CHAPTER – 1

INTRODUCTION

1.1 Introduction to Microprocessor

A *microprocessor* is a multipurpose, programmable, clock-driven, register-based electronic device that reads binary instructions from a storage device called *memory*, accepts binary data as input and processes data according to those instructions, and provides results as output.

A typical programmable machine can be represented with four components: microprocessor, memory, input, and output as shown in Figure 1.1. These four components work together or interact with each other to perform a given task; thus, they comprise a system. The physical components of this system are called *hardware*. A set of instructions written for the microprocessor to perform a task is called a *program*, and a group of programs is called *software*. The fact that the microprocessor is programmable means it can be instructed to perform given tasks within its capability.

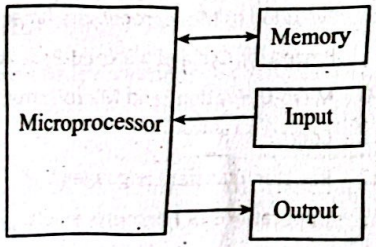


Figure 1.1 A programmable machine

1.2 Microprocessor as a CPU (MPU)

Traditionally, the computer is represented with four components: memory, input, output, and central processing unit (CPU), which consists of the arithmetic logic unit (ALU) and the control unit (CU). The CPU contains various registers to store data, ALU to perform arithmetic and logical operations, instruction decoders, counters, and control lines. The CPU reads instructions from the memory and performs the tasks specified. It communicates with input/output devices (I/O devices) either to accept or to send data; the I/O devices are also known as *peripherals*.

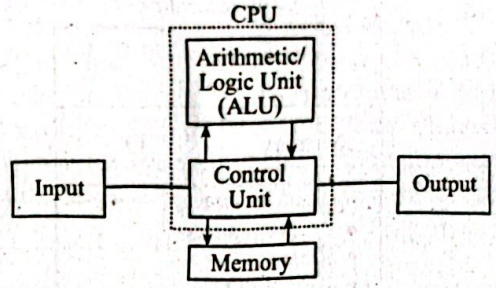


Figure 1.2 Traditional block diagram of a computer.

The advent of integrated circuit technology made possible to build the CPU on a single chip; this was coined *microprocessor*. A computer with a microprocessor as its CPU is known as a *microcomputer*. The terms *microprocessor* and *microprocessor unit* (MPU) are often used synonymously. MPU implies a complete processing unit with the necessary control signals. Because of the limited number of available pins on a microprocessor package, some of the signals (such as control and multiplexed signals) need to be generated by using discrete devices to make the microprocessor a complete functional unit or MPU.

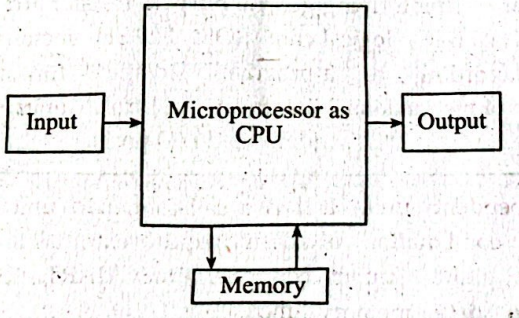


Figure 1.3 Block diagram of a computer with the microprocessor as CPU

With the advancement in technology, manufacturers were able to place memory and I/O interfacing circuits along with MPU on a single chip; this is known as a *microcontroller* or *microcontroller unit* (MCU). Figure 1.4 shows the block diagram of a microcontroller.

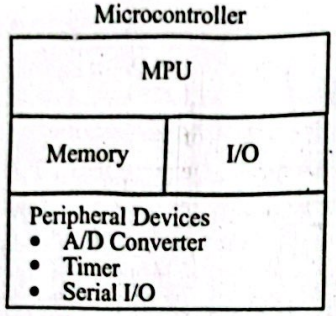


Figure 1.4 Block diagram of a microcontroller

Differences Between Microprocessor and Microcontroller

The differences between microprocessor and microcontroller are:

Microprocessor	Microcontroller
A microprocessor is a silicon chip representing a central processing unit (CPU), which is capable of performing arithmetic as well as logical operations according to a predefined set of instructions.	A microcontroller is an integrated chip that contains a CPU, scratchpad RAM, special and general purpose resister arrays, on chip ROM/FLASH memory for program storage, timer and interrupt control units, and dedicated I/O ports.
It is a dependent unit. It requires the combination of other chips like timers, program and data memory chips, interrupt controllers, etc. for functioning.	It is a self-contained unit and it doesn't require external interrupt controller, timer, UART, etc. for its functioning.

Microprocessor	Microcontroller
Microprocessors are most of the time general purpose in design and operation.	Microcontrollers are mostly application-oriented or domain-specific.
It does not contain a built in I/O port. The I/O port functionality needs to be implemented with the help of external programmable peripheral interface chips like 8255.	Most of the processors contain multiple built-in I/O ports which can be operated as a single 8- or 16- or 32-bit port or as individual port pins.

1.3 Organization of a Microprocessor-Based System

Microprocessor-based system includes three components: microprocessor, input/output, and memory (read-only memory and read/write memory). Figure 1.5 shows a simplified structure of a microprocessor-based system.

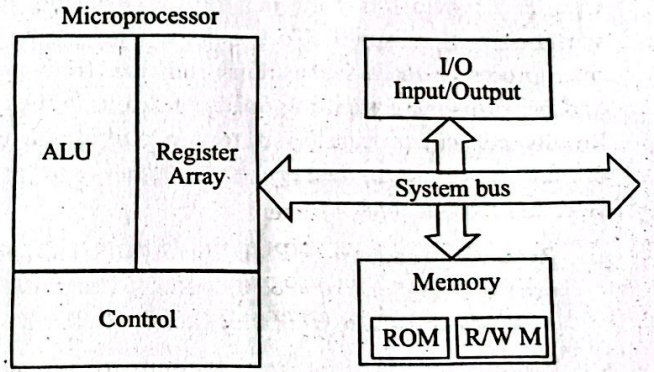


Figure 1.5 Microprocessor-based system with bus architecture

i. Microprocessor

It is a clock driven semiconductor device consisting of electronic logic circuits manufactured by using either a large-scale integration (LSI) or very-large-scale integration (VLSI) technique. It is capable of performing various

computing functions and making decisions to change the sequence of a program execution. It consists of three segments:

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

b. **Register array:** This part consists of various registers identified by letters such as B, C, D, E, H, and L. These registers are primarily used to store data temporarily during the execution of a program and are accessible to the user through instructions.

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

ii. **Memory**

Memory stores binary information such as instructions and data, and provides that information to the microprocessor whenever necessary. To execute programs, the microprocessor reads instructions and data from memory and performs the computing operations in its ALU section. Results are either transferred to the output section for display or stored in memory for later use. The memory block has two sections:

a. **Read-only memory (ROM):** The ROM is used to store programs that do not need alterations. Programs stored in ROM can only be read; they cannot be altered.

b. **Read/write memory or random-access memory (RAM):** It is also known as *user memory* which is used to store user programs and data. The information stored in this memory can be easily read and altered.

Input/ Output (I/O)

I/O includes two types of devices: input and output; these I/O devices are also known as *peripherals*. The input devices such as keyboard, switches, and an analog to digital (A/D) converter transfer binary information (data and instructions) from the outside world to the microprocessor. The output devices transfer data from the microprocessor to the outside world. They include the devices such as LED, CRT, digital to analog (D/A) converter, printer etc.

iv. **System Bus**

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

1.4 Bus Organization

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

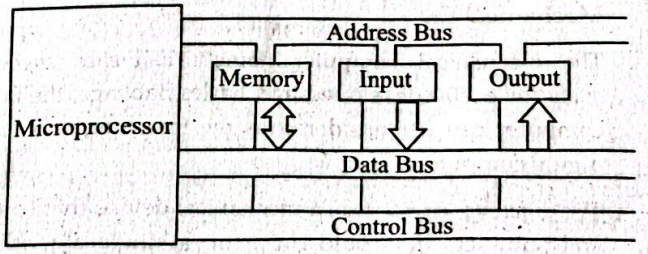


Figure 1.6 Bus organization

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

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

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

1.5 Historical Background of the Development of Computers

For understanding the historical background of the development of computers, it is a wise decision to classify computers under mechanical and electronic era.

1. Mechanical Era

The mechanical computer namely *difference engine* and *analytical engine* developed by Charles Babbage, the father of computer can be considered as the forerunners of modern digital computers.

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

2. Electronic Era

The First Generation: Vacuum Tubes

The ENIAC (Electronic Numerical Integrator And Calculator), designed and constructed at the University of Pennsylvania, was the world's first general-purpose electronic digital computer. The ENIAC was a decimal rather than a binary machine. That is, numbers were represented in decimal form, and arithmetic was performed in the decimal system. It's memory consisted of 20 accumulators, each capable of holding a 10-digit decimal number. A ring of 10 vacuum tubes represented each digit.

In 1947, Eckert and Mauchly formed the Eckert-Mauchly Computer Corporation. Their first successful machine was the UNIVAC I (Universal Automatic Computer). It was the first successful commercial computer. It was intended for both scientific and commercial applications. This computer was thought to perform matrix algebraic computations, statistical problems, premium billings for a life insurance company, and logistical problems. The UNIVAC II, which had greater memory capacity and higher performance than the UNIVAC I, was delivered in the late 1950s.

Later in 1953, IBM introduced its first electronic stored-program computer, the 701, and was intended primarily for scientific applications. In 1955, IBM delivered the companion 702 product that suited for business applications. These computers established IBM as the overwhelmingly dominant computer manufacturer.

The Second Generation: Transistors

The first major change in the electronic computer came with the replacement of the vacuum tube by the transistor. The

transistor is smaller, cheaper, and dissipates less heat than a vacuum tube. The use of the transistor defines the second generation of computers. The second generation saw the introduction of more complex arithmetic and logic units and control units, the use of high-level programming languages, and the provision of system software with the computer. In 1957, Digital Equipment Corporation (DEC) was founded, and in that year, delivered its first computer, the PDP-1. IBM developed 7090 in 1960, 7094 I in 1962, and 7094 II in 1964 (successive products had increased performance, capacity, and/or low cost).

The Third Generation: Integrated Circuits

Throughout the 1950s and early 1960s, electronic equipment was composed largely of discrete components – transistors, resistors, capacitors, and so on. Discrete components were manufactured separately, packaged in their own containers, and soldered or wired together onto masonite-like circuit boards, which were then installed in computers, oscilloscopes, and other electronic equipment. Whenever an electronic device called for a transistor, transistor had to be soldered to a circuit board. The entire manufacturing process, from transistor to circuit board, was expensive and cumbersome.

The use of the integrated circuit defines the third generation of computers. The entire circuit was fabricated in a tiny piece of silicon. Initially, only a few gates or memory cells could be reliably manufactured and packaged together. These early integrated circuits are referred to as *small-scale integration* (SSI) The most important members of the third generation are: IBM System/360 and the DEC PDP-8. These computers were small enough that it could be placed on top of a lab bench or be built into other equipment.

Later Generations

Beyond the third generation, there is less general agreement on defining generations of computers. Based on advances in integrated circuit technology, there have been a number of later generations. With the introduction of large-scale integration (LSI), more than 1000 components can be placed on a chip. Very-large-scale integration (VLSI) achieved more than 10,000 components per chip, while current ultra-large-scale integration (ULSI) chips can contain more than one billion components.

1.6 Stored-Program Concept and Von-Neumann Machine:

According to this concept, instructions are stored in computer memory to enable it to perform a variety of tasks in sequence or intermittently. The idea was introduced in the late 1940s by John Von-Neumann, who proposed that a program be electronically stored in binary-number format in a memory device so that instructions could be modified by the computer as determined by intermediate computational results. Other engineers, notably John W. Mauchly and J. Presper Eckert, contributed to this idea, which enabled digital computers to become much more flexible and powerful.

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 represented in a form suitable for storing in memory alongside the data. Then, the computer could get its instructions by reading them from memory, and a program could be set or altered by setting the values of a portion of memory. This approach is known as *stored-program concept*, and such architecture is named as *Von-Neumann architecture*.

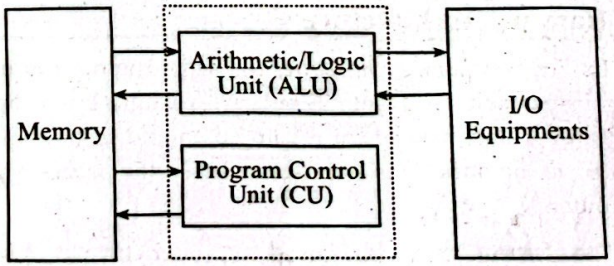


Figure 1.7 Von-Neumann architecture

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

The Von-Neumann architecture is the fundamental basis for the architecture of modern digital computers. The storage location of control unit and ALU are called *registers* and the various models of registers are:

- **MAR (memory address register)** – contains the address in memory of the word to be written into or read from the MBR.
- **MBR (memory buffer register)** – consists of a word to be stored in memory or sent to the I/O unit, or is used to receive a word from memory or from the I/O unit.
- **IR (instruction register)** – contains the 8-bit op-code instruction being executed.
- **IBR (instruction buffer register)** – used to temporarily hold the instruction from a word in memory.
- **PC (program counter)** – contains the address of the next instruction to be fetched from memory.
- **AC & MQ (accumulator and multiplier quotient)** – employed to hold temporarily operands and results of ALU operations. For example, the result of multiplying two 40-bit numbers is an 80-bit number; the most significant 40 bits are stored in the AC and the least significant in the MQ.

1.7 Harvard Architecture

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

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

In Figure 1.8, there are two data and two address buses multiplexed for data bus and address bus. Hence, there are two blocks of RAM chips: one for program memory and another for data memory addresses.

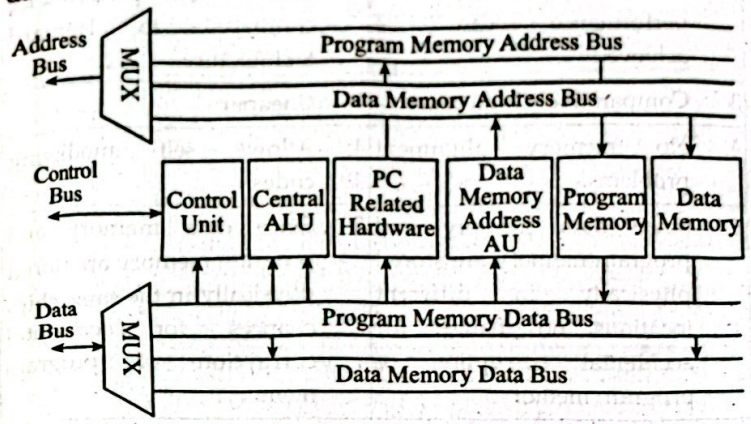


Figure 1.8 Harvard architecture

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

Differences Between Harvard and Von-Neumann Architecture

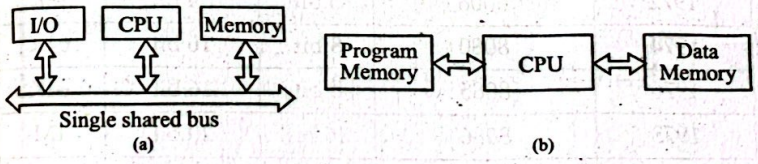


Figure 1.9 Harvard vs Von-Neumann architecture (a) Von-Neumann architecture (b) Harvard architecture

The differences between Harvard and Von-Neumann architecture are:

Harvard architecture	Von-Neumann architecture
1. Separate buses for instruction and data fetching	1. Single shared bus for instruction and data fetching
2. Easier to pipeline, so high performance can be achieved	2. Low performance compared to Harvard architecture
3. Comparatively high cost	3. Cheaper
4. No memory alignment problems.	4. Allows self modifying codes
5. Since data . memory and program memory are stored physically in different locations, no changes for accidental corruption of program memory.	5. Since data memory and program memory are stored physically in the same chip, chances for accidental corruption of program memory.

1.8 Evolution of Microprocessors (Intel Series)

The evolution of microprocessor is dependent on the development of integrated circuit technology from single scale integration (SSI) to giga scale integration (GSI).

Table 1.1 Evolution of microprocessors (Intel series)

Date	Microprocessor	Data Bus	Address Bus	Memory
1971	4004	4 bit	10 bit	640 Bytes
1972	8008	8 bit	14 bit	16K
1974	8080	8 bit	16 bit	64K
1976	8085	8 bit	16 bit	64K
1978	8086	16 bit	20 bit	1M
1979	8088	8 bit	20 bit	1M
1982	80286	16 bit	24 bit	16M
1985	80386	32 bit	32 bit	4G
1989	80486	32 bit	32 bit	4G
1993	Pentium	32/64 bit	32 bit	4G
1995	Pentium pro	32/64 bit	36 bit	64G
1997	Pentium II	64 bit	36 bit	64G
1998	Celeron	64 bit	36 bit	64G
1999	Pentium III	64 bit	36 bit	64G
2000	Pentium IV	64 bit	36 bit	64G
2001	Itanium	128 bit	64 bit	64G
2002	Itanium 2	128 bit	64 bit	64G
2003	Pentium M/Centrino (wireless capability) for Mobile version e.g. Laptop			
2006	Dual Core (32-bit or 64-bit processor)			
2006	Core 2 Series (64-bit processor)- Dual and Quad Core Processor			
2008	Atom (32-bit or 64-bit processor) – Single or Dual Core Processor			
2010	Core i3 (Dual Core Processor)			
2009	Core i5 (Dual and Quad Core Processor)			
2008	Core i7 (Dual and Quad Core Processor)			
2011	Core i7 Extreme Edition			
2017	Core i9 X-series			

Some of the highlights of the evolution of the Intel product line are listed below:

- **8080:** The world's first general-purpose microprocessor. This was an 8-bit machine, with an 8-bit data path to memory The 8080 was used in the first personal computer, the Altair.
- **8086:** A far more powerful, 16-bit machine which enable addressing a 1 MB of memory, and is the first appearance

the x86 architecture. A variant of this processor, the 8088, was used in IBM's first personal computer

- **80286:** This is an extension of 8086 which enabled addressing a 16 MB memory.
- **80386:** Intel's first 32-bit microprocessor that supported multitasking.
- **80486:** The 80486 introduced the use of much more sophisticated and powerful cache technology and sophisticated instruction pipelining, and offered a built-in math coprocessor.
- **Pentium:** This processor introduced the use of superscalar techniques, that is, ability to execute multiple instructions in parallel.
- **Pentium Pro:** Along with the features of Pentium, Pentium Pro enabled use of register renaming, branch prediction, data flow analysis, and speculative execution.
- **Pentium II:** The Pentium II incorporated Intel MMX technology, which is designed specifically to process video, audio, and graphics data efficiently.
- **Pentium III:** This version offered additional floating-point instructions to support 3D graphics software.
- **Core:** This is the first Intel x86 microprocessor with a dual core, that is, two processors on a single chip
- **Core 2:** The Core 2 is a 64-bit architecture. The Core 2 Quad provides four processors on a single chip.

1.9 Processing Cycle of a Stored-Program Computer

The basic function performed by a computer is execution of a program, which consists of a set of instructions stored in memory. Each instruction has two parts: one is the task to be performed, called the **operation code** (op-code) field, and the second is the data to be operated on, called the **operand** or address field. The processor does the actual work by executing instructions specified in the program. In its simplest from, *instruction processing*

consists of two steps: The processor reads (fetches) instructions from memory one at a time and executes each instruction. Program execution consists of repeating the process of instruction fetch and instruction execution. The instruction execution may involve several operations and depends on the nature of the instruction.

The processing required for a single instruction is called an *instruction cycle*. Program execution halts only if the machine is turned off, some sort of unrecoverable error occurs, or a program instruction that halts the computer is encountered.

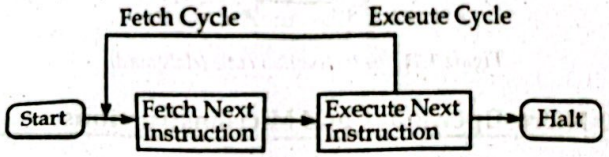


Figure 1.10 Basic instruction cycle

In fact, the processor has to do the following things:

- **Fetch instruction:** The processor reads an instruction from memory (register, cache, main memory).
- **Interpret instruction:** The instruction is decoded to determine what action is required.
- **Fetch data:** The execution of an instruction may require reading data from memory or an I/O module.
- **Process data:** The execution of an instruction may require performing some arithmetic or logical operation on data.
- ***Write* data:** The results of an execution may require writing *data* to memory or an I/O module.

To elaborate instruction cycle, we include two more stages. Thus, an instruction cycle consists of:

- **Fetch:** Read the next instruction from memory into the processor.
- **Execute:** Interpret the opcode and perform the indicated operation.
- **Interrupt:** If interrupts are enabled and an interrupt has occurred, save the current process state and service the interrupt.

- **Indirect:** If any indirect addressing is involved after an instruction is fetched, the required operands are fetched using indirect addressing.

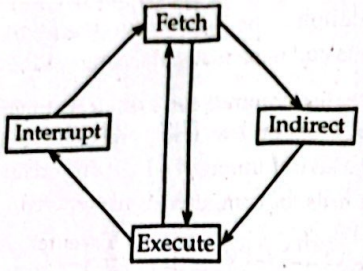


Figure 1.11 The instruction cycle (elaborated).

1.10 Micro-Operations and Microinstructions

Micro-Operations

A computer executes a program consisting sequence of instructions. Each instruction is made up of shorter sub-cycles (machine cycles): fetch, indirect, execute cycle (read, write), and interrupt. Performance of each cycle has a number of shorter operations called *micro-operations*. Micro-operations are functional atomic operations of CPU. Events of any instruction cycle can be described as a sequence of micro-operations. The prefix "micro" in micro-operations refers to the fact that each step is very simple and accomplishes very little.

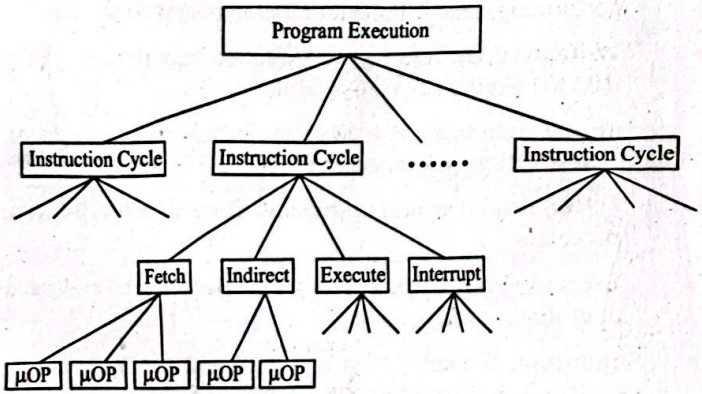


Figure 1.12 Constituent elements of a program execution

Microinstructions

Each instruction is characterized with many machine cycles and each cycle is characterized with many T-states; one complete cycle of clock is called as T-state. The lower instruction level patterns which are the numerous sequences for a single instruction are known as *microinstructions*. We can visualize the microinstruction with the help of any machine cycle. Here we consider the fetch cycle.

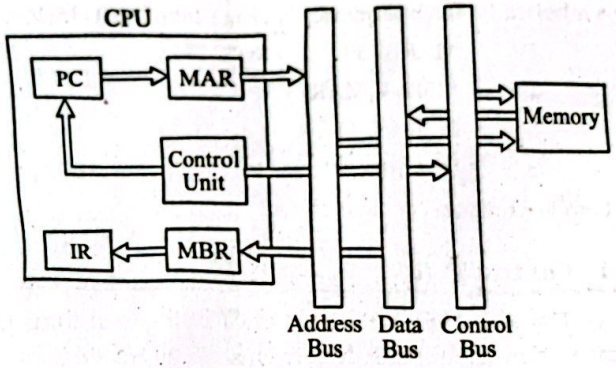


Figure 1.13 Data flow in fetch cycle.

Various registers involved in fetch cycle are:

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

The fetch sequence can be explained as follows:

- Address of next instruction is in PC. This address is
  to memory address register (MAR).

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

Symbolically, fetch sequence is completed in 3 clock cycles.

T_1: MAR ← PC

T_2: MBR ← [MAR]

T_3: PC ← PC +1

IR ←MBR

where T_i = time unit or clock cycle.

1.11 Control Unit

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

The functions of control unit include:

- **Sequencing**
  Causing the CPU to step through a series of micro-operations
- **Execution**
  Causing the performance of each micro-operations

Two types of control unit can be implemented in computing systems: *hardwired control unit* and *micro-programmed control unit.*

1. **Hardwired Control Unit**

This control unit is essentially a combinational circuit. In hardwired control unit, for each control signal, Boolean expression has to be derived for that signal as a function of the inputs. Hardwired control unit has faster mode of operation. A hardwired control unit needs rewiring if design has to be modified.

2. **Micro-Programmed Control Unit**

In a modern complex processor, the number of Boolean equations needed to define the control unit is very large. The task of implementing a hardwired control unit that satisfies all of these equations becomes extremely difficult. The result is to opt for a far simpler control unit, known as *micro-programmed control unit.*

In micro-programmed control unit, the logic of the control unit is specified by a microprogram. A microprogram consists of a sequence of instructions in a microprogramming language. These are very simple instructions that specify micro-operations. Modifications in micro-programmed control unit can be done by changing the microinstructions.

1.12 Register Transfer Language (RTL)

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

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

An Example of RTL

Consider the execution of instruction MOV A, B that consists of two machine cycles namely *fetch cycle* and *execution cycle.*

Fetch Cycle

Within the fetch cycle, the operations performed are:

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

In the first operation of fetch cycle, the contents of program counter will be transferred to the memory address register (MAR). The memory address register then uses the address bus to transmit its contents that specifies the address of memory location from where that instruction code of MOV A, B is to be fetched. Let T_1 indicates the period of first operation

T_1 : MAR ← PC

ii. When the control unit issues the memory read signal, the contents of the address memory location specified by MAR will be transferred to the memory buffer register (MBR). Suppose T_2 is the time period for this operation.

T_2 : MBR ← [MAR]

iii. Finally the contents of MBR will be transferred to the instruction register and then the program counter gets incremented. Let T_3 be the time required by the CPU to complete these operations.

T_3 : IR ← MBR

PC ← PC + 1

Execute Cycle

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

i. At the start of execution cycle, the instruction register (IR) consists of instruction code for instruction MOV A, B. The address field of instructions specifies the addresses of the two memory locations A & B. The first step needed is to obtain the data from the location B. For this, the address field of IR indicating the address of memory location will be transferred to address bus through the MAR. Let T_1 be this time taken.

T_1 : MAR ← (IR(Address of B))

ii. When the control unit issues a memory read signal, the contents of location B will be output (written) to the memory buffer register (MBR). Now the content of B which is to be written to memory location A is contained in MBR. Let T_2 be the time taken for that operation.

T_2 : MBR ← (B)

iii. Now, we need the memory location of A because it is being written with the data of location B. For this the address field of IR indicating the address of memory location A will be transferred to MAR in time T_3.

T_3 : MAR ← (IR(Address of A))

iv. When the control unit issues the memory write signal, the contents of MBR will be written to the memory location indicated by the contents of MAR in time T_4.

T_4 : A ← MBR or T_4 : [MAR] ← MBR

Note: [MAR] = A

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

Some RTL Examples

1. MOV Rd, Rs

RTL of MOV Rd, Rs	*Opcode Fetch Cycle*		
		T_1 :	MAR ← PC
		T_2 :	MBR ← [MAR]
		T_3 :	IR ← MBR, PC←PC+1
		T_4 :	Unspecified

2. MVI R, 8-bit Data

RTL of MVI R, 8-bit Data			
Opcode Fetch Cycle	T_1:	MAR	← PC
	T_2:	MBR	← [MAR]
	T_3:	IR	← MBR, PC←PC+1
	T_4:	Unspecified	
Memory Read Cycle	T_5:	MAR	← PC
	T_6:	MBR	← [MAR]
	T_7:	R	← MBR, PC ←PC + 1

3. MOV R, M

RTL of MOV R, M			
Opcode Fetch Cycle	T_1:	MAR	← PC
	T_2:	MBR	← [MAR]
	T_3:	IR	← MBR, PC←PC + 1,
	T_4:	Unspecified	
Memory Read Cycle	T_5:	MAR	←HL
	T_6:	MBR	← [MAR]
	T_7:	R	← MBR

4. MOV M, R

RTL of MOV M, R			
Opcode Fetch Cycle	T_1:	MAR	← PC
	T_2:	MBR	← [MAR]
	T_3:	IR	← MBR, PC←PC + 1,
	T_4:	Unspecified	
Memory Write cycle	T_5:	MAR	←HL
	T_6:	MBR	← R
	T_7:	[MAR]	← MBR

5. MVI M, 8-bit Data

RTL of MVI M, 8-bit Data			
Opcode Fetch Cycle	T_1:	MAR	← PC
	T_2:	MBR	← [MAR]
	T_3:	IR	← MBR, PC←PC+1
	T_4:	Unspecified	
Memory Read Cycle	T_5:	MAR	← PC
	T_6:	MBR	← [MAR]
	T_7:	Z	← MBR, PC ←PC + 1
Memory Write Cycle	T_8:	MAR	←HL
	T_9:	MBR	← Z
	T_{10}:	[MAR]	← MBR

6. LXI R_P, 16-bit Data

RTL of LXI D, 16-bit Data			
Opcode Fetch Cycle	T_1:	MAR	← PC
	T_2:	MBR	← [MAR]
	T_3:	IR	← MBR, PC←PC + 1
	T_4:	Unspecified	
Memory Read Cycle	T_5:	MAR	← PC
	T_6:	MBR	← [MAR]
	T_7:	RP_L	← MBR, PC ←PC + 1
Memory Read cycle	T_8:	MAR	← PC
	T_9:	MBR	← [MAR]
	T_{10}:	RP_H	← MBR, PC ←PC + 1

7. LDA 16-bit Address

RTL of LDA 16-bit Address			
Opcode Fetch Cycle	T_1:	MAR	← PC
	T_2:	MBR	← [MAR]
	T_3:	IR	← MBR, PC←PC + 1
	T_4:	Unspecified	

	Memory Read	T_5:	MAR ← PC
		T_6:	MBR ← [MAR]
		T_7:	Z ← MBR, PC ← PC + 1
	Memory Read Cycle	T_8:	MAR ← PC
		T_9:	MBR ← [MAR]
		T_{10}:	W ← MBR, PC ← PC + 1
	Memory Read cycle	T_{11}:	MAR ← WZ
		T_{12}:	MBR ← [MAR]
		T_{13}:	A ← MBR

8. **STA 16-bit Address**

RTL of STA 16-bit Address	Opcode Fetch Cycle	T_1:	MAR ← PC
		T_2:	MBR ← [MAR]
		T_3:	IR ← MBR, PC ← PC + 1
		T_4:	Unspecified
	Memory Read Cycle	T_5:	MAR ← PC
		T_6:	MBR ← [MAR]
		T_7:	Z ← MBR, PC ← PC + 1
	Memory Read Cycle	T_8:	MAR ← PC
		T_9:	MBR ← [MAR]
		T_{10}:	W ← MBR, PC ← PC + 1
	Memory Write Cycle	T_{11}:	MAR ← WZ
		T_{12}:	MBR ← A
		T_{13}:	[MAR] ← MBR

9. **LDAX R_P**

RTL of LDAX R_P	Opcode Fetch Cycle	T_1:	MAR ← PC
		T_2:	MBR ← [MAR]
		T_3:	IR ← MBR, PC ← PC + 1
		T_4:	Unspecified
	Memory Read Cycle	T_5:	MAR ← R_P
		T_6:	MBR ← [MAR]
		T_7:	A ← MBR

10. **STAX R_P**

RTL of STAX R_P	Opcode Fetch Cycle	T_1:	MAR ← PC
		T_2:	MBR ← [MAR]
		T_3:	IR ← MBR, PC ← PC + 1
		T_4:	Unspecified
	Memory Write Cycle	T_5:	MAR ← R_P
		T_6:	MBR ← A
		T_7:	[MAR] ← MBR

11. **LHLD 16-bit Address**

RTL of LHLD 16-bit Address	Opcode Fetch Cycle	T_1:	MAR ← PC
		T_2:	MBR ← [MAR]
		T_3:	IR ← MBR, PC ← PC + 1
		T_4:	Unspecified
	Memory read cycle	T_5:	MAR ← PC
		T_6:	MBR ← [MAR]
		T_7:	Z ← MBR, PC ← PC + 1

	Memory Read Cycle	T_8 :	MAR ← PC
		T_9 :	MBR ← [MAR]
		T_{10} :	W ← MBR, PC ← PC + 1
	Memory Read Cycle	T_{11} :	MAR ← WZ
		T_{12} :	MBR ← [MAR]
		T_{13} :	L ← MBR
	Memory Read Cycle	T_{14} :	MAR ← WZ + 1
		T_{15} :	MBR ← [MAR]
		T_{16} :	H ← MBR

12. SHLD 16-bit Address

RTL of SHLD 16-bit Address	*Opcode Fetch Cycle*	T_1 :	MAR ← PC
		T_2 :	MBR ← [MAR]
		T_3 :	IR ← MBR, PC ← PC + 1
		T_4 :	Unspecified
	Memory Read Cycle	T_5 :	MAR ← PC
		T_6 :	MBR ← [MAR]
		T_7 :	Z ← MBR, PC ← PC + 1
	Memory Read Cycle	T_8 :	MAR ← PC
		T_9 :	MBR ← [MAR]
		T_{10} :	W ← MBR, PC ← PC + 1
	Memory Write Cycle	T_{11} :	MAR ← WZ
		T_{12} :	MBR ← L
		T_{13} :	[MAR] ← MBR

	Memory Write Cycle	T_{14} :	MAR ← WZ + 1
		T_{15} :	MBR ← H
		T_{16} :	[MAR] ← MBR

13. IN 8-bit Address

RTL of IN 8-bit Address	*Opcode Fetch Cycle*	T_1 :	MAR ← PC
		T_2 :	MBR ← [MAR]
		T_3 :	IR ← MBR, PC ← PC + 1
		T_4 :	Unspecified
	Memory Read Cycle	T_5 :	MAR ← PC
		T_6 :	MBR ← [MAR]
		T_7 :	Z ← MBR, PC ← PC + 1
	I/O Write Cycle	T_8 :	IOAR ← Z
		T_9 :	IOBR ← A [IOAR]
		T_{10} :	A ← IOBR

14. OUT 8-bit Address

RTL of OUT 01H	*Opcode Fetch Cycle*	T_1 :	MAR ← PC
		T_2 :	MBR ← [MAR]
		T_3 :	IR ← MBR, PC ← PC + 1
		T_4 :	Unspecified
	Memory Read Cycle	T_1 :	MAR ← PC
		T_2 :	MBR ← [MAR]
		T_3 :	Z ← MBR, PC ← PC + 1
	I/O Write Cycle	T_1 :	IOAR ← Z
		T_2 :	IOBR ← A
		T_3 :	[IOAR] ← IOBR

5. XCHG

RTL of XCHG	Opcode Fetch Cycle		
		T_1:	MAR ← PC
		T_2:	MBR ← [MAR]
		T_3:	IR ← MBR, PC←PC+1
		T_4:	Unspecified

6. ADD R

RTL of ADD R	Opcode Fetch Cycle		
		T_1:	MAR ← PC
		T_2:	MBR ← [MAR]
		T_3:	IR ← MBR, PC←PC+1
		T_4:	Unspecified

7. ADI 8-bit Data

RTL of ADI 8-bit Data	Opcode Fetch Cycle		
		T_1:	MAR ← PC
		T_2:	MBR ← [MAR]
		T_3:	IR ← MBR, PC←PC+1
		T_4:	Unspecified
	Memory Read Cycle	T_5:	MAR ← PC
		T_6:	MBR ← [MAR]
		T_7:	A ← MBR + A, PC ←PC + 1

8. ADC R

RTL of ADC R	Opcode Fetch Cycle		
		T_1:	MAR ← PC
		T_2:	MBR ← [MAR]
		T_3:	IR ← MBR, PC←PC+1
		T_4:	Unspecified

ACI 8-bit Data

RTL of ACI 8-bit Data	Opcode Fetch Cycle		
		T_1:	MAR ← PC
		T_2:	MBR ← [MAR]
		T_3:	IR ← MBR, PC←PC+1
		T_4:	Unspecified
	Memory Read Cycle	T_5:	MAR ← PC
		T_6:	MBR ← [MAR]
		T_7:	A ← MBR + A + CY, PC ←PC + 1

20. ADD M

RTL of ADD M	Opcode Fetch Cycle		
		T_1:	MAR ← PC
		T_2:	MBR ← [MAR]
		T_3:	IR ← MBR, PC←PC + 1,
		T_4:	Unspecified
	Memory Read Cycle	T_5:	MAR ←HL
		T_6:	MBR ← [MAR]
		T_7:	A ← MBR + A

21. ADC M

RTL of ADC M	Opcode Fetch Cycle		
		T_1:	MAR ← PC
		T_2:	MBR ← [MAR]
		T_3:	IR ← MBR, PC←PC + 1,
		T_4:	Unspecified
	Memory Read Cycle	T_5:	MAR ←HL
		T_6:	MBR ← [MAR]
		T_7:	A ← MBR + A + CY

22. INR M

RTL of INR M	Opcode Fetch Cycle		
		T_1:	MAR ← PC
		T_2:	MBR ← [MAR]
		T_3:	IR ← MBR, PC←PC + 1,
		T_4:	Unspecified
	Memory Read Cycle	T_5:	MAR ←HL
		T_6:	MBR ← [MAR]
		T_7:	Z ← MBR

	Memory Write Cycle	T_8:	MAR ← HL
		T_9:	MBR ← Z + 1
		T_{10}:	[MAR] ← MBR

1.13 Applications of Microprocessors

Microprocessors are applicable to a wide range of information processing tasks, ranging from general computing to real-time monitoring systems. Most electronic systems – including everything from personal computers, laptops, remote controls, washing machines, microwave ovens to mobile phones, complex military and space systems, and industrial automation contain a built-in microprocessor in it. These applications are listed below:

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

Chapter 2

Programming with 8085 Microprocessor

CHAPTER - 2

PROGRAMMING WITH 8085 MICROPROCESSOR

2.1 Internal Architecture of 8085 Microprocessor

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

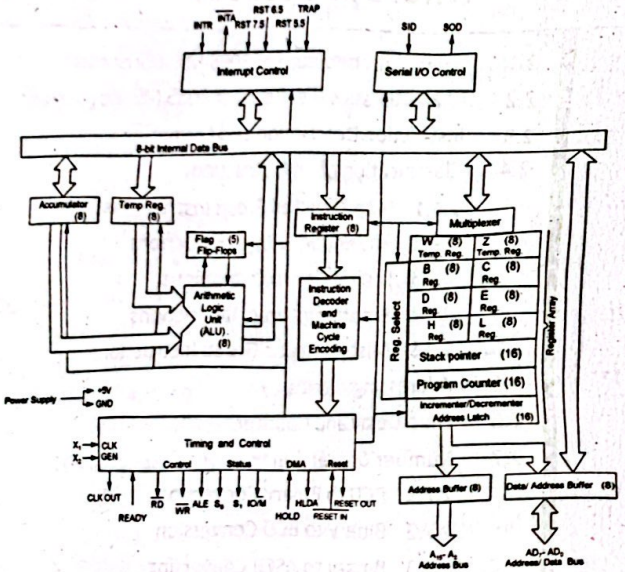


Figure 2.1: The 8085A microprocessor functional block diagram

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

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

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

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

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

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

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

7. **Stack Pointer (SP):** It is a 16-bit register used as a memory pointer. It points to a memory location in R/W memory, called the stack. The beginning of the stack is defined by loading a 16-bit address in the stack pointer.

8. **Program Counter (PC):** Microprocessor uses the PC register to sequence the execution of the instructions. The function of PC is to point to the memory address from which the next byte is to be fetched. When a byte is being fetched, the PC is incremented by one to point to the next memory location.

9. **Flags:**

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

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

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

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

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

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

Programmer's Model of an 8085 Microprocessor

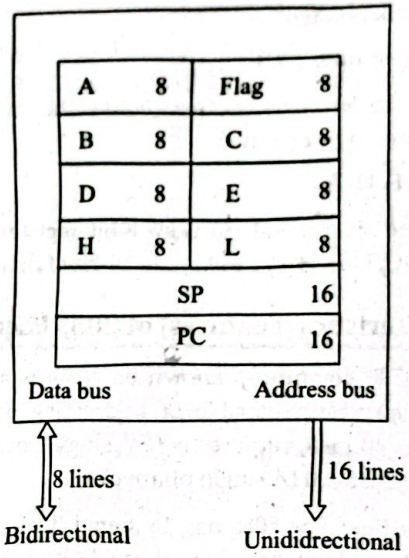


Figure 2.2 Programmer's model of an 8085 microprocessor

The programmer's model of an 8085 microprocessor consists of:

Accumulator

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

Flags

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

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

Stack pointer (SP)

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

Program counter (PC)

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

B, C, D, E, H, L

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

2.2 Characteristics (Features) of 8085 Microprocessor

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

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

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

3. **Control and Status Signals:** This group of signals includes two control signals ($\overline{RD}$ and $\overline{WR}$), three status signals (IO/ $\overline{M}$, S_1 and S_0) to identify the nature of the operation, and one special signal (ALE) to indicate the beginning of the operation.

- **ALE (Address Latch Enable):** This is a positive going pulse generated every time 8085 begins an operation (machine cycle). It indicates that the bits AD_7-AD_0 are address bits. This signal is used primarily to latch the low-order address from the multiplexed bus and generate a separate set of eight address lines A_7–A_0.

- $\overline{RD}$ **(Read):** This is a read control signal (active low). This signal indicates that the selected I/O or memory device is to be read and data are available on the data bus.

- $\overline{WR}$ **(Write):** This is a write control signal (active low). This signal indicates that the data on the data bus are to be written into a selected memory or I/O location.

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

- S_1 **and** S_0 : These status signals, similar to IO/ $\overline{M}$, can identify various operations, but they are rarely used in small systems.

4. **Power Supply and Clock Frequency:**
   - V_{CC}: +5V power supply
   - V_{SS}: Ground reference
   - X_1 **and** X_2: A crystal (RC or LC network) is connected at these two pins for frequency.
   - **CLK OUT:** It can be used as the system clock for other devices.

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

Stack pointer (SP)

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

Program counter (PC)

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

B, C, D, E, H, L

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

2.2 Characteristics (Features) of 8085 Microprocessor

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

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

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

   Control and Status Signals: This group of signals includes two control signals ($\overline{RD}$ and $\overline{WR}$), three status signals (IO/$\overline{M}$, S_1 and S_0) to identify the nature of the operation, and one special signal (ALE) to indicate the beginning of the operation.

- **ALE (Address Latch Enable):** This is a positive going pulse generated every time 8085 begins an operation (machine cycle). It indicates that the bits AD_7-AD_0 are address bits. This signal is used primarily to latch the low-order address from the multiplexed bus and generate a separate set of eight address lines A_7-A_0.
- $\overline{RD}$ **(Read):** This is a read control signal (active low). This signal indicates that the selected I/O or memory device is to be read and data are available on the data bus.
- $\overline{WR}$ **(Write):** This is a write control signal (active low). This signal indicates that the data on the data bus are to be written into a selected memory or I/O location.
- **IO/$\overline{M}$:** This is a status signal used to differentiate between I/O and memory operations. When it is high, it indicates an I/O operation; when it is low, indicates a memory operation. This signal is combined with $\overline{RD}$ (Read) and $\overline{WR}$ (Write) to generate I/O and memory signals.
- **S_1 and S_0 :** These status signals, similar to IO/$\overline{M}$, can identify various operations, but they are rarely used in small systems.

4. **Power Supply and Clock Frequency:**
   - **V_{CC}:** +5V power supply
   - **V_{SS}:** Ground reference
   - **X_1 and X_2:** A crystal (RC or LC network) is connected at these two pins for frequency.
   - **CLK OUT:** It can be used as the system clock for other devices.

Externally Initiated Signals:

- **INTR (Input):** Interrupt request, used as a general purpose interrupt.
- **$\overline{\text{INTA}}$ (Output):** This is used to acknowledge an interrupt.
- **RST 7.5, 6.5, 5.5 (Inputs):** These are vectored interrupts that transfer the program control to specific memory locations. They have higher priorities than INTR interrupt. Among these three, the priority order is 7.5, 6.5, and 5.5.
- **TRAP (Input):** This is a non-maskable interrupt with highest priority.
- **HOLD (Input):** This signal indicates that a peripheral such as a DMA (Direct Memory Access) controller is requesting use of address and data bus.
- **HLDA (Output):** HLDA stands for Hold Acknowledge. This signal acknowledges the HOLD request
- **READY (Input) :** This signal is used to delay the microprocessor Read or Write cycles until a slow-responding peripheral is ready to send or accept data. When this signal goes low, the microprocessor waits for an integral number of clock cycles until it goes high.
- **$\overline{\text{RESET IN}}$:** When the signal on this pin goes low, the program counter is set to zero, the buses are tri-stated, and MPU is reset.
- **RESET OUT:** This signal indicates that the MPU is being reset. The signal can be used to reset other devices.
- **Serial I/O Ports:** The 8085 has two signals to implement the serial transmission: SID (Serial Input Data), and SOD (Serial Output Data). In serial transmission, data bits are sent over a single line, one bit at a time, such as the transmission over telephone lines.

2.3 Instruction Description and Format

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

Instruction Cycle

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

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

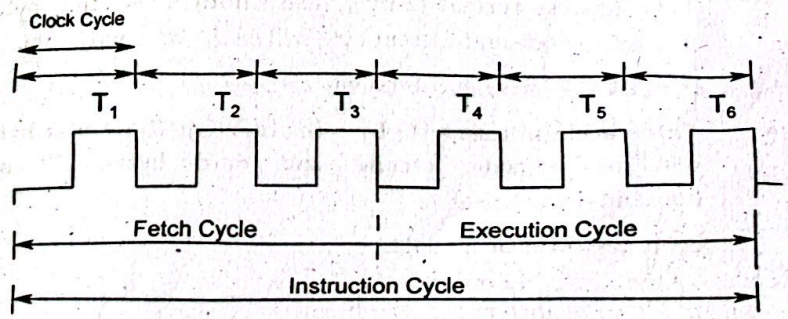


Figure 2.3 Instruction cycle

Instruction Format

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

ADD	R_1, R_0
op-code	*address*

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

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

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

E.g., ADD B; MOV A,B

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

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

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

2nd byte- lower order data.

3rd byte – higher order data

E.g., LXI B, 4050 H

2.4 Classification of an Instruction

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

- **Data Transfer Group:** The instructions which are used to transfer data from one register to another register or register to memory fall in this category.
- **Arithmetic Group:** The instructions which perform arithmetic operations such as addition, subtraction, increment, decrement, etc. are categorized in this group.
- **Logical Group:** The instructions which perform logical operations such as AND, OR, XOR, etc. comes under this group.
- **Branching Group:** The instructions which are used for looping and branching such as jump, call, etc. are categorized in this group.
- **Miscellaneous Group:** The instructions relating to stack operation, controlling purposes such as interrupt operations including machine control instructions like HLT, NOP, etc. fall under miscellaneous group.

2.4.1 Data Transfer Group Instructions

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

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

   E.g., MOV A, B; A ← B

2. **MVI R, 8-bit Data (Move Immediate Data to Register)**

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

E.g., MVI C, 53H ; C ← 53H

3. **MOV M, R (Move to Memory from Register)**

- 1-byte instruction
- Copies the contents of the specified register to memory. Here, memory is the location specified by the content of HL register pair.

E.g., MOV M, B

4. **MOV R, M (Move to Register from Memory)**

- 1-byte instruction
- Copies the contents of memory location as specified by HL register pair to a register.

E.g., MOV B, M

5. **LXI R_P, 16-bit Data (Load Register Pair with Immediate Data)**

- 3-byte instruction
- Load immediate 16-bit data to register pair
- Register pair may be BC, DE, HL, and SP
- First it loads lower 8-bit data into lower order register and then loads higher 8-bit data into higher order register.

E.g., LXI B, 4532H; B ← 45, C ← 32H

MVI M, Data (Load Memory with Immediate Data)

- 2-byte instruction
- Loads the 8-bit data to the memory location whose address is specified by the contents of HL pair.

E.g., MVI M , 35H; [HL] ← 35H

7. **LDA 16-bit Address (Load Accumulator Direct)**

- 3-byte instruction
- Loads the accumulator with the contents of memory location whose address is specified by 16 bit address.

E.g., LDA 4035H; A ← [4035H]

8. **LDAX R_P (Load Accumulator Indirect)**

- 1-byte instruction
- Loads the contents of memory location pointed by the contents of register pair to accumulator.

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

9. **STA 16-bit Address (Store Accumulator Content Direct)**

- 3-byte instruction
- Stores the contents of accumulator to specified address

E.g., STA FA00H; [FA00] ← [A]

10. **STAX R_P (Store Accumulator Content Indirect)**

- 1-byte instruction
- Stores the contents of accumulator to memory location specified by the contents of register pair.

E.g., STAX B ; [BC] ← A

11. **IN 8-bit Address (Input Data from Input Port)**

- 2-byte instruction
- Reads data from the input port address specified in the second byte and loads data into the accumulator i.e., input port to accumulator.

E.g., IN 40H ; A ← [40H]

12. **OUT 8-bit Address (Output Data to Output Port)**

- 2-byte instruction
- Copies the contents of the accumulator to the output port address specified in the 2nd byte i.e., accumulator to output port.

E.g., OUT 40H ; [40] ← A

13. **LHLD 16-bit Address (Load HL Pair Direct)**

- 3-byte instruction
- Loads the contents of specified memory location to L register and contents of next higher location to H register.

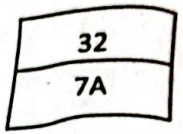
E.g., LXI H, 9500H

MVI M, 32H; 9500

MVI L, 01H; 9501

MVI M, 7AH

LHLD 9500H; H=7A, L=32



14. ***SHLD 16-bit* Address (Store HL Pair Direct)**

- 1-byte instruction
- It is opposite to LHLD instruction
- Stores the contents of L-register to specified memory location and contents of H-register to next higher memory location.

E.g., LXI H, 9500H

SHLD 8500H; [8500] = 00, [8501] = 95

15. **XCHG (Exchange)**

- 1-byte instruction
- Exchanges DE pair with HL pair.

E.g., LXI H, 7500H; H= 75, L=00

LXI D, 9532H; D=95. E=32

XCHG ; H=95, L=32 ; D=75, E=00

Examples

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

Using MOV

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

Using LDAX

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

Using LDA

```
LDA 2050H
HLT
```

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

Using MOV

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

Using STAX

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

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

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

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

```
MVI A, 20H
STA 2050H
```

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

5. **Register B contains 45H and register D contains C2H. WAP to swap the contents of register B and D.**

```
MVI B, 45H
MVI D, C2H
MOV A, B
MOV B, D
MOV D, A
HLT
```

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

```
LXI B, 1122H    ; B=11, C=22
LXI D, 3344H    ; D=33, E=44
MOV H, B
MOV L, C
XCHG            ; Exchange DE pair with HL pair
MOV B, H
MOV C, L
HLT
```

hmetic Group Instructions

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

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

     E.g., Add B; A ← [A] + [B]
   - All flags are affected

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

     E.g., ADI 9BH ; A← A+9BH
   - All flags are affected

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

     E.g., SUB D ; A ← A – D
   - All flags are affected

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

     E.g., SUI D3H; A ← A – D3H
   - All flags are affected

5. **INR R/M (Increment)**

   DCR R/M (Decrement)
   - 1-byte instructions
   - INR R/M and DCR R/M increases and decreases the contents of R (register) or M (memory) by 1 respectively.

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

- All flags are affected except carry.

6. INX Rp (Increment Register Pair)

DCX Rp (Decrement Register Pair)

- 1-byte instructions
- Increases and decreases the content of register pair by
- Acts as 16-bit counter in looping

E.g., INX B ; BC=BC+1

DCX D ; DE=DE+1

- No flags are affected

7. ADC R/M (Addition with Carry)

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

E.g., ADC B ; A ←A + B + CY

- All flags are affected

8. ACI 8-bit Data (Addition with Carry Immediate)

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

E.g., ACI 70H ; A ←A + 70 + CY

- All flags are affected

SBB R/M (Subtraction with Borrow)

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

E.g., SBB D ; A ←A – D – Borrow

- All flags are affected

10. SBI 8-bit Data (Subtraction with Borrow Immediate)

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

E.g., SBI 70H ; A ←A – 70 – Borrow

- All flags are affected

9. DAD R_p (Double Addition)

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

E.g., LXI H, 7320H

LXI B, 4220H

DAD B; HL ← HL + BC (7320 + 4220 = B540)

- Only carry flag is affected if the result is greater than 16 bit.

10. DAA (Decimal Adjustment Accumulator)

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

E.g., MVI A, 78H ; A=78

MVI B, 42H ; B=42

ADD B ; A=A+B = BA

DAA ; A=20, CY=1

- All flags are affected

Examples

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

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

Register BC contain 2793H and register DE contain 3182H. Write instruction to add these two 16 bit numbers and place the sum in memory locations 2050H and 2051H.

```
MOV A, C
ADD E
MOV L, A
MOV A, B
DC D
OV H, A
LD 2050H
T
```

ister BC contains 8538H and register DE contain 5H. Write instructions to subtract the contents of DE the contents of BC and place the result in BC.

```
A, C
```



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

Features of Arithmetic Instructions

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

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

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

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

Addition Operation in 8085

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

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

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

Subtraction Operation in 8085

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

its 1's complement.

2. Adds 1 to 1's complement to obtain 2's complement of subtrahend.
3. Adds 2's complement to the minuend (the contents of accumulator).
4. Complements the carry flag.

```
e.g MVI A, 97 H:        0110 0101
    MVI B, 65 H:        1001 1010
    SUB B      2's comp.:1001 1011
                97 :+1001 0111
                  [1] 0011 0010

Complement carry = 0
Accumulator = 32 H
```

BCD Addition

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

The microprocessor cannot recognize BCD numbers; it adds any two numbers in binary. In BCD addition, any number larger than 9 (from A to F) is invalid and needs to be adjusted by adding 6 in binary.

E.g.,

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

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

E.g., Add BCD 77 and 48


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

10010 0101→ 125_{BCD}

2.4.3 Logical Group Instructions

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

The following features hold true for all logic instructions:

1. The instructions implicitly assume that the accumulator is one of the operands.
2. All instructions reset (clear) carry flag except for complement where flag remain unchanged.
3. They modify Z, P, and S flags according to the data conditions of the result.
4. Place the result in the accumulator.
5. They do not affect the contents of the operand register.

The logical operations have the following instructions.

1. ANA R/M (Logical AND)

- 1-byte instruction.
- Logically AND the contents of register/memory with the contents of accumulator and stores result into accumulator.
- CY flag is reset, AC is set and others as per result.

E.g., ANA C; A ← A && C

2. **ANI 8-bit Data (Logical AND with Immediate Data)**

- 2-byte instruction.
- Logically AND 8-bit immediate data with the cont of accumulator and stores result into accumulator.
- CY flag is reset, AC is set and others as per result.

E.g., ANI 85H; A ← A && 85H

3. ***ORA R/M (Logical OR)***

- 1-byte instruction.
- Logically OR the contents of register/memory with t contents of accumulator and stores result in accumulator.
- CY and AC are reset and others as per result.

E.g., ORA C; A ← A || C

4. **ORI 8-bit Data (Logical OR with Immediate Data)**

- 2-byte instruction.
- Logically OR 8-bit immediate data with the contents o the accumulator and stores result into accumulator.
- CY and AC are reset and others as per result.

E.g., ORI 54H; A ← A || 54H

5. ***XRA R/M (Logical XOR)***

- 1-byte instruction
- Logically exclusive OR the contents of register memory with the contents of accumulator and stores result into accumulator.
- CY and AC are reset and others as per result.

E.g., XRA M; A ← A ⊕ [HL]

XRI 8-bit Data (Logical XOR with Immediate Data)

- 2-byte instruction
- Logically exclusive OR 8-bit data immediate with the content of accumulator and stores result into accumulator.

- CY and AC are reset and others as per result.

E.g., XRI 28H; A ← A ⊕ 28H

7. **CMA (Complement Accumulator)**

- 1-byte instruction
- Complements the contents of the accumulator.
- No flags are affected.

E.g., CMA ; A ← A'

8. **Logically Compare Instructions**

CMP R/M (Compare with Accumulator)

- 1-byte instruction
- All flags are modified

CPI 8-bit Data (Compare Immediate with Accumulator)

- 2-byte instruction
- All flags are modified.

These instructions compare the content of register/memory or 8-bit data with the content of accumulator by subtracting the data from accumulator. However, the content of operands are not modified. It is used to compare the data which can be used to indicate end of data. The status of comparison is shown by flags as illustrated below.

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

E.g., CMP C; compares register C with accumulator

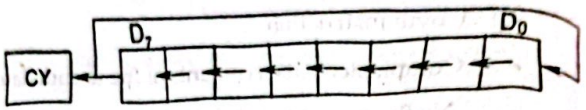
CPI 76H; compares 76H with accumulator

9. **Logically Rotate Instructions**

RLC (Rotate Accumulator Left)

- 1-byte instruction

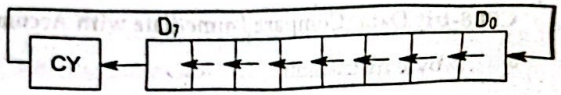
- Each bit is shifted to the adjacent left position. Bit D_7 becomes D_0.
- The carry flag is modified according to D_7.



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

RAL (Rotate Accumulator Left through Carry)

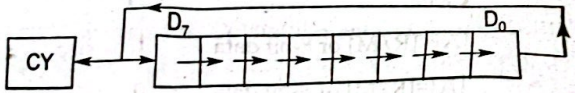
- 1-byte instruction
- Each bit is shifted to the adjacent left position. Bit D_7 becomes the carry bit and the carry bit is shifted into D_0.
- The carry flag is modified according to D_7.



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

RRC (Rotate Accumulator Right)

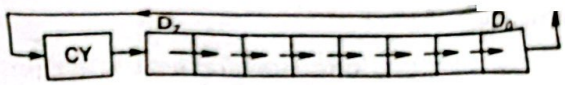
- 1-byte instruction
- Each bit is shifted right to the adjacent position. Bit D_0 becomes D_7.
- The carry flag is modified according to D_0.



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

RAR (Rotate Accumulator Right through Carry)

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



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

The rotate instructions are used in arithmetic multiply and divide operations and for serial data transfer.

For example, if the accumulator has data 0000 1000 = 8

- By rotating 08H left, accumulator will have 0001 0000 = 16 which is equivalent to multiplying by 2.
- By rotating 08H right, accumulator will have 0000 0100 = 4 which is equivalent to dividing by 2.

However, these are invalid when logic 1 is rotated left from D_7 to D_0 and rotated right from D_0 to D_7.

10. CMC (Complement Carry)

- 1-byte instruction
- It complements the carry flag.
- No other flags are affected.

11. STC (Set Carry Flag)

- 1-byte instruction
- It sets the carry flag to 1
- No other flags are affected

12. CMA (Complement Accumulator)

- 1-byte instruction
- It complements the content of accumulator.
- No flags are affected

Data Masking (Setting and Resetting Specific Bits)

In various situations, we need to set or reset a specific bit without affecting the other bits. This process is referred to as *data masking*.

Setting bits

Logical OR instructions are used to set particular b without affecting other bits. This is done by ORing the particul bits with logic 1 and other bits with logic 0. This is known as O *masking*.

For example, if the accumulator has data 1100 1000 = C8H

- By ORing C8H with 03H (0000 0011) using instructio ORI 03H, accumulator will have 1100 1011 = CBH, tha is, D_0 and D_1 of accumulator are both set to 1, and othe bits are unchanged.

Resetting bits

Logical AND instructions are used to reset particular bit without affecting other bits. This is done by ANDing the particula bits with logic 0 and other bits with logic 1. This is known as AND *masking*.

For example, if the accumulator has data 1100 1000 = C8H

- By ANDing C8H with 3FH (0011 1111) using instruction ANI 3FH, accumulator will have 0000 1000 = 08H, that is, D_7 and D_6 of accumulator are both reset to 0, and other bits are unchanged.

Complementing bits

Logical XOR instructions are used to complement particular bits without affecting other bits. This is done by XORing the particular bits with logic 1 and other bits with logic 0. This is known as *XOR masking*.

For example, if the accumulator has data 1100 1000 = C8H

- By XORing C8H with 0CH (0000 1100) using instruction XRI 0CH, accumulator will have 1100 0100 = C4H, that is, D_3 is complemented to 0 and D_2 is complemented to 1, and other bits are unchanged.

Examples

1. Write a program to AND the content of memory location C050H and the content of register C, and store the result into location C051H.

```
LDA C050H
ANA C
STA C051H
HLT
```

2. Write a program to complement the content of register D using instructions XRI and CMA.

Using XRI

```
MOV A, D
XRI FFH
HLT
```

Using CMA

```
MOV A, D
CMA
HLT
```

3. Write a program to change the lower and upper nibble of the data stored in location C020H.

```
LDA C020H
RLC
RLC
RLC
RLC
STA C020H
HLT
```

4. Register C contains 95H. Write instructions to unpack this data to 09H and 05H, and store into memory locations 9050H and 9051H.

```
MVI C,95H
MOV A,C      ; A ← 95H
ANI F0H      ; A ← 90H
```

```
RLC
RLC
RLC
RLC              ; A ← 09H
STA 9050H
MOV A,C          ; A ← 95H
ANI 0FH          ; A ← 05H
STA 9051H
HLT
```

5. **Register B contains A6H. Write instructions to set D_6 and to reset D_2.**

Using Masking

```
MVI B, A6H
MOV A, B
ORI 40H          ; Sets D6 [OR 0100 0000 : A ← 1110 0110]
ANI FBH          ; Resets D2 [AND 1111 1011 : A ← 1110 0010]
MOV B,A
HLT
```

Using Rotation

```
MVI B, A6H
MOV A, B         ; CY ← x, A ← 1010 0110
RAL              ; CY ← 1, A ← 0100 110x
RAL              ; CY ← 0, A ← 1001 10x1
STC              ; CY ← 1
RAR              ; CY ← 1, A ← 1100 110x
RAR              ; CY ← x, A ← 1110 0110 : Sets D6
RAR              ; CY ← 0, A ← x111 0011
RAR              ; CY ← 1, A ← 0x11 1001
RAR              ; CY ← 1, A ← 10x1 1100
STC              ; CY ← 1
CMC              ; CY ← 0
RAL              ; CY ← 1, A ← 0x11 1000
RAL              ; CY ← 0, A ← x111 0001
RAL              ; CY ← x, A ← 1110 0010 : Resets D1
HLT
```


2.4.4 Branching Group Instructions

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

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

- Jump Instructions
- Call and Return Instruction
- Restart Instruction

1. Jump Instructions

The jump instructions specify the memory location explicitly. They are 3-byte instructions, one byte for the operation code followed by a 2-byte (16-bits) memory address. Jump instructions can be used to create loops and are classified into unconditional and conditional jump.

a. Unconditional Jump

8085 includes unconditional jump instructions to enable the programmer to set up continuous loop without depending on any type of conditions.

JMP 16-bit Address

- 3-byte instruction
- It loads the program counter by 16-bit address and program execution transfers to that memory location.

  E.g., JMP 4000H

The jump location can also be specified using a label (name). However, we should not specify both a label and its 16-bit address in a jump instruction. Furthermore, we cannot use the same label for different memory locations.

Example

Using 16-bit Address

Address	Mnemonics
C000H	MVI A, 00H
C002H	OUT 40H
C004H	INR A
C005H	JMP C002H
C008H	HLT

Using Label

```
        MVI A, 00H
L1:     OUT 40H
        INR A
        JMP L1
        HLT
```

Using Name

```
        MVI A, 00H
NEXT:   OUT 40H
        INR A
        JMP NEXT
        HLT
```

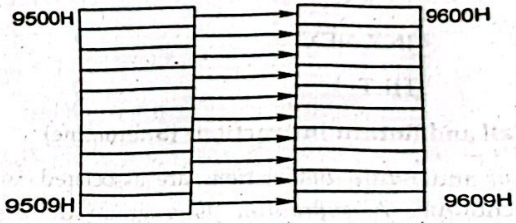
b. Conditional Jump

The *conditional jump instructions* allow the microprocessor to make decisions based on certain conditions indicated by the flags. After logic and arithmetic operations, flags are set or reset to reflect the conditions of data. These instructions check the conditions and make decisions to change or not to change the sequence of program. The four flags namely carry, zero, sign, and parity are used by the *conditional jump instructions.*

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

Examples

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




```
        MVI B, 0AH
        LXI H, 9500H
        LXI D, 9600H
NEXT:   MOV A, M
        STAX D
        INX H
        INX D
        DCR B
        JNZ NEXT
        HLT
```

```
        LXI H, 8500H    ; Source
        LXI D, 9500H    ; Destination
NEXT:   MOV A, M
        RRC
        JC L1           ; If data is odd, then go to L1.
        MVI A, 00H
        JMP L3
L1:     RLC
L3:     STAX D
        INX D
        INX H
        DCR C
        JNZ NEXT
        HLT
```

Call and Return Instructions (Subroutine)

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

Subroutine instructions are classified into conditional and unconditional.

a. **Unconditional Subroutine Instructions**

CALL 16-bit Address /Label

- 3-byte instruction
- It calls subroutine unconditionally
- It saves the content of program counter (PC) on the stack pointer (SP), loads the PC by jump address (16-bit address), program control transfers into that location to execute subroutine.

  E.g., CALL C040H

RET

- 1-byte instruction
- Returns from the subroutine unconditionally
- Inserts the contents of stack pointer to program counter and program control transfers into main program to execute it.

  E.g., RET

b. **Conditional Subroutine Instructions**

CC/CNC/CZ/CNZ/CP/CM/CPE/CPO 16-bit Address /Label

- 3-byte instructions.
- Call subroutine conditionally.
- Same as CALL except that it executes on the basis of flag conditions.

  E.g., CC C050H

RC/RNC/RZ/RNZ/RP/RM/RPE/RPO

- 1-byte instructions.
- Return subroutine conditionally.
- Same as RET except that if executes on the basis of flag conditions.

  E.g., RZ

Examples

Write an ALP to add two numbers using subroutines.

```
8000    MVI B, 4AH
8002    MVI C, A0H
8004    CALL 9000H      ; SP ← 8007H (PC), PC ← 9000H
8007    MOV B, A
8008    HLT
9000    MOV A, B
9001    ADD C
9002    RET             ; PC ← 8007H (SP)
```

WAP to sort in ascending order for 10 bytes from 9000H.

```
START: LXI H, 9000H   ; source
       MVI D, 00H     ; notification
       MVI C, 09H     ; counter
L2:    MOV A, M
       INX H
       CMP M
       JC L1          ; if A<M
       MOV B, M
       MOV M, A
       DCX H
       MOV M, B
       INX H
       MVI D, 01H
L1:    DCR C
       JNZ L2
       MOV A, D
       RRC
       JC START
       HLT
```


3. Restart Instruction

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

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

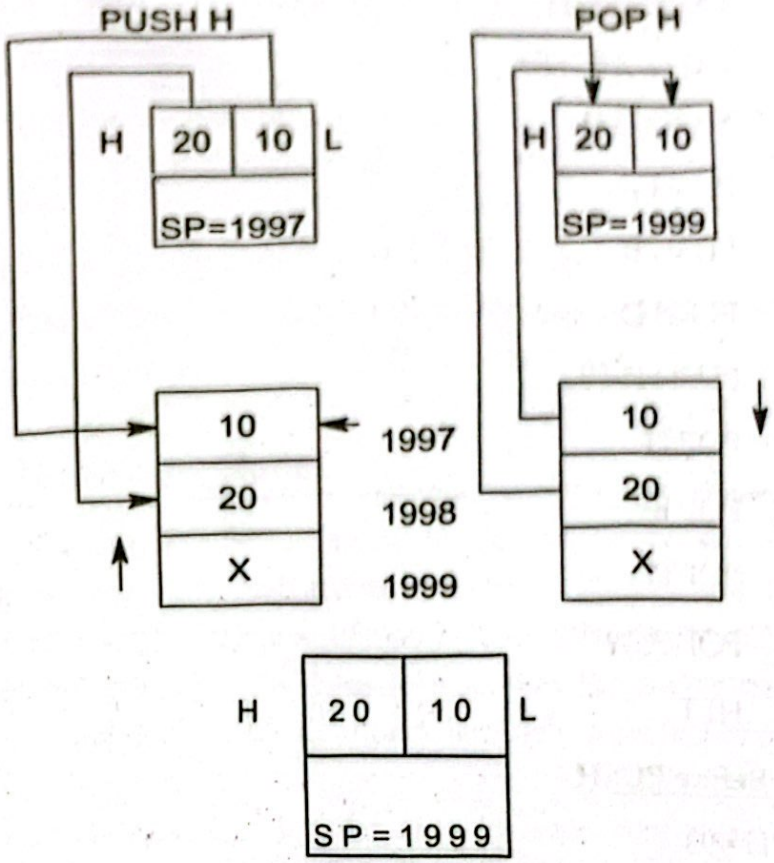
When RST instruction is executed, the 8085 stores the contents of PC on SP and transfers the program to the restart location. Actually these restart instructions are inserted through additional hardware. These instructions are part of interrupt process.

2.4.5 Miscellaneous Group Instructions

STACK

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

The beginning of the stack is defined in the program by using the instruction LXI SP, 16-bit address. Once the stack location is defined, storing of data bytes begins at the memory



Some Other Instructions

XTHL – exchanges top of stack (TOS) with HL

SPHL – move HL to SP

PCHL – move HL to PC

DI – disable interrupt

EI – enable interrupt

SIM – set interrupt mask

RIM – read interrupt mask

NOP – no operation

HLT - halt

Examples

1. **Execute the program below and illustrate the content of all registers before and after using PUSH and POP instructions.**

LXI SP, 8FFFH

LXI H, 9320H

```
LXI B, 4732H
LXI D, ABCDH
MVI A, 34H
PUSH H
PUSH B
PUSH D
PUSH PSW
POP H
POP B
POP D
POP PSW
HLT
```

Before PUSH

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

After POP

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

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

Write a program to set carry flag, reset sign flag and keep other flags as it is.

```
LXI SP, CFFFH
PUSH PSW        ; Push flag into stack
POP H           ; Retrieve flag into L register [HL ← A+F]
MOV A, L
ORI 01H         ; Set carry flag
ANI 7FH         ; Reset sign flag
MOV L, A
PUSH H          ; Push HL into stack
POP PSW         ; Retrieve updated flag into flag register
HLT
```


2.5 Addressing Modes

Instructions are command to perform a certain task in microprocessor. The instruction consists of *op-code* and *operand* [data or address]. The operand may be the source only, destination only or both of them. In these instructions, the source can be a register, a memory, or an input port. Similarly, destination can be a register, a memory location, or an output port. The methods by which the address of source of data or the address of destination of result is given in the instruction are called *addressing modes*. In other words, the various formats (ways) of specifying the operands are called *addressing modes*. So, addressing mode specifies where the operands are located rather than their nature. The 8085 has five addressing modes.

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

   ```
   LDA 9500H
   SHLD C050H
   IN 80H
   ```

   This type of addressing is also called *absolute addressing*.

2. **Register Direct Addressing Mode:** This mode specifies the register or register pair that contains the data rather than address. That means the operand is in general purpose register. Some examples are:

ORA C

XCHG

3. **Register Indirect Addressing Mode:** In this mode, operand part of the instruction specifies the register whose contents are the address of the operand. That me the address of operand is specified by a register pair. So, this type of addressing mode, it is the address of the addr rather than address itself. Some examples are:

STAX B

LDAX D

MOV A, M

4. **Immediate Addressing Mode:** In this mode, the operan position is the immediate data. That means, the operand specified within the instruction itself. For 8-bit dat instruction size is 2 bytes and for 16-bit data, instruction siz is 3 bytes. Some examples are:

MVI A, 32H

LXI B, 4567H

SUI 95H

XRI 55H

5. **Implied or Inherent Addressing Mode:** The instructions o this mode do not have operands. If address of source of dat as well as address of destination of result is fixed, then there is no need to give any operand along with the instruction Some examples are:

NOP

HLT

CMA

EI

2.6 Time Delay and Counter

Counter

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

Loop

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

Loops can be classified into two categories:

- **Continuous loop**

  It is set up by using the unconditional Jump instruction. A program with a continuous loop does not stop repeating the tasks until the system is reset.

- **Conditional loop**

  It is set up by using the conditional Jump instructions. These instructions check flags and repeat the specified tasks if the conditions are satisfied. These loops usually include counting and indexing.

The looping in 8085 microprocessor is performed by following steps:

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

Time Delay

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

Suppose, we have an 8085 microprocessor with 2 MHz clo
frequency.

Clock frequency of system (f) = 2 MHz

Clock period (T) = $\frac{1}{f} = \frac{1}{2} \times 10^{-6} = 0.5\ \mu s$

Time delay example:

	Instructions	T-Sates
	MVI C, FFH	7
LOOP:	DCR C	4
	JNZ LOOP	10/7

Here, register C is loaded with count FFH (255_{10}) by usin
MVI which takes 7 T-states. Time to execute MVI instructio
(outside loop) = 7 T-states × 0.5 = 3.5 μs

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

T_l = (T × T-states of loop × count)

Where,

T_l = time delay in loop

T= system clock period

Count = decimal value for counter

$T_l = 0.5 \times 10^{-6} \times 14 \times 255 = 1785\ \mu s$

But JNZ takes only 7 T-states when exited from loop. Adjusted loop delay is calculated as

$T_{la} = T_l - (3\ T\text{-states}) = 1785\ \mu s - 3 \times 0.5\ \mu s = 1783.5\ \mu s$

Total delay loop of program is expressed as

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

= 3.5 μs + 1783.5 μs = 1787 μs = 1.8 ms

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

		T-States	Clocks
	MVI B, 40H; 64	7	7×1
L2:	MVI C, 80H; 128	7	7×64
L1:	DCR C	4	4×128×64
	JNZ L1	10/7	(10×127+7×1) ×64
	DCR B	4	4×64
	JNZ L2	10/7	10×63+7×1
	RET	10	10×1
			Total clocks =115854

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

2.7 Number Conversion

2.7.1 BCD to Binary Conversion

In most microprocessor-based products, data are entered and displayed in decimal numbers. For example, in an instrumentation laboratory, readings such as voltage and current are maintained in decimal numbers, and data are entered through decimal keyboard. The system monitors program of the instrument, converts each key into an equivalent 4-bit binary number, and stores two BCD numbers in an 8-bit register or a memory location. These numbers are called packed BCD.

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

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

unpacked BCD digits i.e., BCD_1 and BCD_2.

0111 0010 → 0000 0010 (02H) Unpacked BCD_1

→ 0000 0111 (07H) Unpacked BCD_2

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

  $BCD_1 = 02H$

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

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

  $02H + 46H = 48H$

Example

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

```
    LXI H, 2020H
    MVI E, 0A H
    MOV A, M        ; 0111 0010
    ANI F0H         ; 0111 0000
    RRC
    RRC
    RRC
    RRC
    MOV B,A
    XRA A
L1: ADD B           ; 7× 10+2
    DCR E
    JNZ L1
    MOV C, A
    MOV A, M
    ANI 0FH
    ADD C
    STA 2030H
    HLT
```

2.7.2 Binary to BCD Conversion

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

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

Else subtract 100 (64H) repetitively.

Quotient is BCD_1 (Divide by 100)

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

Else subtract 10 (0AH) repetitively.

Quotient is BCD_2 (Divide by 10)

Step 3: Remainder from step 2 is BCD_3

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

Example

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

```
    LXI SP, 1999H
    LXI H, 2020H;     Source
    MOV A, M
    CALL PWRTEN
    HLT
PWRTEN:
    LXI H, 2030H;     Destination
    MVI B, 64H
```

```
        CALL BINBCD
        MVI B, 0AH
        CALL BINBCD
        MOV M, A
        RET
BINBCD:
        MVI M, FFH
NEXT:
        INR M
        SUB B
        JNC NEXT
        ADD B
        INX H
        RET
```

2.7.3 Binary to ASCII Conversion

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

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

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

If number < 10, then add 30H
Else add 37H (30H + 07H)
For example: A = A + 30H + 07H = 41H

Example

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

```
        LXI SP, 1999H
        LXI H, 1120H  ; Source
        LXI D, 1160H  ; Destination
        MOV A, M
        ANI F0H
        RRC
        RRC
        RRC
        RRC
        CALL ASCII
        STAX D
        INX H
        MOV A, M
        ANI 0FH
        CALL ASCII
        STAX D
        HLT
ASCII:  CPI 0AH
        JC BELOW
        ADI 07H
BELOW:  ADI 30H
        RET
```

2.7.4 ASCII to Binary Conversion

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

Step 1: Subtract 30H

Step 2: If < 0AH, then binary as it is

Else subtract 07H

E.g., if ASCII is 41H, then 41H – 30H = 11H; 11H – 07H = 0AH

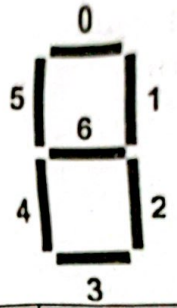
Example

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

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

2.7.5 BCD to 7-Segment LED Code Conversion

Each segment in a seven-segment display is identified by an index from 0 to 6, with the positions given in figure. For this conversion, *table lookup technique* (TLT) is used. In TLT, the codes of digits to be displayed are stored sequentially in memory so that these codes can be used effectively and efficiently.



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

Example

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

```
LXI SP, 1999H
LXI H, 1150H
```

```
        MVI D, 00H
        LXI B, 1190H
NEXT:   MOV A, M
        ANI F0H
        RRC
        RRC
        RRC
        RRC
        CALL CODE
        INX B
        MOV A, M
        ANI 0FH
        CALL CODE
        INX B
        INX H
        DCR D
        JNZ NEXT
        HLT
CODE:   PUSH H
        LXI H, 1170H
        ADD L
        MOV L, A
        MOV A, M
        STAX B
        POP H
        RET
```


2.8 Multiplication and Division

2.8.1 Multiplication

Multiplication can be performed by repeated addition. However it is inefficient technique for a large multiplier. A more efficient technique can be devised by following the manual multiplication of decimal numbers. Each bit of multiplier is taken one-by-one and it is checked whether it is 1 or 0. If the bit of multiplier is 1, the multiplicand is added to the product and the product is shifted to left. When the bit of multiplier is 0, the product is simply shifted to left by one bit.

Example

1. Write a program to multiply two 8-bit numbers stored at C050H where multiplicand at C050H and multiplier at C051H; store 16-bit result at memory location C052H.

```
        MVI D, 00H
        LDA C050H
        MOV E, A        ; Multiplicand in DE
        LDA C051H       ; Multiplier in A
        LXI H, 0000H
        MVI C, 08H      ; Counter
LOOP1:  DAD H           ; Left shift partial product by 1 bit
        RAL             ; Rotate multiplier by 1 bit
        JNC NEXT
        DAD D           ; Product = Product + Multiplicand
NEXT:   DCR C
        JNZ LOOP1
        SHLD C052H
        HLT
```

2.8.2 Division

Division can be performed by repetitive subtraction. The divisor is subtracted from the dividend until dividend will be less than the divisor. After successful subtraction, quotient will be

cremented by 1. Final value of multiplicand registers will b
mainder.

mple

Write a program to divide 16-bit number stored in mem locations 8800H and 8801H by the 8-bit number stored memory location 8802H. Store the quotient in mem locations 8900H and 8901H and remainder in mem locations 8902H and 8903H.

```
        LHLD 8800H      ; Dividend in HL
        LDA 8802H
        MOV C, A        ; Divisor in C
        LXI D, 0000H    ; Quotient = 0
NEXT:   MOV A, L
        SUB C           ; Subtract divisor
        MOV L, A        ; Save partial result
        JNC SKIP        ; if CY 1 jump
        DCR H ; Subtract borrow of previous subtraction
SKIP:   INX D           ; Increment quotient
        MOV A, H
        CPI 00H         ; Check if dividend < divisor
        JNZ NEXT        ; if no repeat
        MOV A, L
        CMP C
        JNC NEXT
        SHLD 8902H      ; Store the remainder
        XCHG
        SHLD 8900H      ; Store the quotient
        HLT
```


SOLUTION TO IMPORTANT QUESTIONS

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

⇒

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

2. Registers BC contain 2793H and registers DE contain 3182H. Write instructions to add these two 16 bit numbers and place the sum in memory locations 2050H and 2051H. *[2062 Baishakh]*

⇒

```
        LXI B, 2793H
        LXI D, 3182H
        MOV A, C
        ADD E
        MOV L, A
        MOV A, B
        ADC D
        MOV H, A
        SHLD 2050H
        HLT
```

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

```
MVI L, ....            ... 16 bits
MOV E, M        ;get LS byte of 16 bits data to E
INX H           ;increment the memory pointer
MOV D, M        ;get MS byte of the 16 bit data to D
REPEAT: INX H   ;increment the memory pointer
MOV A,M         ;get LS Byte of NEXT data to A
ADD E           ;add it with PREVIOUS data in E
DAA
MOV E,A
INX H           ;increment the memory pointer
MOV A,M         ;get MS Byte of NEXT data to A
ADC D           ;add it with PREVIOUS data in D
DAA
MOV D,A
JNC PASS
MOV A,B
ADI 01H         ;increment the carry by 1
DAA
PASS:   DCR C   ;update counter
JNZ REPEAT ;continue for 10 words
INX H           ;24 bit results in B, D, E respectively
MOV M,E         ;storing final 24 bit sum at end of
                ;table
INX H
MOV M,D
INX H
MOV M,B
HLT
```

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

⇒

```
      LXI H, 9050H    ; loads memory in HL
      MVI C, 32H      ; counter for 50 codes.
NEXT: MOV A, M
      CPI 61H         ; compares with ASCII value of 'a'
      JC NOC
      CPI 7BH         ; compare with ASCII value of 'z' + 1
      JNC NOC
      SUI 20H         ; convert uppercase ASCII to
                      ; lowercase ASCII
      MOV M, A
NOC:  INX H
      DCR C
      JNZ NEXT
      HLT
```

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

⇒

```
        LXI B, 9080H    ; source
        LXI D, 9090H    ;destination
        MVI L, 0AH      ; counter
LOOP1:  LDAX, B         ; load A with content of [BE]
        MOV H, A        ; H←A
        ANI 28H         ; AND with 0010 1000
        CPI 20H
        JNZ CHANGE
        MOV A, H        ; A←H
        JMP STORE
CHANGE: MOV A,H         ; A←H
        XRI 44H   ; A←data with charged bit of D2 & D6
STORE:  STAX D
        INX B
        INX D
```

```
        CPI 5AH          ; L from 50H to 59H
        JNZ NEXT
        HLT
```

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

⇒ Assume: The data are located from C050H

```
        LXI H, C050H
        MOV C, 10H     ; counter 16
        MOV B, 00H     ; count-ve element
NEXT:   MOV A, M
        RLC
        JNC DOWN
        INR B
DOWN:   INX H
        DCR C
        JNZ NEXT
        MOV A, B
        CPI 09H
        JNC STORE
        MVI A, 00H
STORE:  MOV M, A
        HLT
```

8. **Write a program for 8085 to swap nibbles (upper four bits and lower four bits) of ten eight bit number stores at 8000H and transfer to new location 8050H if the number have D_5=1 else store FFH in the destination.** *[2066 Shrawan]*

⇒

```
        LXI  H, 8000H      ; source
        LXI  D, 8050 H     ; destination
        MVI B, 0A H        ; counter
NEXT:   MOV A, M
        MOV  C, A
```

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

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

⇒ *Assume 1st table is at 2050H, 2nd table is at 2060 H, and 3rd tabl... at 2070 H.*

```
        LXI SP, 2FFFH
        LXI H, 2050H    ;  Source 1st table
        LXI D, 2060H    ;  Source 2nd table
        LXI B, 2070H    ;  Destination
NEXT:   PUSH B      ;Stores the BC content in stack
        LDAX D      ;load accumulator with data of [DE]
        MOV B,A       ;B←2nd table content B
        MOV A,M       ;A←[HL] 1st table content
        CMP B         ;carry occurs when B>A
                    ;i.e., 2nd table is greater than 1st table
```



```
        JC ADDITION   ; 1st table content<2nd table
        SUB B         ; subtract 2nd table from 1st table
        JMP BELOW
ADDITION: ADD B       ; add 1st table and 2nd table
BELOW:  POP B
        STAX B        ; stores addition or subtraction
        INX H
        INX D
        INX B
        MOV A,L       ; counter for 10
        CPI 5A
        JNZ NEXT
        HLT
```

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

⇒ *Assume there are ten numbers in a table starting from C050H*

```
        LXI H, C050H    ; source
        MVI C, 0AH      ; counter
        LXI D, 0000H    ; for sum
NEXT:   MOV A, M
        ANI F0H         ; to get upper nibble
        RLC
        RLC
        RLC
        RLC             ; A←upper nibble
        CPI 07H
        JC SKIP   ; if number < 07 i.e. no > 06
        MOV A, M
        ADD E           ; sum in E
        MOV E, A
        JNC SKIP
```

```
        INR D          ; carry in D
SKIP:   INX H
        DCR C
        JNZ NEXT
        MOV M, E
        INX H
        MOV M, D
        HLT
```

11. A set of three reading is stored in memory startin 9040H. Write an assembly language program to sor readings in ascending order. Store the smallest valu address 9054H and so on in higher addresses. *[2067 Man*

⇒ Set of three numbers means there are 06 numbers [three bit numbers] in a table.

```
        MVI D, 06H     ; Main Counter
AGAIN:  LXI H, 9040H   ; Source
        MVI C, 06H     ; Sub Counter
NEXT:   MOV A, M
        INX H
        CMP M
        JC NOSWAP      ; For ascending order
        MOV B, M
        MOV M, A
        DCX H
        MOV M, B
        INX H
NOSWAP: DCR C
        JNZ NEXT
        DCR D
        JNZ AGAIN
        LXI H, 9040H
        LXI D, 9054H
```



```
        MVI C, 06H
STORE:  MOV A, M
        STAX D
        DCR C
        JNZ STORE
        HLT
```

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

⇒

```
        LXI H, 9350H      ; source
        LXI D, 9450 H     ; destination
        MVI C, 0AH        ; counter
NEXT:   MOV A, M
        CPI 91H           ;CY flag set for data<=90H
        JNC SWAP
        ADI 48H
        JMP BELOW
SWAP:   MOV B, A
        ANI 44H           ; A←all bits zero except
                          ;D6& D2
        CPI 44H           ; D6& D2both are 1
        JZ BELOW          ; no need to swap
        CPI 00H           ; D6& D2both are 0
        JZ BELOW          ; no need to swap
        MOV A, B
        XRI 44H           ; swap D6 and D2
BELOW:  STAX D
        INX D
        INX H
```

```
        DCR C
        JNZ NEXT
        HLT
```

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

⇒

```
        LXI H, A000H    ; source
        LXI D, A030H    ; destination
        MVI C, 0AH      ; counter
NEXT:   MOV A, M
        CPI 25H
        JC BELOW
        CPI 70H
        JNC BELOW
        STAX D
        INX D
BELOW:  INX H
        DCR C
        JNZ NEXT
        HLT
```

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

Suppose first table is having ten 8-bit numbers starting from A000H and second table starting from A030H.

```
        LXI H, A000H    ; source
        LXI D, A030H    ; destination
        MVI C, 0AH      ; counter
NEXT:   MOV A, M
        CPI 80H
        JC BELOW        ; if number < 80H
        ANI BFH         ; resets D6
        JMP STORE
BELOW:  ORI20H          ; sets D5
STORE:  STAX D
        INX H
        INX D
        DCR C
        JNZ NEXT
        HLT
```

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

⇒

```
        LXI H, 8090H    ; source
        MVI C, C8H      ; counter 200
NEXT:   MOV A, M
        ANI 44H
        CPI 44H         ; check if both bits are 1
        JZ NOSWAP
        CPI 00H         ; check if both bits are 0
        JZ NOSWAP
        MOV A,M
        XRI 44H         ; swap bits D6 and D2
        MOV M, A
NOSWAP: INX H
        DCR C
        JNZ NEXT
        HLT
```

after the table.

```
        LXI H, 8B20H    ; source
        LXI D, 8B2AH    ; destination
        MVI C, 0AH      ; counter
NEXT:   MOV A, M
        ANI F0H
        RRC
        RRC
        RRC
        RRC
        MOV B, A        ; upper nibble
        MOV A, M
        ANI 0FH         ; lower nibble
        ADD B
        STAX D
        INX H
        INX D
        DCR C
        JNZ NEXT
        HLT
```

Write a program in 8085 to calculate the number of ones in the upper nibble of ten 8-bit numbers stored in table. Store the count of ones in a location just after the table. *[2072 Ashwin]*

⇒ *Suppose table starts from 8050H.*

```
        LXI H, 8050H    ; Source
        MVI C, 0AH      ; Counter
        MVI D, 00H      ; to counter result
```



```
NEXT:   MOV A, M
        ANI F0H
        MVI E, 04H      ; upper nibble counter
CHECK:  RLC
        JNC BELOW
        INR D
BELOW:  DCR E
        JNZ CHECK
        INX H
        DCR C
        JNZ NEXT
        MOV M, D
        HLT
```

18. Write a program for 8085 to generate multiplication table of a number stored at 8230H and store the generated table starting at 8231H. For example, if location 8230H has number 05H then store 05H at 8231H, 0AH at 8232H and so on. *[2072 Magh]*

⇒

```
        LXI H, 8231H    ; destination
        MVI C, 0AH      ; counter
        LDA 8230H       ; source
        MOV B, A
        MVI A, 00H
NEXT:   ADD B
        MOV M, A
        INX H
        DCR C
        JNZ NEXT
        HLT
```

19. **Write an assembly language program for 8085 to find the square of ten 8-bit numbers which are ≤ 0FH, stored from memory location C090H. Store the result from the end of the source table.** *[2073 Magh]*

⇒

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

20. **There are two tables holding twenty data whose starting address is 9000H and 9020H respectively. WAP to add the content of first table with the content of second table having same array index. Store sum and carry into the third and fourth table indexing from 9040H and 9060H respectively.** *[2074 Bhadra]*

```
        LXI B, 9000H   ; Table 1
        LXI H, 9020H   ; Table 2
NEXT:   MVI E, 00H     ; for carry
        LDAX B
        ADD M
        MOV D, A       ; sum
        JNC BEL
        MVI E, 01H
BEL:    PUSH B
        PUSH H
        MOV A, C
        ADI 40H
        MOV C, A       ; makes [BC] as Table 3
        MOV A, D
        STAX B         ; stores sum in Table 3
        MOV A, L
        ADI 40H
        MOV L, A       ; makes [HL] as Table 4
        MOV M, E       ; stores carry in Table 4
        POP H
        POP B
        INX B
        INX H
        MOV A, C
        CPI 14H        ; checks for 20 data
        JNZ NEXT
        HLT
```

21. **Write a program for 8085 to count the numbers for which upper nibble is higher than the lower nibble; and store the count at the end of table having 50 bytes data from C050H.** *[2075 Bhadra]*

⇒

```
        LXI H, C050H   ; Source
        MVI C, 32H     ; Counter
        MVI D, 00H     ; Counter for upper nibble greater
                       ;than lower nibble
NEXT:   MOV A, M
        ANI F0H
        RLC
        RLC
        RLC
        RLC
```

```
        ANI 0FH
        CMP B
        JNC BELOW
        INR D
BELOW:
        INX H
        DCR C
        JNZ NEXT
        MOV M, D
        HLT
```

22. Write a program in 8085 to find the largest and smallest bytes from the list of 20 bytes stored starting from memory location C050H. Store the largest byte and smallest byte in C070H and C071H respectively. *[2076 Baishakh]*

⇒

```
        MVI D, 14H    ; Main Counter
AGAIN:  LXI H, C050H  ; Source
        MVI C, 14H    ; Sub Counter
NEXT:   MOV A, M
        INX H
        CMP M
        JC NOSWAP     ; For ascending order
        MOV B, M
        MOV M, A
        DCX H
        MOV M, B
        INX H
NOSWAP: DCR C
        JNZ NEXT
        DCR D
        JNZ AGAIN
```



```
        MOV A, M
        STA C070H   ; Largest number
        LDA C050H
        STA C071H   ; Smallest number
        HLT
```

Alternative Method

```
        MVI D, 14H      ; Counter
        LXI H, C050H    ; Source
        MVI D 00H       ; Largest Number
        MVI E, 00H      ; Smallest Number
NEXT:   MOV A, M
        CMP D
        JC BEL
        MOV D, A
BEL:    CMP E
        JNC BEL1
        MOV E, A
BEL1:   INX H
        DCR C
        JNZ NEXT
        MOV A, D
        STA C070H   ; Largest number
        MOV A, E
        STA C071H   ; Smallest number
        HLT
```

23. Write a program in 8085 to sort the 10 data bytes stored in a table in descending order. The data bytes are stored in a table from memory address 8920H. *[2076 Bhadra]*

⇒

```
        MVI D, 0AH; counter for outer loop
AGAIN: LXI H, 8920H; source
        MVI C, 0AH; counter for inner loop
```

```
NEXT: MOV A, M
      INX H
      CMP M
      JNC NOSWAP; if descending
      MOV B, M
      MOV M, A
      DCX H
      MOV M, B
      INX H
NOSWAP: DCR C
      JNZ NEXT
      DCR D
      JNZ AGAIN
      HLT
```

24. **Write a program in 8085 to count the odd and even parity numbers of 150 data stored in the memory location C050H. Store the counts at memory locations D000H and D001H.**

[2077 Chaitra]

⇒

```
      LXI H, C050H; source
      MVI C, 96H; count 150
      MVI D, 00H ; count for odd parity
      MVI E, 00H; count for even parity
NEXT : MOV A, M
      ADI 00H
      JPE BEL
      INR D
      JMP DOWN
BEL : INR E
DOWN : INX H
      DCR C
      JNZ NEXT
      MOV A, D
      STA D000H
      MOV A, E
      STA D001H
      HLT
```

25. **There are 40 8-bit numbers in a table with address starting from 9090H. Write a program in 8085 to transfer these numbers to another table with address A010H if lower nibble of a number is greater than higher nibble. Otherwise transfer by setting bit D_2 and resetting bit D_6.**

[2078 Baishakh]

⇒

```
      MVI C, 28H; counter
      LXI H, 9090H; source
      LXI D, A010H; destination
NEXT: MOV A, M
      ANI 0FH; lower nibble
      MOV B, A
      MOV A, M
      ANI F0H
      RLC
      RLC
      RLC
      RLC; higher nibble
```

```
        CMP B
        JC BEL
        MOV A, M
        ORI 04H; setting D2
        ANI BFH; resetting D6
BEL: STAX D
        INX H
        INX D
        DCR C
        JNZ NEXT
        HLT
```

❖ ❖ ❖