A Notebook on Microprocessor System

Book · August 2012

CITATIONS
0

READS
6,356

1 author:

Shree Krishna Khadka
Tribhuvan University
34 PUBLICATIONS **9** CITATIONS

SEE PROFILE

A Notebook on

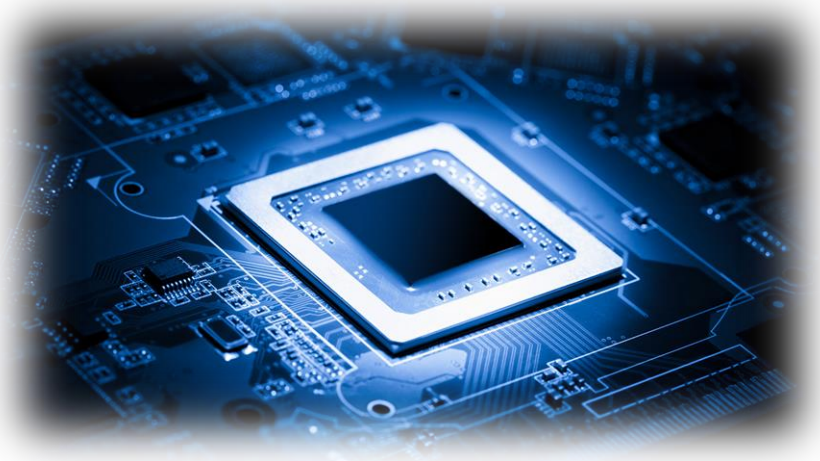
Microprocessor System

(With 8085 & 8086 Programming)

Prepared By

Er. Shree Krishna Khadka

teaching.official @gmail.com



Inside

Content

Chapter: 1 Introduction

Chapter: 2 Microprocessor Instructions

Chapter: 3 Assembly Language Programming

1 CHAPTER

Introduction

Contents:

- Introduction to Development of Computer
- Microprocessor: Introduction
- Microprocessor Based System
- Microprocessor Architecture
- 8085 & 8086 CPU

INTRODUCTION TO DEVELOPMENT OF A COMPUTER

Historical Background

Computer is the most efficient and versatile electronic machine, basically developed by the enhancement of calculator. **Charles Babbage**, the father of computer developed the mechanical computer named **"Difference Engine"** and **"Analytical Engine"**.

- The Difference engine could perform the arithmetic operations like add and subtract. But it could perform only single algorithm. Its output system was compatible to write on medium such as punch card and early optical disk.
- The Analytical engine provides more advanced features in comparison to Difference engine. It consists of mainly four components.

  - **The store (Memory)**
  - **The mill (Computation Unit)**
  - **Input section**
  - **Output section**

The store is used to hold the variables and results. The mill accepts operands from the store and performs addition, subtraction, multiplication or division and returns a result back to store. It reads the instruction and data from punched cards operates on it and have the results back to the store or writes to the punch card.

The evolution of vacuum tubes led to the development of computer of new era. The world first general purpose computer was **ENIAC (Electronic Numerical Integrator and Calculator)**. It was constructed in **1940** under the supervision of **John Mauchly** and **John Presper Eckert** at the **Universities of Pennsylvania**. The **ENIAC** built by using vacuum tubes was enormous in size and consumed very high power. However it was faster than mechanical computers. The **ENIAC** was a decimal machine in which numbers were decimal number system and the operation was performed in the same numbering system. Its memory consisted of **20** accumulators each capable of holding **10** digit decimal numbers. The major drawback of **ENIAC** was that it had to be programmed manually by setting switches and plugging and unplugging cables.

Automated Calculator

An automated calculator is a data processing device that carries out logic and arithmetic operation but has limited capability for the user. The calculator accepts data from a small keyboard one digit at a time performs the arithmetic and logic calculation and shows the result on the visual display i.e. **LCD** or **LED**. The calculators programs are stored in **ROM** while the data that the user enters in **RAM**.

FEATURES:

- Able to interface easily with keyboards and displays.
- Able to handle decimal digit as units.
- Able to execute the standard program store in ROM.
- Extendible so that mathematical function such as percentage, square root, trigonometric etc. can be easily executed.
- Low cost, small in size and consumes low power.
- Flexible and can be used in any application like engineering, business etc.

Stored Program Computer

Store program computer is one of the most important breakthroughs in the entire field of information technology. Stored program computer are those where the program (instruction) are stored in same memory locations and brought into processor and decoded to perform the intended operations. The idea was first glimpsed by **Ada Lovelace & Charles Babbage** in **1840**'s; later in **1946 Von Neumann** and his colleagues began the design of new stored program computer called **IAS** (**Institute of Advance Studies**) in **Princeton, NJ, USA**.

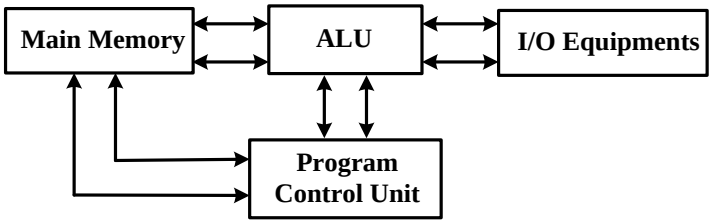


Fig: Structure of IAS Computer

The Von-Neumann Architecture

Since, the task of entering and altering the programs for the **ENIAC** was extremely tedious. The programming process could be facilitated if the program could be facilitated if the program could be represented in a form suitable for storing in memory alongside the data. Then a computer could get its instructions by reading them from the memory and a program could be set or altered by setting the values of a portion of memory. This approach was based on "**Stored Program Concept"** and was first adopted by **John Von Neumann**. The general structure of Von Neumann's machine is shown below.

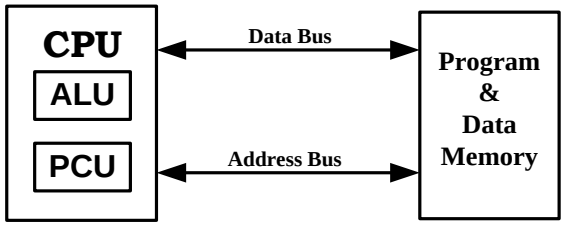


Fig: Von Neumann Architecture

The main memory is used to store both data and instructions. The arithmetic logic unit is capable of performing arithmetic and logical operations on binary data. The program control unit interprets the instructions in memory and causes them to be executed. The I/O unit gets operated from the control unit.

Harvard Architecture

This architecture based computer consists of separate memory spaces for the program and data. Each memory space has its own address and data buses. As a result of this both instruction and data can be fetched from memory concurrently. This feature provides a significant processing speed improvement when compared with microprocessor devices based on Von-Neumann type architecture.

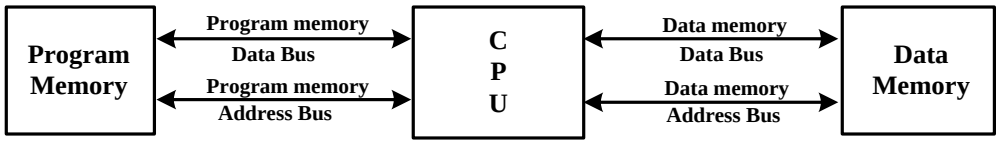


Fig: Harvard Architecture

Von Neumann Architecture Vs Harvard Architecture

Von-Neumann Architecture	SN	Harvard Architecture
It is designed by John Von Neumann.	1.	It is designed by Harvard.
The Von Neumann architecture uses the single data bus for instruction and data	2.	The structure where different data bus is used for data and instruction are referred as Harvard architecture.
In this architecture there is only one shared memory for instruction (program) and data.	3.	In this architecture data and instruction are stored in different memories.
In this architecture on bus is used so data cannot be fetched until the code fetch is completed.	4.	The parallel bus used for code words and data words allows instruction and data to be fetched at the same time.
This architecture is referred as **"Stored Program Computer".**	5.	This architecture is also referred as **"Superscalar"** as it can perform more than one instruction per cycle.
Instruction and data have to be fetched in sequential order, limiting the operational bandwidth.	6.	Different program and data bus width is possible, allowing program and data to be better optimized to the architectural requirements.
Its design is simpler.	7.	Its design is complex than Von Neumann's architecture.
Example: **8085, 8086.**	8.	Example: **TMS320CXX** from Texas instrumentation

MICROPROCESSOR

Introduction

A microprocessor is a multipurpose, programmable, clock driven, register based electronic device that reads binary information from a storage device called memory, accepts binary data as input and processes data according to those instruction and provides results as output. Thus a microprocessor is an **LSI** chip that has capability to control, process and decision making similar to that of human mind.

A typical programmable machine can be represented with four components as show in figure below. They work together or interact with each other to perform a given task; thus they comprise a system.

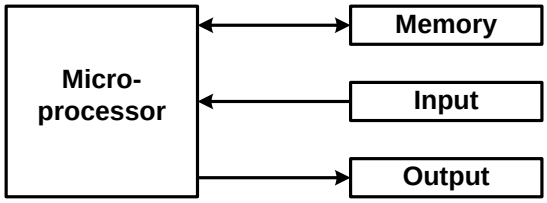


Fig: A Programmable Machine

This system may be simple or sophisticated, depending on its applications and recognized by various names depending upon the purpose for which it is designed. The microprocessor applications are classified primarily in two categories:

- **Reprogrammable Systems.**
- **Embedded Systems**.

In reprogrammable systems, such as microcomputers, the microprocessor is used for computing and data processing. These systems include general purpose microprocessors capable of handling large data, mass storage devices and peripherals. A personal computer is a typical illustration.

In embedded systems, the microprocessor is a part of a final product and is not available for reprogramming to the end user. A copying machine is a typical example of an embedded system.

Microcomputer

The microprocessor when embedded in a larger system can be a standalone unit controlling processes or it can function as a **CPU** of a computer is called microcomputer. So, a computer designed using a microprocessor as its **CPU** is the microcomputer.

Evolution of Microprocessor

The microprocessor is essentially the product of research and development in two allied fields: digital computer and semiconductor technology. Recognition as the first digital computer is given to a machine called **ENIAC**, which was designed in **1940**. After the development of transistor in **1948** and development of integrated circuit, mainframe computer was developed in **1960**'s and mainframe followed it.

The first microprocessor Intel **4004** was announced in **1971** by **Intel Corporation**. The **Intel 4004** was the first chip to contain the entire component on a **CPU** in a single chip. It was a **4**-bit microprocessor which was shortly replaced by **Intel 4040**. At the same time **Rockwell International's PPS4** and **Toshiba's T3 472** were developed.

The first **8**-bit microprocessor **8008** was introduced again by **Intel** in **1972** that was followed by a better version the **8080** from the same company. Some other **8**-bit microprocessor like **Motorola's M6800**, **National Semiconductors SC/MP**, **Zilog Corporation's Z800**, **Fair Child's F8** and **Intel 8005**. Since then **Intel & Motorola** are leading companies for developing microprocessor. After that **Intel 8086, 80186, 80286, 80386, 80486 & Pentium Series** whereas **Motorola 6802, 68000** etc. were developed.

Microprocessor as a Programmable Device

Microprocessor is a multipurpose electronic device used to perform various sophisticated function as well as simple task that has been programmed during the manufacturing processes. A programmer can select appropriate instructions and ask the microprocessor to perform various tasks on a given set of data.

Microprocessor has a several set of instruction embedded in its memory to perform the various task intended by the programmer. Thus the fact that the microprocessor is programmable means it can be instructed to perform given tasks within its capability.

Microprocessor as a CPU (MPU)

In a traditional computer we can view microprocessor as a primary component of a computer. It consists of various elements like **ALU**, Control Unit and various registers to hold data and to operate on them. Figure below shows microprocessor used as a **CPU**. In the late **1960**'s, using the advance feature of integrated circuit technology various discrete elements were added on a single chip that can be used as a **CPU**.

A computer with a microprocessor as a **CPU** is called microcomputer. The microprocessor is thus sometimes referred as microprocessor unit **(MPU)** or microcontroller unit **(MCU)**. A microcontroller is essentially an entire computer on a single chip that includes additional devices such as analog to digital converter, serial **I/O** and timers.

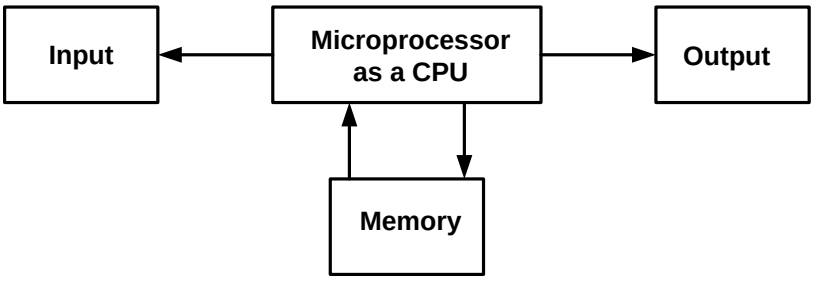


Fig: Block Diagram of Computer with Microprocessor as a CPU

Microprocessor Based System (Microcomputer System)

A microcomputer based system or microcomputer system consists of different units working together to perform various operation in correct order. Basically the microcomputer system includes four components. They are:

- **Microprocessor**
- **Memory**
- **Input unit**
- **Output unit**

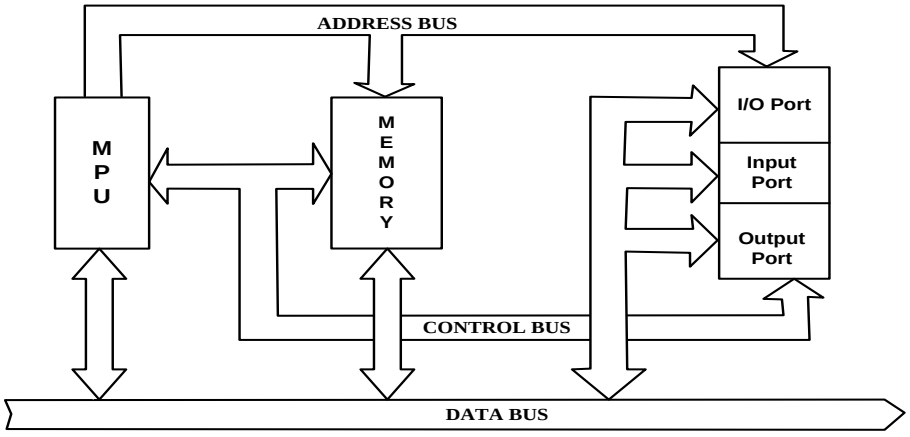


Fig: Microprocessor Based System

The jobs that a microcomputer system performs can be listed as follows.

- To read instruction and data.
- Store inputted data and instruction.
- Executing the instruction i.e. processing the data.
- Storing the result of processing.
- To output the result of processing.

All these operations part of the communication and processing of data between the microprocessor and peripheral devices (including memory). To communicate with the peripheral or a memory, the microprocessor needs to perform the following steps.

Step 1: Identify the peripheral or memory location with its address.
Step 2: Transfer binary information (data and instruction) using buses.
Step 3: Provide timing and synchronization signals.

In **8085** microprocessor all these functions are performed by using three sets of communication lines called Buses. They are **Address Bus, Data Bus** and **Control Bus** as shown in figure. In other word, it can also be called as **"Three Bus Architecture System".**

Address Bus

In a microcomputer system each peripheral or memory location is identified by a binary number called an address and the address bus is used to carry the address. The address bus is unidirectional i.e. bit flows in one direction from the microprocessor to memory or peripheral devices. The microprocessor uses the address bus to perform the first function mentioned in step one. The width of the address bus clearly determines the maximum possible memory capacity of the system.

In Intel **8085,** there is **16**-bit address, so it has capability of addressing memory up to **64 KB**.

Data Bus

The data bus is a group of lines used to carry the data between various components of microcomputer system. Data bus is bidirectional i.e. data flows in both direction between the microprocessor and memory and peripheral devices. The microprocessor uses the data bus to perform the second function mentioned in step **2**.

In Intel **8085**, there is **8**-bit data bus so it can process **8**-bit of data at a time. That's why it is called **8**-bit microprocessor.

Control Bus

The control lines are somewhat different from address bus and data bus. The control bus is comprised of various single lines that carry timing or synchronization signals. The microprocessor uses the control bus to perform the third function mentioned in step **3**.

The microprocessor generates specific control signals for every operation such as memory read\write or **I\O** read\write. Some of the control signals are:

- **Memory Read**: It causes data from the memory to be placed on the data bus.
- **Memory write**: It causes the data on the data bus to be loaded into the memory.
- **I\O Read**: It causes data from the addressed **I\O** port to be placed on data bus.
- **I\O Write**: It causes the data on the data bus to be outputted to the addressed **I\O** port.
- **Interrupt Request**: Indicate that interrupt has been pending.

MICROPROCESSOR ARCHITECTURE

- A microprocessor has general purpose and special purpose registers. The general purpose registers are use for storing intermediate data and operands. The special purpose registers are the Accumulator, Program Counter and Stack Pointer.

- The Accumulator is used to store operands and/or to store the result of operations.

- The program counter and stack pointer registers are used to keep track of the memory location from where the next instruction is to be fetched and location of the most recent stack entry, respectively.

- The **ALU** in the microprocessor can add and operand and operand in the accumulator to an operand in another register or add and operand in the accumulator with data specified by the program. It can also carryout the Boolean operations and shift operations.

- The control circuitry in the microprocessor is responsible for issuing signal to control functions of various components within as well as external to the microprocessor as per the instruction executed. All the operations are coordinated by clock signals.

The 8085A CPU

The Intel **8085** is a general purpose **8-bit** microprocessor capable of addressing up to **64 Kb** of memory. The internal logic design of a microprocessor is called architecture which determines how and where various operations are performed by the microprocessor. The main components of **8085** is shown in the functional block diagram and described below.

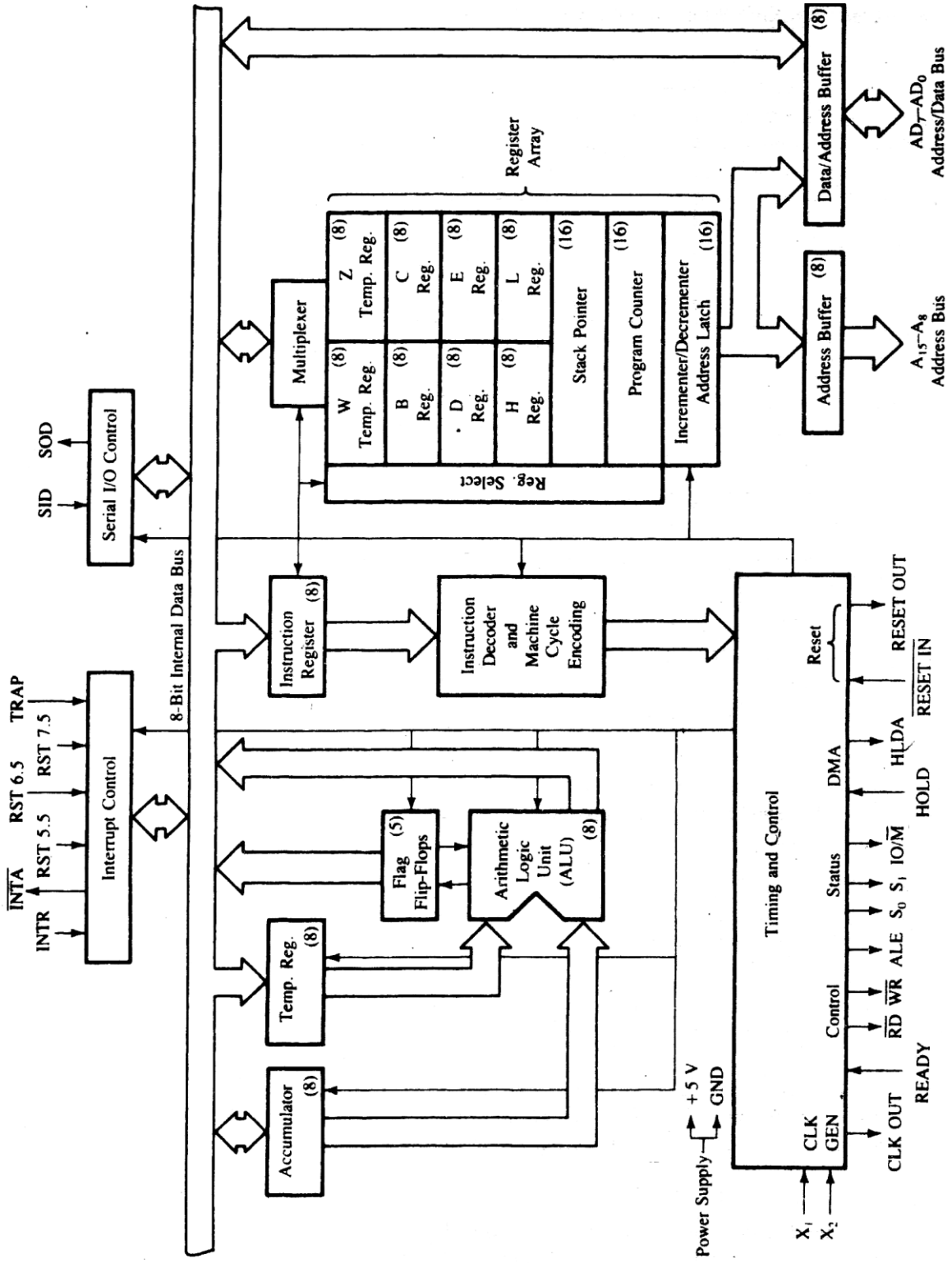


Fig: The 8085A microprocessor functional block diagram

DESCRIPTION OF 8085A BLOCK DIAGRAM

1. Register

The **8085A** has both **8-bit** and **16-bit** registers. It has **8-bit** registers: **B, C, D, E, H, L, A (Accumulator) & F (Flag) and 16-bit** registers: **PC (Program Counter), SP (Stack Pointer) & IR (Instruction Register).** The **PC** and **SP** are addressable while **IR** is not. The registers of **8085A** can be classified as:

- **General Purpose Registers**
- **Special Purpose Register**

1.1 GENERAL PURPOSE REGISTERS

B, C, D, E, H and **L** are **8-bit** general purpose registers and can be used either single or as **16-bit** register pairs **BC, DE** and **HL**. When used in the register pair mode, the high order byte resides the first register and the low order byte in the second.

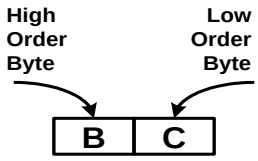


Fig: BC Register Pair

Whereas register pair **HL** can be used as two independent **8-bit** registers, functions as a data pointer. It can hold memory addresses that are referred to a number of instructions, which use '**Register Indirect Addressing'.** Hence, general purpose registers are used to temporarily store operands or intermediate data in calculations.

1.2 SPECIAL PURPOSE REGISTERS

- **Accumulator**

The accumulator (register A) is an **8-bit** register accessible to the programmer. This register is used to store **8-bit** data and to perform arithmetic and logical operations. It is part of the arithmetic and logical unit because one of the operand in the **ALU** operation is an accumulator. The result of the operations is also stored in the accumulator. It is also to access the data from the input output ports. When the data is read from input port, it is first moved to the accumulator. Similarly, when data is send to the output port, it must be first placed in the accumulator.

- **Flags**

Registers containing of five flip-flops which are set or reset according to the arithmetic and logical operation and result in accumulator is known as flag register and each flip-flops are flags. These flags are **Carry (CY), Zero (Z), Sign (S), Parity (P)** and **Auxiliary carry (AC)**. The microprocessor uses these flags to test data condition and for decision making processes. These are described below.

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

The Carry Flag (CY): If an arithmetic operation results in a carry, the carry flag is set otherwise it is reset. The carry flag also serves as a borrow flag for subtraction.

The Zero Flag (Z): The zero flag is set if the **ALU** operation results in zero otherwise it is reset. This flag is modified by the results in the accumulator as well as in other registers.

The Auxiliary Carry (AC): In an arithmetic operation when a carry is generated by digit D_3 and passed onto the digit D_4, the **AC** flag is set otherwise it is reset. This flag is used in **BCD** arithmetic.

The Parity Flag (P): Since **8085** operates in even parity, parity flag is set when the result of arithmetic and logical operation has even number of **1**'s. Otherwise it is reset.

The Sign Flag (S): If the bit D_7 or **MSB** (most significant bit) of the result is **1**, the sign flag is set. This flag is used with signed number. If D_7 is **1**, the number will be viewed as negative; if it is **0**, then it will be viewed as positive.

○ **Program Counter:**

This is 16-bit register and used by microprocessor to sequence the execution of instructions. A computer program consists of sequence of coded instruction. These instructions are stored sequentially in the memory location. As the **8085** begins to execute an instruction, the memory address of the first instruction to be executed is placed in program counter. So it contains the memory address from which the next byte is to be fetched. Hence the microprocessor uses these registers to sequences the execution of instruction.

○ **Stack Pointer**

A stack is an area of memory set aside for the purpose of storing data by operation known as stacking. As data is stored and retrieved, its location is specified by a **16-**bit register known as the stack pointer. The beginning of the stack is defined by loading a **16-bit** address on the stack pointer.

The stack pointer register holds the address of the last byte written into the stack. The stack pointer is decremented automatically each time data is pushed onto the stack and is incremented automatically when the data is popped off the stack.

○ **Temporary Register**

The temporary register is an **8-bit** register which is not accessible to the programmer. These registers are internally used by the microprocessor to hold **8-bit** data during the execution of some instruction as a '**Memory Reference'** instruction such as reading data from a given address of memory or writing data to a given address of memory.

○ **Instruction Register & Instruction Decoder**

When an instruction is fetched from a memory it is loaded in the memory register. The **IR** receives the operation codes of instructions from the internal data bus and passes it to the instruction decoder and machine cycle encoding circuit. The instruction decoder decodes the instruction so that microprocessor knows which type of operation is to be performed before executing it. The output of the instruction decoder is fed to the control and timing unit. Then control and timing unit then generates the necessary control and timing signals. This register is not accessible to the programmer.

2. Arithmetic & Logic Unit (ALU)

The **ALU** performs the arithmetic operations like add, subtract etc and logical operations like **AND, OR, XOR** etc. Besides this **ALU** carries out left and right shifting of the **8-bit** data stored in accumulator. When **ALU** perform arithmetic and logical operations one of the data is stored in accumulator and the result of the operations is also referred to the accumulator itself. The flags are affected by the arithmetic and logical operations in **ALU**.

3. Timing & Control Unit

This unit synchronizes all the microprocessor operations with the clock and generates the control signals necessary for communications between the microprocessor and peripherals. The **RD** and **WR** signals are sync pulses indicating the availability of data on the data bus.

The control circuitry is responsible for all the operation. All the operations are driven by the clock signal. All operations are synchronously timed according to the clock signals. This includes the following stages:

- **Instruction Fetching Operations**
- **Instruction Decoding and**
- **Instruction Execution.**

The control circuitry is also responsible for interaction with signals external to the **CPU** e.g. receiving signals like interrupt request and wait request.

DESCRIPTION OF 8085A PIN DIAGRAM

Instructions can be categorized according to their method of addressing the hardware registers and/or memory. Figure below shows the logic pin out of **8085** microprocessor. All the signals can be classified into six groups.

1. **Address Bus**
2. **Multiplexed Address/Data Bus**
3. **Control & Status Signals**
4. **Power Supply & Clock Frequency**
5. **External Initiated Signals Including Interrupts**
6. **Serial Input/Output Ports**

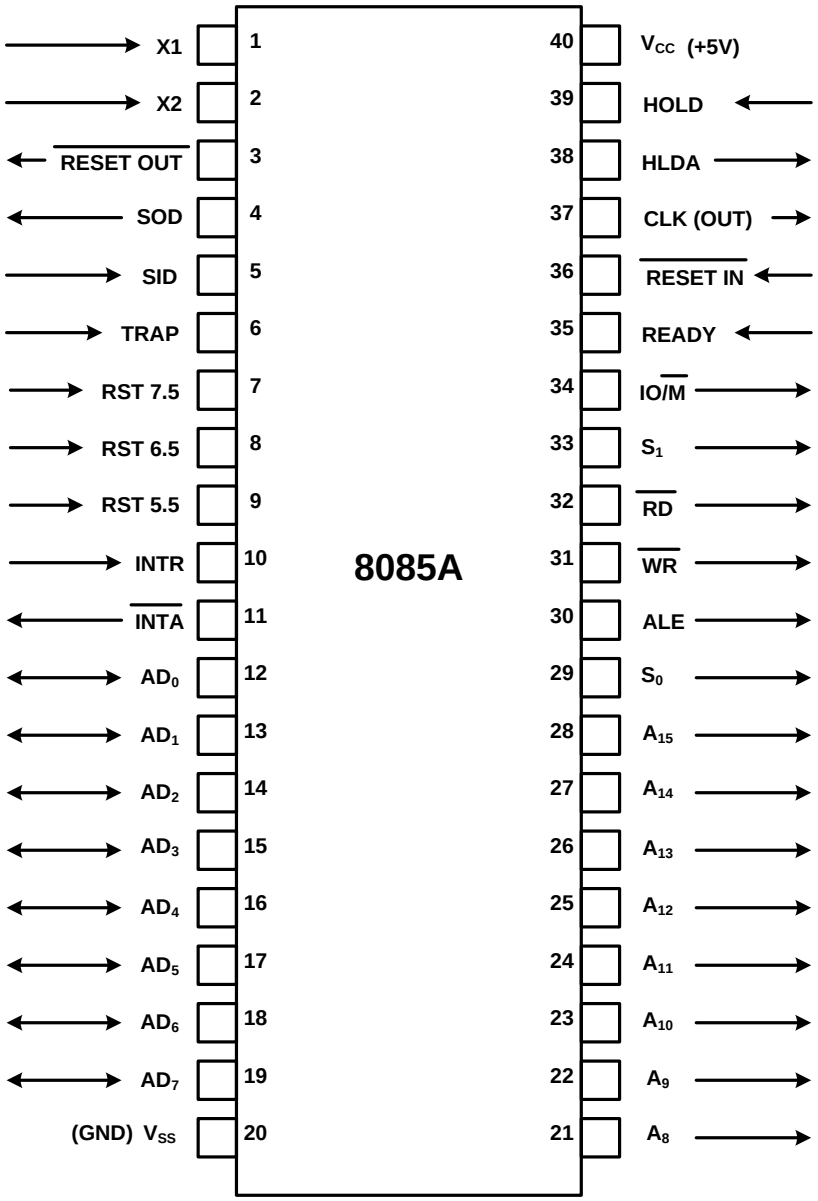


FIG: 8085A PIN DIAGRAM

1. Address Bus

The **8085** has eight signal lines **(A_{15}-A_8),** which are unidirectional and used as a higher order address bus.

2. Multiplexed Address/Data Bus

The signal lines **AD_7-AD_0** are bidirectional. They serve for a dual purpose. They are used as a low order address bus in earlier part of the cycle and as a data bus in the later part of the cycle, while executing an instruction. This process is known as multiplexing the bus.

3. Control & Status Signal

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

- **ALE - ADDRESS LATCH ENABLE**
  This is a positive going pulse generated every time the **8085** begins an operation (machine cycle) indicating that the bits on **AD_7-AD_0** are address bits. This signal is used primarily to latch the low order address bus from the multiplexed bus and generate a separate set of eight address lines **A_7-A_0**.

- $\overline{RD}$ **- READ**
  This is an active low Read control signal that indicates 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.

- $IO/\overline{M}$ **- INPUT/OUTPUT & MEMORY**
  This is a status signal used to differentiate between **I\O** and memory operations. When it is high it indicates an **I\O** operation and when it is low it indicates a memory operation. This signal is combined with $\overline{RD}$ (Read) and $\overline{WR}$ (Write) to generate **I\O** and memory control signals.

- **S_1 & S_0 - SWITCH STATUS**
  These status signals similar to $IO/\overline{M}$ can identify various operations but they are rarely used in small system.

Machine Cycle	Status			
	$IO/\overline{M}$	S_1	S_0	Control signal
Op-code Fetch	0	1	1	$\overline{RD}$=0
Memory Read	0	1	0	$\overline{RD}$=**0**
Memory Write	0	0	1	$\overline{WR}$=**0**
I/O Read	1	1	0	$\overline{RD}$=**0**
I/O Write	1	0	1	$\overline{WR}$=**0**
Interrupt Acknowledge	1	1	1	$\overline{INTA}$=**0**
Halt	Z	0	0	
Hold	Z	×	×	$\overline{RD}$=**Z,** $\overline{WR}$=**Z,** $\overline{INTA}$=**1**
Reset	Z	×	×	

Table: 8085 Machine Cycle and Control signals
Note: **Z = Tristate** (High Impedance), × = Unspecified

4. Power Supply & Clock Frequency

The power supply and frequency signals are as follows:

- **V_{CC}: +5v** Power Supply
- **V_{SS}**: Ground Reference
- **X_1, X_2**: A crystal (or **RC, LC** Network) is connected at these two pins. The frequency is internally divided by two; therefore, to operate a system at **3MHz**, the crystal should have a frequency of **6MHz**.
- **CLK (OUT):** Clock Output: This signal can be used as system clock for other devices.

5. Externally Initiated Signals, Including Interrupts

Externally initiated signals are described below.

- **INTR (INPUT):** This pin is interrupt request used as a general purpose interrupt.
- $\overline{\text{INTA}}$ **(OUTPUT):** This is an interrupt acknowledge pin used to acknowledge an interrupt.
- **RESTART INTERRUPT:** There are three restart interrupt; **RST7.5, RST6.5** and **RST5.5** in 8085. These are vectored interrupt and transfer the program control to specific memory locations. They have higher priorities than the **INTR** interrupt. Among these three the priority order is as **7.5**, **6.5** and **5.5**.
- **TRAP (INPUT):** This is a non mask able interrupt and has a highest priority.
- **HLDA (OUTPUT):** This hold acknowledge signal. 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 clocks until it goes high.
- **$\overline{\text{RESET IN}}$:** When the signal on this pin goes low, the program counter is set to zero, the buses are tri-stated and the microprocessor is reset.

- **RESET OUT:** This signal indicates that the microprocessor is being reset. This signal can also be used to reset other devices.

6. Serial Input/Output Ports

The **8085** has two signals to implement the serial transmission; **SID (Serial Input Data)** and **SOD (Serial Output Data).**

APPLICATION OF 8085 MICROPROCESSOR

- Directly compatible with **TTL IC**'s leading to low cost of integrating the complete microprocessor based system.
- **8085**- based microprocessor
- Performing the high speed arithmetic and logic operations.

8085 Microprocessor Control Temperature System (MCTS)

Based on the concepts we discussed in the general instrumentation system, we can examine a typical microprocessor controlled temperature system. This system is expected to read the temperature in a room, display the temperature ate a liquid crystal display (**LCD**) panel, turn on fan if the temperature is above a set point, and turn on a heater if the temperature is below a set point.

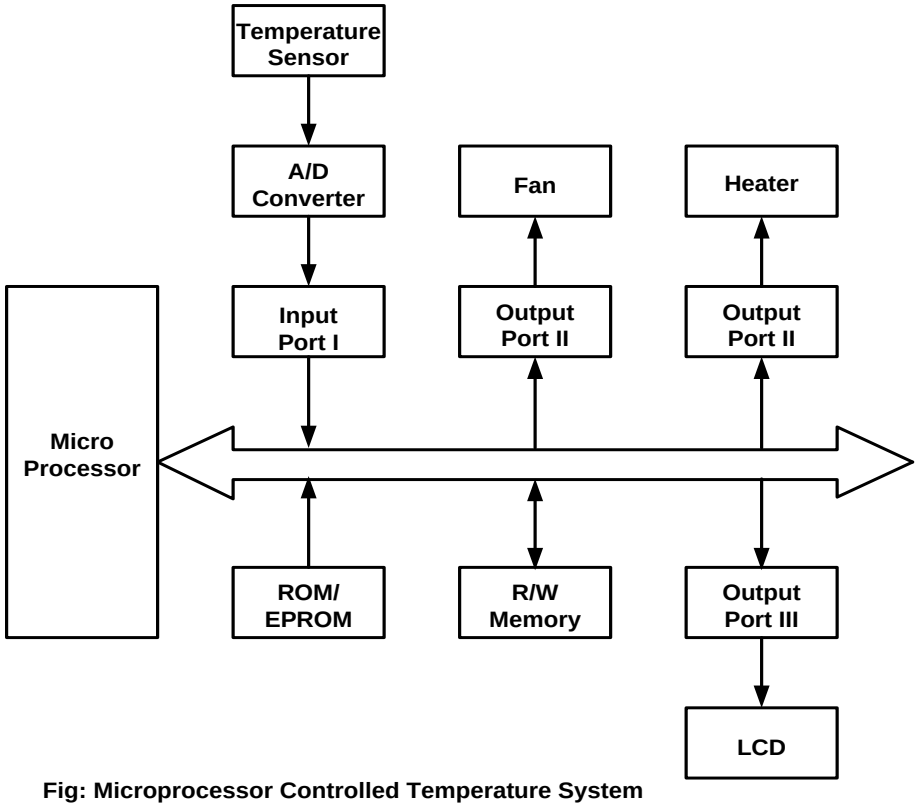


Fig: Microprocessor Controlled Temperature System

Microprocessor:

The above figure shows a processor with a system bus. The processor will read the binary instructions from memory and execute those instructions continuously. It will read the temperature, display it at the **LCD** display panel, and turn on/off the fan and the heater based on the temperature.

Memory:

The system includes two types of memory. **ROM** will be used to store the program, called the monitor program that is responsible for providing the necessary instructions to the processor to monitor the system. This will be a permanent program stored in **ROM** and will not be altered. The **R/W** memory is needed for temporary storage of data.

Input:

In this system, we need a device that can translate temperature into an equivalent electrical signal; a device that translates one form of energy into another form is called a transducer. On these days temperature sensors are available as integrated circuits. A temperature sensor is a three-terminal semiconductor electronic device that generates a voltage signal that is proportional to the temperature However, this is an analog signal and since the processor is capable of handling only binary bits. Therefore, this signal must be converted into digital bits. The analog-to-digital converter performs that function. It is an electronic semiconductor chip that converts an input analog signal into the equivalent eight binary output signals. In microprocessor-based systems, devices that provide binary inputs are connected to the processor using devices such as buffers called input ports. In our system **A/D** converter is an input port, and it will be assigned a binary number called an address. The microprocessor reads this digital signal from the input port.

Output:

The above figure shows three output devices: fan, heater and liquid crystal display **(LCD).** These devices are connected to the processor using latches called output ports.

Fan: This is an output device, identified as **Port 1** that is turned on by the processor when the temperature reaches a set higher limit.
Heater: This is also an output device, identified as **Port 2** that is turned on by the processor when the temperature reaches a set lower limit.
Liquid Crystal Display (LCD): This display is made of crystal material placed between two plates in the form of a dot matrix or segments. It can display letters, decimal digits, or graphic characters. The **LCD** in above figure will be used to display temperatures.

System Software (Programs):

The program that runs the system is called monitor program or system software. Generally, the entire program is divided into subtasks and written as independent modules, and it is stored in **ROM** or **EPROM**. When the system is reset, the microprocessor reads the binary command from the first memory location of **ROM** and continues in sequence to execute the program.

INTEL 8086 CPU

The **8086** is a far more powerful **16-bit** machine with **1 MB** memory capacity, in addition to wider data path and larger registers. It has a feature of instruction pipelining. For this purpose it consists of an instruction cache that pre-fetches a few instructions before they are executed. It has **16- bit** wide data bus and can operate on **16-bit** of data from memory and ports.

Description of 8086 Block Diagram

This **16-bit 8086** microprocessor is divided internally into two functional units

- **Bus Interface Unit (BIU)**
- **The Execution Unit (EU)**

Both these units are designed to work simultaneously. The **BIU** is designed to function as the external interface of the processor. The **BIU** generates address and sends them on the address bus and then reads (or writes) data from memory (and ports) through the data bus. Therefore **BIU** acts an interface of execution unit to the outside world.

The **EU** handles all controlling functions such as reading and writing the data, decoding the instruction and executing the instruction. Block diagram of internal architecture of **8086** microprocessor is shown and described below.

1. Bus Interface Unit

The Bus Interface Unit can be divided into the following sub-units.

1. **INSTRUCTION QUEUE**:

   The Instruction Queue is a **6-byte First-In-First-Out (FIFO)** register which fetches **6 bytes** of instruction from memory ahead of time. It helps to speed up execution of program by overlapping instruction fetch with execution and this mechanism is known as pipelining.

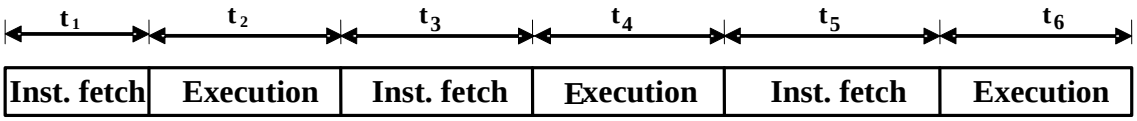
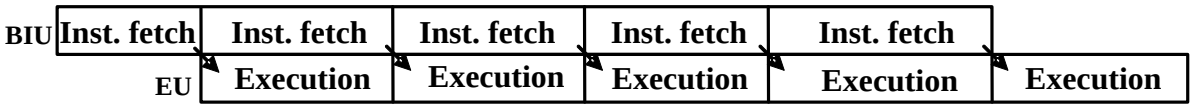


Fig: Sequential Execution 8085



Pipelined Execution 8086

2. **INSTRUCTION POINTER**:

The instruction pointer stores the offset of the physical address in the segment of memory as defined by the segment registers.

3. **SEGMENT REGISTERS**:

The **8086** has memory segmentation. Memory segmentation allows the **8086** microprocessor to be able to access four **64 Kbytes** segments within its **1 MB** memory range at any given time. These four segments registers are:

- **Code Segment Register (CS):** All the instruction are located in memory (Code Segment) pointed by the **16-bit** code segment register and a **16-bit** offset contained in the instruction pointer (**IP**). The **BIU** computes **20-bit** address called physical address by using the content of **CS** and **IP** register.

- **Data Segment Register (DS):** The data segment register points to the current data segment. Operands for the instruction are fetched from this segment.

- **Stack Segment Register (SS):** The stack segment register points to the current stack.

- **Extra Segment Register (ES):** The extra segment register points to the extra segment in which data (in excess of **64 K** pointed to the **DS**) is used. String instruction always uses extra segment.

4. **BUS CONTROL AND ADDRESS GENERATOR**:

This unit handles the external bus as well as calculates the effective address from where the instruction has to be fetched.
Example: **Segment register × 1011 + offset value = Physical Address**

2. The Execution Unit

The execution unit of **8086** may be divided into the following subunits.

1. **GENERAL PURPOSE REGISTERS**:

The **8086** has four **16-bit** general purpose registers **AX, BX, CX** and **DX.** These **16-bit** registers can also be used as **8-bit** registers as:

AX - AH and **AL** (**Accumulator Register**)
BX - BH and **BL** (**Base Register**)
CX - CH and **CL** (**Counter Register**)
DX - DH and **DL** (**Data Register**)

2. Special Purpose Registers:

- **Pointer Registers**:

The two pointer registers; Stack Pointer (**SP**) and Base Pointer (**BP**) are used to access data in the stack segment. These registers are used to store **16-bit** data but not **8-bit** data.

- **Index Registers**:

The two index register Source Index (**SI**) and Destination Index (**DI**) are used in indexed addressing mode.

- **Flag Register**:

The **8086** has a **16-bit** flag register which contains nine active flags. These active flags can be divided into two types.

×	×	×	×	**OF**	**DF**	**IF**	**TF**	**SF**	**ZF**	×	**AF**	×	**PF**	×	**CF**

Fig: 8086 flag register

- **Conditional Flags**:

The flag register in the **EU**, holds the conditional/status flags. These flags are set or reset on the basis of result generated from the **ALU**. Out of nine, the **8086** has six status flags. They are:

1. Auxiliary Carry Flag-**AF:** It is set if a carry from the lower nibble into the higher nibble or borrow from higher nibble into the lower nibble of the low or high **8-bits** of **16-bit** number. This flag is used by **BCD** arithmetic instruction; otherwise it remains reset.
2. Carry Flag-**CF:** It is set if a carry from addition or borrow from subtraction is produced otherwise remains reset.
3. Overflow Flag-**OF**: It is set if arithmetic overflows, i.e. if the size of the result exceeds the capacity of the destination location.
4. Sign Flag-**SF**: It is set if the most significant bit of the result is **1**; otherwise it is zero.
5. Parity Flag-**PF**: It is set if the result has even parity and remains zero for odd parity of the result.
6. Zero Flag-**ZF**: It is set if the result is **0**. Zero flag is **0** for non zero result.

- **Control Flags**:

The **8086** contains three control flags which are used for controlling the microprocessor. The control bits in the flag register can be set or cleared by the programmer.

1. Trap Flag-**TF**: It is used for single stepping through a program.
2. Interrupt Flag-**IF**: When the interrupt flag is set a program can be interrupted for other more important operations.
3. Direction Flag-**DF**: It is used with string operation.

3. Arithmetic & Logic Unit (ALU)

The arithmetic and logic unit of **8086** microprocessor can perform various arithmetic and logical operation on **16-bit** data.

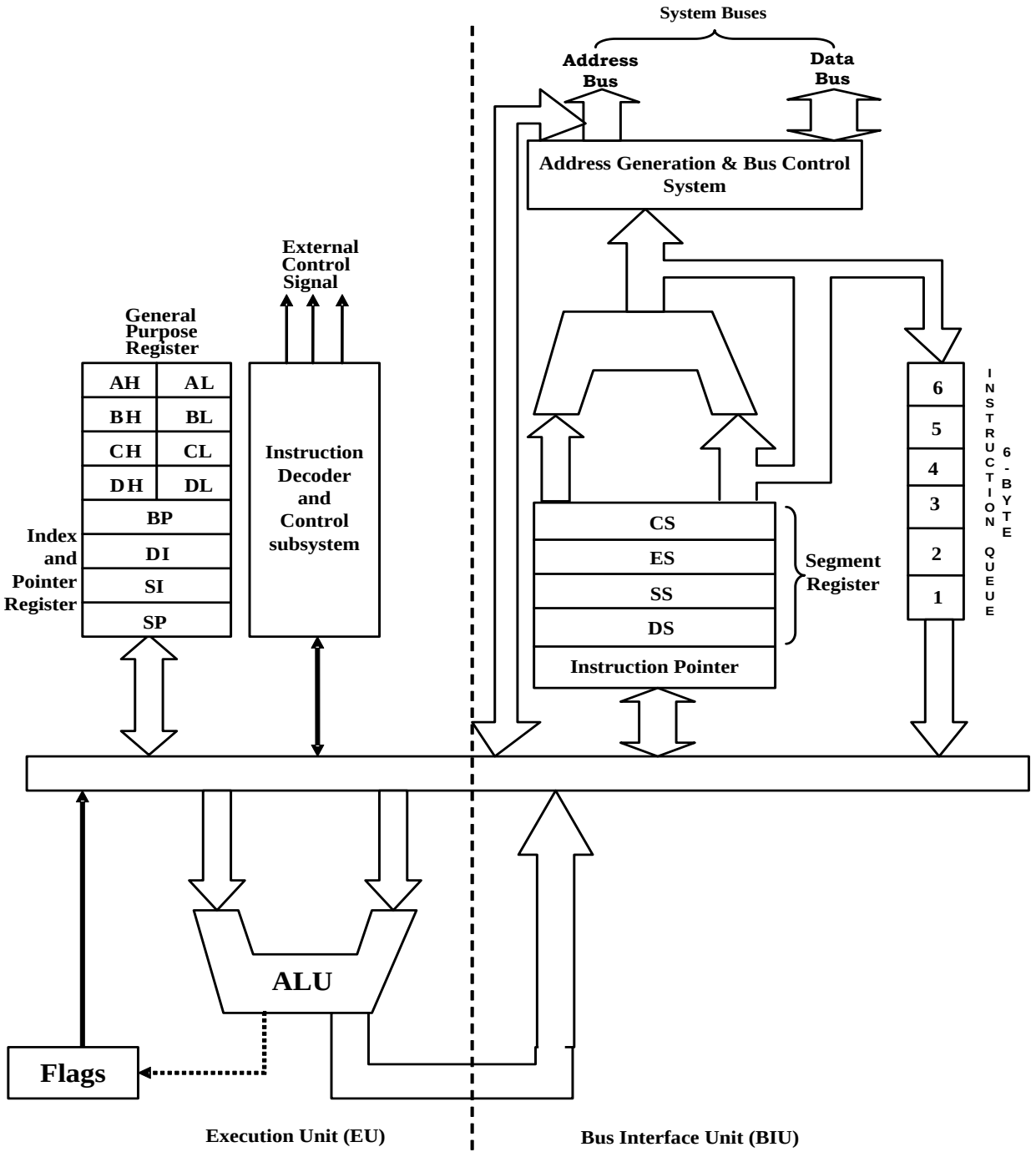


Fig: Block Diagram of 8086 Microprocessor

DESCRIPTION OF INTEL 8086 PIN DIAGRAM

The **Intel 8086** has two modes of operations:

- **Minimum Mode and**
- **Maximum Mode**

In microcomputer system, where only one **8086**-microprocessor works as **CPU,** it operates in minimum mode of operation. But in multiprocessor system, the **8086**-microprocessor works in maximum mode of operation.

The status of the pin **MN/$\overline{MX}$** decides the operating mode of **8086.** When the signal in this pin goes high, **8086** operates in minimum mode. When the signal goes low, it operates in maximum mode.

Minimum Mode of Operation

- **AD_{15}-AD_0 (Bidirectional):** These lines are time-multiplexed address/data bus. During **T_1** clock cycle, it carries lower order **16-bits** of address and during **T_2, T_3 & T_4**; it carries **16-bit** data (During Op-code Fetch).
- **A_{19}/S_6-A_{16}/S_3 (Unidirectional):** These lines are also time-multiplexed lines. They carry high order **4-bits** of memory address and during **T_2, T_3 & T_4**; these lines carries status signals.

Encoding of S_3 and S_4			S_5 and S_6
S_4	S_3	Segment Register Employed	
0	0	ES	The S_5 status signal is the interrupt enable flag bit whereas bit S_6 is always at logic **0**.
0	1	SS	
1	0	CS or None	
1	1	DS	

- **$\overline{BHE}$ (Bus/Bank High Enable)/S_7:** The **8086** has a **20-bit** address bus and each address can store **8-bits** of data. So when the instruction is executed, the word is actually written into two consecutive memory locations. It will take two machine cycles, as it needs to write the data two times in memory. So to make it possible to read or write a word in one machine cycle, the memory for **8086** is set up as two '**Banks'.**

  One memory bank contains all the bytes which have even addresses. The data lines of this memory bank are connected to the lower eight data line **D_0-D_7**of **8086.**

  The memory bank contains all the bytes which have odd addresses. The data lines of this memory bank are connected to the higher eight data line **D_8-D_{15}** of **8086.**

  Address line **A_0** is used to enable the memory device in the lower bank. When **A_0** become low, it selects the lower bank memory devices as it indicates the even address. Address lines **A_1** through **A_{19}** are used to select the desired memory location in upper and lower bank of memory.

$\overline{\textbf{BHE}}$ signal is use to enable the upper memory bank. $\overline{\textbf{BHE}}$ in conjunction with $\mathbf{A_0}$ determines whether a byte or words will be transferred from/to memory location as show below.

$\overline{\textbf{BHE}}$	$\mathbf{A_0}$	**Word/Byte Access**
0	**0**	Whole word (from/to upper and lower bank)
0	**1**	Upper byte from/to odd address
1	**0**	Lower byte from/to even address
1	**1**	None (Both memory will be unselected)

During $\mathbf{T_2}$ and $\mathbf{T_4}$, the status line $\mathbf{S_7}$ is transmitted and it remains always high.

- **NMI:** Non Maskable Interrupt Request
- **INTR:** Interrupt Request
- **CLK:** Clock for **8086**
- **RESET:** It is a system reset and an active high signal. When it goes high the microprocessor goes into reset state.
- **READY:** It is an acknowledgement received from the addressed memory or **I/O** device. It is an active high signal. When it goes high, it indicates that the peripheral or memory device is ready to receive or transfer a data.
- **TEST:** This signal is used to test the status of math coprocessor **8087**. The **BUSY** pin of **8087** is connected to the **TEST** pin of **8086**.

  Sometimes the **8086** needs the result of some computations that **8087** is doing, before it could go to the next instruction of the program. When **8087** is busy, it sends the high signal on $\overline{\textbf{TEST}}$ pin, which cause **8086** to wait in an idle state. When **8087** complete computation, it makes **BUSY** signal low, thereby making $\overline{\textbf{TEST}}$ signal low. When $\overline{\textbf{TEST}}$ signal become low the **8086** goes to the next instruction.
- $\overline{\textbf{INTA}}$ **(Interrupt Acknowledge):** On receiving interrupt request signal, the microprocessor issues an interrupt acknowledge signal through this pin.
- **ALE:** This signal is used to latch the address into the address latch (**8282/8283**).

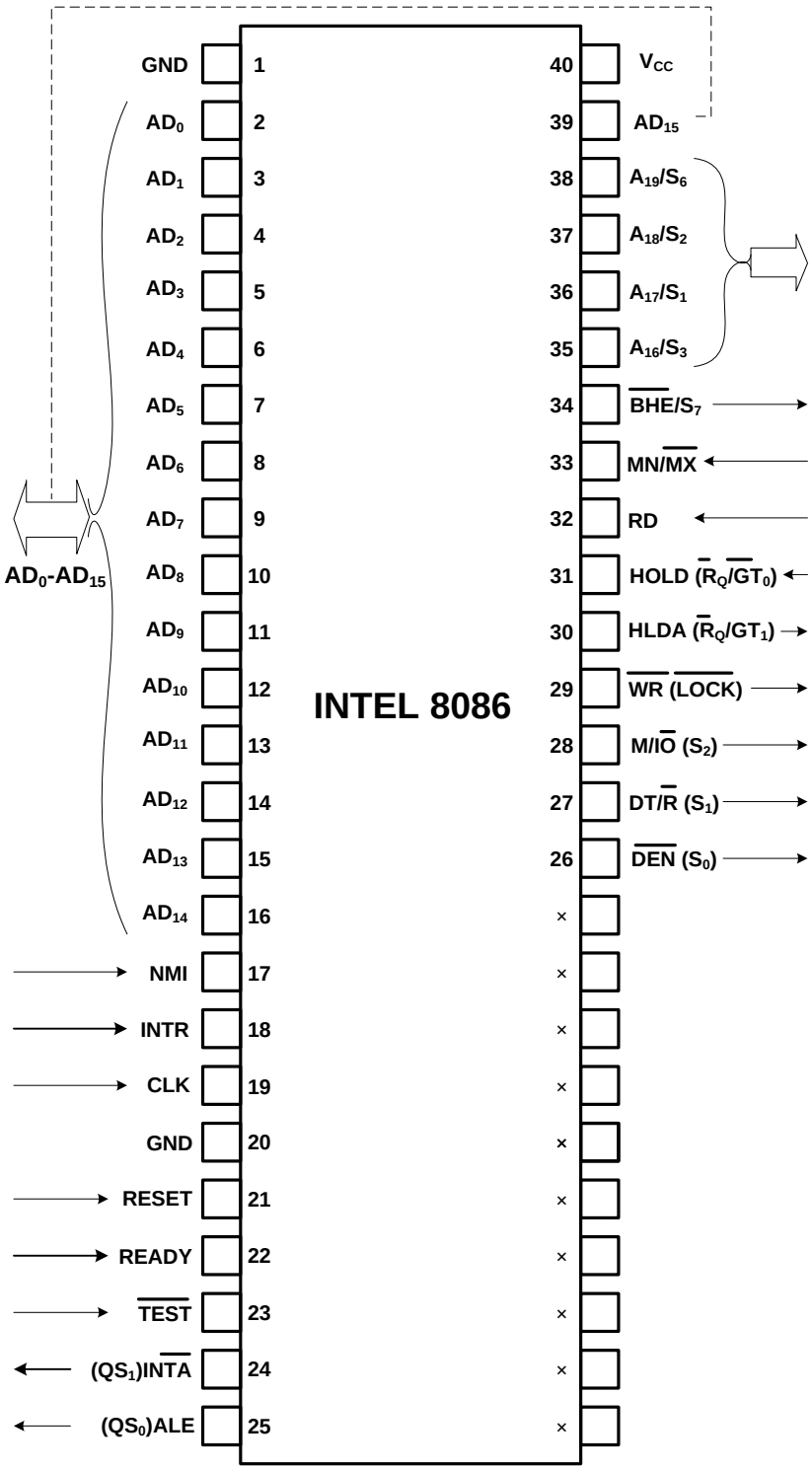


FIG: INTEL 8086 PIN DIAGRAM

CHAPTER 2

Microprocessor Instruction

Contents:

- Definitions
- Instruction Description
- Instruction Sheets & Instruction Format
- Register Transfer Language
- 8085 & 8086 Addressing Modes
- 8085 Instruction Set
- 8085 Machine Cycles & Timing Diagram
- 8085A MP Machine Codes
- Counters & Time Delays

Definitions

Instruction

An instruction is a binary pattern designed inside a microprocessor to perform a specific function. In another word it can be defined as command in binary (i.e. machine language) that is recognized and executed by a computer to accomplish a task.

Computer instructions are normally 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, and so on. This type of instruction sequencing needs a counter to calculate the address of the next instruction after the execution of the current instruction is completed. Thus execution of a program consists of a sequence of instruction cycles, with one machine instruction per cycle.

Instruction Cycle

The entire group of instruction that determines what functions the microprocessor can perform is called instruction set. Instruction cycle 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. Since, instruction is decoded in binary form and stored in memory; on the basis of its clock speed computer takes a certain period to complete the task such as fetching, decoding and execution.

The instruction cycle is combination of different smaller units. Basically, the instruction cycle consists of two cycles, they are:

- **Fetch and Decode Cycle**
- **Execute Cycle**

The essential component of every instruction cycle is the fetch 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 thus consists of different operations such as calculating the address of the operands, fetching them, performing operations on them and finally outputting the results to a specified location.

Machine Cycle

Machine cycle is defined as the time required for completing one operation of accessing memory, **I\O** or acknowledging an external request. This cycle may consist of three to six **T**-states. It consists of four operations: **Op-code Fetch, Memory Read, Memory Write, I/O Read and I/O Write**. These micro operations will be discussed later.

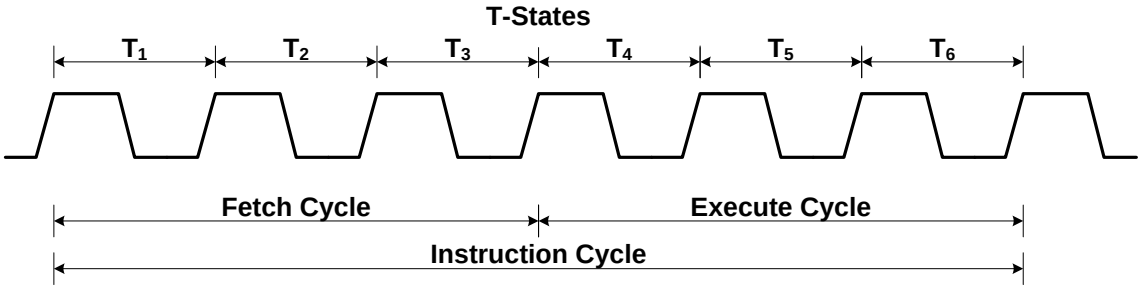


Fig: Instruction Description

T-State

T-state is defined as the sub division of the operation performed in one clock period. During **T**-second microprocessor (a sequential machine) stays in a particular state called **T**-state. So a **T**-state lasts for a period of one clock cycle. T state and clock cycle are used synonymously.

Instruction Description

An instruction manipulates stored data and a sequence of instructions constitutes a program. In general an instruction has two components, they are:

- **Operation Code Field (Op-code)**
- **Address Field (Operand)**

The Op-code field specifies how data is to be manipulated. A data item may reside within a microprocessor register or in the main memory. Thus, the purpose of the address field is to indicate the address of a data item. The address field may contain more than one address to store a data. For example let us consider the following example.

ADD	**R_1 R_0**
Op-code	**Operand**

If **R_0** is the source register and **R_1** is the destination register. The instruction then adds the contents of register **R_0** with that of **R_1** and stores result in **R_1**. Here, the combination of source register **(R_0)** and destination register **(R_1)** is an operand whereas the command **ADD** is an **Op-code**. Finally, **Op-code** and **Operand** form an instruction.

Instruction Sheet & Instruction Formats

Each microprocessor can handle predefined number of instructions. For example an **8085A** can handle at the maximum of **256** instructions **($2^{\text{word length}}$)**. The sheet which contains all these instructions with their hex code, mnemonics, descriptions and functions is called an instruction sheet.

Depending upon the number of address specified in instruction sheet, the instruction format can be classified into three categories, they are:

- **One Address Format (1 Byte Instruction)**
- **Two Address Format (2 Byte Instruction)**
- **Three Address Format (3 Byte Instruction)**

1 Byte Instruction:

One byte will be Op-code and Operand will be default for that instruction. A one byte instruction includes the Op-code and Operand in the same byte. These instructions are stored in **8-bit** binary format in memory; each requires one memory location.

Examples:

Task	Op-Code	Operand	Binary Code	Hex Code
Copy The contents of the accumulator in register C.	**MOV**	**C, A**	**0100 1111**	**4FH**
Add the contents of register B to the contents of the accumulator.	**ADD**	**B**	**1000 0000**	**80H**
Invert/Compliment each bit in the accumulator.	**CMA**	**-**	**00101111**	**2FH**

2 Byte Instruction:

In two byte instruction, the first byte specifies the operation code and the second byte specifies the operand. These instructions would require two memory locations each to store the binary codes.

Examples:

Task	Op-Code	Operand	Binary Code	Hex Code
Load an **8-bit** data byte in the accumulator.	**MVI**	**A, 32H**	**0011 1110** **0011 0010**	**3E** **32**
Load an **8-bit** data byte in the register **B**.	**MVI**	**B, F2H**	**0000 0110** **1111 0010**	**06** **F2**

2 BYTE INSTRUCTION:

In three byte instruction, the first byte specifies the operation code and the following two bytes specify the **16-bit** address where the second byte is the low order address and the third byte is the high order address.

Examples:

Task	Op-Code	Operand	Binary Code	Hex Code
Load contents of memory **2050H** into **A**	**LDA**	**2050H**	**0011 1010** **0101 0000** **0010 0000**	**3A** **50** **20**
Transfer the program sequence to memory location **2085H.**	**JMP**	**2085H**	**1100 0011** **1000 0101** **0010 0000**	**C3** **85** **20**

Register Transfer Language (RTL)

A digital system is an interconnection of digital hardware modules that accomplish a specific information processing task. Digital modules are best defined by the register they contain and the operations that are performed on the data stored in them.

A language that uses the symbolic notation to literally define, express and describe the transfer of data among the register called register transfer language. **RTL** describes the micro operations. The operations executed on data stored in registers are called microoperations. Microoperations are the functional or atomic operations of a processor, which is an elementary operation performed on information stored in one or more registers e.g. shift, count, clear, load etc. In micro operation each step is very simple and accomplishes very literally.

The term "**Register Transfer"** implies the availability of hardware logic circuits that can perform a stated microoperation and transfer the result of the operation to the same or another register. The word "**Language"** is borrowed from programmers, who apply this term to programming language.

Example: **A ← B;** denotes a transfer of the content of register **B** into register **A**. It designates a replacement of the content of **A** by the content of **B.**

Function of Control Unit of Microprocessor

The control unit of microprocessor performs the following two tasks.

- Causes the microprocessor to execute the micro operation in a proper sequence determined by the program being executed.
- Generate the control signal that causes each micro operation to be executed. The control signal generated by the control unit causes the opening and closing of logic gates, thus resulting in the transfer of data to and from register and the operation of **ALU**.

8085 Addressing Modes

As we know, instructions are command to the microprocessor to perform a certain task. The instruction consists of **Op-code** and **Data** called **Operand**. It deals with the **Source**-register or input port or a memory location and the **Destination**-a register or an output port or a memory location. The various formats for specifying the operands for an instruction are called addressing modes. The **8085** instruction set has different addressing modes which are described below.

- **Direct Addressing (Absolute Addressing)**
- **Register Indirect Addressing**
- **Register Direct Addressing**
- **Implied Addressing (Inherent Addressing)**
- **Immediate Addressing**

1) Direct or Absolute Addressing:

In this mode **8** or **16-bit** address of an operand will be given just after the **Op-code**. It means address is written directly in instruction. These instructions are of **2 or 3** byte with first byte as the **Op-code** followed by **1 or 2** bytes of address of data.

Examples:

(i) LDA 2035H
This instruction loads the accumulator with the contents of the memory location having **16-bit** address of **2035H.**
(ii) IN FCH
This is an example which has only one byte address. This will read the data at port **FCH** and store the data at the accumulator.
(iii) OUT FFH
This instruction will display/write the data stored in accumulator at port address **FFH.**

2) Register Indirect Addressing:

In this mode, the address part of instruction specifies the memory location whose content is the address of the operand. So this type of addressing mode contains a register pair which stores the address of actual data. Since register pair is used indirectly to store the address of data, it is called register indirect addressing.

Examples:

(i) MOV A, M
This instruction moves the contents of the memory location whose address is specified by the contents of the register **H** & **L** to the accumulator. If register **H** & **L** contains **20H** and **50H** respectively then the contents of memory location **2050H** will be transferred to the accumulator.

(ii) ADD M
This instruction adds the contents of the memory location whose address is specified by the content of the register **H** & **L** to the content of the accumulator.
(iii) MVI M, 8bit data
This instruction moves the immediate **8-bit** data to the memory location whose address is specified by the contents of the register **H** & **L.**

3) Register Direct Addressing:

This mode of addressing specifies the register or register pair that contains the actual data to be manipulated or operated on by the microprocessor.

Examples:

MOV A, B
This instruction moves the contents of the register **B** directly to register **A** (Accumulator).
ADD B
This instruction adds the contents of the register **B** with the contents of accumulator directly and stores the result in accumulator.
ORA B
This instruction stores the result in accumulator after bitwise **OR**-operation between the contents of register **B** and Accumulator.

4) Implied or inherent addressing:

The instruction of this mode does not require operand and the operand is implicitly implied in the instruction. It implicitly assumes that the operand is accumulator. These instructions are **1-byte** instruction.

Examples

HLT: Halt the operation.
EI: Enable interrupt
STC: Set the carry flat.

5) Immediate Addressing:

In this mode the data is specified immediately after **Op-code** in the instruction. For an **8-bit** data, this mode uses **2-bytes** instruction, with first byte as the **Op-code** followed by **1** or **2 byte** of data whereas for **16-bit data**, this mode uses **3-bytes** instruction with first byte as the **Op-code** followed by **2 bytes** of data. In both cases, the actual data is part of the instruction, and hence called immediate addressing.
Examples:
MVI A, 05H (8-bit Data)
This instruction moves the immediate data **05H** to the accumulator.
LXI H, 7A21J (16-bit Data)
This instruction loads register **H** with **7AH** and register **L** with **21H.**

8086 Addressing Modes

The addressing mode is a way of specifying a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced. The way the operands are chosen during program execution is dependent on the addressing mode of the instruction.

PURPOSE:

- To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data and program relocation.
- To reduce the number of bits in the addressing field of the instruction.
- To provide flexibility for writing programs that are more efficient with respect to the number of instructions and execution time.

The **8086** processors let us access memory in many different ways. The **8086** memory addressing modes provide flexible access to memory, allowing us to easily access variables, arrays, records, pointers and other complex data types. The **8086** provides various addressing modes to access instruction operands. Operands may be contained in registers, within the instruction Op-code, in memory or in **I/O** ports.

The **8086** has twelve addressing modes. These modes can be classified into five groups. They are:

- **Register & Immediate Mode**
- **Memory Addressing Mode**
- **Port Addressing Mode**
- **Relative Addressing Mode**
- **Implied Addressing Mode**

1. REGISTER & IMMEDIATE MODE

In register mode source operands, destination operands or both may be contained in registers. For example, **MOV AX, BX** moves the **16-bit** contents of **BX** into **AX**. On the other hand, **MOV AL, BL** moves the **8-bit** contents of **BL** into **AL.**

In immediate mode, **8-** or **16-bit** data can be specified as part of the instruction. For example, **MOV CX, 5062H** moves the **16-bit** data $\mathbf{5062_{16}}$ into register **CX.**

2. MEMORY ADDRESSING MODE

The execution unit (**EU**) has direct access to all registers and data for register and immediate modes. However, the **EU** cannot directly access the memory operands. For example, when the **EU** needs a memory operand, it sends an offset value to the bus interface unit (**BIU**). This offset is added to the contents of a segment register after shifting it **4** times toe the left, generating a **20-bit** physical address. For example, suppose that the content of a segment register is $\mathbf{2052_{16}}$ and the offset is $\mathbf{0020_{16}}$. Now, in order to generate the **20-bit** physical address, the **EU** passes this offset to the **BIU**. The **BIU** then shifts the segment register **4** times to the left, obtains $\mathbf{20520_{16}}$ and then adds the $\mathbf{0020_{16}}$ offset to provide the **20-bit** physical address $\mathbf{20540_{16}}$. The **8086** must use as segment register whenever it accesses the memory. Also, every memory addressing mode has a standard default segment register.

However, a segment override instruction can be placed before most of the memory operand instructions whose default segment register is to be overridden. For example, **INC BYTE PTR [START]** will increment the **8-bit** contents of a memory location in data segment (**DS**) with offset **START** by one. However, segment **DS** can be overridden by extra segment (**ES**) as follows: **INC ES: BYTE PTR [START]**. Segments cannot be overridden for stack reference instructions such as **PUSH** and **POP**. The destination segment of a string segment, which must be **ES** (if a prefix is used with a string instruction, only the source segment **DS** can be overridden) cannot be overridden. The code segment (**CS**) register used in program memory addressing cannot be overridden.

The **EU** calculates an offset from the instruction for a memory operand. This offset is called the operand's effective address (**EA**). It is a **16-bit** number that represents the operand's distance in bytes from the start of the segment in which it resides.

The various memory addressing modes are as follows:

- **Memory Direct Addressing**
- **Register Indirect Addressing**
- **Based Addressing**
- **Indexed Addressing**
- **Based Indexed Addressing**
- **String Addressing**

1. Memory Direct Addressing:

In this mode, **EA** is taken directly from the displacement field of the instruction. No registers are involved. For example, **MOV BX, START** moves the contents of the **20-bit** address computed from **DS** and **START** to **BX**.

2. Register Indirect Addressing:

EA of a memory operand may be taken directly from one of the base or indeed registers **(BX, BP, SI, DI).** For example, consider **MOV CX, [BX].** If **[DS]** = $\mathbf{2000_{16}}$, **[BX] =** $\mathbf{004_{16}}$, and **[**$\mathbf{20004_{16}}$**] =** $\mathbf{0224_{16}}$, then after **MOV CX, [BX]**, the contents of **CX** is $\mathbf{0224_{16}}$. Here the segment register used in **MOV CX, [BX]** can be overridden, such as **MOV CX, ES: [BX].** Now, the **MOV** instruction will use **ES** instead of **DS**. If **[ES] =** $\mathbf{1000_{16}}$ **and [**$\mathbf{10004_{16}}$**] =** $\mathbf{0002_{16}}$, then after **MOV CX, ES: [BX]**, the register **CX** will contain $\mathbf{0002_{16}}$.

3. Based Addressing:

EA is the sum of a displacement value (signed **8-bit** or unsigned **16-bit**) and the contents of register **BX** or **BP**. For example, **MOV AX, 4[BX]** moves the contents of **20-bit** address computed from a segment register and **[BX+4]** into **AX.** The segment register is **DS** or **SS**. If a displacement (**4** in this case) is **8-bit**, then the **8086** sign extends this to **16-bit** in order to use **SS** register when the stack is accessed; otherwise, this mode uses segment register **DS**. When memory is accessed, the **20-bit** physical address is computed from **BX** and **DS**. On the other hand, when the stack is accessed, the **20-bit** physical address is computed from **BP** and **SS**. The **BP** may be considered as the user stack pointer while **SP** is the system stack pointer.

4. **Indexed Addressing:**

EA is calculated from the sum of a displacement value and the contents of register **SI** or **DI.** For example, **MOV AX, VALUE [SI]** moves the contents of the **20-bit** address computed from **VALUE, SI** and the segment register into **AX**. The segment register is **DS.** The displacement (**VALUE** in this case) can be unsigned **16-bit** or sign-extended **8-bit.**

5. **Base Indexed Addressing:**

EA is computed from the sum of a base register **(BX or BP),** and index register **(SI or DI),** and a displacement. For example, **MOV AX, 4[BX] [SI]** moves the contents of the **20-bit** address computed from the segment register and **[BX] + [SI] + 4** into **AX.** The segment register is **ES.** The displacement can be unsigned **16-bit** or sign-extended **8-bit.**

6. **String Addressing:**

This mode uses index registers. **SI** is assumed to point to the first byte or word of the source string ad **DI** is assumed to point to the first byte or word of the destination when a string instruction is executed. The **SI** or **DI** is automatically incremented or decremented to point to the next byte or word depending on **DF**. The default segment register for source is **DS** and it may be overridden. The segment register for destination must be **ES**.

3. Port Addressing Mode

Two **I/O** port addressing modes can be used: the direct port and indirect port modes. In either case, **8**-or **16-bit I/O** transfers must take place via **AL** or **AX**, respectively.

In **direct port mode**, the port number is an **8-bit** immediate operand to access **256** ports. For example, **IN AL, 02** move the contents of port **02** to **AL**.

In **indirect port mode**, the port number is taken from **DX** allowing **64K** bytes or **32** words or ports. For example, suppose **[DX] = 0020, [port 0020] = 02_{16}, and [port 0021] = 03_{16}**, then after **IN AX, DX** register **AX** contains **0302_{16}**. On the other hand, after **IN AL, DX,** register **AL** contains **02_{16}.**

4. Relative Addressing Mode

Instructions using this mode specify the operand as a signed **8-bit** displacement relative to **PC**. An example is: **JNC START.** This instruction means that if carry = **0**, then **PC** is loaded with current **PC** contents plus the **8-bit** signed value of **START**; otherwise, the next instruction is executed.

5. Implied Addressing Mode

Instructions using this mode have no operands. An example is **CLC**, which clears the carry flag to zero.

8085 Instruction Set

An instruction is a binary pattern designed inside a microprocessor to perform a specific function. The entire group of instruction is called instruction set, which determines what functions the microprocessor can perform. The **8085** instruction set can be classified into following five functional categories

- **Data Transfer Operation**
- **Arithmetic Operation**
- **Logical Operation**
- **Branching Operation**
- **Machine Control Operation**

1. Data Transfer Operations:

This group of instructions copies data from a location called a source to another location, called a destination, without modifying the contents of the source. The data transfer operations in 8085 consists of 88 different Op-codes. The transfer of data may be between the registers or between register and memory or between and I/O device and accumulator. None of these instructions changes the flags.

Data transfer group consists of following set of instructions.

Mnemonics	Operation	Examples
1. MVI r, data (8 bit)	Move data immediate to register	MVI B, 4FH
2. MVI M, data (8 bit)	Move data immediate to memory, whose address is in H & L	MVI M, 3AH
3. MOV r_1, r_2	Move data from register to register (r_2 to r_1)	MOV B, A
4. MOV M, r	Move data from register r to memory whose address is in H&L	MOV M, C
5. MOV r, M	Move data from location specified by H&L register to register r.	MOV B, M
6. LXI rp, data	Load immediate register pair with double byte data.	LXI B 2050H
7. STA addr	Store the data from A direct at the address that follows.	STA 2070H
8. LDA addr	Load the data into A direct from the address that follows.	LDA 2050H
9. SHLD addr	Store H & L direct.	SHLD 2050H
10. LHLD addr	Load H & L direct.	LHLD 2040H
11. LDAX rp	Load A with the contents of the memory location whose address is in rp.	LDAX B
12. STAX rp	Store the contents of A in memory whose address is in rp.	STAX D
13. XCHG	Exchange the content of H with D and L with E.	XCHG
14. IN 8bit (port addr)	Read the data from input device and store in accumulator.	IN 07H
15. OUT 8bit (port addr)	Write the data from an accumulator to output device.	OUT 32H

2. ARITHMETIC OPERATIONS:

Arithmetic Operation performs following task

- **Addition**
- **Subtraction**
- **Increment (Add by 1)**
- **Decrement (Subtract by1)**

This group consists of following set of instruction

Mnemonics	Operation	Examples
1. INR r	Increment the contents of register	INR D
2. INR M	Increment the contents of memory, the address of which is in HL.	INR M
3. INX rp	Increment the contents of register pair.	INX H
4. DCR r	Decrement the contents of register pair	DCR E
5. DCR M	Decrement the contents of memory, which address is in HL.	DCR M
6. DCX rp	Decrement the contents of register pair	DCX H
7. ADD r	Add the contents of register to the contents of accumulator.	ADD B
8. ADD M	Add the contents in memory to A; the address of which is in HL.	ADD M
9. ADI data	Add data immediate to A.	ADI 37H
10. ADC r	Add register r with carry to A.	ADC B
11. ADC M	Add data in memory to A with carry.	ADC M
12. ACI data	Add immediate data to A with carry.	ACI 05H
13. DAD rp	Add register pair rp to H and H. here rp can be BC, DE, HL or SP.	DAD H
14. SUB r	Subtract register r from A.	SUB C
15. SUB M	Subtract the contents of memory from A; the address of memory is in HL.	SUB M
16. SUI data	Subtract data immediate from A.	SUI 37H
17. SBB r	Subtract register r from A with borrow	SBB D
18. SBB M	Subtract the contents of memory from A with borrow.	SBB M
19. SBI data	Subtract immediate data from A with borrow.	SBI 37H
20. DAA	Decimal adjust accumulator.	DAA

This adjusts A to packed **BCD** after addition of two **BCD**s. It functions in two steps.

a) If the lower **4-bit** of **A** are greater than **9** or the auxiliary carry is set, then it adds **06H** to **A**.
b) Subsequently if the higher **4-bits** of **A** are now greater than **9** or the carry flag is set, it adds **60H** to **A.**

Example:

$A = 39_{BCD}$
Add 12_{BCD}

$A = 39_{BCD}$	= 0011	1001
$+ 12_{BCD}$	= 0011	0010
51_{BCD}	= 0100	1011 = 4B

The binary sum is **4BH.** The value of low order four bit is larger than **9**. Add **06** to low order **4**-bits.

	0011	1001
+	0001	0010
4B	0100	1011
+06		0110
51H	0101	0001

3. Logical Instructions:

This instruction performs following operations

- **AND**
- **OR**
- **XOR (Exclusive OR)**
- **Compare**
- **Rotate Bits**
- **Complement**

This group consists of following set of instructions:

Mnemonics	Operation	Example
1. ANA r	Logically AND the contents of a register with the contents of A.	ANA B
2. ANA M	Logically AND the contents of A with the contents of memory whose address is in HL register.	ANA M
3. ANI data	AND immediate with A.	ANI 32H
4. XRA r	Exclusive OR register with A.	XRA B
5. XRA M	Exclusive OR memory with A.	XRA M
6. XRI data	Exclusive OR immediate with A.	XRI 07H
7. ORA r	OR register r with A.	ORA B
8. ORA M	Logically OR the contents of memory with contents of A; the address of which is in HL register.	ORA M
9. ORI data	Logically OR immediate data with the contents of A.	ORI 20H
10. CMP r	Compare the contents of register r with the contents of A.	CMP B
11. CMP M	Compare the contents of memory with contents of A; the address of which is in HL.	CMP M
12. CPI data	Compare data immediate with A	CPI 27H
13. STC	Set carry	STC
14. CMC	Complement carry	CMC
15. CMA	Complement A.	CMA
16. RLC	Rotate A left without carry.	RLC
17. RRC	Rotate A right without carry.	RRC
18. RAL	Rotate A left through carry.	RAL
19. RAR	Rotate A right through carry.	RAR

4. Branching Instruction:

This group consists of following set of instructions.

- JMP 16 bit address Jump unconditionally to the address.
- $J_{Cond.}$ Address; Jump conditionally to the address given.

This instruction causes a jump to the given address if a specified condition is satisfied. The condition could be:

JC	Jump on carry	C = 1
JNC	Jump on not carry	C = 0
JP	Jump on positive	S = 0
JM	Jump on minus	S = 1
JPE	Jump on parity even	P = 1
JPO	Jump on parity Odd	P = 0
JZ	Jump on zero	Z = 1
JNZ	Jump on not zero	Z = 0

- PCHL: Copies the contents of HL into the following program counter.
- CALL address: Call unconditionally a sub routine at the address given.
- $C_{cond.}$ Address: Call sub routine at the given address conditionally.

This instruction calls the subroutine at the given address if a specified condition is satisfied. The following conditional calls can be made

CC	Call on carry	C = 1
CNC	Call on no carry	C = 0
CP	Call on positive	S = 0
CM	Call on minus	S = 1
CPE	Call on parity even	P = 1
CPO	Call on parity odd	P = 0
CZ	Call on Zero	Z = 1
CNZ	Call on no zero	Z = 0

- RET: Return from the subroutine unconditionally.
- $R_{cond.}$: Return from the subroutine conditionally.

This instruction returns the control to the main program if the specified condition is satisfied.

RC	Return on carry	C = 1
RNC	Return on no carry	C = 0
RP	Return on positive	S = 0
RM	Return on minus	S = 1
RPE	Return on parity even	P = 1
RPO	Return on parity odd	P = 0
RZ	Return on zero	Z = 1
RNZ	Return on no zero	Z = 0

5. Stack Operations And Machine Control Instruction:

- **Stack Operations:**

  This group consists of the following set of instructions

  - PUSH rp: Push registers pair rp onto stack.
  - PUSH PSW: Push processor status word onto stack
  - POP rp: Pop register pair off the stack
  - XTHL: Exchange top of stack with H and L.
  - SPHL: H and L to stack pointer.

- **Machine Control:**

  - NOP: No Operation
  - HLT: Halt the processor

8085 Machine Cycle & Timing Diagram

To execute an instruction the **8085** needs to perform various operations such as memory read\write and **I\O** read\write. All instructions are divided into a few basic machine cycles and these machine cycles are divided into precise clock period.

Basically, the microprocessor external communication functions can be divided into three categories.

1.) **Memory read and write**
2.) **Input output read and write**
3.) **Request acknowledge**

These functions are further divided into various operations (machine cycles) as shown below.

Machine cycle	Status			Control signals
	IO\M	S_1	S_0	
Op-code fetch	0	1	1	$\overline{RD}$ = 0
Memory Read	0	1	0	$\overline{RD}$ = 0
Memory Write	0	0	1	$\overline{WR}$ = 0
I\O Read	1	1	0	$\overline{RD}$ = 0
I\O Write	1	0	1	$\overline{WR}$ = 0
Interrupt Acknowledge	1	1	1	$\overline{INTA}$ = 0

Op-Code Fetch Machine Cycles

The first operation in any instruction is Op-code fetch. The microprocessor needs to get (fetch) this machine cycle code from the memory register where it is stored before the microprocessor can begin to execute the instruction. Therefore the Opcode fetch cycle constitutes the first machine cycle of an instruction cycle, i.e. $\mathbf{M_1}$. This Opcode fetch cycle begins from **T**-state $\mathbf{T_1}$ by placing the content of program counter on address bus and continues till the Opcode decoding and execution process.

For example to fetch the byte, the processor needs to identify the memory location **2005H** and enable the data flow from memory. The data flow is shown in figure and the timing diagram is also shown below.

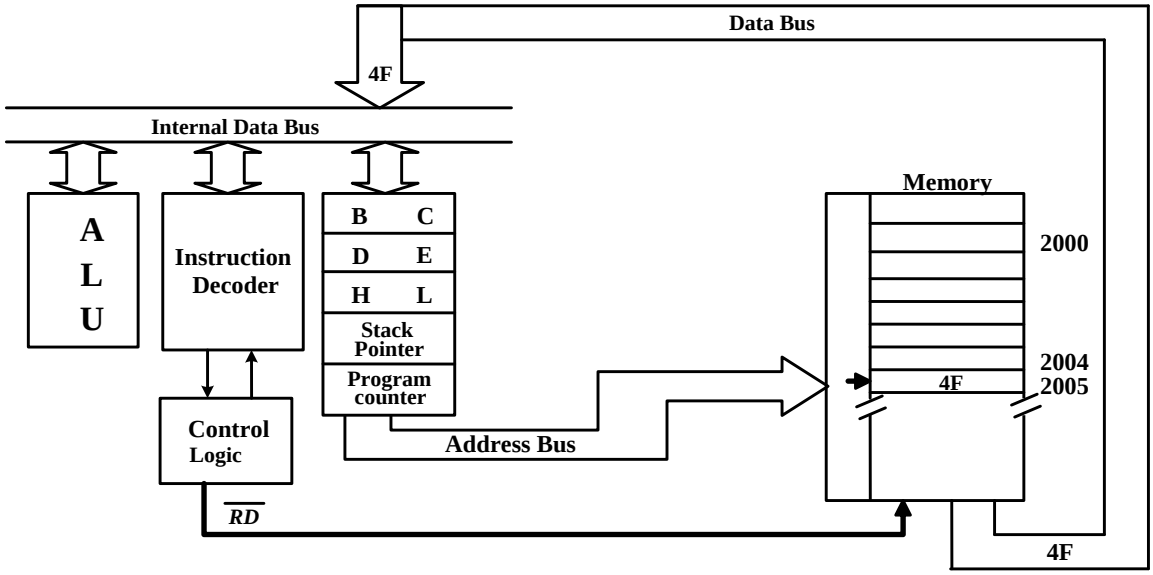


Fig: Data Flow from memory to MPU

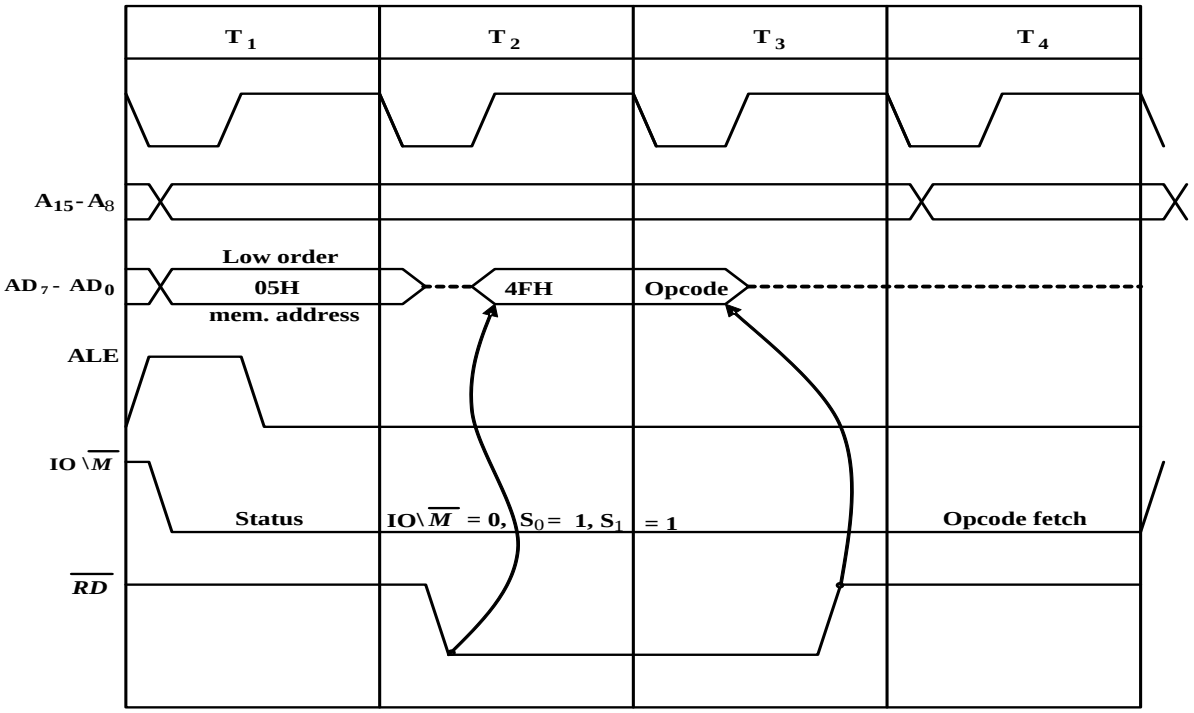


Fig: Timing Diagram for Opcode Fetch

To fetch a byte, the processor performs the following steps:

Step 1: The program counter places the **16**-bit memory address on the address bus.

Step 2: The control unit sends the control signal $\overline{RD}$ to enable the memory chip.

Step 3: The byte from the memory location is placed on the data bus.

Step 4: The byte is placed in the instruction decoder of the microprocessor and the task is carried out according to the instruction.

Note:

- The crossover of the lines indicates that a new byte (information) is placed on the bus.
- The dashed straight line indicates the high impedance state.

Memory Read Machine Cycles

A memory read cycle is a process of transferring a byte from memory to the microprocessor. To read a data from memory, microprocessor needs to identify and enable the memory cell. For this purpose microprocessor places the address of memory cell on address bus and issues different control signal from control unit.

The **MPU** performs the following steps to read the data from memory. Suppose the memory read cycle need to perform to read the data byte **32h** stored in memory location **2000h**.

Step 1: The microprocessor places the content of program counter on the address bus. In our case, it is **2000h**. The lower address byte '**00'** is latched with the help of **ALE** signal. If it is not latched, the lower address byte will disappear or lost when microprocessor places the data in data bus.

Step 2: The address decoding unit decodes the address bits and find out the desired memory cell.

Step 3: The control unit sends the control signal **RD** to enable the memory. The status signal **IO/M** goes low to indicate that it is memory related operations. The next two status signals go high and low respectively to indicate that it is read operation.

Step 4: The data byte from memory location **2000h** is placed on the data bus and transfer to the microprocessor.

To illustrate the memory read machine cycle, we need to examine the execution of a **2 byte** or **3 byte** instruction because in a **1 byte** instruction the machine code is an Op-code fetch. Let us illustrate with an example:

Memory Location	Machine Code	Instruction
2000H	3EH	MVI A, 32H; Load 32H in accumulator
2001H	32H	

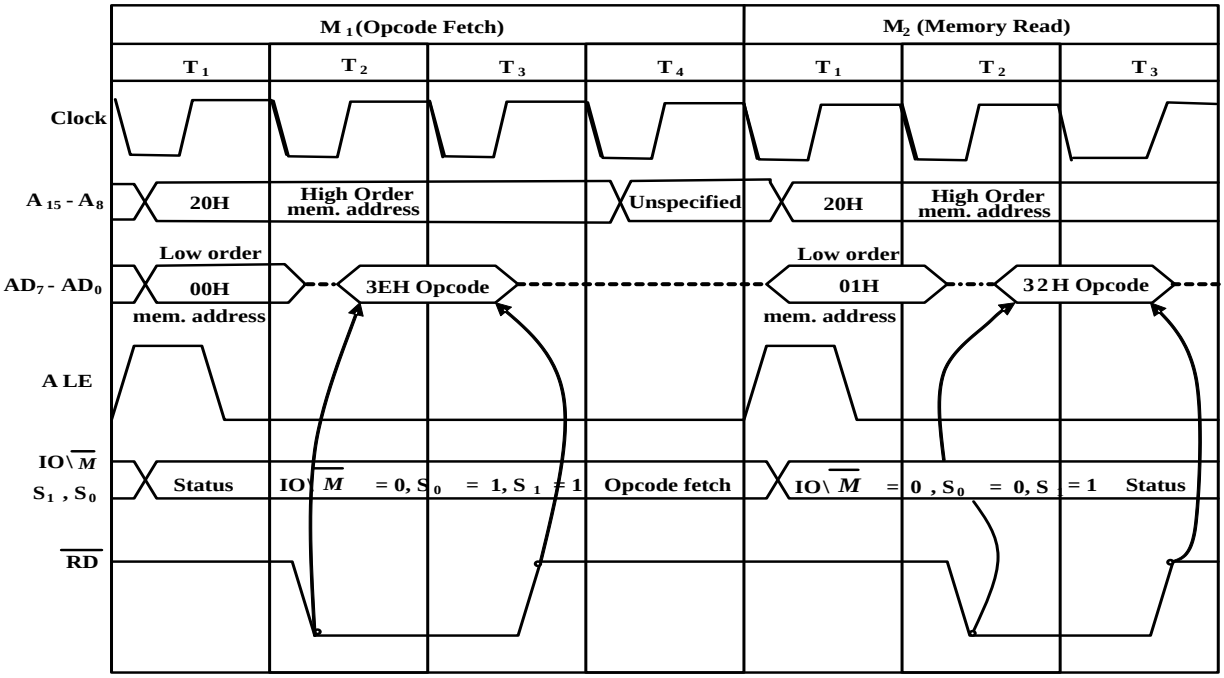


Fig: Timing Diagram for MVI A 32 h

Memory Write Machine Cycles

To write into a register, the microprocessor performs similar steps as it reads from a register. The following example shows the memory write cycle with an example of **STA** instruction. In the write operation, the **8085** places the address & data and asserts the **IO/M** signal. After allowing sufficient time for data to become stable, it asserts the write (**WR**) signal. The **IO/M** and **WR** signals can be combined to generate the **MEMW** control signal that enables the input buffer of the memory chip and stores the byte in the selected memory register.

Timing Diagram for STA Instruction

Memory Address	Machine Code	Mnemonics	Comment
2050H	32	STA 8000H	Store Contents of Accumulator in Memory Location 8000H
2051H	00	-	
2052H	80	-	

Here it is assumed that the instruction is stored in memory location **2050H**, **2051H** and **2052H**. The timing diagram for instruction **STA 8000H** is shown below:

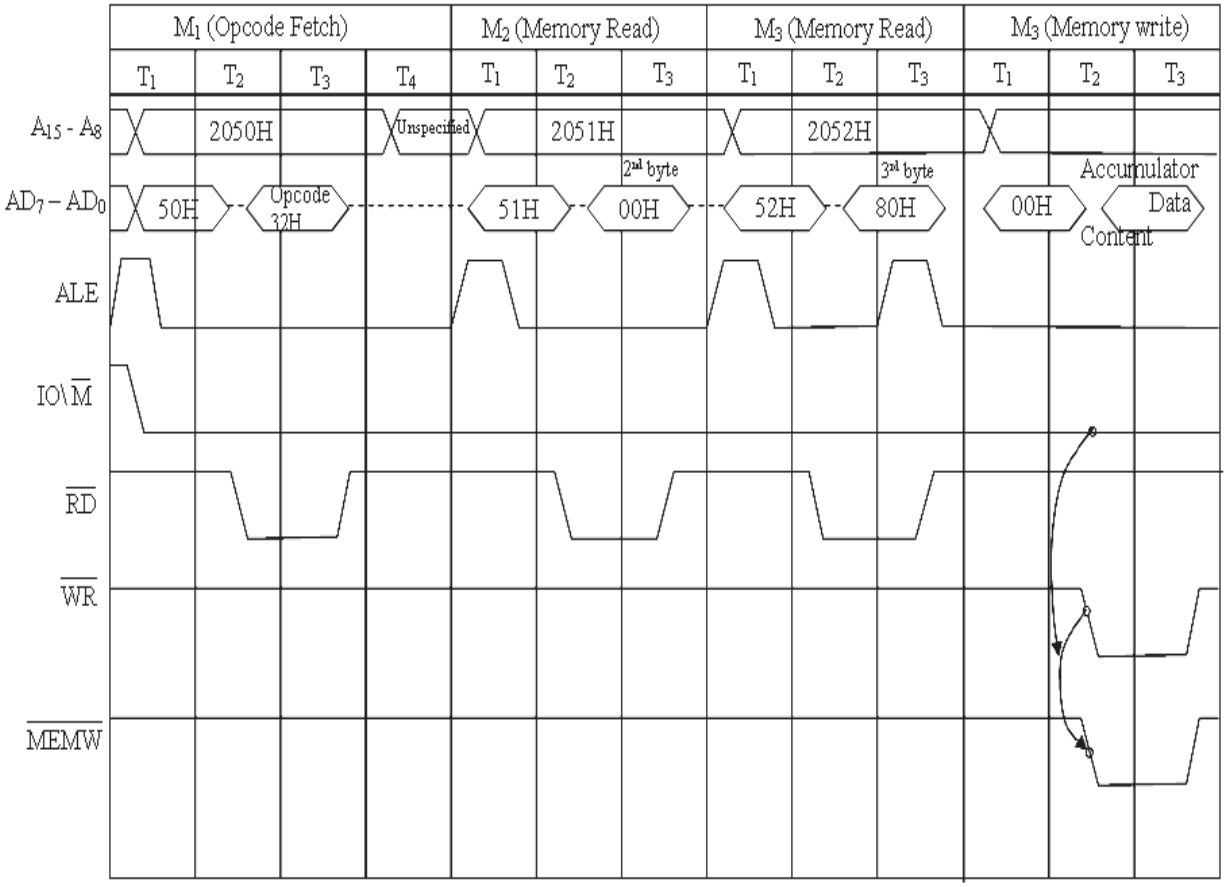


Fig: Timing Diagram for STA 8000H

I/O Read Machine Cycle

The purpose of **I/O** Read is to read data from input device such as keyboard and places the data in the accumulator. To illustrate the timing diagram for **I/O** Read, we examine the execution of instruction **IN 84h**, stored at memory location: **2065.**

Timing Diagram For IN Instruction

IN Instruction: **IN 8-bit address**
2 Byte Instruction
M - Cycles: **3 (Op-Code Fetch + Memory Read + I\O Read)**
T-States: **10**

Example: **IN 84H**

Memory Address	Machine Code	Mnemonics	Memory Contents
2065H	DBH	In 84H	2065: 1101 1011H = DBH
2066H	84H	-	2066: 1000 0100H = 84H

The timing diagram for instruction **IN 84H** is shown below:

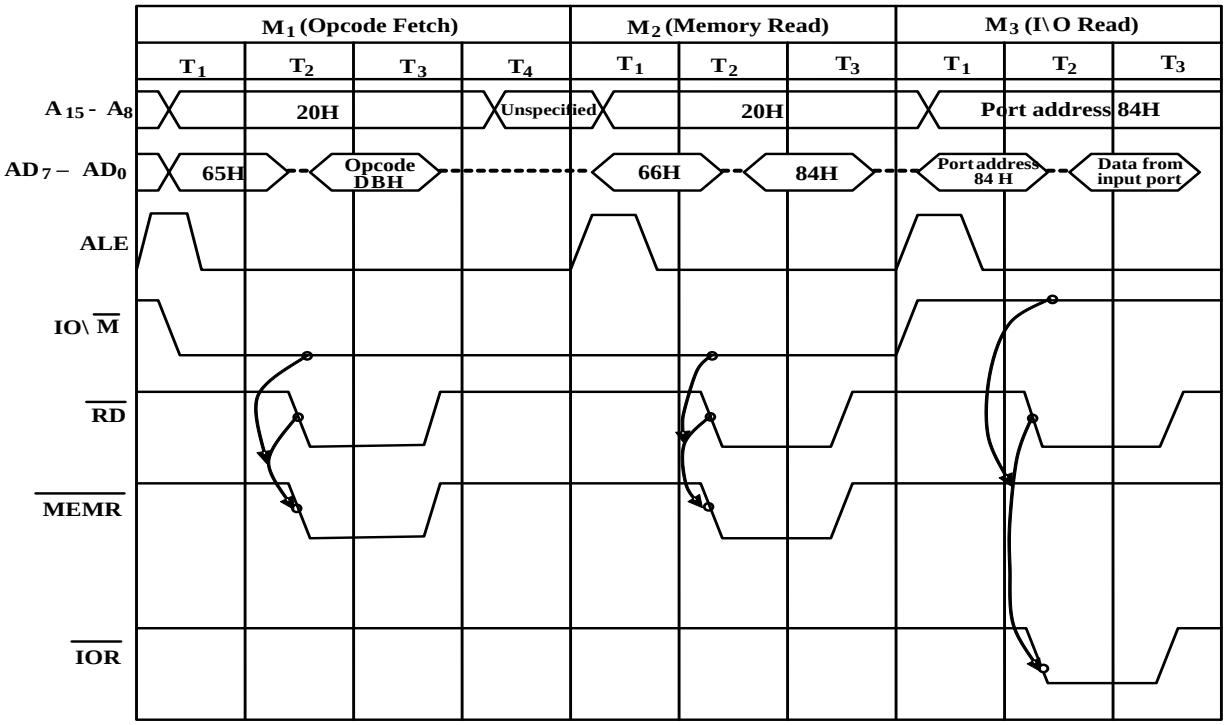


Fig: Timing Diagram of IN 84 h

EXPLANATION:

- **The First Machine Cycle (M_1): Op-Code Fetch**

  The **8085** places the high order memory address **20H** on **A_{15} –A_8** and the low order address **50H** on **AD_7-AD_0.** At the same time, **ALE** goes high and **IO/$\overline{M}$** goes low. The **IO\$\overline{M}$** low indicates that it is a memory operation. At **T_2**, the microprocessor sends the $\overline{RD}$ control signal which is combined with **IO\$\overline{M}$** to generate the $\overline{MEMR}$ signal and the processor fetches the instruction code **D_3** using the data bus and the instruction is decoded.

- **The Second Machine Cycle (M_2): Memory Read**

  In the second machine cycle, **M_2** (Memory Read), the **8085** places the next address **2051H** on the address bus and get the device address **84H** via data bus.

- **The Third Machine Cycle (M_3): I/O Write**

  In the third machine cycle, **M_3** (**I\O** Write), the **8085** places the device address **84H** on the low order **(AD_7 – AD_0)** as well as the high order **(A_{15} – A_8)** address bus. The **IO\$\overline{M}$** signal goes high to indicate that it is an **I\O** operation. The **IOR** signal is generated combining **RD** and **IO/M**. The **IOR** signal enables the input port and data from input port are placed on the data bus and transformed into the accumulator.

Input/Output Write Machine Cycle

The purpose of **I/O** write is to send the content of accumulator to an **O/P** device such as **LED** display. To illustrate the timing diagram for **I/O** write, we examine the execution of instruction **OUT 01h**, stored in memory location: **2050**as follows:

Timing Diagram For OUT Instruction

OUT Instruction: **OUT 8-bit address**
2 Byte Instruction
M-Cycles: **3 (Op-Code Fetch + Memory Read + I\O Write)**
T-States: **10**

Example: **OUT 01H**

Memory Address	Machine Code	Mnemonics	Memory Contents
2050H	D3	OUT 01H	2050: 1101 0011H = D3H
2051H	01	-	2051: 0000 0001H = 01H

The timing diagram for **OUT 01h** is shown below:

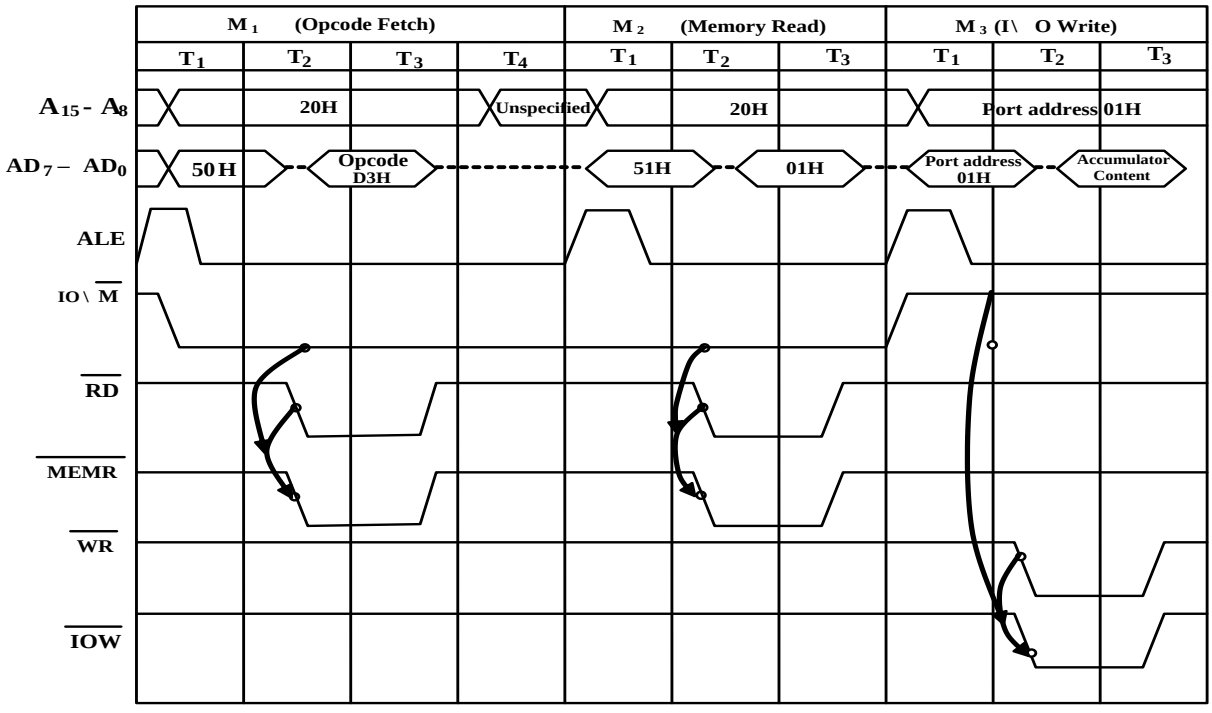


Fig: Timing Diagram for OUT 01 h

EXPLANATION:

- **The First Machine Cycle (M_1): Op-Code Fetch**

  The **8085** places the high order memory address **20H** on A_{15} –A_8 and the low order address **50H** on AD_7-AD_0. At the same time, **ALE** goes high and **IO**/$\overline{M}$ goes low. The **IO**\$\overline{M}$ low indicates that it is a memory related operation. At T_2, the microprocessor sends the $\overline{RD}$ control signal which is combined with **IO**\$\overline{M}$ to generate the $\overline{MEMR}$ signal and the processor fetches the instruction code D_3 using the data bus and instruction is decoded.

- **The Second Machine Cycle (M_2): Memory Read**

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

- **The Third Machine Cycle (M_3): I/O Write**

  In the third machine cycle, M_3 (**I\O** Write), the **8085** places the device address **01H** on the low order **(AD_7 – AD_0)** as well as the high order **(A_{15} – A_8)** address bus. The **IO**\$\overline{M}$ signal goes high to indicate that it is an **I\O** operation. At T_2, the accumulator contents are placed on the data bus **(AD_7 – AD_0)** followed by the control signal $\overline{WR}$. By adding the **IO**\$\overline{M}$ and $\overline{WR}$ signals, the $\overline{IOW}$ signal can be generated to enable an output device and data from accumulator are displayed on **O/P** device such as **LED**s.

Regular Examination: 2004 [Draw the Timing Diagram of OUT 23H Instruction]

Memory Address	Machine Code	Mnemonics	Memory Contents
2050H	D3	OUT 23H	2050: 1101 0011H = D3H
2051H	01	-	2051: 0010 0011H = 23H

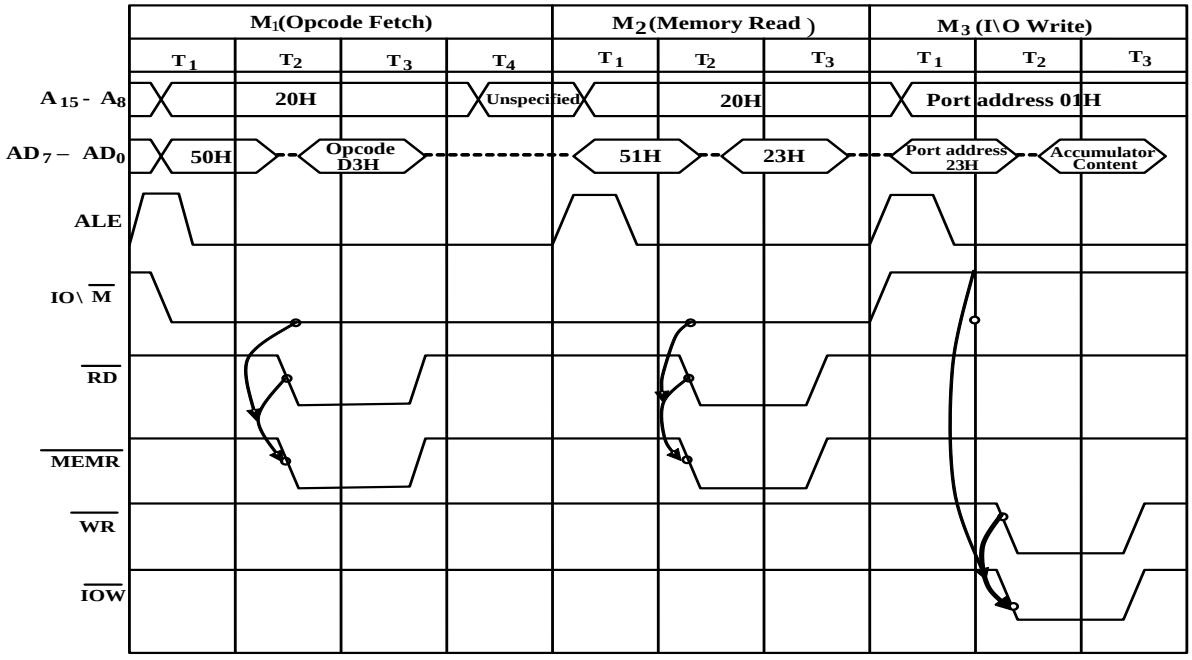


Fig: Timing Diagram of OUT 23 h

Fetch Execution Overlap

As we know that the specified task is performed during the execution cycle i.e. at the end of last **T**-state of the instruction cycle. But this may not be always true. This is just the generalization for easier purpose.

Let's consider the instruction **MOV R_d, R_s** which takes one machine cycle **M_1** and consists of **4 T**-states as shown in following timing diagram.

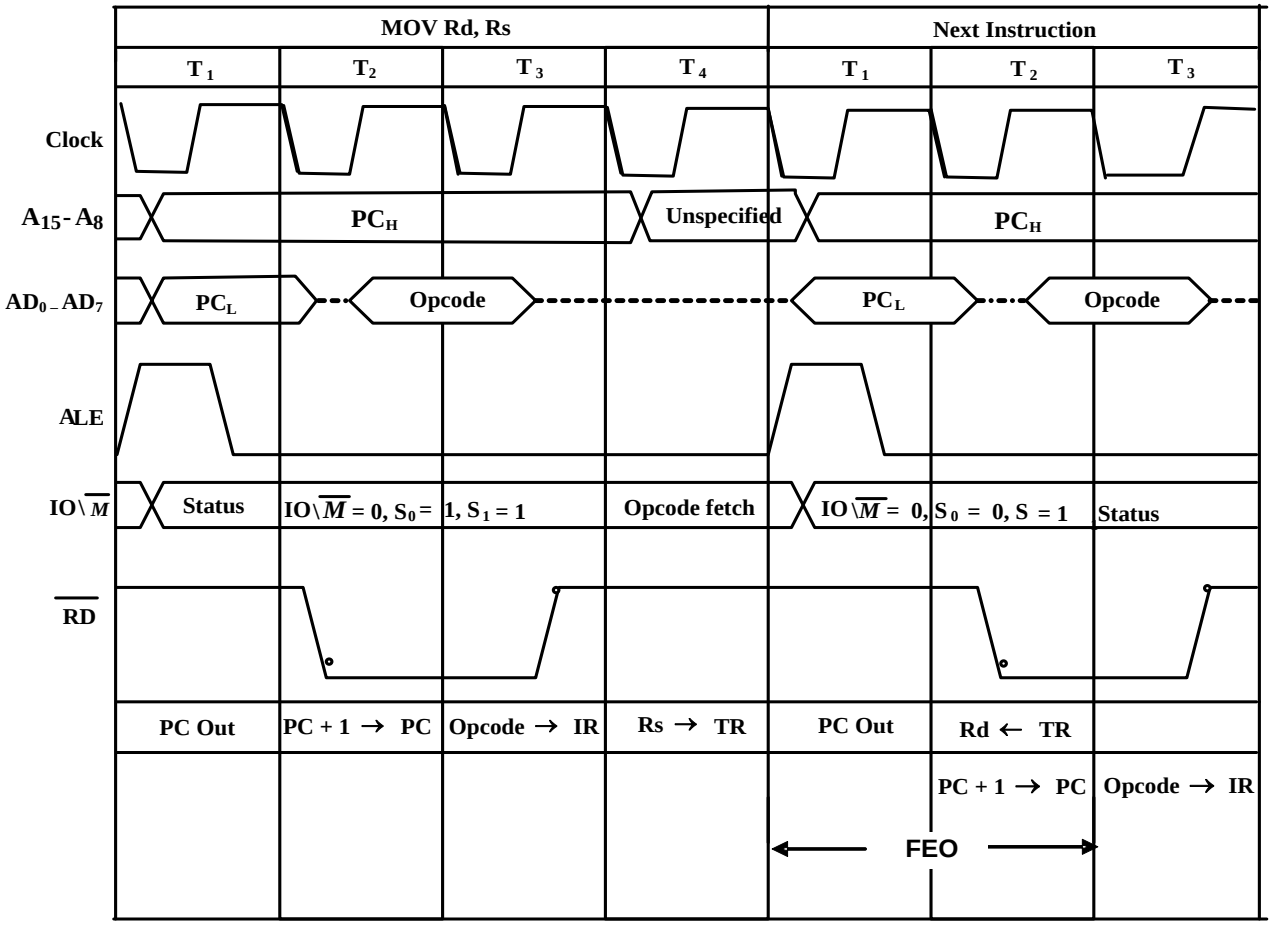


Fig: Fetch Execution Overlap

The **T_4**, **T**-state is used as instruction decoding cycle and least duration of **T_4** is used to perform the specified task. But for the **MOV R_d, R_s** instruction, at the end of **T_4**, only the contents of source register **(R_s)** is transferred to temporary register **(T_r).** The transferring of data from temporary register **(T_r)** to destination register **(R_d)** is done at the **T_2** state of first machine cycle next instruction. We know that the next immediate machine cycle is Op-code fetch as **MOV Rd, Rs** is one byte instruction. It is clear from the timing diagram that **T_1** and **T_2** of the fetch cycle of next instruction is used to fetch the Op-code as well as transferring of data from temporary register to destination register of previous instruction. This is called fetch execution overlap **(FEO).**

COUNTERS AND TIME DELAYS

Designing a counter is frequent programming application. Counters are used primarily to keep track of events; time delays are important in setting up reasonably accurate timing between two events. The process of designing counters and time delays using software instructions is far more flexible and less time consuming than the design process using hardware.

COUNTER

A counter is designed simply by loading an appropriate number into one of the registers and using the **INR** (Increment by One) or the **DCR** (Decrement by One) instructions. A loop is established to update the count and each count is checked to determine whether it has reached the final number; if not, the loop is repeated.

The flowchart shown below illustrates these steps. However, this counter has one major drawback; the counting is performed at such high speed that only the last count can be observed. To observe counting, there must be appropriate time delay between counts.

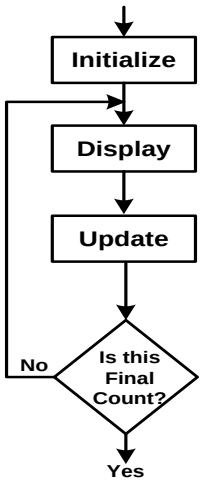


Fig: Flowchart of a Counter

TIME DELAY

The procedure used to design a specific delay is similar to that used to set up a counter. A register is loaded with a number, depending on the time delay required, and then the register is decremented until it reaches zero by setting up a loop with a conditional Jump instruction. The loop causes the delay, depending upon the clock period of the system.

TIME DELAY USING ONE REGISTER

A count is loaded in a register and the loop is executed until the count reaches zero. An example of set of instructions necessary to set up the loop is given below with an **8085**-based microcomputer with **2 MHz** clock frequency:

Label	Op-Code	Operand	Comments	T-States
	MVI	C, FFH	; Load Register C	7
LOOP:	DCR	C	; Decrement C	4
	JNZ	LOOP	; Jump Back to Decrement C	10/7

Delay Calculation:

Clock Frequency of the System **(f) = 2 MHz**
Clock Period **(T) = 1/f = 1/(2x10^{-6}) = 0.5 μs**
Time to Execute **MVI = 7 T-States x 0.5 = 3.5 μs**

To calculate the time delay in a loop, we must account for the **T**-states required for each instruction and for the number of times the instructions are executed in the loop. In **above** example, register **C** is loaded with the count **FFH** (**255$_{10}$**) by the instruction MVI, which is executed once and takes seven T-states. The next two instructions, **DCR** and **JNZ**, form a loop with a total of **14 (4+10) T**-states. The loop is repeated **255** times until register **C = 0.**

Now, the time delay in the loop **T_L** with **2 MHz** clock frequency is calculated as:

$$T_L = (T \times \text{Loop T-states} \times N_{10})$$

Where, **T_L** = Time Delay in the Loop
T = System Clock Period
N_{10} = Equivalent Decimal Number of the Hexadecimal Count Loaded in the Delay Register.

$\therefore$ $T_L = (0.5 \times 10^{-6} \times 14 \times 255)$
$= 1785$ **μs**
= 1.8 **ms**

Note: The **T**-states for **JNZ** instruction are shown as **10/7**. This is why the **8085** microprocessor requires **10 T**-states to execute a conditional Jump instruction when it jumps or changes the sequence of the program and **7 T**-states when the program falls through the loop (goes to the instruction following the **JNZ**). This difference can be accounted for the delay calculation by subtraction the execution time of three states. There for, adjusted loop delay is:

$T_{LA} = T_L - (3\ T\text{-states} \times \text{Clock Period})$
$= 1785$ **μs** **–** 1.5 **μs**
= 1783.5 **μs**

Now the total delay must take into account the execution time of the instructions outside the loop. In the above example, we have only one instruction (**MVI C**) outside the loop. Therefore, the total delay is:

Total Delay = Time to Execute Instructions Outside Loop + Time to Execute Loop Instructions
$T_D = T_O + T_{LA}$
$= (7 \times 0.5) + 1783.5 = 1787$ **μs**
= 1.8 **ms**

TIME DELAY USING REGISTER PAIR

The time delay can be considerably increased by setting a loop and using a register pair with a **16**-bit number (maximum **FFFFH**). The **16**-bit number is decremented by using the instruction **DCX**. However, the instruction DCX does not set the **Zero Flag** and without the test flags, Jump instructions cannot check desired data conditions. Additional techniques, therefore, must be used to set the **Zero Flag**.

The following set of instructions uses a register pair to set up a time delay.

Label	Op-Code	Operand	Comments	T-States
	LXI	B, 2384H	; Load BC with 16-bit count	10
LOOP:	DCX	B	; Decrement BC by one	6
	MOV	A,C	; Place Contents of C in A	4
	ORA	B	; OR (B) with (C) to Set zero Flag	4
	JNZ	LOOP	; If result = 0, Jump Back to LOOP	10/7

Delay Calculation:

The loop includes four instructions: DCX, MOV, ORA and JNZ and takes 24 clock periods for execution. The loop is repeated 2384H times, which is converted to decimal as:

$2384H = 2 \times (16)^3 + 3 \times (16)^2 + 8 \times (16)^1 + 4 \times (16)^0$
$= 9092_{10}$

If the clock period of the system = 0.5 **μs,** the delay in the loop T is:

$T_L = (0.5 \times 24 \times 9092_{10})$
$= 109$ ms (without adjusting for the last cycle)

Total Delay (T_D) $= 109$ ms + T_O
$= 109$ ms (The instruction LXI adds only 5 **μs**)

TIME DELAY USING A LOOP WITHIN A LOOP TECHNIQUE

A time delay similar to that of a register pair can also be achieved by using two loops; one loop inside the other loop as shown in example below.

Label	Op-Code	Operand	T-States
	MVI	B, 38H	7
LOOP2:	MVI	C, FFH	7
LOOP1:	DCR	C	4
	JNZ	LOOP1	10/7
	DCR	B	4
	JNZ	LOOP2	10/7

Delay Calculation:

The delay in LOOP1 is $T_{L1} = 1783.5$ **μs** as per the previous calculation. Now the calculation for the delay in LOOP2 is as follows:

$T_{L2} = 56(T_{L1} + 21$ T-states x 0.5 **μs)**
$= 56$ (1783.5 **μs** + 10.5 **μs**)
$= 100.46$ **ms**

3 CHAPTER

Assembly Language Programming

Contents:

- Computer Programs & Programming Languages
- Types or Levels of Programming Languages
- Compiler & Interpreter
- Assembly Language Programming
- Assembler Instruction Format
- Assembler & Its Types
- Assembler Directives
- Sample Assembly Language Program

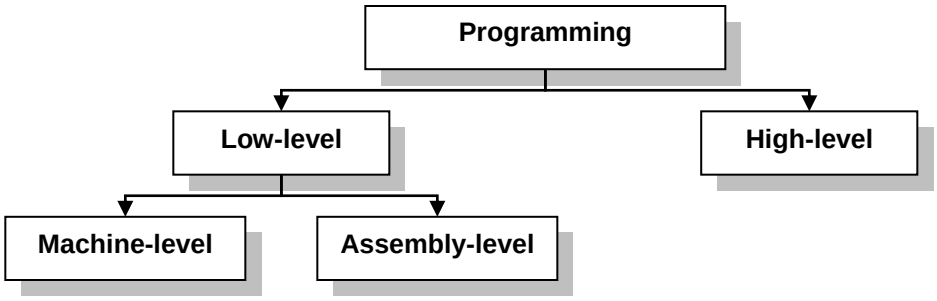
COMPUTER PROGRAM & PROGRAMMING LANGUAGES: A REVIEW

Computer program is a set of instructions that, when executed, causes the computer to behave in a predetermined manner. Without programs, computers are useless and can do nothing. Most people are confused that computers are intelligent devices and can do anything; but this concept is wrong. Computer can do anything, but only after it has appropriate program(s). Without programs, it can not even add two numbers.

Computer cannot understand human natural languages like English or Nepali. To instruct a computer to perform a certain job, we need languages which can be understood by the computer. The languages which are used to instruct the computer to do certain jobs are called "**Computer Programming Languages"**. There are many programming languages like **C, C++, Pascal, BASIC, FORTRAN, COBOL, LISP**, etc.

Types or Levels of Programming Languages

There are mainly two types of programming languages – **High-Level Programming Languages & Low-Level Programming Languages.** Low-level languages are closer to the hardware than high-level programming languages.



LOW-LEVEL LANGUAGE

Low-level language is a programming language in which each statement or instruction is directly translated into a single machine code. It is machine dependent, i.e. a particular low-level language works only for a certain machine.

MACHINE-LEVEL LANGUAGE

Machine language is a language that a computer actually understands. The least possible level at which we can program a computer is in its own native machine code, consisting of strings of **1**'s and **0**'s, and are stored as binary numbers. Thus, machine language is a sequence of instructions written in the form of binary numbers consisting of **1**'s and **0**'s to which the computer responds directly. The main advantage of machine languages is that they execute fast as the computer understands them and doesn't need translation into other forms. However, machine languages are difficult to write.

Example: To add two numbers **3** and **6** in machine language, program may be like –

11	**10111001**	**110**
3	machine code for addition (say)	**6**

There are many different instruction sets (machine codes) for each operation. It is difficult to remember them for programming.

Advantages of Machine Language

The **CPU** directly understands machine instructions, and hence no translation is required. Therefore, the computer directly starts executing the machine language instructions, and it takes less execution time.

Disadvantages of Machine Languages

Difficult to use: As computer instructions are written in binary forms, it is hard to write, understand, and debug.

Machine dependent: The code written for one processor may be incompatible for other. The programmer has to remember machine characteristics while developing a program. As the internal design of the computer is different across types, which in turn is determined by the actual design or construction of the **ALU**, **Control Unit** and size of the word length of the memory unit, the machine language also varies from one type of computer to another.

Difficult to debug and modify: Checking machine instructions to determine errors are about as tedious as writing the instructions. It is difficult to modify the program.

Assembly Language

Assembly language is a symbolic representation (called **mnemonics**) of machine code. They are close to machine code but the computer cannot understand them. The assembly-language program must be translated into machine code by a separate program called an **assembler.** The assembler program recognizes the character strings that make up the symbolic names of the various machine operations, and substitutes the required machine code for each instruction. The above program for addition two numbers in assembly language may look like –

11 ADD 110 → Suppose **ADD** is mnemonic for addition

Some of the examples of instructions for assembly language are as follows:

Code	**Meaning**
ADD	Addition
SUB	Subtraction
INR	Increase
DCR	Decrease
CMP	Compare

It is relatively easy for writing programs in assembly languages, but is slow in execution as it has to be converted into machine language before execution.

Advantages of Assembly Language

Writing a program in assembly language is more convenient than writing one in machine language. Instead of binary sequence, as in machine language, a program in assembly language is written in the form of symbolic instructions. This gives the assembly language program improved readability. Again, it is useful writing program in embedded system because it needs fewer codes than high level language. It is also used for device programming.

Disadvantages of Assembly Language

Hard to remember mnemonics: There are large numbers of mnemonics for a machine. It is difficult to remember all the mnemonics for writing assembly program

Machine dependent*:* Assembly language is specific to particular machine architecture. Thus, assembly language programs written for one processor will not work on a different processor if it is architecturally different.

Less efficient than machine language: The assembly language is to be translated into machine language before execution. Thus, a program written in assembly language is less efficient compared to an equivalent machine language program but more efficient than equivalent high level language program.

HIGH LEVEL LANGUAGE

A high-level programming language is a programming language that is more user-friendly, to some extent platform-independent and abstracts from low-level computer processor operations such as memory accesses. They are similar to natural languages (like English) and so are easy to write and remember. They are easy to learn and work but while executing, they have to be translated into assembly language and then to machine language. So it is slow in execution but is efficient for developing programs. **C, C++, Java, VB,** & **VB.net** are known as high level languages.

Advantages of High Level Language

Easy to use*:* The statements in high level language are much more like English language. Thus, they are easier to understand than those written in assembly and machine language.

Portability: High level programming languages can be run on different machines with little or no change. It is, therefore, possible to exchange software, leading to creation of program libraries.

Easy Debugging: Errors can be easily detected and removed.

Easy and Fast development of software: Since the commands of these languages are closer to the English language, software can be developed with ease and speed.

Disadvantages of machine language

More execution time: As the program written in high level language can not directly generate executable code, it has to be translated into assembly language and then to machine language. Thus, it takes more time for execution.

Needs own translator: Each language has its own translator to change high level language program into machine language

Compiler & Interpreter

High level source program must be translated first into a form the machine can understand. This is done by the software called compiler. The compiler takes the source code as input and produces the machine language code (object code) for the machine on which it is to be executed as output. During the process of translation, the compiler reads the source programs statement-wise and checks for syntax errors. In case of any error, the computer generates message about the error. Examples**: C, C++, Java, FORTRAN, Pascal** etc

An interpreter, like compiler, is also translator which translates high level language into a machine level language. The difference between compiler and interpreter is their working principle. The interpreter translates and executes the program line by line. Each time the program is executed; every line is checked for syntax error and then converted to the equivalent machine code. Examples: **QBASIC, PERL, PHP, ASP**

Compiler Vs Interpreter

Compiler	Interpreter
1) Compiler scans the entire program before translating it into machine code.	1) Interpreter translates and executes the program line by line.
2) Syntax errors are found only after the compilation of complete program.	2) Syntax errors can be trapped after translation of every line.
3) It takes less execution time	3) It takes more execution time

Assembly Language Programming

Programs in assembly language are represented by instructions that use **English Language Type** commands called **mnemonics**. It is more convenient to write the programs in assembly than in machine language. However a translator (assembler) must be used to convert the assembly language programs into binary machine language programs (object codes) so that the processor can execute them.

An assembly language program written for one microprocessor is not transferable to a computer with another microprocessor unless the tow microprocessors are compatible in their machine codes.

Example: The binary code **0011 1100** of the **8085** microprocessor is represented by the mnemonic **INR A**, which suggests the operation of incrementing the accumulator contents by one.

Tools for Developing Assembly Language Programs

The program development utilities enable the user to write, assemble and test assembly language programs; they include programs such as **Editor, Assembler, Linker (or Loader) and Debugger.** These utility programs enable the user to perform such functions as copying, printing, deleting and renaming files. Therefore they are necessary to develop assembly language programs.

EDITOR:

The editor is a program that allows the user to enter, modify and store a group of instructions or text under a file name. The editor programs can be classified in two groups: line editors and full-screen editors. Line editors, such as **EDIT** in **MS-DOS**, work with and manage one line at a time. Full screen editors (also known as word processor), such as **MS Word** and **WordPerfect**, manage the full screen or a paragraph at a time. To write text, the use must call the Editor under control of the operating system. The exit command of the Editor program will save the program on the disk under the file name and will transfer the program control to the operation system.

ASSEMBLER:

The assembler is a program that translates source code or mnemonics into the binary code, called object code, of the microprocessor and generates a file called the object file. This function is similar to manual assembly, whereby the user looks up the code for each mnemonic in the listing. The translation requires that the source program be written strictly according to the specified syntax of the assembler. The assembly language source program includes three types of statements, they are:

- The program statement in **8085** mnemonics that are to be translated into binary code.
- Comments that are reproduced as part of the program documentation.
- Directives to the assembler that specify items such as starting memory locations, label, definitions and required memory spaces for data

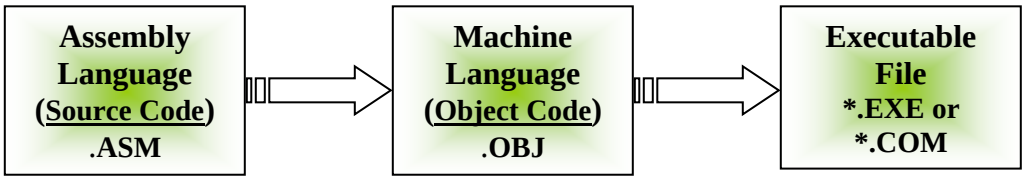


Fig: Process of Compilation and Linking Assembly Language

An assembler is a tool for developing assembly language program. In addition to translating mnemonics, the assembler performs various functions as given below:

- Translates mnemonics into binary code with speed and accuracy, thus elimination human errors in looking up the codes.
- Assigns appropriate values to the symbols used in program. This facilitates specifying jump locations.
- Easy to insert or delete instructions in a program; the assembler can reassemble the entire program quickly with new memory locations and modified addresses for jump locations. This avoids rewriting the program manually.
- Checks syntax errors, such as wrong labels and expressions, and provides error messages. However it cannot check logic errors in a program.
- Reserve memory locations for data or results.
- Provide files for documentation.
- A debugger program can be used in conjunction with the assembler to test and debug an assembly language program.

Loader (Linker):

The loader is a program that takes the object file generated by the assembler program and generates a file in binary code called the **COM** file or the **EXE** file. The **COM** or **EXE** file is the only executable file i.e.; the only file that can be executed by the microcomputer. To execute the program, the **COM** file is called under the control of the operating system and executed. In different assemblers, the **COM** file may be labeled by other names.

Debugger:

The Debugger is a program that allows the user to test and debug the object file. The user can employ this program to perform the following functions:

- Make changes in the object code.
- Examine and modify the contents of memory.
- Set breakpoints, execute a segment of the program and display register contents after the execution.
- Trace the execution of the specified segment of the program and display the register and memory contents after the execution of each instruction.
- Disassemble a section of the program i.e. convert the object code into the source code or mnemonics.

Assembler Instruction Format

Microprocessors recognize and operate in binary numbers. However, each microprocessor has its own binary words, meanings and language. The words are formed by combining a number of bits for a given machine. The basic terms related to computer language are explained shortly below.

- **Word (Word Length):** The word (or word length) is defined as the number of bits the microprocessor recognizes and processes at a time. The word length ranges from **4**-bits for a small microprocessor based systems to **64** bits for high speed large computers.
- **Byte:** A byte is defined as a group of **8**-bits. For example a **16**-bit microprocessor has word length equal to **2** byte. One byte constitutes one character.
- **Nibble:** The term nibble is defined as a group of **4**-bits. A byte has two nibbles

Each machine has its own set of instructions based on the design of its **CPU** or of its microprocessor. To communicate with the computer, one must give instructions in binary sets of **0**s and **1**s.

- **Op Code:** An instruction is a command to the microprocessor to perform the given task. It consists of task to be performed by the microprocessor called the operation code (Op-code). Thus the Op-code field specifies how data is to be manipulated.
- **Operand:** Instruction also contains the data to be operated on which is called operand. The operand (or data) may be **8-bit** (or **16-bit**) data, an internal register, a memory location or an **8-bit** (or **16-bit**) address. In some instruction the operand is implicit.
- **Mnemonics:** Each microprocessor has a fixed set of instructions in the form of binary patterns called a machine language. However it is difficult for human beings to communicate in the language of **0**'s and **1**'s. Therefore, the binary instructions are given abbreviated names, called mnemonics, which forms the assembly language for a given microprocessor.

A typical assembly language programming statement is divided into four parts called fields: **Label, Operation Code (Op code), Operand** and **Comments.** These fields are separated by delimiters as shown in table below:

Delimiter	Placement
1. Colon	After label (Optional)
2. Space	Between an op code and operand
3. Comma	Between two operands
4. Semicolon	Before the beginning of a comment

As an example, a typical assembly language statement is written as follows:

Label	Opcode	Operand	Comments
START:	LXI	SP, 20FFH	;Initialize Stack Pointer

In above sample assembly language program delimiters include the colon following **START**, the space following **LXI**, the comma following **SP**, and the semicolon preceding the comments.

Types of Assembler

One Pass & Two Pass Assembler

One Pass Assembler goes through the assembly language program once and translates the assembly language program. This assembler has the program of defining forward references. This means that a **JUMP** instruction using an address that appears later in the program must be defined by the programmer after the program is assembled.

Two Pass Assembler scans the assembly language program twice. In the first pass the assembler generates the table of the symbols. A symbol table consists of labels with addresses assigned to them. In this way labels can be used for **JUMP** statements and no address calculation here to be done by the user. On the second pass, the assembler translates the assembly language program into the machine code. The two pass assembler is hence more efficient and easier to use.

One Pass Assembler	Two Pass Assembler
1. An assembler which goes through an assembly language program only once and produces its object code is known as one pass assembler. 2. In assembly language program, labels are used to point forward and backward references. Since the one pass assembler goes through an assembly language program once, such an assembler must have some means to take forward reference into consideration. 3. Difficult to use hence undesirable.	1. An assembler which goes through an assembly language program twice to produce object code is known as two pass assembler. 2. In the first pass, the assembler generates the symbol table consisting of labels with address assigned to it. In second pass, the assembler translates the assembly language program into object \ machine code using the symbol table. 3. Easier to use, hence desirable.

Cross Assembler

This type of assembler is typically resident on the processor and assembles programs for another for which it is written. The cross assembler program is written in a high level language so that it can run on different types of processor that understands the same high level language.

For example, the **8085** cross-assembler from **2500AD Software Inc.** has two programs: one is as assembler named **X8085** and the other is a linker with the file name **LINK**. After assembling a program, the **HEX** file can be directly transferred to **R/W** memory of **8085** single-board microcomputer by using a **Download Program**. Thus programs and/or hardware-related laboratory experiments can be easily performed. An assembly language program using cross-assembler has following steps of operations:

- **Call an Editor** (Such as **EDIT** in **MS-DOS** & **WordPerfect** or **MS-Word** in **Windows**) and write an assembly language program in **8085** mnemonics. This is called source file.

- **Call a Cross-Assembler** (Such as **X8085**) and assemble the source file. The **X8085** generates an intermediate binary file called an object file and provides a list of errors. Reassembling the program would give a message of zero errors. The cross assembler also generates a list file that includes memory address, machine codes in **HEX, Labels** and **Comments**. This file is used primarily for documentation.

- **Call a Link** program and use the intermediate binary file (Object Code) to generate either an Executable file or **HEX** file.

- **Execute** the program on the single-board machine.

After the end of the assembly process, we will have the following files on our **PC** disk.

- **ASM File**: This is the source file written by the user using an **Editor**. The filename can be one to eight characters long with an extension of a maximum of three characters. The filename and the extension are separated by a dot, e.g. **Microprocessor.ASM**, the extension **ASM** suggests that this is an assembly language file.

- **OBJ File**: This is the intermediate binary file generated by the **Cross-Assembler**.

- **LST File**: This is the list file generated by the **Assembler Program** for documentation purposes. It contains memory locations, hex code, mnemonics and comments.

- **HEX File**: This is generated by the **Link Program** and contains program code in hexadecimal notations. This file can be used for debugging the program and to transfer files from one system to another.

- **COM File**: This is the executable file generated by the **Link Program**, and it contains binary code. However, this file cannot be executed on the **PC**. This type of a file can be executed if it has to write an assembly language program for the microprocessor in the **PC**. However, in such a situation cross-assembler will be not in use. We have to use the assembler for the **PC** microprocessor.

Resident Assembler

This type of assembler assembles programs for a processor in which it is resident. The resident assembler may slow down the operation of processor on which it runs.

Meta Assembler

This type of assembler can assemble programs for many different types of processors. The programmer usually defines the particular processor being used.

Macro Assembler

Macro assembler is a type of assembler that translates a program written in micro language into the machine language. A macro language is the one in which all the instruction sequence can be defined using macros. A macro is an instruction sequence that appears repeatedly in program assigned with a specific name.

Macro assembler lets the programmer define all instruction sequence using macros. By using macros, the programmer can assign a name to an instruction sequence that appears repeatedly in a program. The programmer can thus avoid writing an instruction sequence that is required many times in a program by using macros. The macro assembler replaces a macro name with the appropriate instruction sequence each time it encounters a macro name.

As in subroutine program, there is no need to use **CALL** and **RET** instruction to execute and to resume program. A macro does not cause the program execution to branch out of the main program. Each time a macro occurs, it is replaced with the appropriate instruction sequence in the main program. Typical advantages of using macros are shorter source program and better program documentation.

Conditional macro assembly is very useful in determining whether or not an instruction sequence shall be included in the assembly depending on a condition that is true or false. If two different programs are to be executed repeatedly based on a condition that can be either true or false, it is convenient to use conditional macros. Based on each condition, a particular program is assembled.

For example consider a sequence of instruction such as:

```
IN AX, PORT
ADD AX, BX
OUT PORT, AX
```

If above sequence of instructions are grouped and assigned a specific name '**Addition**' (called macro name '**Addition**'), the whole macro becomes:

```
Addition {
        IN AX, PORT
        ADD AX, BX
        OUT PORT, AX
        }
```

When above instruction sequence is to be executed repeatedly macro assemblers allow the macro name only to be typed instead of all instructions provided the macro is defined.

Assembler Directives

Assembler directives are the instructions to the assembler concerning the program being assembled. They are also called pseudo instructions or pseudo codes. They are neither translated into machine code nor assigned any memory location. They just provide the direction to the assembler.

It performs the operations such as:

- **Telling the assembler to place the segment in standard order.**
- **Indicating starting and ending of procedures.**
- **Indicating ending of entire program.**
- **Selecting the size of segments.**

Some of the important assembler directives are:

1. Data Definition Directives

Data definition directive creates storage for data. There are various types of data definition directives summarized below.

- **DB- Define Byte (1 Byte)**: The **DB** directive creates storage for a byte or group of bytes in memory. It is used to declare a byte type variable.

  Example: Count DB? (Initialization)
  Data 1 DB 10H
  List DB 10H, 20H, 30H

- **DW-Define Word (2 byte)**: The **DW** directive creates storage for a word or list of words. It is used to declare a word type variable.

  Example: **Data 1 DW 1234H**
  List DW 1234H, 6678H, 7981H

- **DD- Define Double Word** (**4 byte**): The **DD** directive creates storage for **32** bits double word variable. It is used to declare double word type variable.

  Example: **Data 1 DD 12345678H**

- **DQ- Define Quad Word (8-byte)**: **DQ** directive creates storage for **8** byte or group of **8** bytes. It is use to declare quad word type variable.

  Example: **Data1 DQ 123456789123456H.**

- **DT-Define Ten Bytes**: **DT** directive creates storage for ten bytes. It is used to declare ten byte type variable.

  Example: **List DT 123456789A1234564455H**

2. DUP-Directive (Duplicate)

The **DUP** directive can be used to initialize several locations to zero. For example the statement **START DW 4DUP (0)** reserves four words starting at the offset **START** in **DS** and initializes them to zero. The **DUP** directive can also be used to reserve several locations that need not be initialized. A question mark should used with **DUP** in this case. For example the statement **BEGIN DB100 DUP (?)** reserves **100** bytes of uninitialized data space to an offset **BEGIN** in **DS**.

3. EQU-Directive (Equate)

The **EQU** directive is used to give a name to some value or symbol. Each time the assembler finds the given name in the program, it will replace the name with the value or symbol equated with that name. Example **count EQU 10H.**

4. ORG-Directive (Origin)

The **ORG** directive allows setting location counter to desired value. For example **ORG 200H** tells assembler to set location counter to **2000H.**

5. ASSUME-Directive

The **ASSUME** directive tells the assembler what names have been used for used for the code, data, extra and stack segment. It identifies the segment name used for code, data and stack segment. Example: **ASSUME CS: CODE, DS: DATA, SS: STACK.**

6. MODEL-Directive

The **MODEL** directive selects standard memory model for the program. It determines the way segments are linked together as well as maximum size of each segment.

Model	Description
Tiny	Code and Data segment together may not be greater than 64K
Small	Neither Code or data may be greater than 64K
Medium	Only the code may be greater than 64K.
Compact	Only the data may be greater than 64K.
Large	Both code and data may be greater than 64K.
Huge	All available memory may be used for code and data.

7. Program Segment-Directive

The program segment directives are:

- **.STACK**: The stack directive sets the size of the program stack which may be any size up to **64K**. The stack segment is addressed by **SS** and **SP** registers.
- **.CODE**: The code directive identifies the part of the program that contains the declaration and initialization of all variables. This segment is addressed by **DS** and all other register except **IP**, **SP** and **BP**.
- **.DATA**: The data directive identifies the start of the data segment.
- **.DOSSEG**: The **DOSSEG** segment tells the assembler to place the segments in order basically code, data and stack.

8. Procedure-Directive

A procedure is the group of related program instructions with common function or task.

- **PROC Directive**: It is used to identify the start of the procedure.
- **ENDP Directive**: It indicates the end of the procedure.
- **ENDS Directive**: It indicates the end of the segment.
- **END Directive**: The end directive terminates the assembly language program and is placed at the end of the program. Any instruction after this is ignored.

Sample Assembly Language Program for 8086 Microprocessor

The sample program given below, when assembled and executed, exchanges the contents of two variables. The following source program is created using a text editor.

```
TITLE: Exchange of Two Variables
DOSSEG
.MODEL SMALL
.STACK 100H
.CODE

Main PROC
    MOV AX, @Data       ; Initialize DS Register
    MOV DS, AX

SWAP;
    MOV AL, Value1      ; Load the AL Register
    XCHG Value2, AL     ; Exchange AL and Value2
    MOV Value1, AL      ; Move AL back into value1
    INT 21h

Main ENDP
.Data
Value1      DB 0Ah
Value2      DB 14h
END Main
```

Program Instruction

The first two instructions in the sample program initialize the data segment (**DS**) register.

```
MOV AX, @ Data
MOV DS, AX
```

The name **@ Data** is a standard **MASM** identifier that stands for the location of the data segment. By setting **DS** equal to the start of the data segment, we make it possible for **MASM** to locate variables correctly. The next three instructions exchange the contents of **value1** and **value2**.

```
MOV AL, Value1
XCHG Value2, AL
MOV Value1, AL
```

4
CHAPTER

Bus Structure & Memory Devices

Contents:

- Bus Structure
- Memory & Its Types
- Memory Interfacing
- Address Decoding

BUS STRUCTURES

A microprocessor unit basically performs four operations; memory read, memory write, **I\O** read and **I\O** write. These operations are part of communication between **MPU** and peripheral devices.

A communication includes identifying peripheral or memory location, transfer of data and control functions. These are carried out using address bus, data bus and control bus respectively. All the buses together are called the system bus.

DATA BUS

The data bus provides a path for data flow between system modules. It consists of a number of a separate line generally **8, 16, 32** or **64**. The number of lines is referred as width of the data bus. A single line can only carry one bit at a time, thus the number of lines determines how many bits can be transmitted at a time. The width of the data bus is a key factor in determining the overall systems performance.

- **Asynchronous Data Bus**

  In an Asynchronous data bus the timing is maintained in such that occurrence of one event on the bus follows and depends on the occurrence of previous events.

  Let us consider an example of simple memory read operation for an Asynchronous bus where the events occur as follows.

  - The **CPU** places the memory read and address signals on the bus.
  - After allowing for these two signals to stabilize 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 **(SSYNC)** signal.

  Thus at a certain time it cannot be identified when there is data or an address on the bus.

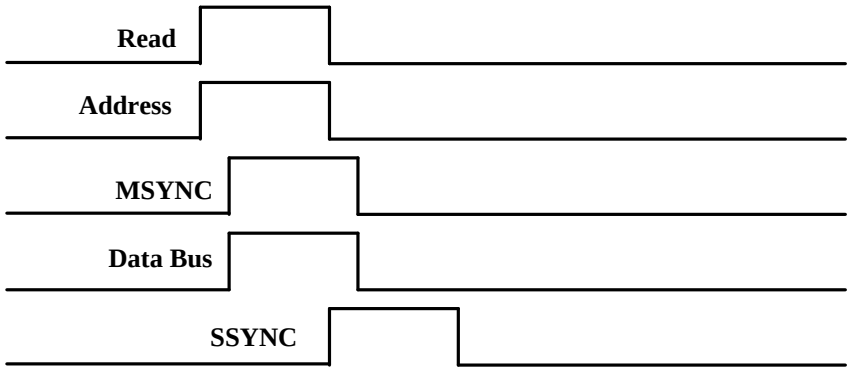


Fig: Asynchronous Bus

- **Synchronous Data Bus**

A synchronous bus has its events tied by a clock. The clock transmits a regular sequence of a **0**'s and **1**'s of equal duration. A single **1-0** transmission is called clock cycle or bus cycle. All other devices work with the clock cycle. The events start at the beginning of the clock cycle. A memory read in case of synchronous bus progresses as follows:

- 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 acknowledgement signals on the buses.

In a synchronous bus all the devices are tied to a fixed rate.

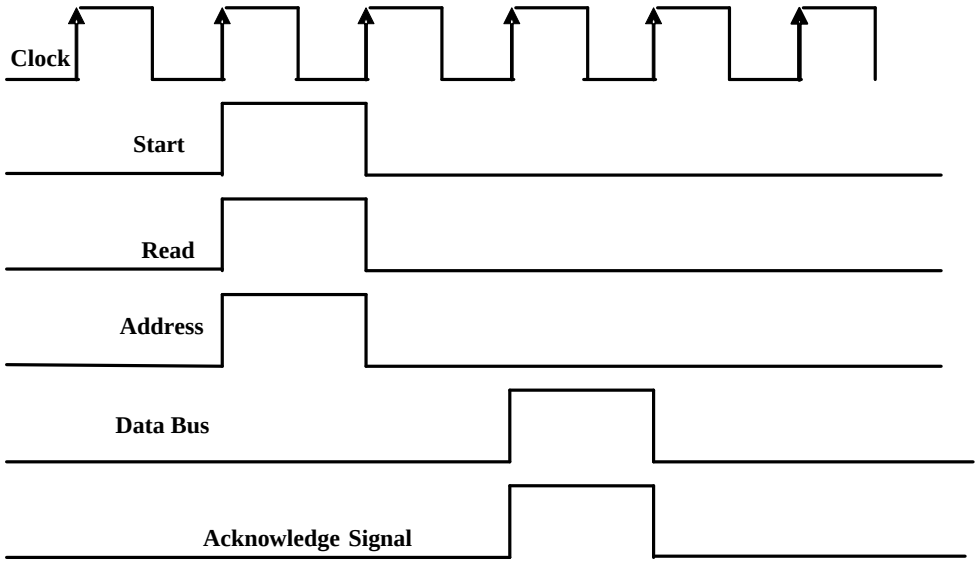


Fig: Synchronous Bus

Address Bus

The address bus is used to designate the source and destination of data in data bus. In a computer system, each peripheral or memory location is identified by a binary number called address.

The width of the address bus determines the maximum possible memory capacity of the system. The address bus is used to address **I\O** ports. Usually higher order bits are used to select particular modules and lower order bit is used to select a memory location or **I\O** port within a module.

Control Bus

The control bus is a group of lines used to control the data address bus. Since the bus is shared by all the components of the microcomputer system there must be some control mechanism to distinguish between data and address. The timing signals indicate the validity of data and address information while command signal specify operations to be performed. Some of the control signals are:

- **Memory Write**
- **Memory Read**
- **I\O Read**
- **I\O Write**
- **ALE**
- **Interrupt Request**
- **Interrupt Acknowledge**

Memory & Its Types

Memory is a collection of cells capable of storing binary information. In addition to these cells, memory contains electronic circuits for storing and retrieving the information. Memory is used in many different parts of a modern computer, providing temporary or permanent storage for substantial amount of binary information. Memory locations are identified by an address. The address bus of a microprocessor defines the maximum amount of memory a microprocessor can address.

Microcomputer memory is categorized as:

- **Processor Memory**
- **Primary or Main Memory**
- **Secondary Memory**

Processor Memory

Processor memory is a group of register along with processor. They temporarily hold the data during computation since processor and register are fabricated with same technology and they have same speed.

Separate memory is kept alongside the processor to keep the most frequently needed information. This memory is known as cache memory.

Primary Memory

This is the memory the microprocessor uses in executing and storing programs. Usually the size of the primary memory is larger and speed is slower than the processor memory. Primary memory can be categorized as:

- **Random Access Memory (RAM)**
- **Read Only Memory (ROM)**

Random Access Memory

A random access memory (**RAM**) is a collection of binary storage cells together with associated circuits needed to transfer information into and out of the cells configurations used in such a way that memory cells can be accessed to transfer information to or from any desired location with the access taking the same time regardless of location.

Thus **RAM** is read\write memory in which data can be written into and read from any selected address in any sequence. A **RAM** is typically used for short term data storage because it cannot retain stored data when power is turned off.

Properties of RAM

- Integrated circuit **RAM** may be either static or dynamic
- Stores binary information in the form of electric charges on capacitors.
- Data can be accessed randomly.
- Read and write operation can be performed.
- Volatile; lose stored information when power is turned off.

Types of RAM

There are two types of **RAM**

- **Static Random Access Memory (SRAM)**
- **Dynamic Random Access Memory (DRAM)**

Static Ram (SRAM)

SRAM consists of internal flip-flops and latches as a storage element that stores binary information as long as dc power is applied. It stores bit as a voltage. Each memory cell requires six transistor, they have low packaging density but high speed and consume more power. **SRAM** is easier to use and has shorter read and write cycles.

Dynamic Ram (DRAM)

DRAM consists of capacitors as a storage element and cannot retain data very long without the capacitor being recharged by a process called refreshing. This is done by cycling through words every few milliseconds, reading them and rewriting them. **DRAM** offers reduced power consumption and large storage capacity. The disadvantage is that the storage capacitor cannot hold its charge over an extended period of time and will lose the stored data bit unless its charge is refreshed periodically.

Read Only Memory

A read only memory (**ROM)** is a programmable logic device which contains permanently or semi permanently stored data which can be read from memory but either cannot be changed at all or cannot be changed without specialized equipment. The information must be specified by the designer and is them embedded into the **ROM** to form the required interconnection or electronic device pattern. A **ROM** stores data that are used repeatedly in system application and operation. **ROM**'s retain stored data when the power is off and therefore non volatile memories.

Types of RAM

The different types of **ROM** are described below:

- **The Masked ROM**
- **Programmable ROM (PROM)**
- **Erasable PROM (EPROM)**
  - **Ultraviolet Erasable PROM**
  - **Electrically Erasable PROM**

The Masked ROM

The mask **ROM** is permanently programmed during the manufacturing process to provide widely used standard functions, such as popular conversions or to provide user defined functions. Once the memory is programmed it cannot be changed. Generally manufacturers use this process to produce **ROM** in large numbers.

Programmable ROM

These are unprogrammed **ROM**. A **PROM** uses some type of fusing process to store bits, in which a memory link is burn open or left intact to represent a **0** or a **1**. The fusing process is irreversible, once a **PROM** is programmed, it cannot be changed.

Erasable PROM

An **EPROM** is an erasable **PROM**. An **EPROM** can be reprogrammed if an existing program in the memory array is erased first.

The basic types of erasable **PROM** are ultraviolet erasable **PROM** (**UVEPROM**) and the electrically erasable **PROM** (**EEPROM**).

Ultraviolet Erasable PROM

In **UVEPROM**, the programming process causes electrons to be removed from the floating gate. Erasure is done by exposure of the memory array chip to high intensity ultra violet radiation through the quartz window on top of the package. The positive charge stored on the gate is neutralized after several minutes to an hour of exposure time. The **UVEPROM** is programmed by inserting the chip into a socket of the **PROM** programmer and providing addresses. The programming time varies from **1-2** minutes.

Electrically Erasable PROM

EEPROM, an electrically erasable **PROM** can be both erased and programmed with electrical pulses. Since it can be both electrically written into and electrically erased, the **EEPROM** can be rapidly programmed and erased in circuits for reprogramming.

SECONDARY MEMORY

Secondary memories are storage devices. These device have high data holding capacity and can hold huge programs such as compilers and data based management system that are not needed by the processor frequently. They are slow and have larger size. The secondary memories are also referred to as auxiliary or back up storage.

DIFFERENCES

DRAM	SRAM
1. **DRAM** stores a data bit in a small capacitor. 2. **DRAM** cannot retain data very long without the capacitors being refreshed by a process called refreshing. 3. Data access time is greater than **SRAM**. 4. **DRAM** can store much more data for a given size and cost. 5. Data is kept as a charge. 6. It has high density.	1. **SRAM** uses flip-flops as a storage element. 2. Data can be stored indefinitely as long as dc power is applied. 3. Data can be read much faster from **SRAM's**. 4. **SRAM** can store less data than **DRAM** for a give physical size and cost. 5. Data is kept as a voltage. 6. It has low density.
RAM	**ROM**
1. **RAM** is a type of memory in which all addresses are accessible in an equal amount of time and can be selected in any order for read and write operation. 2. A **RAM** stores currently used data. 3. A **RAM** has both read and writes capability. 4. **RAM**, lose stored data when power is turned off so they are volatile.	1. **ROM** is a type of memory in which data are stored permanently or semi permanently. 2. **ROM** stores data that are repeatedly in used system applications. 3. Data can be read from a **ROM**, but there is no write operation. 4. Non volatile; data is not loosed even if power is turned off.
Primary Storage	**Secondary Storage**
1. This is the memory the microprocessor uses in executing and storing program. 2. Smaller in size so holds the data and programs currently in use. 3. The microprocessor can directly communicate with primary storage. 4. It has to communicate with microprocessor that has high speed and is expensive. 5. Generally volatile; data cannot be retained when power is turned off. 6. **RAM** and **ROM** are primary storage.	1. This is the memory which stores all the data and programs. 2. Larger in size and hence holds larger data files and huge programs. 3. The microprocessor cannot directly execute or process program stored in this memory. 4. It has low speed and thus it is cheaper than primary memory. 5. Non volatile; data can be retained even after power is turned off. Magnetic disk and tapes are secondary storage.

MEMORY INTERFACING

Memory is an essential component of a microcomputer system; it stores binary instructions and data for the microprocessor. To communicate with memory, the **MPU** should be able to

- **Select the chip,**
- **Identify the register, and**
- **Read from or write into the register.**

In a memory chip, all registers are arranged in a sequence and identified by binary numbers called memory address. Each register has a group of flip-flops or field-effect transistors that store bits of information; these flip-flops are called memory cells. The number of bits stored in a register is called a memory word.

The **MPU** uses its address bus to send the address of a memory register and uses the data bus and control lines to read from or write into that register.

REQUIREMENT OF MEMORY CHIP

- A memory chip requires address lines to identify a memory register. The number of address lines required is determined by the number of registers in a chip. (2^n = Number of registers where n is the number of address lines.)
- A memory chip requires a **Chip Select ($\overline{CS}$)** signal to enable a chip. The remaining address lines of the microprocessor can be connected to the **$\overline{CS}$** signal through an interfacing logic.
- The address lines connected to **$\overline{CS}$** select the chip, and the address lines connected to the address lines of the memory chip select the register. Thus the memory address of a register is determined by the logic levels (**0/1**) of all the address lines (including the address lines used for **$\overline{CS}$**)
- The control signal Read (**$\overline{RD}$**) enables the output buffer and the data from the selected register are made available on the output lines. Similarly the control signal Write (**$\overline{WR}$**) enables the input buffer and data on the input lines are written into memory cells.

A model of a typical memory chip representing the above requirements is shown below.

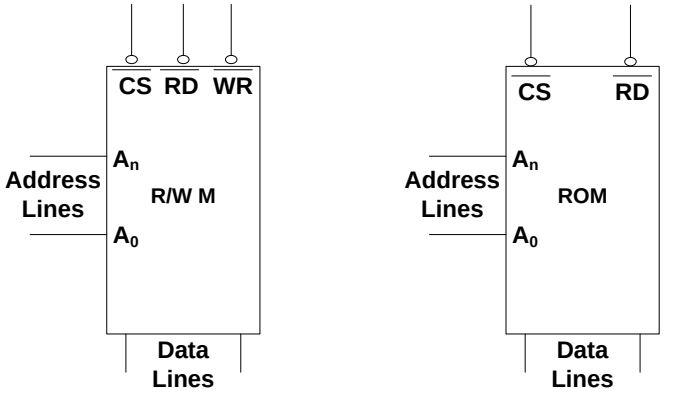


Fig: (a) R/W Memory Model (b) ROM Model

Memory Map & Addresses

In an **8-bit** microprocessor system like **8085, 16** address lines are available for memory. This means it is a numbering system of **16** binary bits and is capable of identifying 2^{16} **= 65,536** memory registers, each register with a **16-bit** address. The entire memory addresses can range from **0000** to **FFFF** in **HEX**, A memory map is a pictorial representation in which memory devices are located in the entire range of addresses. Memory addresses provide the locations of various memory devices in the system and the interfacing logic defines the range of memory addresses for each memory device.

Address range of a memory chip can be easily changed by modifying the hardware of the chip select line, which is generated by keeping the higher order address lines of the microprocessor at fixed logic levels. The hardware to generate the chip select address could be inverters and **NAND** gates or decoder logic.

Memory & Instruction Fetch

The primary function of memory is to store instructions and data and to provide that information to the **MPU** whenever the **MPU** request it. The **MPU** requests the information by sending the address of a specific memory register on the address bus and enables the data flow by sending the control signal as illustrated in the following example.

The instruction code **0100 1111 (4FH)** is stored in memory location **2005H**. Illustrate the data flow and list the sequence of events when the instruction code is fetched by the **MPU**

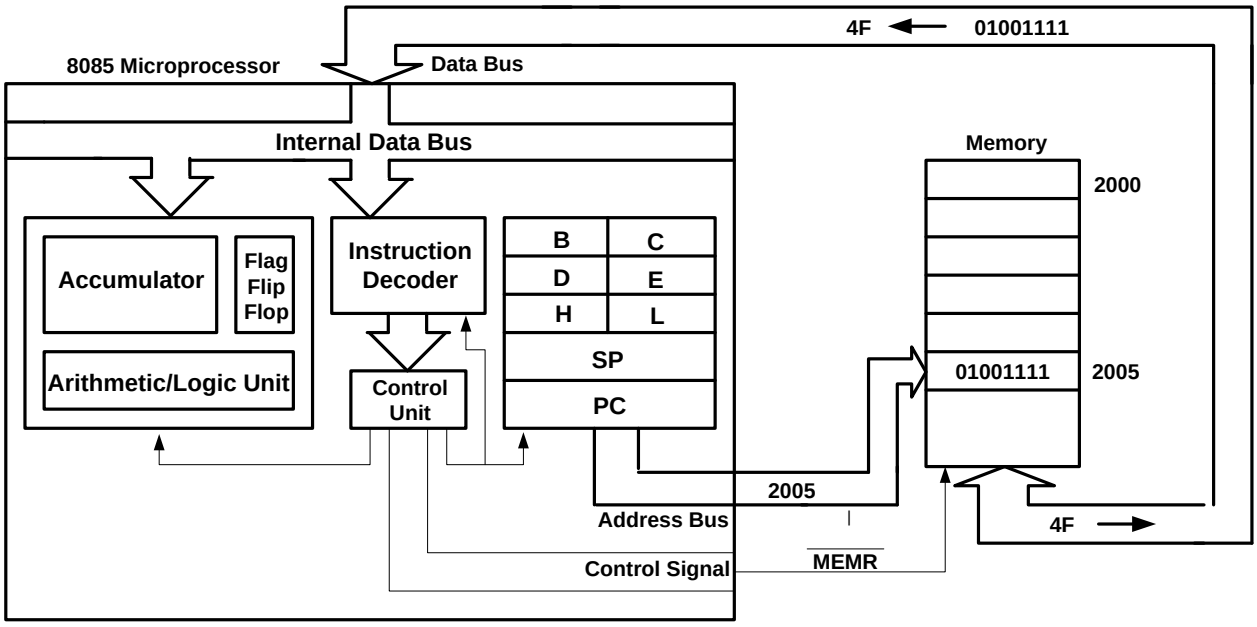


Fig: Instruction Fetch Operation

To fetch the instruction located in memory location **2005H**, the following steps are performed:

Step 1: The Program Counter (**PC**) places the **16**-bit address **2005H** of the memory location on the address bus.
Step 2: The control unit sends the Memory Read control signal (**MEMR**, active low) to enable the output buffer of the memory chip.
Step 3: The instruction **4FH** stored in the memory location is placed on the data bus and transferred (copied) to the instruction decoder of the microprocessor.
Step 4: The instruction is decoded and executed according to the binary pattern of the instruction.

While executing a program, the microprocessor needs to access memory quite frequently to read instruction codes and data store in memory; the interfacing circuit enables that access. Memory has certain signal requirements to write into and read from its registers. Similarly, the microprocessor initiates a set of signals when it wants to read from and write into memory. The interfacing process involves designing a circuit that will match the memory requirements with the microprocessor signals.

Thus **Memory Interfacing** is the process of designing appropriate circuit arrangement for enabling the microprocessor to access any register inside any of the many memory chips using unconflicting appropriate address for fetching instruction codes and data from that memory location into the microprocessor.

The primary function of memory interfacing is that the microprocessor should be able to read from and writhe into a given register of a memory chip. To perform these operations, the microprocessor should

- Be able to select the chip.
- Identify the register.
- Enable the appropriate buffer**.**

Example 1: Illustrate the memory address range of the chip with 1K (1024×8) bytes of memory, shown in figure below and explain how the range can be changed by modifying the hardware of the Chip Select line.

Explanation:
- Given Memory Size: **1** Kilobyte = **1024** Bytes
- Therefore the memory chip has **1024** registers.
- Number of address lines **(n) = ln1024/ln2 = 10(i.e. A_9-A_0)**, which is required to identify the registers.
- The remaining six address lines (i.e. **A_{15}-A_{10}**) of the microprocessor are used for the Chip - Select (**$\overline{CS}$**) signal. In the given figure below, the memory chip is enabled when the address lines (i.e. **A_{15}-A_{10}**) are at logic **0**. The address lines **A_9-A_0** can assume any address of the **1024** registers, starting from all **0**s to all **1**s as shown next.

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0000H**
Chip Select Logic Levels						↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	
						1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**03FFH**

- The memory addresses now range from **0000H** to **03FFH.**
- The address lines **A_{15}-A_{10}**, which are used to select the chip, must have fixed logic levels and these lines are called high-order address lines. The address lines **A_9-A_0**, which are used to select a register, are called low-order address lines and they can be assigned logic levels from all **0**s to all **1**s and any in between combination.
- By combining the high order and the low order address lines, we can specify the complete memory address range of a given chip.
- The memory addresses of the **1K** chip in given figure can be changed to any other location by changing the hardware of the **CS** line. For example, if **A_{15}** is connected to the **NAND** gate without and inverter, the memory addresses will range from **8000H** to **83FFH** as shown below.

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**8000H**
Chip Select Logic Levels						↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
						1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**83FFH**

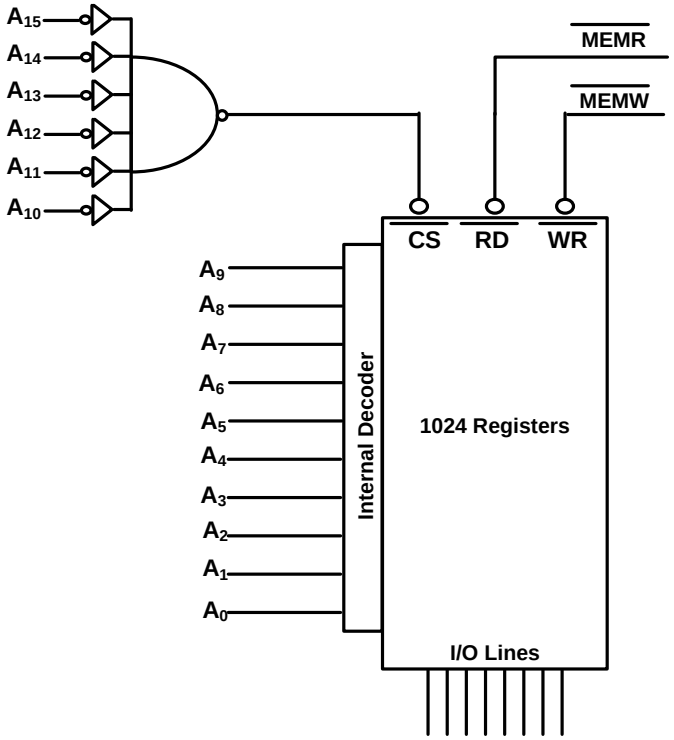


Fig: Memory Address Range: 1024 Bytes of Memory

Example 2: Design an interfacing circuit so that ROM 1 of 8K and ROM 2 of 8K and R/W M_1 of 1K can be interfaced with 8085. The memory address specified by the memory devices are:

ROM 1: 0000H to 1FFFH
ROM 2: E000H to FFFFH
R/W M_1: 8000H to 83FFH

For ROM 1

- Given Memory Size: **8** Kilobyte = **1024x8** Bytes
- Therefore the memory chip has **8192** registers.
- Number of address lines **(n) = ln8192/ln2 = 13(i.e. A_{12}-A_0)**, which is required to identify the registers.
- The remaining three address lines (i.e. **A_{15}-A_{13}**) of the microprocessor are used for the Chip - Select (**$\overline{CS}$**) signal. The memory chip for **ROM 1** is enabled when the address lines (i.e. **A_{15}-A_{13}**) are at logic **0**. The address lines **A_{12}-A_0** can assume any address of the **8192** registers, starting from all **0**s to all **1**s.

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0000H**
Chip Select			↓					**Register Select**							↓	
0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	**1FFFH**

For ROM 1

Given Memory Size: **8** Kilobyte = **1024x8** Bytes

- Therefore the memory chip has **8192** registers.
- Number of address lines **(n) = ln8192/ln2 = 13(i.e. A_{12}-A_0)**, which is required to identify the registers.
- The remaining three address lines (i.e. **A_{15}-A_{13}**) of the microprocessor are used for the Chip - Select (**$\overline{CS}$**) signal. The memory chip for **ROM 2** is enabled when the address lines (i.e. **A_{15}-A_{13}**) are at logic **1**. The address lines **A_{12}-A_0** can assume any address of the **8192** registers, starting from all **0**s to all **1**s.

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	**E000H**
Chip Select			↓					**Register Select**							↓	
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**FFFFH**

For R/W M_1

- Given Memory Size: **1** Kilobyte = **1024** Bytes
- Therefore the memory chip has **1024** registers.
- Number of address lines **(n) = ln1024/ln2 = 10(i.e. A_9-A_0)**, which is required to identify the registers.
- The remaining three address lines (i.e. **A_{15}-A_{10}**) of the microprocessor are used for the Chip - Select (**$\overline{CS}$**) signal. The memory chip for **R/W M_1** is enabled when the address lines i.e. **A_{15}** is at logic **1** and **A_{14}-A_{10}** are at logic **0**. The address lines **A_9-A_0** can assume any address of the **1024** registers, starting from all **0**s to all **1**s.

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**8000H**
		Chip Select				↓				**Register Select**					↓	
1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	**83FFH**

The interfacing circuit is given below with all above designation:

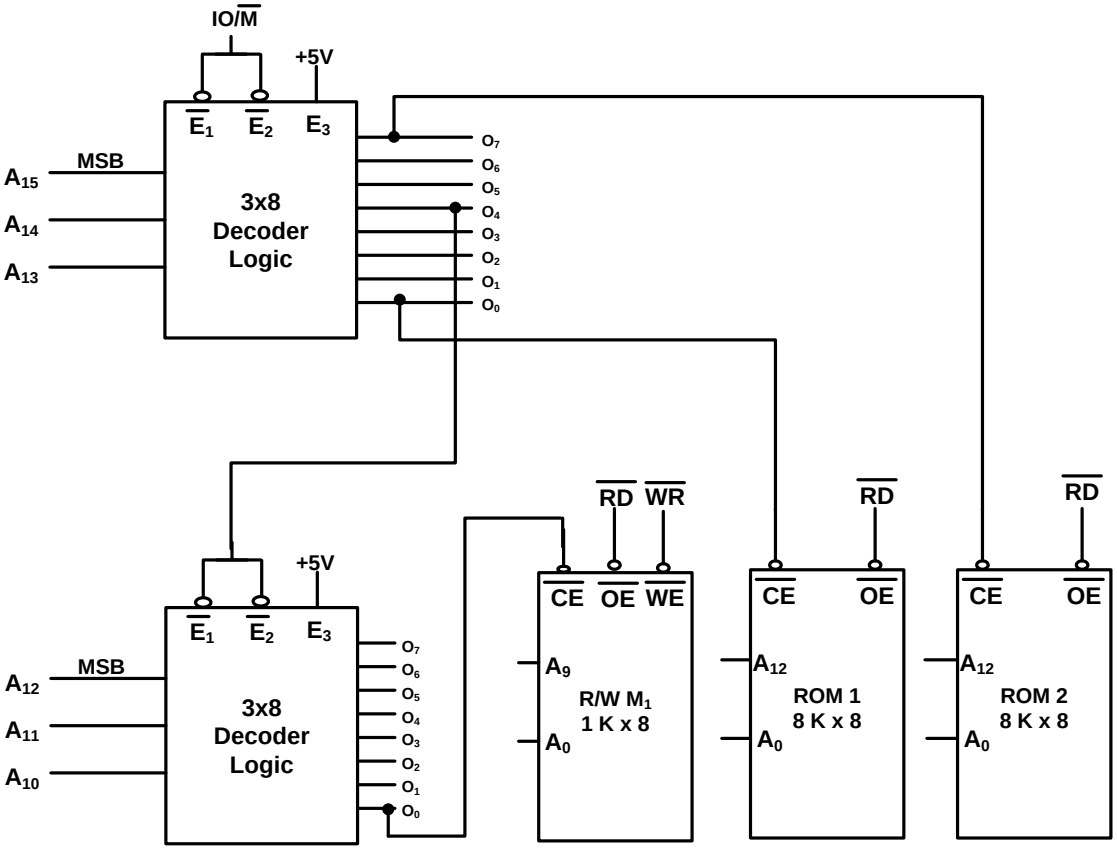


Fig: Memory Design

CONTROL SIGNAL GENERATION

- **Generation of Read/Write Control Signal**

Memory chip consists of different control signals; memory read (**$\overline{\text{MEMR}}$**), Memory Write (**$\overline{\text{MEMW}}$**), Input/Output Read (**$\overline{\text{IOR}}$**) and Input/Output Write (**$\overline{\text{IOW}}$**). They are generated by combining the signals **RD**, **WR** and **IO/$\overline{\text{M}}$**.

- To generate the control signal **$\overline{\text{MEMR}}$** we do have the combination of **IO/$\overline{\text{M}}$** and **$\overline{\text{RD}}$** signals as shown in logic diagram below:
- The signal **IO/$\overline{\text{M}}$** goes low for the memory operation. This signal is **AND**ed with **$\overline{\text{RD}}$** and **$\overline{\text{WR}}$** signal to produce **$\overline{\text{MEMR}}$** and **$\overline{\text{MEMW}}$** control signal respectively when both input signals go low.
- When the **IO/$\overline{\text{M}}$** signal goes high, it indicates the peripheral **I/O** operation. This signal is complemented using the inverter and **AND**ed with **$\overline{\text{RD}}$** and **$\overline{\text{WR}}$** signals to generated **$\overline{\text{IOR}}$** and **$\overline{\text{IOW}}$** control signals respectively.

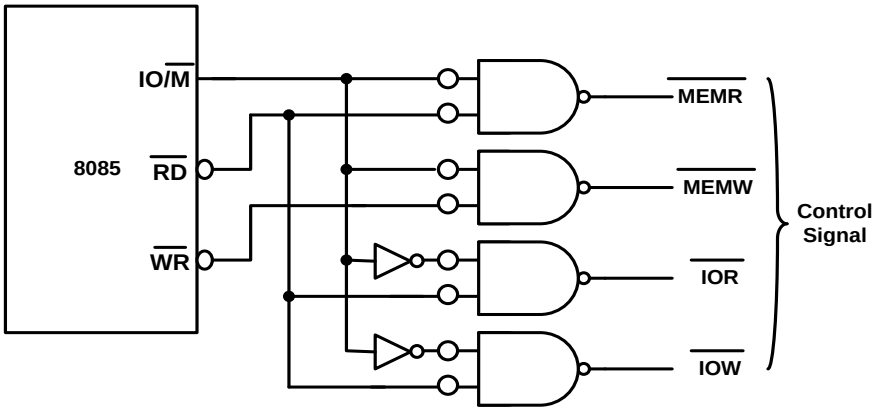


Fig: Schematic to Generate Read/Write Control Signal for Memory & I/O

- **Generation of Chip Select Control Signal**
  - Using NAND Gate
  - Using 3X8 Decoder Logic

- In **8**-bit microprocessor like **8085**, **16**-bit address lines are used. The number of address lines can be determined by the size of memory chip as given by **2^n**, where **n** is the number of address lines; i.e. out of sixteen address lines '**n**' number of address lines are used for identify the memory register while remaining are used to enable **$\overline{\text{CS}}$** control signal using **NAND** gate as shown below:

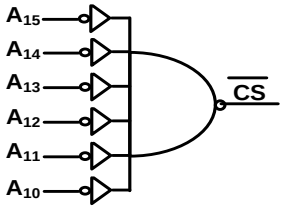


Fig: Chip Select Using NAND Gate

Now, by choosing any logic level either **1** or **0** the combination of input to the **NAND** gate can be varied and the **Chip Select** line can be generated as per our requirement. By the modification of input bits of **NAND** gate, memory address range can be defined as per our need. For above example of **6** inputted **NAND** gate the memory address range can be varied from **000H** to **11FH.**

- The decoder is used to decode the address lines. The output of the decoder is connected to Chip Enable (Chip Select) control signal. The Chip Select line is asserted only when the address on address lines is equivalent to **0**. The control signal can be generated using **3x8** decoder logic as shown below:

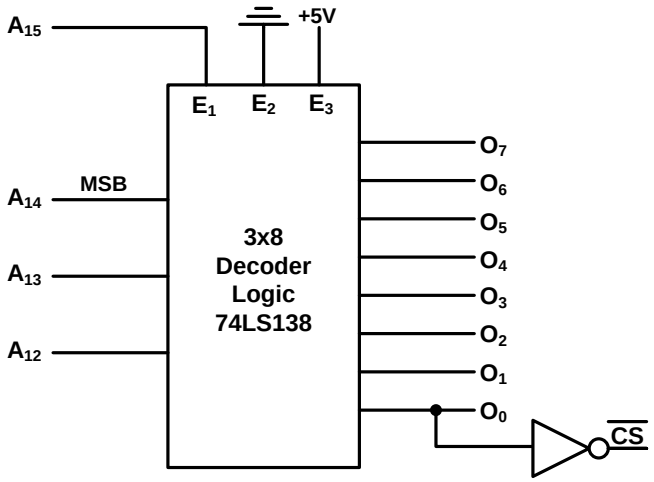


Fig: Chip Select Using 3x8 Decoder Logic

ADDRESS DECODING

A microcomputer system consists of microprocessor, memory and **I/O** interfacing circuit. The microprocessor communicates with all the components interconnected in system through a common address and data bus. As a result, only one device can transmit data through the bus at a time and other can only receive that data. If more than one device attempt to send data through the bus at the same time, data collision may take place. So to avoid such situation and to ensure that the proper device gets addressed in proper time, the technique used is called address decoding.

In address decoding method, all the devices like memory, **I/O** units etc are assigned with specific addresses. The address of device is determined from the way in which the address line (from the processor) are used to enable a special device selection signal known as **Chip Select (CS).**

Suppose, if the microprocessor need to communicate with the memory (either to write or read from), the **Chip Select (CS)** pin of that particular memory should be enabled. At that time, the address decoding circuit must ensure that the **Chip Select (CS)** pins of other devices are not activated.

Let's consider **2732 EPROM** need to be interface with microprocessor. It is **4K× 8** bit memory, so its address ranges from **0000H** to **0FFFH**.

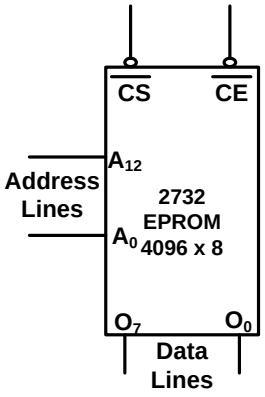


Fig: 2732 EPROM

To read the data from this memory, the microprocessor need to perform the following operations:

- **Select the Chip.**
- **Identify the Register. (Memory Location)**
- **Enable the OE (Output Enable) Pin.**

Since, the memory range from **0000H** to **0FFFH**; only **12** address line (A_0-A_{12}) will be sufficient to select all the memory location uniquely. Assume the rest address lines (A_{12}-A_{15}) must be **0000H** to enable the **Chip Select (CS).**

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Chip Select/Chip Enable				↓						Register Select					↓
0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1

Fig: Address Decoding

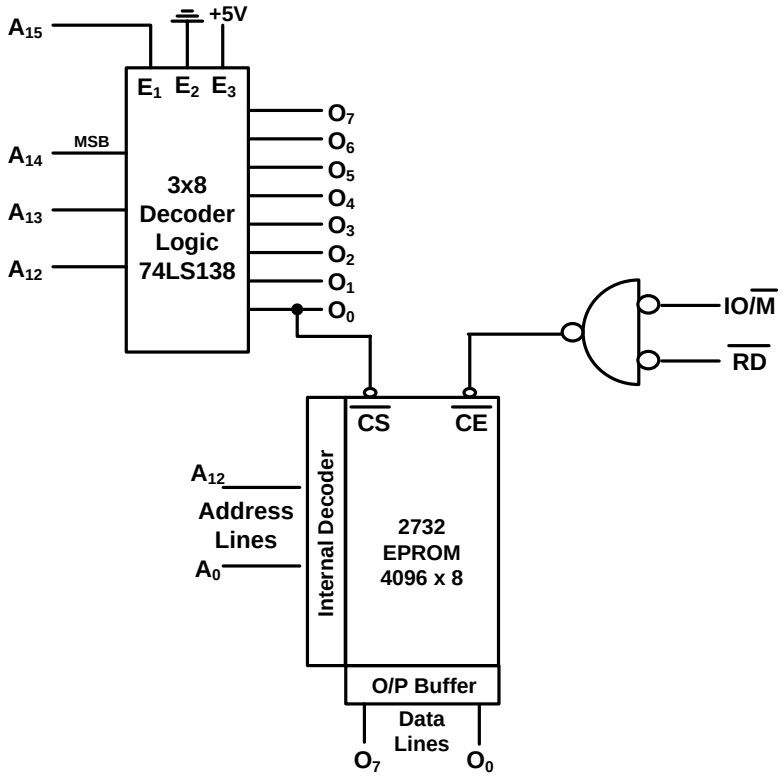


Fig: EPROM Interfacing

This process of assigning address to a specific device depends on how many address lines are assigned to access that device. Generally, there are two common methods; for mapping address of these devices, they are:

- **I/O Mapped I/O**
- **Memory Mapped I/O**

I/O Mapped I/O (Peripheral-Mapped I/O)

In this method of address decoding, the **I/O** devices are addressed with **8**-bit address, which means that chip select signals for the devices are derived by using **8** address lines. In **8085** microprocessor, generally lower **8**-bit address lines are used to derive the chip select signal. The higher order address lines are unused and consider as don't care condition. The instruction that are used in **I/O mapped I/O** to communicate with the device are **IN** and **OUT**.

The instruction **IN** (Code **DP**) inputs data from an input device into the accumulator and the instruction **OUT** (Code **D3**) send the contents of the accumulator to an output device. These are two byte instructions, with the second byte specifying the address or the port number of an **I/O** device. Typically, to display the contents of the accumulator at an output device with the address, for example, **01H**, the instruction will be written and stored in memory as follows:

Memory Address	Machine Code	Mnemonics	Memory Contents
2050H	D3	OUT 01H	2050: 1101 0011H = D3H
2051H	01	-	2051: 0000 0001H = 01H

If the output port with the address **01H** is designed as a **LED** display, the instruction **OUT** will display the contents of the accumulator at the port. The second byte of this **OUT** instruction can be any of the **256** combinations of eight bits, from **00H** to **FFH**. Therefore, the **8085** can communicate with **256** different output ports with device addresses ranging from **00H** to **FFH.** Similarly, the instruction **IN** can be used to accept data from **256** different input ports.

Memory Mapped I/O

In this method of address decoding, the **I/O** devices are address with **16**-bit address. It means that chip select signals for the devices are derived by using all **16** address lines from the processor. So, all the **I/O** devices are treated as one of the memory location of the microcomputer. All the instruction and signals that are used for memory access can be used for accessing the **I/O** devices.

For example, **LDA 5000H** instruction will load the accumulator from the contents of address **5000H**. If an input device, instead of memory location is connected at this address, the data from that input device will load in accumulator. This is called **memory mapped I/O** technique.

On the previous example of **EPROM** interfacing, if A_{14} and A_{12} will be high as follow:

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
Chip Select/Chip Enable 5H				↓											↓
									Register Select 000H						

Then, some other **I/O** device connected at output (O_5) of **3-to-8** decoder will be selected. In this case, **EPROM** will not activate, because **EPROM** will be selected only when A_{12}-A_{15} will **0000H.**

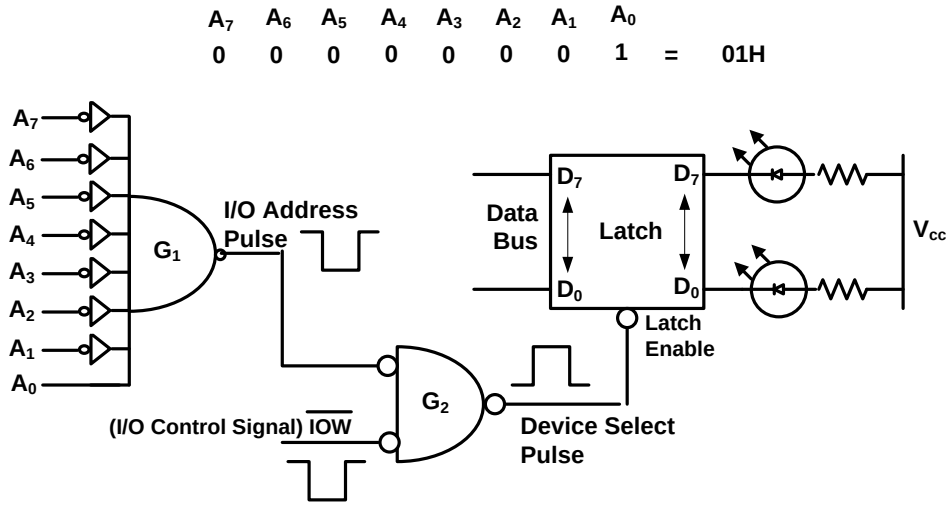
Usually in microcomputer based on **8085**, memory mapping is used for memory devices like **RAM**, **ROM**, **EPROM** etc and **I/O** mapping is used for **I/O** devices like **8255A** (**Programmable Peripheral Interface**), **8251A** (**Programmable Communication Interface**), input/output latches etc.

In both **I/O Mapped I/O** and **Memory Mapped I/O** mode, depending on the number of address line used to select the device, there are two types of address decoding, they are:

- **Unique Address Decoding/Absolute Decoding**
- **Non-Unique Address Decoding/Partial Decoding**

Unique Address Decoding

If all of the address lines available on the mapping mode (i.e. available to select the chip) are used for address decoding, then that decoding is called **Unique Address Decoding**. As the name implies, in this mode, every devices will have single/unique address. This address decoding technique is complicated but is best for all cases.



Fig(1): Unique Address Decoding

Here, we have the illustration for **Unique Address Decoding**. As shown in the **figure 1** the output latch gets enabled for low level logic (**0**). This will happen if the outputs latch of gate G_2 goes low, which is possible only if the signal **IOW** and the output of gate G_1 both go low. When the signal **IOW** goes low the processor starts to transfer (write) the data to the address location.

The address decoding circuit uses only the lower order **8-bit** address lines A_0-A_7. Hence the outputs latch is configured in **I/O** mapping mode. The output of gate G_1 can only become low in the above case. So if A_0 is high and A_1 to A_7 are low the latch gets enabled if the **I/O** control signal **IOW** from the processor becomes low. The data to the output device can be transferred in only one case and hence the device has unique address of **01H**. This process of assigning the single address to a device is called **Unique Address Decoding**.

Non-Unique Address Decoding

If all of the address lines available for the mapping are not used in address decoding, then that address decoding is called non-unique address decoding. The un-used address lines are considered as don't care condition. This technique of address decoding is simple to implement but there may be a chance of address conflict. A single device may have more than one address.

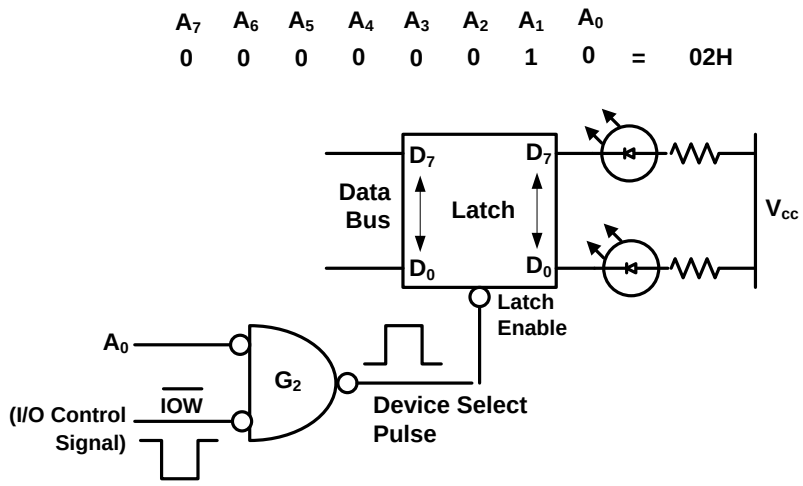


Fig: Non Unique Address Decoding

Here, we have the illustration for **Non Unique Address Decoding**. As shown in the **figure 2** the output latch is again the low-level active device. Only one address line A_0 is used to generate the enable signal for the latch. The higher order address lines A_1 to A_7 are unused and can be assumed as don't care conditions.

The enable signal to the latch gets low if A_0 goes low and the processor sends low level output to its control pin **IOW**. When **IOW** goes low the latch gets enabled for every even address because for even addresses $\mathbf{A_0}$ should be zeroes. For example if $\mathbf{A_1 = 1}$, $\mathbf{A_2}$ to $\mathbf{A_7}$ are zeroes, the address of the latch is determined as enumerated in above figure i.e. **02H**. The multiple addresses for the device are **00, 02, 04 … FE** etc. This type of address decoding is called **Non Unique Address Decoding.**

Note: **THE CONCEPT OF ADDRESS DECODING REGARDING PERIPHERAL MAPPED I/O & MEMORY MAPPED I/O TECHNIQUES WILL BE FURTHER DISCUSSED IN NEXT CHAPTER "INTERFCING I/O DEVICES"**

5 CHAPTER

Input/Output Interfaces

Contents:

- Input/Output Devices
- Input/Output Interfaces
- Serial Interface
- 8251 Programmable Communication Interface
- RS 232-C Standard
- Parallel Interface
- 8279 Programmable Keyboard/Display Interface
- IEEE-488 Bus Interface
- 8255A Programmable Peripheral Interface

Input/Output Devices

The input and output devices in a microcomputer system establish communication between the microcomputer and external world. The **MPU** accepts data as input from devices and sends data to output devices. There are two different methods by which **I/O** devices can be identified: one uses an **8-bit** address and the other uses a **16-bit** address.

I/Os with 8-Bit Addresses (Peripheral-Mapped I/O)

In this type of **I/O**, the **MPU** uses eight address lines to identify an input or an output device; this is known as peripheral-mapped **I/O** (also known as **I/O-mapped I/O**). This is an **8-bit** numbering system for **I/Os** used in conjunction with Input and Output instructions. This is also known as **I/O** space, separate from memory space, which is a **16-bit** numbering system. The eight address lines can have **256** (2^8 combinations) addresses; thus the **MPU** can identify **256** input devices and **256** output devices with addresses ranging from **00H** to **FFH**. The input and output devices are differentiated by the control signals; the **MPU** uses the **I/O Read Control** Signal for input devices and the **I/O Write Control Signal** for output devices. The entire range of **I/O** addresses from **00H to FFH** is known as an **I/O Map** and individual addresses are referred to as **I/O Devices Addresses** or **I/O Port Numbers.**

If we use **LED**s as output or switches as input, we need to resolve two issues: how to assign addresses and how to connect these **I/O** devices to the data bus. In bus architecture, these devices cannot be connected directly to the data bus or the address bus; all connections must be made through tri-state interfacing devices so they will be enabled and connected to the buses only when the **MPU** chooses to communicate with them. In the case of memory, we did not have to be concerned with these problems because of the internal address decoding, **Read/Write Buffers** and availability of $\overline{\mathbf{CS}}$ and control signals of the memory chip. In the case of **I/O** devices, we need to use external interfacing devices.

The steps in communicating with an **I/O** device are similar to those in communicating with memory and can be summarized as follows:

- The **MPU** places an **8-bit** address on the address bus, which is decoded by external decode logic.
- The **MPU** sends a control signal (**I/O Read** or **I/O Write**) and enables the **I/O** device.
- Data are transferred using the data bus.

I/Os with 16-Bit Addresses (Memory-Mapped I/O)

In this type of **I/O**, the **MPU** uses **16** address lines to identify an **I/O** device; an **I/O** is connected as if it is a memory register. This is known as **Memory-Mapped I/O.** The **MPU** uses the same control signal (**Memory Read or Memory Write**) and instructions as those of memory. In some microprocessors, such as the **Motorola 6900**, all **I/O**s have **16-bit** addresses; **I/O**s and memory share the same **Memory Map** (**64K**). In **Memory-Mapped I/O**, the **MPU** follows the same steps as if it is accessing a memory register.

INPUT/OUTPUT INTERFACE

The link between the **I/O** devices and the microcomputer is maintained by a circuitry known as **I/O Interface**. **I/O Interface** is a circuitry that connects **I/O** devices to the microcomputer. For interfacing microprocessor to common peripherals (also called **I/O** devices); such as **Keyboard, CRT Terminal, Printer, Floppy Disk** etc requires **I/O Interface Circuit**. In the **8085** based systems, **I/O** devices can be interfaced using both techniques as we have discussed earlier i.e. Peripheral Mapped **I/O** and Memory Mapped **I/O** Techniques. The process of data transfer in both is identical. Each device is assigned a binary address, called device address or port number, through it interfacing circuit. When the microprocessor executes a data transfer instruction for an **I/O** device, it places the appropriate address on the address bus, sends the control signals, enables the interfacing device, and transfer data. The interfacing device is like a gate for data bus, which is opened by the **MPU** whenever it intends to transfer data. The two major types of **I/O** interfaces are:

1. **Serial Interface**
2. **Parallel Interface**

SERIAL INTERFACE

Serial communication involves the transmission of data from one place to another using serial device. It means in serial transmission, only one line is used to transmit the complete binary data bit by bit. Thus a serial interface exchanges data with the peripheral in serial mode where the data are transmitted one bit at a time along a single communication link.

For example, to transmit the data **10010001**, the data on the channel would be as shown below:

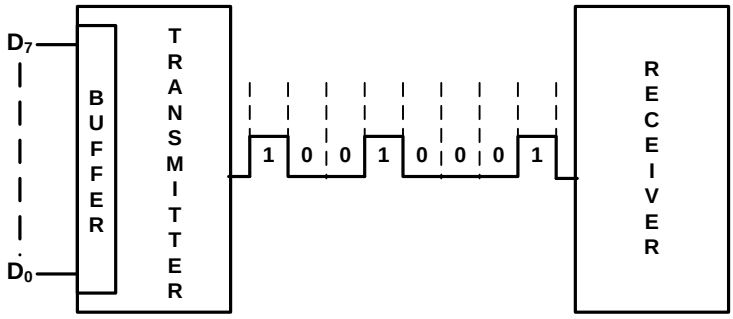


Fig: Serial Data Transmission

A parallel to serial converter is used to convert the incoming parallel data to a serial form and then data is sent out with **The Least Significant Bit** (**LSB**) first and **Most Significant Bit** (**MSB**) at last. Serial transmission is slow but inexpensive to implement as far as the number of wire is concerned. Since only one wire is used for data transfer the amount of cross talk (interference between different lines) decreases. Serial data transmission can be divided into following two types.

- **Synchronous Serial Data Transmission**
- **Asynchronous Serial Data Transmission**

Synchronous Data Transmission

The basic feature of synchronous data transmission is that the data is transmitted or received based on a clock signal and both the transmitter and receiver are synchronized with the common clock signal.

It is also called clock oriented transmission because all bytes of block are transmitted constantly. In this transmission, a block of data byte is transmitted along with the synchronization information. Usually, one or two **SYNC** characters are used to indicate the start of each synchronous data stream as shown below:

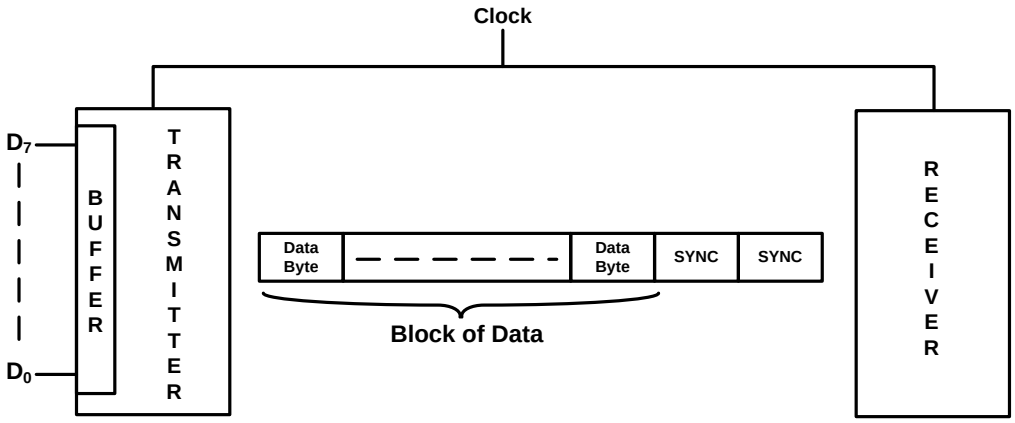


Fig: Synchronous Transmission

At the receiver side, as soon as it matches one or two **SYNC** characters based on the number of **SYNC** character used, the receiver starts interpreting the data. In synchronous transmission, the transmitting device needs to send data continuously to the receiving device. However, if data is not ready to be transmitting, the transmitter will send **SYNC** characters until the data is available.

Asynchronous Data Transmission

In asynchronous data transmission, the transmitter and receiver need not to be synchronized with the common clock signals. It is also called character oriented transmission, because only one character is transmitted at a time. Each character carries a start bits and stop bits as synchronization information. Transmission begins with one start bit that is at logic **0** (called space), followed by the required information byte, which is always transmitted with **LSB** first and finally the stop bits that is at logic **1** (called Mark). This formatting of data is known as "**Framing**".

Asynchronous data transmission is shown below with the framing information.

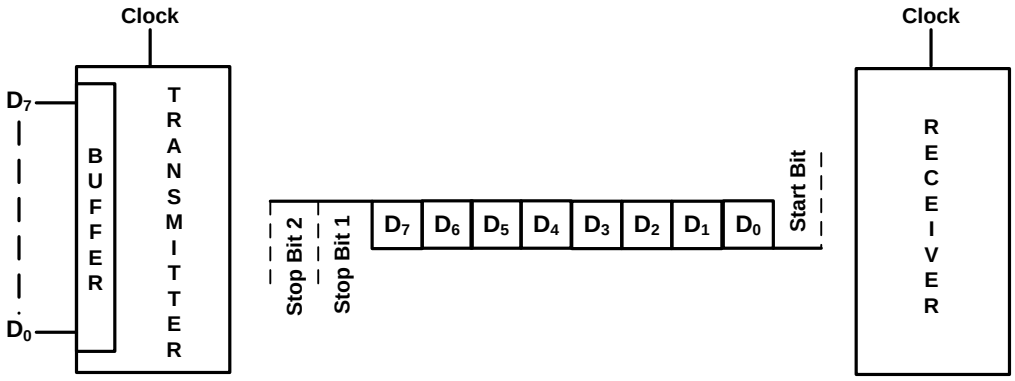


Fig: Asynchronous Transmission

The asynchronous transmission is generally used in low speed transmission less than **20 Kbps**, but synchronous transmission transmits data greater than **20Kbps**.

Each asynchronous serial data unit can be divided into equal time intervals called bit intervals. An **8-bit** data will have **8-bit** intervals. Thus each data bit will correspond to one of the **8** intervals. The format for asynchronous serial data contains the following information:

- A low **START** bit that indicates beginning of the data to be transferred.
- **8** data bits, denoting the actual data being transferred.
- An optional parity bit for either odd or even parity.

Baud Rate

The serial data transmission rate is also called the baud rate. The baud rate is defined as the number of bits of data transmitted per second. Since each bit is transmitted over a duration of one interval, **Baud rate = 1/Bit Interval = Bits/Sec.**

In parallel **I/O**, data bits are transferred when a control signal enables the interfacing device; the transfer takes place in less than three T-states. However, in serial **I/O**, one bit is sent out at a time; therefore, how long the bit stays on or off is determined by the speed at which the bits are transmitted. Furthermore, the receiver should be set up to receive the bits at the same rate as the transmission; otherwise, the receiver may not be able to differentiate between two consecutive **0**s and **1**s.

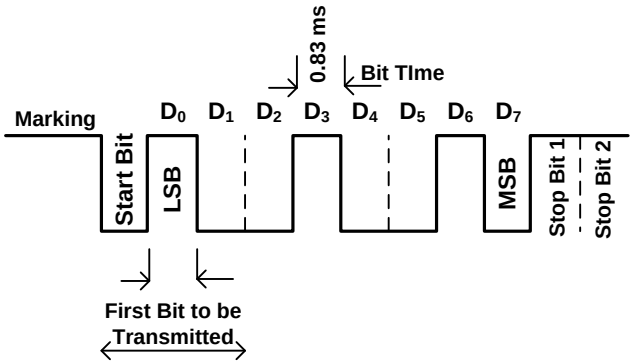


Fig: Serial Bit Format

Above figure shows how the **ASCII** character **I (49H)** will be transmitted with **1200** baud with the framing information of one Start and two Stop bits. The transmission begins with an active low Start it, followed by the **LSB**-bit $\mathbf{D_0}$. The bit time-the delay between any two successive bits is **0.83 ms**; this is determined by the baud as follows:

1200 bits = 1 second
For 1 bit = 1/1200 = 0.83ms

Therefore, to transmit one character, a parallel byte (**49H**) should be converted into a stream of **11** bits by adding framing bits and each bit must be transmitted at the interval of **0.83 ms**. To receive a character in the serial mode, the process is reversed-one bit at a time is received and the bits are converted into a parallel word.

ASCII Character

The acronym **ASCII** stands for the "**American Standard Code for Information Interchange**". It is an **8-bit** code commonly used with microprocessors for representing alphanumeric codes. Out of the **8** bits first **7** bits are used to represent the character while the **8th** bit is used to test for errors and is referred to as parity bit. The parity bit can be set to **1** or **0** so that the number of **1-bits** in the byte either always odd or always even. The standard **ASCII Chart** is given below:

STANDARD ASCII CHART (0 – 127)

CHARACTER	DECIMAL	HEX	CHARACTER	DECIMAL	HEX
NUL	0	00h	SOH	1	01h
STX	2	02h	ETX	3	03h
EOT	4	04h	ENQ	5	05h
ACK	6	06h	BEL	7	07h
BS	8	08h	HT	9	09h
LF	10	0Ah	VT	11	0Bh
FF	12	0Ch	CR	13	0Dh
SO	14	0Eh	SI	15	0Fh
DLE	16	10h	DC1	17	11h
DC2	18	12h	DC3	19	13h
DC4	20	14h	NAK	21	15h
SYN	22	16h	ETB	23	17h
CAN	24	18h	EM	25	19h
SUB	26	1Ah	ESC	27	1Bh
FS	28	1Ch	GS	29	1Dh
RS	30	1Eh	US	31	1Fh
SP	32	20h	!	33	21h
"	34	22h	#	35	23h
$	36	24h	%	37	25h
&	38	26h	‘	39	27h
(	40	28h	)	41	29h
*	42	2Ah	+	43	2Bh
,	44	2Ch	-	45	2Dh
.	46	2Eh	/	47	2Fh
0	48	30h	1	49	31h
2	50	32h	3	51	33h
4	52	34h	5	53	35h
6	54	36h	7	55	37h

8	56	38h	9	57	39h
:	58	3Ah	;	59	3Bh
<	60	3Ch	=	61	3Dh
>	62	3Eh	?	63	3Fh
B	64	40h	A	65	41h
	66	42h	C	67	43h
D	68	44h	E	69	45h
F	70	46h	G	71	47h
H	72	48h	I	73	49h
J	74	4Ah	K	75	4Bh
L	76	4Ch	M	77	4Dh
N	78	4Eh	O	79	4Fh
P	80	50h	Q	81	51h
R	82	52h	S	83	53h
T	84	54h	U	85	55h
V	86	56h	W	87	57h
X	88	58h	Y	89	59h
Z	90	5Ah	[	91	5Bh
\	92	5Ch	]	93	5Dh
^	94	5Eh	_	95	5Fh
`	96	60h	a	97	61h
b	98	62h	c	99	63h
d	100	64h	e	101	65h
f	102	66h	g	103	67h
h	104	68h	i	105	69h
j	106	6Ah	k	107	6Bh
l	108	6Ch	m	109	6Dh
n	110	6Eh	o	111	6Fh
p	112	70h	q	113	71h
r	114	72h	s	115	73h
t	116	74h	u	117	75h
v	118	76h	w	119	77h
x	120	78h	y	121	79h
z	122	7Ah	{	123	7Bh
|	124	7Ch	}	125	7Dh
~	126	7Eh	DEL	127	7Fh

Notes:

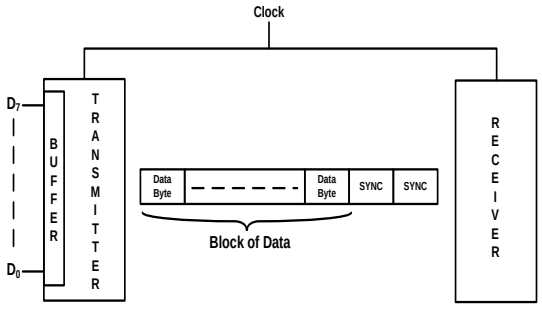
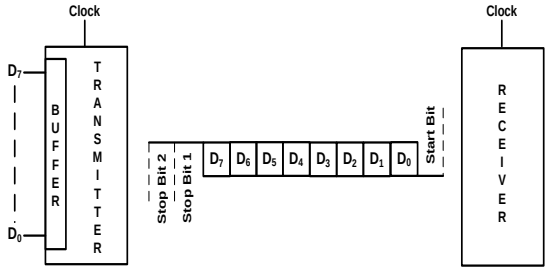
With synchronous transmission, a block of bits is transmitted in a steady stream without start and stop bit code. The block may be many bits in length. So, to prevent timing drift between the transmitter and receiver, these blocks must somehow be synchronized, for which there are two possibilities.

One possibility is to provide a separate clock line between transmitter and receiver. One side pulses the line regularly and the other side uses it as a clock. This technique works well over short distance; but over long distances, the clock pulse are subjected to some impairment as data signal and timing error can occur.

Another efficient possibility is to synchronize the transmitter and receiver and then send large block of data one after another. To indicate the start of transmission the transmitter sends out one or more unique character called '**SYNC'** character or a unique bit pattern called **'FLAG'**. Receiver uses the **'SYNC'** character or **'FLAG'** to synchronize its internal clock with that of the transmitter, receiver then shifts in the data following the **'SYNC'** character and convert them to parallel form to be read by microprocessor.

SYNCHRONOUS VS ASYNCHRONOUS SERIAL DATA TRANSMISSION

Synchronous Transmission	S.N	Asynchronous Transmission
▪ Synchronous means at the same time. So, both transmitter and receiver must be synchronous with the common or same clock.	1	▪ Asynchronous means at irregular intervals. So, transmitter and receiver need not to be synchronous with the common clock.
▪ Serial data is transmitted along with the clock signal itself, which provides the timing information for the receiver.	2	▪ Serial data is transmitted without transmitting the clock information.
▪ Also called clock oriented transmission because a block of characters is transmitted along with the synchronization information.	3	▪ Also called character oriented transmission because only one character is transmitted at a time.
▪ A complete block of characters (bytes) treated individually and is transmitted at a time in a steady stream.	4	▪ Each character is treated individually.
▪ Transmission begins by sending one or two unique characters called **'SYNC** character or unique bit pattern called '**FLAG'** to the receiver that a new synchronous data stream is coming. Receiver uses the **'SYNC'** character or '**FLAG'** to synchronize its internal clock with that of the transmitter.	5	▪ Transmission begins with one start bit which is always at logic low (0) called '**SPACE'** followed by the required information byte (which is always transmitted with its **LSB** first) and finally one or two stop bits that are always at logic high (1) called **'MARK'.** This formatting of data is called **Framing.**
▪ It is used for high speed transmission (greater than 20Kbps).	6	▪ It is used for low speed transmission (less that 20Kbps).
▪ It uses more sophisticated error detection and correction mechanism.	7	▪ It uses less sophisticated error detection mechanisms, e.g. parity bit.
▪ No pre-adjustment of data rate is required as the receiver can respond to various clock rates (as long as its maximum frequency isn't exceeded) by simply detecting the low high transition of the clock signal.	8	▪ Manual pre-adjustment or presetting of both transmitter and receiver is necessary for operation at specific bit period or data rate because separate clocks are used by the transmitter and the receiver and because the receiver can make no assumptions about the bit period.
▪ In order to interpret the data correctly, the receiving device must know the start and end of each device data unit; so in synchronous data transfer, the transmitter must know the number of data units to be transmitted.	9	----------
▪ The synchronous receiver waits in a "**HUNT"** mode while looking for data.	10	----------

As soon as it matches the one or two **'SYNC'** characters, the receiver starts interpreting the data.		
▪ In synchronous transmission, the transmitter must send data continuously to the receiver. However, if the data isn't ready to be transmitted, the transmitter will send **'SYNC'** characters until the data is available.	11	▪ In asynchronous transmission, the transmitter need not send data continuously. It can start sending character at any time. When no data re being transmitted a receiver stays high at logic 1 called **MARK**.
▪ Basic Feature: The data is transmitted or received based on a clock signal i.e. a common clock signal is used to synchronize both the transmitter and receiver on a block by block basis. At a specific rate of data transfer, the transmitting device sends a data bit at each clock pulse to the receiver synchronized at the same clock rate without sending start and stop bit(s).	12	▪ Basic Feature: Synchronization is maintained within each character. The receiver has an opportunity to resynchronize itself at the beginning of each new character i.e. synchronization is maintained on a character by character basis by the help of start bit and stop bit(s).
▪ Example: High Speed Modems & Digital Communication Channel use synchronous data transfer.	13	▪ Example: Data transfer between the microprocessor and the serial peripherals such as terminals, floppy disks etc is primarily asynchronous.
▪ Clock lines are needed, hence more number of wires.	14	▪ No clock lines are needed, hence less number of wires.
▪ Somewhat sophisticated and expensive.	15	▪ Simple and cheap, however requires and overhead of 2 to 3 bits per character.
▪ Works for sufficiently long block of data bytes after some **'SYNC'** character(s).	16	▪ Doesn't work for long block of data between start and stop bit because it increases the cumulative timing error.
 Fig: Synchronous Transmission		 Fig: Asynchronous Transmission

8251A Programmable Communication Interface (PCI)

The **8251A** is **USART** (**Universal Synchronous/Asynchronous Receiver/Transmitter**), designed for synchronous and asynchronous serial data communication. It is a programmable **28-Pin Chip.** It includes five different sections, they are:

- **Read/Write Control Logic & Registers**
- **Transmitter**
- **Receiver**
- **Data Bus Buffer**
- **Modem Control**

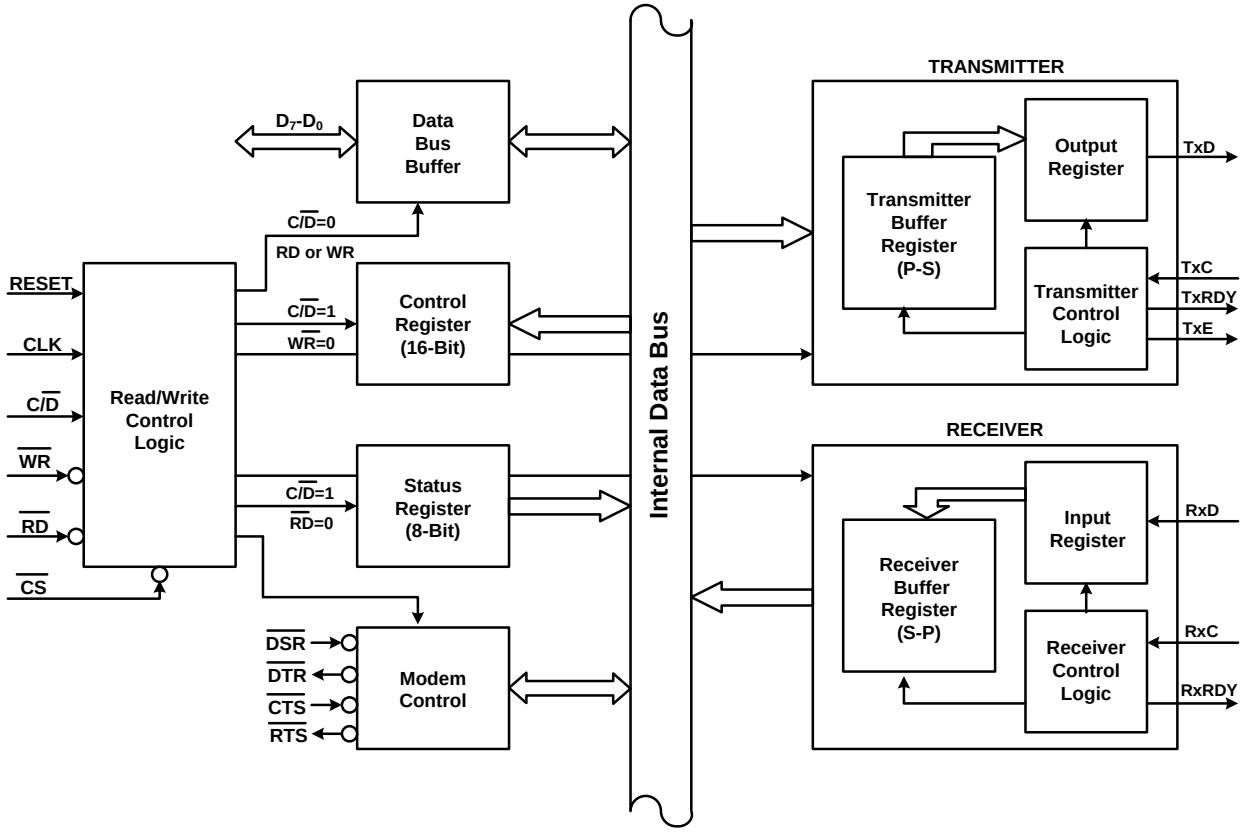


Fig: Block Diagram of 8251A PCI

Read/Write Control Logic & Registers

This section includes control logic, six input signals and three buffer registers: **Data Register, Control Register & Status Register.**

Input Signals

$\overline{CS}$ (Chip Select): When this signal goes low, **8251A** is selected.

RESET: A high signal on this pin reset **8251A**.

C/$\overline{D}$ (Control/Data): When this signal is high, the control register or status register is selected: The control and status register are differentiate by **WR** and **RD** signals.

- **$\overline{WR}$ (Write):** When this signal goes low, the **MPU** writes in the control register or sends output data to the **Data Buffer Register.**
- **$\overline{RD}$ (Read):** When this signal goes low, the MPU either reads a status from **Status Register** or accept data from the **Data Buffer Register.**
- **CLK (Clock):** This is the clock input for **8251A** and is connected to the system clock. The clock is necessary for communication with the microprocessor.

Control Register

The **16-bit** control register split into two bytes. The first byte is called **Mode Word Instruction Register** and second byte is **Command Word Instruction Register.**

Status Register

It checks the ready status of a peripheral, status of transmission, transmission error etc.

Data Buffer Register

This is **3-state** bidirectional buffer register used to interface **8251A** to the system bus.

Summary of Control Signal for 8251A PCI

CS	C/D	RD	WR	Function
0	1	1	0	MPU writes instruction in the control register.
0	1	0	1	MPU reads status from status register.
0	0	1	0	MPU outputs data to the data buffer register.
0	0	0	1	MPU accepts data from the data buffer register
1	x	x	x	8251A is not selected

Transmitter Section

The transmitter accepts parallel data from the **MPU** and converts them into serial data for the transmission. It has two register: A **Buffer Register** to hold eight bit data and an **Output Register** to convert parallel data into serial data.

Transmitter control logic to controls all the operations related to the data transmission. The transmission section has three output signals and one input signal.

- **TxD (Transmit Data):** Serial bits are transmitted in this line.
- **$\overline{\text{TxC}}$ (Transmitter Clock):** The transmitter clock controls the rate at which bits are transmitted. This clock frequency can be 1 times the baud rate in synchronous transmission mode and can be **1, 16** or **64** times in asynchronous transmission mode. Suppose, **Baud Rate = 110**
  In Synchronous Transmission Mode, **$\overline{\text{TxC}}$ = 110 Hz**
  In Asynchronous Transmission Mode, **$\overline{\text{TxC}}$** = **110 Hz** in **1X** Mode, **$\overline{\text{TxC}}$** = **1.76 KHz** in **16X** Mode**, $\overline{\text{TxC}}$ = 7.04 KHz** in **64X** Mode
- **TxRDY (Transmitter Ready):** This output signal when goes high, it indicates that the **Buffer Register** is empty and is ready to accept a data byte. This signal is reset when a data byte is loaded into the buffer.
- **TxE (Transmitter Empty):** This is also an output signal and when it goes high, it indicates that output register is empty or **8251A** has no character to send. This signal is reset, when a data byte is transferred from **Buffer Register** to the output register.

Receiver Section

The receiver accepts serial data from the peripheral devices and converts them into the parallel data and transfer to the microprocessor. It has two registers: A **Buffer Register** to hold **8**-bit data and an **Input Register** to accept the serial data from the peripheral & convert them into the parallel data.

Receiver Control Logic controls all the operations related to the receiving data. The receiver section has two input signals and one output signal.

- **RxD (Receive Data):** Bits are received serially on this line and converted into a parallel byte in the receiver input register.
- **RxC (Receiver Clock):** The receiver clock controls the rate at which bits are received by the **8251A**. In synchronous mode, this clock can be 1 time the baud rate and in asynchronous mode, it can be **1, 16,** or **64** times the baud rate.
- **RxRDY (Receiver Ready):** This is an output signal and when it goes high, it indicates that **8251A** has character in the receiver buffer register and is ready to transfer it to the microprocessor.

Modem Control

The **Modem Control** is used to establish data communication through modems over telephone lines. However, telephone lines are designed to handle voice; the bandwidth of telephone lines ranges from **300 Hz** to **3300 Hz.** The digital signal with rise time in nsec requires a bandwidth of several **MHz.** Therefore, data bits should be converted into audio tones; this is accomplished through modems. A modem is a circuit that translates digital data into audio tone frequencies for transmission over the telephone lines and converts audio frequencies into digital data for reception.

RS 232-C Standard in Serial Input/Output

Serial transmission of data is used as an efficient means for transmitting digital information across long distances. The obvious advantage is the savings in hardware. Also the existing communication lines usually the telephone lines can be used to transfer information.

The **RS 232-C** is an interface convention developed standardizes the interface between **Data Terminal Equipment (DTE)** and **Data Communication Equipment (DCE)** employing serial binary data exchange. One of the most common uses for **RS 232-C** is the connection of computer terminals to computers. This is done both directly and indirectly through modems.

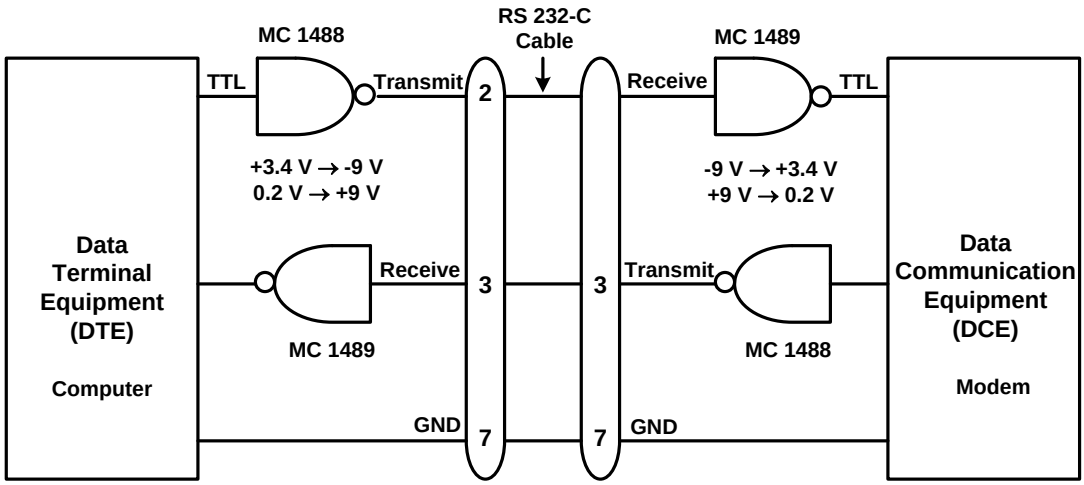


Fig: RS 232-C Standard For Serial I/O

As we have known that modem and other devices used to send serial data are also called **Data Communication Equipment (DCE).** The terminals or computers that are sending or receiving the data are called **Data Terminal Equipment (DTE). RS 232-C** is the interface standard developed by **Electronic Industries Association (EIA)** in response to the need for signal and handshake standards between the **DTE** and **DCE.**

This standard describes the functions of **25** signal and handshake pins for serial data transfer. It also describes the voltage levels, impedance levels, rise and fall times, maximum bit rate and maximum capacitance for these signal lines. It also specifies that the **DTE** connector should be male and the **DCE** connector should be female. Usually the **9-pin** and **25-pin** connectors are available. The voltage levels for all **RS 232-C** are as follow:

Logic high or 1 is a voltage between -3V and -15V
Logic low or 0 is a voltage between +3V and +15V
The commonly used voltages are +12V and -12V

The signal levels of **RS 232-C** are not compatible with that of The **DTE** and **DCE**, which are **TTL** signals. Hence **TTL** to **RS 232-C** and **RS 232-C** to **TTL** conversions are required for establishing the link between **DTE** and **DCE**. The line driver such as **MC1488** ca n be used between **DTE** and **RS 232-C** cable and **MC1489** can be used between the **RS 232-C** cable and **DCE.**

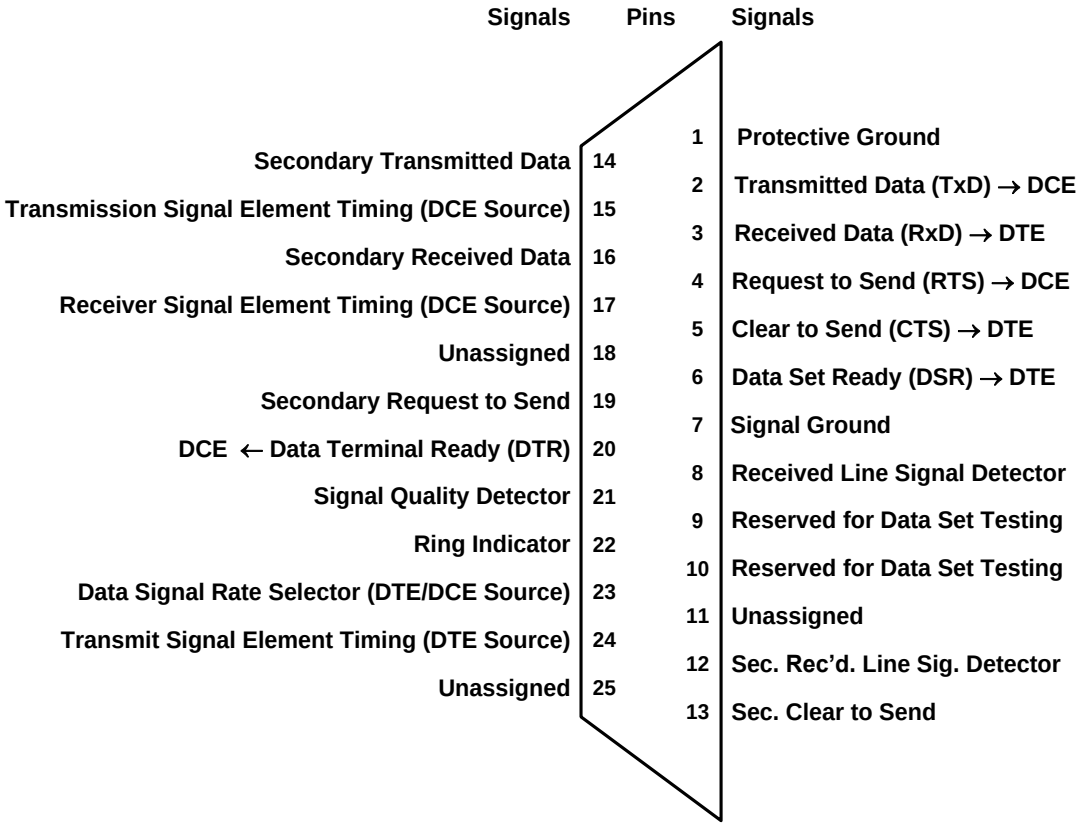


Fig: RS 232-C Pin Diagram

Among **25** pins of **RS 232-C**, the important pin numbers, the signal names and their descriptions are listed in table below:

Pin No.	Signals	Functions
	Transmit Data TxD	Output, transmit data from DTE to DCE.
3	Receive Data RxD	Input, DTE receives data from DCE.
4	Request to Send RTS	General purpose output from DTE.
5	Clear to Send CTS	Input to DTE, used as handshake.
6	Data Set Ready DSR	Input to DTE, indicates that DCE is ready.
7	Signal Ground GND	Common reference between DTE and DCE.
8	Data Carrier Detect DCD	Used by DTE to disable data reception.
20	Data Terminal Ready DTR	Output means DTE is ready.

Parallel Interface

A Parallel Interface exchanges data with the peripheral in parallel mode, where all the n-bits of data are transmitted simultaneously. Each bit in a word is transmitted on a separate line. So to transmit a word of n-bits, there must be n-lines in parallel data transfer.

For example, to transmit the data **10010001**, the data on the channel would be as shown below:

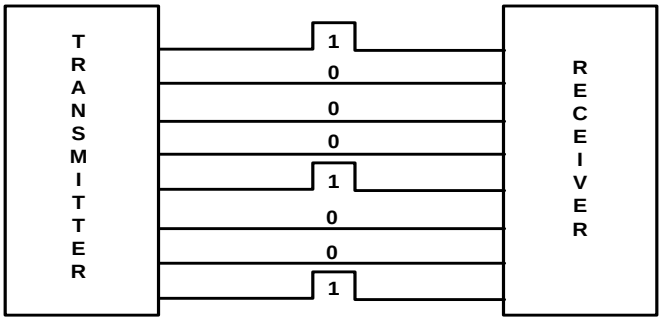


Fig: Parallel Data Transmission

Parallel interface provides a faster exchange of data but becomes expensive due to the needs of multiple communication links. So, it is impractical for exchanging data over a long distance.

Method of Parallel Data Transfer

- **Simple Input & Output**

  In the simple input and output parallel data transfer method, microprocessor assumes that peripheral devices are always present and ready to communicate with the microprocessor.

Fig: Simple I/P and O/P

For example, consider the simple display device such as **LED** is always there and ready, so microprocessor can send data to it at any time.

- **Simple Strobe Input & Output**

  In many applications, valid data is present on an external device only at a certain time, so it must be read in at that time.

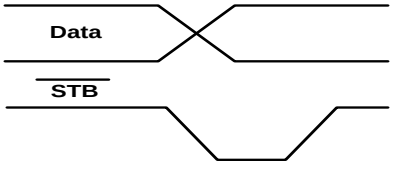


Fig: Simple Strobe I/P & O/P

For example, consider the keyboard. When a key is pressed, keyboard sends out the **ASCII Code** for the pressed key on eight parallel data lines, and then sends out a strobe signal **(STB)** on another line to indicate that valid data is present to transmit.

- **SINGLE HANDSHAKE WITH INPUT & OUTPUT**

In this method, the peripheral sends some parallel data and then sends the **STB** signals to the microprocessor. The microprocessor detects the **STB** signal on a polled or interrupt basis and read in the byte of data. Then microprocessor sends an acknowledge signal (**ACK**) to the peripheral device to indicate that the data has been read and that the peripheral can send the next byte of data.

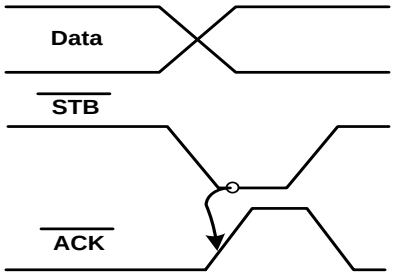


Fig: Single Handshake with I/P & O/P

- **DOUBLE HANDSHAKE WITH INPUT & OUTPUT**

For data transfer where even more coordination is required between the sending system and receiving system, a double handshake is used.

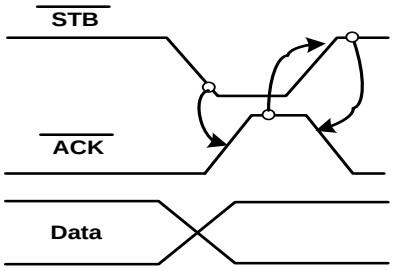


Fig: Double Handshake with I/P & O/P

First of the entire data sending device asserts **STB** signal low to ask, "**Are you ready**?" The receiving system raises its **ACK** line high to indicate that it is ready. The sending device (**Computer**/**Peripheral**) then sends the bytes of data and raises its **STB** signal high to say, "Here is some valid data for you". After receiving the data, the receiving system low its **ACK** signal to indicate that it has received the data and waiting for next byte of data.

SYNCHRONIZING THE COMPUTER WITH PERIPHERALS

The information exchanged between a microprocessor and peripheral devices consists of **I/O** data, control information and status information.

The status information tells the microprocessor about the readiness or status of the peripheral devices. It may also enable the microprocessor to monitor the devices, to ensure that it is in good operation condition. As for example, a printer indicates whether it has run out of paper or the paper is jammed. Another example is about **READY** signal, while communication with the memory device.

Control information comprises the command send by microprocessor to peripheral devices to take some specific action. If the peripheral device operates at different speeds, microprocessor should have some means or technique to communicate with those devices.

Different technique used to transfer data between different speed device and microprocessor is called "**Synchronization".**

8279 Programmable Keyboard/Display Interface

The data input and display are an integral part of many microprocessor based systems. The system designer needs an interface that can control this operation. With this in mind, Intel Corporation brought out **8279 Chip** to interface with their **8-bit** microprocessor. The **8279** is a hardware approach to interfacing keyboard and display.

The **8279** is a **40-pin** device, which functionally consists of four different sections:

- **MPU Interface Section**
- **Keyboard Section**
- **Scan Section**
- **Display Section**

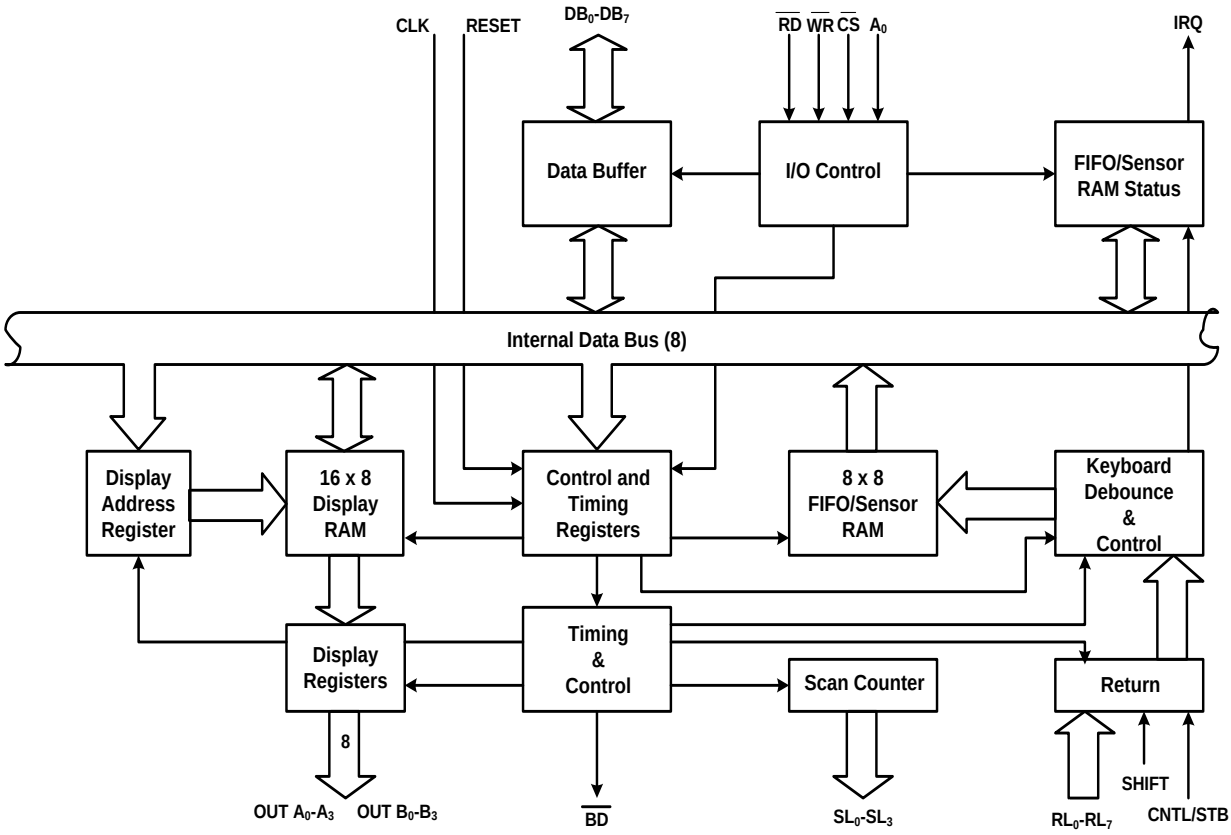


Fig: The 8279 Logic Block Diagram

MPU Interface Section

This section includes eight bidirectional data lines (**DB_0-DB_7**), one interrupt request line (**IRQ**) and six input signals. The six input signals are: **RD** (Read), **WR** (Write), **CS** (Chip Select), **A_0** (Buffer Address Line), **RESET** & **Clock**.

The **IRQ** is an output line that becomes active when key data exists in an internal **FIFO** (First-In-First-Out) **RAM** of **8279**. This line is usually connected to one of the hardware interrupt line of the **MPU**.

The **RD** and **WR** are the control signals used to perform read and write operation respectively. The **CS** and **RESET** signal are used to select and reset the **8279** respectively. The clock signals are used to generate internal timing. A high on the **A_0** pin indicates that the signals are command or status signal and a low indicate that they are data.

Keyboard Section

This section includes eight return lines **(RL_0-RL_7)** that can be connected to eight columns of a keyboard, **SHIFT** and **CNTL/STB** (**Control/Strobe**) lines. Basically there are three modes of operations:

- **Scanned Keyboard Mode**
- **Sensor Matrix Mode**
- **Strobed Input Mode**

Scanned Keyboard Mode

In this mode, when a key is pressed, a unique **6-bit** code is generated, along with the **CNTL** and **SHIFT** states. So all together **8-bit** code is formed and stored in **FIFO RAM**. As soon as the **8-bit** code is stored in **FIFO RAM**, the interrupt request (**IRQ**) pin of the device goes high to indicate that the data is present in **RAM**. As the **FIFO RAM** is read, the **IRQ** goes low, but becomes high again if the **FIFO RAM** contains further data. The data format for the scanned keyboard mode is as follows:

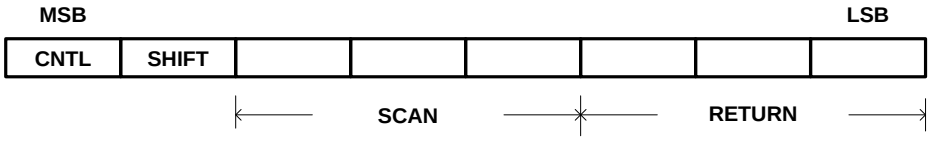


Fig: Scanned Keyboard Mode

In the above figure, the three return bits **D_0**, **D_1** and **D_2** comprise the column of keyboard and the three scan line **D_3, D_4** and **D_5** be the row of the keyboard.

In this scanned keyboard mode, there are two further alternative ways of operations:

- **2-Key Lockout**
- **N-Key Rollover**

In **2-Key Lockout Mode**, if two keys are pressed almost simultaneously (i.e. without releasing the first pressed key, a second key is pressed); only the first key is recognized. The recognized key code is then taken into **FIFO RAM** and **IRQ** is activated. If the **FIFO RAM** is full, the key code is not entered, rather error flag is set.

In **N-Key Rollover Mode**, when a key is pressed, the debounce logic is set and the debounce circuits waits for two keyboard scans and then clocks again whether the key is till pressed. If it still is, the key code is stored in **FIFO RAM**. In this mode, there is no limit to the number of keys that can be pressed for a simultaneous depression, the key code are entered into **FIFO RAM** according to the order of the key pressed. If during single debounce cycle, two keys are pressed, then error flag will be set.

Debounce: When a mechanical switch/key is pressed, there is a transient during which the contact is not firm a microscopic bouncing of a contact elements take place. This may lead to ambiguous interpretation of the states of the switch. To avoid this problem either hardware or software solutions are used, this is known as debouncing.

Sensor Matrix Mode

In this mode, the keys are placed in the form of a matrix (scan lines comprising the columns and the return lines comprising the rows). The data on each of the eight return lines are directly entered into the sensor **RAM**. The **SHIFT** and **CNTL** status are not taken as input in this mode. The data formats for the sensor matrix mode are as follows:

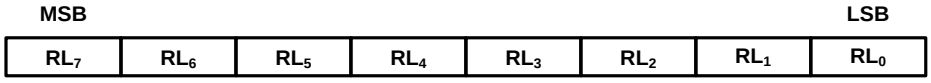


Fig: Sensor Matrix Mode

In this mode, the debounce logic is inhibited. The **IRQ** line goes high, if a sensor value is found to have changed at the end of a sensor matrix scan.

Strobed Input Mode

In this mode, the data is accepted from the return lines and stored in the **FIFO RAM** during the rising edge of a **STB** (**Strobe**) signal. The data formats for the strobed input mode are as follows:

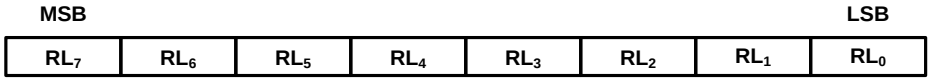


Fig: Strobed Input Mode

Scan Section

The scan section consists of a scan counter and four scan lines (SL_0-SL_3). The scan counter can be operated in either encoded or decoded mode.

In the encoded mode, generally out of four scan lines, **3** scan lines (SL_0-SL_2) are fed to the **3-to-8** decoder (external to the **8279**), that generates eight decoded scan lines. Combining these with the eight return lines, one can form as microprocessor to **8x8** keyboard matrix, which can be used for defining **64** different characters.

In the decoded mode, which uses an internal decoder (present inside the **8279**), tow least significant bits are decoded which is combined with eight return lines (RL_0-RL_7) and the **SHIFT** and **CNTL** line to permits **4 x 8 x 4 = 128** different character definition.

Display Section

The display section consists of display address registers, **16x8** display **RAM**, Display register and eight output lines divided into two groups A_0-A_3 and B_0-B_3. It also consist the single **BD** (**Blank Display**) output line, which is used to blank the display during digit switching.

The display address registers hold the address of the word currently being written or read by the **CPU**. These addresses can be set to auto increment after each read or write. The display **RAM** can be directly read by the **CPU** after the correct mode and address is set.

The **A** and **B** nibble, out of **8** output lines can be entered independently or as one word, according to the mode set. Data entry to the display can be set to either left or right entry.

Left Entry (Type Writer) Mode: In the left entry mode, the first character typed appears at the extreme left, then the carriage advances one step (auto-increment), then second character appears on the right of the first character and so on.

Right Entry (Calculator) Mode: In right entry mode, the first character entered appears at the right most display position. When a second character enters, the first character shift one place to the left on the display and the second character appears to its right. At the third entry, both the first and the second characters move one position left and the third character appears again at the right most position.

IEEE-488 Bus Interface

The **IEEE - 488** standards defines a byte-serial, **8-bit** parallel, asynchronous type instrument interface. It is a document that describes the rules, specifications, timing relationships, physical characteristics etc of an interfacing technique that allows digital instruments and devices to be interconnected. It is a hardware (wires, connectors etc) that is used to implement the standard. Sixteen signals line comprise the complete **IEEE – 488** bus structure as shown below.

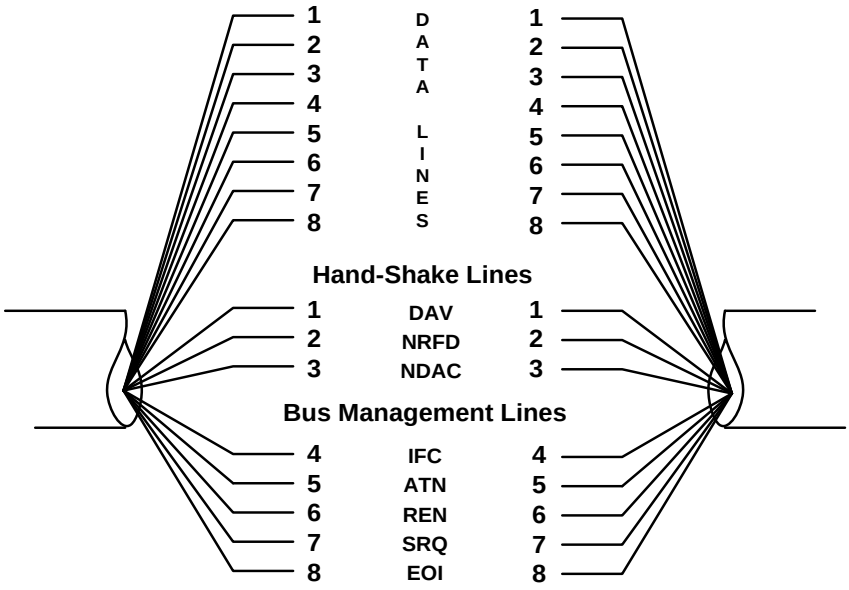


Fig: IEEE – 488 Interface

Legend:

Bus Management Lines

EOI	–	End or Identity
SRQ	–	Service Request
REN	–	Remote Enable
ATN	–	Attention
IFC	–	Interface Clear

Hand-Shake Lines

NDAC	–	Not Data Accepted
NRFD	–	Not Ready For Data
DAV	–	Data Valid

Since, it is not guaranteed by the standard that instruments will send information coded in this suggested manner, two **IEEE – 488** interconnected instruments may always be able to talk to each other, but they may not always be able to understand each other.

SERIAL VS PARALLEL COMMUNICATION

Serial Communication	S N	Parallel Communication
▪ Individual data bits and the eventual parity bit are transmitted successively i.e. in a serial manner via a single data line one after the other.	1	▪ Data word (a group of bits) is transmitted at a time parallelly over a group of wires (entire data bus). Each bit is transmitted over its own wire.
▪ The individual data word/unit is split into single bits and transferred bit by bit.	2	▪ The individual data units (i.e. data bytes) are transmitted as a whole.
▪ Simple due to reduced size of data path i.e. only a single communication link.	3	▪ Hardware required is complicated.
▪ Data transmission rate is slow.	4	▪ Data transmission rate is high.
▪ Inexpensive.	5	▪ Expensive.
▪ For long distance Transmission.	6	▪ For short distance transmission.
▪ Universal in nature and more common.	7	▪ May be impractical or impossible in many situations.
▪ Sometimes they tend to be faster than conventional parallel data transfer (clocking frequency concerned).	8	▪ ----------
▪ In some media (wireless) serial is the only form of communication e.g. RS 232 C.	9	▪ Some devices such as a CRT Terminal or a Cassette Tape etc can't designed for parallel I/O. e.g. IEE 488
▪ To transmit a word of n-bits, serial data transmission takes n times more time than taken in parallel data transmission.	10	▪ ----------
▪ The devices/peripherals commonly used for serial data transfer with the microprocessor are CRT Terminals, Printer/also used in parallel mode, cassette tapes, modems for telephone lines etc.	11	▪ The devices commonly used for parallel data transfer with microprocessor are Keyboards, 7 Segment LEDs, Data Converters and memory.

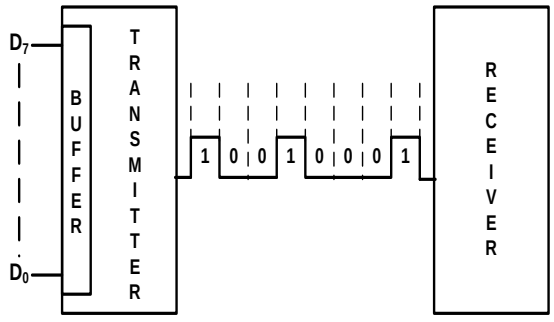


Fig: Serial Data Transmission

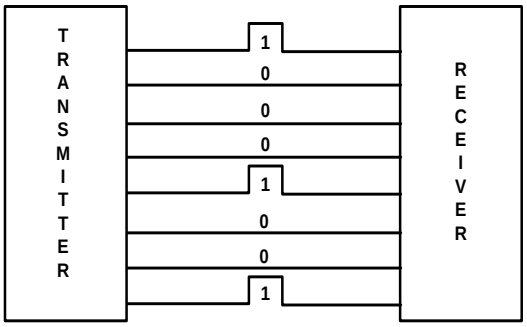


Fig: Parallel Data Transmission

8255A Programmable Peripheral Interface (PPI)

The **8255A** is a widely use, programmable, parallel **I/O** device. It can be programmed to transfer data under various conditions, from simple **I/O** to interrupt **I/O**. It is flexible, versatile and economical (when multiple **I/O** ports are required), but somewhat complex.

The **8255A** has **24 I/O** pins that can be grouped primarily in two **8-bit** parallel ports: **A** and **B**, with the remaining eight bits as port **C**. The eight bits of port **C** can be used as individual bits or be grouped in two **4-bit** ports: $\mathbf{C_{UPPER}}$ ($\mathbf{C_U}$) and $\mathbf{C_{LOWER}}$ ($\mathbf{C_L}$) as in figure given below. The functions of these ports are defined by writing a control word in the control register.

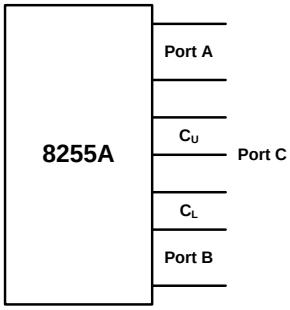


Fig: 8255A I/O Ports

The following figure shows all the functions of the **8255A**, classified according to two modes: **The Bit Set/Reset (BSR) Mode** and **The I/O Mode**. The **BSR Mode** is used to set or reset the bits in **Port C**. The **I/O Mode** is further divided into three modes: **Mode 0**, **Mode 1** and **Mode2**. In **Mode 0**, all ports function as simple **I/O** ports. **Mode 1** is a handshake mode whereby ports **A** and/or **B** use bits from port **C** as handshake signals. In the handshake mode, two types of **I/O** data transfer can be implemented: **Status Check** and **Interrupt**. In **Mode 2**, port **A** can be set up for bidirectional data transfer using handshake signals from port **C** and port **B** can be set up either in **Mode 0** or **Mode 1.**

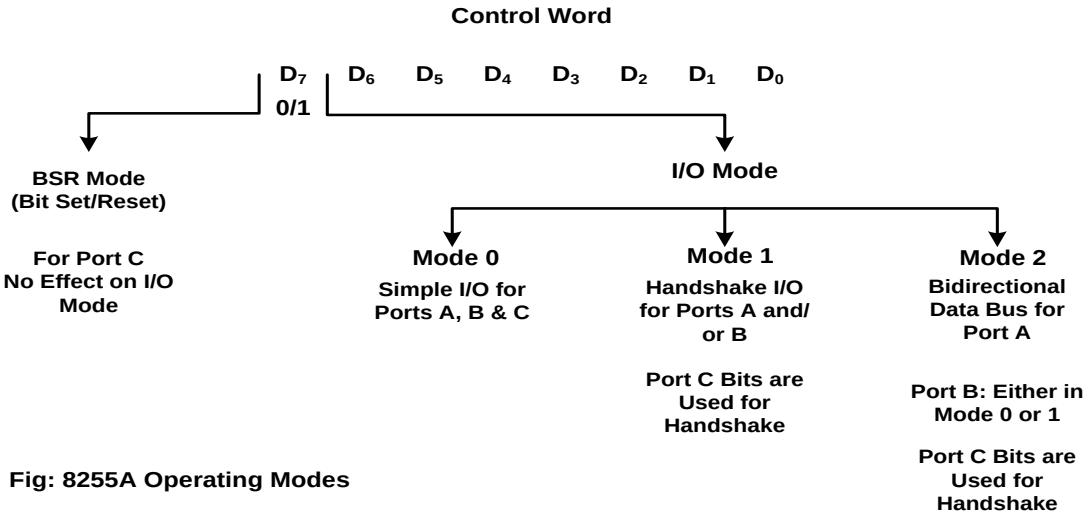


Fig: 8255A Operating Modes

The block diagram of **8255A** is shown below. It has two **8-bit** ports (**A** and **B**), two **4-bit** ports ($\mathbf{C_U}$ and $\mathbf{C_L}$), and the data bus buffer and control logic. The simplified but expanded version of the internal structure, including a control register can be shown in the next figure.

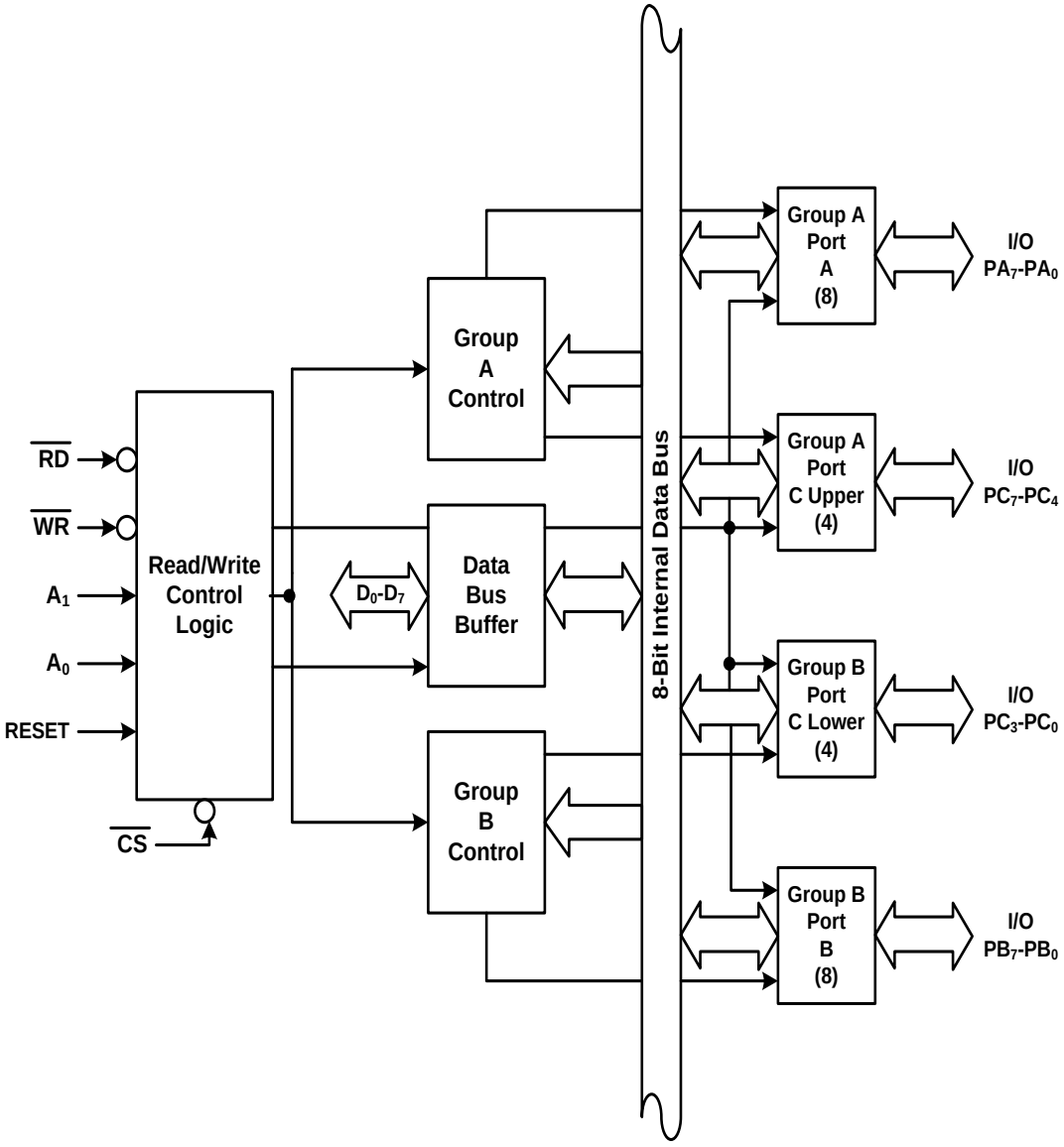


Fig: The 8255 A Block Diagram

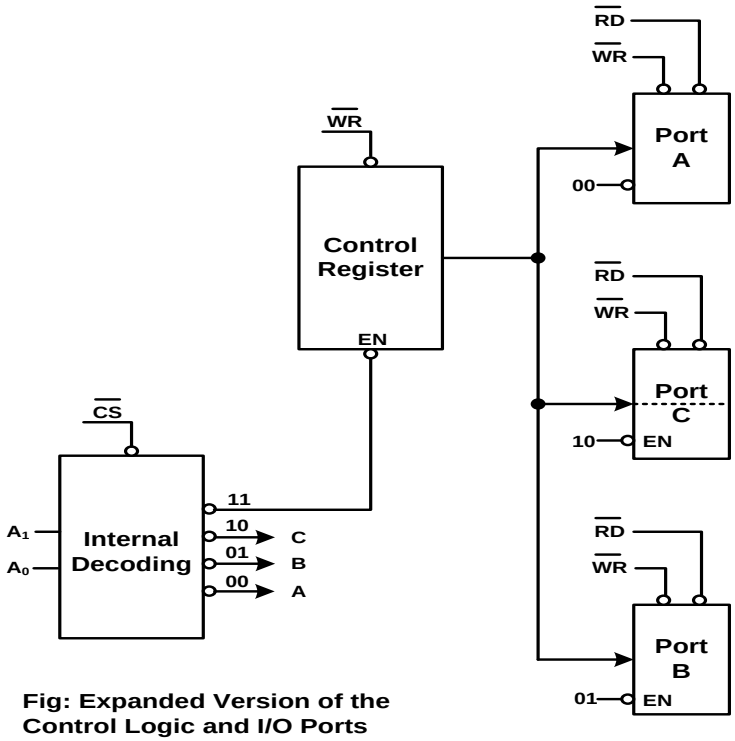


Fig: Expanded Version of the Control Logic and I/O Ports

8255A Block Diagram Description

Control Logic

The control section has six lines. Their functions and connections are as follows:

- $\overline{\textbf{RD}}$ (Read): This control signal enables the read operation. When the signal is low, the **MPU** reads data from a selected **I/O** port of **8255A.**
- $\overline{\textbf{WR}}$ (Write): This control signal enables the write operation. When the signal goes low, the **MPU** writes into a selected **I/O** port or the control register.
- **RESET** (Reset): This is an active high signal. It clears the control register and sets all ports in the input mode.
- $\overline{\textbf{CS}}$**,** $\mathbf{A_0}$ and $\mathbf{A_1}$: These are device select signals. **CS** is connected to a decoded address. And $\mathbf{A_0}$ and $\mathbf{A_1}$ are generally connected to **MPU** address lines $\mathbf{A_0}$ and $\mathbf{A_1}$ respectively.

The **CS** signal is the **Master Chip Select**, and $\mathbf{A_0}$ and $\mathbf{A_1}$ specify one of the **I/O** ports or the control register as given below:

CS	A_1	A_0	Selected
0	0	0	Port A
0	0	1	Port B
0	1	0	Port C
0	1	1	Control Register
1	x	x	8255A is not selected

I/O Ports

It has two **8-bit** ports (**A** and **B**), two **4-bit** ports (C_U and C_L).

- **Port A**
  - Can be used as an **8-bit** input port.
  - Can be used as an **8-bit** output port.
- **Port B**
  - Can be used as an **8-bit** input port.
  - Can be used as an **8-bit** output port.
- **Port C**
  - Can be used as an **8**- input or output port.
  - Can be used as **4**-bit ports.
  - Can be used to produce handshake signals for ports **A** and **B.**

Data Bus Buffer

- Bidirectional **8-bit** buffer, used to interface the **8255A** to the system data bus.
- Data is transmitted or received by the buffer upon execution of input or output instruction by the microprocessor.
- Control words and status information are also transferred through the data bus buffer.

Group A & Group B Control

- Each of the control blocks (**Group A** & **Group B**) accepts "**Commands**" from the internal data bus and issues the proper commands to its associated ports.
- **Control Group A → Port A** and **Port C_{UPPER} (C_7-C_4)**
- **Control Group B → Port B** and **Port C_{LOWER} (C_3-C_0)**

Data I/O Lines

- **D_0-D_7**
  - These are the data input/output lines for the device.
  - All information read from and written to the **8255** occurs via these **8** data lines.

Control Word

The above figure of **8255A Expanded Version of Control Logic & I/O Ports** shows a register called the control register. The contents of this register, called the control word, which specify an **I/O** function for each port. This register can be accessed to write a control word when A_0 and A_1 are at logic **1**, as mentioned in above table. The register is not accessible for a Read Operation.

Bit D_7 of the control register specifies either the **I/O** function or the Bit Set/Reset function as classified in fig: **8255 A** Operating Modes. If bit $D_7 = 1$, bits D_6-D_0 determine **I/O** functions in various mode as shown in figure below. If bit $D_7 = 0$, port **C** operates in the **Bit Set/Reset** (**BSR**) mode. The **BSR** control word does not affect the functions of ports **A** and **B**.

To communicate with peripherals through the **8255 A**, three steps are necessary.

- Determine the addresses of ports **A, B** & **C** and of the control register according to the **Chip Select Logic** and **Address Lines** A_0 and A_1.
- Write a control word in the control register.
- Write **I/O** instructions to communicate with peripherals through port **A, B** & **C.**

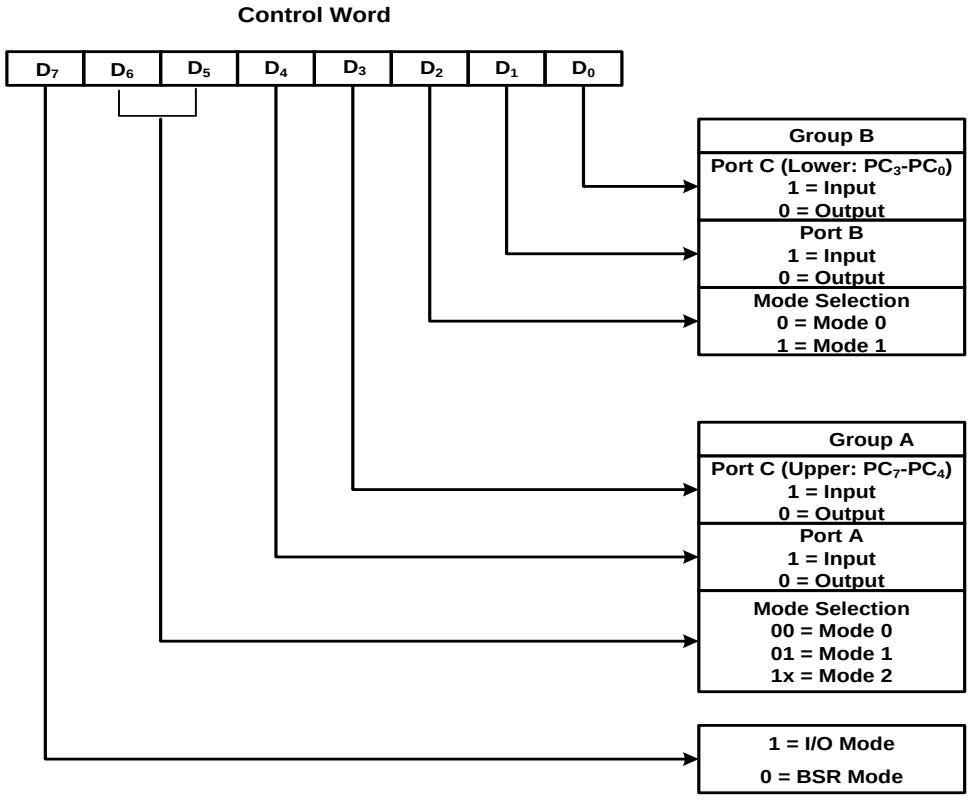


Fig: 8255 A Control Word Format for I/O Mode

Operating Modes of 8255 A PPI

- **BSR (Bit Set/Reset) Mode**

The **BSR** mode is concerned only with the eight bits of port **C,** which can be set or reset by writing an appropriate control word in the control register. A control word with bit D_7 **= 0** is recognized as **BSR** control word and it does not alter nay previously transmitted control word with bit D_7 **= 1**; thus the **I/O** operations of ports **A** and **B** are not affected by a **BSR** control word. In the **BSR** mode, individual bits of port **C** can be used for applications such as an on/off switch.

BSR Control Word

This control word, when written in the control register, sets or resets one bit at a time, as specified in following figure.

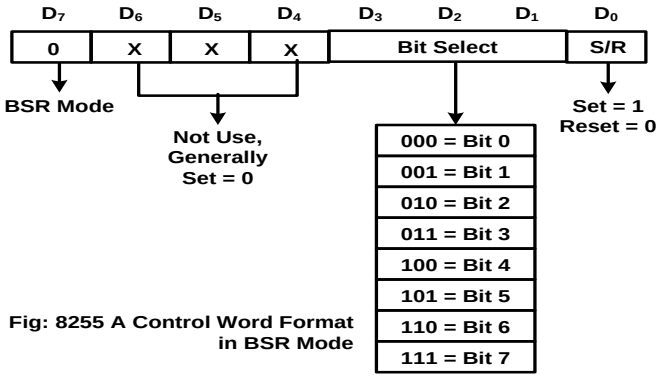


Fig: 8255 A Control Word Format in BSR Mode

- **Input/Output Mode**

Mode 0: Simple Input or Output

In this mode, ports **A** and **B** are used as two simple **8-bit I/O** ports and port **C** as two **4-bit** ports. Each port (or half-port, in case of **C**) can be programmed to function as simply an input port or an output port. The input/output features in mode **0** are as follows:

- Outputs are latched.
- Inputs are not latched.
- Ports do not have handshake or interrupt capability.

Mode 1: Input or Output with Handshake

In this mode, handshake signals are exchanged between the **MPU** and peripherals prior to data transfer. The features of this mode include the following:

- Two ports (**A** and **B**) function as **8-bit I/O** ports. They can be configured either as input or output ports.
- Each port uses three lines from port **C** as handshake signals. The remaining two lines of port **C** can be used for simple **I/O** functions.
- Input and Output data are latched.
- Interrupt logic is supported.

Mode 2: Bidirectional Data Transfer

This mode is used primarily in applications such as data transfer between two computers or floppy disk controller interface. In this mode, port **A** can be configured as the bidirectional port and port **B** either in **Mode 0** or **Mode 1**. Port **A** uses five signals from port **C** as handshake signals for data transfer. The remaining three signals from port **C** can be used either as simple **I/O** or as handshake for port **B**.

8255 A Chip Select Logic & I/O Port Addresses

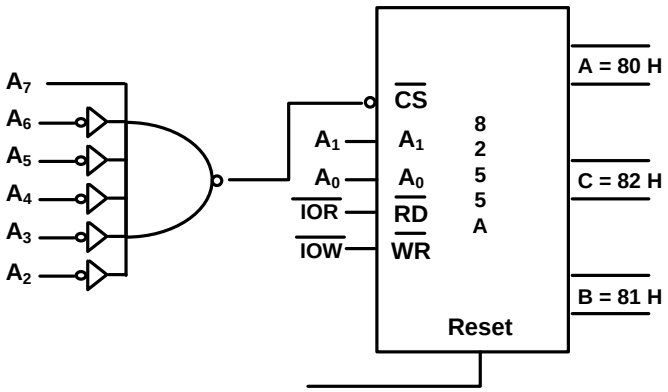


Fig: 8255 A Chip Select

$\overline{CS}$								Hex Address	Port
A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0		
1	0	0	0	0	0	0	0	= 80 H	A
						0	1	= 81 H	B
						1	0	= 82 H	C
						1	1	= 83 H	Control Register

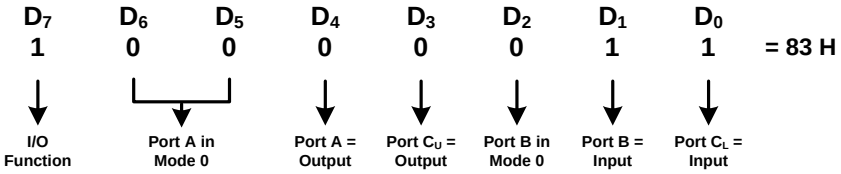
Fig: 8255 A I/O Port Addresses

Example 1: I/O Mode
Write a program to read the **DIP** switches and display the reading from port **B** at port **A** and from port $\mathbf{C_L}$ at port $\mathbf{C_U}$.

Solution
1. Port Addresses: This is a memory-mapped **I/O**; when the address line $\mathbf{A_{15}}$ is high, the **Chip Select** line is enabled. Assumming all don't care lines are at logic **0**, the port addresses are as follows:

Port A	=	**8000H (A_1 = 0, A_0 = 0)**
Port B	=	**8001H (A_1 = 0, A_0 = 1)**
Port C	=	**8002H (A_1 = 1, A_0 = 0)**
Control Register	=	**8003H (A_1 = 1, A_0 = 1)**

2. Control Word:



3. Program

```
MVI A, 83 H     ; Load Accumulator with the control word
STA 8003 H      ; Write word in the control register to initialize the ports
LDA 8001 H      ; Read switches at port B
STA 8000 H      ; Display the reading at port A
LDA 8002 H      ; Read switches at port C
ANI 0F H        ; Mask the upper four bits of port C; these bits are not input data.
RLC             ; Rotate and place data in the upper half of the accumulator.
RLC
RLC
RLC
STA 8002 H      ; Display data at port CU.
HLT
```

4. Program Description: The circuit shown below for this problem is designed for memory-mapped **I/O**; therefore, the instructions are written as if all the **8255A** ports are memory location. The ports are initialized by the control word 83H in the control register. The instructions **STA** and **LDA** are equivalent to the instructions **OUT** and **IN** respectively.

In this example, the low four bits of port **C** are configured as input and the high four bits are configured as output; even though port **C** has one address for both halves $\mathbf{C_U}$ and $\mathbf{C_L}$ (**8002H**). Reand and Write operations are differentiated by the control signals **MEMR** and **MEMW**. When the **MPU** reads port **C** (e.g. **LDA 8002H**), it receives eight bits in the accumulatro. However, the high-order its ($\mathbf{D_7}$-$\mathbf{D_4}$) must be ignored because the input data bits are in $\mathbf{PC_3}$-$\mathbf{PC_0}$. To displaythese bits at the upper half of port **C**, bits ($\mathbf{PC_3}$-$\mathbf{PC_0}$) must be shifted to $\mathbf{PC_7}$-$\mathbf{PC_4}$.

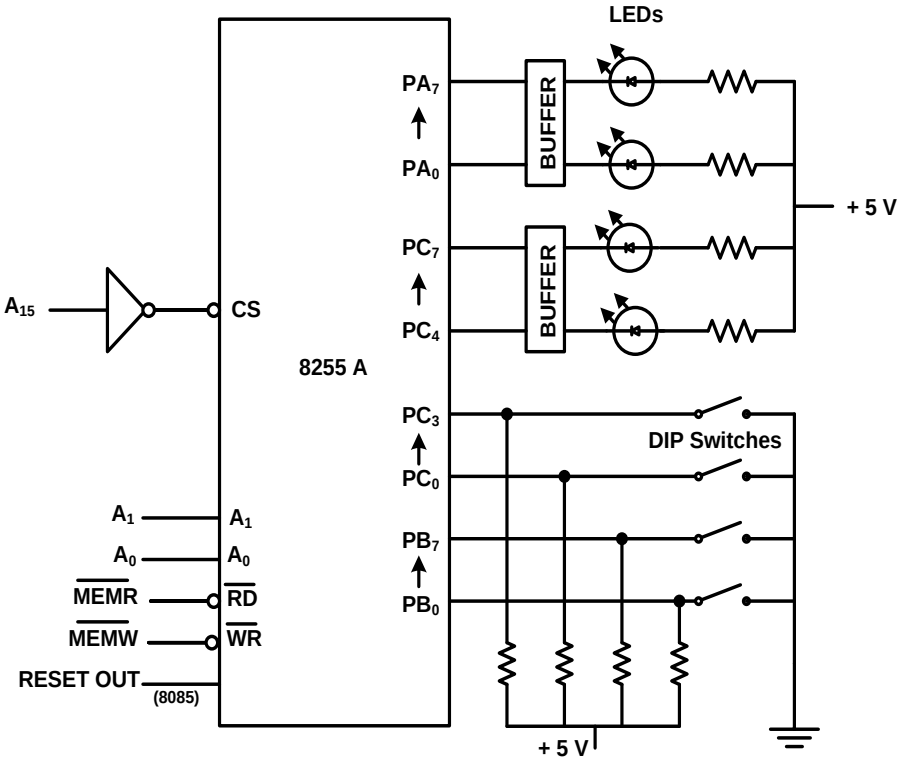


Fig: Interfacing 8255A I/O Ports in Mode 0

Example 2: BSR Mode
Write a **BSR** control word subroutine to set bits PC_7 and PC_3 and reset them after **10** ms. Use the schematic in the figure: **8255A Chip Select Logic and I/O Port Address** and assume that a delay subroutine is available.

Solution

1. BSR Control Words

	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	
To Set Bit PC_7	0	0	0	0	1	1	1	1	= 0FH
To Reset Bit PC_7	0	0	0	0	1	1	1	0	= 0E H
To Set Bit PC_3	0	0	0	0	0	1	1	1	= 07 H
To Reset Bit PC_3	0	0	0	0	0	1	1	0	= 06 H

2. Port Address

Control Register Address = 83 H

3. Subroutine

```
BSR:  MVI A, 0F H     ; Load byte in accumulator to set PC7
      OUT 83 H        ; Set PC7 = 1
      MVI A, 07 H     ; Load byte in accumulator to set PC3
      OUT 83H         ; Set PC3 = 1
      CALL DELAY      ; This is a 10-ms delay
      MVI A, 06H      ; Load accumulator with the byte to reset PC3
      OUT 83 H        ; Reset PC7
      MVI A, 0E H     ; Load accumulator with the byte to reset PC7
      OUT 83 H        ; Reset PC7
      RET
```

4. Analysis

- To set/reset bits in port **C**, a control word is written in the **Control Register** and not in port **C**.
- A **BSR** contrl word affects only one bit in port **C**.
- The **BSR** control word does not affect the **I/O** mode.

6 CHAPTER

Interrupts

Contents:

- Interrupt: Introduction
- 8085 Interrupts
- Interrupt Service Routine & Its Requirement
- Interrupt Structure
- 8086 Interrupt System
- 8259A Programmable Interrupt Controller

An interrupt is process, getting attention of the microprocessor to perform some another task while it is busy in doing something else.

When microprocessor is busy in executing one program, it can be stop for the execution of the current program temporarily and can deviates its execution process to another one. After performing the requested task, it returns to the previous program from where it has deviated. This process is called interrupt. The interrupt process is asynchronous, that means it can be initiated at any time without reference to the system clock. However, the response to an interrupt request is directed or controlled by the microprocessor.

8085 Interrupts

Microcomputers can transfer data to or from an external device using the interrupt **I/O**. In order to accomplish this, the microcomputer uses a pin on the microprocessor chip called the **Interrupt Pin** (**INTR**). The external **I/O** device is connected to this pin.

When the device wants to communicate with the microcomputer, it makes the signal on the interrupt line high or low, depending on the microcomputer. The microcomputer responds by sending signal via its pin called Interrupt Acknowledgement (**INTA**).

Mainly, the interrupts are typically classified into two types, they are:

- **External Interrupt**
- **Internal Interrupt**

External Interrupt

An external interrupt is caused by a device that is external to the processor i.e. these are usually initiated via the microprocessor's interrupt pins by external devices such as A/D converters. External Interrupt can be further divided into two types.

- **MI** (Maskable Interrupt)
- **NMI** (Non Maskable Interrupt)

An interrupt is called **Maskable Interrupt**, if it can be disabled or delayed or rejected i.e. a Maskable Interrupt can be enabled or disabled by executing instructions such as **EI** (E**nable Interrupt**) and **DI** (**Disable Interrupt**). If the microprocessor's interrupt flag is disabled, it ignores a Maskable interrupt. The **8085** provides two instructions to enable or disable interrupts.

- **EI** (**Enable Interrupt**): This is a one byte instruction. This instruction sets the interrupt enable flip flop and enables all the interrupts. The interrupt request is disable by system reset or by interrupt response process.
- **DI** (**Disable Interrupt**): This is also one byte instruction. This instruction resets the interrupt enable flip flop and hence disable all the interrupt except **TRAP**.

When the interrupt is disabled, the interrupt enable flip flop is reset to **0** and no **Maskable Interrupts** are recognized by the processor. **RST 7.5, RST 6.5, RST 5.5** and **INTR** are **Maskable Interrupts.**

If the interrupt request cannot be disabled and microprocessor must response to it, such interrupt is called **Non Maskable Interrupt**. The microprocessor can ignore or delay a **Maskable Interrupt** request if it is performing some critical task; however, it has to respond to a **Non Maskable Interrupt** request immediately. The **8085** has one **Non Maskable Interrupt TRAP**

The **Non Maskable Interrupt** has the higher priority over the **Maskable Interrupt** and cannot be enabled or disabled by instructions. This means that if both the **Maskable** and **Non Maskable Interrupts** are activated at the same time the then processor will service the **Non Maskable Interrupt** first. This gives the concept of **Interrupt Priority** which will be discussed in later. So, the priorities for servicing the interrupt requests among a number of devices depend on the type of interrupt and the design of the specific microcomputer system (both hardware and software).

Internal Interrupt

Internal interrupts are activated internally by exceptional conditions such as overflow, division by zero and execution of an illegal Opcode. They are handled in the same way as external interrupt. Internal Interrupts can also be activated by execution of **TRAP**.

When **TRAP** instruction is executed, the processor is interrupted and serviced in the same as External Interrupts. This interrupt is useful for operating the microprocessor in **Single Step Mode** and hence important in **Debugging.**

The difference between internal and external interrupt is that the internal interrupt is initiated by some exceptional conditions caused by the program itself rather than by an external events. Internal interrupts are synchronous with the program, while external interrupts are asynchronous. If the program is return, the internal interrupt will occur in the same place each time. External interrupts depend on external conditions that are independent of the program being executed at the time.

Hardware & Software Interrupts

An interrupt request on one of the interrupt lines of the **8085** by peripheral device is called hardware interrupt. It is generated by peripheral devices. The hardware interrupt lines are **TRAP, RST 7.5, RST 6.5** & **RST 5.5**, which have fixed vectored address.

But in case of the **INTR** interrupt line, it has eight vectored addresses and the hardware in the interrupting device decides where to deviate the execution. But it is possible to deviate the execution of control to any one of the eight vectored address by using the **RST** (**Restart**) instruction in the program. These instructions are: **RST 0** to **RST 7**. This is termed as software interrupt in **8085**. Software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call. It can be used by the programmer to initiate an interrupt procedure at any desired point in the program. The most common used of software interrupt is associated with a supervisor call instruction. This instruction provides means for switching from a **CPU** user mode to the supervisor mode. Certain operations in the computer may be assigned to the supervisor mode only, as for example, a complex input or output transfer procedure.

Vectored & Non Vectored Interrupts

There are two ways of redirecting the execution to the **ISR** depending on whether the interrupt is **Vectored** or **Non-Vectored**.

In **Vectored Interrupt**, the address of the subroutine is already known to the microprocessor i.e. they are automatically transferred to the specific location on memory page **00H** without any external hardware. They don't require **INTA** (**Interrupt Acknowledge**) signal or an input port. The necessary hardware is already implemented inside the **8085**. **RST 5.5, RST 6.5, RST 7.5** and **TRAP** all are automatically vectored. **TRAP** has the highest priority followed by **RST 7.5, RST 6.5** and **RST 5.5**. The interrupts and their call locations are:

Interrupts	Call Location
TRAP	0024 H
RST 7.5	0030 H
RST 6.5	0034 H
RST 5.5	0020 H

In **Non-Vectored Interrupt**, the device will have to supply the address of the subroutine to the microprocessor i.e. they require external hardware to supply a call location to restart the execution. The **8085** has only **Non Vectored Interrupt** i.e. **INTR** (**Interrupt Request**).

- **INTR**

The **INTR** is different from other four interrupts. This interrupt is not automatically vectored as other interrupts do. When **INTR** is requested, an external hardware connected to the data bus delivers one restart instruction out of eight restart instruction discussed below and according to which the microprocessor is vectored or transferred to the particular memory location.

Mnemonics	Binary Code								HEX Code	Call Location in HEX
	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0		
RST 0	1	1	0	0	0	1	1	1	C7	0000 H
RST 1	1	1	0	0	1	1	1	1	CF	0008 H
RST 2	1	1	0	1	0	1	1	1	D7	0010 H
RST 3	1	1	0	1	1	1	1	1	DF	0018 H
RST 4	1	1	1	0	0	1	1	1	E7	0020 H
RST 5	1	1	1	0	1	1	1	1	EF	0028 H
RST 6	1	1	1	1	0	1	1	1	F7	0030 H
RST 7	1	1	1	1	1	1	1	1	FF	0038 H

These are **1-byte Call** instructions that transfer the program execution to a specific memory location on page **00H**. The **RST** instructions are executed in a similar way to hat of **Call** instructions. The address in the **Program Counter** (meaning the address of the next instruction to an **RST** instruction) is stored on the stack before the program execution is transferred to the **RST** call location. When a processor encounters a Return instruction in the subroutine associated with the **RST** instruction, the program returns to the address that was stored on the stack.

For example to implement the instruction **RST 5**, a circuit can be used as shown in below; where **RST 5** is decoded, a **1-byte Call** instruction to location **0028H.**

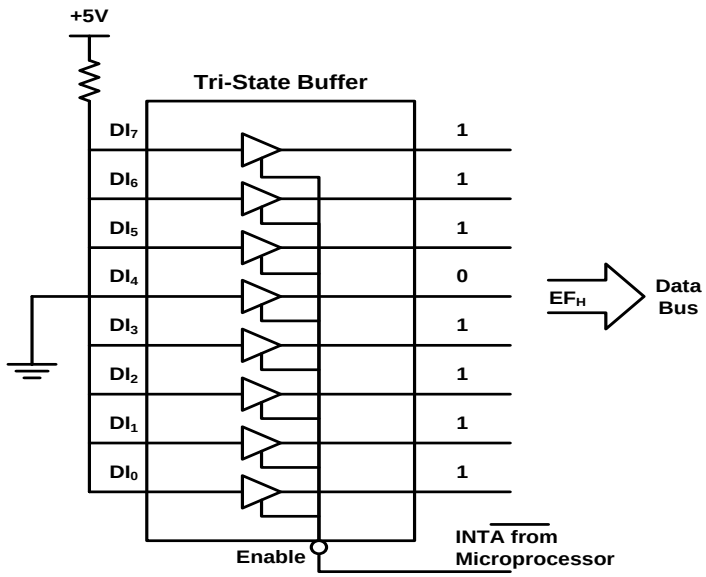


Fig: A Circuit to Implement the Instruction RST 5

- **TRAP**

TRAP, a **Non Maskable Interrupt** known as **NMI**. It has the highest priority among the interrupt signals, it need not be enabled and it cannot be disabled. It is level and edge sensitive, meaning that the input should go high and stay high to be acknowledged. It cannot be acknowledged again until it makes a transition from high to low to high. When this interrupt is triggered, the program control is transferred to location **0024H** without any external hardware or the interrupt enable instruction **EI**. **TRAP** is generally used for such critical events as power failure and emergency shut-off.

- **RST 7.5, 6.5 & 5.5**

These **Maskable Interrupts** are enabled under program control with two instructions: **EI** (**Enable Interrupt**) and **SIM** (**Set Interrupt Mask**).

RST 7.5: This is positive-edge sensitive and can be triggered with a short pulse. The request is stored internally by the **D** flip-flop until the microprocessor responds to eh request or until it is cleared by Reset or by bit D_4 in **SIM** instruction.

RST 6.5 & RST 5.5: These interrupts are level sensitive, meaning that the triggering level should be on until the microprocessor completes the execution of the current instruction. If the microprocessor is unable to respond to these requests immediately, they should be stored or held by external hardware.

SIM Instruction

This is a multipurpose **1-byte** instruction and used to implement the **8085** interrupts (**RST 7.5, 6.5 & 5.5**) and **Serial Data Output** (**SOD**). **SIM** Instruction can be used fro three different functions:

- One function is to set mask for **RST 7.5, 6.5** & **5.5** interrupts. This instruction reads the content of the accumulator and enables or disables the interrupts according to the content of the accumulator. Bit D_3 is a control bit and should be equal to logic level **1** for bits D_0, D_1 and D_2 to be effective. Logic **0** on D_0, D_1 and D_2 will enable the corresponding interrupts and logic **1** will disable the interrupts.

- The second function is to reset **RST 7.5** flip flop. Bit D_4 is additional control for **RST 7.5**. If D_4 = **1**, **RST 7.5** is reset. This is used to override or ignore **RST 7.5** without servicing it.

- The third function is to implement serial **I/O**. Bits D_7 and D_6 of the accumulator are used for serial **I/O** and do not affect the interrupts. Bit D_6 = **1** enables the serial **I/O** and bit D_7 is used to transmit (output) bits.

The instruction interprets the accumulator contents as follows:

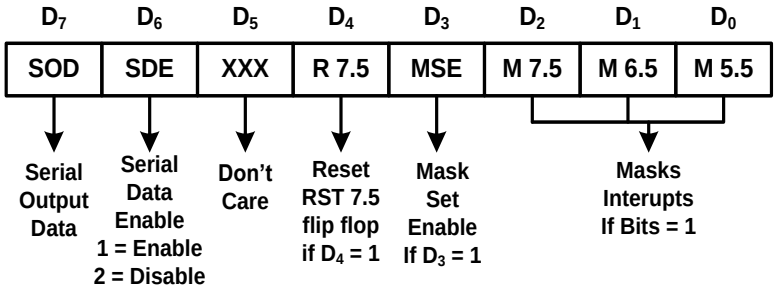


Fig: SIM Instruction

Because there are several interrupt lines, when one interrupt request is being served, other interrupt requests may occur and remain pending. The **8085** has an additional instruction called **RIM** (**Read Interrupt Mask**) to sense these pending interrupts.

The functions of **RIM** instruction is discussed below:

RIM Instruction

This is also multipurpose **1** byte instruction used to read the status of interrupts **RST 7.5, 6.5** & **5.5** and to read serial data input bit. **RIM** instruction can be used for the following functions:

- This instruction loads the accumulator with **8-bits** indicating the current status of the interrupt masks.
- Bit D_4, D_5 and D_6 identify the pending interrupts.
- To receive serial data Bit D_7 is used.

The **RIM** instruction loads eight bits in the accumulator with the following interpretations.

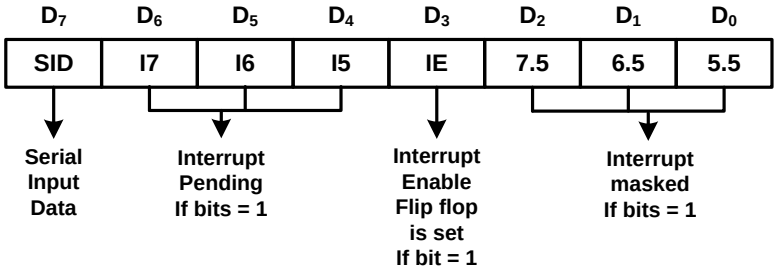


Fig: RIM Instruction

Interrupt Priority

Since, the **8085** has five interrupt lines (**TRAP, RST 7.5, RST 6.5, RST 5.5** & **INTR**). We know all the interrupts can be masked except **TRAP**. Microprocessor responses to interrupt instantly when interrupt request appears on one of interrupt line. If two or more than two interrupts request appears at the same time, there will be confusion for the microprocessor that which request to serve first. But this problem is solved by assigning priority to each interrupt lines. The **TRAP** has highest priority followed by **RST 7.5, RST 6.5, RST 5.5** & **INTR** respectively.

Data transfer between the **CPU** and an **I/O** device is initiated by the **CPU**. However, the **CPU** cannot start the transfer unless the device is ready to communicate with the **CPU**. The readiness of the device can be determined from and interrupt signal. The **CPU** responds to the interrupt request by storing the return address from **PC** into a memory stack and then the program branches to a service routine that process the required transfer.

In microcomputer a number of **I/O** device are attached to the processor, with each device being able to originate an interrupt request. The first task of the interrupt system is to indemnify the source of the interrupt. There is also the possibility that several sources will request service simultaneously. In this case the system must also decide which device to service first.

An interrupt priority is a system that established a priority over the various sources to determine which condition is to be serviced first when two or more request arrive simultaneously. The system may also determine which conditions are permitted to interrupt the computer while another interrupt is being serviced. Higher-priority interrupt levels are assigned to request which, if delayed or interrupted, could have serious consequences. Device with higher speed transfers such as magnetic disks are given high priority, and slow devices such as keyboards receive low priority. When two devices interrupt the processor at the same time, the processor services the device with the higher priority at first.

There are mainly two ways of servicing multiple interrupts, they are: **Polled Interrupt and Chained (Vectored) Interrupt.** The details on these two categories are explained later.

Interrupts Response

An interrupt is considered to be an emergency signal that may be services. The microprocessor may respond to it as soon as possible. Responding to an interrupt may be immediate or delayed depending on whether the interrupt is **Maskable** or **Non Maskable** and whether interrupts are being masked or not.

What happens when microprocessor is interrupted?

When the microprocessor receives an interrupt signal, it suspends the currently executing program and jumps to an **Interrupt Service Routine** (**ISR**) to respond to the incoming interrupt. Each interrupt will most probably have its own **ISR**. When a device interrupts, it actually wants the microprocessor to give a service which is equivalent to asking the microprocessor to call a subroutine.

The **8085** interrupt process can be summarized as follows:

Step 1: The interrupt process should be enabled by writing the instruction **EI** in the main program.
Step 2: **MPU** checks the **INTR** line during the execution of each instruction.
Step 3: If **INTR** line is high and interrupt is enabled, **MPU** completed the current instruction, disable the interrupt and sent an acknowledge signal **INTA**.
Step 4: The signal **INTA** is used to insert a restart (**RST**) instruction through external hardware. The **RST** instruction transfers the program control to specific memory location.
Step 5: When microprocessor receives **RST** instruction, it saves the memory address of next instruction on the stack. The program control is transferred to call location.
Step 6: The **ISR** perform the specific task of the interrupt. The **ISR** must include **EI** instruction to enable the interrupt.
Step 7: At the end of **ISR**, the **RET** instruction returns the memory address from where the program was interrupted and continues the execution.

Interrupt Service Routine (ISR) & Its Requirements

A special program caused to service an interrupt request is called **Interrupt Service Routine** (**ISR**). When interrupt occurs, the **CPU** reads the **ISR** address from **Interrupt Vector Table** and program ntrol transfers to **ISR**. The **ISR** then saves the processor registers on the stack and body of **ISR** executes. When it finishes, the registers are restored and the instruction **RET** at the end of **ISR** returns the program control from where it is interrupted.

Interrupt increases the efficiency of a computer system because the external devices require the attention of the processor as needed. If the system had no interrupt, the processor would have to poll every device in the system periodically to see if any of then required attention. Many of the standard **I/O** devices generate interrupts, when they need the processor to process data being received by or being sent by these interfaces.

As interrupt is a process of getting attention of microprocessor to perform some task while it is busy in doing something else. The task or routine that the microprocessor executes in response to interrupt is termed as **Interrupt Service Routine** (**ISR**).

Before executing the **ISR**, the microprocessor needs to save the required information to resume the current program after executing the **ISR**. So the fundamental requirement of the **ISR** is that it should begin by saving the contents of all registers and flags. After the completion of the **ISR**, it needs to restore all the registers before returning to main program.

Call & Return Instructions

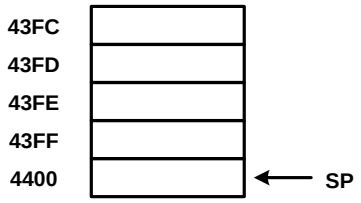
Call and **Return** instructions are associated with the subroutine technique. A '**Subroutine'** is a group of instructions that performs a subtask. The subroutine is written as a separate unit, apart from the main program and the microprocessor transfer the program execution from the main program to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

The **8085** microprocessor has two instructions to implement subroutine: **CALL** (call a subroutine) and **RET** (return to main program from subroutine). The **CALL** instruction is used in the main program to call a subroutine and **RET** instruction is used at the end of the subroutine to return to the main program. When a subroutine is called, the contents of program counter, which is the address of the instruction following the **CALL** instruction is stored in the stack and the program execution is transferred to the subroutine address. When the **RET** instruction is executed at the end of the subroutine, the memory address stored in the stack is retrieved and the sequence of execution is resumed in the main program.

Consider the following assembly language program, which illustrate what happens when the **CALL** and **RET** instructions are used.

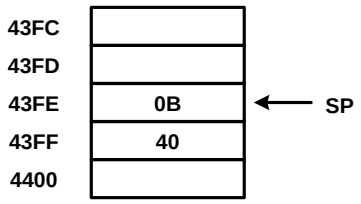
```
4000    LXI SP, 4400H       ; Main Program
4003    MVI A, 05H
4005    MVI B, 06H
4007    ADD B
4008    CALL 4050H
400B    OUT PORT A
400D    HLT

4050    MVI C, FFH          ; Subroutine
4052    DCR C
4053    JNZ 4052
4056    RET
```

The main program is loaded within the address of **4000H** to **400DH**. The subroutine starts from address **4050H** and ends at **4056H**. The first instruction **LXI SP, 4400** in the main program initializes the stack pointer register as shown in figure below:



The stack pointer points to memory location whose address is **4400H**, after execution of instruction **LXI SP, 4400H**. From address **4008H** to **400AH**, the **3**-byte **CALL** instructions is loaded, which specifies the address of the first instruction of the subroutine to be **4050H**. The address of the next instruction following the **CALL** instruction is **400BH**, which is the content of program counter at the instant when **CALL** instruction is executed.

At the instant of execution of **CALL** instruction **PC = 400BH**. After the execution of **CALL** instruction, the contents of **PC** will be loaded to the top of stack. In this process, the stack pointer decrements by one and pointes to **43FFH** and the higher order address (higher byte) of program counter which is **40** in this case will be loaded at this address. Then the stack pointer further decrements by one and the lower order address (lower byte) of program counter which is **0B** in this case will be loaded to the address **43FE**. So the new address of stack pointer will be **43FE**.



The next thing that will happen after execution of **CALL** at address **4008** is that the program counter will be loaded with the address **4050H** and the sequence of program transfers to the address **4050H** where the microprocessor executes its first instruction. At the end of the subroutine there is **RET** instruction. After execution of **RET** the following things will happen:

The microprocessor pops off the data, which is actually the address of the next instruction following the **CALL** instruction in the main program and loads into the program counter. The program counter first gets loaded with the data **0B** of the memory location **43FE** where the stack pointer is currently pointing. The stack pointer then increments by on and points to memory location **43FF**, whose contents is **40**. The program counter is then loaded with this higher order address so that the **16**-bit loaded address of **PC** will become **4000B** and the stack pointer will be again incremented by one. So execution of **RET** transfer the sequence of program control to the main program from the subroutine. So this is the process how the program control switches between the main program and subroutine when **CALL** and **RET** instruction is executed.

Interrupt Structure

The **8085 Interrupt Structure** is shown in figure below. In order to enable the interrupt, we use the instructions **EI** and **SIM**. The execution of **EI** instruction sets the **RS** flip flop and makes one of the inputs to the **AND** gate **1** to **4 HIGH**. Hence, in order for all the interrupts (except **TRAP**) to work, the interrupt system must be enabled.

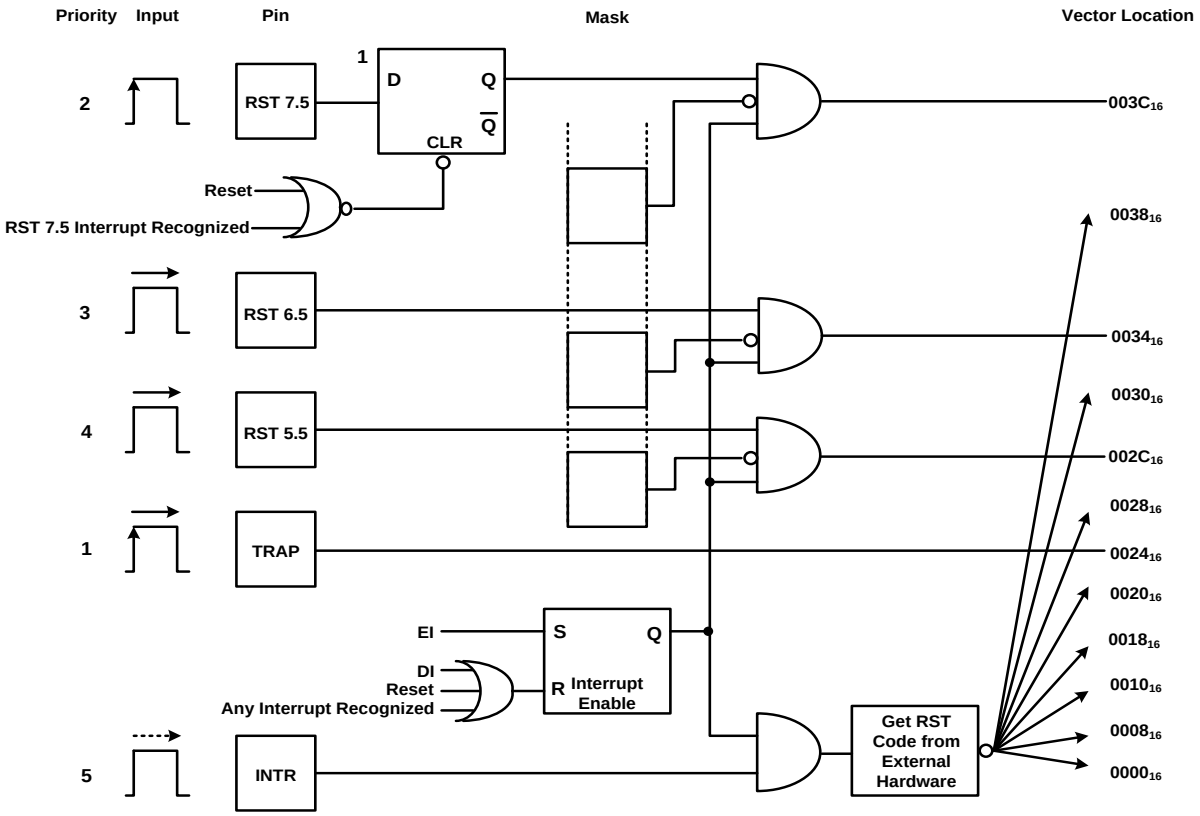


Fig: The 8085 Interrupts and Vector Locations

- Consider only the **RST 7.5** Interrupt is to be enabled, the interrupt mask bit for **RST 7.5** is set to low, making the two inputs to **AND** gate **1 HIGH**. Now, if there is the interrupt signal to **RST 7.5** pin, it sets the **D** flip flop, thus making the **O/P** of **AND** gate **1 HIGH** and enables **RST 7.5**.
- The **8085** branches to location **003CH** to service routine.
- **RST 6.5** and **RST 5.5** can be similarly explained.
- Execution of the **DI** instruction clears **RS** flip flop and disables all the interrupts except **TRAP**.

A microprocessor may be connected to more than one peripheral and most of the peripherals can originate an interrupt request. But there are only five interrupt input lines in **8085** microprocessor. So, more than one device needs to be connected in the single interrupt line. When the interrupt request generates in such line, first of all microprocessor needs to identify the interrupting device, so to handle this problem microprocessor provides two approaches:

- **Polling**
- **Chaining**

Polling

Polling is the software approach used to identify the interrupting device on the priority basis. In this method, when an interrupt generates, the execution control transfers to one common branch address and execute one general initial service routine for all devices. All the devices connected to the single interrupt request lines are polls in sequence. The order in which they are polled or tested determines the priority of each device.

The highest priority device is polled first and if this device has generates the interrupt, the execution control branches to the service routine for that device. Otherwise, the next or second higher priority device is tested and so on.

Polled interrupt is very simple but for a large number of devices, the time required to poll each device may exceed the time to service the device. In such case, the faster mechanism called vectored or chained interrupt is used.

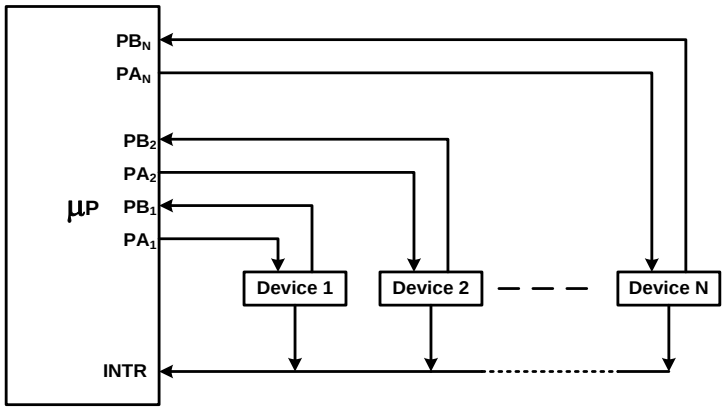


Fig: Polled Interrupt

Chaining

Chaining is the hardware approach used to identify the interrupting device on the priority basis. In this method, there is one common interrupt request line through which all the devices connected to it request the interrupt. Two or more devices that can originate the interrupt are connected in series as shown below:

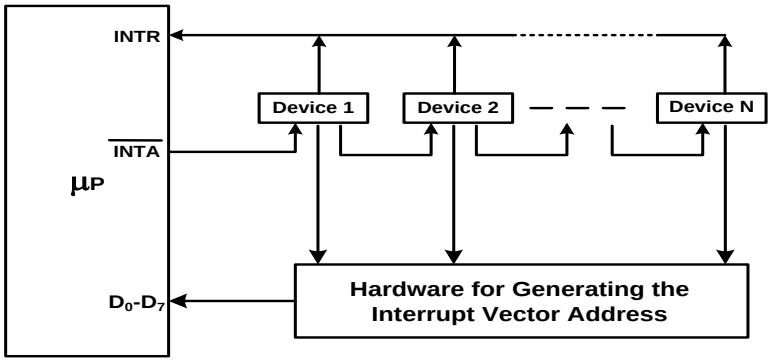


Fig: Chained Interrupt

When a device requests an interrupt, the microprocessor issues the **INTA** (Interrupt Acknowledge) signal in response. The **INTA** signal is first received by highest priority device, which is device 1 in this case. If the device has generated the interrupt, it will accept the **INTA** signal and place the vectored/transferred address on data bus. Otherwise, it will pass the **INTA** signal to next device until the **INTA** signal is accepted by the interrupting device.

The word or address placed on data bus is referred to as vector, which the microprocessor uses as a pointer to points the appropriate device service routine. This avoids the need to execute a general initial service routine first as in the case of polling.

Summary

The **8085** interrupts and their requirements are listed in the table below in order of their priority. **TRAP** has the highest priority and **INTR** has the lowest priority. It must be emphasized that an interrupt can be recognized only if the **HOLD** signal is inactive.

Interrupts	Type	Instructions	Hardware	Trigger	Vector
TRAP	Non Maskable	Independent of EI & DI	No External Hardware	Level & Edge Sensitive	**0024H**
RST 7.5	Maskable	Controlled by EI & DI Unmasked by SIM	No External Hardware	Edge Sensitive	**003CH**
RST 6.5	Maskable	Controlled by EI & DI Unmasked by SIM	No External Hardware	Level Sensitive	**0034H**
RST 5.5	Maskable	Controlled by EI & DI Unmasked by SIM	No External Hardware	Level Sensitive	**002CH**
INTR	Maskable	Controlled by EI & DI	RST Code from External Hardware	Level Sensitive	**0038H ↑ 0000H**

Table: Summary of Interrupts in 8085A Microprocessor

8086 Interrupt System

The **8086** assigns every interrupts a type code so that the **8086** can identify it. Interrupts can be initiated by:

- External Devices
- Internally]
- By **S/W** Instructions
- Exceptional Condition such as Divide by **0**, Overflow etc.

The **8086** interrupts can be classified into three types.

- **Pre Defined Interrupts**
- **User Defined Software Interrupts**
- **User Defined Hardware Interrupts**

Pre Defined Interrupt

The pre defined interrupts are called so because the interrupt vectors for these five interrupts types (which are reserved fro specific functions) are predefined by **INTEL**. They are:

Type 0: INT 0 Divide by Zero
Type 1: INT 1 Single Step
Type 2: INT 2 Non Maskable Interrupt (NMI Pin)
Type 3: INT 3 Break Point
Type 4: INT 4 Interrupt on Overflow

The user must provide the desired **IP** and **CS** values in the interrupt pointer table. The user may also initiate these interrupt through **H/W** or **S/W**.

Type 0: INT 0 (Divide by Zero)
This interrupt occur whenever the result of division overflow or whenever an attempt is made to divide by zero. It is **Non Maskable.**

Type 1: INT 1 (Single Step)
This occurs after the execution of each instruction if the **TRAP** flag bit is set. This is used to execute program one instruction at a time.

Type 2: INT 2 (Non Maskable Interrupt) (NMI Pin)
It is initiated via the **8086 NMI** pin. It is normally used for power failure. The **8086** obtains the interrupt vector address by automatically executing the **INT 2** instruction internally.

Type 3: INT 3 (Break Point)
It is used to set break point in a program for debugging.

Type 4: INT 4 (Interrupt on Overflow)
This interrupt occurs if the overflow flag is set and the **INT 0** instruction is executed.

User Defined Software Interrupts

The user can generate an interrupt by execution a two byte interrupt instruction, **INT N** (Where, **N = 0** to **FFH**). The **INT N** is not Maskable by interrupt enable flag (**IF**). The **INT N** instruction can be used to test an interrupt service routine for external interrupts.

User Defined Hardware Interrupts

The **8086 Maskable Interrupts** are initiated via the **INTR** pin. These interrupts can be enabled or disabled by **STI: Set Interrupt Flag (IF = 1) or CLI: Clear Interrupt Flag (IF = 0)** respectively. If **(IF) = 1** and **INTR** is active without the occurrence of any other interrupts, the **8086**, after completing the current instructions generates **INTA** low twice. **INTA** is only generated by **8086** in response to **INTR**.

First Pulse **INTA**, tells to **Get Ready**
Second Pulse **INTA**, **Interrupt Type** is sent.

Interrupt Vector & Interrupt Vector Table

An **Interrupt Vector** is a **4-byte** number stored in the first **1024** bytes of memory **000000H – 0003FFH**. These are **256** different d\interrupt vectors and each vector contains the address of an interrupt service routine. Each vector contains the value for the **IP** and **CS** that form the address of the interrupt service routine. The first two bytes contain **IP** and last two bytes contain **CS**.

The Interrupt Vector Table is a table which stores the interrupt vectors for **256** interrupts. It is located in first **1024** bytes of memory at address **000000H-0003FFH**. The lowest five types are dedicated to specific interrupt (**Pre-defined Interrupts**). Interrupt type **5** to **31** are reserved by the **INTEL**. Interrupt type **32** to **255** are available for user interrupt (**Hardware & Software Interrupts**).

Interrupt Procedures in 8086

Once the **8086** has the interrupt-type code (via the bus for **H/W Interrupts**, from **S/W Interrupt Instructions INT N** or from the **Predefined Interrupts**), the type code is multiplied by 4 to obtain the corresponding interrupt vector in the **Interrupt Vector Table**.

- The **Interrupt Vector** consists of **IP** and **CS**.
- During the transfer of control, **8086** pushes the **Flags** and current **Code Segment Register** and **Instruction Pointer** onto stack.
- The new **CS** and **IP** values are loaded.
- The flags **TF** and **IF** are cleared to **0**.
- No segment registers are used when accessing the **Interrupt Pointer Table.**

The 8259A Programmable Interrupt Controller (PIC)

The **8259A** is a **Programmable Interrupt Controller** designed to work with **INTEL Microprocessors: 8085, 8086** and **8088**. The **8259 Interrupt Controller** can:

- Manage eight interrupts according to the instructions written into its control registers. This is equivalent to providing eight interrupt pins on the processor in place of one **INTR (8085)** pin.
- Vector an interrupt request anywhere in memory map. However, all eight interrupts are spaced at the interval of either four or eight locations. This eliminates the major drawback of the **8085** interrupts in which all interrupts are vectored to memory location on page **00H**.
- Resolve eight levels of interrupt priorities in a variety of modes, such as fully nested mode, automatic rotation mode, and specific rotation mode.
- Mask each interrupt request individually.
- Read the status of pending interrupts, in-service interrupts, and masked interrupts.
- Be set up to accept either the level-triggered or the edge-triggered interrupt request.
- Be expanded to **64** priority levels by cascading additional **8259A**s.
- Be set up to work with either the **8085** microprocessor mode or the **8086/8088** microprocessor mode.

Block Diagram of 8259A PCI

The following figure shows the **Internal Block Diagram** of the **8259A**. It includes eight blocks: **Control Logic, Read/Write Logic, Data Bus Buffer**, Three Registers (**IRR**, **ISR** and **IMR**), **Priority Resolver** and **Cascade Buffer**. This diagram shows all the elements of a programmable device plus additional blocks. The functions of some of these blocks are explained below:

Read/Write Logic

This is a typical Read/Write control logic. When the address line $\mathbf{A_0}$ is at logic **0**, the controller is selected to write a command or read a status. The **Chip Select Logic** and $\mathbf{A_0}$ determine the port address of the controller.

Control Logic

This block has two pins: **INT** (**Interrupt**) as an output and **INTA** (**Interrupt Acknowledge**) as an input. The **INT** is connected to the interrupt pin of the **MPU**. Whenever a valid interrupt is asserted, this signal goes high. The **INTA** is the **Interrupt Acknowledge** signal from the **MPU**.

Interrupt Registers & Priority Resolver

The **Interrupt Request Register** (**IRR**) has eight input lines ($\mathbf{IR_0}$-$\mathbf{IR_7}$) for interrupts. When these lines go high, the requests are stored in the register. The **In-Service Register** (**ISR**) stores all the levels that are currently being serviced, and the **Interrupt Mask Register** (**IMR**) stores the masking bits of the interrupt lines to be masked. The **Priority Resolver** (**PR**) examines these three registers and determines whether **INT** should be sent to the **MPU**.

Cascade Buffer/Comparator

This block is used to expand the number of interrupt levels by cascading two or more **8259A**s.

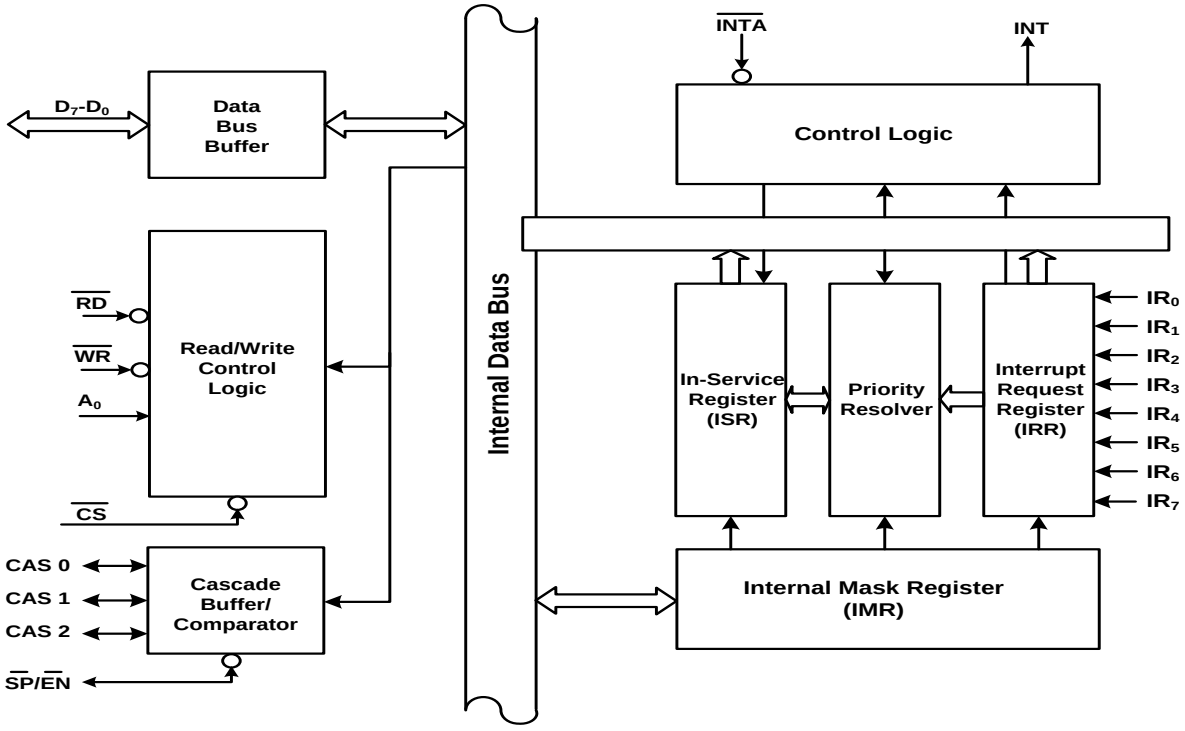


Fig: Block Diagram of 8259A PCI

Interrupt Operation

To implement interrupts, the Interrupt Enable flip flop in the microprocessor should be enabled by writing the **EI** instruction and the **8259A** should be initialized by writing control words in the control register. The **8259A** requires two types of control words: Initialization Command Words (**ICW**s) and Operational Command Words (**OCW**s). The **ICW**s are used to set up the proper conditions and specify **RST** vector addresses. The **OCW**s are used to perform functions such as masking interrupt, setting up status-read operations etc. After the **8259A** is initialized, the following sequence of events occurs when one or more interrupt request lines go high.

- The **IRR** stores the requests.
- The Priority Resolver checks three register: the **IR** for interrupt requests, the **IMR** for masking bits and the **ISR** for the interrupt request being served. It resolves the priority and sets the **INT** high when appropriate.
- The **MPU** acknowledges the interrupt by sending **INTA**.
- After the **INTA** is received, the appropriate priority bit in the ISR is set to indicate which interrupt level is being served, and the corresponding bit in the IRR is reset to indicate that the request is accepted. Then the Opcode for the **CALL** instruction is placed on the data bus.
- When the **MPU** decodes the **CALL** instruction, it places two more **INTA** signals on the data bus.
- When the **8259A** receives the second **INTA**, it places the low order byte of the **CALL** address on the data bus. At the third **INTA**, if places the high order byte on the data bus. The **CALL** address is the vector memory location for the interrupt; this address is placed in the control register during the initialization.
- During the third **INTA** pulse, the ISR bit is reset either automatically (Automatic-End-of-Interrupt-**AEOI**) or by a command word that must be issued at the end of the service routine (End-of-Interrupt-**EOI**). This option is determined by the initialization command word (**ICW**).
- The program sequence is transferred to the memory location specified by the **CALL** instruction.

7 CHAPTER

Programming Exercises

Contents:

- Programming with 8085: Examples
- Programming with 8086: Examples

Programming with 8085

Example: The Most & Frequently Asked Problems & Their Solutions

Que. 1 Add two numbers stored at memory location 2040h and 2041h and store the result at memory location 2042h.

Label	Mnemonics	Comments
	LXI H, 2040h	(H)(L) = 2040h
	MOV A, M	(A) ← ((H)(L))
	INX H	Point to Next Data
	ADD M	(A) = (A) + ((H)(L))
	INX H	Store Data in Next Memory Location.
	MOV M, A	((H)(L)) ← A
	HLT	

Que. 2 Clear the memory locations from 2040h to 204Fh.

Label	Mnemonics	Comments
	LXI H, 203Fh	(H)(L) = 203Fh (As a Pointer)
	MOV C, M	C as a Counter
	XRA A	A = 0
Loop	INX H	Increment Pointer
	MOV M, A	((H)(L)) ← A
	DCR C	Decrement Counter
	JNZ Loop	Continue if Counter ≠ 0

Que. 3 Sum the given series of numbers. The length is given in memory location 203Fh and the series itself starts from 2040h. Store the result at 2060h.

Label	Mnemonics	Comments
	LXI H, 203Fh	(H)(L) = 203Fh (As a Pointer)
	MOV C, M	C as a Counter
	XRA A	Sum = 0
Loop	INX H	Increment Pointer
	ADD M	(A) → (A) + ((H0(L))
	INX H	Increment Pointer
	DCR C	Decrement Counter
	JNZ Loop	Continue if Counter ≠ 0
	STA 2060h	Store Result
	HLT	

Que. 4 Write a program to find the square of a given number.

Label	Mnemonics	Comments
	LXI D, 2060h	(D)(E) = 2060h (Base Address)
	LDA 2040	Get Data
	MOV L, A	Prepare H & L to
	MVI H, 00h	Base Address
	DAD D	Get Effective Add. In H & L
	MOV A, M	Get Result in A
	STA 2042h	Store the Result
	HLT	

Que. 5 Find the maximum in a given series of data. The length is given in memory location 203Fh and the series itself starts from 2040h. Store the result at 2060h.

Label	Mnemonics	Comments
	LXI H, 203Fh	(H)(L) = 203Fh (As a Pointer)
	MOV C, M	C as a Counter
	DCR C	Comparison is 1 Less
	XRA A	Maximum = 0
Loop	INX H	Increment Pointer
	CMP M	Compare Ac^r with Next Data
	JNC Next	Jump If No Change
	MOV A, M	Change If Ac^r Less
Next	DCR C	Decrement Counter
	JNZ Loop	Continue if Counter ≠ 0
	STA 2060h	Store Result
	HLT	

Que. 6 Write a program to copy 16 data from 2040h onwards to 2060 onwards.

Label	Mnemonics	Comments
	LXI H, 203Fh	(H)(L)=203Fh (As a Pointer 1)
	MOV C, M	C as a Counter
	LXI D, 2060h	(D)(E)=2060h (As a Pointer 2)
Loop	MOV A, M	(A) ← ((H)(L))
	STAX D	(A) → ((D)(E))
	INX H	Increment Pointer 1
	INX D	Increment Pointer 2
	DCR C	Decrement Counter
	JNZ Loop	Continue if Counter ≠ 0
	HLT	

Que. 7 Write a program to Ex-or two data without using Ex-or instruction.

Label	Mnemonics	Comments
	LXI H, 2040h	(H)(L)=2040h
	MOV A, M	(A) ← ((H)(L))
	MOV B, A	Keep a Copy in Register B
	CMA	Complement contents of Ac^r
	INX H	Point to Next Mem. Location
	ANA M	AND the Ac^r with 2^{nd} Data
	MOV C,A	Put ANDed Result in Reg: C
	MOV A,M	2^{nd} Number in Ac^r
	CMA	Complement contents of Ac^r
	ANA B	AND Ac^r with 1^{st} Data
	ORA C	OR the contents of Ac^r with the Contents of Reg: C
	INX H	Point to Next Mem. Location
	MOV M, A	Store the Result
	HLT	

Que. 8 Write a program to generate the Fibonacci Series.

Label	Mnemonics	Comments
	LXI H, 203Fh	(H)(L)=203Fh
	MOV C, M	(C) ← ((H)(L)) (As a Counter)
	INX H	Point to First Data
	MOV A, M	(A) ← (303Fh)
	INX H	Point to Next Data
Loop	ADD M	(A) = (A) + (203Fh)
	INX H	Point to Next Mem. Loc. For
	MOV M, A	Storing Next Fibonacci No
	DCX H	Get Previous Fibonacci No
	MOV A, M	(A) ← ((H)(L))
	INX H	Point to Latest Fibonacci No
	DCR C	Decrement Counter
	JNZ Loop	Continue if Counter ≠ 0
	HLT	

Que. 9 Find the number of negative, positive and zero elements in a given series of data. The length is given in memory location 203Fh and the series itself starts from 2040h. Store the result at 2060 onwards.

Label	Mnemonics	Comments
	LXI H, 203Fh	(H)(L)=203Fh (As a Pointer 1)
	MOV C, M	C as a Counter
	XRA A	Clear Ac^r
	MOV B, A	No of Positive Element = 0
	MOV D, A	No of Negative Element = 0
	MOV E, A	No of Zero Element = 0
Loop	INX H	Increment Pointer
	CMP M	Compare Ac^r with Next Data
	JNZ Neg	Jump If No is Negative
	INR E	Increment Zero Counter
	JMP Ahead	Check Next Number
Neg	MVI A, 7Fh	Max, +Ve No for Comparison
	CMP M	Compare Ac^r with Next Data
	JNC Pos	If No Carry, Increment +Ve Counter
	INR D	Else Increment -Ve Counter
Pos	INR B	Increment +Ve Counter
Ahead	XRA A	Initialize Ac^r for Next Comparison
	DCR C	Decrement Counter
	JNZ Loop	Continue if Counter ≠ 0
	LXI H, 2060h	Pointer to Store Data
	MOV M, B	Store +Ve Counter
	INX H	Increment Pointer
	MOV M, D	Store –Ve Counter
	INX H	Increment Pointer
	MOV M, E	Store Zero Counter
	HLT	

Que. 10 Write a program to multiply 05h and 08h.

Label	Mnemonics	Comments
	MVI A, 00h	Clear the Acr Contents.
	MVI B, 08h	(B) ← 08h
	MVI C, 05h	(C) ← 05h (As a Counter)
Loop	ADD B	(B) + (A)
	DCR C	Decrement Counter
	JNZ Loop	Continue if Counter ≠ 0
	STA C000h	Store the Result

Que. 11 Sort the given list in ascending order. The length of the series is given memory location 203Fh and the series itself starts from 2040 onwards.

Label	Mnemonics	Comments
	LXI H, 203Fh	(H)(L)=203Fh (As a Pointer 1)
	MOV C, M	C as a Counter
	DCR C	Comparison is 1 Less
	MOV B, C	Keep a Copy in Reg: B
	INX H	Get Starting Address of Data
Loop	MOV A, M	Assume First Data is Lowest
	INX H	Increment Pointer
	CMP M	Compare A with M
	JC Next	If (A)<((H((L)), Interchange
	MOV D, M	Contents of Two Memory
	MOV M, A	Location
	DCX H	Decrement Pointer
	MOV M, D	((H)(L)) ← D
	INX H	Increment Pointer
Next	DCR B	Decrement Counter
	JNZ Loop	Continue if Counter ≠ 0
	MOV C, B	Repeat It for One Less
	JNZ Again	Jump to Again if Counter ≠ 0
	HLT	
Again	LXI H, 2040h	((H)(L)) ← 2040h
	JMP Loop	Continue if Counter ≠ 0

Que. 12 Write a program to display the absolute value.

Label	Mnemonics	Comments
	MVI A, Any	Get Any Data Byte.
	ORA A	Flag Set
	JM Loop	Jump on Minus to Loop
	OUT 01h	Display at Port 1
	HLT	
Loop	CMA	Find 2's Complement
	ADI 01h	
	OUT 01h	
	HLT	

Que. 13 Convert binary number stored at 2040h to 4 digit BCD number. Store the BCD result at 2041h and 2042h.

Label	Mnemonics	Comments
	LDA 2040h	(A) = 2040h
	MOV B, A	Keep a Copy in Reg: B
	LXI D, 0000h	BCD Number = 0
	MVI C, FFh	C ← FFH
	ANA C	If Number = 0 then
	JZ Done	Conversion Not Needed
Loop	DCR B	Decrement Binary Counter
	MOV A, E	Increment BCD Counter in
	INRA	BCD Mode
	DAA	Decimal Adjust Ac^r
	MOV E, A	(D) ← (A)
	JNC Next	If No Carry Jump to Next
	INR D	Increment MSD (BCD)
Next	MOV A, C	Repeat the Loop until No = 0
	ANA B	AND B with Ac^r Contents
	JNZ Loop	Continue if Counter ≠ 0
	XCHG	Exchange DE & HL Reg:Pair
Done	SHLD 2041h	(L) →(2041h), (H) →(2042h)
	RST 1	

Que. 14 Writer a program to load the data byte A8H in register C. Mask the high order bit and display the low order bits at a output port.

Label	Mnemonics	Comments
	MVI C, A8h	C ← A8h
	MOV A, C	A ← C
	ANI 0Fh	Mask ($D_7 – D_4$)
	OUT Port 1	Display at Port 1
	HLT	

Que. 15 Write instructions to read the input port and continue to read it until both switches are closed. When both switches are closed, turn on LEDS.

Label	Mnemonics	Comments
	IN 2FH	Read Input Port
	ANI 00000011B	Mask all bits except D_1 & D_0.
	JNZ Start	If a switch is open, reads again
	OUT 8FH	Turn on all LEDS
	HLT	

Que. 16 When all of 8 switches are off, the microprocessor reads data FFh and when a switch is turned on, it goes to logic 0 (for all switches on, the data will be 00h). Write instruction to read input port and if all switches are open, set zero flag.

Label	Mnemonics	Comments
	IN 07h	Port Address
	CMA	Complement data from port 07h
	ORA A	Set zero flag, if all switches are open.
	HLT	

Programming with 8086

Example: The Most & Frequently Asked Problems & Their Solutions

Que. 1 Find the largest number from a table of 10 bytes.

```
TITLE: Obtaining Smallest Number in a Table Using Bubble Sorting Algorithm
.MODEL SMALL
.STACK 32
.DATA
TABLE DB 01, 21, 32, 26, 57, 64, 79, 50, 33, 42
LARGE DB?
COUNT DW 0010

.CODE
    MAIN PROC FAR
    MOV AX, @Data
    MOV DS, AX
    MOV SI, 00
    MOV CX, COUNT
    MOV DL, 0FFH
L1:
    MOV AL, TABLE [SI]
    INC SI
    CMP DL, AL
    JAE Next
    MOV DL, AL
Next:
    LOOP L1
    MOV LARGE, DL
    MOV AX, 4C00H
    INT 21 H

MAIN ENDP
END MAIN
```

Que. 2 Find the smallest number from a table of 10 bytes.

```
TITLE: Obtaining Smallest Number in a Table Using Bubble Sorting Algorithm
.MODEL SMALL
.STACK 32
.DATA
TABLE DB 01, 21, 32, 26, 57, 64, 79, 50, 33, 42
SMALL DB?
COUNT DW 0010

.CODE
    MAIN PROC FAR
    MOV AX, @Data
    MOV DS, AX
    MOV SI, 00
    MOV CX, COUNT
    MOV DL, 0FFH
L1:
    MOV AL, TABLE [SI]
    INC SI
    CMP DL, AL
    JB Next
    MOV DL, AL
Next:
    LOOP L1
    MOV SMALL, DL
    MOV AX, 4C00H
    INT 21 H

MAIN ENDP
    END MAIN
```

Que. 3 Suppose you have two tables of 10 unsigned integers (mad allowed being 255), find the sum of all ten numbers in each table and store the greatest of the two sums in memory.

TITLE: Add Corresponding Bytes of Two Tables of 10 Bytes And Store the Result in The Third Table.

```
.MODEL SMALL
.STACK 32
.DATA
TABLE1 DB 1,2,3,2,5,6,7,5,3,4
TABLE2 DB 3,4,5,2,6,7,8,6,4,2
SUM1 DB?
SUM2 DB?
LARGE DB?
COUNT DW 10

    .CODE
    MAIN PROC FAR
    MOV AX, @DATA
    MOV DA, AX
    MOV BX, 00
    MOV CX, COUNT
    MOV AH, 00
    MOV DL, 00
L1:
    MOV DL, TABLE1 [BX]
    ADD AH, DL
    INC BX
    LOOP L1
    MOV SUM1, AH
    MOV BX, 00
    MOV CX, COUNT
    MOV AH, 00
    MOV DL, 00
L2:
    MOV DL, TABLE2 [BX]
    ADD AH, DL
    INC BX
    LOOP L2
    MOV SUM2, AH
    MOV DL, SUM1
    MOV DH, SUM2
    MOV LARGE, DL
    CMP DL, DH
    JAE EXIT
    MOV LARGE, DH
EXIT:
    MOV AX, 4C00H
    INT 21H

MAIN ENDP
    END MAIN
```

Que. 4 Arrange the numbers of a table in ascending order.

TITLE: Arranging the Numbers of a Table in Ascending Order Using Bubble Sorting.

```
.MODEL SMALL
.STACK 32
.DATA
TABLE DB 34, 31, 32, 36, 33, 38, 39, 30, 37, 35
COUNT DW 0010

.CODE
    MAIN PROC FAR
    MOV AX, @Data
    MOV DS, AX
    MOV SI, 00              ; SI => int i = 0
    MOV DI, 00              ; DI => int j = 0
    ; for(i=o; i<10;i++) // Outer Loop
    ; for(j=i+1; j<10;j++) // Ineer Loop
    ; if(TABLE[j] < TABLE [i])
     swap (TABLE[j],TABLE[i]);
    ; // End of Inner Loop
    ; // End of Outer Loop
    ; Outer Loop Initialization
    MOV CX, COUNT
    DEC CX
    ; Outer Loop Begin
Outer Loop:
    ; Inner Loop Initialization
    MOV BX, SI
    INC BX
    MOV DI, BX
    ; Inner Loop Begin
Inner Loop:
    MOV DL, TABLE[DI]
    MOV DH, TABLE[SI]
    CMP DL, DH
    JAE Skip_swap
    CALL SWAP
    MOV TABLE[DI],DL
    MOV TABLE[SI},DH
Skip_swap
    INC DI
    CMP DI, 1
    JB Inner Loop
    ; Inner Loop Ends
    INC SI
    Loop Outer Loop
    ; Outer Loop Ends
    MOV AX, 4C00H
    INT 21H
MAIN ENDP
    SWAP PROC NEAR
      MOV AL, DL
      MOV DL, DH
      MOV DH, AL
      RET
    SWAP ENDP
END MAIN
```