memory 2000H. (OF machine cycle)
- Read (move) the data 43H from memory 2001H. (memory read)

Address	Mnemonics	Opcode
2000	MVI B, 43H	06H
2001		43H

Timing diagram for INR M

- Fetching the Opcode 34H from the memory 4105H. (OF cycle)
- Let the memory address (M) be 4250H. (MR cycle -To read Memory address and data)
- Let the content of that memory is 12H.
- Increment the memory content from 12H to 13H. (MW machine cycle)

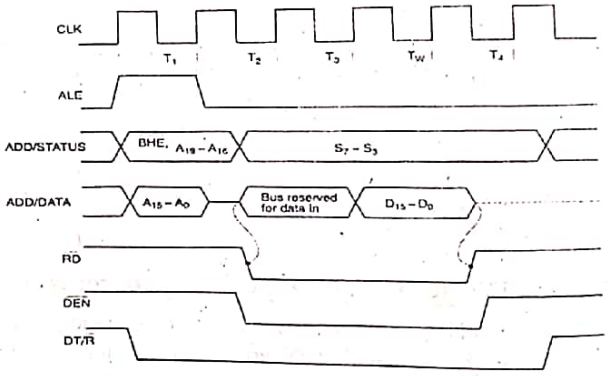
Address	Mnemonics	Opcode
4105	INR M	34H



Read and Write Bus Timing of 8086 Microprocessor

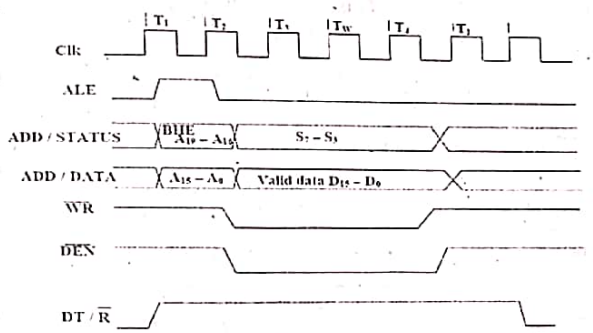
- The working of the minimum mode configuration system can be better described in terms of the timing diagrams rather than qualitatively describing the operations.

- The op-code fetch and read cycles are similar. Hence the timing diagram can be categorized in two parts, the first is the timing diagram for read cycle and the second is the timing diagram for write cycle.
- The read cycle begins in T_1 with the assertion of address latch enable (ALE) signal and also M / $\overline{IO}$ signal. During the negative going edge of this signal, the valid address is latched on the local bus.
- The $\overline{BHE}$ and A_0 signals address low, high or both bytes. From T_1 to T_4, the M/ $\overline{IO}$ signal indicates a memory or I/O operation.
- At T_2, the address is removed from the local bus and is sent to the output. The bus is then tri-stated. The read (RD) control signal is also activated in T_2.
- The read (RD) signal causes the address device to enable its data bus drivers. After RD goes low, the valid data is available on the data bus.
- The addressed device will drive the READY line high. When the processor returns the read signal to high level, the addressed device will again tristate its bus drivers



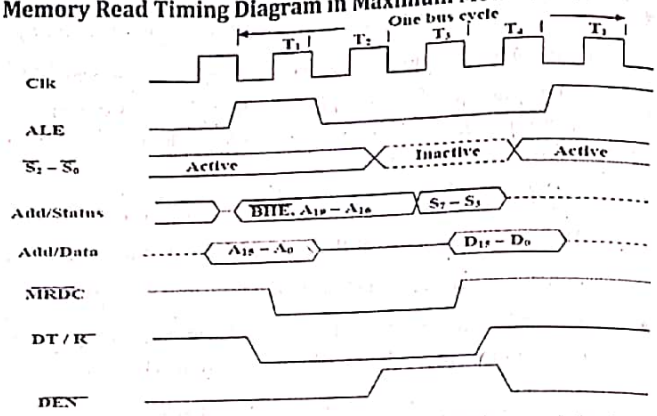
Write cycle timing diagram for minimum mode operation

- A write cycle also begins with the assertion of ALE and the emission of the address. The M/ $\overline{IO}$ signal is again asserted to indicate a memory or I/O operation. In T_2, after sending the address in T_1, the processor sends the data to be written to the addressed location.
- The data remains on the bus until middle of T_4 state. The $\overline{WR}$ becomes active at the beginning of T_2 (unlike $\overline{RD}$ is somewhat delayed in T_2 to provide time for floating).
- The BHE and A_0 signals are used to select the proper byte or bytes of memory or I/O word to be read or write.
- The M/ $\overline{IO}$, $\overline{RD}$ and $\overline{WR}$ signals indicate the type of data transfer as specified in table below.



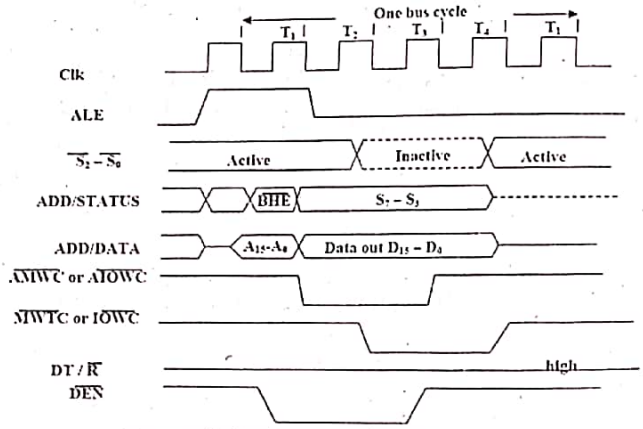
Write Cycle Timing Diagram for Minimum Mode

Memory Read Timing Diagram in Maximum Mode of 8086



Memory Read Timing in Maximum Mode

Memory Write Timing in Maximum mode of 8086



Memory Write Timing in Maximum mode.

Memory Devices

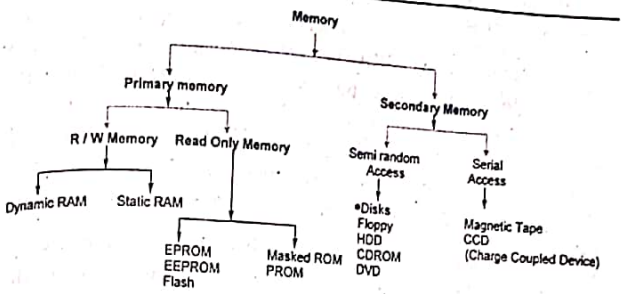


Fig: Classfication of memory system

Memory is an essential component of the microcomputer system. It is used to store both instructions and data. It is used to store both instructions and data. Memory is made up of registers and the number of bits stored in a register is called memory word .Memory word is identified by an address .If microprocessor uses 16 bit address , then there will be maximum of 2^{16}= 65536 memory addresses ranging from 0000H to FFFFH.

There are various types of memory which can be classified in to two main groups i.e. Primary memory and Secondary memory.

1. **Primary Memory:**

It is the memory used by microprocessor to execute programs. The microprocessor can access only those items that are stored in this memory. Hence, all data and program must be within primary memory prior to its execution. Primary memory is much larger than processer memory that is included in the microprocessor chip.

Primary memory is divided in to two groups.

i. **R/W Memory (RAM)**

Microprocessor can read from and write into this memory. This memory is used for information that are likely to be altered such as writing program or receiving data. This

memory is volatile i.e. the content will be lost if the power is turned off and commonly known as RAM. RAM are basically of two types.

a. **Static RAM (SRAM):** This memory is made up of flip flops and it stores bit as voltage. A single flip flop stores binary data either 1 or 0. Each flip flop is called storage cell. Each cell requires six transistors. Therefore, the memory chip has low density but high speed. This memory is more expensive and consumes more power.

b. **Dynamic RAM (DRAM):** This memory is made up of MOS transistor gates and it stores the bit as charge. The advantage of DRAM are it has high density, low power consumption and cheaper than SRAM. But the bit information leaks therefore needs to be rewritten again every few milliseconds. It is called refreshing the memory and requires extra circuitry to do this. It is slower than SRAM.

ii. **Read Only Memory (ROM):**

ROM contains a permanent pattern of data that cannot be changed. It is non volatile that is no power source is required to maintain the bit values in memory. ROM are basically of 5 types.

a. **Masked ROM:** A bit pattern is permanently recorded by the manufactures during production.

b. **Programmable ROM:** In this ROM, a bit pattern may be written into only once and the writing process is performed electrically. That may be performed by a supplier or customer.

c. **Erasable PROM (EPROM):** This memory stores a bit in the form of charge by using EPROM programmer which applies high voltage to charge the gate .Information can be erased by exposing ultra violet radiation. It is reusable. The disadvantages are :

(i) It must be taken out of circuit to erase it

(ii) The entire chip must be erased

(iii) The erasing process takes 15 to 20 minutes.

d. **Electrically Erasable PROM (EEPROM):** It is functionally same as EPROM except that information can be altered by using electrical signal at the register level rather than erasing all the information. It is expensive compared to EPROM and flash and can be erased in 10 ms.

e. **Flash Memory:** It is variation of EPROM. The difference is that EPROM can be erased in register level but flash memory must be erased in register level but flash memory must be erased in its entirety or at block level.

2. **Secondary memory**

The devices that provide backup storage are called secondary memory. It includes serial access type such as magnetic disks and random access type such as magnetic disks. It is nonvolatile memory.

Performance of memory:

1. Access time (t_a):

Read access time: It is the average time required to read the unit of information from memory.

Write access time: It is the average time required to write the unit of information on memory.

Access rate (r_a) = $1/t_a$

2. Cycle time (t_c):

It is the average time that lapses between two successive read operation .

Cycle rate (r_c)= bandwidth = $1/t_c$

Access modes of memory:

1. Random access: In random access mode, the t_a is independent of the location from which the data is accessed like MOS memory.

2. Sequential access: In that mode, the t_a is dependent of the location form which the data is accessed like magnetic type.

3. Semi random-access: the semi random access combines these two. for eg. In magnetic disk, any track can be accessed at random. But the access within the truck must be in serial fashion.

The Memory Hierarchy

- Capacity, cost and speed of different types of memory play a vital role while designing a memory system for computers.
- If the memory has larger capacity, more application will get space to run smoothly.
- It's better to have fastest memory as far as possible to achieve a greater performance. Moreover for the practical system, the cost should be reasonable.
- There is a tradeoff between these three characteristics cost, capacity and access time. One cannot achieve all these quantities in same memory module because
  - If capacity increases, access time increases (slower) and due to which cost per bit decreases.
  - If access time decreases (faster), capacity decreases and due to which cost per bit increases.
- The designer tries to increase capacity because cost per bit decreases and the more application program can be accommodated. But at the same time, access time increases and hence decreases the performance.

  So the best idea will be to use memory hierarchy.
- Memory Hierarchy is to obtain the highest possible access speed while minimizing the total cost of the memory system.
- Not all accumulated information is needed by the CPU at the same time.
- Therefore, it is more economical to use low-cost storage devices to serve as a backup for storing the information that is not currently used by CPU
- The memory unit that directly communicate with CPU is called the *main memory*

- Devices that provide backup storage are called *auxiliary memory*
- The memory hierarchy system consists of all storage devices employed in a computer system from the slow by high-capacity auxiliary memory to a relatively faster main memory, to an even smaller and faster cache memory
- The main memory occupies a central position by being able to communicate directly with the CPU and with auxiliary memory devices through an I/O processor
- A special very-high-speed memory called cache is used to increase the speed of processing by making current programs and data available to the CPU at a rapid rate
- CPU logic is usually faster than main memory access time, with the result that processing speed is limited primarily by the speed of main memory
- The cache is used for storing segments of programs currently being executed in the CPU and temporary data frequently needed in the present calculations
- The memory hierarchy system consists of all storage devices employed in a computer system from slow but high capacity auxiliary memory to a relatively faster cache memory accessible to high speed processing logic. The figure below illustrates memory hierarchy.

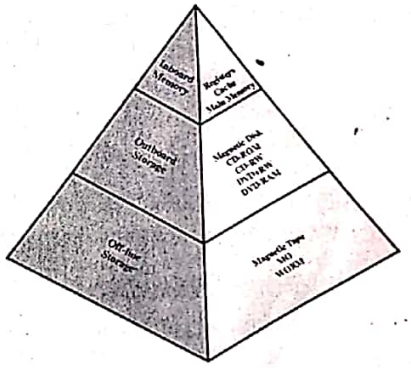


Fig: Memory Hierarchy

- As we go down in the hierarchy
  - Cost per bit decreases
  - Capacity of memory increases
  - Access time increases
  - Frequency of access of memory by processor also decreases.

Hierarchy List

- Registers
- L1 Cache
- L2 Cache
- Main memory
- Disk cache
- Disk
- Optical
- Tape

Address Decoding

Microprocessor is connected with memory and I/O devices via common address and data bus. Only one device can send data at a time and other devices can only receive that data. If more than one device sends data at the same time, the data gets garbled. In order to avoid this situation, ensuring that the proper device gets addressed at proper time, the technique called address decoding is used.

In address decoding method, all devices like memory blocks, I/O units etc. are assigned with a specific address. The address of the device is determined from the way in which the address lines are used to derive a special device selection signal known as chip select ($\overline{CS}$). If the microprocessor has to write or to read from a device, the $\overline{CS}$ signal to that block should be enabled and the address decoding circuit must ensure that $\overline{CS}$ signal to other devices are not activated.

Depending upon the no. of address lines used to generate chip select signal for the device, the address decoding is classified as:

a. **I/O mapped I/O**

In this method, a device is identified with an 8 bit address and operated by I/O related functions IN and OUT for that IO/ $\overline{M}$ =1. Since only 8bit address is used, at most 256 bytes can be identified uniquely. Generally low order address bits A_0-A_7 are used and upper bits A_8-A_{15} are considered don't care. Usually I/O mapped I/O is used to map devices like ~~8255A,~~ 8251A etc.

b. **Memory mapped I/O**

In this method , a device is identified with 16 bit address and enabled memory related functions such as STA , LDA for which IO/ $\overline{M}$ =0, here chip select signal of each device is derived from 16 bit address lines thus total addressing capability is 64K bytes . Usually memory mapped I/O is used to map memories like RAM, ROM etc.

Depending on the address that are allocated to the device the address decoding are categorized in the following two groups.

1. **Unique Address Decoding:**

If all the address lines on that mapping mode are used for address decoding then that decoding is called unique address decoding. It means all 8-lines in I/O mapped I/O and all 16 lines in memory mapped I/O are used to derive $\overline{CS}$ signal. It is expensive and complicated but fault proof in all cases.

For example, if A_0 is high and A_1- A_7 are low and if IOW becomes low, the latch gets enabled. The data to the LED can be transferred in only one case and hence the device has unique address of 01H.

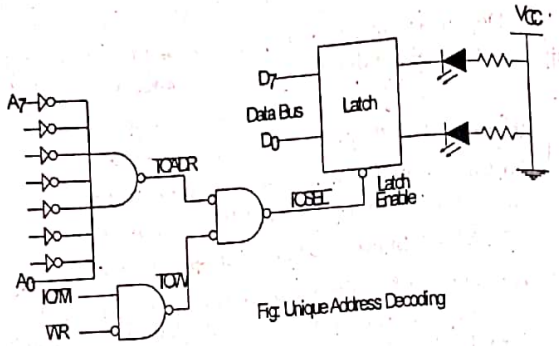


Fig: Unique Address Decoding

Similarly, for interfacing eight I/P switches at 53H (01010011); the unique address decoding circuit will be as shown.

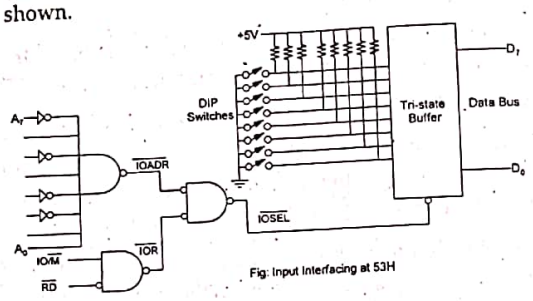


Fig: Input Interfacing at 53H

Non-unique address decoding:

If all the address lines available on that mode are not used in address decoding then that decoding is called non unique address decoding. Though it is cheaper there may be a chance of address conflict.

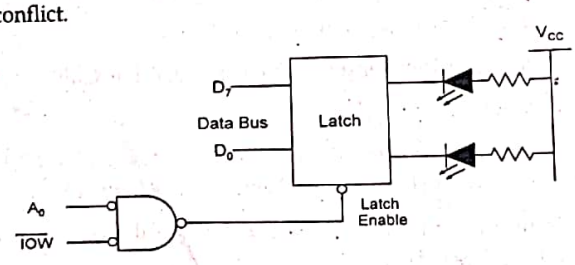


Fig: Non unique Address Decoding

- If A_0 is low and $\overline{IOW}$ is low, then latch gets enabled.

  Here A_1-A_7 is neglected that is any even address can enable the latch.

Address Decoding with 8086 Microprocessor

The 8086 microprocessor provides a 20 bit memory address that allows up to 1 MB main memory. Out of these several address lines are unused, but these extra lines determine the range of addresses the memory interface occupies. Address decoder circuit determines these extra address lines and enables the memory for a specific range of addresses. Depending up on number of lines used for decoder, we get

- **Full Decoding (Absolute Decoding):** All the unused lines (zero lines) are used.
- **Partial Decoding (Linear Decoding):** All the unused lines (zero lines) are not used.
- **Block Decoding:** Same as full decoding except that in this case blocks of memory is enabled using the unused lines.

Important Points to be considered for Memory Interfacing

- After reset CS contains FFFFH and IP contains 0000H. Therefore, the physical address is FFFF0H. Here instruction execution starts from FFFF0H which is normally a jump from some other location where a longer program resides. This program always resides in a ROM. For example, we want to interface 4 chips of 2K memory that means 8K bytes of memory requires 13 address lines A0 to A12. So, 8K means 01FFFH bytes, therefore EPROM address starts from FFFFFH - 01FFFH = FE000H.
- Since ROMs and EPROMs are read-only devices, A_0 and $\overline{BHE}$ are not required to be part of the chip enable/select decoding. The 8086 address lines must be connected to the ROM/EPROM chip starting with A_1 and higher to all the

address lines of the ROM/EPROM chips. The 8086 unused address lines can be used as chip enable/select decoding.

- Since static RAMs are read/write memories, both A_0 and BHE' must be included in the chip select/enable decoding of the devices and write timing must be considered in the compatibility analysis.

EPROM Interfacing with 8086

Whenever the 8086 CPU is reset, its value is set to FFFFH and IP value is set to 0000H that corresponds to physical address FFFF0H which is always a part of ROM. This means FFFF0H to FFFFFH should be always included in the ROM.

Let us take an example to address 16 KB of EPROM to 8086 microprocessor, 16 KB means 3FFFH bytes. Hence the EPROM memory should start from FFFFFH – 03FFFH = FC000H.

To address 16 KB, we require 14 address lines. Of the 16 KB, 8 KB will be at even addresses and 8 KB will be at odd addresses. Hence, we use 2 EPROM chips, each of 8 KB capacity one for storing bytes at even address and another for storing bytes for odd address.

	A_{19}	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_{09}	A_{08}	A_{07}	A_{06}	A_{05}	A_{04}	A_{03}	A_{02}	A_{01}	A_{00}
Start:	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
End:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	Used for Chip Select						Address within the 16KB													

The EPROM address ranges from FC000H to FFFFFH

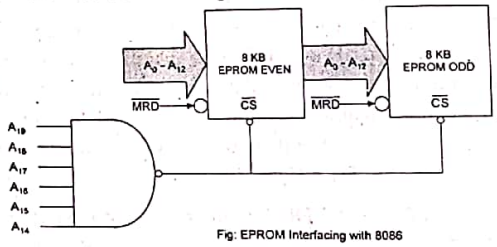


Fig: EPROM Interfacing with 8086

A_0 and BHE' is not used for interfacing of EPROM. Both the 8 KB EPROM chips are selected whenever any address in the range FC000H – FFFFFH comes on the address bus.

Static RAM Interfacing with 8086

The general procedure of static memory interfacing with 8086 is briefly described as follows.

- Arrange the available memory chips so as to obtain 16 bits data bus with the upper 8 bit bank is called "Odd address memory bank" and the lower 8 bit bank is called "Even address memory bank".
- Connect available memory address lines of memory chips with those of the microprocessor and also connect the memory RD' and WR' inputs to the corresponding processor control signals. Connect the 16 bit data bus of the memory bank was that of microprocessor 8086.
- The remaining address lines of the microprocessor, BHE' and A_0 are used for loading the required chip select signals for the odd and even memory banks. The CS' of the memory is derived from the O/P of the decoding.

Let us take an example to address 16 KB of RAM to 8086 microprocessor. We will split 16 KB into two blocks each of 8 KB one of even addressed and another of odd addressed. Depending upon the bit on A_0, either the even or odd bank will be selected. If A_0 = 0, the even bank is selected. Now the BHE' signal will be 0 whenever a byte or a word is being accessed at odd address. Also for an even addressed word, both the banks will have to be enabled at the same time. Hence, A_0 has to be given to CS' of even bank and BHE' has to be given to CS' of odd bank.

- If only A0 is low, memory location from even bank is accessed.
- If only BHE' is low, memory location from odd bank is accessed.
- If both are low, 2 memory locations are accessed from each bank.

To address 16 KB, we require 14 address lines. Of the 16 KB, 8 KB will be at even addresses and 8 KB will be at odd addresses. Hence, we use 2 RAM chips, each of 8 KB capacity one for storing

bytes at even address and another for storing bytes for odd address. We will start the RAM address from 80000H.

	A_{19}	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_{09}	A_{08}	A_{07}	A_{06}	A_{05}	A_{04}	A_{03}	A_{02}	A_{01}	A_{00}
Start:	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
End:	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
							Block A													
Start:	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
End:	1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
							Block B													

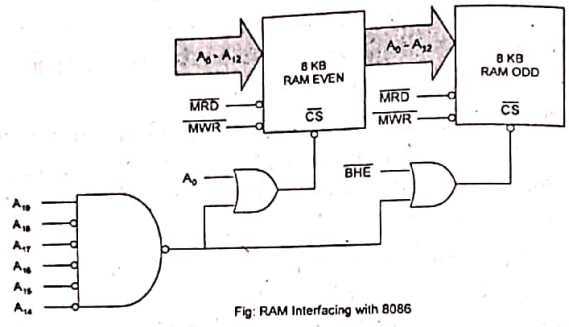
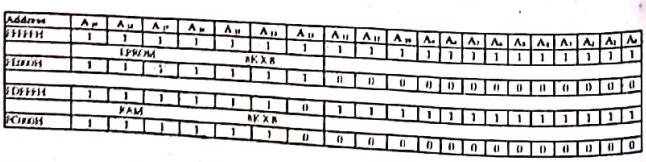


Fig: RAM Interfacing with 8086

As a good and efficient interfacing practice, the address map of the system should be continuous as far as possible i.e. these should be no windows and no feedback space should be allowed. A memory location should have a single address corresponding to it i.e., absolute decoding should be preferred.

 # Interface two 4K X 8 EPROMs and two 4K X 8 RAM chips with 8086, select suitable maps.

We know that, after reset, the IP and CS are initiated to from address FFFF0H. Hence this address must lie in the EPROM. The address of RAM may be selected anywhere in the 1MB space of 8086, but we will select the RAM address such that the address map of the system is continuous as shown in table below.



Total 8K of EPROM need 13 address lines A_0-A_{13} (Since 2^{13} = 8K). Address lines A_{13}-A_{19} are used for decoding to generate the chip select. The $\overline{BHE}$ signal goes low when a transfer is at odd address or higher byte of data is to be accessed, let us assume that the latched address, BHE' and de-multiplexed data lines are readily available for interfacing.

The two 4K X 8 chips of RAM and EPROM are arranged in parallel to obtain 16-bit data bus width. If A_0 is 0 i.e. the address is even and is in RAM, then the lower RAM chip is selected indicating 8-bit transfer at even address. If A_0 is 1 i.e. the address is odd and is in RAM, the BHE' goes low, the upper RAM chip is selected further indicating that the 8 bit transfer is at an odd address. If the selected addresses are in ROM, the respective ROM chips are selected. If at a time A0 and BHE' both are zero, both the RAM or ROM chips are selected i.e. the data transfer is of 16 bits. The selection of chips takes place as shown in table below.

Decoder I/P Address / BHE'	A_2	A_1	A_0	Selection / Comment
	A_{13}	A_0	BHE'	
Word transfer on D_0-D_{15}	0	0	0	Even and Odd address in RAM
Byte transfer on D_0-D_7	0	0	1	Only even address in RAM
Byte transfer on D_8-D_{15}	0	1	0	Only odd address in RAM
Word transfer on D_0-D_{15}	1	0	0	Even and Odd address in ROM
Byte transfer on D_0-D_7	1	0	1	Only even address in ROM
Byte transfer on D_8-D_{15}	1	1	0	Only odd address in ROM

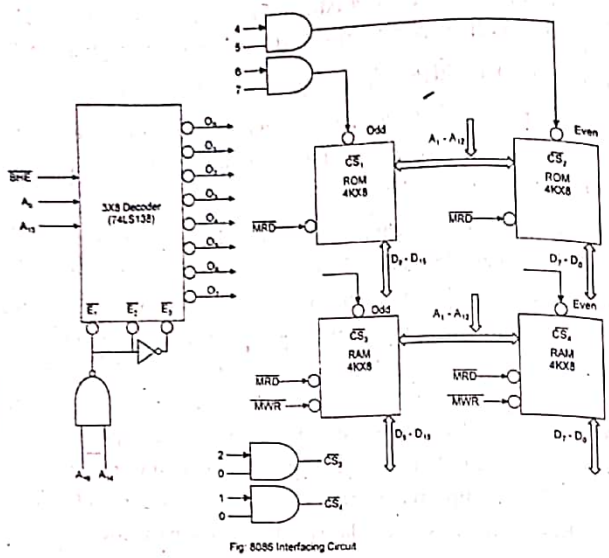


Fig: 8086 Interfacing Circuit

Input/ Output Devices

Input / Output devices are the means through which the microcomputer unit communicates with the outside world. The link between the I/O devices and the microprocessor is maintained by a circuitry known as I/O module. This circuitry includes the specific interfaces needed for I/O devices as well as control functions that implement the I/O transfers within the computer. I/O devices usually are appeared as passive devices which take action only when instructed to do. The CPU monitors the status of the I/O devices and selects them according to availability and need.

Consider the keyboard as input device and the steps when the key is pressed are

- Microprocessor detects the key change in status of keyboard i.e., the key is pressed.
- It receives the encoded information corresponding to pressed key.
- It checks the validity of required signal.

- Consider the printer as output device
- Here, microprocessor checks ideal condition of printer, if ideal then sends the data to be printed and required command for that.

For interfacing of typical microprocessor to I/O devices such as keyboard, CRT, printer etc., all need I/O interface circuits which are of mainly two types.

Parallel Interface

- The device which can handle data at higher speed cannot support with serial interface.
- N bits of data are handled simultaneously by the bus and the links to the device directly.
- Achieves faster communication but becomes expensive due to need of multiple wires.

Data transfer modes of parallel interfacing

The information exchanged between a microprocessor and an I/O interface circuit consists of input or output data and control information. The status information enable the microprocessor monitor the device and when it is ready then send or receive data. Control information is the command by microprocessor to cause I/O device to take some action. If the device operates at different speeds, then microprocessor can be used to select a particular speed of operation of the device. The techniques used to transfer data between different speed devices and computer is called synchronizing. Different techniques under synchronizing are:

i. **Simple I/O**

For simple I/O, the buffer switch and latch switches i. e. LED are always connected to the input and output ports. The devices are always ready to send or receive data.

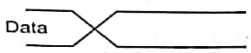


Fig: Simple I /O

Here cross line indicate the time for new valid data.

ii. **Wait Interface(Simple strobe I/O)**

In this technique, MP need to wait until the device is ready for the operation.

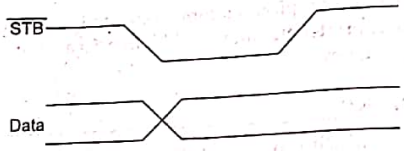


Fig: Simple Strobe I/O

Consider a simple keyboard consisting of 8 switches connected to a MP through a parallel interface CKT (Tri-state buffer). The switch is of dip switches. In order to use this keyboard as an input device the MP should be able to detect that a key has been activated. This can be done by observing that all the bits are in required order. The processor should repeatedly read the state of input port until it finds the right order of bits i.e. at least 1 bit of 8 bits should be 0.

Consider the tri-state A/D converter

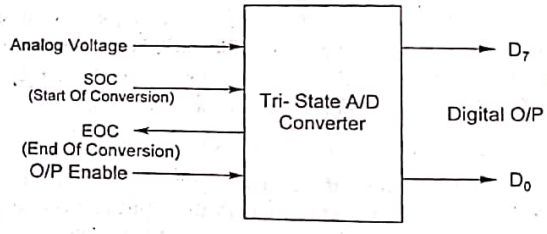


Fig: Tri-State A/D Converter

- Used to convert analog to digital data which can be read by I/O unit of MP
- When SOC appears 1, I/O unit should ready for reading binary data/digital data.

- When EOC's status is 1, then I/O unit should stop to read data.
- Strobe signal indicates the time at which data is being activated to transmit.

iii. **Single Handshaking:**

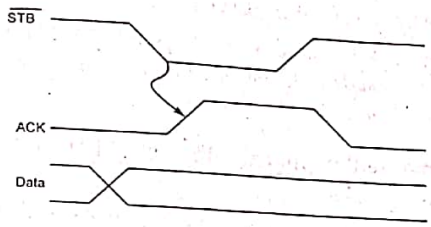


Fig: Single Handshaking

The peripheral outputs some data and send $\overline{STB}$ signal to MP. "here is the data for you."

MP detects asserted $\overline{STB}$ signal, reads the data and sends an acknowledge signal (ACK) to indicate data has been read and peripheral can send next data. "I got that one, send me another."

MP sends or receives data when peripheral is ready.

iv. **Double Handshaking**

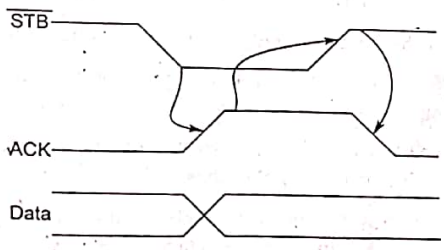


Fig: Double Handshaking

The peripheral asserts its $\overline{STB}$ line low to ask MP "Are you ready?"

The MP raises its ACK line high to say " I am ready".

Peripheral then sends data and raises its $\overline{STB}$ line low to say "Here is some valid data for you."

MP then reads the data and drops its ACK line to say, "I have the data, thank you, and I await your request to send the next byte of data."

Programmable Peripheral Interface (PPI) - 8255A

- The INTEL 8255 is a device used to parallel data transfer between processor and slow peripheral devices like ADC, DAC, keyboard, 7-segment display, LCD, etc.
- The 8255 has three ports: Port-A, Port-B and Port-C.
- Port-A can be programmed to work in any one of the three operating modes mode-0, mode-1 and mode-2 as input or output port.
- Port-B can be programmed to work either in mode-0 or mode-1 as input or output port.
- Port-C (8-pins) has different assignments depending on the mode of port-A and port-B.
- If port-A and B are programmed in mode-0, then the port-C can perform any one of the following functions.
  - As 8-bit parallel port in mode-0 for input or output.
  - As two numbers of 4-bit parallel ports in mode-0 for input or output.
  - The individual pins of port-C can be set or reset for various control applications.
- If port-A is programmed in mode- 1/mode-2 and port-B is programmed in mode-1 then some of the pins of port-C are used for handshake signals and the remaining pins can be

used as input/ output lines or individually set/reset for control application.

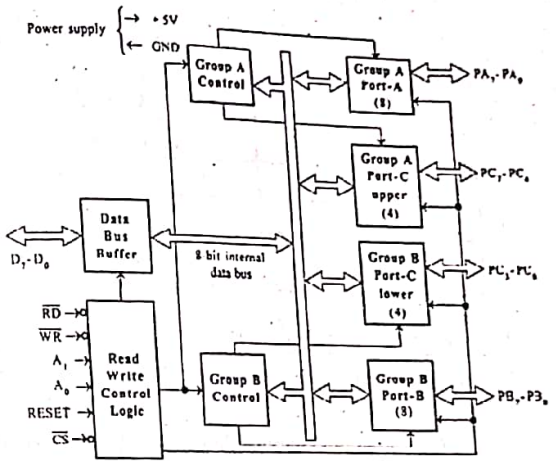


Fig2: Internal Block Diagram of 8255

Key Features of Mode-0,Mode-1 and Mode-2

Mode 0: Ports A and B operate as either inputs or outputs and Port C is divided into two 4-bit groups either of which can be operated as inputs or outputs

- Mode 1: Same as Mode 0 but Port C is used for handshaking and control
- Mode 2: Port A is bidirectional (both input and output) and Port C is used for handshaking. Port B is not used.

The read/write control logic requires six control signals. These signals are given below.

1. RD (low): This control signal enables the read operation. When this signal is low, the microprocessor reads data from a selected I/O port of the 8255A.
2. WR (low): This control signal enables the write operation. When this signal goes low, the microprocessor writes into a selected I/O port or the control register.

3. RESET: This is an active high signal. It clears the control register and set all ports in the input mode.

4. CS (low), A0 and A1: These are device select signals. They are,

Internal Devices	A_1	A_0
Port A	0	0
Port B	0	1
Port C	1	0
Control Register	1	1

Serial Interface /Serial Data Transmission

- Data are transferred serially one bit at a time starting from Least Significant bit.
- Slow due to single communication link but inexpensive to implement.
- It uses clock to separate consecutive bits.
- Its function is to deal with the data on the bus in the parallel mode and communicate with the connected device in serial mode.
- Its data bus has n data lines, the serial I/O interface accepts n bit of data simultaneously from the bus and n bits are sent one at a time thus requiring n time slots.
- Not suitable for fast operation needed microprocessor.

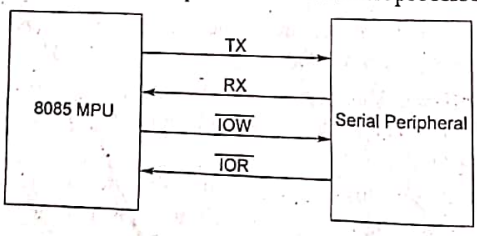


Fig: Serial Interface between microprocessor and I/O device.

Serial I/O transfer is more common than the parallel I/O. The two major forms of serial data transmission are:

1. **Synchronous serial data transmission:**
   - Data is transmitted or received based on a clock signal i. e. synchronously.
   - The transmitting device sends a data bit at each clock pulse.
   - Usually one or more SYNC characters are used to indicate the start of each synchronous data stream.
   - SYNC characters for each frame of data.
   - Transmitting device sends data continuously to the receiving device. If the data is not ready to be transmitted, the transmitter will send SYNC character until the data is available.
   - The receiving device waits for data, when it finds the SYNC characters then starts interpreting the data.

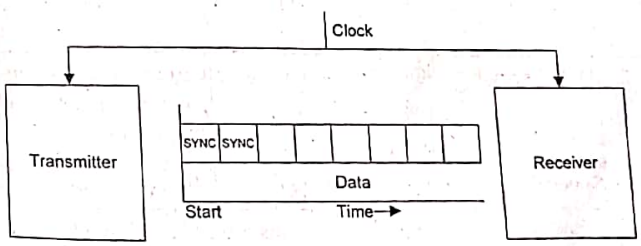


Fig: Synchronous Serial Transmission Format

2. **Asynchronous serial data transmission:**
   - The receiving device does not need to be synchronized with the transmitting device.
   - Transmitting device send data units when it is ready to send data.
   - Each data unit must contain start and stop bits for indicating beginning and the end of data unit. And also one parity bit to identify odd or even parity data.

- For e.g. To send ASCII character (7 bit)
- We need: 1 start bit: beginning of data
  1 stop bit: End of data
  1 Parity bit: even or odd parity
  7 or 8 bit character: actual data transferred

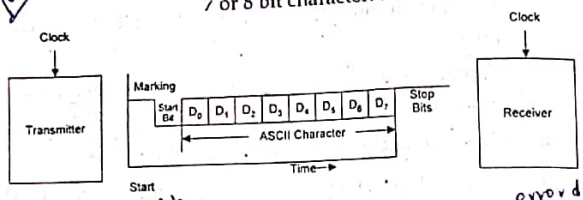


Fig: Asynchronous Serial Transmission Format

Synchronous vs. Asynchronous Data Transmission

S.N.	Parameter	Asynchronous	Synchronous
1.	Fundamental	Transmission does not based on clock signal	Transmission based on clock signal
2.	Data Format	One character at a time	Group of characters i.e. a block of characters
3.	Speed	Low (< 20 kbps)	High (> 20 kbps)
4.	Framing Information	Start and stop bits are sent with each character.	SYNC characters are sent with each character.
5.	Implementation	Hardware / Software	Hardware

Universal Synchronous Asynchronous Receiver Transmitter (USART) – 8251A

The 8251A is a programmable serial communication interface chip designed for synchronous and asynchronous serial data communication. As a peripheral device of a microcomputer system, it receives parallel data from the CPU and transmits serial data after conversion. This device also receives serial data from the outside and transmits parallel data to the CPU after conversion.

Features of 8251A

- Wide power supply voltage range from 3V to 6V
- Wide temperature range from -40°C to 85°C
- Synchronous communication up to 64 Kbaud
- Asynchronous communication up to 38.4 Kbaud
- Transmitting / Receiving operations under double buffered configuration.
- Error detection capability (Parity, Overrun and Framing)

The functional block diagram of 8251A consists five sections. They are:

- Read/Write control logic
- Transmitter
- Receiver
- Data bus buffer
- Modem control.

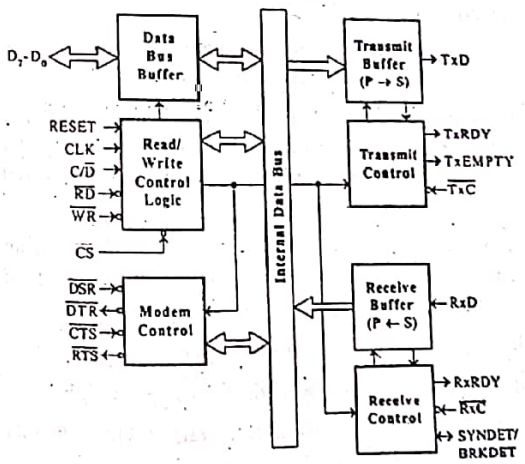


Fig: Functional block diagram of 8251A-USART

Read/Write control logic:

- The Read/Write Control logic interfaces the 8251A with CPU, determines the functions of the 8251A according to the control word written into its control register.
- It monitors the data flow.
- This section has three registers and they are control register, status register and data buffer.
- The active low signals RD, WR, CS and C/D(Low) are used for read/write operations with these three registers.
- When C/D(low) is high, the control register is selected for writing control word or reading status word.
- When C/D(low) is low, the data buffer is selected for read/write operation.
- When the reset is high, it forces 8251A into the idle mode.
- The clock input is necessary for 8251A for communication with CPU and this clock does not control either the serial transmission or the reception rate.

Transmitter section:

- The transmitter section accepts parallel data from CPU and converts them into serial data.
- The transmitter section is double buffered, i.e., it has a buffer register to hold an 8-bit parallel data and another register called output register to convert the parallel data into serial bits.
- When output register is empty, the data is transferred from buffer to output register. Now the processor can again load another data in buffer register.
- If buffer register is empty, then TxRDY is goes to high.
- If output register is empty then TxEMPTY goes to high.
- The clock signal, TxC (low) controls the rate at which the bits are transmitted by the USART.
- The clock frequency can be 1, 16 or 64 times the baud rate.

Receiver Section:

- The receiver section accepts serial data and converts them into parallel data.
- The receiver section is double buffered, i.e., it has an input register to receive serial data and convert to parallel, and a buffer register to hold the parallel data.
- When the RxD line goes low, the control logic assumes it as a START bit, waits for half a bit time and samples the line again.
- If the line is still low, then the input register accepts the following bits, forms a character and loads it into the buffer register.
- The CPU reads the parallel data from the buffer register.
- When the input register loads a parallel data to buffer register, the RxRDY line goes high.
- The clock signal RxC (low) controls the rate at which bits are received by the USART.
- During asynchronous mode, the signal SYNDET/BRKDET will indicate the break in the data transmission.
- During synchronous mode, the signal SYNDET/BRKDET will indicate the reception of synchronous character.

MODEM Control:

- The MODEM control unit allows to interface a MODEM to 8251A and to establish data communication through MODEM over telephone lines.
- This unit takes care of handshake signals for MODEM interface.

Baud rate /bit rate

The difference between Bit and Baud rate is complicated and intertwining. Both are dependent and inter-related.

Bit rate is how many data bits are transmitted per second.

A baud rate is the number of times per second a signal in a communications channel changes.

Bit rates measure the number of data bits (that is 0's and 1's) transmitted in one second in a communication channel. A figure of 2400 bits per second means 2400 zeros or ones can be transmitted in one second, hence the abbreviation "bps." Individual characters (for example letters or numbers) that are also referred to as bytes are composed of several bits.

A baud rate is the number of times a signal in a communications channel changes state or varies. For example, a 2400 baud rate means that the channel can change states up to 2400 times per second. The term "change state" means that it can change from 0 to 1 or from 1 to 0 up to X (in this case, 2400) times per second. It also refers to the actual state of the connection, such as voltage, frequency, or phase level).

The main difference between the two is that one change of state can transmit one bit, or slightly more or less than one bit, that depends on the modulation technique used. So the bit rate (bps) and baud rate (baud per second) have this connection:

If signal is changing every 10/3 ns then,

Baud rate = 1/10/3ns = $3/10*10^9$ = $3*10^8$

= 300 MBd

Note:

- If 1 frame of data is coded with 1 bit then band rate and bit rate are same.
- Sometimes frame of data are coded with two of three bits then baud rate and bit rate are not same.

RS -232

- Serial transmission of data is used as an efficient means for transmitting digital information across long distances, the existing communication lines usually the telephone lines can be used to transfer information which saves a lot of hardware.

- RS-232C is an interface developed to standardize the interface between data terminal equipment (DTE) and data communication equipment (DCE) employing serial binary data exchange. Modem and other devices used to send serial data are called data communication equipment (DCE). The computers or terminals that are sending or receiving the data are called data terminal.
- Equipment (DTE) RS-232C is the interface standard developed by electronic industries Association (EIA) in response to the need for the signal and handshake standards between the DTE and DCE.
- It uses 25 pins (DB - 25P) or 9 Pins (DE - 9P) standard where 9 pin standard does not use all signals i. e. data, control, timing and ground.
- It describes the voltage levels, impendence levels, rise and fall times, maximum bit rate and maximum capacitance for all signal lines.
- It also specifies that DTE connector should be male and DCE connector should be female.
- It can send 20kBd for a distance of 50 ft.
- The voltage level for RS-232 are:
  - A logic high or 1 , -3V to -15V
  - A logic low or 0, +3V to +15V
  - Normally ±12V voltage levels are used

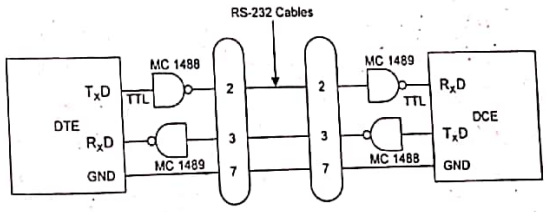


Fig: Connection of DTE and DCE through RS-232C Interface

- MC1488 converts logic 1 to -9V, logic 0 to +9V
- MC1489 converts RS-232 to TTL

- Signal levels of RS-232 are not compatible with that of the DTE and DCE which are TTL signals. For that reason, line driver such as MC1488 and line receiver MC1489 are used.
- RS-232 signals used in handshaking:

Signal Flow	DE-9P	DB-25P	Signal	Description
Reference	-	1		Protective Ground
DTE to DCE	3	2	TxD	Transmitted Data
DCE to DTE	2	3	RxD	Received Data
DTE to DCE	7	4	$\overline{RTS}$	Request To Send
DCE to DTE	8	5	$\overline{CTS}$	Clear To Send
DCE to DTE	6	6	$\overline{DSR}$	Data Set Ready
Common Ref	5	7	GND	Signal Ground
DCE to DTE	1	8	$\overline{DCD}$	Data Carrier Detect
DTE to DCE	4	20	$\overline{DTR}$	Data Terminal Ready
DCE to DTE	9	22	RI	Ring Indicator
DCE to DTE	-	23	DSRD	Data Signal Rate Detector

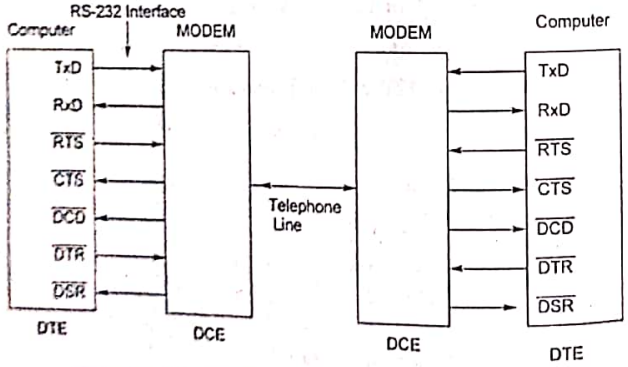


Fig. Digital Data transmission using MODEM and Telephone Line

- DTE asserts $\overline{DTR}$ to tell the modem it is ready.
- Then, DCE asserts $\overline{DSR}$ signal to the terminal and dials up.
- DTE asserts $\overline{RTS}$ signal to the modem.
- Modem then asserts $\overline{DCD}$ signal to indicate that it has established connection with the computer.
- DCE asserts $\overline{CTS}$ signals, then DTE sends serial data.
- When sending is completed, DTE asserts $\overline{RTS}$ high, this causes modem to unassert its $\overline{CTS}$ signal and stop transmitting similar handshake taken between DCE and DTE other side.
- To communicate from serial port of a computer to serial port of another computer without modem, null-modem is used.

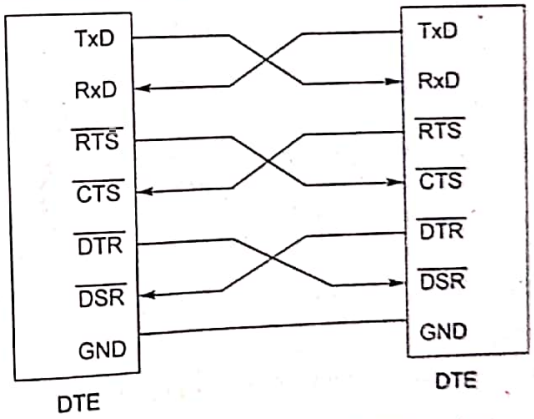


Fig: Null MODEM Connection for RS-232 Terminals

RS-423A and RS-422A Serial Standards

RS-423A

- A major problem with RS-232C is that it can only transmit data reliably for about 50 ft. at its maximum rate of 20Kbd. If longer lines are used, the transmission rate has to be drastically reduced due to open signal lines with a common signal ground.
- Another EIA standard which is improvement over RS-232C is RS-423A.
- This standard specifies a low impendence single ended signal which can be sent over 50Ω coaxial cable and partially terminated at the receiving end to prevent reflection.

Voltage levels

- High 4 -6V negative
- Low 4-6V positive
- Transmission rate 100 Kbd over 40 ft
  1 Kbd over 4000 ft

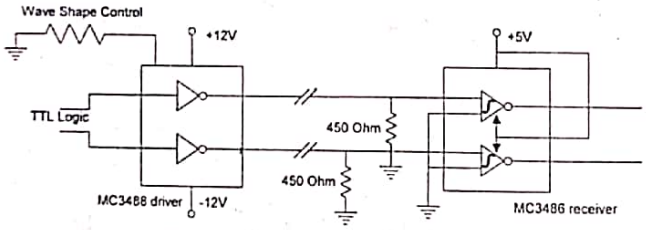


Fig: MC3488 driver and MC3489 receiver used for RS-423A Interface

RS 422A

- A newer standard for serial data transfer.
- It specifies that can signal will be send differentially over two adjaecent wires in a ribbon cable or a twisted pair of wires uses differential amplifier to reject noise.

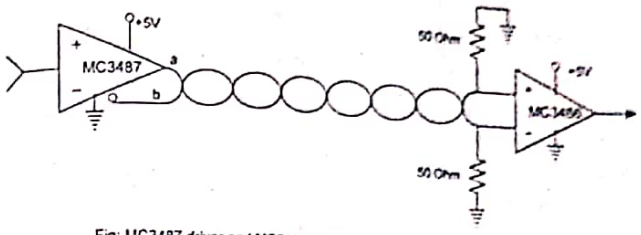


Fig: MC3487 driver and MC3486 receiver used for RS-422A Interface

- Voltage level:
  - Logic high making 'a' line more positive than 'b' line.
  - Logic low making 'b' line more positive than 'a' line.
- The voltage difference between the two lines must be greater than 0.4V but less than 12V.
- The mc3487 driver provides a differential voltage of about 2V.
- The center or common mode voltage on the lines must be between -7V and +7V
- Transmission rate:
  - 10000 KBd for 40 ft.
  - 100 KBD for 4000 ft.
- The high data transfer is because of differential lline functions as a fully terminated transmission line.
- Mc 3486 receiver only responds to the differential voltage eliminating noise.

Introduction to Direct Memory Access (DMA) & DMA Controllers

During any given bus cycle, one of the system components connected to the system bus is given control of the bus. This component is said to be the master during that cycle and the component it is communicating with is said to be the slave. The CPU with its bus control logic is normally the master, but other specially designed components can gain control of the bus by sending a bus request to the CPU. After the current bus cycle is

completed the CPU will return a bus grant signal and the component sending the request will become the master.

Taking control of the bus for a bus cycle is called cycle stealing. Just like the bus control logic, a master must be capable of placing addresses on the address bus and directing the bus activity during a bus cycle. The components capable of becoming masters are processors (and their bus control logic) and DMA controllers. Sometimes a DMA controller is associated with a single interface, but they are often designed to accommodate more than one interface.

This is a process where data is transferred between two peripherals directly without the involvement of the microprocessor. This process employs the HOLD pin on the microprocessor. The external DMA controller sends a signal on the HOLD pin to the microprocessor. The microprocessor completes the current operation and sends a signal on HLDA and stops using the buses. Once the DMA controller is done, it turns off the HOLD signal and the microprocessor takes back control of the buses.

Basic DMA operation

- The direct memory access (DMA) technique provides direct access to the memory while the microprocessor is temporarily disabled.
- A DMA controller temporarily borrows the address bus, data bus, and control bus from the microprocessor and transfers the data bytes directly between an I/O port and a series of memory locations.
- The DMA transfer is also used to do high-speed memory-to memory transfers.
- Two control signals are used to request and acknowledge a DMA transfer in the microprocessor-based system.
- The HOLD signal is a bus request signal which asks the microprocessor to release control of the buses after the current bus cycle.
- The HLDA signal is a bus grant signal which indicates that the microprocessor has indeed released control of its buses by placing the buses at their high-impedance states.

- The HOLD input has a higher priority than the INTR or NMI interrupt inputs.

DMA Data Transfer scheme

- Data transfer from I/O device to memory or vice-versa is controlled by a DMA controller.
- This scheme is employed when large amount of data is to be transferred.
- The DMA requests the control of buses through the HOLD signal and the MPU acknowledges the request through HLDA signal and releases the control of buses to DMA.
- It's a faster scheme and hence used for high speed printers.

Block (Burst) mode of data transfer

In this scheme the I/O device withdraws the DMA request only after all the data bytes have been transferred.

Cycle stealing technique

In this scheme the bytes are divided into several parts and after transferring every part the control of buses is given back to MPU and later stolen back when MPU does not need it.

Programmable DMA Controller - Intel 8257

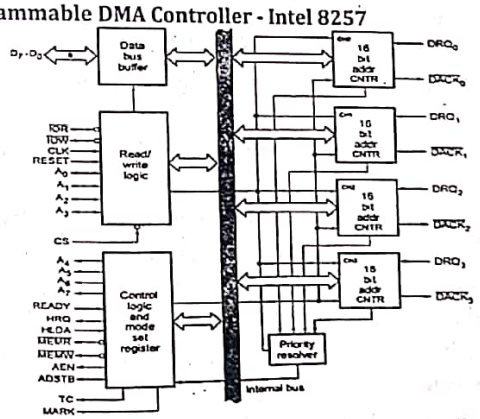


Fig: Functional block diagram of DMA Controller

It is a device to transfer the data directly between IO device and memory without through the CPU. So it performs a high-speed data transfer between memory and I/O device.

The features of 8257 is,

- The 8257 has four channels and so it can be used to provide DMA to four I/O devices.
- Each channel can be independently programmable to transfer up to 64kb of data by DMA.
- Each channel can be independently perform read transfer, write transfer and verify transfer.

The functional blocks of 8257 as shown in the above figure are data bus buffer, read/write logic, control logic, priority resolver and four numbers of DMA channels.

Operation of 8257 DMA Controller

- Each channel of 8257 has two programmable 16-bit registers named as address register and count register.
- Address register is used to store the starting address of memory location for DMA data transfer.
- The address in the address register is automatically incremented after every read/write/verify transfer.
- The count register is used to count the number of byte or word transferred by DMA.
- In read transfer the data is transferred from memory to I/O device.
- In write transfer the data is transferred from I/O device to memory.
- Verification operations generate the DMA addresses without generating the DMA memory and I/O control signals.
- The 8257 has two eight bit registers called mode set register and status register.

ADDITIONAL QUESTIONS

1. **Design an interfacing circuit for following problem.**
   - i. **74LS138: 3 to 8 Decoder**
   - ii. **2732 (4K × 8): EPROM, address range should begin at 0000H and additional 4K memory space should be available for future expansion.**
   - iii. **6116(2K × 8): CMOS R/W memory.**

[2073 Bhadra]

To design a memory address decoding circuit, first calculate the number of address pins used by the memory device depending upon the memory capacity they can handle. Then determine the mapping addresses for memory devices (RAM, ROM). Then determine the different address pins which need to provide the input for n × 2^n decoder. Finally select the memory chips by using the appropriate decoder output line and required control signals (IO/$\overline{M}$, $\overline{RD}$, $\overline{WR}$).

Let us draw a circuit diagram of a memory device interfacing where one 4K×8 EPROM chip, additional 4K x8 for future expansion and one 2K×8 CMOS R/W chip at address 0000H.

Step 1: Calculate the number of address pins

4K×8 chip requires 12 no. of address pins and 2K×8 chip requires 11 no. of address pins. This can be calculated from

n = log (memory capacity in bytes) / log (2)

Step 2: Memory mappingtable

Memory Block	Address	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
EPROM	Start:0000H	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	End:0FFFH	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1
RAM	Start:2000H	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	End: 27FFH	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1

Here, after placing EPROM, we need to reserve 4K ×8 memory for future expansion; hence starting address of RAM comes to 2000H instead of 1000H.

Step 3: Decide decoder pins

Here, bits A_{11} and A_{13} address lines are different for 2 chips referring to end address lines. We need to use a 3×8 decoder, so we need to add one address line and here we can use A_{12}. We can use address lines A_{14} and A_{15} to generate chip enable signals for 3×8 decoder.

Step 4: Draw a decoding circuit

The output from decoder Y_1 (001) selects the EPROM and Y_4 (100) selects RAM. The address decoding circuit is drawn below.

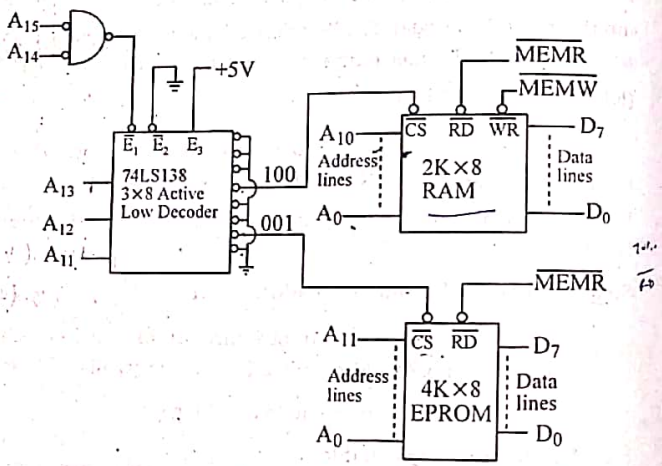


Fig.: Address decoding circuit

2. **Design an interfacing circuit to interface one 4KB EPROM and two 2KB R/W memory for 8085 microprocessor.**

[2072 Magh]

To design a memory address decoding circuit, first calculate the number of address pins used by the memory device

depending upon the memory capacity they can handle. Then determine the mapping addresses for memory devices (RAM, ROM). Then, determine the different address pins which need to provide the input for n× 2^n decoder. Finally, select the memory chips by using the appropriate decoder output line and required control signals (IO/$\overline{M}$, $\overline{RD}$, $\overline{WR}$).

Drawing a circuit diagram of a memory device interfacing with one 4KB EPROM chip, and two 2KB RAM chips at address 8000H involves the following steps.

Step 1: Calculate the number of address pins

4K×8 chip requires 12 no. of address pins and 2K×8 chip requires 11 no. of address pins. This can be calculated from

n = log (memory capacity in bytes) / log (2)

Step 2: Memory mapping table

Memory Block	Address	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
EPROM	Start:8000H	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	End:8FFFH	1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1
RAM1	Start:9000H	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
	End: 97FFH	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1
RAM2	Start:9800H	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
	End: 9FFFH	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1

Step 3: Decide decoder pins

Here, bits A_{11}, A_{12} and A_{13} address lines are different for 3 chips referring to end address lines. So, we need to use a 3×8 decoder. We can use address lines A_{14} and A_{15} to generate chip enable signals for 3×8 decoder.

Step 4: Draw a decoding circuit

The output from decoder Y_1 (001) selects the EPROM, Y_2 (010) selects RAM1 and Y_3 (011) selects RAM2 chips. The address decoding circuit is drawn below.

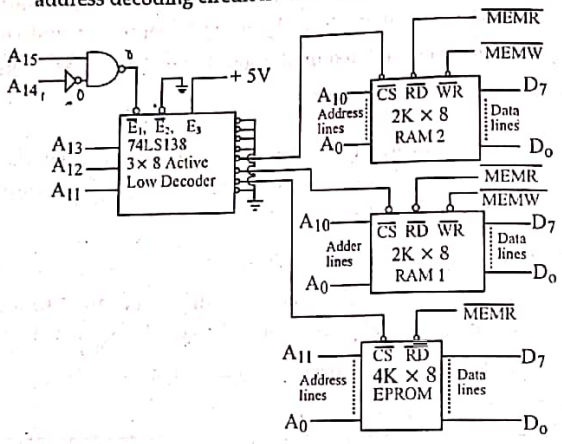


Fig.: Address decoding circuit

3. With a neat diagram, explain the interfacing circuit using a 3:8 decoder (74LS138) needed to connect the following memory units to the 8085 microprocessor consecutively starting from memory location A000H.

   i. 2K×8 ROM chip
   ii. 2K×8 RAM chip
   iii. 2K×8 EPROM chip *[2073 Magh]*

To design a memory address decoding circuit, first calculate the number of address pins used by the memory device depending upon the memory capacity they can handle. Then determine the mapping addresses for memory devices (RAM, ROM). Then, determine the different address pins which need to provide the input for $n \times 2^n$ decoder. Finally, select the memory chips by using the appropriate decoder output line and required control signals (IO/$\overline{M}$, $\overline{RD}$, $\overline{WR}$).

Drawing a circuit diagram of a memory device interfacing with one 2K×8 ROM chip, 2K×8 RAM chip and one 4K ×8 EPROM chip at address A000H involves the following steps.

Step 1: Calculate the number of address pins

2K×8 chip requires 11 no. of address pins and 4K×8 chip requires 12 no. of address pins. This can be calculated from

n = log (memory capacity in bytes) / log (2)

Step 2: Memory mapping table

Memory Block	Address	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
ROM	Start:A000H	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	End:A7FFH	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1
RAM	Start:A800H	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0
	End: AFFFH	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1
EPROM	Start: B000H	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	End: BFFFH	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Step 3: Decide decoder pins

Here, bits A_{11} and A_{12} address lines are different for 3 chips referring to start or end address lines. We need to use a 3×8 decoder, so we need to add one address line and here we can use A_{13}. We can use address lines A_{14} and A_{15} to generate chip enable signals for 3×8 decoder.

Step 4: Draw a decoding circuit

The output from decoder Y_4 (100) selects the ROM, Y_5 (101) selects RAM and Y_6 (110) selects the EPROM. The address decoding circuit is drawn below.

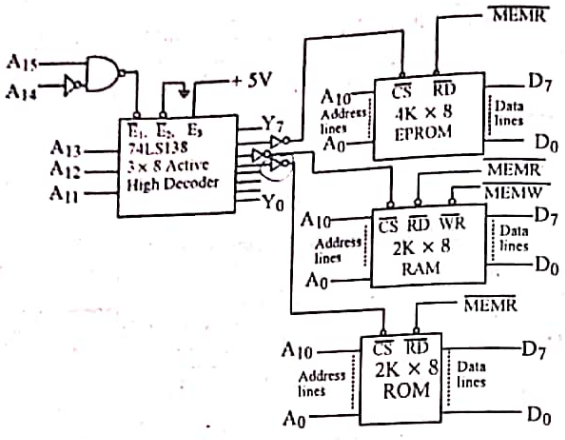


Fig.: Address decoding circuit

4. **Design an address decoding circuit to interface an input device with eight input switches and a LED output device at 41H and 42H respectively.**

[2064 Shrawan]

Address Decoding Circuit:

I/O Mapping	$A_7A_6A_5A_4$	$A_3A_2A_1A_0$
Input Device (41H):	0 1 0 0	0 0 0 1
Output Device (42H):	0 1 0 0	0 0 1 0

- Bit A_2 to A_7 are used to enable the external decoder.
- Bits A_0 and A_1 are used for the device selection (as an input to the external decoder).
- Among the above two address, A_0 and A_1 bits decides, whether the given device is input or output (i.e. for selection of I/O device).

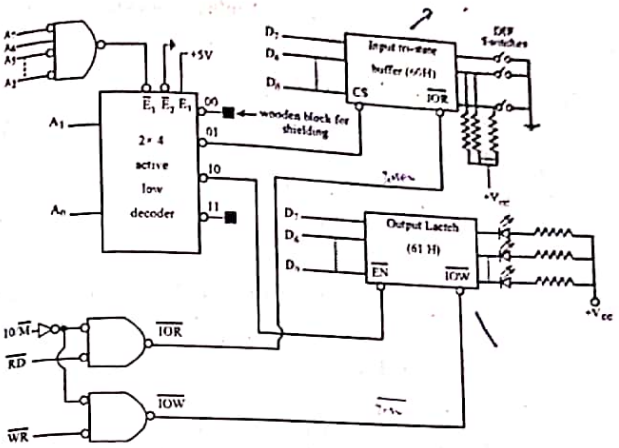


Fig.: Address decoding circuit

5. **Design an address decoding circuit to interface two RAM blocks and a ROM block each of 4 KB starting at address 4000H.**

[2064 Poush]

Address Decoding Circuit:

RAM (4KB):

Base Address = 4000H

End Address = base Address + (Number of locations in 4 KB RAM – 1)

= 4000H + (1024X4-1)D

= 4000H + OFFFH

= 4FFFH

RAM 2(4KB)

Base Address = End Address of RAM 1 + 1

= 4FFFH + 0001H

= 5000H

End Address = Base Address +(Number of locations in 4KB RAM -1)

= 5000H+0FFFH

= 5FFFH

ROM (4KB)

Base Address = End Address of ROM +1

= 5FFFH + 0001H

= 6000H

End Address = Base Address+ (Number of location in 4 KB RAM-1)

6000H + OFFFH =6FFFH

	$A_{15}A_{14}A_{13}A_{12}$	$A_{11}A_{10}A_9A_8$	$A_7A_6A_5A_4$	$A_3A_2A_1A_0$
RAM 1(4KB)				
Base Address (4000):	0 1 0 0	0 0 0 0	0 0 0 0	0 0 0 0
End Address (4FFFH):	0 1 0 0	1 1 1 1	1 1 1 1	1 1 1 1
RAM 2 (4KB)				
Base Address (5000H):	0 1 0 1	0 0 0 0	0 0 0 0	0 0 0 0
End Address (5FFFH):	0 1 0 1	1 1 1 1	1 1 1 1	1 1 1 1
ROM (4KB)				
Base Address (6000H):	0 1 1 0	0 0 0 0	0 0 0 0	0 0 0 0
End Address (6FFFH):	0 1 1 0	1 1 1 1	1 1 1 1	1 1 1 1

Hence, bit A_{12} and A_{13} are used for RAMs and ROM selection (as an i/p to external decoder), bit A_{14} and A_{15} are used to enable external device and bit A_0 to A_{11} are given to ID of RAMs and ROM.

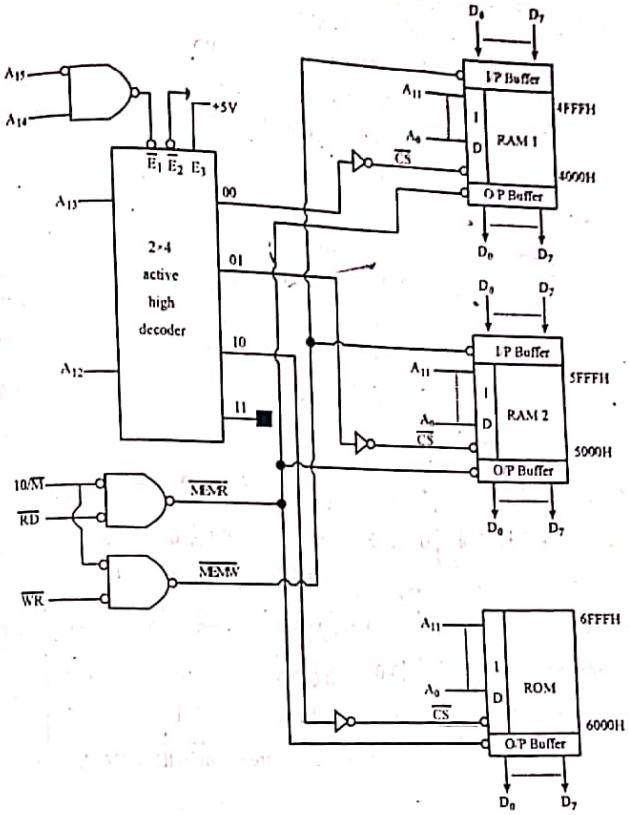


Fig.: Address decoding circuit

6. **Present a complete plan to use 2 RAM chips of 16 KB each with 8085 microprocessor.**

[2065 Chaitra]

Let the starting address be 4000H

RAM 1 (16 KB)

Base Address = 0000H

End Address = Base Address + (number of locations in 16KB RAM-1)

= 0000H + (16 X 2^{10} -1)D

= 0000H + (16384-1)D

= 0000H + 16383D

= 3FFFH

RAM 2 (16 KB)

Base Address = End Address of RAM 1 +1

= 3FFFH + 0001H

= 4000H

End Address = 4000H + (16 X 2^{10} -1)D

= 7FFFH

	$A_{15}A_{14}A_{13}A_{12}$	$A_{11}A_{10}A_9A_8$	$A_7A_6A_5A_4$	$A_3A_2A_1A_0$
RAM 1 (16 KB)				
Base Address:(OOOOH)	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0
End Address: (3FFFH)	0 0 1 1	1 1 1 1	1 1 1 1	1 1 1 1
RAM 2 (16 KB)	bit A_0 to A_{13} are given to ID of RAM1			
Base Address: (4000H)	0 1 0 0	0 0 0 0	0 0 0 0	0 0 0 0
End Address :(7FFFH)	0 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1

bit A_0 to A_{13} are given to ID of RAM2

Hence, bit A_{14} is used for RAM selection (as an input to external decoder) and bit A_{15} is used to enable external device and bits A_0 to A_{13} to Internal Decoder(ID) of RAM 1 and bits A_0 to A_{13} to ID of RAM 2.

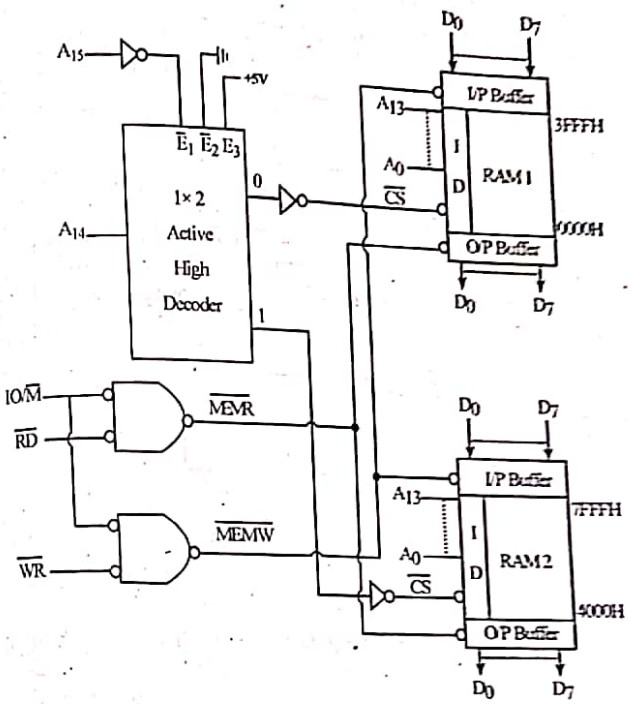


Fig.: Address decoding circuit

7. Design a unique address decoding circuit using memory mapped I/O interface to read input from port address FFF9H and output port address FFF8H.

[2067 Mangsir]

Among the above two addresses, A_0 bit decides whether the given device is I/P or O/P (i.e., for selection of input/output device)

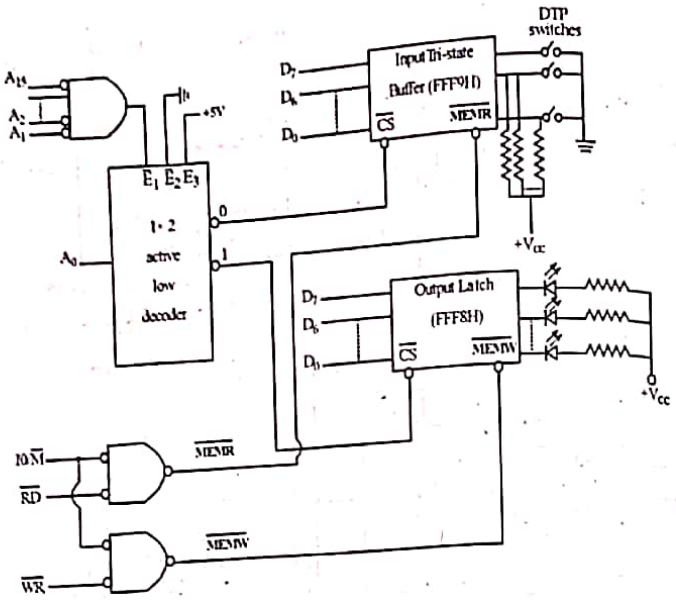


Fig.: Address decoding circuit

8. Draw the bus timing diagram when the instruction ADI 34H is executed. *[2067 Shrawan]*

2060H : opcode of ADI

2061H : 34H

+Instruction cycle of ADI 34H	*Fetch cycle*	T_1 :	MAR ← PCs
		T_2 :	MBR ← [MAR]
		T_3 :	IR ← MBR, PC←PC+1
		T_4 :	Unspecified
	Memory Read cycle	T_5 :	MAR ← PC
		T_6 :	MBR ← [MAR]
		T_7 :	Z ← MBR, PC ← PC + 1

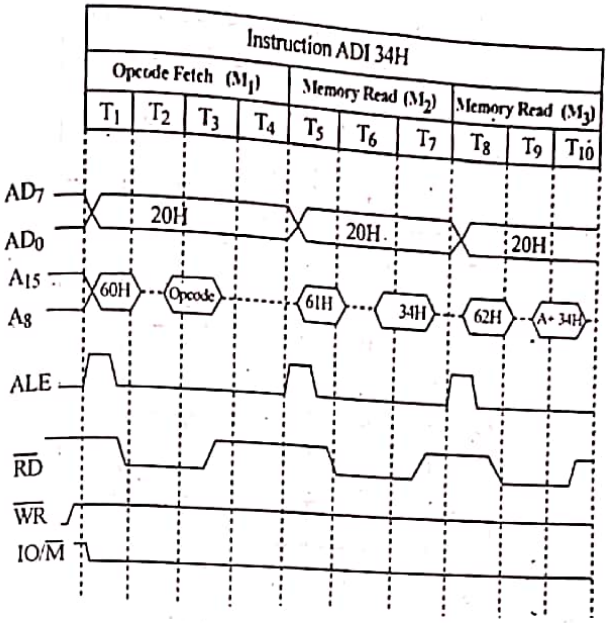


Fig.: Timing diagram

9. In 8085, memory and I/O read/write instructions use extra machine cycle for memory and I/O read write operation. Use the time diagram for MOV R, M and OUT 01H instructions to illustrate the statement. *[2067 Mangsir]*

2050H : opcode of MOV R, M

Instruction cycle of MOV R, M	*Fetch cycle*	T_0 :	MAR ← PC
		T_1 :	MBR ← [MAR]
		T_2 :	IR ← MBR, PC←PC + 1,
		T_3 :	Unspecified
	Memory Read cycle	T_4 :	MAR ← HL
		T_5 :	MBR ← [MAR]
		T_6 :	R ← MBR, PC ← PC + 1, SC ← O

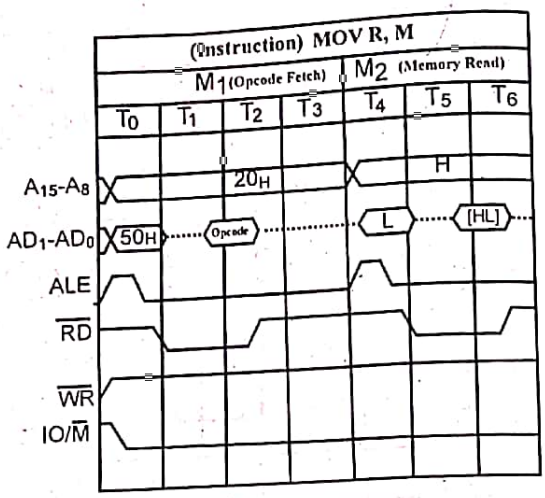


Fig.: Timing diagram

2050H : opcode of OUT (D3)
2051H : 01H

Instruction cycle of OUT 01H	*Fetch cycle*	T_1 :	MAR ← SP
		T_2 :	MBR ← [MAR]
		T_3 :	IR ← MBR, PC←PC+1
		T_4 :	Unspecified
	Memory Read cycle	T_1 :	MAR ← PC
		T_2 :	MBR ← [MAR]
		T_3 :	Z ← MBR, PC ←PC + 1
	I/O Write cycle	T_1 :	IOAR ← Z
		T_2 :	IOBR ← A
		T_3 :	[IOAR] ← IOBR, SC ← O

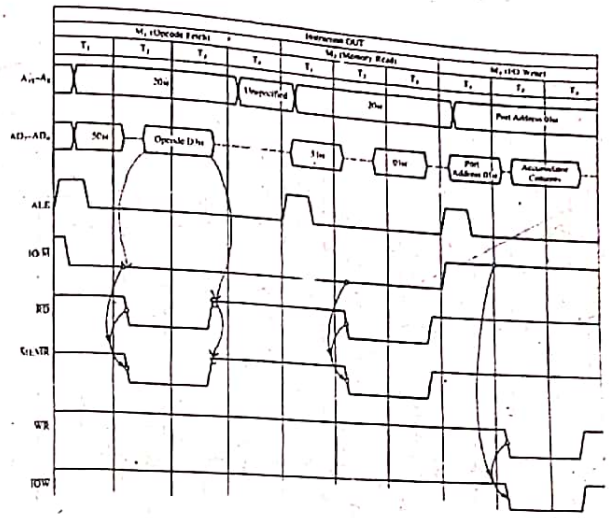


Fig.: Timing diagram

10. Draw the bus timing diagram for IN 86H when the instruction is at location 8256H. *[2068 Jestha]*

8256 : opcode of IN
8257 : 86H

Instruction cycle of IN instruction	*Fetch cycle*	T_1 :	MAR ← SP
		T_2 :	MBR ← [MAR]
		T_3 :	IR ← MBR, PC←PC + 1
		T_4 :	Unspecified
	Memory Read cycle	T_1 :	MAR ← PC
		T_2 :	MBR ← [MAR]
		T_3 :	Z ← MBR, PC ←PC + 1
	I/O Write cycle	T_1 :	IOAR ← Z
		T_2 :	IOBR ← A
		T_3 :	A ← IOAR, SC ← O

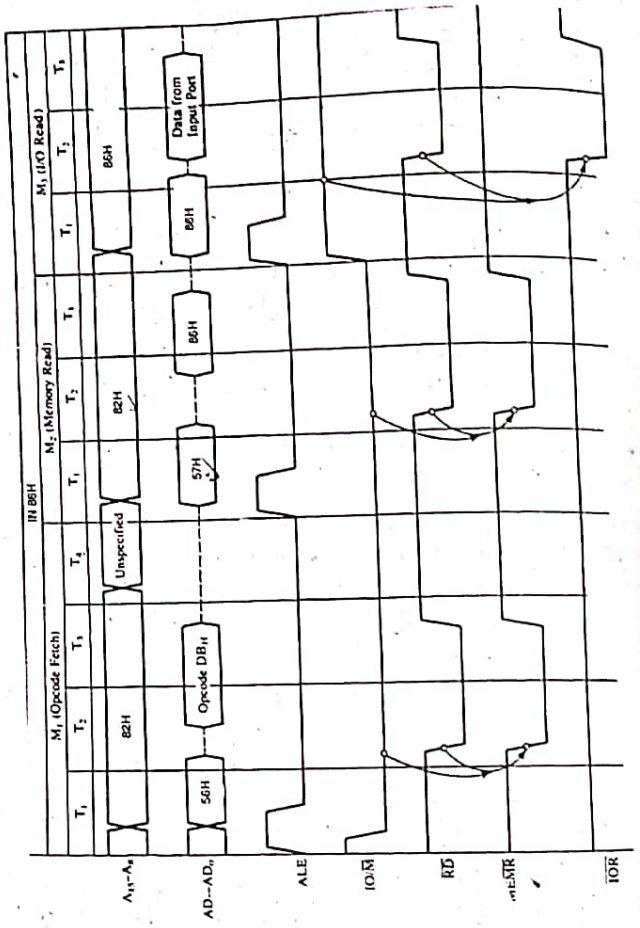


Fig.: Timing diagram

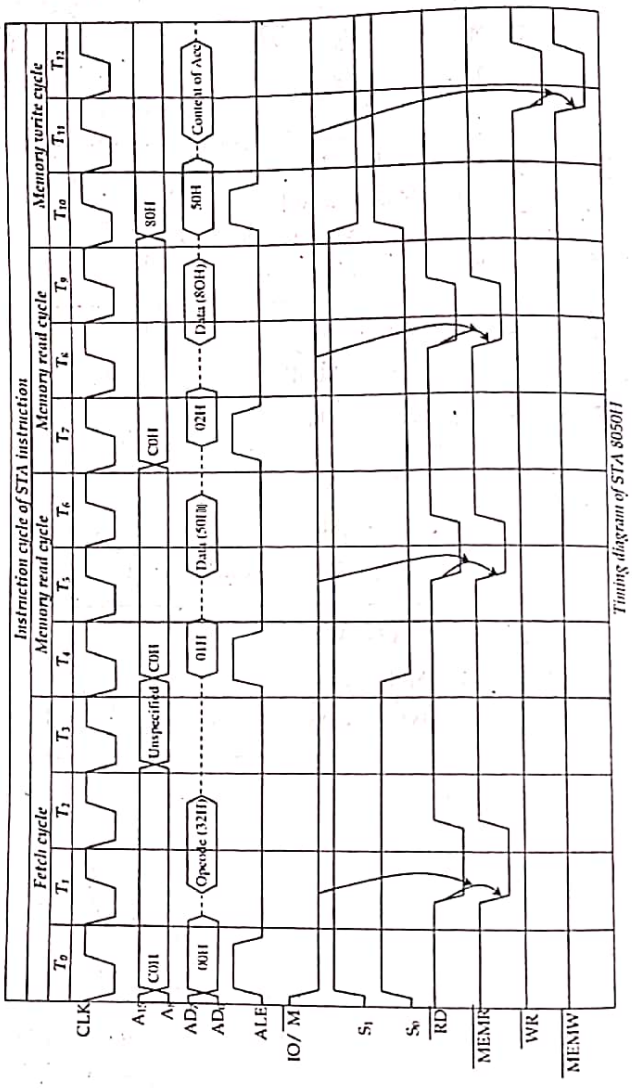
11. Draw the timing diagram of the 8085 instruction STA 8050H. *[2073 Bhadra]*

C000H: Opcode

C001H: 50H

C002H: 80H

Instruction cycle of STA 8050H instruction			
	Fetch cycle	T_0:	MAR ← PC
		T_1:	MBR ← [MAR]
		T_2:	IR ← MBR, PC←PC + 1
		T_3:	Unspecified
	Memory Read cycle	T_4:	MAR ← PC
		T_5:	MBR ← [MAR]
		T_6:	Z ← MBR, PC ←PC + 1
	Memory Read cycle	T_7:	MAR ← PC
		T_8:	MBR ← [MAR]
		T_9:	W ← MBR, PC ←PC + 1
	Memory Write cycle	T_{10}:	MAR ← WZ
		T_{11}:	MBR ← A
		T_{12}:	[MAR] ← MBR, SC ← 0



Timing diagram of STA 8050H

12. **Calculate the time taken to execute the following program if T = 1 micro second.**

MVI A, 05H
ADI 20H
OUT 80H
HLT

[2072 Magh]

Instructions	T-States	Clocks
MVI A, 05H;	7	7×1
ADI, 20H;	7	7×1
OUT 80H;	10	10×1
HLT;	5	5×1
	Total Clocks:	29

If T = 1 micro second, time taken to execute the program = 29 micro seconds.

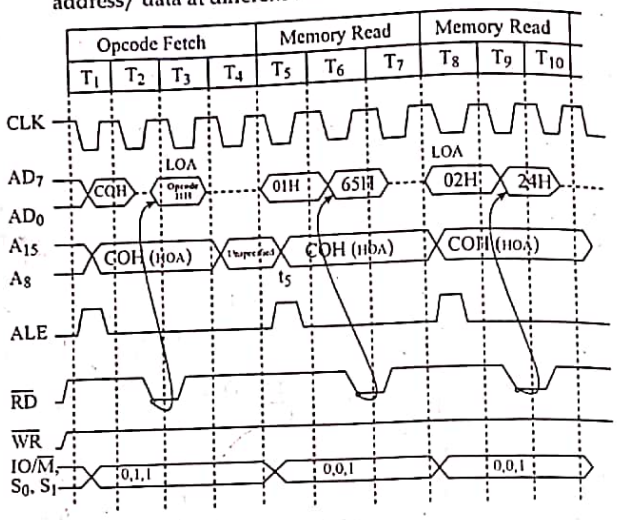
13. **Draw the timing diagram of LXI D, 2465H. Calculate the time required to execute this instruction if the crystal frequency is 6 MHz.** *[2073 Magh]*

Assume that this instruction LXI D, 2465H is located in memory address C000H.

C000H : opcode (LXI D)
C001H : 65H
C002H : 24H

Instruction cycle of LXI D, 2465H			
	Fetch cycle	T_1 :	MAR ← PC
		T_2 :	MBR ← [MAR]
		T_3 :	IR ← MBR, PC←PC + 1,
		T_4 :	Unspecified
	Memory Read cycle	T_5 :	MAR ← PC
		T_6 :	MBR ← [MAR]
		T_7 :	D ← MBR, PC ←PC + 1, SC ← 0
	Memory Read cycle	T_8 :	MAR ← PC
		T_9 :	MBR ← [MAR]
		T_{10} :	D ← MBR, PC ←PC + 1, SC ← 0

The timing diagram of LXI D, 2465 H showing the flow of address/ data at different T-states is shown below.



LOA = Low Order Address
HOA = HIgh Order Address

Fig.: Timing diagram

As per the RTL and timing diagram, LXI D uses 10 clock cycles, and crystal frequency is 6 MHz.

Time required to execute 1 clock cycle $= 1/6\times10^{-6}$

Hence, time required to execute 10 clock cycles

$= 10/6\times10^{-6} = 1.66\ \mu s$

14. An instruction is stored at memory location as follows:

Memory Location	Hex Code
2050	3A (op-code)
2051	80
2052	20

This instruction loads the content of memory location 2080 into accumulator. Draw timing diagram of this instruction.

The instruction is LDA 2080H.

Instruction cycle of LDA 2080H	*Fetch cycle*	T_0:	MAR ← PC
		T_1:	MBR ← [MAR]
		T_2:	IR ← MBR, PC ← PC + 1
		T_3:	Unspecified
	Memory read cycle	T_4:	MAR ← PC
		T_5:	MBR ← [MAR]
		T_6:	Z ← MBR, PC ← PC + 1
	Memory write cycle	T_7:	MAR ← PC
		T_8:	MBR ← [MAR]
		T_9:	W ← MBR, PC ← PC + 1
	Memory write cycle	T_{10}:	MAR ← WZ
		T_{11}:	MBR ← [MAR]
		T_{12}:	A ← MBR, SC ← 0

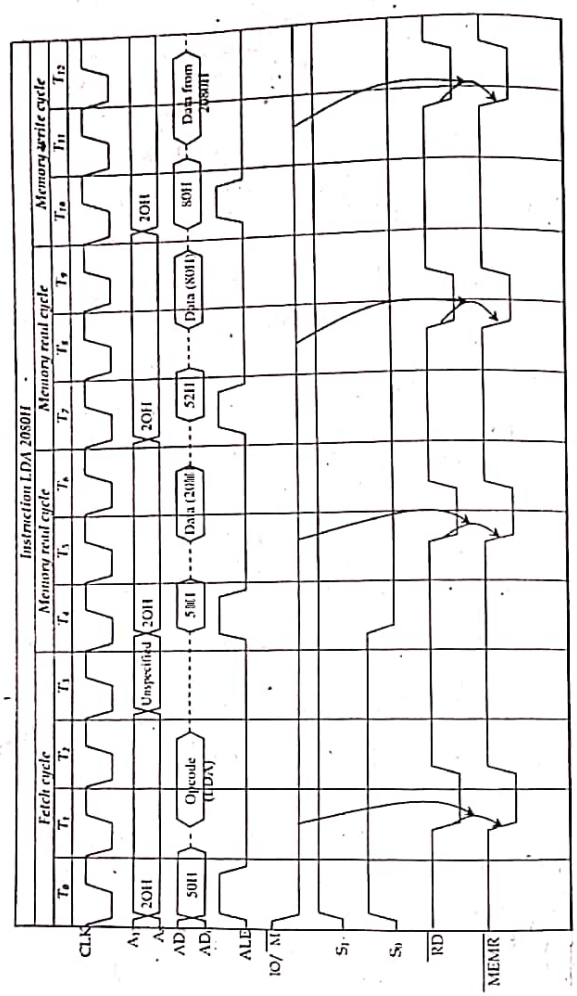


Fig.: Timing diagram

❖ ❖ ❖

Chapter - 5

INTERRUPT OPERATIONS

Introduction

Interrupt is signal send by an external device to the processor, to request the processor to perform a particular task or work. Mainly in the microprocessor based system, the interrupts are used for data transfer between the peripheral and the microprocessor. The processor will check the interrupts always at the second T-state of last machine cycle. If there is any interrupt, it accepts the interrupt and send the INTA (active low) signal to the peripheral. The vectored address of particular interrupt is stored in program counter. The processor executes an interrupt service routine (ISR) addressed in program counter. It returns to main program by RET instruction.

Interrupts are particularly useful when interfacing I/O devices that provide or require data at relatively low data transfer rate.

Interrupt operations

The transfer of data between the microprocessor and input/ output devices takes place using various modes of operations like programmed I/O, interrupt I/O and direct memory access. In programmed I/O, the processor has to wait for a long time until I/O module is ready for operation. So the performance of entire system degraded. To remove this problem CPU can issue an I/O command to the I/O module and then go to do some useful works. The I/O device will then interrupt the CPU to request service when it is ready to exchange data with CPU. In response to an interrupt, the microprocessor stops executing its current program and calls a procedure which services the interrupt.

The interrupt is a process of data transfer whereby an external device or a peripheral can inform the processor that it is

ready for communication and it requests attention. The response to an interrupt request is directed or controlled by the microprocessor.

Process of interrupt operation:

From the point of view of I/O unit

- I/O device receives command from CPU
- The I/O device then processes the operation
- The I/O device signals an interrupt to the CPU over a control line.
- The I/O device waits until the request from CPU.

From the point of view of processor

- The CPU issues command and then goes off to do its work.
- When the interrupt from I/O device occurs, the processor saves its program counter & registers of the current program and processes the interrupt.
- After completion for interrupt, processor requires its initial task.

Polling versus interrupt

- Each time the device is given a command, for example ``*move the read head to sector 42 of the floppy disk*'' the device driver has a choice as to how it finds out that the command has completed. The device drivers can either poll the device or they can use interrupts.
- Polling the device usually means reading its status register every so often until the device's status changes to indicate that it has completed the request.
- Polling means the CPU keeps checking a flag to indicate if something happens.
- An interrupt driven device driver is one where the hardware device being controlled will cause a hardware interrupt to occur whenever it needs to be serviced.

- With interrupt, CPU is free to do other things, and when something happens, an interrupt is generated to notify the CPU. So it means the CPU does not need to check the flag.
- Polling is like picking up your phone every few seconds to see if you have a call. Interrupts are like waiting for the phone to ring.
- Interrupts win if processor has other work to do and event response time is not critical.
- Polling can be better if processor has to respond to an event ASAP; may be used in device controller that contains dedicated secondary processor.

Advantages of interrupt over polling

- Interrupts are used when you need the fastest response to an event. For example, you need to generate a series of pulses using a timer. The timer generates an interrupt when it overflows and within 1 or 2 sec, the interrupt service routine is called to generate the pulse. If polling were used, the delay would depend on how often the polling is done and could delay response to several msecs. This is thousands times slower.
- Interrupts are used to save power consumption. In many battery powered applications, the microcontroller is put to sleep by stopping all the clocks and reducing power consumption to a few micro amps. Interrupts will awaken the controller from sleep to consume power only when needed. Applications of this are hand held devices such as TV/VCR remote controllers.
- Interrupts can be a far more efficient way to code. Interrupts are used for program debugging.

Interrupt Structures

A processor is usually provided with one or more interrupt pins on the chip. Therefore a special mechanism is necessary to handle interrupts from several devices that share one of these interrupt lines. There are mainly two ways of servicing multiple

interrupts which are polled interrupts and daisy chain (vectored) interrupts.

1. **Polled interrupts**

Polled interrupts are handled by using software which is slower than hardware interrupts. Here the processor has the general (common) interrupt service routine (ISR) for all devices. The priority of the devices is determined by the order in which the routine polls each device. The processor checks the starting with the highest priority device. Once it determines the source of the interrupt, it branches to the service routine for that device.

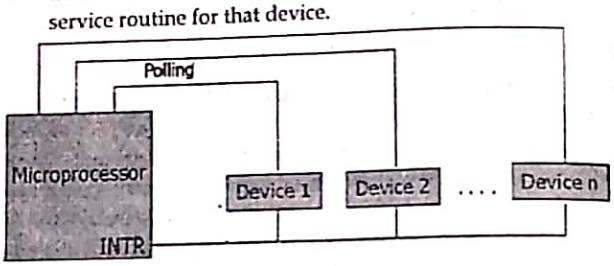


Fig.: Polled interrupt

Here several eternal devices are connected to a single interrupt line (INTR) of the microprocessor. When INTR signal goes up, the processor saves the contents of PC and other registers and then branches to an address defined by the manufactures of the processor. The user can write a program at this address to find the source of the interrupt by starting the polled from highest priority device.

2. **Daisy chain (vectored) interrupt**

In polled interrupt, the time required to poll each device may exceed the time to service the device through software. To improve this, the faster mechanism called vectored or daisy chain interrupt is used. Here the devices are connected in chain fashion. When INTR pin goes up, the processor saves its current status and then generates $\overline{\text{INTA}}$ signal to the highest priority device. If this device has generated the

interrupt, it will accept the INTA; otherwise it will push $\overline{\text{INTA}}$ to the next priority device until the $\overline{\text{INTA}}$ is accepted by the interrupting device.

When $\overline{\text{INTA}}$ is accepted, the device provides a means to the processor for findings the interrupt address vector using external hardware. The accepted device responds by placing a word on the data lines which becomes the vector address with the help of any hardware through which the processor points to appropriate device service routine. Here no general interrupt service routine need first that means appropriate ISR of the device will be called.

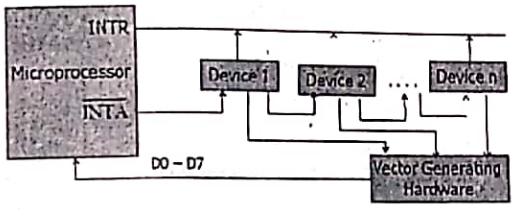


Fig: Vectored (daisy chain) interrupt

Interrupt Processing Sequence

The occurrence of interrupt triggers a number of events, both in processor hardware and in software. The interrupt driven I/O operation takes the following steps.

- The I/O unit issues an interrupt signal to the processor for exchange of data between them.
- The processor finishes execution of the current instruction before responding to the interrupt.
- The processor sends an acknowledgement signal to the device that it issued the interrupt.
- The processor transfers its control to the requested routine called "Interrupt Service Routine (ISR)" by saving the contents of program status word (PSW) and program counter (PC).

- The processor now loads the PC with the location of interrupt service routine and the fetches the instructions. The result is transferred to the interrupt handler program.
- When interrupt processing is completed, the saved register's value are retrieved from the stack and restored to the register.
- Finally it restores the PSW and PC values from the stack.

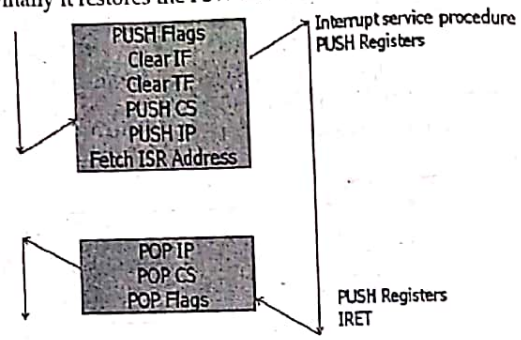


Fig.: Interrupt response for 8086 microprocessor

The figure summarizes these steps. The processor pushes the flag register on the stack, disables the INTR input and does essentially an indirect call to the interrupt service procedure. An IRET function at the end of interrupt service procedure returns execution to the main program.

Interrupt priority:

Microcomputers can transfer data to or from an external devices using interrupt through INTR pin. When device wants to communicate with the microcomputer, it connects to INTR pin and makes it high or low depending on microcomputer. The microcomputer responds by sending signal via its pin called interrupt acknowledgement INTA. In differentiation with the occurrence of interrupts, basically following interrupts exist.

1. **External interrupts:**

   These interrupts are initiated by external devices such as A/D converters and classified on following types.

- Maskable interrupt:

  It can be enabled or disabled by executing instructions such as EI and DI. In 8085, EI sets the interrupt enable flip flop and enables the interrupt process. DI resets the interrupt enable flip flop and disables the interrupt.

- Non-maskable interrupt:

  It has higher priority over maskable interrupt and cannot be enabled or disabled by the instructions.

2. **Internal interrupts:**

- These are indicated internally by exceptional conditions such as overflow, divide by zero, and execution of illegal op-code. The user usually writes a service routine to take correction measures and to provide an indication in order to inform the user that exceptional condition has occurred.
- There can also be activated by execution of TRAP instruction. This interrupt means TRAP is useful for operating the microprocessor in single step mode and hence important in debugging.
- These interrupts are used by using software to call the function of an operating system. Software interrupts are shorter than subroutine calls and they do not need the calling program to know the operating system's address in memory.

If the processor gets multiple interrupts, then we need to deal these interrupts one at a time and the dealing approaches are:

a. **Sequential processing of interrupts**

When user program is executing and an interrupt occurs interrupts are disabled immediately. After the interrupt service routine completes, interrupts are enabled before resuming the user program and the processor checks to see if additional interrupts have occurred.

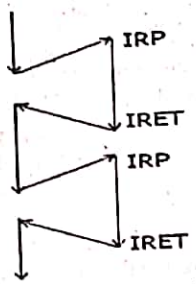


Fig: Sequential Interrupt Service

b. **Priority wise processing of interrupts:**

The drawback of sequential processing is that it does not take account of relative priority or time critical needs. The alternative form of this is to define priorities for interrupts and to allow an interrupt of higher priority to cause a lower priority interrupts pause until high priority interrupt completes its function.

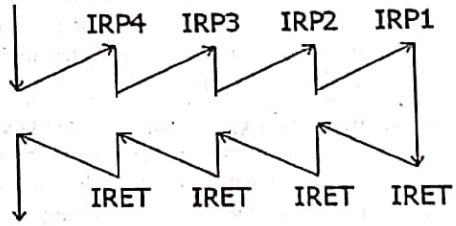


Fig: Priority wise Interrupt service

Interrupt Service Routine

- An interrupt service routine (ISR) is a software routine that hardware invokes in response to an interrupt.
- ISRs examine an interrupt and determine how to handle it.
- ISRs handle the interrupt, and then return a logical interrupt value.
- Its central purpose is to process the interrupt and then return control to the main program.

- An ISR must perform very fast to avoid slowing down the operation of the device and the operation of all lower priority ISRs.
- As in procedures, the last instruction in an ISR should be iret. (ret)

ISR is responsible for doing the following things:

1. **Saving the processor context**

   Because the ISR and main program use the same processor registers, it is the responsibility of the ISR to save the processor's registers before beginning any processing of the interrupt. The processor context consists of the instruction pointer, registers, and any flags. Some processors perform this step automatically.

2. **Acknowledging the interrupt**

   The ISR must clear the existing interrupt, which is done either in the peripheral that generated the interrupt, in the interrupt controller, or both.

3. **Restoring the processor context**

   After interrupt processing, in order to resume the main program, the values that were saved prior to the ISR execution must be restored. Some processors perform this step automatically.

Interrupt Processing in 8085

- Interrupt is signals send by an external device to the processor, to request the processor to perform a particular task or work.
- Mainly in the microprocessor based system the interrupts are used for data transfer between the peripheral and the microprocessor.
- The processor will check the interrupts always at the 2nd T-state of last machine cycle.
- If there is any interrupt it accept the interrupt and send the INTA (active low) signal to the peripheral.

- The vectored address of particular interrupt is stored in program counter.
- The processor executes an interrupt service routine (ISR) addressed in program counter.

It returned to main program by RET instruction.

Types of Interrupts

It supports two types of interrupts.

1. Hardware
2. Software

(1) Software interrupts:

The software interrupts are program instructions. These instructions are inserted at desired locations in a program.

The 8085 has eight software interrupts from RST 0 to RST 7. The vector address for these interrupts can be calculated as follows.

Interrupt number * 8 = vector address

For RST 5; 5 * 8 = 40 = 28H

Vector address for interrupt RST 5 is 0028H

The table shows the vector addresses of all interrupts.

Interrupt	Vector address
RST0 RST1	0000_H 0008_H
RST2 RST3	0010_H 0018_H
RST4 RST5	0020_H 0028_H
RST6 RST7	0030_H 0038_H

(2) Interrupt pins and priorities (hardware interrupts):

An external device initiates the hardware interrupts and placing an appropriate signal at the interrupt pin of the processor.

If the interrupt is accepted then the processor executes an interrupt service routine.

The 8085 has five hardware interrupts

(1) TRAP (2) RST 7.5 (3) RST 6.5

(4) RST 5.5 (5) INTR

Interrupt type	Trigger	Priority	Maskable	Vector address
TRAP	Edge and Level	1st	No	0024H
RST 7.5	Edge	2nd	Yes	003CH
RST 6.5	Level	3rd	Yes	0034H
RST 5.5	Level	4th	Yes	002CH
INTR	Level	5th	Yes	-

TRAP:

- This interrupt is a non-maskable interrupt. It is unaffected by any mask or interrupt enable.
- TRAP bas the highest priority and vectored interrupt.
- TRAP interrupt is edge and level triggered. This means that the TRAP must go high and remain high until it is acknowledged.
- In sudden power failure, it executes a ISR and send the data from main memory to backup memory.
- The signal, which overrides the TRAP, is HOLD signal. (i.e., If the processor receives HOLD and TRAP at the same time then HOLD is recognized first and then TRAP is recognized).
- There are two ways to clear TRAP interrupt.
  1. By resetting microprocessor (External signal)
  2. By giving a high TRAP ACKNOWLEDGE (Internal signal)

RST 7.5:

- The RST 7.5 interrupt is a maskable interrupt.
- It has the second highest priority.
- It is edge sensitive. ie. Input goes to high and no need to maintain high state until it recognized.
- Maskable interrupt. It is disabled by,
  1. DI instruction
  2. System or processor reset.
  3. After reorganization of interrupt.
- Enabled by EI instruction.

RST 6.5 and 5.5:

- The RST 6.5 and RST 5.5 both are level triggered. . ie. Input goes to high and stay high until it recognized.
- Maskable interrupt. It is disabled by
  1. DI, SIM instruction
  2. System or processor reset.
  3. After reorganization of interrupt.
- Enabled by EI instruction.
- The RST 6.5 has the third priority whereas RST 5.5 has the fourth priority.

INTR:

- INTR is a maskable interrupt.
- It is disabled by,
  1. DI, SIM instruction
  2. System or processor reset.
  3. After reorganization of interrupt.
- Enabled by EI instruction.
- Non- vectored interrupt. After receiving INTA (active low) signal, it has to supply the address of ISR.

- It has lowest priority.
- It is a level sensitive interrupts. ie. Input goes to high and it is necessary to maintain high state until it recognized.

The following sequence of events occurs when INTR signal goes high.

1. The 8085 checks the status of INTR signal during execution of each instruction.
2. If INTR signal is high, then 8085 complete its current instruction and sends active low interrupt acknowledge signal, if the interrupt is enabled.
3. In response to the acknowledge signal, external logic places an instruction's opcode on the data bus. In the case of multibyte instruction, additional interrupt acknowledge machine cycles are generated by the 8085 to transfer the additional bytes into the microprocessor.
4. On receiving the instruction, the 8085 save the address of next instruction on stack and execute received instruction.

Priority Interrupt Controller (PIC)

The INTR pin can be used for multiple peripherals and to determine priorities among these devices when two or more peripherals request interrupt service simultaneously, PIC is used. If there are simultaneous requests, the priorities are determined by the encoder, it responds to the higher level input, ignoring the lower level input. The drawback of the scheme is that the interrupting device connected to input I_7 always has the highest priority. The PIC includes a status register and a priority comparator in addition to a priority encoder.

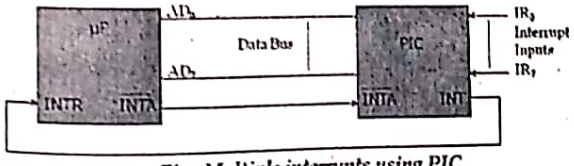
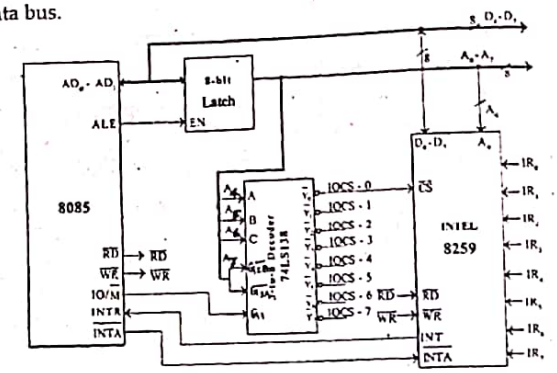


Fig.: Multiple interrupts using PIC

Today this device is replaced by a more versatile one called a programmable interrupt controller 8259A. When an 8259A receives an interrupt signal on one of its IR inputs, it sends an interrupt request signal to the INTR input of the μP. Then INTA pulses will cause the PIC to release vectoring information onto the data bus.



- It requires two internal address and they are A =0 or A = 1.
- It can be either memory mapped or I/O mapped in the system. The interfacing of 8259 to 8085 is shown in figure is I/O mapped in the system.
- The low order data bus lines D0-D7 are connected to D0-D7 of 8259.
- The address line A0 of the 8085 processor is connected to A0 of 8259 to provide the internal address.
- The 8259 require one chip select signal. Using 3-to-8 decoder generates the chip select signal for 8259.
- The address lines A4, A5 and A6 are used as input to decoder.
- The control signal IO/M (low) is used as logic high enables for decoder and the address line A7 is used as logic low enable for decoder.
- The I/O addresses of 8259 are shown in table below.

	Binary Address								Hexa address
	Decoder inputs/ enable			Input to address pin of 8259					
	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
For A_0 of 8259 to be zero	0	0	0	0	x	x	x	0	00
For A_0 of 8259 to be one	0	0	0	0	x	x	x	1	01

Note : *Don't care 'x' is considered as zero.*

Working of 8259 with 8085 Processor

- First the 8259 should be programmed by sending Initialization Command Word (ICW) and Operational Command Word (OCW). These command words will inform 8259 about the following,
  1. Type of interrupt signal (Level triggered / Edge triggered).
  2. Type of processor (8085/8086).
  3. Call address and its interval (4 or 8)
  4. Masking of interrupts.
  5. Priority of interrupts.
  6. Type of end of interrupts.
- Once 8259 is programmed it is ready for accepting interrupt signal. When it receives an interrupt through any one of the interrupt lines IR0-IR7 it checks for its priority and also checks whether it is masked or not.
- If the previous interrupt is completed and if the current request has highest priority and unmasked, then it is serviced.
- For servicing this interrupt the 8259 will send INT signal to INTR pin of 8085.

- In response it expects an acknowledge INTA (low) from the processor.
- When the processor accepts the interrupt, it sends three INTA (low) one by one.
- In response to first, second and third INTA (low) signals, the 8259 will supply CALL opcode, low byte of call address and high byte of call address respectively. Once the processor receives the call opcode and its address, it saves the content of program counter (PC) in stack and load the CALL address in PC and start executing the interrupt service routine stored in this call address.

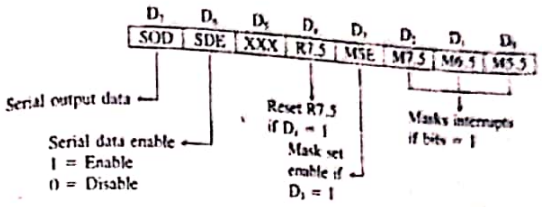
How is INTR pin used in 8085?

The microprocessor checks INTR, one clock period before the last T- state of an instruction cycle. In the 8085, the call instructions require 18 T-states; therefore the INTR pulse should be high at least for 17.5 T-states. The INTR can remain high until the interrupt flip flop is set by the EI instruction in the service routine. If it remains high after the execution of the EI instruction, the processor will be interrupted again, as if it was a new interrupt.

Interrupt Instructions

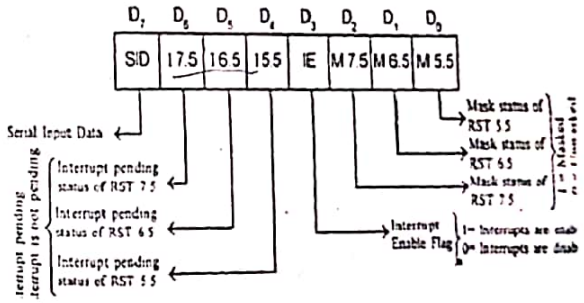
SIM instruction:

- The 8085 provide additional masking facility for RST 7.5, RST 6.5 and RST 5.5 using SIM instruction.
- This is a multipurpose instruction and used to implement the 8085 interrupts 7.5, 6.5, 5.5, and serial data output.
- The masking or unmasking of RST 7.5, RST 6.5 and RST 5.5 interrupts can be performed by moving an 8-bit data to accumulator and then executing SIM instruction.
- The format of the 8-bit data is shown below.



- SOD — Serial Output Data: Bit D_7 of the accumulator is latched into the SOD output line and made available to a serial peripheral if bit $D_6 = 1$.
- SDE — Serial Data Enable: If this bit = 1, it enables the serial output. To implement serial output, this bit needs to be enabled.
- XXX — Don't Care
- R7.5 — Reset RST 7.5: If this bit = 1, RST 7.5 flip-flop is reset. This is an additional control to reset RST 7.5.
- MSE — Mask Set Enable: If this bit is high, it enables the functions of bits D_2, D_1, D_0. This is a master control over all the interrupt masking bits. If this bit is low, bits D_2, D_1, and D_0 do not have any effect on the masks.
- M7.5 — D_2 = 0, RST 7.5 is enabled.
  = 1, RST 7.5 is masked or disabled.
- M6.5 — D_1 = 0, RST 6.5 is enabled.
  = 1, RST 6.5 is masked or disabled.
- M5.5 — D_0 = 0, RST 5.5 is enabled.
  = 1, RST 5.5 is masked or disabled.

RIM instruction:



- The status of pending interrupts can be read from accumulator after executing RIM instruction.
- This is a multipurpose instruction used to read the status of RST 7.5, 6.5, 5.5 and read serial data input bit.

- When RIM instruction is executed an 8-bit data is loaded in accumulator, which can be interpreted as shown in above fig.
- Bits 0-2 show the current setting of the mask for each of RST 7.5, RST 6.5 and RST 5.5. They return the contents of the three masks flip flops. They can be used by a program to read the mask settings in order to modify only the right mask.
- Bit 3 shows whether the maskable interrupt process is enabled or not. It returns the contents of the Interrupt Enable Flip Flop. It can be used by a program to determine whether or not interrupts are enabled.
- Bits 4-6 show whether or not there are pending interrupts on RST 7.5, RST 6.5, and RST 5.5. Bits 4 and 5 return the current value of the RST5.5 and RST6.5 pins. Bit 6 returns the current value of the RST7.5 memory flip flop.
- Bit 7 is used for Serial Data Input. The RIM instruction reads the value of the SID pin on the microprocessor and returns it in this bit.

DI

- Disable interrupts
- The interrupt enable flip-flop is reset and all the interrupts except the TRAP are disabled. No flags are affected.
- 1 byte instruction
- Example: DI

EI

- Enable interrupts
- The interrupt enable flip-flop is set and all interrupts are enabled.
- No flags are affected.
- After a system reset or the acknowledgement of an interrupt, the interrupt enable flip flop is reset, thus disabling the interrupts.

- This instruction is necessary to enable the interrupts (except TRAP).
- 1 byte instruction
- Example: EI

Interrupt Processing in 8086

The meaning of 'interrupts' is to break the sequence of operation. While the CPU is executing a program, an 'interrupt' breaks the normal sequence of execution of instructions, diverts its execution to some other program called Interrupt Service Routine (ISR).After executing ISR , the control is transferred back again to the main program. Interrupt processing is an alternative to polling.

Interrupt Pins

INTR and NMI Hardware inputs

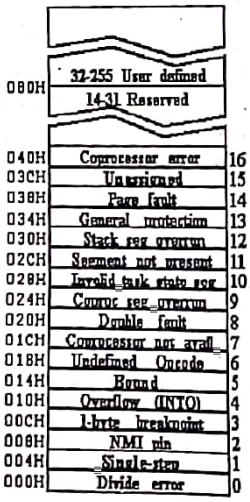
- INTR is a maskable hardware interrupt. The interrupt can be enabled/disabled using STI/CLI instructions or using more complicated method of updating the FLAGS register with the help of the POPF instruction.
- When an interrupt occurs, the processor stores FLAGS register into stack, disables further interrupts, fetches from the bus one byte representing interrupt type, and jumps to interrupt processing routine address of which is stored in location 4 * <interrupt type>. Interrupt processing routine should return with the IRET instruction.
- NMI is a non-maskable interrupt. Interrupt is processed in the same way as the INTR interrupt. Interrupt type of the NMI is 2, i.e. the address of the NMI processing routine is stored in location 0008h. This interrupt has higher priority than the maskable interrupt.
- Ex: NMI, INTR.

Interrupt vector table and its organization:

- An interrupt vector is a pointer to where the ISR is stored in memory.

- All interrupts (vectored or otherwise) are mapped onto a memory area called the Interrupt Vector Table (IVT).
  - The IVT is usually located in memory page 00 (0000H - 00FFH).
  - The purpose of the IVT is to hold the vectors that redirect the microprocessor to the right place when an interrupt arrives.

Interrupt Vector Table (IVT) is a 1024 bytes sized table that contains addresses of interrupts. Each address is of 4 bytes long of the form offset:segment, which represents the address of a routine to be called when the CPU receives an interrupt. IVT can hold maximum of 256 addresses (0 to 255). The interrupt number is used as an index into the table to get the address of the interrupt service routine. IVT act as pointers, unlike function call IVT need number as an argument then as a result IVT point us to interrupt service routine (ISR). ISR executes its code, when ISR finished then returns back to original statement. Interrupt vector table is a global table situated at the address 0000:0000H. The interrupt vector table is a feature of the Intel 8086/8088 family of microprocessors.



The interrupt vector table is located in the first 1024 bytes of memory at addresses 000000H through 0003FFH.

There are 256 4-byte entries (segment and offset in real mode).

Seg high	Seg low	Offset high	Offset low
Byte 3	Byte 2	Byte 1	Byte 0

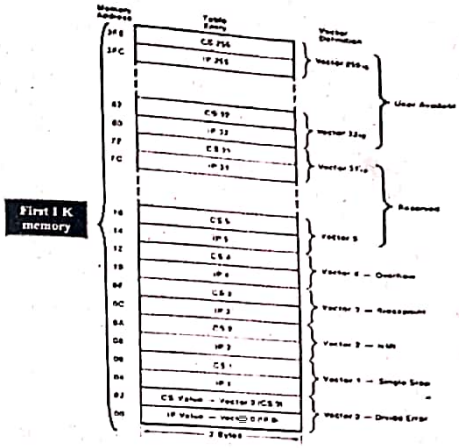


Fig: IVT Structure (Organization)

Functions associated with INT00 to INT04:

INT Number	Physical Address
INT 00	00000
INT 01	00004
INT 02	00008
:	:
:	:
INT FF	003FC

INT 00 (divide error)

- INT00 is invoked by the microprocessor whenever there is an attempt to divide a number by zero.
- ISR is responsible for displaying the message "Divide Error" on the screen

INT 01

- For single stepping the trap flag must be 1
- After execution of each instruction, 8086 automatically jumps to 00004H to fetch 4 bytes for CS: IP of the ISR.
- The job of ISR is to dump the registers on to the screen

INT 02 (Non maskable Interrupt)

- When ever NMI pin of the 8086 is activated by a high signal (5v), the CPU Jumps to physical memory location 00008 to fetch CS:IP of the ISR associated with NMI.

INT 03 (break point)

- A break point is used to examine the CPU and memory after the execution of a group of Instructions.
- It is one byte instruction whereas other instructions of the form "INT nn" are 2 byte instructions.

INT 04 (Signed number overflow)

- There is an instruction associated with this INT 0 (interrupt on overflow).
- If INT 0 is placed after a signed number arithmetic as IMUL or ADD the CPU will activate INT 04 if 0F = 1.
- In case where 0F = 0 , the INT 0 is not executed but is bypassed and acts as a NOP.

Software and Hardware Interrupts

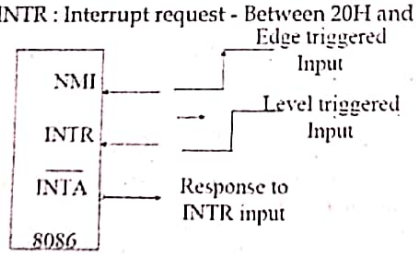
Types of Interrupts: There are two types of Interrupts in 8086. They are:

I. **Hardware interrupts (external interrupts):** The Intel microprocessors support hardware interrupts through:

- Two pins that allow interrupt requests, INTR and NMI
- One pin that acknowledges, INTA, the interrupt requested on INTR.

Performance of hardware interrupts:

- NMI : Non maskable interrupts - TYPE 2 Interrupt
- INTR : Interrupt request - Between 20H and FFH



II. **Software interrupts (internal interrupts and instructions):** Software interrupts can be caused by:

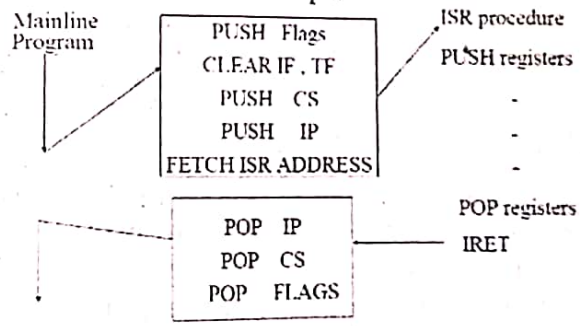
- INT instruction - breakpoint interrupt. This is a type 3 interrupt.
- INT <interrupt number> instruction - any one interrupt from available 256 interrupts.
- INTO instruction - interrupt on overflow
- Single-step interrupt - generated if the TF flag is set. This is a type 1 interrupt. When the CPU processes this interrupt it clears TF flag before calling the interrupt processing routine.
- Processor exceptions: Divide error (Type 0), unused opcode (type 6) and Escape opcode (type 7).
- Software interrupt processing is the same as for the hardware interrupts.

  E.g. INT n (Software instructions)

  Control is provided through:

  - IF and TF flag bits
  - IRET and IRETD

Performance of software interrupts:



- It decrements SP by 2 and pushes the flag register on the stack.
- Disables INTR by clearing the IF.

- It resets the TF in the flag Register.
- It decrements SP by 2 and pushes CS on the stack.
- It decrements SP by 2 and pushes IP on the stack.
- Fetch the ISR address from the interrupt vector table.

Interrupt Priorities

Interrupt	Priority
Divide Error, INT(n),INTO	Highest
NMI	↓
INTR	
Single Step	Lowest

8086 Interrupts in Summary

An interrupt is a signal indicating that an event needing immediate attention has occurred. There are two types of interrupts; external interrupt generated outside CPU by other hardware, and internal interrupt generated within CPU as a result of an instruction or operation.

In 8086, two types of interrupts occur.

1. **Hardware interrupts**

These are external interrupts which uses NMI and INTR. When hardware interrupt is detected, the interrupt controller sends the interrupt code to the processor. When the code is finally acquired by the processor either from TNT opcode or from the interrupt controller. This is used by the processor to index the interrupt vector table to find the address of the interrupt handler. The 8086 specifies 256 different interrupts specified by type number or vector which is a pointer into IVT. The pointer is cs:ip values .

2. **Software interrupts**

Software interrupts are used to publish internal services to outside world. These are internal interrupts like INT and INTO, and trap such as divide by zero or single step. Other software interrupts also included by 8086 processor. INT is used for breakpoint and INTO is used for overflow

interrupt. Single step is the debugging mode interrupt for each instruction and divide error is dividing by zero interrupt.

Interrupt vector table:

It is located in the first 1024 bytes of memory at address 000000-0003FFH. It contains 256 different 4-byte interrupt vectors. An interrupt vector contains the segment & offset address of the interrupt service provider.

DOS & BIOS interrupts:

Dos interrupts services link applications with os services such as opening file, reading, writing content using certain functions of INT 4H. BIOS interrupts control the screen disk controller and keyboard operation using INT 10H, 13H,16H etc.

❖ ❖ ❖

Chapter – 6

ADVANCED TOPICS

Multiprocessing Systems

Traditionally, the computer has been viewed as a sequential machine. Most computer programming languages require the programmer to specify algorithms as sequence of instructions. Processor executes programs by executing machine instructions in a sequence and one at a time. This view of the computer has never been entirely true.

As computer technology has evolved and as the cost of computer hardware has dropped, computer designers have sought more and more opportunities for parallelism, usually to enhance performance and availability. Multiprocessing is an example of parallelism which uses multiple CPUS sharing the common resources such as memory, storage device etc.

Characteristics of multiprocessing system

- Contains two or more similar general purpose processors of comparable capability.
- All processors share access to common memory.
- All processors share access to I/O devices either through the same channels that provide paths to the same devices.
- System is controlled by an integrated operating system that provides interaction between processors and their programs.

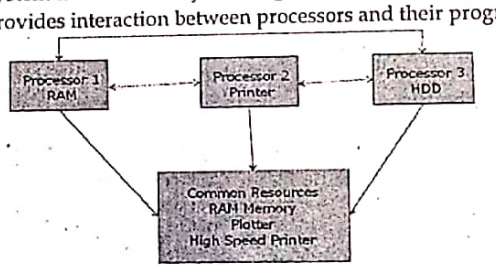


Fig.: Organization of multiprocessing system

Here, the processors can communicate with each other through memory. The CPUs can directly exchange signals as indicated by dotted line. The organization of multiprocessor system can be divided into three types.

1. **Time shared or common bus:**

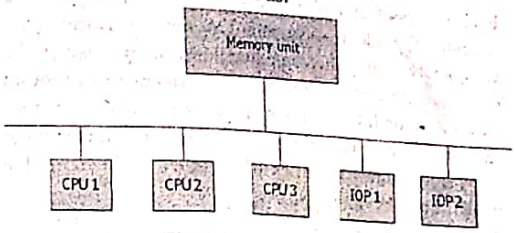


Fig.: Time shared system

There are number of CPUs, I/O modules and memory modules connected to the same bus. So the time shared system must distinguish the modules on the bus to determine the source and destination of the data. Any module in the bus can temporarily act as a master. When one module is controlling the bus, the other should be locked out. The access to each module is divided on the basis of time. The time shared multiprocessing system has the following advantages and drawbacks.

Advantages

Simplicity

The physical interface and the addressing time sharing logic of each processor remains the same as in a single processor system, so it is very simplest approach.

Flexibility

It is easy to expand the system by attaching more CPUs to the bus.

Reliability

The failure of any attached device should not the failure of the whole system.

Drawback

The speed of the system is limited by the cycle time because all memory references must pass through the common bus.

2. **Multiport memory:**

Each processor and I/O module has dedicated path to each memory module this system has more performance and complexity than earlier one. For this system, it is possible to configure portions of memory as private to one or more CPUs and/or I/O modules. This feature allows increasing security against unauthorized access, and the storage of recovery routines in areas of memory not susceptible to modification by other processors.

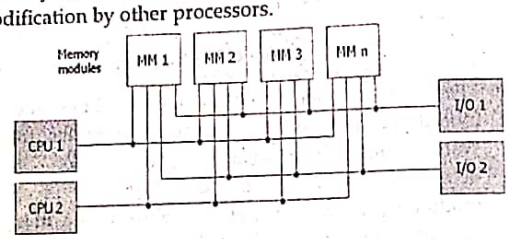


Fig.: Multiport memory system

3. **Central control unit :**

It manages the transfer of separate data streams back and forth between independent modules like CPU, memory and I/O. The controller can buffer requests and perform arbitration and timing functions. It can also pass status and control messages between CPUs. All the co-ordination is concentrated in the central control unit un-disturbing the modules. It is more flexible and complex as well.

Real and Pseudo-Parallelism

- Traditionally, software has been written for serial computation:
  - To be run on a single computer having a single Central Processing Unit (CPU).
  - A problem is broken into a discrete series of instructions.
  - Instructions are executed one after another.
  - Only one instruction may execute at any moment in time.
- In the simplest sense, parallelism is the simultaneous use of multiple compute resources to solve a computational problem:
  - To be run using multiple CPUs
  - A problem is broken into discrete parts that can be solved concurrently
  - Each part is further broken down to a series of instructions
  - Instructions from each part execute simultaneously on different CPUs
- Real parallelism consists of the parallel modes of physical devices so that each can carry parallel operations to each other. Core parallelism consists of real parallelism. Multiple core processes which are physically different and performs their own operations in parallel.

  Two or more processes are actually running at once because the computer system is a parallel processor i.e., has more than one processor.

  Process 1 CPU1 →

  Process 2 CPU2 →

  Fig.: True/real parallelism
- Pseudo parallelism consists of the same device carrying the parallel operation. We can logically manage the parallelism for system. Concurrent processing using parallelism is the pseudo parallelism which operates either in time division or using other types of parallel algorithms.

  One have the illusion of a parallel processor. Two processes are switched and executed concurrently through a single processor.

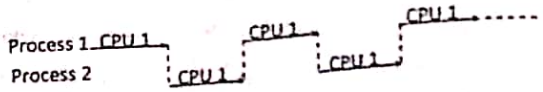


Fig.: Pseudo-parallelism

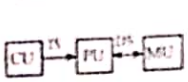
Flynn's Classification

There are different ways to classify parallel computers. One of the more widely used classifications, in use since 1966, is called Flynn's classification. Flynn's classification distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of *Instruction* and *Data*. Each of these dimensions can have only one of two possible states: *Single* or *Multiple*. The matrix below defines the 4 possible classifications according to Flynn:

SISD Single Instruction, Single Data	**SIMD** Single Instruction, Multiple Data
MISD Multiple Instruction, Single Data	**MIMD** Multiple Instruction, Multiple Data

1. Single Instruction, Single Data (SISD):

A single processor executes a single instruction stream to operate on data stored in a single memory.



CU: Control Unit
PU: Processing Unit
MU: Memory Unit
IS: Instruction Stream
DS: Data Stream

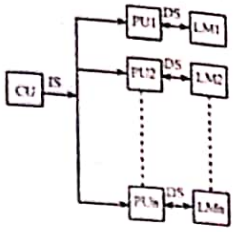
Fig.: SISD

Examples: Uniprocessors such as older generation mainframes, minicomputers and workstations, most modern day PCs.

2. Single Instruction, Multiple Data (SIMD):

Single machine instructions controls the simultaneous execution of a number of processing elements on a lockstep basis. Each processing element has an associated data

memory, so that each instruction is executed on a different set of data by the different processors.



LM: Local Memory

Fig.: SIMD

Examples:

- Array Processors: Connection Machine CM-2, MasPar MP-1 & MP-2, ILLIAC IV
- Vector Processors : IBM 9000, Cray X-MP, Y-MP & C90, Fujitsu VP, NEC SX-2, Hitachi S820, ETA10

Most modern computers, particularly those with graphics processor units (GPUs) employ SIMD instructions and execution units.

3. Multiple Instruction, Single Data (MISD):

A sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence. MISD structure is only of theoretical interest since no practical systems has been constructed.

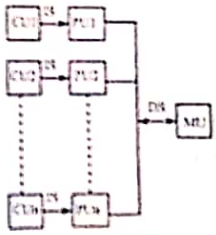


Fig.: MISD

Some conceivable uses of MISD might be:

- Special purpose stream processor (e.g. digital filters)
- Multiple cryptography algorithms attempting to crack a single coded message.

4. **Multiple Instruction, Multiple Data (MIMD):**

A set of proccessors simultaneously execute different instructions sequences on different data sets.

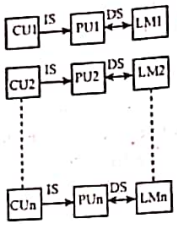


Fig.: MIMD

Examples:

- Most current supercomputers, networked parallel computer clusters and "grids", multi-core PCs which include SMPs (symmetric multiprocessors), NUMA systems.

Instruction Level, Thread Level and Process Level Parallelism

Instruction level parallelism

Instruction-level parallelism (ILP) is a measure of how many of the operations in a computer program can be performed simultaneously. Consider the following program:

1. e = a + b
2. f = c + d
3. g = e * f

Operation 3 depends on the results of operations 1 and 2, so it cannot be calculated until both of them are completed. However,

operations 1 and 2 do not depend on any other operation, so they can be calculated simultaneously. If we assume that each operation can be completed in one unit of time, then these three instructions can be completed in a total of two units of time, giving an ILP of 3/2.

A goal of compiler and processor designers is to identify and take advantage of as much ILP as possible. Ordinary programs are typically written under a sequential execution model where instructions execute one after the other and in the order specified by the programmer. ILP allows the compiler and the processor to overlap the execution of multiple instructions or even to change the order in which instructions are executed.

How much ILP exists in programs is very application specific. In certain fields, such as graphics and scientific computing the amount can be very large. However, workloads such as cryptography exhibit much less parallelism.

Micro-architectural techniques that are used to exploit ILP include:

- Instruction pipelining where the execution of multiple instructions can be partially overlapped.
- Superscalar execution, VLIW, and the closely related Explicitly Parallel Instruction Computing concepts, in which multiple execution units are used to execute multiple instructions in parallel.

Thread Level Parallelism

Thread parallelism (also known as task parallelism, function parallelism and control parallelism) is a form of parallelization of computer code across multiple processors in parallel environments. Thread parallelism focuses on distributing execution processes (threads) across different parallel computing nodes. It contrasts to data parallelism as another form of parallelism.

It was later recognized that finer-grain parallelism existed with a single program. A single program might have several

threads (or functions) that could be executed separately or in parallel. Some of the earliest examples of this technology implemented input/output processing such as direct memory access as a separate thread from the computation thread. A more general approach to this technology was introduced in the 1970s when systems were designed to run multiple computation threads in parallel. This technology is known as multi-threading (MT).

As a simple example, if we are running code on a 2-processor system (CPUs "a" & "b") in a parallel environment and we wish to do tasks "A" and "B" , it is possible to tell CPU "a" to do task "A" and CPU "b" to do task 'B" simultaneously, thereby reducing the run time of the execution.

Thread parallelism emphasizes the distributed (parallelized) nature of the processing (i.e., threads), as opposed to the data (data parallelism). Most real programs fall somewhere on a continuum between thread parallelism and data parallelism.

The pseudocode below illustrates task parallelism:

```
program:
...
if CPU="a" then
  do task "A"
else if CPU="b" then
  do task "B"
end if
...
end program
```

The goal of the program is to do some net total task ("A+B"). If we write the code as above and launch it on a 2-processor system, then the runtime environment will execute it as follows.

- In an SPMD system, both CPUs will execute the code.
- In a parallel environment, both will have access to the same data.
- The "if" clause differentiates between the CPU's. CPU "a" will read true on the "if" and CPU "b" will read true on the "else if", thus having their own task.

- Now, both CPU's execute separate code blocks simultaneously, performing different tasks simultaneously.

Code executed by CPU "a":

```
program:
...
do task "A"
...
end program
```

Code executed by CPU "b":

```
program:
...
do task "B"
...
end program
```

This concept can now be generalized to any number of processors.

Data parallelism

Data parallelism is parallelism inherent in program loops, which focuses on distributing the data across different computing nodes to be processed in parallel. Parallelizing loops often leads to similar (not necessarily identical) operation sequences or functions being performed on elements of a large data structure. Many scientific and engineering applications exhibit data parallelism.

A loop-carried dependency is the dependence of a loop iteration on the output of one or more previous iterations. Loop-carried dependencies prevent the parallelization of loops. For example, consider the following pseudocode that computes the first few Fibonacci numbers:

```
PREV1 := 0
PREV2 := 1
do:
  CUR := PREV1 + PREV2
```

PREV1 := PREV2

PREV2 := CUR

while (CUR < 10)

This loop cannot be parallelized because CUR depends on itself (PREV2) and PREV1, which are computed in each loop iteration. Since each iteration depends on the result of the previous one, they cannot be performed in parallel. As the size of a problem gets bigger, the amount of data-parallelism available usually does as well.

Process Level Parallelism

Process-level parallelism is the use of one or more central processing units (CPUs) within a single computer system. The term also refers to the ability of a system to support more than one processor and/or the ability to allocate tasks between them. There are many variations on this basic theme, and the definition of process level parallelism can vary with context, mostly as a function of how CPUs are defined (multiple cores on one die, multiple dies in one package, multiple packages in one system unit, etc.).

This varies, depending upon who you talk to. In the past, a CPU (Central Processing Unit) was a singular execution component for a computer. Then, multiple CPUs were incorporated into a node. Then, individual CPUs were subdivided into multiple "cores", each being a unique execution unit. CPUs with multiple cores are sometimes called "sockets". The result is a node with multiple CPUs, each containing multiple cores.

- During the past 20+ years, the trends indicated by ever faster networks, distributed systems, and multi-processor computer architectures (even at the desktop level) clearly show that parallelism is the future of computing.
- In this same time period, there has been a greater than 1000x increase in supercomputer performance, with no end currently in sight.

Inter-process communication

In computing, inter-process communication (IPC) is a set of methods for the exchange of data among multiple threads in one or more processes. Processes may be running on one or more computers connected by a network. IPC methods are divided into methods for message passing, synchronization, shared memory, and remote procedure calls (RPC). The method of IPC used may vary based on the bandwidth and latency of communication between the threads, and the type of data being communicated.

There are several reasons for providing an environment that allows process cooperation:

- Information sharing
- Speedup
- Modularity
- Convenience
- Privilege separation

IPC may also be referred to as inter-thread communication and inter-application communication.

The combination of IPC with the address space concept is the foundation for address space independence/isolation.

The single operating system controls the use of system resources in a multiprocessing environment. In this system, multiple jobs or process may be active at one time. The responsibility of operating system or system software is to schedule the execution and to allocate resources. The functions of multiprocessor operating system are:

- An interface between users and machine
- Resource management
- Memory management
- Prevent deadlocks
- Abnormal program termination

- Process scheduling
- Managers security

Resource Allocation

In computing, **resource allocation** is necessary for any application to be run on the system. When the user opens any program, this will be counted as a process, and therefore, requires the computer to allocate certain resources for it to be able to run. Such resources could be access to a section of the computer's memory, data in a device interface buffer, one or more files, or the required amount of processing power.

A computer with a single processor can only perform one process at a time, regardless of the amount of programs loaded by the user (or initiated on start-up). Computers using single processors appear to be running multiple programs at once because the processor quickly alternates between programs, processing what is needed in very small amount of time. This process is known as multitasking or *time slicing*. The time allocation is automatic, however higher or lower priority may be given to certain processes, essentially giving high priority programs more/bigger slices of the processor's time.

On a computer with multiple processors, different processes can be allocated to different processors so that the computer can truly multitask. Some programs, such as Adobe Photoshop, which can require intense processing power, have been coded so that they are able to run on more than one processor at once, thus running more quickly and efficiently.

Deadlock

A process requests resources; if the resources are not available at that time, the process enters a wait state. Waiting processes may never again change state, because the resources they have requested are held by other waiting processes. This situation is called a deadlock.

Processes need access to resources in reasonable order. Suppose a process holds resource A and requests resource B, at

same time another process holds B and requests A; both are blocked and remain in deadlock.

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause. Usually the event is release of a currently held resource. None of the processes can run, release resources and then be awakened.

Deadlock can be studied under two categories: (a) process deadlock (b) system deadlock.

Conditions for deadlock

(i) **Mutual exclusion**

Shared resources are used in a mutually exclusive manner.

(ii) **Hold and wait**

Processes hold onto resources they already have while waiting for allocation of other resources.

(iii) **No preemption**

Resources cannot be preempted until the process release them.

(iv) **Circular wait**

Circular chain of processes exist in which each process holds resources wanted by the next process in chain.

Operating System

An **operating system** is a collection of software that provides services for computer programs. In other terms, an operating system is a composition of a kernel and utility programs; the kernel controls the allocation of hardware resources while the utility programs enhance the usefulness of the computer. An operating system kernel is a computer program that serves as an intermediary layer between the hardware and application programs. A kernel makes it possible for software to interact with the underlying hardware of the operating system. Such software (mostly application software) achieves this by issuing service

requests to the kernel. (These requests are called system calls.) When a service request is received, the kernel translates it into instructions for the CPU or other electronic components of the computer to execute.

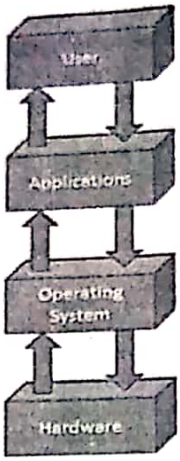


Fig.: Operating system

Features of Operating System

When talking about *features* of an operating system, often they get mixed up with its *functions*. We define an operating system's feature as a prominent attribute of the operating system; in other words, its major components.

i. **System calls**

On modern microprocessors, there are at least two modes of operation: kernel mode and user mode. If an application running in user mode tries to perform a privileged operation (such as directly accessing the hardware), the CPU will most likely throw an exception. So, then, how does an application read input from the keyboard or write to the screen? It does so by sending a request to the kernel. Of course, this slows down the operation, but ensures that application programs

do not execute code that could damage or compromise the system. These requests that application programs send to the kernel are called *system calls*. A system call can thus be simply defined as a request by a computer program to the operating system's kernel.

When a system call is invoked, control is transferred to the kernel which, in turn, determines whether the calling application should be granted the requested service. If granted, the kernel executes the necessary instructions, causes a switch into user mode, and returns control back to the calling program. Most operating systems however provide an intermediary interface that sits between applications and the system calls layer, in a form of a library or an Application Programming Interface (API). Such an intermediary interface makes it possible for programs written in high level languages to invoke system calls. Moreover, it is easier and more portable to use a library or an API than to code the system call in assembly language instructions.

ii. **Device drivers**

A computer system is usually made up of several devices such as disk drives, keyboards, mice, video adapters, sound cards, etc. When a user attaches such devices to their computer, they expect the operating system to identify the device and make use of it. Indeed, the operating system may know *what* the device is but not *how* to communicate with it. The latter problem is solved by means of a driver. A **device driver** is a computer program that controls a particular device attached to a computer. It provides an interface through which the operating system can transparently make calls to the device. In fact, device drivers have built-in functions that are meant to be called by the operating system or other privileged programs.

Devices are generally slower compared with the CPU. This means that whereas the CPU could be doing other stuff, it many times waits for a busy but slow device to finish

whatever job it is doing. This bad behavior, however, is mitigated by the use of *hardware interrupts*. Interrupts cause control to be transferred to a routine designed to process the interrupt. For example, when a key is pressed on a computer keyboard, a hardware interrupt is generated, which invokes the keyboard device driver. After the driver has finished processing the event, control is returned back to the interrupted program. However, only a few peripherals support interrupts which means that drivers have to poll the hardware, i.e. ask whether there is an event to process.

iii. **File system**

Every computer file is stored in a linear space on a storage device of finite capacity. Each file has its *address* on storage, which is determined by the number of byte offsets from the beginning of the storage medium. But then, there is the need for a structure that tells where one piece of data begins and a where it ends—a file system. File systems keep track of unused space on the disk as well as additional information about each file such as the name, size, owner, creation date, access control, encryption, etc. What is more, file systems manage the directory structure and the mapping of file names to file control blocks. A file system can thus be defined as a structured data representation and a set of metadata that describe stored data.

iv. **User interface**

This is another feature of an operating system: its user interface. An operating system's user interface determines how the user interacts with the computer. The two most common forms of a user interface are the Command Line Interface (CLI) and the Graphical User Interface (GUI). A CLI provide a prompt at which commands can be given line-by-line. This kind of interface is usually implemented with a program called a *command line shell*, which accepts commands as text input and converts them to the appropriate operating system functions. CLIs can be quite powerful for experienced users, but if one does not know the

system well enough, they can become quite lost. Examples of CLIs are the UNIX shells and the Windows Command Prompt.

In contrast, a GUI provides a visual environment where a device (such as a mouse) is used to navigate the system and perform tasks. Unlike CLIs where performing a task can become slow and error-prone (such as when very long commands are to be entered), GUIs present the user with widgets that trigger some of the operating system's commands, reducing complexity and the need to memorize command names and their parameters. For many users, a GUI presents a more accessible user interface; however, the choice of a user interface is simply a matter of personal preference. Examples of GUIs are those implemented in Microsoft Windows, Apple's Mac OS X, and GNOME/KDE for the X Windows system on Unix-like operating systems.

Objectives of operating System

The objectives of the operating system are :

- To make the computer system convenient to use in an efficient manner.
- To hide the details of the hardware resources from the users.
- To provide users a convenient interface to use the computer system.
- To act as an intermediary between the hardware and its users, making it easier for the users to access and use other resources.
- To manage the resources of a computer system.
- To keep track of who is using which resource, granting resource requests, and mediating conflicting requests from different programs and users.
- To provide efficient and fair sharing of resources among users and programs.

Characteristics of operating system

Here is a list of some of the most prominent characteristic features of operating systems.

- **Memory management** – Keeps track of the primary memory, i.e., what part of it is in use by whom, what part is not in use, etc. and allocates the memory when a process or program requests it.
- **Processor management** – Allocates the processor (CPU) to a process and de-allocates the processor when it is no longer required.
- **Device management** – Keeps track of all the devices. This is also called I/O controller that decides which process gets the device, when, and for how much time.
- **File management** – Allocates and de-allocates the resources and decides who gets the resources.
- **Security** – Prevents unauthorized access to programs and data by means of passwords and other similar techniques.
- **Job accounting** – Keeps track of time and resources used by various jobs and/or users.
- **Control over system performance** – Records delays between the request for a service and from the system.
- **Interaction with the operators** – Interaction may take place via the console of the computer in the form of instructions. The Operating System acknowledges the same, does the corresponding action, and informs the operation by a display screen.
- **Error-detecting aids** – Production of dumps, traces, error messages, and other debugging and error-detecting methods.
- **Coordination between other software and users** – Coordination and assignment of compilers, interpreters,

assemblers, and other software to the various users of the computer systems.

Different Microprocessor Architectures

Accumulator based architecture:

- Accumulator is a most significant register then compared to other registers and most of the arithmetic and logic operations are performed using the accumulator .performed via the accumulator.
- The internal architecture of 8085 shows that the registers B,C,D,E,H and L are connected with the ALU through the accumulator and temporary register.
- Data can only enter into the ALU from accumulator and the out pot of the ALU can be stored in accumulator through data bus.

Register based architecture:

- All the arithmetic and logical operation consists of the operands of any registers (A, B, C, D etc). The input/output operations here register A and register B are similar.
- The internal architecture of 8086 shows that the registers A, B, C, D and others directly with the ALU.
- Data can enter into the ALU from any registers.
- For I/O operation register A only can be used.
- The advantages of register based architecture is that extendibility and flexibility in programming.
- The processor will be enhanced in register based architecture.
- The disadvantage is requirement of complex circuitry.

RISC and CISC Architectures

CISC based architecture:

Complex instruction set computers is the acronym for CISC and machine based on this architecture have complex instructions.

Most of the personal computers which are used today are based on CISC architecture. Following are the characteristics of CISC machines.

- CISC machines have complex instructions which need a large cycles for execution.
- The transfer of data among the register is much faster than among memory and processor. In CISC, various instructions based on memory and processer so the processing speed gets reduced.
- Pipelining is the process of fetching one instruction when another instruction is executing in parallel. Due to complex instruction this feature cannot be heavily used in CISC machines.
- Micro-operations form the instruction and instruction form he micro-program which is written in control memory to perform timing and sequencing of the micro-operations implemented in CISC.
- CISC machines have large number of complex instructions based on multiple numbers of addressing modes.
- CISC machines processer does not consist of large number of registers due to large cost. So these machines have to perform various memory read and write operations.
- CISC Machines are preferable where the speed of processer is not the prime issue and where general applications are to be handled. Processers like 8085, 8086, 8086,8086,8086,8086 etc are based on CISC processers and even today's pc.

RISC based architecture:

The term RISC represents reduced instruction set computing/ computers. It focuses on a small set of instructions which simplifies the hardware design and improves the processer performance. Generally, RISC processers include the following features.

- The number of instructions is minimized i.e., less than 100 and each can be executed in a single clock cycle the data

path cycle time is the time required to fetch the operations from the registers, run them through ALU and store the result back to register which is very small in RISC.

- the number of addressing modes is relatively low i.e. less than 3 and only few instructions are memory referencing link lode and store so that RISC permits heavy pipelining.
- The processing is register intensive, meaning it includes many more registers and most of the computing is performed using registers range from 32 registers to more than 100 registers.
- There is no micro-programs in RISC machines for interpreting the instructions. The most instructions are directly executed by the hardware.
- Design considerations include support for high level languages through appropriate selection of instruction and optimization of compilers.
- Usually in scientific and research oriented tasks where the speed of processers is apex issue, RISC machines are used and replacing CISC now due to reduce the price of hardware.
- RISC based system are R4400SC from MIPS, PA7100 from HP, Power PC from Apple, IBM and Motorola, super sparc from sun, i860™ from Intel etc.

Differences between RISC and CISC architecture:

RISC	CISC
1. Simple instructions taking one cycle.	1. Complex instructions taking multiple no. of cycles.
2. Most operations are register to register with only LOAD and STORE operations accessing memory.	2. In CISC machine, memory read & write operations form inherent part for executing the instructions.
3. Heavily pipelined, increasing performance.	3. Not/less pipelined

RISC	CISC
4. Multiple register sets.	4. Single register set
5. Instructions have fixed format.	5. Instructions have variable format.
6. Few number of instructions and addressing modes.	6. Large number of instructions and addressing modes.
7. Complexity is in the design of compiler.	7. Complexity is in microprogram.
8. Instructions are executed by hardware.	8. Instructions in CISC machine are interpreted by microprogram.
9. Since RISC processors are simpler than CISC processors, they can be designed more quickly.	9. CISC processors have comparatively longer design cycle.
10. Examples include SPARC, MIPS R4000, etc.	10. Examples include IBM 370/168, VAX 11/780, 8085, 8086, 80286, 80386, etc.

Digital Signal Processors

A special-purpose CPU that provides ultra-fast instruction sequences such as shift & add, and multiply & add which are commonly used in math-intensive signal processing application is termed as digital signal processor (DSP). Put simply, a DSP is a specialized microprocessor with an architecture optimized for the fast operational needs of digital signal processing.

The real time signals such as pressure, temperature, voice are continuous time varying signals are known as analog signals. The process of conversion of the analog signals into discrete signal means digital signal which reduces the redundancy and make more immune to noise is called digital signal processing .The digital signal processors take input the digital data for that every analog data is to be converted into digital by using A/D converter. For example microphone converts sound into electrical energy i.e.

analog, it is sampled by A/D converter and converted into dig one which is fed to the DSP processor. After performing these data DSP can transfer the discrete data to D/A converter which further to speaker to convert electrical signal to sound.

The whole function is carried out by DSP processers using hardware like microphone, transducer, A/D converter, D/A converter, speaker etc and software like C or MATLAB which carried out FFT (fast Fourier transform). DSP processors have low processing speed due to very curtail signals need to be operated. The micro-processers and computers we used today are based on non Neumann architecture where the instruction defines both the operation and data.

So the DSP processors should be fast processing and for that we need to design best architecture which follows Harvard Architecture where separate buses for instructions and data are used. DSP chips are specially designed for particular application and they are not used for general type of processing like microprocessors do. There are very few manufacturers for DSP chip, one of them is the Texas instruments, USA. Its TMS320C series is worldwide popular and can be used for implementing various types of signal processing application.

❖❖❖

Er. Hari Prasad Aryal

M.Sc. in Information System Engineering, PU

Er. Hari Prasad Aryal is Sr. Technical Officer in Smart Choice Technologies (P.) Ltd. (SCT-Network) and is aligned with the company since 2006 AD. Mr. Aryal with teaching experience of more than 12 years is also an Assistant Professor in Himalaya College of Engineering. He lectures Microprocessors, Computer Organization and Architecture, and Instrumentation-II in several engineering colleges (Himalaya College of Engineering, Advanced College of Engineering and Management, Sagarmatha Engineering College, Hillside College of Engineering).

Er. Shyam Dahal

M.S[illegible] and Knowledge
Eng[illegible]chowk Campus

Er. Shyam Dahal is [illegible] Department of Computer Engineering, Kathmandu Engin[illegible] He has been teaching Engineering Electromagnetics [illegible] Electronics Engineering to undergraduate students since [illegible] also the author of two books: *Insights on Basic Electronics Engineering* (2016 AD) and *Insights on Engineering Electromagnetics* (2[illegible]D).

9 789937 028103

Price Rs. 320/-

SYSTEM INCEPTION