NEHRU COLLEGE OF ENGINEERING AND RESEARCH CENTRE
(NAAC Accredited)
(Approved by AICTE, Affiliated to APJ Abdul Kalam Technological University, Kerala)

DEPARTMENT OF MECHATRONICS ENGINEERING

COURSE MATERIALS

MRT206 MICROPROCESSOR AND EMBEDDED SYSTEMS

VISION OF THE INSTITUTION
To mould true citizens who are millennium leaders and catalysts of change through excellence in education.
MISSION OF THE INSTITUTION
NCERC is committed to transform itself into a center of excellence in Learning and Research in Engineering and Frontier Technology and to impart quality education to mould technically competent citizens with moral integrity, social commitment and ethical values. We intend to facilitate our students to assimilate the latest technological know-how and to imbibe discipline, culture and spiritually, and to mould them in to technological giants, dedicated research scientists and intellectual leaders of the country who can spread the beams of light and happiness among the poor and the underprivileged.

ABOUT DEPARTMENT

- Established in: 2013
- Course offered: B.Tech Mechatronics Engineering

- ♦ Approved by AICTE New Delhi and Accredited by NAAC
- ♦ Affiliated to the University of Dr. A P J Abdul Kalam Technological University.

DEPARTMENT VISION

To develop professionally ethical and socially responsible Mechatronics engineers to serve the humanity through quality professional education.

DEPARTMENT MISSION

1) The department is committed to impart the right blend of knowledge and quality education to create professionally ethical and socially responsible graduates.
2) The department is committed to impart the awareness to meet the current challenges in technology.
3) Establish state-of-the-art laboratories to promote practical knowledge of mechatronics to meet the needs of the society

PROGRAMME EDUCATIONAL OBJECTIVES

I. Graduates shall have the ability to work in multidisciplinary environment with good professional and commitment.
II. Graduates shall have the ability to solve the complex engineering problems by applying electrical, mechanical, electronics and computer knowledge and engage in lifelong learning in their profession.
III. Graduates shall have the ability to lead and contribute in a team with entrepreneur skills, professional, social and ethical responsibilities.
IV. Graduates shall have ability to acquire scientific and engineering fundamentals necessary for higher studies and research.

PROGRAM OUTCOME (PO'S)

Engineering Graduates will be able to:

PO 1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO 2. Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO 3. Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO 4. Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO 5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO 6. The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO 7. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO 8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO 9. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO 10. Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO 11. Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO 12. Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOME(PSO'S)

PSO 1: Design and develop Mechatronics systems to solve the complex engineering problem by integrating electronics, mechanical and control systems.

PSO 2: Apply the engineering knowledge to conduct investigations of complex engineering problem related to instrumentation, control, automation, robotics and provide solutions.

MRT206	MICROPROCESSOR & EMBEDDED SYSTEMS	CATEGORY	L	T	P	CREDIT
		PCC	3	1	0	4

Preamble:

The Purpose of the course is to provide the students the knowledge of Microprocessors, Microcontroller and embedded systems. This course is emphasis on architecture, Programming and system design of 8085 microprocessor and 8051 microcontrollers. The course is intended for making the basic knowledge in Embedded systems, Embedded C and development tools.

Prerequisite:

MRT203 DIGITAL AND ANALOG CIRCUITS

Course Outcomes: After the completion of the course the student will be able to

CO 1	**Understand the basic concepts of 8085 microprocessor**
CO 2	**Understand the basic concepts of 8085 interfacing with input output devices and memory device**
CO 3	**Understand the overview of an Embedded Systems**
CO 4	**Interpret the basic concepts of 8051 microcontroller**
CO 5	**Interface peripheral devices with 8051 microcontrollers**
CO 6	**Write C/Assembly Program for a microcontroller**

Mapping of course outcomes with program outcomes

	PO 1	**PO 2**	**PO 3**	**PO 4**	**PO 5**	**PO 6**	**PO 7**	**PO 8**	**PO 9**	**PO 10**	**PO 11**	**PO 12**
CO 1	3	2	2	2	3				1			3
CO 2	3	3	3	3	3				1			3
CO 3	3	2	2	2	1				1			3
CO 4	3	2	2	2	1				1			3
CO 5	3	3	3	3	3				1			3
CO 6	3	3	3	3	3				1			3

Assessment Pattern

Bloom's Category	**Continuous Assessment Tests**	**End Semester Examination**

	1	2	
Remember	10	10	10
Understand	20	20	20
Apply	20	20	70
Analyse			
Evaluate			
Create			

Mark distribution

Total Marks	**CIE**	**ESE**	**ESE Duration**
150	50	100	3 hours

Continuous Internal Evaluation Pattern:

Attendance	: 10 marks
Continuous Assessment Test (2 numbers)	: 25 marks
Assignment/Quiz/Course project	: 15 marks

End Semester Examination Pattern: There will be two parts; Part A and Part B. Part A contain 10 questions with 2 questions from each module, having 3 marks for each question. Students should answer all questions. Part B contains 2 questions from each module of which student should answer any one. Each question can have maximum 2 sub-divisions and carry 14 marks.

Course Level Assessment Questions

Course Outcome 1 (CO1):

1. Describe various interrupt sources on a 8085 processor

2. List the various jump instructions by 8085 processor

3. Develop a assembly program to sort N number in ascending order

Course Outcome 2 (CO2)

1. State the functionality of Program counter in a microprocessor

2. Describe memory interface in 8085 processor

3. Define the instruction cycle for an 8085 processor

Course Outcome 3(CO3):

1. List the various tools used in embedded systems development

2. Differentiate a Microprocessor and Microcontroller

3. Describe the features and characteristics of embedded systems

Course Outcome 4 (CO4):

1. Describe 8051 architecture with a neat block diagram.

2. Illustrate Memory organization in 8051 microcontrollers.

3. Describe addressing modes of 8051 with example

Course Outcome 5 (CO5):

1. Show the program for generating 1 KHz signal

2. Demonstrate the working of serial peripheral in 8051

3. Design a system to actuate a stepper motor to 45 degree clock wise

Course Outcome 6 (CO6):

1.Show the program to add two 16-bit number using 8051 controllers

2. Write a C program to send string "Hello" through serial port

3. Demonstrate bit manipulating instruction with example

Model Question paper

Course Code: MRT206

Course Name: MICROPROCESSOR & EMBEDDED SYSTEMS

Max.Marks:100 **Duration: 3 Hours**

PARTA

Answer all Questions. Each question carries 3 Marks

1. Describe flag register in the 8085 microprocessors

2. Differentiate register and memory addressing mode with an example

3. Discuss mode 1 of 8255 PPI with diagram

4. Draw the timing diagram for Memory Read operation.

5. Differentiate between hard & soft real time systems.

6. What are the demerits of Waterfall Model?

7. Explain the following instructions used in 8051 microcontrollers.

i) MOV R1, #05H ii) ADD A, #01H iii) MOV R2, 07H

8. Explain with neat diagram the RAM of 8051.

9. Define the structure of an Embedded C program

10. Explain I/O ports and its functions in 8051.

PART B

Answer any one full question from each module. Each question carries 14 Marks

Module 1

11.a. Draw and explain 8085 Architecture with neat diagram

b. List the various jump instructions by 8085 processor

12 a. Develop an assembly program to sort N number in ascending order

Module 2

13. Design a LED blinking system with 8085 and 8255

14.a. Explain Fetch cycle & Execute cycle in 8085.

b. Describe memory interface in 8085 processor

Module 3

15. Explain i) Compiler ii) Assembler iii) Linker iv) Loaders.

16. a. List the field of applications for an embedded system.

b. List out the challenges in Embedded Systems.

Module 4

17 a. Write an ALP in 8051 to add two 32-bit numbers & store the result.

b. Explain with neat diagram the Register organisation and SFR in 8051.

18 Explain with neat block diagram the architecture of 8051 Microcontroller

Module 5

19. Write a C program to send string "Hello" through serial port

20. Explain with suitable diagram and program, how an ADC can be interfaced with 8085 Microprocessor.

Syllabus

Module	**Topics**	**Hr**
1	8085 Microprocessor: Evolution of Microprocessors- 8085 Architecture – Addressing modes- Classification of Instruction set- Interrupts-introduction to assembly language programming –code conversion, sorting–binary and BCD arithmetic.	**9**
2	Timing and control–Machine cycles, instruction cycle and T states–fetch and execute cycles– Timing diagram for instructions. IO and memory interfacing –Address decoding–I/O ports – Programmable peripheral interface PPI 8255 -Modes of operation. Interfacing of LEDs	**9**
3	Introduction to Embedded Systems-Application domain of embedded systems, features and characteristics, System model, Microprocessor Vs Microcontroller, current trends and challenges, hard and soft real time systems, Embedded product development, Life Cycle Management (water fall model), Tool Chain System, Assemblers, Compilers, linkers, Loaders, Debuggers Profilers & Test Coverage Tools-cross compilation	**9**
4	8051 Microcontroller: Selection of Microcontrollers - 8051 Microcontroller Architecture-Memory organization –Special function registers –Addressing modes – Instruction set - Introduction to assembly language programming using 8051(basic arithmetic operations)- Interrupts.	**9**
5	Embedded C Programming: structure of an embedded C program -data type-key words- basic programming using embedded C (bit level manipulations-accessing and configuring of different status, control and peripheral registers) Peripheral Programming: I/O port programming – Timer programming – Serial communication programming – Peripheral Interfacing diagram and programming of A/D and D/A converters, Stepper motor.	**9**

Text Books

1. Ramesh S Gaonkar, Microprocessor Architecture, Programming and applications with the 8085, Architecture, Programming and Applications, Penram International Publishing PVT Ltd. 6th Edition

2.Mazidi Muhammad Ali, Mazidi Janice Gillispie and McKinlayRolin, ―The 8051 Microcontroller and Embedded Systems, 2 nd Edition, Prentice Hall of India, New Delhi, 2013.

3. Lyla B Das – Embedded Systems – An Integrated Approach, Pearson Publication, sixth edition 2014

Reference Books

1.Douglas V. Hall, Microprocessors and Interfacing, Tata McGraw Hill, Education, New

2. Mathur A., Introduction to Microprocessors, Tata McGraw Hill, New Delhi,1992.

3. Rafiquzzaman, Microprocessor Theory and Application, PHI Learning, First Edition. 7.

4. Ray A joy and Burchandi, Advanced Microprocessor & Peripherals, Tata McGraw Hill, Education, New Delhi, Second Edition.

5. Scott MacKenzie, Raphael C W Phan, "The8051Microcontroller", Fourth Edition, Pearson education Delhi, Third Edition. /Prentice hall of India International Publishing; Sixth edition,2014.

Course Contents and Lecture Schedule

No	Topic	No. of Lectures
1	8085 Microprocessor	
1.1	Evolution of Microprocessors- 8085 Architecture	1
1.2	Addressing modes	1
1.3	Classification of Instruction set	3
1.4	Interrupts	2
1.5	Introduction to assembly language programming –code conversion, sorting–binary and BCD arithmetic.	2
2	8085 Interfacing	
2.1	Timing and control–Machine cycles, instruction cycle and T states	2
2.2	fetch and execute cycles– Timing diagram for instructions.	2
2.3	IO and memory interfacing	1
2.4	Address decoding–I/O ports	1
2.5	Programmable peripheral interface PPI8255 -Modes of operation.	2
2.6	Interfacing of LEDs	1

3	Introduction to Embedded Systems	
3.1	Application domain of embedded systems, features and characteristics, System model	2
3.2	Microprocessor Vs Microcontroller, current trends and challenges, hard and soft real time systems,	2
3.3	Embedded product development, Life Cycle Management (water fall model)	2
3.4	Tool Chain System, Assemblers, Compilers, linkers, Loaders, Debuggers Profilers & Test Coverage Tools-cross compilation	3
4	8051 Microcontroller	
4.1	Selection of Microcontrollers - 8051 Microcontroller Architecture	1
4.2	Memory organization	1
4.3	Special function registers	1
4.4	Addressing modes	1
4.5	Instruction set	2
4.6	Introduction to assembly language programming using 8051(basic arithmetic operations)	2
4.7	Interrupts.	1
5	Embedded C Programming	
5.1	structure of an embedded C program -data type-key words- basic programming using embedded C (bit level manipulations-accessing and configuring of different status, control and peripheral registers)	3
5.2	I/O port programming	1
5.3	Timer programming	1
5.4	Serial communication programming	1
5.5	Peripheral Interfacing diagram and programming of A/D and D/A converters, Stepper motor.	3

MODULE 1

History of microprocessor:-

The invention of the transistor in 1947 was a significant development in the world of technology. It could perform the function of a large component used in a computer in the early years. Shockley, Brattain and Bardeen are credited with this invention and were awarded the Nobel prize for the same. Soon it was found that the function this large component was easily performed by a group of transistors arranged on a single platform. This platform, known as the integrated chip (IC), turned out to be a very crucial achievement and brought along a revolution in the use of computers. A person named Jack Kilby of Texas Instruments was honored with the Nobel Prize for the invention of IC, which laid the foundation on which microprocessors were developed. At the same time, Robert Noyce of Fairchild made a parallel development in IC technology for which he was awarded the patent.

ICs proved beyond doubt that complex functions could be integrated on a single chip with a highly developed speed and storage capacity. Both Fairchild and Texas Instruments began the manufacture of commercial ICs in 1961. Later, complex developments in the IC led to the addition of more complex functions on a single chip. The stage was set for a single controlling circuit for all the computer functions. Finally, Intel corporation's Ted Hoff and Frederico Fagin were credited with the design of the first microprocessor.

The work on this project began with an order from a Japanese calculator company Busicom to Intel, for building some chips for it. Hoff felt that the design could integrate a number of functions on a single chip making it feasible for providing the required functionality. This led to the design of Intel 4004, the world's first microprocessor. The next in line was the 8 bit 8008 microprocessor. It was developed by Intel in 1972 to perform complex functions in harmony with the 4004.

This was the beginning of a new era in computer applications. The use of mainframes and huge computers was scaled down to a much smaller device that was affordable to many. Earlier, their use was limited to large organizations and universities. With the advent of microprocessors, the use of computers trickled down to the common man. The next processor in line was Intel's 8080 with an 8 bit data bus and a 16 bit address bus. This was amongst the most popular microprocessors of all time.

Very soon, the Motorola corporation developed its own 6800 in competition with the Intel's 8080. Fagin left Intel and formed his own firm Zilog. It launched a new microprocessor Z80 in 1980 that was far superior to the previous two versions. Similarly, a break off from Motorola prompted the design of 6502, a derivative of the 6800. Such attempts continued with some modifications in the base structure.

The use of microprocessors was limited to task-based operations specifically required for company projects such as the automobile sector. The concept of a 'personal computer' was still a distant dream for the world and microprocessors were yet to come into personal use. The 16 bit microprocessors started becoming a commercial sell-out in the 1980s with the first popular one being the TMS9900 of Texas Instruments.

Intel developed the 8086 which still serves as the base model for all latest advancements in the microprocessor family. It was largely a complete processor integrating all the required features in it. 68000 by Motorola was one of the first microprocessors to develop the concept of microcoding in its instruction set. They were further developed to 32 bit architectures. Similarly, many players like Zilog, IBM and Apple were successful in getting their own products in the market. However, Intel had a commanding position in the market right through the microprocessorera.

The 1990s saw a large scale application of microprocessors in the personal computer applications developed by the newly formed Apple, IBM and Microsoft corporation. It witnessed a revolution in the use of computers, which by then was a household entity.

This growth was complemented by a highly sophisticated development in the commercial use of microprocessors. In 1993, Intel brought out its 'Pentium Processor' which is one of the most popular processors in use till date. It was followed by a series of excellent processors of the Pentium family, leading into the 21st century. The latest one in commercial use is the Pentium Dual Core technology and the Xeon processor. They have opened up a whole new world of diverse applications. Supercomputers have become common, owing to this amazing development in microprocessors.

INTRODUCTION TO MICROPROCESSOR AND MICROCOMPUTER ARCHITECTURE:

A *microprocessor* is a programmable electronics chip that has computing and decision making capabilities similar to central processing unit of a computer. Any microprocessor-based systems having limited number of resources are called *microcomputers*. Nowadays, microprocessor can be seen in almost all types of electronics devices like mobile phones, printers, washing machines etc. Microprocessors are also used in advanced applications like radars, satellites and flights. Due to the rapid advancements in electronic industry and large scale integration of devices results in a significant cost reduction and increase application of microprocessors and their derivatives.

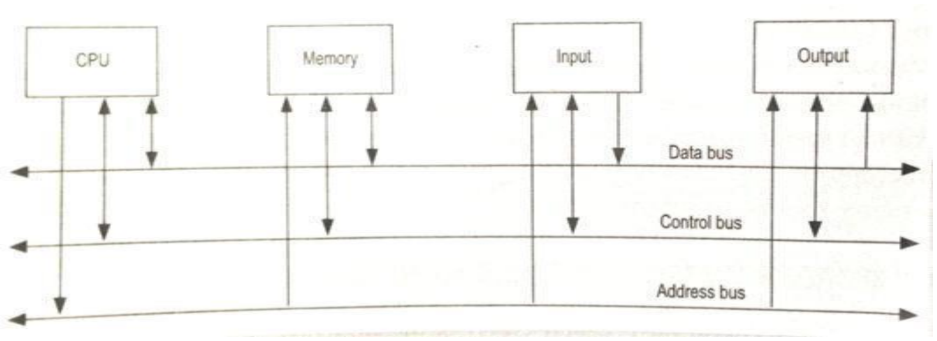


Fig.1 Microprocessor-based system

- **Bit**: A bit is a single binary digit.
- **Word**: A word refers to the basic data size or bit size that can be processed by the arithmetic and logic unit of the processor. A 16-bit binary number is called a word in a 16-bit processor.
- **Bus**: A bus is a group of wires/lines that carry similar information.
- **System Bus**: The system bus is a group of wires/lines used for communication between the microprocessor and peripherals.
- **Memory Word**: The number of bits that can be stored in a register or memory element is called a memory word.
- **Address Bus**: It carries the address, which is a unique binary pattern used to identify a memory location or an I/O port. For example, an eight bit address bus has eight lines and thus it can address 2^8 = 256 different locations. The locations in hexadecimal format can be written as 00H – FFH.
- **Data Bus**: The data bus is used to transfer data between memory and processor or between I/O device and processor. For example, an 8-bit processor will generally have an 8-bit data bus and a 16-bit processor will have 16-bit data bus.
- **Control Bus**: The control bus carry control signals, which consists of signals for selection of memory or I/O device from the given address, direction of data transfer and synchronization of data transfer in case of slow devices.

A typical microprocessor consists of arithmetic and logic unit (ALU) in association with control unit to process the instruction execution. Almost all the microprocessors are based on the principle of store-program concept. In *store- program concept*, programs or instructions are sequentially stored in the memory locations that are to be executed. To do any task using a microprocessor, it is to be programmed by the user. So the programmer must have idea about its internal resources, features and supported instructions. Each microprocessor has a set of instructions, a list which is provided by the microprocessor manufacturer. The instruction set of a microprocessor is provided in two forms: *binary machine code and mnemonics*.

Microprocessor communicates and operates in binary numbers 0 and 1. The set of instructions in the form of binary patterns is called a *machine language* and it is difficult for us to understand. Therefore, the binary patterns are given abbreviated names, called mnemonics, which forms the *assembly language*. The conversion of assembly-level language into binary machine-level language is done by using an application called *assembler.*

<u>Technology Used</u>:

The semiconductor manufacturing technologies used for chips are:

- Transistor-Transistor Logic (TTL)
- Emitter Coupled Logic (ECL)
- Complementary Metal-Oxide

Semiconductor (CMOS)<u> Classification of</u>

<u>Microprocessors</u>:

Based on their specification, application and architecture microprocessors are classified.

Based on size of data bus:

- 4-bit microprocessor
- 8-bit microprocessor
- 16-bit microprocessor
- 32-bit microprocessor

Based on application:

- General-purpose microprocessor- used in general computer system and can be used by programmer for any application. Examples, 8085 to Intel Pentium.
- Microcontroller- microprocessor with built-in memory and ports and can be programmed for any generic control application. Example, 8051.
- Special-purpose processors- designed to handle special functions required for an application. Examples, digital signal processors and application-specific integrated circuit (ASIC) chips.

Based on architecture:

- Reduced Instruction Set Computer (RISC) processors
- Complex Instruction Set Computer (CISC) processors

2. 8085 MICROPROCESSOR ARCHITECTURE

The 8085 microprocessor is an 8-bit processor available as a 40-pin IC package and uses +5 V for power. It can run at a maximum frequency of 3 MHz. Its data bus width is 8-bit and address bus width is 16-bit, thus it can address 2^{16} = 64 KB of memory. The internal architecture of 8085 is shown is Fig. 2.

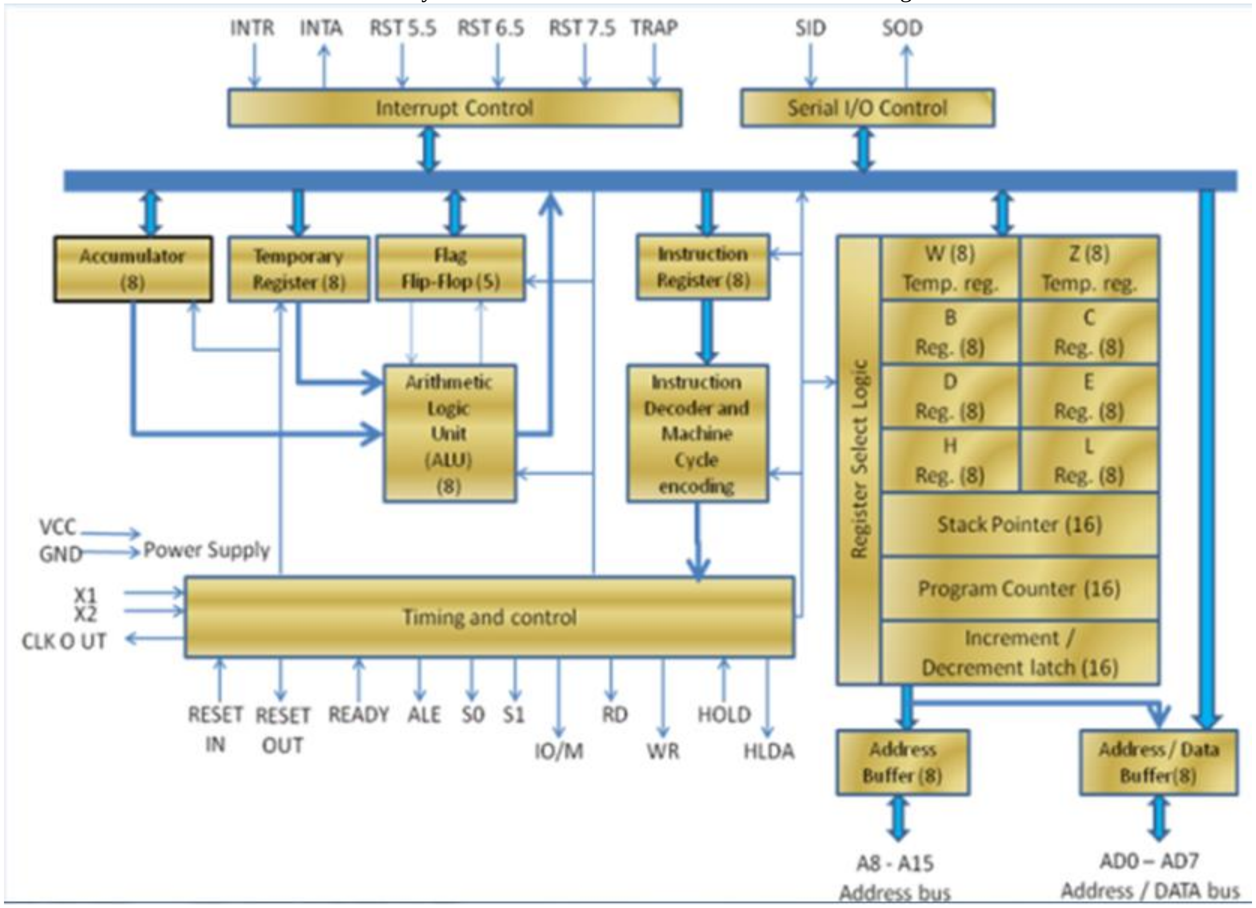


Fig. 2 Internal Architecture of 8085

Arithmetic and Logic Unit

The ALU performs the actual numerical and logical operations such as Addition (ADD), Subtraction (SUB), AND, OR etc. It uses data from memory and from Accumulator to perform operations. The results of the arithmetic and logical operations are stored in the accumulator.

Registers

The 8085 includes six registers, one accumulator and one flag register, as shown in Fig. 3. In addition, it has two 16-bit registers: stack pointer and program counter. They are briefly described as follows.

The 8085 has six general-purpose registers to store 8-bit data; these are identified as B, C, D, E, H and L. they can be combined as register pairs - BC, DE and HL to perform some

16- bit operations. The programmer can use these registers to store or copy data into the register by using data copy instructions.

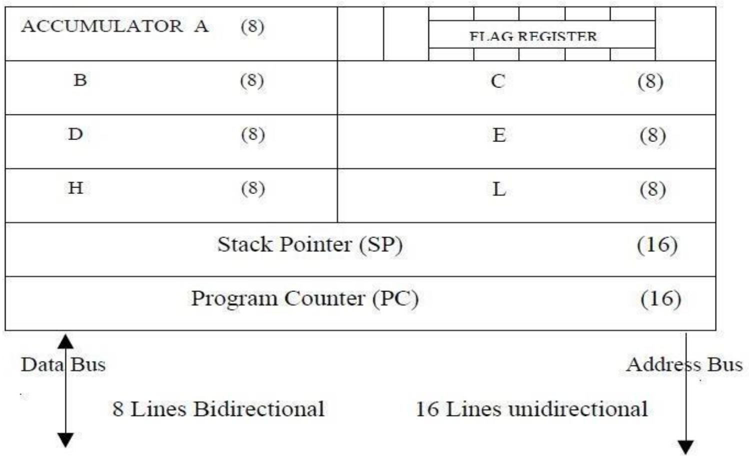


Fig. 3 Register organisation

Accumulator

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

Flag register

The ALU includes five flip-flops, which are set or reset after an operation according to data condition of the result in the accumulator and other registers. They are called Zero (Z), Carry (CY), Sign (S), Parity (P) and Auxiliary Carry (AC) flags. Their bit positions in the flag register are shown in Fig. 4. The microprocessor uses these flags to test data conditions.

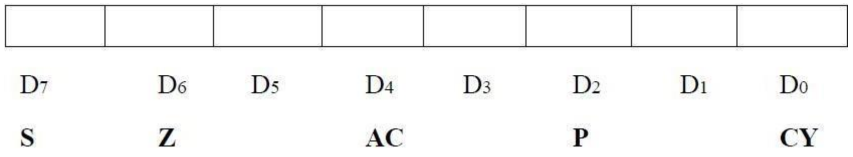


Fig. 4 Flag register

For example, after an addition of two numbers, if the result in the accumulator is larger than 8-bit, the flip-flop uses to indicate a carry by setting CY flag to 1. When an arithmetic operation results in zero, Z flag is set to 1. The S flag is just a copy of the bit D7 of the accumulator. A negative number has a 1 in bit D7 and a positive number has a 0 in 2’s complement representation. The AC flag is set to 1, when a carry result from bit D3 and passes to bit D4. The P flag is set to 1, when the result in accumulator contains even number of 1s.

Program Counter (PC)

This 16-bit register deals with sequencing the execution of instructions. This register is a memory pointer. The microprocessor uses this register to sequence the execution of the instructions. The function of the program counter is to point to the memory address from which the next byte is to be fetched. When a byte is being fetched, the program counter is automatically incremented by one to point to the next memory location.

Stack Pointer (SP)

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

Instruction Register/Decoder

It is an 8-bit register that temporarily stores the current instruction of a program. Latest instruction sent here from memory prior to execution. Decoder then takes instruction and decodes or interprets the instruction. Decoded instruction then passed to next stage.

Control Unit

Generates signals on data bus, address bus and control bus within microprocessor to carry out the instruction, which has been decoded. Typical buses and their timing are described as follows:

- *Data Bus*: Data bus carries data in binary form between microprocessor and other external units such as memory. It is used to transmit data i.e. information, results of

  arithmetic etc between memory and the microprocessor. Data bus is bidirectional in nature. The data bus width of 8085 microprocessor is 8-bit i.e. 2^8 combination of binary digits and are typically identified as D0 – D7. Thus size of the data bus determines what arithmetic can be done. If only 8-bit wide then largest number is 11111111 (255 in decimal). Therefore, larger numbers have to be broken down into chunks of 255. This slows microprocessor.
- *Address Bus*: The address bus carries addresses and is one way bus from microprocessor to the memory or other devices. 8085 microprocessor contain 16-bit address bus and are generally identified as A0 - A15. The higher order address lines (A8 – A15) are unidirectional and the lower order lines (A0 – A7) are multiplexed (time-shared) with the eight data bits (D0 – D7) and hence, they are bidirectional.
- *Control Bus*: Control bus are various lines which have specific functions for coordinating and controlling microprocessor operations. The control bus carries control signals partly unidirectional and partly bidirectional. The following control and status signals are used by 8085 processor:
  - I. ALE (output): Address Latch Enable is a pulse that is provided when an address appears on the AD0 – AD7 lines, after which it becomes 0.

II. $\overline{RD}$ (active low output): The Read signal indicates that data are being read from the selected I/O or memory device and that they are available on the data bus.

III. $\overline{WR}$ (active low output): The Write signal indicates that data on the data bus are to be written into a selected memory or I/O location.

IV. IO/$\overline{M}$ (output): It is a signal that distinguished between a memory operation and an I/O operation. When IO/$\overline{M}$ = 0 it is a memory operation and IO/$\overline{M}$ = 1 it is an I/O operation.

V. S1 and S0 (output): These are status signals used to specify the type of operation being performed; they are listed in Table 1.

Table 1 Status signals and associated operations

S1	S0	States
0	0	Halt
0	1	Write
1	0	Read
1	1	Fetch

The schematic representation of the 8085 bus structure is as shown in Fig. 5. The microprocessor performs primarily four operations:

1. Memory Read: Reads data (or instruction) from memory.
2. Memory Write: Writes data (or instruction) into memory.
3. I/O Read: Accepts data from input device.
4. I/O Write: Sends data to output device.

The 8085 processor performs these functions using address bus, data bus and control bus as shown in Fig. 5.

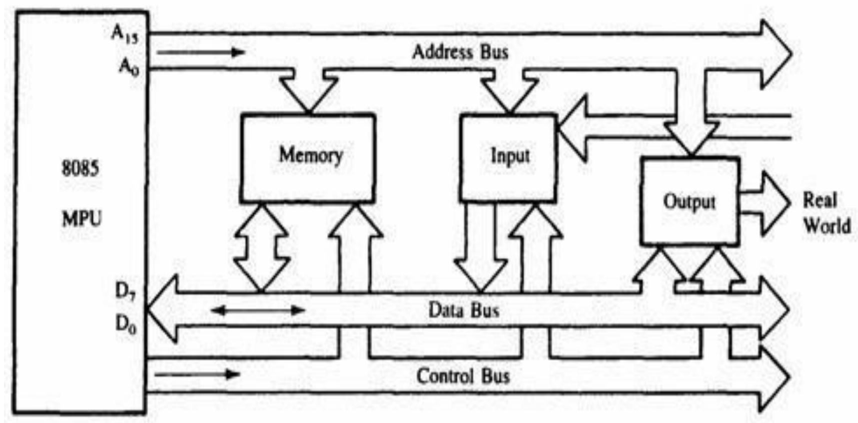


Fig. 5 The 8085 bus structure

3. 8085 PIN DESCRIPTION

Properties:

- It is a 8-bit microprocessor
- Manufactured with N-MOS technology
- 40 pin IC package
- It has 16-bit address bus and thus has 2^{16} = 64 KB addressing capability.
- Operate with 3 MHz single-phase clock
- +5 V single power supply

The logic pin layout and signal groups of the 8085nmicroprocessor are shown in Fig. 6. All the signals are classified into six groups:

- Address bus
- Data bus
- Control & status signals
- Power supply and frequency signals
- Externally initiated signals
- Serial I/O signals

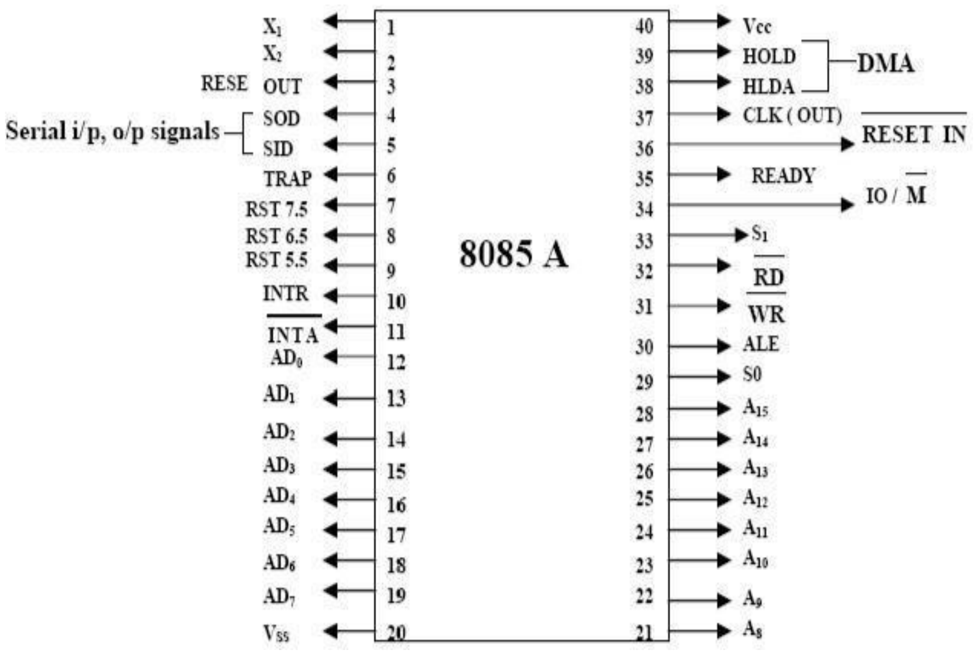


Fig. 6 8085 microprocessor pin layout and signal groups

Address and Data Buses:

- A8 – A15 (output, 3-state): Most significant eight bits of memory addresses and the eight bits of the I/O addresses. These lines enter into tri-state high impedance state during HOLD and HALT modes.
- AD0 – AD7 (input/output, 3-state): Lower significant bits of memory addresses and the eight bits of the I/O addresses during first clock cycle. Behaves as data bus

during third and fourth clock cycle. These lines enter into tri-state high impedance state during HOLD and HALT modes.

Control & Status Signals:

- ALE: Address latch enable
- RD : Read control signal.
- WR : Write control signal.
- IO/M , S1 and S0 : Status signals. Power

Supply & Clock Frequency:

- Vcc: +5 V power supply
- Vss: Ground reference
- X1, X2: A crystal having frequency of 6 MHz is connected at these two pins
- CLK: Clock output

Externally Initiated and Interrupt Signals:

- RESET IN : When the signal on this pin is low, the PC is set to 0, the buses are tri-stated and the processor is reset.
- RESET OUT: This signal indicates that the processor is being reset. The signal can be used to reset other devices.
- READY: When this signal is low, the processor waits for an integral number of clock cycles until it goes high.
- HOLD: This signal indicates that a peripheral like DMA (direct memory access) controller is requesting the use of address and data bus.
- HLDA: This signal acknowledges the HOLD request.
- INTR: Interrupt request is a general-purpose interrupt.
- INTA : This is used to acknowledge an interrupt.
- RST 7.5, RST 6.5, RST 5,5 – restart interrupt: These are vectored interrupts and have highest priority than INTR interrupt.
- TRAP: This is a non-maskable interrupt and has the highest priority.

Serial I/O Signals:

- SID: Serial input signal. Bit on this line is loaded to D7 bit of register A using RIM instruction.
- SOD: Serial output signal. Output SOD is set or reset by using SIM instruction.

4. INSTRUCTION SET AND EXECUTION IN 8085

Based on the design of the ALU provides and decoding unit, the microprocessor manufacturer microprocessor. The instruction set for every machine code and instruction set consists of both mnemonics.

An instruction is a binary pattern designed inside a microprocessor to perform a specific function. The entire group of instructions that a microprocessor supports is called instruction set. Microprocessor instructions can be classified based on the parameters such functionality, length and operand addressing.

Classification based on functionality:

I. Data transfer operations: This group of instructions copies data from source to destination. The content of the source is not altered.
II. Arithmetic operations: Instructions of this group perform operations like addition, subtraction, increment & decrement. One of the data used in arithmetic operation is stored in accumulator and the result is also stored in accumulator.
III. Logical operations: Logical operations include AND, OR, EXOR, NOT. The operations like AND, OR and EXOR uses two operands, one is stored in accumulator and other can be any register or memory location. The result is stored in accumulator. NOT operation requires single operand, which is stored in accumulator.
IV. Branching operations: Instructions in this group can be used to transfer program sequence from one memory location to another either conditionally or unconditionally.
V. Machine control operations: Instruction in this group control execution of other instructions and control operations like interrupt, halt etc.

Classification based on length:

I. One-byte instructions: Instruction having one byte in machine code. Examples are depicted in Table 2.
I. Two-byte instructions: Instruction having two byte in machine code. Examples are depicted in Table 3
II. Three-byte instructions: Instruction having three byte in machine code. Examples are depicted in Table 4.

Table 2 Examples of one byte instructions

Opcode	Operand	Machine code/Hex code
MOV	A, B	78
ADD	M	86

Table 3 Examples of two byte instructions

Opcode	Operand	Machine code/Hex code	Byte description
MVI	A, 7FH	3E	First byte
		7F	Second byte
ADI	0FH	C6	First byte
		0F	Second byte

Table 4 Examples of three byte instructions

Opcode	Operand	Machine code/Hex code	Byte description
JMP	9050H	C3	First byte
		50	Second byte
		90	Third byte
LDA	8850H	3A	First byte
		50	Second byte
		88	Third byte

Addressing Modes in Instructions:

The process of specifying the data to be operated on by the instruction is called addressing. The various formats for specifying operands are called addressing modes. The 8085 has the following five types of addressing:

1. Immediate addressing
2. Memory direct addressing
3. Register direct addressing
4. Indirect addressing
5. Implicit addressing

Immediate Addressing:

In this mode, the operand given in the instruction - a byte or word – transfers to the destination register or memory location.

Ex: MVI A, 9AH

- The operand is a part of the instruction.
- The operand is stored in the register mentioned in the instruction.

Memory Direct Addressing:

Memory direct addressing moves a byte or word between a memory location and register. The memory location address is given in the instruction.

Ex: LDA 850FH

This instruction is used to load the content of memory address 850FH in the accumulator.

Register Direct Addressing:

Register direct addressing transfer a copy of a byte or word from source register to destination register.

Ex: MOV B, C

It copies the content of register C to register B.

Indirect Addressing:

Indirect addressing transfers a byte or word between a register and a memory location.

Ex: MOV A, M

Here the data is in the memory location pointed to by the contents of HL pair. The data is moved to the accumulator.

Implicit Addressing

In this addressing mode the data itself specifies the data to be operated upon.

Ex: CMA

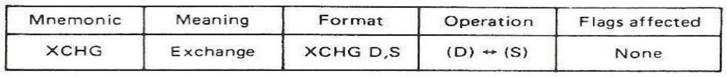
The instruction complements the content of the accumulator. No specific data or operand is mentioned in the instruction.

5. INSTRUCTION SET OF 8085

Data Transfer Instructions:

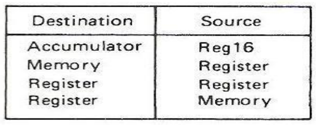
MOV instruction

XCHG Instruct



Mnemonic	Meaning	Format	Operation	Flags affected
XCHG	Exchange	XCHG D,S	(D) ↔ (S)	None

(a)



Destination	Source
Accumulator	Reg16
Memory	Register
Register	Register
Register	Memory

(b)

XLAT

Mnemonic	Meaning	Format	Operation	Flags affected
XLAT	Translate	XLAT	((AL) + (BX) + (DS) *10) AL	none

LEA, LDS, and LES instructions

LEA: Load effective Address, LEA Reg 16, EA
LDS: Load register and DS, LDS Reg 16, EA
LES: Load register and ES, LES Reg 16, EA

Store accumulator direct		
STA	16-bit address	The contents of the accumulator are copied into the memory location specified by the operand. This is a 3-byte instruction, the second byte specifies the low-order address and the third byte specifies the high-order address. Example: STA 4350 or STA XYZ
Store accumulator indirect		
STAX	Reg. pair	The contents of the accumulator are copied into the memory location specified by the contents of the operand (register pair). The contents of the accumulator are not altered. Example: STAX B
Store H and L registers direct		
SHLD	16-bit address	The contents of register L are stored into the memory location specified by the 16-bit address in the operand and the contents of H register are stored into the next memory location by incrementing the operand. The contents of registers HL are not altered. This is a 3-byte instruction, the second byte specifies the low-order address and the third byte specifies the high-order address. Example: SHLD 2470
Exchange H and L with D and E		
XCHG	none	The contents of register H are exchanged with the contents of register D, and the contents of register L are exchanged with the contents of register E. Example: XCHG
Copy H and L registers to the stack pointer		
SPHL	none	The instruction loads the contents of the H and L registers into the stack pointer register, the contents of the H register provide the high-order address and the contents of the L register provide the low-order address. The contents of the H and L registers are not altered. Example: SPHL
Exchange H and L with top of stack		
XTHL	none	The contents of the L register are exchanged with the stack location pointed out by the contents of the stack pointer register. The contents of the H register are exchanged with the next stack location (SP+1); however, the contents of the stack pointer register are not altered. Example: XTHL

Push register pair onto stack		
PUSH	Reg. pair	The contents of the register pair designated in the operand are copied onto the stack in the following sequence. The stack pointer register is decremented and the contents of the high-order register (B, D, H, A) are copied into that location. The stack pointer register is decremented again and the contents of the low-order register (C, E, L, flags) are copied to that location. Example: PUSH B or PUSH A
Pop off stack to register pair		
POP	Reg. pair	The contents of the memory location pointed out by the stack pointer register are copied to the low-order register (C, E, L, status flags) of the operand. The stack pointer is incremented by 1 and the contents of that memory location are copied to the high-order register (B, D, H, A) of the operand. The stack pointer register is again incremented by 1. Example: POP H or POP A
Output data from accumulator to a port with 8-bit address		
OUT	8-bit port address	The contents of the accumulator are copied into the I/O port specified by the operand. Example: OUT 87
Input data to accumulator from a port with 8-bit address		
IN	8-bit port address	The contents of the input port designated in the operand are read and loaded into the accumulator. Example: IN 82

Arithmetic Instructions:

Opcode	Operand	Description
Add register or memory to accumulator		
ADD	R M	The contents of the operand (register or memory) are added to the contents of the accumulator and the result is stored in the accumulator. If the operand is a memory location, its location is specified by the contents of the HL registers. All flags are modified to reflect the result of the addition. Example: ADD B or ADD M
Add register to accumulator with carry		
ADC	R M	The contents of the operand (register or memory) and the Carry flag are added to the contents of the accumulator and the result is stored in the accumulator. If the operand is a memory location, its location is specified by the contents of the HL registers. All flags are modified to reflect the result of the addition. Example: ADC B or ADC M
Add immediate to accumulator		
ADI	8-bit data	The 8-bit data (operand) is added to the contents of the accumulator and the result is stored in the accumulator. All flags are modified to reflect the result of the addition. Example: ADI 45
Add immediate to accumulator with carry		
ACI	8-bit data	The 8-bit data (operand) and the Carry flag are added to the contents of the accumulator and the result is stored in the accumulator. All flags are modified to reflect the result of the addition. Example: ACI 45
Add register pair to H and L registers		
DAD	Reg. pair	The 16-bit contents of the specified register pair are added to the contents of the HL register and the sum is stored in the HL register. The contents of the source register pair are not altered. If the result is larger than 16 bits, the CY flag is set. No other flags are affected. Example: DAD H

Subtract register or memory from accumulator

SUB	R M	The contents of the operand (register or memory) are subtracted from the contents of the accumulator, and the result is stored in the accumulator. If the operand is a memory location, its location is specified by the contents of the HL registers. All flags are modified to reflect the result of the subtraction. Example: SUB B or SUB M

Subtract source and borrow from accumulator

SBB	R M	The contents of the operand (register or memory) and the Borrow flag are subtracted from the contents of the accumulator and the result is placed in the accumulator. If the operand is a memory location, its location is specified by the contents of the HL registers. All flags are modified to reflect the result of the subtraction. Example: SBB B or SBB M

Subtract immediate from accumulator

SUI	8-bit data	The 8-bit data (operand) is subtracted from the contents of the accumulator and the result is stored in the accumulator. All flags are modified to reflect the result of the subtraction. Example: SUI 45

Subtract immediate from accumulator with borrow

SBI	8-bit data	The 8-bit data (operand) and the Borrow flag are subtracted from the contents of the accumulator and the result is stored in the accumulator. All flags are modified to reflect the result of the subtracion. Example: SBI 45

Increment register or memory by 1

INR	R M	The contents of the designated register or memory) are incremented by 1 and the result is stored in the same place. If the operand is a memory location, its location is specified by the contents of the HL registers. Example: INR B or INR M

Increment register pair by 1

INX	R	The contents of the designated register pair are incremented by 1 and the result is stored in the same place. Example: INX H

Decrement register or memory by 1		
DCR	R M	The contents of the designated register or memory are decremented by 1 and the result is stored in the same place. If the operand is a memory location, its location is specified by the contents of the HL registers. Example: DCR B or DCR M
Decrement register pair by 1		
DCX	R	The contents of the designated register pair are decremented by 1 and the result is stored in the same place. Example: DCX H
Decimal adjust accumulator		
DAA	none	The contents of the accumulator are changed from a binary value to two 4-bit binary coded decimal (BCD) digits. This is the only instruction that uses the auxiliary flag to perform the binary to BCD conversion, and the conversion procedure is described below. S, Z, AC, P, CY flags are altered to reflect the results of the operation. If the value of the low-order 4-bits in the accumulator is greater than 9 or if AC flag is set, the instruction adds 6 to the low-order four bits. If the value of the high-order 4-bits in the accumulator is greater than 9 or if the Carry flag is set, the instruction adds 6 to the high-order four bits. Example: DAA

BRANCHING INSTRUCTIONS

Opcode	Operand	Description
Jump unconditionally		
JMP	16-bit address	The program sequence is transferred to the memory location specified by the 16-bit address given in the operand. Example: JMP 2034 or JMP XYZ

Jump conditionally

Operand: 16-bit address

The program sequence is transferred to the memory location specified by the 16-bit address given in the operand based on the specified flag of the PSW as described below.
Example: JZ 2034 or JZ XYZ

Opcode	Description	Flag Status
JC	Jump on Carry	CY = 1
JNC	Jump on no Carry	CY = 0
JP	Jump on positive	S = 0
JM	Jump on minus	S = 1
JZ	Jump on zero	Z = 1
JNZ	Jump on no zero	Z = 0
JPE	Jump on parity even	P = 1
JPO	Jump on parity odd	P = 0

Unconditional subroutine call

CALL 16-bit address

The program sequence is transferred to the memory location specified by the 16-bit address given in the operand. Before the transfer, the address of the next instruction after CALL (the contents of the program counter) is pushed onto the stack.
Example: CALL 2034 or CALL XYZ

Call conditionally

Operand: 16-bit address

The program sequence is transferred to the memory location specified by the 16-bit address given in the operand based on the specified flag of the PSW as described below. Before the transfer, the address of the next instruction after the call (the contents of the program counter) is pushed onto the stack.
Example: CZ 2034 or CZ XYZ

Opcode	Description	Flag Status
CC	Call on Carry	CY = 1
CNC	Call on no Carry	CY = 0
CP	Call on positive	S = 0
CM	Call on minus	S = 1
CZ	Call on zero	Z = 1
CNZ	Call on no zero	Z = 0
CPE	Call on parity even	P = 1
CPO	Call on parity odd	P = 0

Return from subroutine unconditionally

RET none

The program sequence is transferred from the subroutine to the calling program. The two bytes from the top of the stack are copied into the program counter, and program execution begins at the new address.
Example: RET

Return from subroutine conditionally

Operand: none

The program sequence is transferred from the subroutine to the calling program based on the specified flag of the PSW as described below. The two bytes from the top of the stack are copied into the program counter, and program execution begins at the new address.
Example: RZ

Opcode	Description	Flag Status
RC	Return on Carry	CY = 1
RNC	Return on no Carry	CY = 0
RP	Return on positive	S = 0
RM	Return on minus	S = 1
RZ	Return on zero	Z = 1
RNZ	Return on no zero	Z = 0
RPE	Return on parity even	P = 1
RPO	Return on parity odd	P = 0

Load program counter with HL contents		
PCHL	none	The contents of registers H and L are copied into the program counter. The contents of H are placed as the high-order byte and the contents of L as the low-order byte. Example: PCHL
Restart		
RST	0-7	The RST instruction is equivalent to a 1-byte call instruction to one of eight memory locations depending upon the number. The instructions are generally used in conjunction with interrupts and inserted using external hardware. However these can be used as software instructions in a program to transfer program execution to one of the eight locations. The addresses are:

Instruction	Restart Address
RST 0	0000H
RST 1	0008H
RST 2	0010H
RST 3	0018H
RST 4	0020H
RST 5	0028H
RST 6	0030H
RST 7	0038H

The 8085 has four additional interrupts and these interrupts generate RST instructions internally and thus do not require any external hardware. These instructions and their Restart addresses are:

Interrupt	Restart Address
TRAP	0024H
RST 5.5	002CH
RST 6.5	0034H
RST 7.5	003CH

LOGICAL INSTRUCTIONS

Opcode	Operand	Description
Compare register or memory with accumulator		
CMP	R M	The contents of the operand (register or memory) are compared with the contents of the accumulator. Both contents are preserved . The result of the comparison is shown by setting the flags of the PSW as follows: if (A) < (reg/mem): carry flag is set, s=1 if (A) = (reg/mem): zero flag is set, s=0 if (A) > (reg/mem): carry and zero flags are reset, s=0 Example: CMP B or CMP M
Compare immediate with accumulator		
CPI	8-bit data	The second byte (8-bit data) is compared with the contents of the accumulator. The values being compared remain unchanged. The result of the comparison is shown by setting the flags of the PSW as follows: if (A) < data: carry flag is set, s=1 if (A) = data: zero flag is set, s=0 if (A) > data: carry and zero flags are reset, s=0 Example: CPI 89
Logical AND register or memory with accumulator		
ANA	R M	The contents of the accumulator are logically ANDed with the contents of the operand (register or memory), and the result is placed in the accumulator. If the operand is a memory location, its address is specified by the contents of HL registers. S, Z, P are modified to reflect the result of the operation. CY is reset. AC is set. Example: ANA B or ANA M
Logical AND immediate with accumulator		
ANI	8-bit data	The contents of the accumulator are logically ANDed with the 8-bit data (operand) and the result is placed in the accumulator. S, Z, P are modified to reflect the result of the operation. CY is reset. AC is set. Example: ANI 86

Exclusive OR register or memory with accumulator

XRA	R M	The contents of the accumulator are Exclusive ORed with the contents of the operand (register or memory), and the result is placed in the accumulator. If the operand is a memory location, its address is specified by the contents of HL registers. S, Z, P are modified to reflect the result of the operation. CY and AC are reset. Example: XRA B or XRA M

Exclusive OR immediate with accumulator

XRI	8-bit data	The contents of the accumulator are Exclusive ORed with the 8-bit data (operand) and the result is placed in the accumulator. S, Z, P are modified to reflect the result of the operation. CY and AC are reset. Example: XRI 86

Logical OR register or memory with accumulaotr

ORA	R M	The contents of the accumulator are logically ORed with the contents of the operand (register or memory), and the result is placed in the accumulator. If the operand is a memory location, its address is specified by the contents of HL registers. S, Z, P are modified to reflect the result of the operation. CY and AC are reset. Example: ORA B or ORA M

Logical OR immediate with accumulator

ORI	8-bit data	The contents of the accumulator are logically ORed with the 8-bit data (operand) and the result is placed in the accumulator. S, Z, P are modified to reflect the result of the operation. CY and AC are reset. Example: ORI 86

Rotate accumulator left

RLC	none	Each binary bit of the accumulator is rotated left by one position. Bit D_7 is placed in the position of D_0 as well as in the Carry flag. CY is modified according to bit D_7. S, Z, P, AC are not affected. Example: RLC

Rotate accumulator right

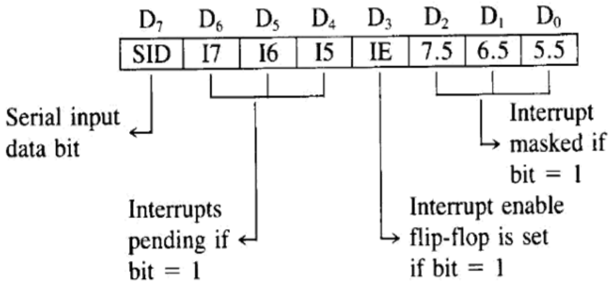
RRC	none	Each binary bit of the accumulator is rotated right by one position. Bit D_0 is placed in the position of D_7 as well as in the Carry flag. CY is modified according to bit D_0. S, Z, P, AC are not affected. Example: RRC

Rotate accumulator left through carry		
RAL	none	Each binary bit of the accumulator is rotated left by one position through the Carry flag. Bit D_7 is placed in the Carry flag, and the Carry flag is placed in the least significant position D_0. CY is modified according to bit D_7. S, Z, P, AC are not affected. Example: RAL
Rotate accumulator right through carry		
RAR	none	Each binary bit of the accumulator is rotated right by one position through the Carry flag. Bit D_0 is placed in the Carry flag, and the Carry flag is placed in the most significant position D_7. CY is modified according to bit D_0. S, Z, P, AC are not affected. Example: RAR
Complement accumulator		
CMA	none	The contents of the accumulator are complemented. No flags are affected. Example: CMA
Complement carry		
CMC	none	The Carry flag is complemented. No other flags are affected. Example: CMC
Set Carry		
STC	none	The Carry flag is set to 1. No other flags are affected. Example: STC

CONTROL INSTRUCTIONS

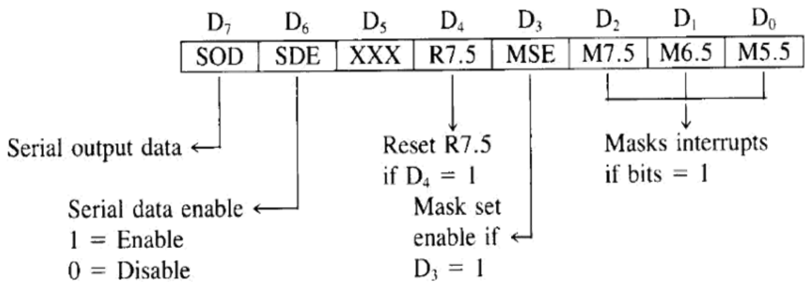
Opcode	Operand	Description
No operation		
NOP	none	No operation is performed. The instruction is fetched and decoded. However no operation is executed. Example: NOP
Halt and enter wait state		
HLT	none	The CPU finishes executing the current instruction and halts any further execution. An interrupt or reset is necessary to exit from the halt state. Example: HLT
Disable interrupts		
DI	none	The interrupt enable flip-flop is reset and all the interrupts except the TRAP are disabled. No flags are affected. Example: DI
Enable interrupts		
EI	none	The interrupt enable flip-flop is set and all interrupts are enabled. No flags are affected. After a system reset or the acknowledgement of an interrupt, the interrupt enable flip-flop is reset, thus disabling the interrupts. This instruction is necessary to reenable the interrupts (except TRAP). Example: EI

Read interrupt mask RIM none	This is a multipurpose instruction used to read the status of interrupts 7.5, 6.5, 5.5 and read serial data input bit. The instruction loads eight bits in the accumulator with the following interpretations. Example: RIM



D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0
SID I7 I6 I5 IE 7.5 6.5 5.5
Serial input data bit
Interrupts pending if bit = 1
Interrupt enable flip-flop is set if bit = 1
Interrupt masked if bit = 1

Set interrupt mask SIM none	This is a multipurpose instruction and used to implement the 8085 interrupts 7.5, 6.5, 5.5, and serial data output. The instruction interprets the accumulator contents as follows. Example: SIM



D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0
SOD SDE XXX R7.5 MSE M7.5 M6.5 M5.5
Serial output data
Serial data enable 1 = Enable 0 = Disable
Reset R7.5 if $D_4 = 1$
Mask set enable if $D_3 = 1$
Masks interrupts if bits = 1

- ☐ SOD — Serial Output Data: Bit D_7 of the accumulator is latched into the SOD output line and made available to a serial peripheral if bit $D_6 = 1$.
- ☐ SDE — Serial Data Enable: If this bit = 1, it enables the serial output. To implement serial output, this bit needs to be enabled.
- ☐ XXX — Don't Care
- ☐ R7.5 — Reset RST 7.5: If this bit = 1, RST 7.5 flip-flop is reset. This is an additional control to reset RST 7.5.
- ☐ MSE — Mask Set Enable: If this bit is high, it enables the functions of bits D_2, D_1, D_0. This is a master control over all the interrupt masking bits. If this bit is low, bits D_2, D_1, and D_0 do not have any effect on the masks.
- ☐ M7.5 — D_2 = 0, RST 7.5 is enabled.
  = 1, RST 7.5 is masked or disabled.
- ☐ M6.5 — D_1 = 0, RST 6.5 is enabled.
  = 1, RST 6.5 is masked or disabled.
- ☐ M5.5 — D_0 = 0, RST 5.5 is enabled.
  = 1, RST 5.5 is masked or disabled.

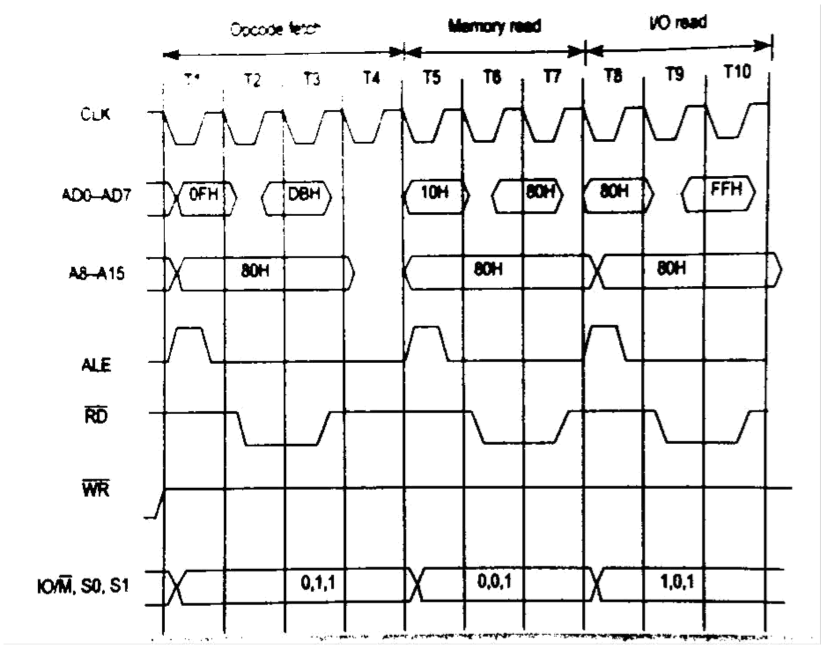


Fig. 12 Timing diagram for the IN instruction

7. 8085 INTERRUPTS

Interrupt Structure:

Interrupt is the mechanism by which the processor is made to transfer control from its current program execution to another program having higher priority. The interrupt signal may be given to the processor by any external peripheral device.

The program or the routine that is executed upon interrupt is called interrupt service routine (ISR). After execution of ISR, the processor must return to the interrupted program. Key features in the interrupt structure of any microprocessor are as follows:

i. Number and types of interrupt signals available.
ii. The address of the memory where the ISR is located for a particular interrupt signal. This address is called interrupt vector address (IVA).
iii. Masking and unmasking feature of the interrupt signals.
iv. Priority among the interrupts.
v. Timing of the interrupt signals.
vi. Handling and storing of information about the interrupt program (status information).

Types of Interrupts:

Interrupts are classified based on their maskability, IVA and source. They are classified as:

i. Vectored and Non-Vectored Interrupts
   - Vectored interrupts require the IVA to be supplied by the external device that gives the interrupt signal. This technique is vectoring, is implemented in number of ways.
   - Non-vectored interrupts have fixed IVA for ISRs of different interrupt signals.

ii. Maskable and Non-Maskable Interrupts
   - Maskable interrupts are interrupts that can be blocked. Masking can be done by software or hardware means.
   - Non-maskable interrupts are interrupts that are always recognized; the corresponding ISRs are executed.

iii. Software and Hardware Interrupts
   - Software interrupts are special instructions, after execution transfer the control to predefined ISR.
   - Hardware interrupts are signals given to the processor, for recognition as an interrupt and execution of the corresponding ISR.

Interrupt Handling Procedure:

The following sequence of operations takes place when an interrupt signal is recognized:

i. Save the PC content and information about current state (flags, registers etc) in the stack.
ii. Load PC with the beginning address of an ISR and start to execute it.
iii. Finish ISR when the return instruction is executed.
iv. Return to the point in the interrupted program where execution was interrupted.

Interrupt Sources and Vector Addresses in 8085:

Software Interrupts:

8085 instruction set includes eight software interrupt instructions called Restart (RST) instructions. These are one byte instructions that make the processor execute a subroutine at predefined locations. Instructions and their vector addresses are given in Table 6.

Table 6 Software interrupts and their vector addresses

Instruction	Machine hex code	Interrupt Vector Address
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	0032H

The software interrupts can be treated as CALL instructions with default call locations. The concept of priority does not apply to software interrupts as they are inserted into the program as instructions by the programmer and executed by the processor when the respective program lines are read.

Hardware Interrupts and Priorities:

8085 have five hardware interrupts – INTR, RST 5.5, RST 6.5, RST 7.5 and TRAP. Their IVA and priorities are given in Table 7.

Table 7 Hardware interrupts of 8085

Interrupt	Interrupt vector address	Maskable or non-maskable	Edge or level Triggered	priority
TRAP	0024H	Non-makable	Level	1
RST 7.5	003CH	Maskable	Rising edge	2
RST 6.5	0034H	Maskable	Level	3
RST 5.5	002CH	Maskable	Level	4
INTR	Decided by hardware	Maskable	Level	5

Masking of Interrupts:

Masking can be done for four hardware interrupts INTR, RST 5.5, RST 6.5, and RST 7.5. The masking of 8085 interrupts is done at different levels. Fig. 13 shows the organization of hardware interrupts in the 8085.

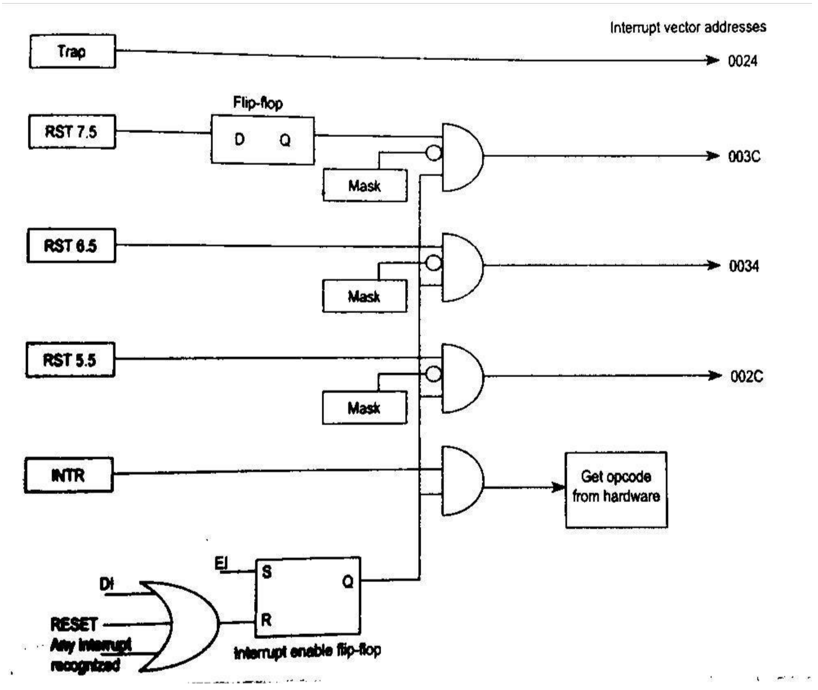


Fig. 13 Interrupt structure of 8085

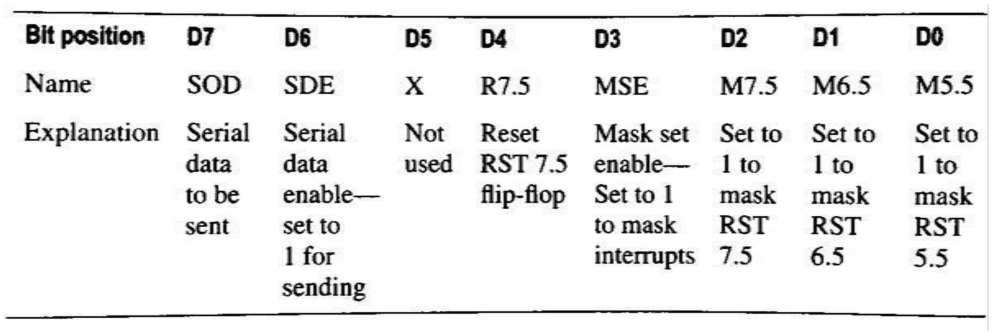
The Fig. 13 is explained by the following five points:

i. The maskable interrupts are by default masked by the Reset signal. So no interrupt is recognized by the hardware reset.
ii. The interrupts can be enabled by the EI instruction.
iii. The three RST interrupts can be selectively masked by loading the appropriate word in the accumulator and executing SIM instruction. This is called software masking.
iv. All maskable interrupts are disabled whenever an interrupt is recognized.
v. All maskable interrupts can be disabled by executing the DI instruction.

RST 7.5 alone has a flip-flop to recognize edge transition. The DI instruction reset interrupt enable flip-flop in the processor and the interrupts are disabled. To enable interrupts, EI instruction has to be executed.

SIM Instruction:

The SIM instruction is used to mask or unmask RST hardware interrupts. When executed, the SIM instruction reads the content of accumulator and accordingly mask or unmask the interrupts. The format of control word to be stored in the accumulator before executing SIM instruction is as shown in Fig. 14.



Bit position	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
Name	SOD	SDE	X	R7.5	MSE	M7.5	M6.5	M5.5
Explanation	Serial data to be sent	Serial data enable—set to 1 for sending	Not used	Reset RST 7.5 flip-flop	Mask set enable—Set to 1 to mask interrupts	Set to 1 to mask RST 7.5	Set to 1 to mask RST 6.5	Set to 1 to mask RST 5.5

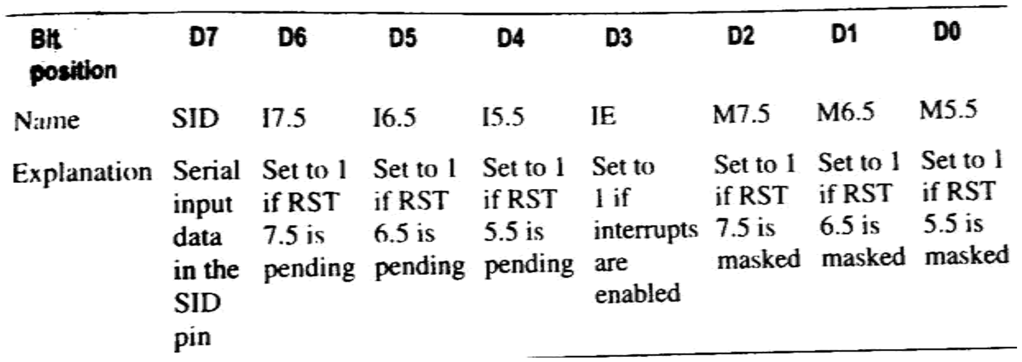
Fig. 14 Accumulator bit pattern for SIM instruction

In addition to masking interrupts, SIM instruction can be used to send serial data on the SOD line of the processor. The data to be send is placed in the MSB bit of the accumulator and the serial data output is enabled by making D6 bit to 1.

RIM Instruction:

RIM instruction is used to read the status of the interrupt mask bits. When RIM instruction is executed, the accumulator is loaded with the current status of the interrupt masks and the pending interrupts. The format and the meaning of the data stored in the accumulator after execution of RIM instruction is shown in Fig. 15.

In addition RIM instruction is also used to read the serial data on the SID pin of the processor. The data on the SID pin is stored in the MSB of the accumulator after the execution of the RIM instruction.



Bit position	D7	D6	D5	D4	D3	D2	D1	D0
Name	SID	I7.5	I6.5	I5.5	IE	M7.5	M6.5	M5.5
Explanation	Serial input data in the SID pin	Set to 1 if RST 7.5 is pending	Set to 1 if RST 6.5 is pending	Set to 1 if RST 5.5 is pending	Set to 1 if interrupts are enabled	Set to 1 if RST 7.5 is masked	Set to 1 if RST 6.5 is masked	Set to 1 if RST 5.5 is masked

Fig. 15 Accumulator bit pattern after execution of RIM instruction

Ex: Write an assembly language program to enables all the interrupts in 8085 after reset.

EI	: Enable interrupts
MVI A, 08H	: Unmask the interrupts
SIM	: Set the mask and unmask using SIM instruction

Timing of Interrupts:

The interrupts are sensed by the processor one cycle before the end of execution of each instruction. An interrupts signal must be applied long enough for it to be recognized. The longest instruction of the 8085 takes 18 clock periods. So, the interrupt signal must be applied for at least 17.5 clock periods. This decides the minimum pulse width for the interrupt signal.

The maximum pulse width for the interrupt signal is decided by the condition that the interrupt signal must not be recognized once again. This is under the control of the programmer.

8085 ASSEMBLY LANGUAGE PROGRAMS & EXPLANATIONS

Write an Assembly language 8085 program to sort an array of data in ascending order

Data

8500 – FF (data 1)

8501 – 07 (data 2)

8502 – DD (data 3)

8503 – E6 (data 4)

8504 – 85 (data 5)

Counter = n-1

n= number of data

Program

LABEL	**INSTRUCTION**	**COMMENT**
	MVI B, 04H	B= 04H
Label3	MOV C,B	B=C
	LXI H,8500H	HL =8500H
Label2	MOV A,M	A=FFH
	INX H	Increment HL
	MOV D,M	D = 07H
	CMP D	COMPARE A & D
	JC Label1	If carry=1 jump to label1
	MOV M,A	8501 = FFH
	DCX H	Decrement HL
	MOV M,D	8500 = 07H

	INX H	Increment HL
Label1	DCR C	Decrement C
	JNZ Label2	If C = not zero jump to label2
	DCR B	Decrement B
	JNZ Label3	If B = not zero jump to label3
	HLT	Stop

Write an Assembly language 8085 program to sort an array of data in descending order

<u>Data</u>

8500 – FF (data 1)

8501 – 07 (data 2)

8502 – DD (data 3)

8503 – E6 (data 4)

8504 – 85 (data 5)

Counter = n-1

n= number of data

Program

LABEL	**INSTRUCTION**	**COMMENT**
	MVI B, 04H	B= 04H
Label3	MOV C,B	B=C
	LXI H,8500H	HL =8500H
Label2	MOV A,M	A=FFH
	INX H	Increment HL

	MOV D,M	D = 07H
	CMP D	COMPARE A & D
	JNC Label1	If carry is not equal to1 jump to label1
	MOV M,A	8501 = FFH
	DCX H	Decrement HL
	MOV M,D	8500 = 07H
	INX H	Increment HL
Label1	DCR C	Decrement C
	JNZ Label2	If C = not zero jump to label2
	DCR B	Decrement B
	JNZ Label3	If B = not zero jump to label3
	HLT	Stop

Write an Assembly language 8085 program to convert binary numbers to gray

1. **STC** is used to set carry flag (CY) to 1.
2. **CMC** is used to take 1's compliment of the contents of carry flag (CY).
3. **LDA 2050** is used load the data from address 2050 in A.
4. **MOV B, A** is used to move the data of A into B.
5. **RAR** is used to rotate the bits of A along with carry flag (CY) to right one time.
6. **XRA B** is used to perform XOR operation between the contents of register A and B.
7. **STA 3050** is used to store the contents of A to 3050.

8. **HLT** is used end the program

Write an Assembly language 8085 program to convert gray to binary numbers

Explanation–

1. **LDA 2050** is used to load the data from address 2050 in A
2. **MVI C, 07** is used to move the data 07 in C
3. **MOV B, A** moves the data of A to B
4. **ANI 80** extracts the MSB(Most Significant Bit) of data available in A
5. **RRC** rotates the bits of A to right without carry
6. **ANI 7F** is used to take AND between data in A and 7F
7. **XRA B** takes XOR between the data present in A and B
8. **DCR C** is used to decrement the contents of C
9. **JNZ 2008** is used to jump to address 2008 if ZF = 0
10. **STA 3050** is used to store the result at memory address 3050
11. **HLT** is used to end the program

CHAPTER 3

8085 Assembly Language Programs & Explanations

1. Statement: Store the data byte 32H into memory location 4000H.

Program 1:

```
MVI A, 32H        : Store 32H in the accumulator
STA 4000H         : Copy accumulator contents at address 4000H
HLT               : Terminate program execution
```

Program 2:

```
LXI H             : Load HL with 4000H
MVI M             : Store 32H in memory location pointed by HL register pair
(4000H)
HLT               : Terminate program execution
```

2. Statement: Exchange the contents of memory locations 2000H and 4000H

Program 1:

```
    LDA 2000H     : Get the contents of memory location 2000H into
accumulator
    MOV B, A      : Save the contents into B register
    LDA 4000H     : Get the contents of memory location 4000Hinto
accumulator
    STA 2000H     : Store the contents of accumulator at address 2000H
    MOV A, B      : Get the saved contents back into A register
    STA 4000H     : Store the contents of accumulator at address 4000H
```

Program 2:

```
    LXI H 2000H   : Initialize HL register pair as a pointer to
memory location 2000H.
    LXI D 4000H   : Initialize DE register pair as a pointer to
memory location 4000H.
    MOV B, M      : Get the contents of memory location 2000H into B
register.
    LDAX D        : Get the contents of memory location 4000H into A
register.
    MOV M, A      : Store the contents of A register into memory
location 2000H.
    MOV A, B      : Copy the contents of B register into accumulator.
    STAX D        : Store the contents of A register into memory location
4000H.
    HLT           : Terminate program execution.
```

3. ***Sample problem***

(4000H) = 14H
(4001H) = 89H
Result = 14H + 89H = 9DH Source program

```
LXI H 4000H              : HL points 4000H
MOV A, M                 : Get first operand
INX H                    : HL points 4001H
ADD M                    : Add second operand
INX H                    : HL points 4002H
MOV M, A                 : Store result at 4002H
HLT                      : Terminate program execution
```

4. **Statement: Subtract the contents of memory location 4001H from the memory location 2000H and place the result in memory location 4002H.**

Program - 4: Subtract two 8-bit numbers

Sample problem:

(4000H)	*= 51H*
(4001H)	*= 19H*
Result	*= 51H - 19H = 38H*

Source program:

```
LXI H, 4000H             : HL points 4000H
MOV A, M                 : Get first operand
INX H                    : HL points 4001H
SUB      M               : Subtract second operand
INX H                    : HL points 4002H
MOV M, A                 : Store result at 4002H.
HLT                      : Terminate program execution
```

5. **Statement: Add the 16-bit number in memory locations 4000H and 4001H to the 16-bit number in memory locations 4002H and 4003H. The most significant eight bits of the two numbers to be added are in memory locations 4001H and 4003H. Store the result in memory locations 4004H and 4005H with the most significant byte in memory location 4005H.**

Program - 5.a: Add two 16-bit numbers - Source Program 1

Sample problem:

(4000H) = 15H
(4001H) = 1CH
(4002H) = B7H
(4003H) = 5AH
Result = 1C15 + 5AB7H = 76CCH (4004H) =
CCH
(4005H) = 76H

Source Program 1:

```
LHLD 4000H          : Get first I6-bit number in HL
XCHG                : Save first I6-bit number in DE
LHLD 4002H          : Get second I6-bit number in HL
MOV A, E            : Get lower byte of the first number
ADD L               : Add lower byte of the second number
MOV L, A            : Store result in L register
MOV A, D            : Get higher byte of the first number
ADC H               : Add higher byte of the second number with CARRY
MOV H, A            : Store result in H register
SHLD 4004H          : Store I6-bit result in memory locations 4004H and
4005H.
HLT                 : Terminate program execution
```

6. Statement: Add the contents of memory locations 40001H and 4001H and place the result in the memory locations 4002Hand 4003H.

Sample problem:

(4000H) = 7FH
(400lH) = 89H
Result = 7FH + 89H = lO8H (4002H) =
08H (4003H) = 0lH

Source program:

```
    LXI H, 4000H        :HL Points 4000H
    MOV A, M            :Get first operand
    INX H               :HL Points 4001H
    ADD M               :Add second operand
    INX H               :HL Points 4002H
    MOV M, A            :Store the lower byte of result at 4002H
    MVIA, 00            :Initialize higher byte result with 00H
```

```
ADC A                  :Add carry in the high byte result
INX H                  :HL Points 4003H
MOV M, A               :Store the higher byte of result at 4003H
HLT                    :Terminate program execution
```

7. Statement: Subtract the 16-bit number in memory locations 4002H and 4003H from the 16-bit number in memory locations 4000H and 4001H. The most significant eight bits of the two numbers are in memory locations 4001H and 4003H. Store the result in memory locations 4004H and 4005H with the most significant byte in memory location 4005H.

Sample problem

(4000H) = 19H
(400IH) = 6AH
(4004H) = I5H (4003H) = 5CH Result = 6A19H -
5C15H = OE04H (4004H) = 04H
(4005H) = OEH

Source program:

```
LHLD 4000H             : Get first 16-bit number in HL
XCHG                   : Save first 16-bit number in DE
LHLD 4002H             : Get second 16-bit number in HL
MOV A, E               : Get lower byte of the first number
SUB L                  : Subtract lower byte of the second number
MOV L, A               : Store the result in L register
MOV A, D               : Get higher byte of the first number
SBB H                  : Subtract higher byte of second number with borrow
MOV H, A               : Store l6-bit result in memory locations 4004H and
4005H.
SHLD 4004H             : Store l6-bit result in memory locations 4004H and
4005H.
HLT                    : Terminate program execution
```

8. Statement: Find the l's complement of the number stored at memory location 4400H and store the complemented number at memory location 4300H.

Sample problem:

(4400H) = 55H

Result = (4300B) = AAB
Source program:

```
LDA 4400B                               : Get the number
CMA                                   : Complement number
STA 4300H                               : Store the result
HLT                                  : Terminate program execution
```

9. Statement: Find the 2's complement of the number stored at memory location 4200H and store the complemented number at memory location 4300H.

Sample problem:

(4200H) = 55H
Result = (4300H) = AAH + 1 = ABH

Source program:

```
LDA 4200H                               : Get the number
CMA                                   : Complement the number
ADI, 01 H                             : Add one in the number
STA 4300H                               : Store the result
HLT                                  : Terminate program execution
```

10. Statement: Pack the two unpacked BCD numbers stored in memory locations 4200H and 4201H and store result in memory location 4300H. Assume the least significant digit is stored at 4200H.

Sample problem: (4200H)
= 04 (4201H) = 09
Result = (4300H) = 94

Source program

```
LDA 4201H                         : Get the Most significant BCD digit
RLC
RLC
RLC
RLC                             : Adjust the position of the second digit (09 is changed to
90)
```

```
ANI FOH                  : Make least significant BCD digit zero
MOV C, A                 : store the partial result
LDA 4200H                : Get the lower BCD digit
ADD C                    : Add lower BCD digit
STA 4300H                : Store the result
HLT                      : Terminate program execution
```

11. Statement: Two digit BCD number is stored in memory location 4200H. Unpack the BCD number and store the two digits in memory locations 4300H and 4301H such that memory location 4300H will have lower BCD digit.

Sample problem

(4200H) = 58
Result = (4300H) = 08 and (4301H) = 05

Source program

```
LDA 4200H                : Get the packed BCD number
ANI FOH                  : Mask lower nibble
RRC
RRC
RRC
RRC                      : Adjust higher BCD digit as a lower digit
STA 4301H                : Store the partial result
LDA 4200H                : .Get the original BCD number
ANI OFH                  : Mask higher nibble
STA 4201H                : Store the result
HLT                      : Terminate program execution
```

12. Statement:Read the program given below and state the contents of all registers after the execution of each instruction in sequence.

Main program:

```
4000H          LXI SP, 27FFH
4003H          LXI H, 2000H
4006H          LXI B, 1020H
4009H          CALL SUB
400CH          HLT
```

Subroutine program:

```
4100H          SUB: PUSH B
4101H          PUSH H
4102H          LXI B, 4080H
4105H          LXI H, 4090H
4108H          SHLD 2200H
4109H          DAD B
410CH          POP H
410DH           POP B
410EH          RET
```

13. Statement:Write a program to shift an eight bit data four bits right. Assume that data is in register C.

Source program:

```
MOV A, C
RAR
RAR
RAR
RAR
MOV C, A
HLT
```

14. Statement: Program to shift a 16-bit data 1 bit left. Assume data is in the HL register pair

Source program:

```
DAD H          :        Adds HL data with HL data
```

15. Statement: Write a set of instructions to alter the contents of flag register in 8085.

```
PUSH PSW              : Save flags on stack
POP H                  : Retrieve flags in 'L'
MOV A, L             : Flags in accumulator
CMA                   : Complement accumulator
MOV L, A             : Accumulator in 'L'
```

```
PUSH H                        : Save on stack
POP PSW                          : Back to flag register
HLT                            :Terminate program execution
```

16. Statement: Calculate the sum of series of numbers. The length of the series is in memory location 4200H and the series begins from memory location 4201H.

1. **Consider the sum to be 8 bit number. So, ignore carries. Store the sum at memory location 4300H.**
2. **Consider the sum to be 16 bit number. Store the sum at memory locations 4300H and 4301H**

a. Sample problem

4200H = 04H
4201H = 10H
4202H = 45H
4203H = 33H
4204H = 22H
Result = 10 +41 + 30 + 12 = H
4300H = H

Source program:

```
LDA 4200H
MOV C, A                         : Initialize counter
SUB A                              : sum = 0
LXI H, 420lH                         : Initialize pointer
BACK:           ADD M                        : SUM = SUM + data
INX H                              : increment pointer
DCR C                              : Decrement counter
JNZ BACK                         : if counter 0 repeat
STA 4300H                          : Store sum
HLT                            : Terminate program execution
```

b. Sample problem

4200H = 04H 420lH
= 9AH 4202H = 52H
4203H = 89H 4204H
= 3EH
Result = 9AH + 52H + 89H + 3EH = H 4300H = B3H
Lower byte
4301H = 0lH Higher byte

Source program:

```
          LDA 4200H
          MOV C, A                      : Initialize counter
          LXI H, 4201H                       : Initialize pointer
          SUB A                           :Sum low = 0
          MOV B, A                      : Sum high = 0
BACK: ADD M                                  : Sum = sum + data
          JNC SKIP
          INR B                            : Add carry to MSB of SUM
SKIP: INX H                                  : Increment pointer
          DCR C                            : Decrement counter
          JNZ BACK                       : Check if counter 0 repeat
          STA 4300H                        : Store lower byte
          MOV A, B
          STA 4301H                        : Store higher byte
          HLT                           :Terminate program execution
```

17. Statement: Multiply two 8-bit numbers stored in memory locations 2200H and 2201H by repetitive addition and store the result in memory locations 2300H and 2301H.

Sample problem:

(2200H) = 03H
(2201H) = B2H
Result = B2H + B2H + B2H = 216H = 216H
(2300H) = 16H
(2301H) = 02H

Source program

```
          LDA 2200H
          MOV E, A
          MVI D, 00                       : Get the first number in DE register pair
          LDA 2201H
          MOV C, A                      : Initialize counter
          LX I H, 0000 H                : Result = 0
BACK: DAD          D                       : Result = result + first number
          DCR         C                   : Decrement count
          JNZ         BACK                     : If count 0 repeat
          SHLD 2300H                       : Store result
          HLT                           : Terminate program execution
```

18. Statement:Divide 16 bit number stored in memory locations 2200H and 2201H by the 8 bit number stored at memory location 2202H. Store the quotient in memory locations 2300H and 2301H and remainder in memory locations 2302H and 2303H.

Sample problem (2200H) =
60H (2201H) = A0H
(2202H) = l2H
Result = A060H/12H = 8E8H Quotient and 10H remainder (2300H) = E8H
(2301H) = 08H
(2302H= 10H (2303H)
00H

Source program

```
      LHLD 2200H              : Get the dividend
      LDA 2202H               : Get the divisor
      MOV C, A
      LXI D, 0000H            : Quotient = 0
BACK: 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, 00                 : Check if dividend < divisor
      JNZ BACK                : if no repeat
      MOV A, L
      CMP C
      JNC BACK
      SHLD 2302H              : Store the remainder
      XCHG
      SHLD 2300H              : Store the quotient
      HLT                     : Terminate program execution
```

19. Statement:Find the number of negative elements (most significant bit 1) in a block of data. The length of the block is in memory location 2200H and the block itself begins in memory location 2201H. Store the number of negative elements in memory location 2300H

Sample problem

(2200H) = 04H

(2201H) = 56H
(2202H) = A9H
(2203H) = 73H
(2204H) = 82H

Result = 02 since 2202H and 2204H contain numbers with a MSB of 1.

Source program

```
        LDA 2200H
        MOV C, A                : Initialize count
        MVI B, 00               : Negative number = 0
        LXI H, 2201H            : Initialize pointer
BACK:   MOV A, M                : Get the number
        ANI 80H                 : Check for MSB
        JZ SKIP                 : If MSB = 1
        INR B                   : Increment negative number count
SKIP:   INX H                   : Increment pointer
        DCR C                   : Decrement count
        JNZ BACK                : If count 0 repeat
        MOV A, B
        STA 2300H               : Store the result
        HLT                     : Terminate program execution
```

20. Statement:Find the largest number in a block of data. The length of the block is in memory location 2200H and the block itself starts from memory location 2201H.
Store the maximum number in memory location 2300H. Assume that the numbers in the block are all 8 bit unsigned binary numbers.

Sample problem

(2200H) = 04
(2201H) = 34H
(2202H) = A9H
(2203H) = 78H
(2204H) =56H
Result = (2202H) = A9H

Source program

```
        LDA 2200H
        MOV C, A                : Initialize counter
        XRA A                   : Maximum = Minimum possible value = 0
        LXI H, 2201H            : Initialize pointer
BACK:   CMP M                   : Is number> maximum
        JNC SKIP                : Yes, replace maximum
```

```
          MOV A, M
SKIP: INX H
          DCR C
          JNZ BACK
          STA 2300H                    : Store maximum number
          HLT                          : Terminate program execution
```

21. Statement:Write a program to count number of l's in the contents of D register and store the count in the B register.

Source program:

```
          MVI B, 00H
          MVI C, 08H
          MOV A, D
BACK: RAR
          JNC SKIP
          INR B
SKIP: DCR C
          JNZ BACK
          HLT
```

22. Statement:Write a program to sort given 10 numbers from memory location 2200H in the ascending order.

Source program:

```
          MVI B, 09                    : Initialize counter
          START                        : LXI H, 2200H: Initialize memory pointer
          MVI C, 09H                   : Initialize counter 2
BACK: MOV A, M                         : Get the number
          INX H                        : Increment memory pointer
          CMP M                        : Compare number with next number
          JC SKIP                      : If less, don't interchange
          JZ SKIP                      : If equal, don't interchange
          MOV D, M
          MOV M, A
          DCX H
          MOV M, D
          INX H                        : Interchange two numbers
SKIP:DCR C                             : Decrement counter 2
          JNZ BACK                     : If not zero, repeat
          DCR B                        : Decrement counter 1
          JNZ START
          HLT                          : Terminate program execution
```

23. **Statement:Calculate the sum of series of even numbers from the list of numbers. The length of the list is in memory location 2200H and the series itself begins from memory location 2201H. Assume the sum to be 8 bit number so you can ignore carries and store the sum at memory location 2*****Sample problem:***

2200H= 4H
2201H= 20H
2202H= l5H
2203H= l3H
2204H= 22H
Result 22l0H= 20 + 22 = 42H = 42H

Source program:

```
          LDA 2200H
          MOV C, A
          MVI B, 00H
          LXI H, 2201H
BACK: MOV A, M
          ANI          0lH
          JNZ SKIP
          MOV A, B
          ADD          M
          MOV B, A
SKIP: INX H
          DCR          C
          JNZ          BACK
          STA          2210H
          HLT
```

24. Statement:Calculate the sum of series of odd numbers from the list of numbers. The length of the list is in memory location 2200H and the series itself begins from memory location 2201H. Assume the sum to be 16-bit. Store the sum at memory locations 2300H and 2301H.

Sample problem:

2200H = 4H
2201H= 9AH
2202H= 52H
2203H= 89H
2204H= 3FH
Result = 89H + 3FH = C8H 2300H= H
Lower byte 2301H = H Higher byte

Source program

```
          LDA 2200H
          MOV C, A                 : Initialize counter
          LXI H, 2201H                          : Initialize pointer
          MVI E, 00                : Sum low = 0
          MOV D, E                 : Sum high = 0
BACK: MOV A, M                     : Get the number
          ANI 0lH                : Mask Bit 1 to Bit7
          JZ SKIP                         : Don't add if number is even
          MOV A, E                          : Get the lower byte of sum
          ADD M                 : Sum = sum + data
          MOV E, A                          : Store result in E register
          JNC SKIP
          INR D                            : Add carry to MSB of SUM
SKIP: INX H                                    : Increment pointer
          DCR C                 : Decrement
```

25. Statement:Find the square of the given numbers from memory location 6100H and store the result from memory location 7000H

Source Program:

```
          LXI H, 6200H             : Initialize lookup table pointer
          LXI D, 6100H                       : Initialize source memory pointer
          LXI B, 7000H             : Initialize destination memory pointer
BACK: LDAX D                        : Get the number
          MOV L, A                  : A point to the square
          MOV A, M                   : Get the square
          STAX B                 : Store the result at destination memory location
          INX D                : Increment source memory pointer
          INX B                           : Increment destination memory pointer
          MOV A, C
          CPI 05H                  : Check for last number
          JNZ BACK                   : If not repeat
          HLT                        : Terminate program execution
```

26. Statement: Search the given byte in the list of 50 numbers stored in the consecutive memory locations and store the address of memory location in the memory locations 2200H and 2201H. Assume byte is in the C register and starting address of the list is 2000H. If byte is not found store 00 at 2200H and 2201H.

Source program:

```
            LX I H, 2000H                    : Initialize memory pointer 52H
            MVI B, 52H                           : Initialize counter
      BACK: MOV A, M                             : Get the number
            CMP C                      : Compare with the given byte
            JZ LAST                        : Go last if match occurs
            INX H                       : Increment memory pointer
            DCR B                                : Decrement counter
            JNZ B                    : I f not zero, repeat
            LXI H, 0000H
            SHLD 2200H
            JMP END                         : Store 00 at 2200H and 2201H
      LAST: SHLD 2200H                                 : Store memory address
      END: HLT                                : Stop
```

27. Statement: Two decimal numbers six digits each, are stored in BCD package form. Each number occupies a sequence of byte in the memory. The starting address of first number is 6000H Write an assembly language program that adds these two numbers and stores the sum in the same format starting from memory location 6200H

Source Program:

```
            LXI H, 6000H                               : Initialize pointer l to first number
            LXI D, 6l00H                             : Initialize pointer2 to second number
            LXI B, 6200H                   : Initialize pointer3 to result
            STC
            CMC                               : Carry = 0
      BACK: LDAX D                            : Get the digit
            ADD M                       : Add two digits
            DAA                              : Adjust for decimal
            STAX.B                      : Store the result
            INX H                                  : Increment pointer 1
            INX D                     : Increment pointer2
            INX B                                  : Increment result pointer
            MOV A, L
            CPI 06H                        : Check for last digit
            JNZ BACK                              : If not last digit repeat
            HLT                              : Terminate program execution
```

28. Statement: Add 2 arrays having ten 8-bit numbers each and generate a third array of result. It is necessary to add the first element of array 1 with the first

element of array-2 and so on. The starting addresses of array l, array2 and array3 are 2200H, 2300H and 2400H, respectively.

Source Program:

```
        LXI H, 2200H          : Initialize memory pointer 1
        LXI B, 2300H          : Initialize memory pointer 2
        LXI D, 2400H          : Initialize result pointer
  BACK: LDAX B                : Get the number from array 2
        ADD M                 : Add it with number in array 1
        STAX D                : Store the addition in array 3
        INX H                 : Increment pointer 1
        INX B                 : Increment pointer2
        INX D                 : Increment result pointer
        MOV A, L
        CPI 0AH               : Check pointer 1 for last number
        JNZ BACK              : If not, repeat
        HLT                   : Stop
```

29. **Statement: Write an assembly language program to separate even numbers from the given list of 50 numbers and store them in the another list starting from 2300H. Assume starting address of 50 number list is 2200H**

Source Program:

```
        LXI H, 2200H          : Initialize memory pointer l
        LXI D, 2300H          : Initialize memory pointer2
        MVI C, 32H            : Initialize counter
  BACK:MOV A, M               : Get the number
        ANI 0lH               : Check for even number
        JNZ SKIP              : If ODD, don't store
        MOV A, M              : Get the number
        STAX      D           : Store the number in result list
        INX D                 : Increment pointer 2
  SKIP: INX H                 : Increment pointer l
        DCR C                 : Decrement counter
        JNZ BACK              : If not zero, repeat
        HLT                   : Stop
```

30. **Statement: Write assembly language program with proper comments for the following:**

A block of data consisting of 256 bytes is stored in memory starting at 3000H. This block is to be shifted (relocated) in memory from 3050H onwards. Do not shift the block or part of the block anywhere else in the memory.

Source Program:

Two blocks (3000 - 30FF and 3050 - 314F) are overlapping. Therefore it is necessary to transfer last byte first and first byte last.

```
        MVI C, FFH                 : Initialize counter
        LX I H, 30FFH              : Initialize source memory pointer 3l4FH
        LXI D, 314FH               : Initialize destination memory pointer
BACK: MOV A, M                     : Get byte from source memory block
        STAX D                     : Store byte in the destination memory block
        DCX H                      : Decrement source memory pointer
        DCX                        : Decrement destination memory pointer
        DCR C                      : Decrement counter
        JNZ BACK                   : If counter 0 repeat
        HLT                        : Stop execution
```

31. Statement: Add even parity to a string of 7-bit ASCII characters. The length of the string is in memory location 2040H and the string itself begins in memory location 2041H. Place even parity in the most significant bit of each character.

Source Program:

```
        LXI H, 2040H
        MOV C ,M                   : Counter for character
REPEAT:INX H                       : Memory pointer to character
        MOV A,M                    : Character in accumulator
        ORA A                      : ORing with itself to check parity.
        JPO PAREVEN                : If odd parity place
        ORI 80H                    even parity in D7 (80).
PAREVEN:MOV M , A                  : Store converted even parity character.
        DCR C                      : Decrement counter.
        JNZ REPEAT                 : If not zero go for next character.
        HLT
```

32. Statement: A list of 50 numbers is stored in memory, starting at 6000H. Find number of negative, zero and positive numbers from this list and store these results in memory locations 7000H, 7001H, and 7002H respectively

Source Program:

```
        LXI H, 6000H            : Initialize memory pointer
        MVI C, 00H              : Initialize number counter
        MVI B, 00H              : Initialize negative number counter
        MVI E, 00H              : Initialize zero number counter
BEGIN:MOV A, M                  : Get the number
        CPI 00H                 : If number = 0
        JZ ZERONUM              : Goto zeronum
        ANI        80H          : If MSB of number = 1i.e. if
        JNZ NEGNUM                number is negative goto NEGNUM
        INR D                   : otherwise increment positive number counter
        JMP LAST
ZERONUM:INR E                   : Increment zero number counter
        JMP LAST
NEGNUM:INR B                    : Increment negative number counter
LAST:INX H                      : Increment memory pointer
        INR C                   : Increment number counter
        MOV A, C
        CPI 32H                 : If number counter = 5010 then
        JNZ BEGIN               : Store        otherwise check next number
        LXI H, 7000             : Initialize memory pointer.
        MOV M, B                : Store        negative number.
        INX H
        MOV M, E                : Store        zero number.
        INX H
        MOV M, D                : Store positive number.
        HLT                     : Terminate execution
```

33. Statement:Write an 8085 assembly language program to insert a string of four characters from the tenth location in the given array of 50 characters

Solution:

Step 1: Move bytes from location 10 till the end of array by four bytes downwards.

Step 2: Insert four bytes at locations 10, 11, 12 and 13.

Source Program:

```
        LXI H, 2l31H            : Initialize pointer at the last location of array.
        LXI D, 2l35H            : Initialize another pointer to point the last
location of array after insertion.
AGAIN: MOV A, M                 : Get the character
```

```
          STAX D                    : Store at the new location
          DCX D                     : Decrement destination pointer
          DCX H                     : Decrement source pointer
          MOV A, L                  : [check whether desired
          CPI 05H                   bytes are shifted or not]
          JNZ AGAIN                 : if not repeat the process
          INX H                     : adjust the memory pointer
          LXI D, 2200H              : Initialize the memory pointer to point the string to
be inserted
REPE: LDAX D                        : Get the character
          MOV M, A                  : Store it in the array
          INX D                     : Increment source pointer
          INX H                     : Increment destination pointer
          MOV A, E                  : [Check whether the 4 bytes
          CPI 04                    are inserted]
          JNZ REPE                  : if not repeat the process
          HLT                       : stop
```

34. Statement:Write an 8085 assembly language program to delete a string of 4 characters from the tenth location in the given array of 50 characters.

Solution: Shift bytes from location 14 till the end of array upwards by 4 characters i.e. from location 10 onwards.

Source Program:

```
LXI H, 2l0DH                        :Initialize source memory pointer at the 14thlocation
of the array.
LXI D, 2l09H                        : Initialize destn memory pointer at the 10th location
of the array.
MOV A, M                  : Get the character
STAX D                    : Store character at new location
INX D                     : Increment destination pointer
INX H                     : Increment source pointer
MOV A, L                  : [check whether desired
CPI 32H                   bytes are shifted or not]
JNZ REPE                  : if not repeat the process
HLT                       : stop
```

35. Statement:Multiply the 8-bit unsigned number in memory location 2200H by the 8-bit unsigned number in memory location 2201H. Store the 8 least significant bits of the result in memory location 2300H and the 8 most significant bits in memory location 2301H.

Sample problem:

```
          (2200)          = 1100 (0CH)
          (2201)          = 0101 (05H)
     Multiplicand          = 1100 (1210)
     Multiplier                = 0101 (510)
     Result                       = 12 x 5 = (6010)
```

Source program

```
          LXI H, 2200                   : Initialize the memory pointer
          MOV E, M                 : Get multiplicand
          MVI D, 00H                    : Extend to 16-bits
          INX H                       : Increment memory pointer
          MOV A, M                 : Get multiplier
          LXI H, 0000                   : Product = 0
          MVI B, 08H                   : Initialize counter with count 8
     MULT: DAD H                          : Product = product x 2
          RAL
          JNC SKIP                 : Is carry from multiplier 1 ?
          DAD D                        : Yes, Product =Product + Multiplicand
     SKIP: DCR B                          : Is counter = zero
          JNZ MULT                  : no, repeat
          SHLD 2300H                     : Store the result
          HLT                     : End of program
```

36. Statement:Divide the 16-bit unsigned number in memory locations 2200H and 2201H (most significant bits in 2201H) by the B-bit unsigned number in memory location 2300H store the quotient in memory location 2400H and remainder in 2401H

Assumption: The most significant bits of both the divisor and dividend are zero.

Source program

```
     MVI E, 00                : Quotient = 0
     LHLD        2200H                  : Get dividend
     LDA 2300                 : Get divisor
     MOV B, A                 : Store          divisor
     MVI C, 08                : Count = 8
NEXT: DAD H                   : Dividend = Dividend x 2
     MOV A, E
     RLC
     MOV E, A                 : Quotient = Quotient x 2
```

```
        MOV A, H
        SUB B                   : Is most significant byte of Dividend > divisor
        JC SKIP                 : No, go to Next step
        MOV H, A                : Yes, subtract divisor
        INR E                   : and Quotient = Quotient + 1
SKIP:DCR C                      : Count = Count - 1
        JNZ NEXT                : Is count =0 repeat
        MOV A, E
        STA 2401H               : Store Quotient
        Mov A, H
        STA 2410H               : Store remainder
        HLT                     : End of program
```

37. DAA instruction is not present. Write a sub routine which will perform the same task as DAA.

Sample Problem:

Execution of DAA instruction:

- **If the value of the low order four bits (03-00) in the accumulator is greater than 9 or if auxiliary carry flag is set, the instruction adds 6 '(06) to the low-order four bits.**
- **If the value of the high-order four bits (07-04) in the accumulator is greater than 9 or if carry flag is set, the instruction adds 6(06) to the high-order four bits.**

Source Program:

```
        LXI SP, 27FFH           : Initialize stack pointer
        MOV E, A                : Store the contents of accumulator
        ANI 0FH                 : Mask upper nibble
        CPI 0A H                : Check if number is greater than 9
        JC SKIP                 : if no go to skip
        MOV A, E                : Get the number
        ADI 06H                 : Add 6 in the number
        JMP SECOND              : Go for second check
SKIP: PUSH PSW                  : Store accumulator and flag contents in stack
        POP B                   : Get the contents of accumulator in B register and
flag register contents in                                C register
        MOV A, C                : Get flag register contents in accumulator
        ANI 10H                 : Check for bit 4
        JZ SECOND               : if zero, go for second check
        MOV A, E                : Get the number
        ADI 06                  : Add 6 in the number
SECOND: MOV E, A                : Store the contents of accumulator
        ANI FOH                 : Mask lower nibble
        RRC
        RRC
        RRC
        RRC                         : Rotate number 4 bit right
        CPI 0AH                     : Check if number is greater than 9
        JC SKIPl                    : if no go to skip 1
        MOV A, E                    : Get the number
        ADI 60 H                    : Add 60 H in the number
        JMP LAST                    : Go to last
SKIP1: JNC LAST                     : if carry flag = 0 go to last
        MOV A, E                    : Get the number
        ADI 60 H                    : Add 60 H in the number
LAST: HLT
```

38. tement:To test RAM by writing '1' and reading it back and later writing '0' (zero) and reading it back. RAM addresses to be checked are 40FFH to 40FFH. In case of any error, it is indicated by writing 01H at port 10H

Source Program:

```
        LXI H, 4000H                : Initialize memory pointer
BACK: MVI M, FFH                    : Writing '1' into RAM
        MOV A, M                    : Reading data           from RAM
        CPI FFH                     : Check for ERROR
        JNZ ERROR                   : If yes go to ERROR
        INX H                       : Increment memory pointer
        MOV A, H
        CPI SOH                     : Check for last check
        JNZ BACK                    : If not last, repeat
        LXI H, 4000H                : Initialize memory pointer
BACKl: MVI M, OOH                   : Writing '0' into RAM
        MOV A, M                    : Reading data from RAM
        CPI OOH                     : Check for          ERROR
        INX H                       : Increment memory pointer
        MOV A, H
        CPI SOH                     : Check for last check
        JNZ BACKl                   : If not last, repeat
        HLT                         : Stop Execution
```

39. tement:Write an assembly language program to generate fibonacci number

Source Program:

```
        MVI D, COUNT MVI            Initialize counter
        B, 00 MVI C, 01              Initialize variable to store previous number
                                     Initialize variable to store current number
        MOV A, B                    :[Add two numbers]
BACK: ADD C                              :[Add two numbers]
        MOV B, C                    : Current number is now previous number
        MOV C, A                    : Save result as a new current number
        DCR D                           : Decrement count
        JNZ BACK                     : if count 0 go to BACK
        HLT                         : Stop
```

40. tement:Write a program to generate a delay of 0.4 sec if the crystal frequency is 5 MHz

Calculation: In 8085, the operating frequency is half of the crystal frequency,

ie.Operating frequency = 5/2 = 2.5 MHz

Time for one T -state =

Number of T-states required =

= 1 x 106

Source Program:

```
LXI B, count                              : 16 - bit count
BACK: DCX B                            : Decrement count
MOV A, C
ORA B                                 : Logically OR Band C
JNZ BACK                                     : If result is not zero repeat
```

41. tement: Arrange an array of 8 bit unsigned no in descending order

Source Program:

```
        START:MVI B, 00                    ; Flag = 0
              LXI H, 4150                  ; Count = length of array
              MOV C, M
              DCR C                        ; No. of pair = count -1
              INX H                        ; Point to start of array
        LOOP:MOV A, M                      ; Get kth element
              INX H
              CMP M                         ; Compare to (K+1) th element
              JNC LOOP 1                    ; No interchange if kth >= (k+1) th
              MOV D, M                    ; Interchange if out of order
              MOV M, A                    ;
              DCR H
              MOV M, D
              INX H
              MVI B, 01H                   ; Flag=1
        LOOP 1:DCR C                      ; count down
              JNZ LOOP                     ;
              DCR B                          ; is flag = 1?
```

```
                JZ START                        ; do another sort, if yes
                HLT                             ; If flag = 0, step execution
```

42. tement: Transfer ten bytes of data from one memory to another memory block. Source memory block starts from memory location 2200H where as destination memory block starts from memory location 2300H

Source Program:

```
                LXI H, 4150                  : Initialize memory pointer
                MVI B, 08                           : count for 8-bit
                MVI A, 54
        LOOP : RRC
                JC LOOP1
                MVI M, 00                           : store zero it no carry
                JMP COMMON
        LOOP2: MVI M, 01                     : store one if there is a carry
COMMON: INX H
                DCR B              : check for carry
                JNZ LOOP
                HLT                      : Terminate the program
```

43. tement: Program to calculate the factorial of a number between 0 to 8

Source program

```
        LXI SP, 27FFH                    ; Initialize stack pointer
        LDA 2200H                            ; Get the number
        CPI 02H                       ; Check if number is greater than 1
        JC LAST
        MVI D, 00H                                    ; Load number as a result
        MOV E, A
        DCR A
        MOV C,A                        ; Load counter one less than number
        CALL FACTO                                  ; Call subroutine FACTO
        XCHG                             ; Get the result in HL
        SHLD 2201H                          ; Store result in the memory
        JMP END
LAST: LXI H, 000lH                                  ; Store result = 01
END: SHLD 2201H
        HLT
```

44. tement:Write a program to find the Square Root of an 8 bit binary number. The binary number is stored in memory location 4200H and store the square root in 4201H.

Source Program:

```
                LDA 4200H                         : Get the given data(Y) in A register
                MOV B,A                         : Save the data in B register
                MVI C,02H                         : Call the divisor(02H) in C register
                CALL DIV                         : Call division subroutine to get initial value(X)
in D-reg
        REP: MOV E,D                         : Save the initial value in E-reg
                MOV A,B                         : Get the dividend(Y) in A-reg
                MOV C,D                         : Get the divisor(X) in C-reg
                CALL DIV                         : Call division subroutine to get initial
value(Y/X) in D-reg
                MOV A, D                         : Move Y/X in A-reg
                ADD E                              : Get the((Y/X) + X) in A-reg
                MVI C, 02H                         : Get the divisor(02H) in C-reg
                CALL DIV                         : Call division subroutine to get ((Y/X) + X)/2
in D-reg.This is XNEW
                MOV A, E                         : Get Xin A-reg
                CMP D                              : Compare X and XNEW
                JNZ REP                         : If XNEW is not equal to X, then repeat
                STA 4201H                         : Save the square root in memory
                HLT                              : Terminate program execution
```

45. tement:Write a simple program to Split a HEX data into two nibbles and store it in memory

Source Program:

```
        LXI H, 4200H                         : Set pointer data for array
        MOV B,M                         : Get the data in B-reg
        MOV A,B                         : Copy the data to A-reg
        ANI OFH                         : Mask the upper nibble
        INX H                              : Increment address as 4201
        MOV M,A                         : Store the lower nibble in memory
        MOV A,B                         : Get the data in A-reg
        ANI FOH                         : Bring the upper nibble to lower nibble position
        RRC
        RRC
        RRC
        RRC
        INX H
        MOV M,A                         : Store the upper nibble in memory
        HLT                              : Terminate program execution
```

46. tement: Add two 4 digit BCD numbers in HL and DE register pairs and store result in memory locations, 2300H and 2301H. Ignore carry after 16 bit.

Sample Problem:

(HL) =3629 (DE)
=4738
Step 1 : 29 + 38 = 61 and auxiliary carry flag = 1
:.add 06
61 + 06 = 67
Step 2 : 36 + 47 + 0 (carry of LSB) = 7D

Lower nibble of addition is greater than 9, so add 6. 7D + 06 = 83
Result = 8367

Source program

```
MOV A, L          : Get lower 2 digits of no. 1
ADD E             : Add two lower digits
DAA               : Adjust result to valid BCD
STA 2300H         : Store partial result
MOV A, H          : Get most significant 2 digits of number
ADC D             : Add two most significant digits
DAA               : Adjust result to valid BCD
STA 2301H         : Store partial result
HLT               : Terminate program execution
```

47. tement: Subtract the BCD number stored in E register from the number stored in the D register.

Source Program:

```
MVI A,99H
SUB E          : Find the 99's complement of subtrahend
INR A          : Find 100's complement of subtrahend
ADD D          : Add minuend to 100's complement of subtrahend
DAA            : Adjust for BCD
HLT            : Terminate program execution
```

48. tement: Write an assembly language program to multiply 2 BCD numbers

Source Program:

```
          MVI C, Multiplier                  : Load BCD multiplier
          MVI B, 00                        : Initialize counter
          LXI H, 0000H                         : Result = 0000
          MVI E, multiplicand                    : Load multiplicand
          MVI D, 00H                         : Extend to 16-bits
BACK:     DAD D                                 : Result Result + Multiplicand
          MOV A, L                        : Get the lower byte of the result
          ADI, 00H
          DAA                             : Adjust the lower byte of result to BCD.
          MOV L, A                        : Store the lower byte of result
          MOV A, H                         : Get the higher byte of the result
          ACI, 00H
          DAA                             : Adjust the higher byte of the result to BCD
          MOV H, A                         : Store the higher byte of result.
          MOV A, B                        : [Increment
          ADI 01H                        : counter
          DAA                             : adjust it to BCD and
          MOV B,A                        : store it]
          CMP C                              : Compare if count = multiplier
          JNZ BACK                          : if not equal repeat
          HLT                            : Stop
```

6. INSTRUCTION EXECUTION AND TIMING DIAGRAM:

Each instruction in 8085 microprocessor consists of two part- operation code (opcode) and operand. The opcode is a command such as ADD and the operand is an object to be operated on, such as a byte or the content of a register.

Instruction Cycle: The time taken by the processor to complete the execution of an instruction. An instruction cycle consists of one to six machine cycles.

Machine Cycle: The time required to complete one operation; accessing either the memory or I/O device. A machine cycle consists of three to six T-states.

T-State: Time corresponding to one clock period. It is the basic unit to calculate execution of instructions or programs in a processor.

To execute a program, 8085 performs various operations as:

- Opcode fetch
- Operand fetch
- Memory read/write
- I/O read/write

External communication functions are:

- Memory read/write
- I/O read/write
- Interrupt request acknowledge

Opcode Fetch Machine Cycle:

It is the first step in the execution of any instruction. The timing diagram of this cycle is given in Fig. 7.

The following points explain the various operations that take place and the signals that are changed during the execution of opcode fetch machine cycle:

T1 clock cycle

i. The content of PC is placed in the address bus; AD0 - AD7 lines contains lower bit address and A8 – A15 conta~~in~~s higher bit address.
ii. IO/M signal is low indicating that a memory location is being accessed. S1 and S0 also changed to the levels as indicated in Table 1.
iii. ALE is high, indicates that multiplexed AD0 – AD7 act as lower order bus.

T2 clock cycle

i. Multiplexed address bus is now changed to data bus.
ii. The RD signal is made low by the processor. This signal makes the memory device load the data bus with the contents of the location addressed by the processor.

T3 clock cycle

i. The opcode available on the data bus is read by the processor and moved to the instruction register.
ii. The RD signal is deactivated by making it logic 1.

T4 clock cycle

i. The processor decode the instruction in the instruction register and generate the necessary control signals to execute the instruction. Based on the instruction further operations such as fetching, writing into memory etc takes place.

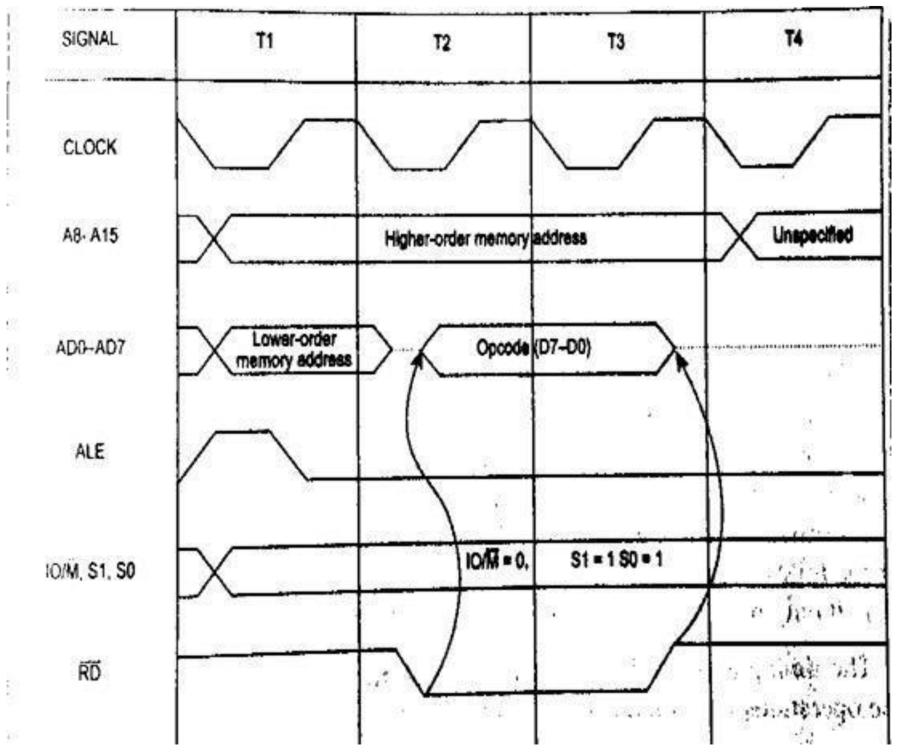


Fig. 7 Timing diagram for opcode fetch cycle

Memory Read Machine Cycle:

The memory read cycle is executed by the processor to read a data byte from memory. The machine cycle is exactly same to opcode fetch except: a) It has three T-states b) The S0 signal is set to 0. The timing diagram of this cycle is given in Fig. 8.

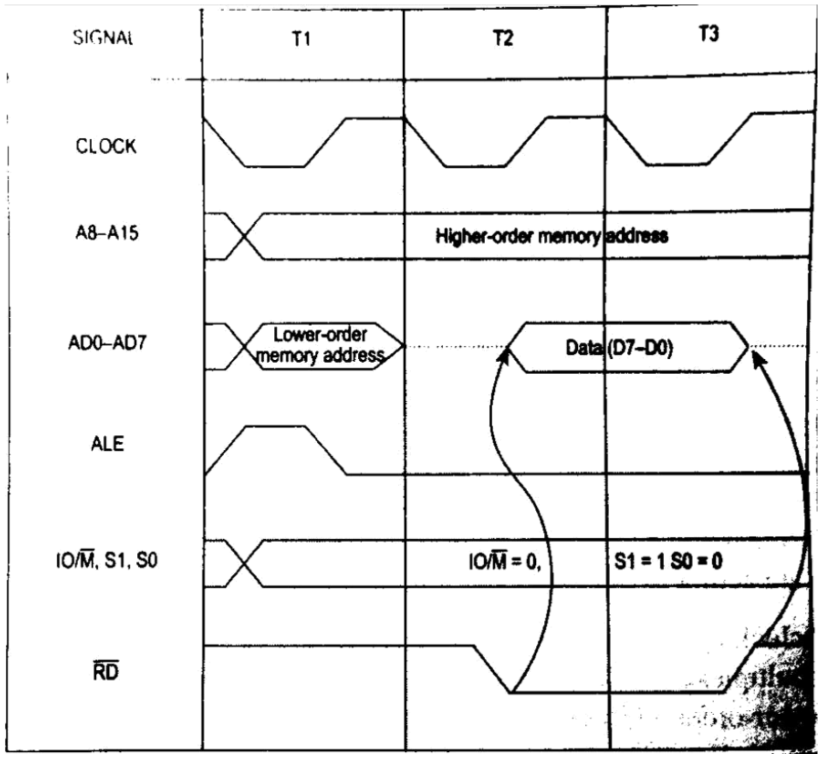


Fig. 8 Timing diagram for memory read machine cycle

Memory Write Machine Cycle:

The memory write cycle is executed by the processor to write a data byte in a memory location. The processor takes three T-states and $\overline{WR}$ signal is made low. The timing diagram of this cycle is given in Fig. 9.

I/O Read Cycle:

The I/O read cycle is executed by the processor to read a data byte from I/O port or from peripheral, which is I/O mapped in the system. The 8-bit port address is placed both in the lower and higher order address bus. The processor takes three T-states to execute this machine cycle. The timing diagram of this cycle is given in Fig. 10.

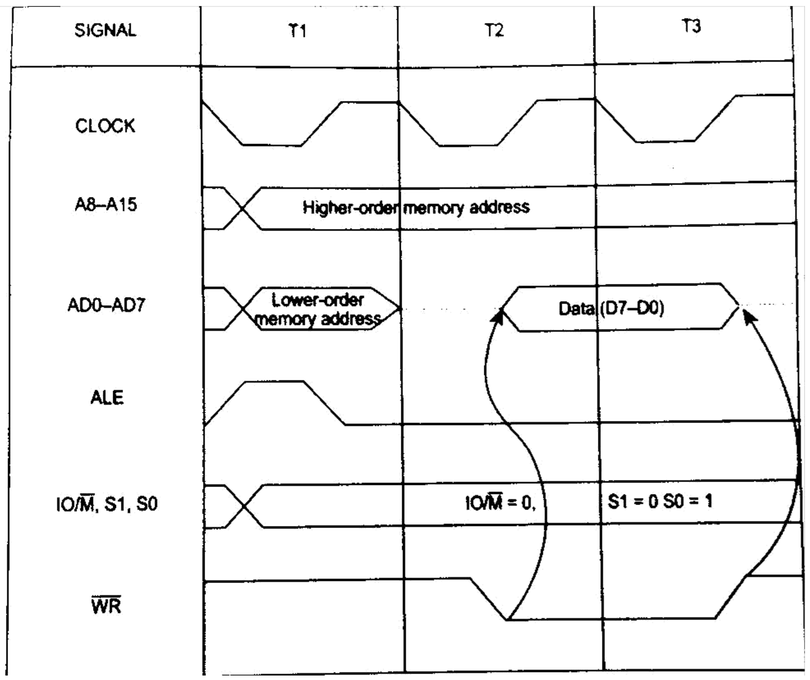


Fig. 9 Timing diagram for memory write machine cycle

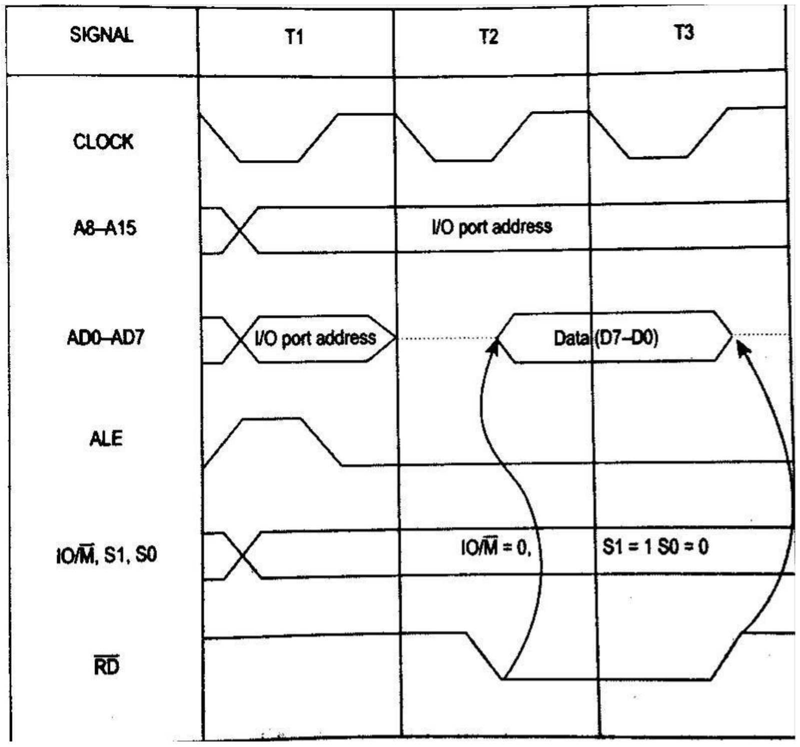


Fig. 10 Timing diagram I/O read machine cycle

I/O Write Cycle:

The I/O write cycle is executed by the processor to write a data byte to I/O port or to a peripheral, which is I/O mapped in the system. The processor takes three T-states to execute this machine cycle. The timing diagram of this cycle is given in Fig. 11.

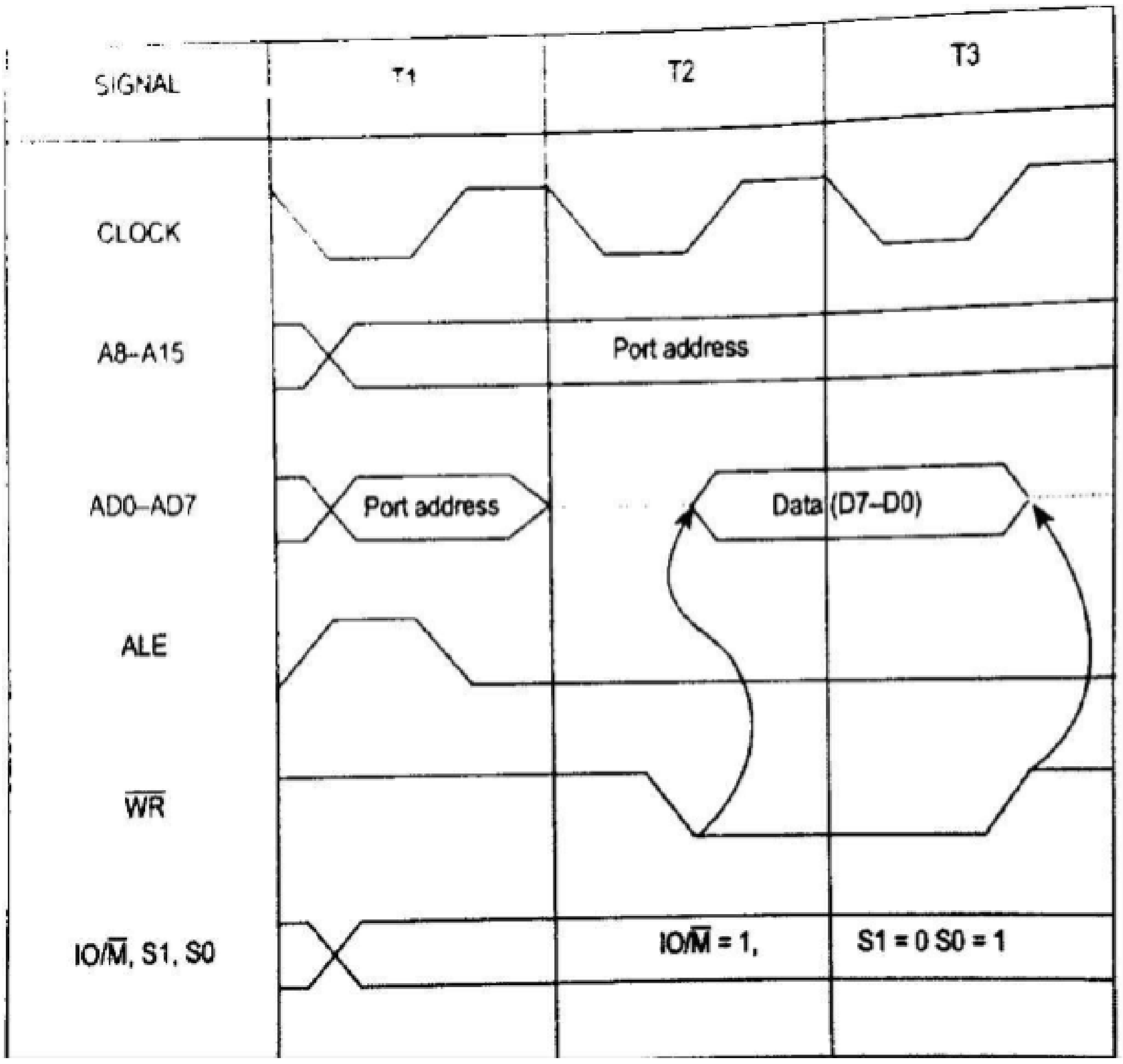


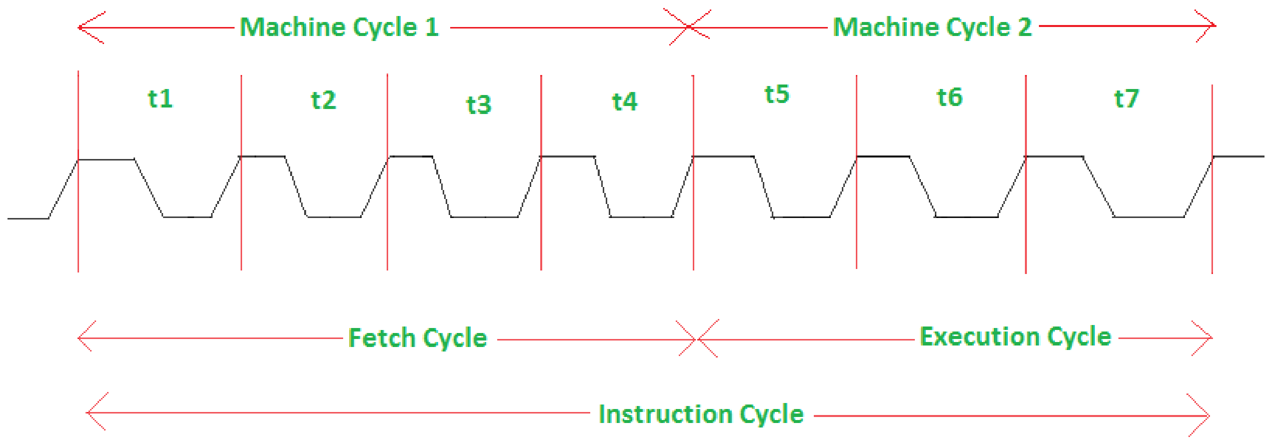
Fig. 11 Timing diagram I/O write machine cycle

Instruction cycle in 8085 microprocessor

Time required to execute and fetch an entire instruction is called *instruction cycle*. It consists:

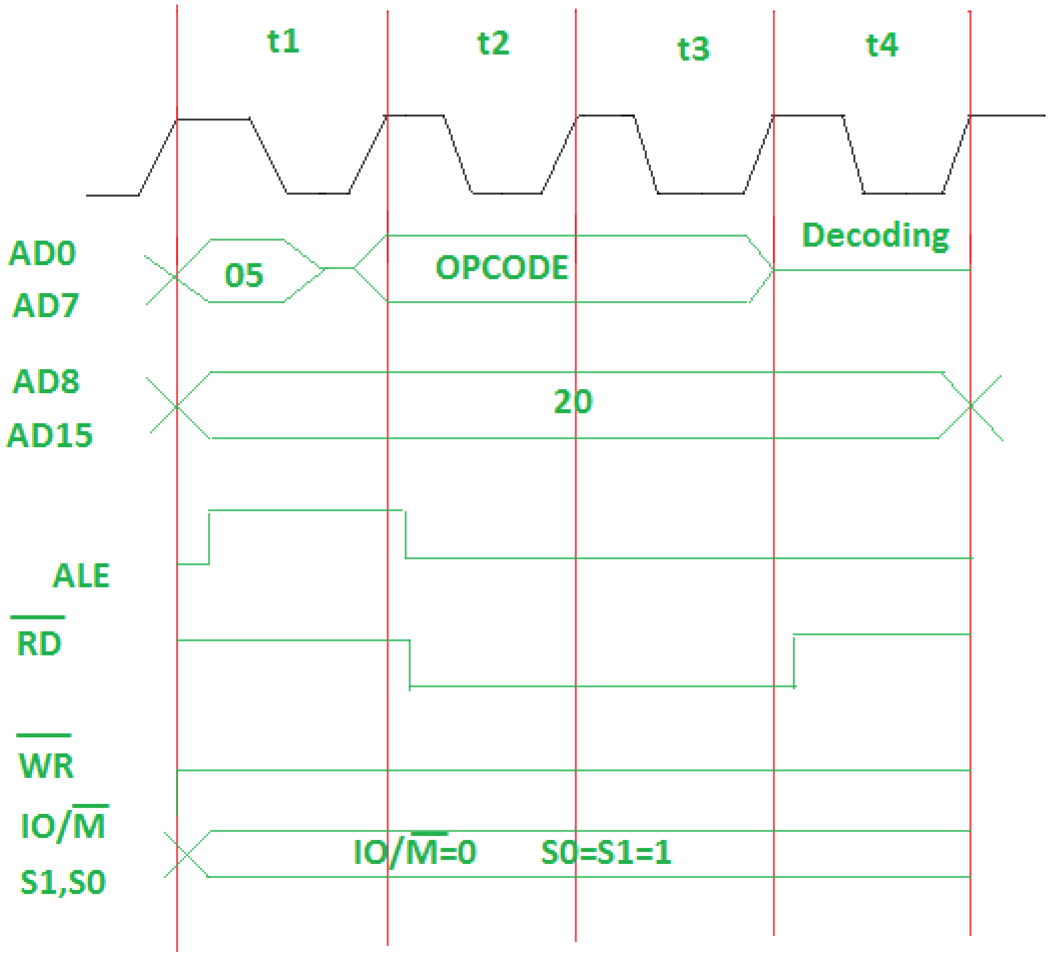
- **Fetch cycle –** The next instruction is fetched by the address stored in program counter (PC) and then stored in the instruction register.
- **Decode instruction –** Decoder interprets the encoded instruction from instruction register.
- **Reading effective address –** The address given in instruction is read from main memory and required data is fetched. The effective address depends on direct addressing mode or indirect addressing mode.
- **Execution cycle –** consists memory read (MR), memory write (MW), input output read (IOR) and input output write (IOW)

The time required by the microprocessor to complete an operation of accessing memory or input/output devices is called *machine cycle*. One time period of frequency of microprocessor is called *t-state*. A t-state is measured from the falling edge of one clock pulse to the falling edge of the next clock pulse.
Fetch cycle takes four t-states and execution cycle takes three t-states.



Instruction cycle in 8085 microprocessor

Timing diagram for fetch cycle or opcode fetch:

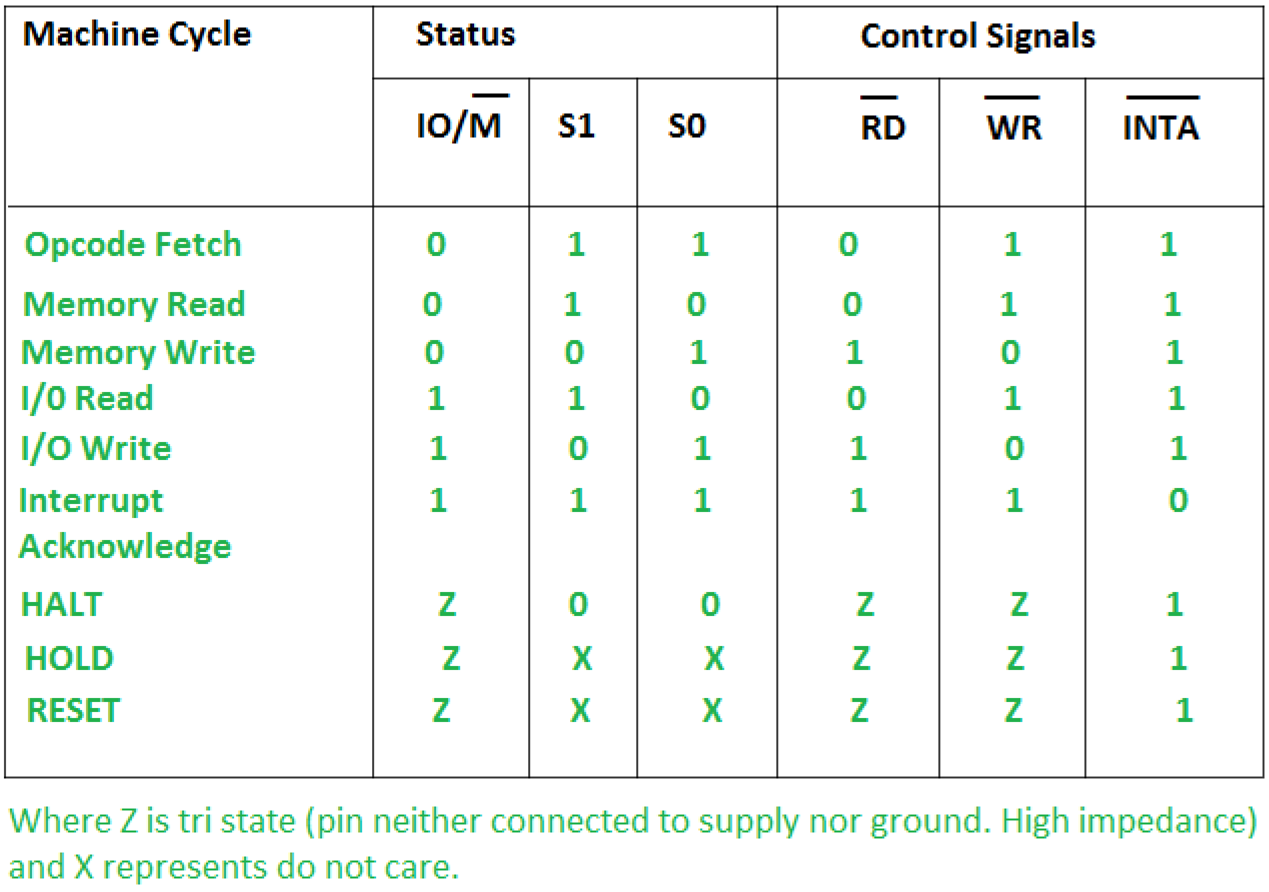


Timing diagram for opcode fetch

Above diagram represents:

- **05 –** lower bit of address where opcode is stored. Multiplexed address and data bus AD0-AD7 are used.
- **20 –** higher bit of address where opcode is stored. Multiplexed address and data bus AD8-AD15 are used.
- **ALE –** Provides signal for multiplexed address and data bus. If signal is high or 1, multiplexed address and data bus will be used as address bus. To fetch lower bit of address, signal is 1 so that multiplexed bus can act as address bus. If signal is low or 0, multiplexed bus will be used as data bus. When lower bit of address is fetched then it will act as data bus as the signal is low.
- **RD (low active) –** If signal is high or 1, no data is read by microprocessor. If signal is low or 0, data is read by microprocessor.
- **WR (low active) –** If signal is high or 1, no data is written by microprocessor. If signal is low or 0, data is written by microprocessor.

- **IO/M (low active) and S1, S0 –** If signal is high or 1, operation is performing on input output. If signal is low or 0, operation is performing on memory.



Machine Cycle	Status			Control Signals		
	$IO/\overline{M}$	S1	S0	$\overline{RD}$	$\overline{WR}$	$\overline{INTA}$
Opcode Fetch	0	1	1	0	1	1
Memory Read	0	1	0	0	1	1
Memory Write	0	0	1	1	0	1
I/0 Read	1	1	0	0	1	1
I/O Write	1	0	1	1	0	1
Interrupt Acknowledge	1	1	1	1	1	0
HALT	Z	0	0	Z	Z	1
HOLD	Z	X	X	Z	Z	1
RESET	Z	X	X	Z	Z	1

Where Z is tri state (pin neither connected to supply nor ground. High impedance) and X represents do not care.

8085 machine cycle status and control signals

1. Introduction to Microprocessor

Definition:

- *"The microprocessor is a multipurpose, clock driven, register based, digital-integrated circuit which accepts binary data as input, processes it according to instructions stored in its memory, and provides results as output."*
- "Microprocessor is a computer Central Processing Unit (CPU) on a single chip that contains millions of transistors connected by wires."

Introduction:

- A microprocessor is designed to perform arithmetic and logic operations that make use of small number-holding areas called registers.
- Typical microprocessor operations include adding, subtracting, comparing two numbers, and fetching numbers from one area to another.

2. Components of Microprocessor

- Microprocessor is capable of performing various computing functions and making decisions to change the sequence of program execution.
- The microprocessor can be divided into three segments as shown in the figure, Arithmetic/logic unit (ALU), register array, and control unit.
- These three segment is responsible for all processing done in a computer

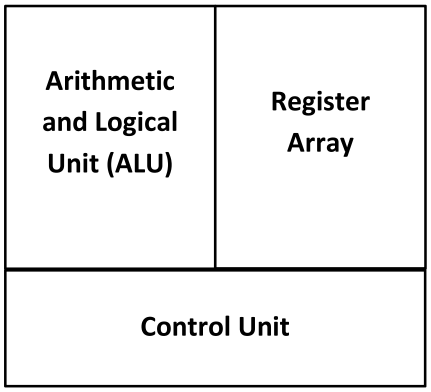


Figure: Components of Microprocessor

Arithmetic and logic unit (ALU)

- It is the unit of microprocessor where various computing functions are performed on the data.
- It performs arithmetic operations such as addition, subtraction, and logical operations such as OR,AND, and Exclusive-OR.
- It is also known as the brain of the computer system.

Register array

- It is the part of the register in microprocessor which consists of various registers identified by letters such as B, C, D, E, H, and L.
- Registers are the small additional memory location which are used to store and transfer data and programs that are currently being executed.

Control unit

- The control unit provides the necessary timing and control signals to all the operations in the microcomputer.
- It controls and executes the flow of data between the microprocessor, memory and peripherals.
- The control bus is bidirectional and assists the CPU in synchronizing control signals to internal devices and external components.
- This signal permits the CPU to receive or transmit data from main memory.

3. System bus (data, address and control bus).

- This network of wires or electronic pathways is called the 'Bus'.
- A system bus is a single computer bus that connects the major components of a computer system.
- It combines the functions of a data bus to carry information, an address bus to determine where it should be sent, and a control bus to determine its operation.
- The technique was developed to reduce costs and improve modularity.

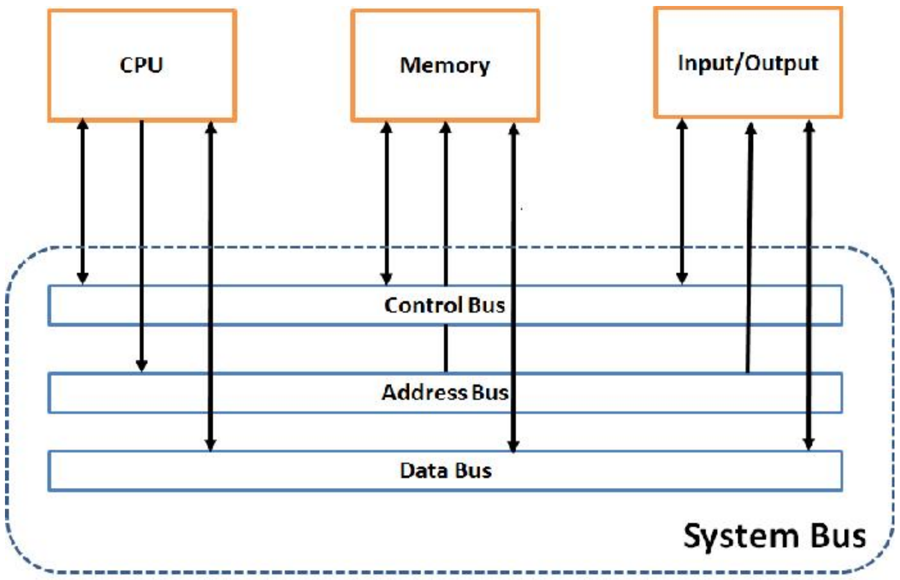


Figure: System Bus

Address Bus

- It is a group of wires or lines that are used to transfer the addresses of Memory or I/O devices.
- It is unidirectional.
- The width of the address bus corresponds to the maximum addressing capacity of the bus, or the largest address within memory that the bus can work with.
- The addresses are transferred in binary format, with each line of the address bus carrying a single binary digit.
- Therefore the maximum address capacity is equal to two to the power of the number of lines present (2^lines).

Data Bus

- It is used to transfer data within Microprocessor and Memory/Input or Output devices.
- It is bidirectional as Microprocessor requires to send or receive data.
- Each wire is used for the transfer of signals corresponding to a single bit of binary data.
- As such, a greater width allows greater amounts of data to be transferred at the same time.

Control Bus

- Microprocessor uses control bus to process data, i.e. what to do with the selected memory location.
- Some control signals are Read, Write and Opcode fetch etc.
- Various operations are performed by microprocessor with the help of control bus.
- This is a dedicated bus, because all timing signals are generated according to control signal.

4. Microprocessor systems with bus organization

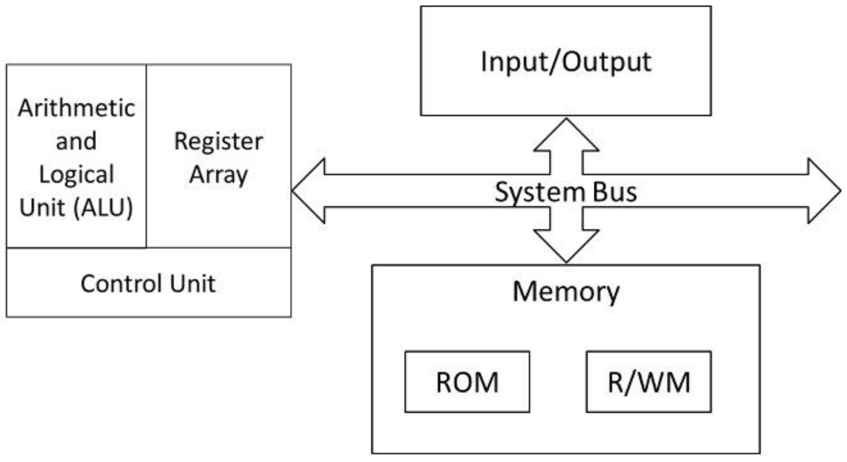


Figure: Microprocessor systems with bus organization

- To design any meaningful application microprocessor requires support of other auxiliary devices.
- In most simplified form a microprocessor based system consist of a microprocessor, I/O (input/output) devices and memory.
- These components are interfaced (connected) with microprocessor over a common communication path called system bus. Typical structure of a microprocessor based system is shown in Figure.
- Here, microprocessor is master of the system and responsible for executing the program and coordinating with connected peripherals as required.
- Memory is responsible for storing program as well as data. System generally consists of two types of memories ROM (Read only and non-volatile) and RAM (Read/Write and volatile).
- I/O devices are used to communicate with the environment. Keyboard can be example of input devices and LED, LCD or monitor can be example of output device.
- Depending on the application level of sophistication varies in a microprocessor based systems. For example: washing machine, computer.

1. Explain Classification of Memory

Ans.

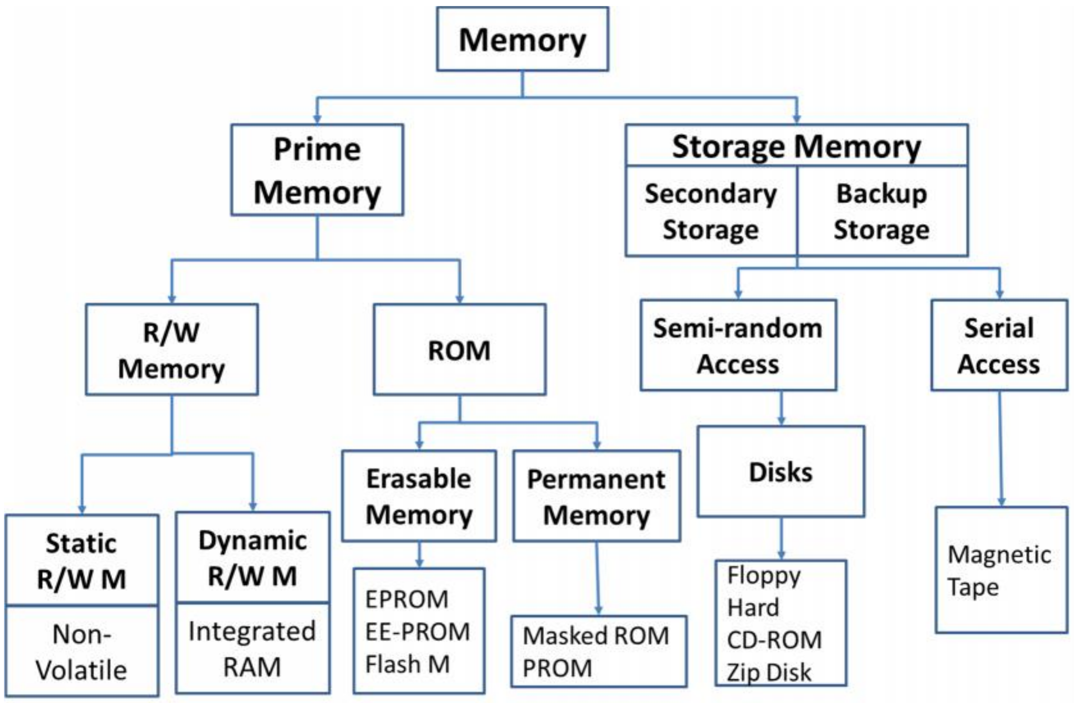


Figure: Classification of Memory

ROM (Read Only Memory):

The first classification of memory is ROM. The data in this memory can only be read, no writing is allowed. It is used to store permanent programs. It is a nonvolatile type of memory.

The classification of ROM memory is as follows:

1. **Masked ROM**: the program or data are permanently installed at the time of manufacturing as per requirement. The data cannot be altered. The process of permanent recording is expensive but economic for large quantities.

2. **PROM (Programmable Read Only Memory):** The basic function is same as that of masked ROM. but in PROM, we have fuse links. Depending upon the bit pattern, the fuse can be burnt or kept intact. This job is performed by PROM programmer.
   To do this, it uses high current pulse between two lines. Because of high current, the fuse will get burnt; effectively making two lines open. Once a PROM is programmed we cannot change connections, only a facility provided over masked ROM is, the user can load his program in it. The disadvantage is a chance of re-growing of the fuse and changes the programmed data because of aging.

3. **EPROM (Erasable Programmable Read Only Memory):** the EPROM is programmable by the user. It uses MOS circuitry to store data. They store 1's and 0's in form of charge. The information stored can be erased by exposing the memory to ultraviolet light which erases the data stored in all memory locations. For ultraviolet light, a quartz window is provided which is covered during normal operation. Upon erasing it can be reprogrammed by using EPROM programmer. This type of memory is used in a project developed and for experiment use. The advantage is it can be programmed erased and reprogrammed. The disadvantage is all the data get erased even if you want to change single data bit.

4. **EEPROM:** EEPROM stands for electrically erasable programmable read only memory. This is similar to EPROM except that the erasing is done by electrical signals instead of ultraviolet light. The main advantage is the memory location can be selectively erased and reprogrammed. But the manufacturing process is complex and expensive so do not commonly used.

R/W Memory (Read/Write Memory):

The RAM is also called as read/write memory. The RAM is a volatile type of memory. It allows the programmer to read or write data. If the user wants to check the execution of any program, user feeds the program in RAM memory and executes it. The result of execution is then checked by either reading memory location contents or by register contents.

Following is the classification of RAM memory.
It is available in two types:

1. **SRAM (Static RAM):** SRAM consists of the flip-flop; using either transistor or MOS. for each bit we require one flip-flop. Bit status will remain as it is; unless and until you perform next write operation or power supply is switched off.

   Advantages of SRAM:
   - Fast memory (less access time)
   - Refreshing circuit is not required.

   Disadvantages of SRAM:
   - Low package density
   - Costly

2. **DRAM (Dynamic RAM):** In this type of memory a data is stored in form of charge in capacitors. When data is 1, the capacitor will be charged and if data is 0, the capacitor will not be charged. Because of capacitor leakage currents, the data will not be held by these cells. So the DRAMs require refreshing of memory cells. It is a process in which same data is read and written after a fixed interval.

   Advantages of DRAM:

   - High package density
   - Low cost

   Disadvantages of DRAM:
   - Required refreshing circuit to maintain or refresh charge on the capacitor, every after few milliseconds.

Secondary Memory

- **Magnetic Disk**: The Magnetic Disk is Flat, circular platter with metallic coating that is rotated beneath read/write heads. It is a Random access device; read/write head can be moved to any location on the platter

- **Floppy Disk**: These are small removable disks that are plastic coated with magnetic recording material. Floppy disks are typically 3.5" in size (diameter) and can hold 1.44 MB of data. This portable storage device is a rewritable media and can be reused a number of times. Floppy disks are commonly used to move files between different computers. The main disadvantage of floppy disks is that they can be damaged easily and, therefore, are not very reliable. The following figure shows an example of the floppy disk. Figure 3 shows a picture of the floppy disk.

- **Hard Disk**: Another form of auxiliary storage is a hard disk. A hard disk consists of one or more rigid metal plates coated with a metal oxide material that allows data to be magnetically recorded on the surface of the platters. The hard disk platters spin at 5 a high rate of speed, typically 5400 to 7200 revolutions per minute (RPM).Storage capacities of hard disks for personal computers range from 10 GB to 120 GB (one billion bytes are called a gigabyte).

- **Optical Disks:** Optical Mass Storage Devices Store bit values as variations in light reflection. They have higher area density & longer data life than magnetic storage. They are also standardized and relatively inexpensive. Their Uses: read-only storage with low performance requirements, applications with high capacity requirements & where portability in a standardized format is needed.

Types of Optical Disk

1. CD-ROM (read only)
2. CD-R: (record) to a CD
3. CD-RW: can write and erase CD to reuse it (re-writable)
4. DVD(Digital Video Disk)

2. Explain I/O devices and their Interfacing

Ans. Input / Output (I/O)

- MPU communicates with outside word through I/O device.
- There are 2 different methods by which MPU identifies and communicates With I/O devices these methods are:
  1- Direct I/O (Peripheral)
  2- Memory-Mapped I/O

The methods differ in terms of the

- No. of address lines used in identifying an I/O device.
- Type of control lines used to enable the device.
- Instructions used for data transfer.

Direct I/O (Peripheral):-

- This method uses two instructions (IN & OUT) for data transfer.
- MPU uses 8 address lines to send the address of I/O device (can identify 256 input devices & 256 output devices).
- The (I/P & O/P devices) can be differentiated by control signals I/O Read (IOR) and I/O Write (IOW).
- The steps in communicating with an I/O device are similar to those in communicating with memory and can be summarized as follows:
  1- The MPU places an 8-bit device address on address bus then decoded.
  2- The MPU sends a control signal (IOR or IOW) to enable the I/O device.
  3- Data are placed on the data bus for transfer.

Memory-Mapped I/O:-

- The MPU uses 16 address lines to identify an I/O device.
- This is similar to communicating with a memory location.
- Use the same control signals (MEMR or MEMW) and instructions as those of memory.
- The MPU views these I/O devices as if they were memory locations.
- There are no special I/O instructions.
- It can identify 64k address shared between memory & I/O devices.

1. Write down main features of 8085 microprocessor.

- It is an 8 bit microprocessor.
- It is manufactured with N-MOS technology.
- It has 16-bit address bus and hence can address up to 216 = 65536 bytes (64KB) memory locations through A0-A15
- The first 8 lines of address bus and 8 lines of data bus are multiplexed AD0 – AD7
- Data bus is a group of 8 lines D0 – D7
- It supports external interrupt request. .
- A 16 bit program counters (PC)
- A 16 bit stack pointer (SP)
- Six 8-bit general purpose register arranged in pairs: BC, DE, HL.
- It requires a signal +5V power supply and operates at 3.2 MHZ single phase clock.
- It is enclosed with 40 pins DIP (Dual in line package).

2. Explain 8085 microprocessor architecture.

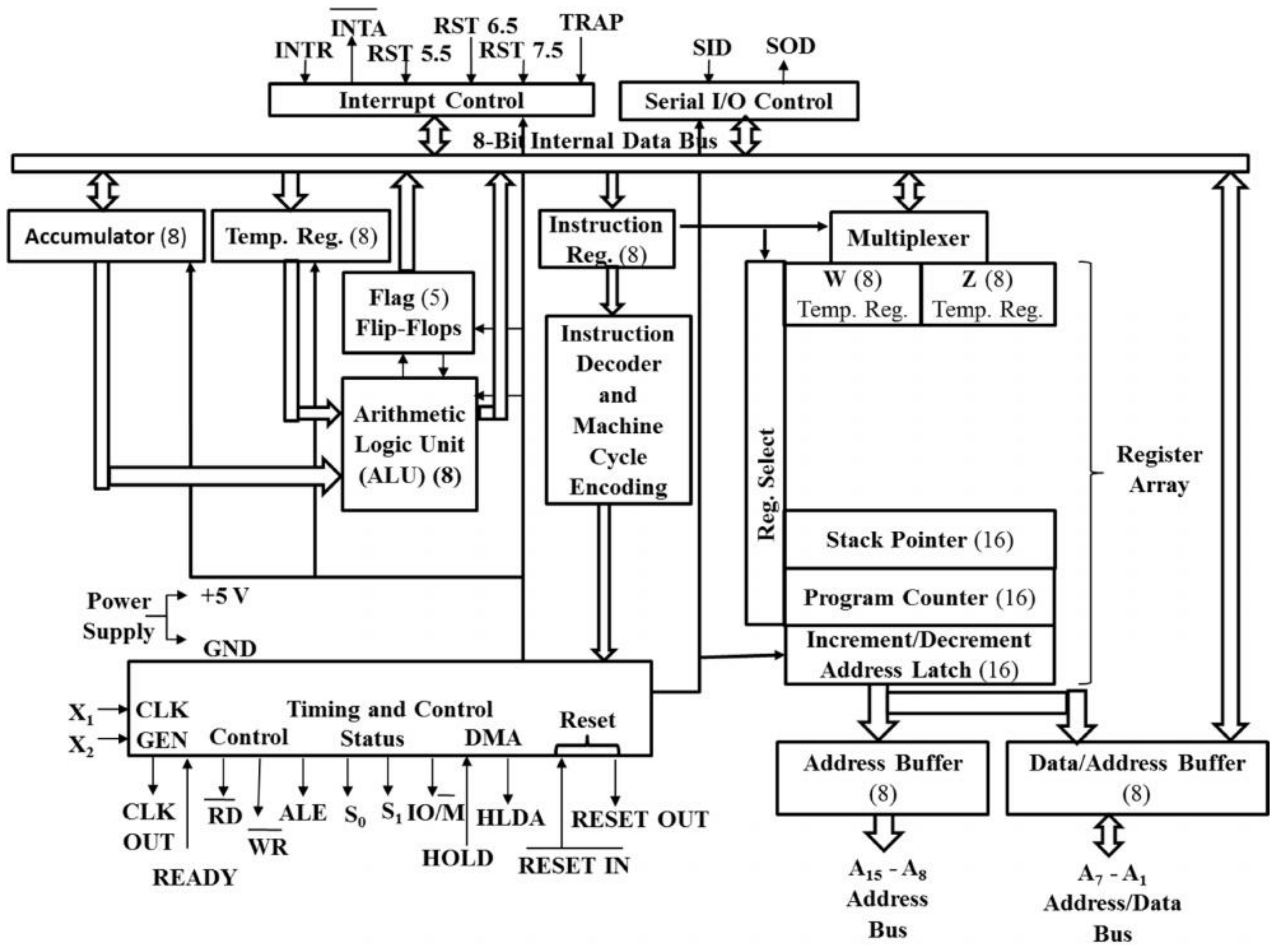


Figure: 8085 microprocessor architecture.

- The architecture of microprocessor 8085 can be divided into seven parts as follows:

Register Unit:

General Purpose Data Register

- 8085 has six general purpose data registers to store 8-bit data.
- These registers are named as B, C, D, E, H and L as shown in fig. 1.
- The user can use these registers to store or copy a data temporarily during the execution of a program by using data transfer instructions.
- These registers are of 8 bits but whenever the microprocessor has to handle 16-bit data, these registers can be combined as register pairs – BC, DE and HL.
- There are two internal registers – W and X. These registers are only for internal operation like execution of CALL and XCHG instructions and not available to the user.

Program Counter (PC)

- 16-bit register deals with sequencing the execution of instructions.
- This register is a memory pointer.
- Memory locations have 16-bit addresses which are why this is a 16-bit register.
- The microprocessor uses this register to sequence the execution of the instructions.
- The function of the program counter is to point to the memory address from which the next byte is to be fetched.
- When a byte (machine code) is being fetched, the program counter is incremented by one to point to the next memory location.

Stack Pointer (SP)

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

MUX/DEMUX unit

- This unit is used to select a register out of all the available registers.
- This unit behaves as a MUX when data is going from the register to the internal data bus.
- It behaves as a DEMUX when data is coming to a register from the internal data bus of the microprocessor.
- The register select will behave as the function selection lines of the MUX/DEMUX.

Address Buffer Register & Data/Address Buffer Register

- These registers hold the address/data, received from PC/internal data bus and then load the external address and data buses.
- These registers actually behave as the buffer stage between the microprocessor and external system buses.

Control Unit:

- The control unit generates signals within microprocessor to carry out the instruction, which has been decoded.
- In reality it causes connections between blocks of the microprocessor to be opened or closed, so that the data goes where it is required and the ALU operations occur.
- The control unit itself consists of three parts; the instruction registers (IR), instruction decoder and machine cycle encoder and timing and control unit.

Instruction Register

- This register holds the machine code of the instruction.
- When microprocessor executes a program it reads the opcode from the memory, this opcode is stored in the instruction register.

Instruction Decoder & Machine Cycle Encoder

- The IR sends the machine code to this unit.
- This unit, as its name suggests, decodes the opcode and finds out what is to be done in response of the coming opcode and how many machine cycles are required to execute this instruction.

Timing & Control unit

- The control unit generates signals within microprocessor to carry out the instruction, which has been decoded.
- In reality, it causes certain connections between blocks of the microprocessor to be opened or closed, so that the data goes where it is required and the ALU operations occur.

Arithmetic & Logical Unit:

- The ALU performs the actual numerical and logical operation such as 'add', 'subtract', 'AND', 'OR', etc.
- ALU uses data from memory and from accumulator to perform the arithmetic operations and always stores the result of the operation in accumulator.
- ALU consists of accumulator, flag register and temporary register.

Accumulator

- The accumulator is an 8-bit register that is a part of ALU.
- This register is used to store 8-bit data and perform arithmetical and logical operations.
- The result of an operation is stored in the accumulator.
- It is also identified as register A.

Flags register

- Flag register includes five flip-flops, which are set or reset after an operation according to the data conditions of the result in the accumulator and other registers.
- They are called zero (Z), carry (CY), sign (S), parity (P) and auxiliary carry (AC) flags; their bit positions in the flag register are shown in fig.
- The microprocessor uses these flags to set and test data conditions.

Interrupt Control

- The interrupt control unit has 5 interrupt inputs TRAP,RST 7.5, RST 6.5, RST 5.5 & INTR and one acknowledge signal INTA.

- It controls the interrupt activity of 8085 microprocessor.

Serial IO control

- 8085 serial IO control provides two lines, SOD and SID for serial communication.
- The serial output data (SOD) line is used to send data serially and serial input data line (SID) is used to receive data serially.

3. Explain Flags Registers in 8085

- Flag register includes five flip-flops, which are set or reset after an operation according to the data conditions of the result in the accumulator and other registers.
- They are called zero (Z), carry (CY), sign (S), parity (P) and auxiliary carry (AC) flags; their bit positions in the flag register are shown in fig.
- The microprocessor uses these flags to set and test data conditions.

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

Figure: Flags registers in 8085.

- The flags are stored in the 8-bit register so that the programmer can examine these flags by accessing the register through an instruction.
- These flags have critical importance in the decision-making process of the microprocessor.
- The conditions (set or reset) of the flags are tested through the software instructions.
- For instance, JC (jump on carry) is implemented to change the sequence of a program when CY flag is set.

Z (Zero) Flag:

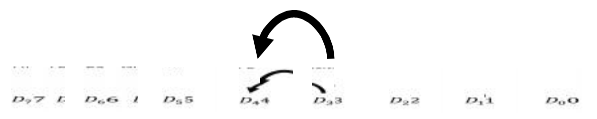
- This flag indicates whether the result of mathematical or logical operation is zero or not.
- If the result of the current operation is zero, then this flag will be set, otherwise reset.

CY (Carry) Flag:

- This flag indicates, whether, during an addition or subtraction operation, carry or borrow is generated or not, if generated then this flag bit will be set.

AC (Auxiliary Carry) Flag:

- It shows carry propagation from D3 position to D4 position.



D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	0	0	0	1	1	0	0
0	0	1	0	1	0	1	1
1	0	1	1	0	1	1	1

Figure: Auxiliary Carry.

- As shown in the fig., a carry is generated from D3 bit position and propagates to the D4 position. This carry is called auxiliary carry.

S (Sign) Flag:

- Sign flag indicates whether the result of a mathematical operation is negative or positive.
- If the result is positive, then this flag will reset and if the result is negative this flag will be set.
- This bit, in fact, is a replica of the D7 bit.

P (Parity) Flag:

- Parity is the number of 1's in a number.
- If the number of 1's in a number is even then that number is known as even parity number.
- If the number of 1's in a number is odd then that number is known as an odd parity number.
- This flag indicates whether the current result is of even parity (set) or of odd parity (reset).

4. Explain 8085 pin diagram.

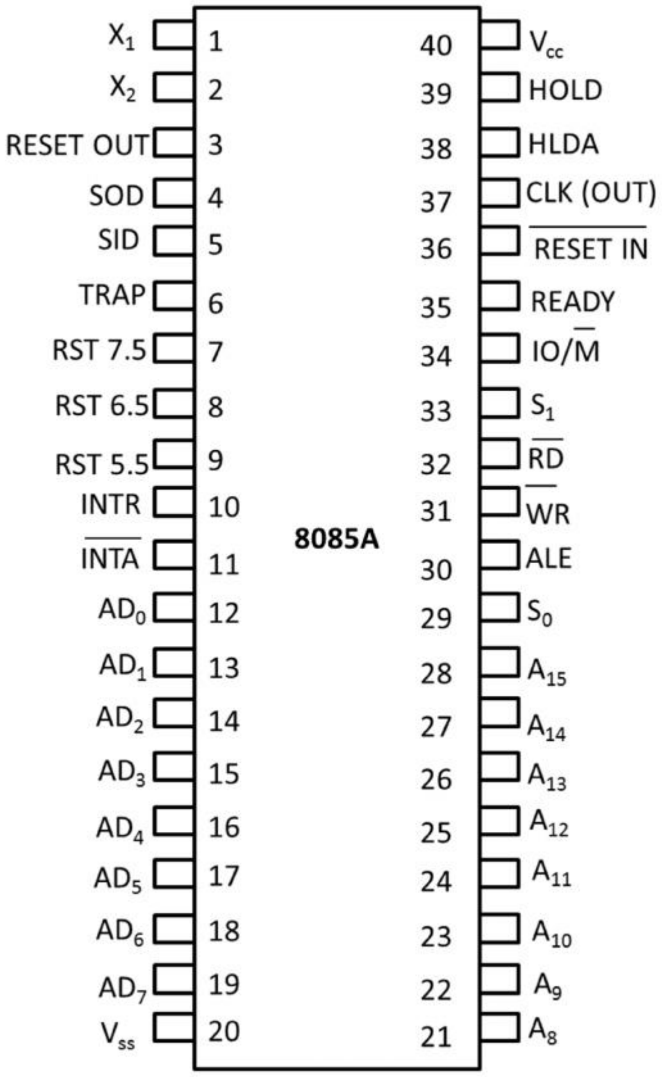


Figure: 8085 pin diagram.

- All signals can be classified into six groups:
  1. Address Bus
  2. Data Bus
  3. Control & Status Signals
  4. Power Supply & Frequency signals
  5. Externally initiated signals
  6. Serial I/O Ports

1) Address Bus (pin 12 to 28)

- 16 signal lines are used as address bus.
- However these lines are split into two segments: A_{15} - A_8 and AD_7 - AD_0
- A_{15} - A_8 are unidirectional and are used to carry high-order address of 16-bit address.
- AD_7 - AD_0 are used for dual purpose.

2) Data Bus/ Multiplexed Address (pin 12 to 19)

- Signal lines AD7-AD0 are bidirectional and serve dual purpose.
- They are used as low-order address bus as well as data bus.
- The low order address bus can be separate from these signals by using a latch.

3) Control & Status Signals

- To identify nature of operation
- Two Control Signals
  1) RD' (Read-pin 32)
     - ✓ This is a read control signal (active low)
     - ✓ This signal indicates that the selected I/O or Memory device is to be read & data are available on data bus.
  2) WR' (Write-pin 31)
     - ✓ This is a write control signal (active low)
     - ✓ This signal indicates that the selected I/O or Memory device is to be write.
- Three Status Signals
  1) S_1 (pin 33)
  2) S_0 (pin 29)
     - ✓ S_1 and S_0 status signals can identify various operations, but they are rarely used in small systems.

S_1	S_0	Mode
0	0	HLT
0	1	WRITE
1	0	READ
1	1	OPCODE FETCH

  3) IO/M' (pin 34)
     - ✓ This is a status signal used to differentiate I/O and memory operation
     - ✓ When it is high, it indicates an I/O operation
     - ✓ When it is low, it indicates a memory operation
     - ✓ This signal is combined with RD' and WR' to generate I/O & memory control signals
- To indicate beginning of operation
  - o One Special Signal called ALE (Address Latch Enable-Pin 30)
  - o This is positive going pulse generated every time the 8085 begins an operation (machine cycle)
  - o It indicates that the bits on AD7-AD0 are address bits
  - o This signal is used primarily to latch the low-address from multiplexed bus & generate a separate set of address lines A7-A0.

4) Power Supply & Frequency Signal

- V_{cc}→Pin no. 40, +5V Supply
- V_{ss} → Pin no.20, Ground Reference
- X1, X2 → Pin no.1 & 2, Crystal Oscillator is connected at these two pins. The frequency is internally divided by two;
  - o Therefore, to operate a system at 3MHz, the crystal should have a frequency of 6MHz.
- CLK (OUT) → Clock output. Pin No.37: This signal can be used as the system clock for other devices.

5) Externally Initiated Signals including Interrupts

- INTR (Input) → Interrupt Request. It is used as general purpose interrupt
- INTA' (Output) → Interrupt Acknowledge. It is used to acknowledge an interrupt.
- RST7.5, RST6.5, RST5.5 (Input) → Restart Interrupts.
  - o These are vector interrupts that transfer the program control to specific memory locations.
  - o They have higher priorities than INTR interrupt.
  - o Among these 3 interrupts, the priority order is RST7.5, RST6.5, RST5.5
- TRAP (Input) → This is a non maskable interrupt & has the highest priority.
- HOLD (Input) → This signal indicates that a peripheral such as DMA Controller is requesting the use of address & data buses
- HLDA (Output) → Hold Acknowledge. This signal acknowledges the HOLD request
- READY (Input) → This signal is used to delay the microprocessor read or write cycles until as low-responding peripheral is ready to send or accept data. When the signal goes low, the microprocessor waits for an integral no. of clock cycles until it goes high.
- RESET IN' (Input) → When the signal on this pin goes low, the Program Counter is set to zero, the buses are tri-stated & microprocessor is reset.
- RESET OUT (Output) → This signal indicates that microprocessor is being reset. The signal can be used to reset other devices.

6) Serial I/O Ports

- Two pins for serial transmission
  1) SID (Serial Input Data-pin 5)
  2) SOD (Serial Output Data-pin 4)
- In serial transmission, data bits are sent over a single line, one bit at a time.

5. Explain Instruction Cycle

- Instruction Cycle is defined as time required to complete execution of an instruction.
- 8085 instruction cycle consists of 1 to 6 Machine Cycles or 1 to 6 operations.

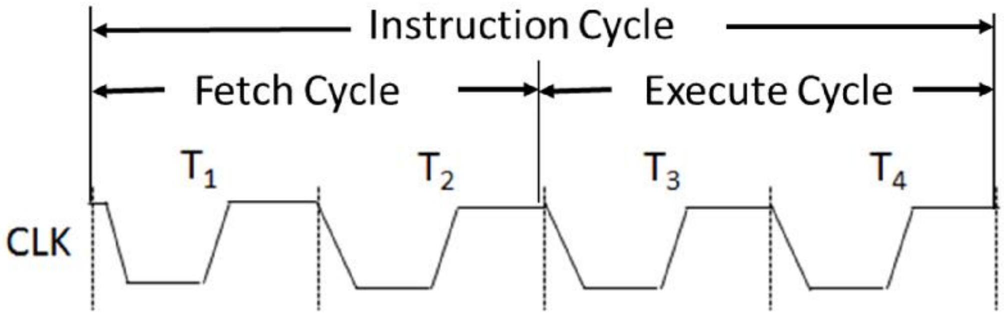


Figure: Instruction Cycle.

6. Explain Machine Cycle

- Machine Cycle is defined as time required by the microprocessor to complete operation of accessing memory device or I/O device.
- This cycle may consist 3 to 6 T-states.
- The basic microprocessor operation such as reading a byte from I/O port or writing a byte to memory is called as machine cycle.

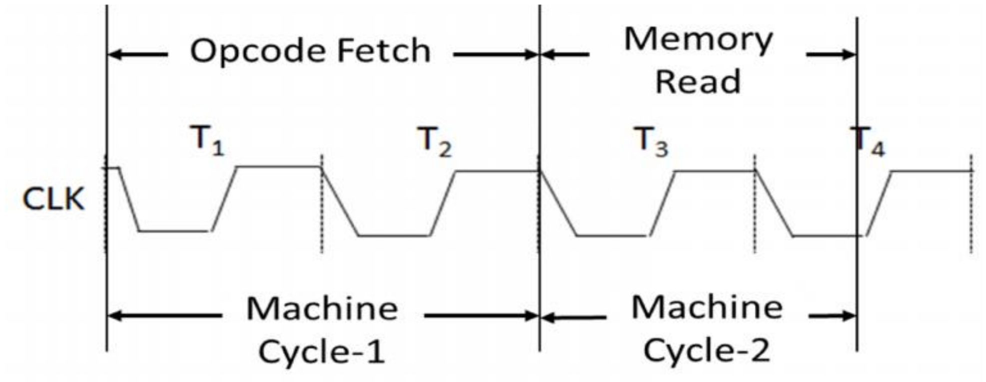


Figure: Machine Cycle.

7. Explain T-States

- T-States are defined as one subdivision of operation performed in one clock period.
- These sub divisions are internal states synchronized with system clock & each T-state is precisely equal to one clock period.

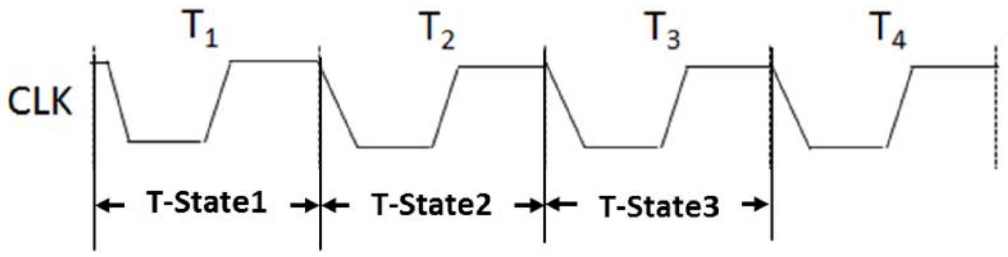


Figure: T-States.

8. Compare Instruction Cycle, Machine Cycle and T-States

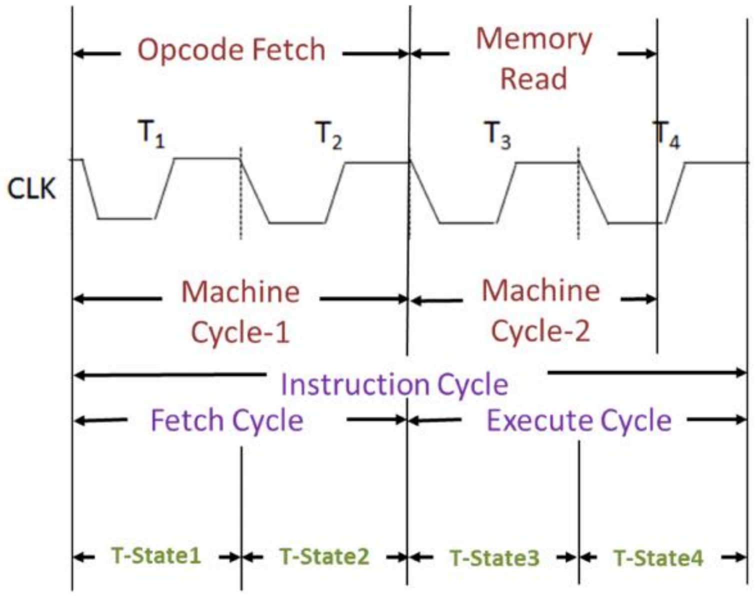


Figure: Comparison between Instruction Cycle, Machine Cycle and T-States.

- Instruction Cycle: Time required to complete execution of an instruction.
- Machine Cycle: Time required by the microprocessor to complete an operation.
- T-States: One subdivision of operation performed in one clock period.

9. Explain 8085 Programming Model

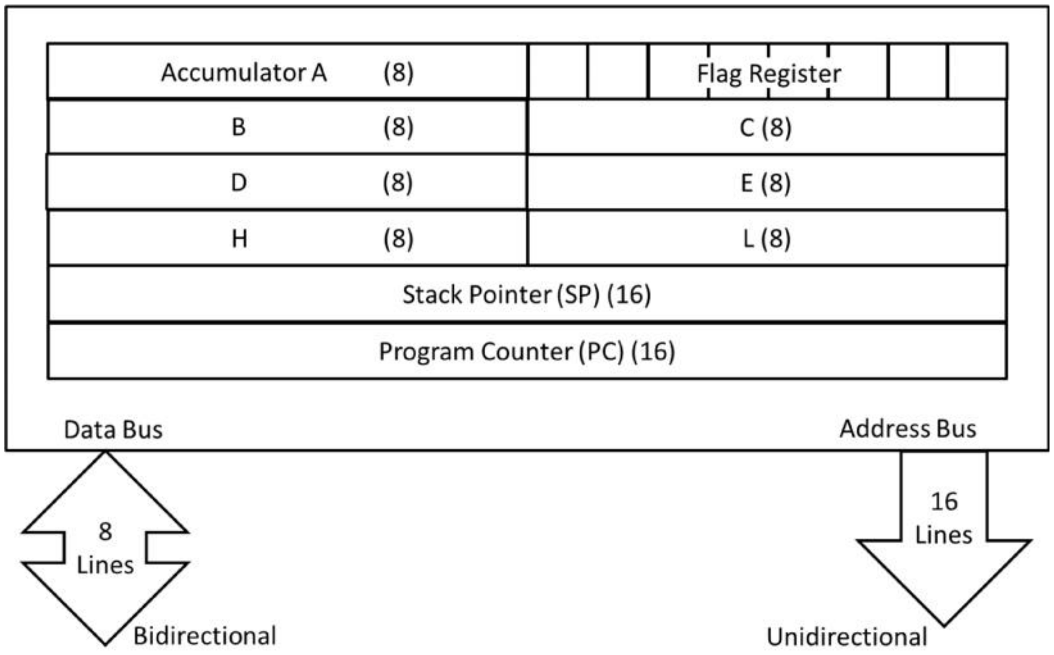


Figure: 8085 Programming Model.

Registers

- 6 general purpose registers to store 8-bit data B, C, D, E, H & L.
- Can be combined as register pairs – BC, DE, and HL to perform 16-bit operations.
- Used to store or copy data using data copy instructions.

Accumulator

- 8 - bit register, identified as A
- Part of ALU
- Used to store 8-bit data to perform arithmetic & logical operations.
- Result of operation is stored in it.

Flag Register

- ALU has 5 Flag Register that set/reset after an operation according to data conditions of the result in accumulator & other registers.
- Helpful in decision making process of Microprocessor
- Conditions are tested through software instructions
- For e.g.
- JC (Jump on Carry) is implemented to change the sequence of program when CY is set.

Program Counter

- 16-bit registers used to hold memory addresses.
- Size is 16-bits because memory addresses are of 16-bits.

- Microprocessor uses PC register to sequence the execution of instructions.
- Its function is to point to memory address from which next byte is to be fetched.
- When a byte is being fetched, PC is incremented by 1 to point to next memory location.

Stack Pointer

- Used as memory pointer
- Points to the memory location in R/W memory, called Stack.
- Beginning of stack is defined by loading a 16-bit address in the stack pointer.

10. Explain Bus Organization of 8085

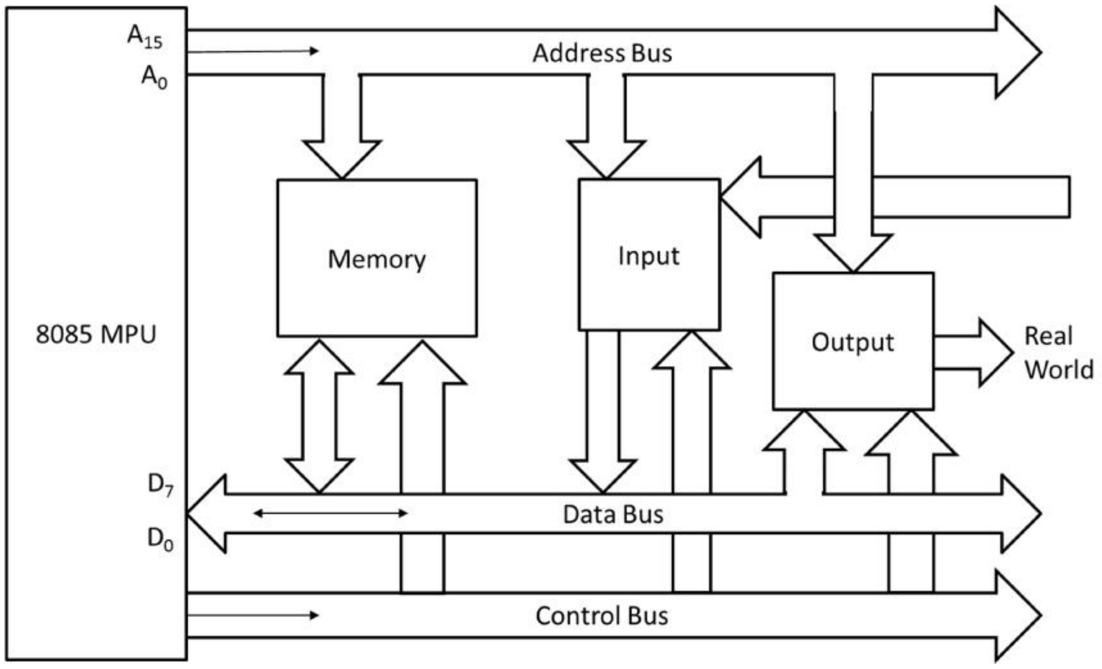


Figure: Bus Organization of 8085.

Address Bus

- Group of 16 lines generally identified as A0 to A15.
- It is unidirectional i.e. bits flow from microprocessor to peripheral devices.
- 16 address lines are capable of addressing 65536 memory locations.
- So, 8085 has 64K memory locations.

Data Bus

- Group of 8 lines identified as D0 to D7.
- They are bidirectional i.e. data flow in both directions between microprocessor, memory & peripheral.
- 8 data lines enable microprocessor to manipulate data ranging from 00H to FFH (28=256 numbers).
- Largest number appear on data bus is 1111 1111 => (255)10.
- As Data bus is of 8-bit, 8085 is known as 8-bit Microprocessor.

Control Bus

- It comprises of various single lines that carry synchronization, timing & control signals.

- These signals are used to identify a device type with which MPU intends to communicate.

11. Explain Demultiplexing AD0-AD7

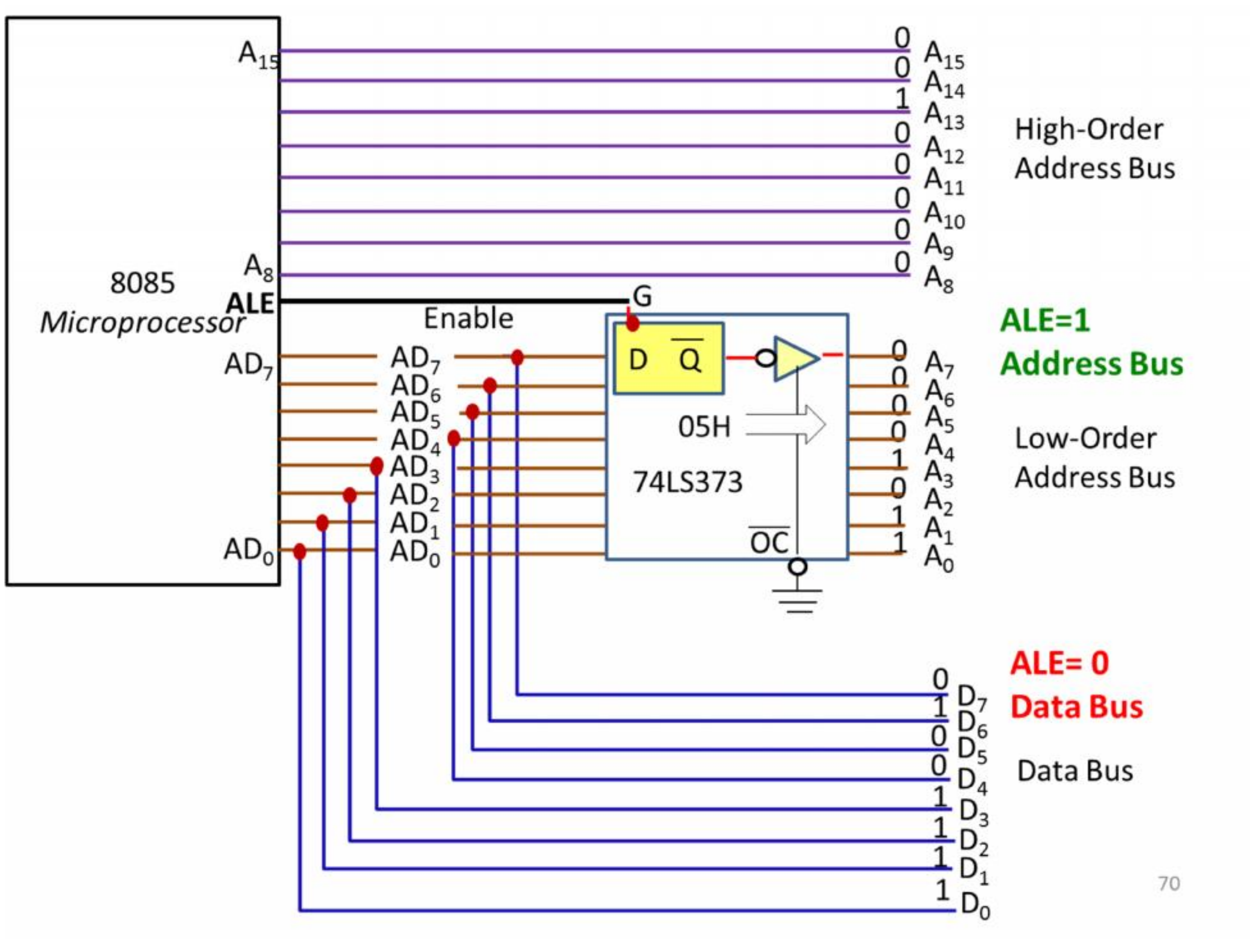


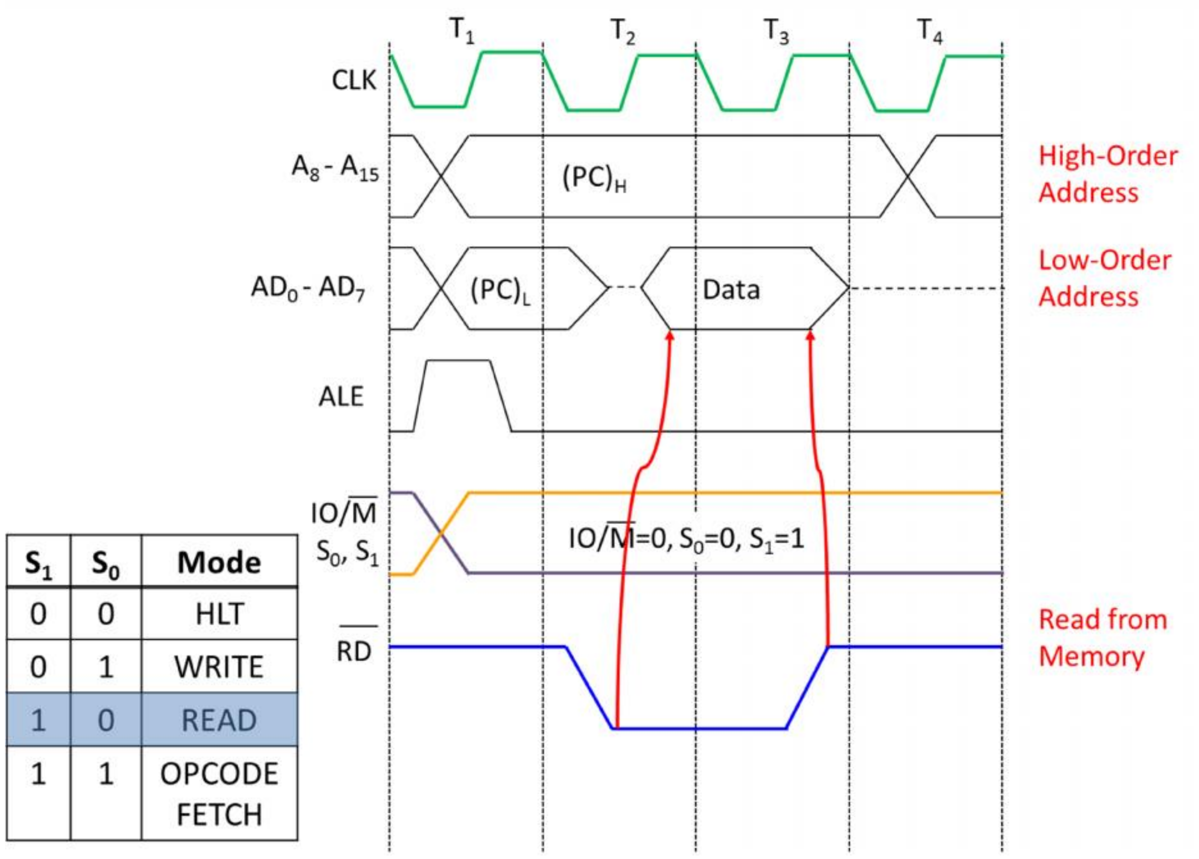
Figure: Demultiplexing AD0-AD7.

- The higher-order bus remains on the bus for three clock periods. However, the low-order address is lost after the first clock period.
- This address need to be latched and used for identifying the memory address. If the bus AD7-AD0 is used to identify the memory location (2005H), the address will change to 204FH after the first clock period.
- Figure shows a schematic that uses a latch and the ALE signal to demultiplex the bus.
- The bus AD7-AD0 is connected as the input to the latch.
- The ALE signal is connected to the Enable pin of the latch, and the output control signal of the latch is grounded.
- Figure shows that the ALE goes high during T1. And during T1 address of lower-order address bus is store into the latch.

12. Explain Memory Interfacing

- When we are executing any instruction, we need the microprocessor to access the memory for reading instruction codes and the data stored in the memory.
- For this, both the memory and the microprocessor requires some signals to read/write to/from registers.
- The interfacing circuit therefore should be designed in such a way that it matches the memory signal requirements with the signals of the microprocessor.

Memory Read Cycle

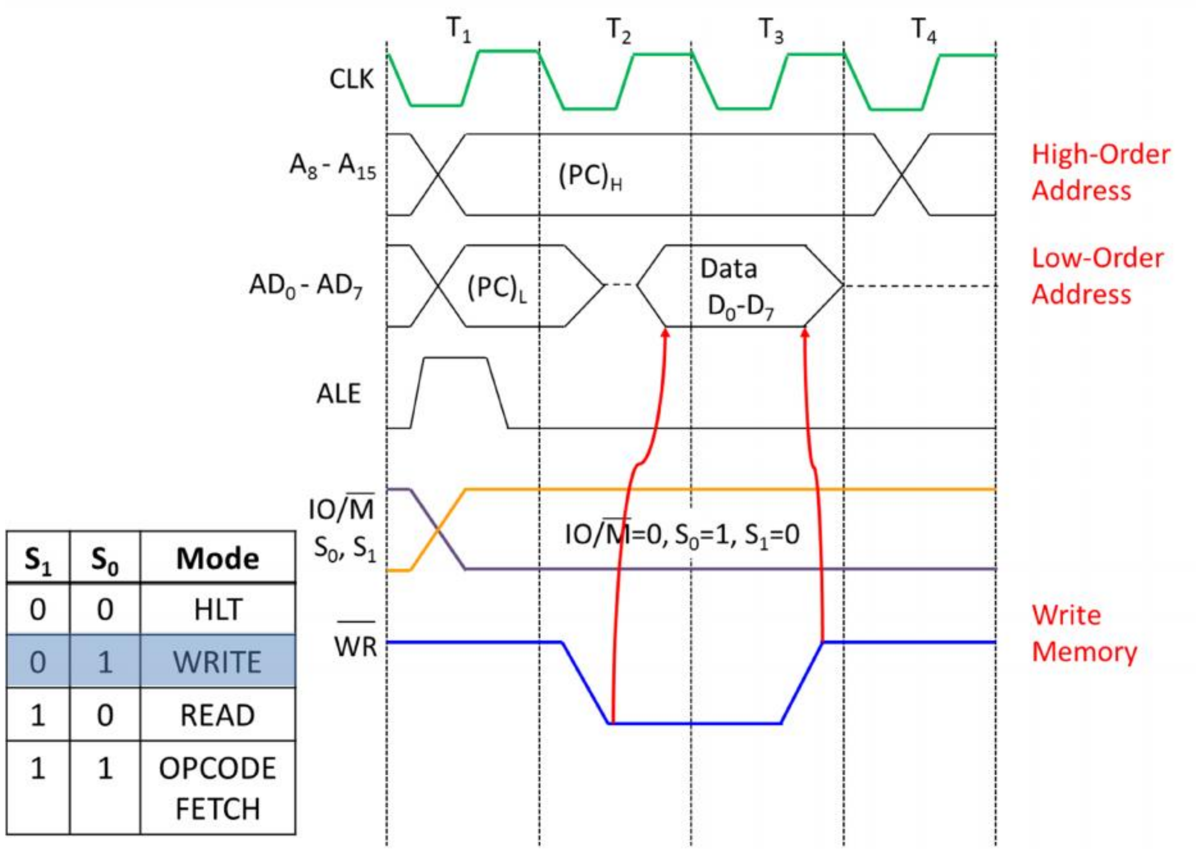


S_1	S_0	Mode
0	0	HLT
0	1	WRITE
1	0	READ
1	1	OPCODE FETCH

Figure: Memory Read Cycle.

- It is used to fetch one byte from the memory.
- It requires 3 T-States.
- It can be used to fetch operand or data from the memory.
- During T1, A8-A15 contains higher byte of address. At the same time ALE is high. Therefore Lower byte of address A0-A7 is selected from AD0-AD7.
- Since it is memory ready operation, IO/M (bar) goes low.
- During T2 ALE goes low, RD (bar) goes low. Address is removed from AD0-AD7 and data D0-D7 appears on AD0-AD7.
- During T3, Data remains on AD0-AD7 till RD (bar) is at low signal.

Memory Write Cycle



S1	S0	Mode
0	0	HLT
0	1	WRITE
1	0	READ
1	1	OPCODE FETCH

Figure: Memory Write Cycle.

- It is used to send one byte into memory.
- It requires 3 T-States.
- During T1, ALE is high and contains lower address A0-A7 from AD0-AD7.
- A8-A15 contains higher byte of address.
- As it is memory operation, IO/M (bar) goes low.
- During T2, ALE goes low, WR (bar) goes low and Address is removed from AD0-AD7 and then data appears on AD0-AD7.
- Data remains on AD0-AD7 till WR (bar) is low.

13. Explain how Control Signals Generated in 8085

Operation	IO/M’	RD’	WR’
MEMR’	0	0	X
MEMW’	0	X	0
IOR’	1	0	X
IOW’	1	X	0

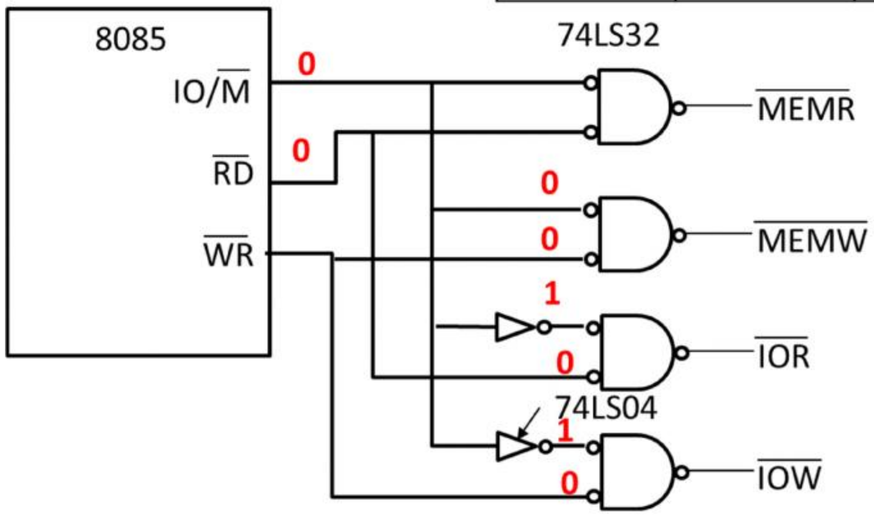


Figure: Control Signals Generated in 8085.

- Figure shows that four different control signals are generated by combining the signals RD (bar), WR (bar), and IO/M (bar).
- The signal IO/M (bar) goes low for the memory operation. This signal is ANDed with RD (bar) and WR (bar) signals b using the 74LS32 quadruple two-input OR gates, as shown in figure 4.5.
- The OR gates are functionally connected as negative NAND gates. When both input signals go low, the output of the gates go low and generate MEMR (bar) and MEMW (bar) control signals.
- When the IO/M (bar) signal goes high, it indicates the peripheral I/O operation.
- Figure shows that this signal is complemented using the Hex inverter 74LS04 and ANDed with the RD (bar) and WR (bar) signals to generate IOR (bar) and IOW (bar) control signals.

1. 8085 instruction set.

Sr.	Instruction	Description	Example
DATA TRANSFER INSTRUCTIONS			
1.	MOV R_d, R_s MOV M, R_s MOV R_s, M	This instruction copies the contents of the source register into the destination register; the contents of the source register are not altered. If one of the operands is a memory location, its location is specified by the contents of the HL registers.	MOV B, C MOV B, M
2.	MVI Rd, data MVI M, data	The 8-bit data is stored in the destination register or memory. If the operand is a memory location, its location is specified by the contents of the HL registers.	MVI B, 57H MVI M, 57H
3.	LDA 16-bit address	The contents of a memory location, specified by a 16-bit address in the operand, are copied to the accumulator. The contents of the source are not altered.	LDA 2034H
4.	LDAX B/D Reg. pair	The contents of the designated register pair point to a memory location. This instruction copies the contents of that memory location into the accumulator. The contents of either the register pair or the memory location are not altered.	LDAX B
5.	LXI Reg.-pair, 16-bit data	The instruction loads 16-bit data in the register pair designated in the operand.	LXI H, 2034H LXI H, XYZ
6.	LHLD 16-bit address	The instruction copies the contents of the memory location pointed out by the 16-bit address into register L and copies the contents of the next memory location into register H. The contents of source memory locations are not altered.	LHLD 2040H
7.	STA 16-bit address	The contents of the accumulator are copied into the memory location specified by the operand. This is a 3-byte instruction, the second byte specifies the low-order address and the third byte specifies the high-order address.	STA 4350H
8.	STAX Reg. pair	The contents of the accumulator are copied into the memory location specified by the contents of the operand (register pair). The contents of the accumulator are not altered.	STAX B

Sr.	Instruction	Description	Example
9.	SHLD 16-bit address	The contents of register L are stored into the memory location specified by the 16-bit address in the operand and the contents of H register are stored into the next memory location by incrementing the operand. The contents of registers HL are not altered. This is a 3-byte instruction, the second byte specifies the low-order address and the third byte specifies the high-order address.	SHLD 2470H
10.	XCHG	The contents of register H are exchanged with the contents of register D, and the contents of register L are exchanged with the contents of register E.	XCHG
11.	SPHL	The instruction loads the contents of the H and L registers into the stack pointer register, the contents of the H register provide the high-order address and the contents of the L register provide the low-order address. The contents of the H and L registers are not altered.	SPHL
12.	XTHL	The contents of the L register are exchanged with the stack location pointed out by the contents of the stack pointer register. The contents of the H register are exchanged with the next stack location (SP+1); however, the contents of the stack pointer register are not altered.	XTHL
13.	PUSH Reg. pair	The contents of the register pair designated in the operand are copied onto the stack in the following sequence. The stack pointer register is decremented and the contents of the high order register (B, D, H, A) are copied into that location. The stack pointer register is decremented again and the contents of the low-order register (C, E, L, flags) are copied to that location.	PUSH B PUSH A
14.	POP Reg. pair	The contents of the memory location pointed out by the stack pointer register are copied to the low-order register (C, E, L, status flags) of the operand. The stack pointer is incremented by 1 and the contents of that memory location are copied to the high-order register (B, D, H, A) of the operand. The stack pointer register is again incremented by 1.	POP H POP A
15.	OUT 8-bit port address	The contents of the accumulator are copied into the I/O port specified by the operand.	OUT F8H
16.	IN 8-bit port address	The contents of the input port designated in the operand are read and loaded into the accumulator.	IN 8CH

ARITHMETIC INSTRUCTIONS

Sr.	Instruction	Description	Example
17.	ADD R ADD M	The contents of the operand (register or memory) are added to the contents of the accumulator and the result is stored in the accumulator. If the operand is a memory location, its location is specified by the contents of the HL registers. All flags are modified to reflect the result of the addition.	ADD B ADD M
18.	ADC R ADC M	The contents of the operand (register or memory) and the Carry flag are added to the contents of the accumulator and the result is stored in the accumulator. If the operand is a memory location, its location is specified by the contents of the HL registers. All flags are modified to reflect the result of the addition.	ADC B ADC M
19.	ADI 8-bit data	The 8-bit data (operand) is added to the contents of the accumulator and the result is stored in the accumulator. All flags are modified to reflect the result of the addition.	ADI 45H
20.	ACI 8-bit data	The 8-bit data (operand) and the Carry flag are added to the contents of the accumulator and the result is stored in the accumulator. All flags are modified to reflect the result of the addition.	ACI 45H
21.	DAD Reg. pair	The 16-bit contents of the specified register pair are added to the contents of the HL register and the sum is stored in the HL register. The contents of the source register pair are not altered. If the result is larger than 16 bits, the CY flag is set. No other flags are affected.	DAD H
22.	SUB R SUB M	The contents of the operand (register or memory) are subtracted from the contents of the accumulator, and the result is stored in the accumulator. If the operand is a memory location, its location is specified by the contents of the HL registers. All flags are modified to reflect the result of the subtraction.	SUB B SUB M
23.	SBB R SBB M	The contents of the operand (register or memory) and the Borrow flag are subtracted from the contents of the accumulator and the result is placed in the accumulator. If the operand is a memory location, its location is specified by the contents of the HL registers. All flags are modified to reflect the result of the subtraction.	SBB B SBB M

Sr.	Instruction	Description	Example
24.	SUI 8-bit data	The 8-bit data (operand) is subtracted from the contents of the accumulator and the result is stored in the accumulator. All flags are modified to reflect the result of the subtraction.	SUI 45H
25.	SBI 8-bit data	The 8-bit data (operand) and the Borrow flag are subtracted from the contents of the accumulator and the result is stored in the accumulator. All flags are modified to reflect the result of the subtraction.	SBI 45H
26.	INR R INR M	The contents of the designated register or memory are incremented by 1 and the result is stored in the same place. If the operand is a memory location, its location is specified by the contents of the HL registers.	INR B INR M
27.	INX R	The contents of the designated register pair are incremented by 1 and the result is stored in the same place.	INX H
28.	DCR R DCR M	The contents of the designated register or memory are decremented by 1 and the result is stored in the same place. If the operand is a memory location, its location is specified by the contents of the HL registers.	DCR B DCR M
29.	DCX R	The contents of the designated register pair are decremented by 1 and the result is stored in the same place.	DCX H
30.	DAA	The contents of the accumulator are changed from a binary value to two 4-bit binary coded decimal (BCD) digits. This is the only instruction that uses the auxiliary flag to perform the binary to BCD conversion, and the conversion procedure is described below. S, Z, AC, P, CY flags are altered to reflect the results of the operation. If the value of the low-order 4-bits in the accumulator is greater than 9 or if AC flag is set, the instruction adds 6 to the low-order four bits. If the value of the high-order 4-bits in the accumulator is greater than 9 or if the Carry flag is set, the instruction adds 6 to the high-order four bits.	DAA

Sr.	Instruction	Description	Example
BRANCHING INSTRUCTIONS			
31.	JMP 16-bit address	The program sequence is transferred to the memory location specified by the 16-bit address given in the operand.	JMP 2034H JMP XYZ
Jump conditionally		*The program sequence is transferred to the memory location specified by the 16-bit address given in the operand based on the specified flag of the PSW as described below.*	
32.	JC 16-bit address	Jump on Carry, Flag Status: CY=1	JC 2050H
33.	JNC 16-bit address	Jump on no Carry, Flag Status: CY=0	JNC 2050H
34.	JP 16-bit address	Jump on positive, Flag Status: S=0	JP 2050H
35.	JM 16-bit address	Jump on minus, Flag Status: S=1	JM 2050H
36.	JZ 16-bit address	Jump on zero, Flag Status: Z=1	JZ 2050H
37.	JNZ 16-bit address	Jump on no zero, Flag Status: Z=0	JNZ 2050H
38.	JPE 16-bit address	Jump on parity even, Flag Status: P=1	JPE 2050H
39.	JPO 16-bit address	Jump on parity odd, Flag Status: P=0	JPO 2050H
40.	CALL 16-bit address	The program sequence is transferred to the memory location specified by the 16-bit address given in the operand. Before the transfer, the address of the next instruction after CALL (the contents of the program counter) is pushed onto the stack.	CALL 2034H CALL XYZ
Call conditionally		*The program sequence is transferred to the memory location specified by the 16-bit address given in the operand based on the specified flag of the PSW as described below. Before the transfer, the address of the next instruction after the call (the contents of the program counter) is pushed onto the stack.*	
41.	CC 16-bit address	Call on Carry, Flag Status: CY=1	CC 2050H
42.	CNC 16-bit address	Call on no Carry, Flag Status: CY=0	CNC 2050H
43.	CP 16-bit address	Call on positive, Flag Status: S=0	CP 2050H
44.	CM 16-bit address	Call on minus, Flag Status: S=1	CM 2050H
45.	CZ 16-bit address	Call on zero, Flag Status: Z=1	CZ 2050H
46.	CNZ 16-bit address	Call on no zero, Flag Status: Z=0	CNZ 2050H
47.	CPE 16-bit address	Call on parity even, Flag Status: P=1	CPE 2050H
48.	CPO 16-bit address	Call on parity odd, Flag Status: P=0	CPO 2050H

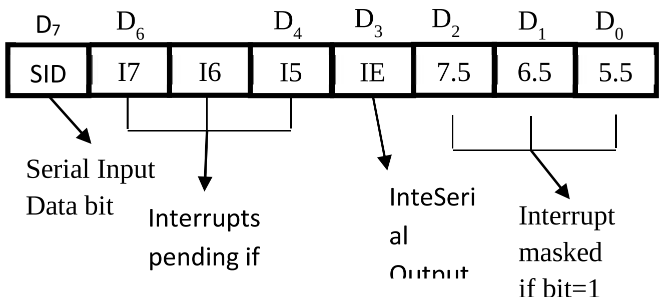
Sr.	Instruction	Description	Example
49.	RET	The program sequence is transferred from the subroutine to the calling program. The two bytes from the top of the stack are copied into the program counter, and program execution begins at the new address.	RET
	Return from subroutine conditionally	*The program sequence is transferred from the subroutine to the calling program based on the specified flag of the PSW as described below. The two bytes from the top of the stack are copied into the program counter, and program execution begins at the new address.*	
50.	RC	Return on Carry, Flag Status: CY=1	RC
51.	RNC	Return on no Carry, Flag Status: CY=0	RNC
52.	RP	Return on positive, Flag Status: S=0	RP
53.	RM	Return on minus, Flag Status: S=1	RM
54.	RZ	Return on zero, Flag Status: Z=1	RZ
55.	RNZ	Return on no zero, Flag Status: Z=0	RNZ
56.	RPE	Return on parity even, Flag Status: P=1	RPE
57.	RPO	Return on parity odd, Flag Status: P=0	RPO
58.	PCHL	The contents of registers H and L are copied into the program counter. The contents of H are placed as the high-order byte and the contents of L as the low-order byte.	PCHL
59.	RST 0-7	The RST instruction is equivalent to a 1-byte call instruction to one of eight memory locations depending upon the number. The instructions are generally used in conjunction with interrupts and inserted using external hardware. However these can be used as software instructions in a program to transfer program execution to one of the eight locations. The addresses are:	RST 3

Instruction	**Restart Address**
RST 0	0000H
RST 1	0008H
RST 2	0010H
RST 3	0018H
RST 4	0020H
RST 5	0028H
RST 6	0030H
RST 7	0038H

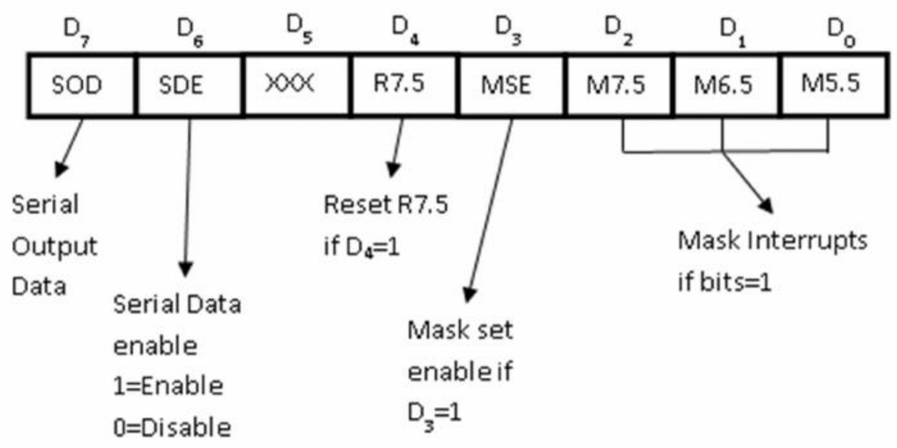
Sr.	Instruction	Description	Example
The 8085 has four additional interrupts and these interrupts generate RST instructions internally and thus do not require any external hardware.			
60.	TRAP	It restart from address 0024H	TRAP
61.	RST 5.5	It restart from address 002CH	RST 5.5
62.	RST 6.5	It restart from address 0034H	RST 6.5
63.	RST 7.5	It restart from address 003CH	RST 7.5
LOGICAL INSTRUCTIONS			
64.	CMP R CMP M	The contents of the operand (register or memory) are compared with the contents of the accumulator. Both contents are preserved. The result of the comparison is shown by setting the flags of the PSW as follows: if (A) < (reg/mem): carry flag is set if (A) = (reg/mem): zero flag is set if (A) > (reg/mem): carry and zero flags are reset	CMP B CMP M
65.	CPI 8-bit data	The second byte (8-bit data) is compared with the contents of the accumulator. The values being compared remain unchanged. The result of the comparison is shown by setting the flags of the PSW as follows: if (A) < data: carry flag is set if (A) = data: zero flag is set if (A) > data: carry and zero flags are reset	CPI 89H
66.	ANA R ANA M	The contents of the accumulator are logically ANDed with the contents of the operand (register or memory), and the result is placed in the accumulator. If the operand is a memory location, its address is specified by the contents of HL registers. S, Z, P are modified to reflect the result of the operation. CY is reset. AC is set.	ANA B ANA M
67.	ANI 8-bit data	The contents of the accumulator are logically ANDed with the 8-bit data (operand) and the result is placed in the accumulator. S, Z, P are modified to reflect the result of the operation. CY is reset. AC is set.	ANI 86H
68.	XRA R XRA M	The contents of the accumulator are Exclusive ORed with the contents of the operand (register or memory), and the result is placed in the accumulator. If the operand is a memory location, its address is specified by the contents of HL registers. S, Z, P are modified to reflect the result of the operation. CY and AC are reset.	XRA B XRA M

Sr.	Instruction	Description	Example
69.	XRI 8-bit data	The contents of the accumulator are Exclusive ORed with the 8-bit data (operand) and the result is placed in the accumulator. S, Z, P are modified to reflect the result of the operation. CY and AC are reset.	XRI 86H
70.	ORA R ORA M	The contents of the accumulator are logically ORed with the contents of the operand (register or memory), and the result is placed in the accumulator. If the operand is a memory location, its address is specified by the contents of HL registers. S, Z, P are modified to reflect the result of the operation. CY and AC are reset.	ORA B ORA M
71.	ORI 8-bit data	The contents of the accumulator are logically ORed with the 8-bit data (operand) and the result is placed in the accumulator. S, Z, P are modified to reflect the result of the operation. CY and AC are reset.	ORI 86H
72.	RLC	Each binary bit of the accumulator is rotated left by one position. Bit D7 is placed in the position of D0 as well as in the Carry flag. CY is modified according to bit D7. S, Z, P, AC are not affected.	RLC
73.	RRC	Each binary bit of the accumulator is rotated right by one position. Bit D0 is placed in the position of D7 as well as in the Carry flag. CY is modified according to bit D0. S, Z, P, AC are not affected.	RRC
74.	RAL	Each binary bit of the accumulator is rotated left by one position through the Carry flag. Bit D7 is placed in the Carry flag, and the Carry flag is placed in the least significant position D0. CY is modified according to bit D7. S, Z, P, AC are not affected.	RAL
75.	RAR	Each binary bit of the accumulator is rotated right by one position through the Carry flag. Bit D0 is placed in the Carry flag, and the Carry flag is placed in the most significant position D7. CY is modified according to bit D0. S, Z, P, AC are not affected.	RAR
76.	CMA	The contents of the accumulator are complemented. No flags are affected.	CMA
77.	CMC	The Carry flag is complemented. No other flags are affected.	CMC
78.	STC	The Carry flag is set to 1. No other flags are affected.	STC
CONTROL INSTRUCTIONS			
79.	NOP	No operation is performed. The instruction is fetched and decoded. However no operation is executed.	NOP

Sr.	Instruction	Description	Example
80.	HLT	The CPU finishes executing the current instruction and halts any further execution. An interrupt or reset is necessary to exit from the halt state.	HLT
81.	DI	The interrupt enable flip-flop is reset and all the interrupts except the TRAP are disabled. No flags are affected.	DI
82.	EI	The interrupt enable flip-flop is set and all interrupts are enabled. No flags are affected. After a system reset or the acknowledgement of an interrupt, the interrupt enable flip-flop is reset, thus disabling the interrupts. This instruction is necessary to re enable the interrupts (except TRAP).	EI
83.	RIM	This is a multipurpose instruction used to read the status of interrupts 7.5, 6.5, 5.5 and read serial data input bit. The instruction loads eight bits in the accumulator with the following interpretations.	RIM



Sr.	Instruction	Description	Example
84.	SIM	This is a multipurpose instruction and used to implement the 8085 interrupts 7.5, 6.5, 5.5, and serial data output. The instruction interprets the accumulator contents as follows.	SIM



2. Explain Addressing mode in 8085

1) Immediate Addressing Mode

- In this mode 8/16 bit data is specified in instruction itself as one of its operand.
- Example

  MVI B 20H ; 20H is copied into register B.

  LXI D 1000H ; 1000H is stored into DE register pair.

2) Direct Addressing Mode

- In this mode 8/16 bit address is directly specified in instruction itself as one of its operand.
- Example

  LDA 2000H ; 2000H is memory address.

  IN 08H ; 08H is port address.

  OUT 10H ; 10H is port address.

3) Register Addressing Mode

- In this mode specifies register or register pair that contains data.
- Example

  MOV A B ; A $\leftarrow$ B.

  ADD B ; A=A+B.

4) Indirect Addressing Mode

- In this mode 16 bit memory address is indirectly provided with the instruction using a register pair.
- Example

  LDAX D ; A $\leftarrow$ M[DE].

  STAX D ; M[DE] $\leftarrow$ A.

5) Implicit Addressing Mode

- This mode doesn't require any operand, data is specified by the Opcode itself.
- Example

  CMA

1. **Write an ALP to load register B with data 14H, register C with FFH, register D with 29H and register E with 67H.**

```
MVI B, 14H

MVI C, FFH

MVI D, 29H

MVI E, 67H

HLT
```

2. **Write an ALP to transfer data from register B to C.**

```
MVI B, 55H

MOV C, B

HLT
```

3. **Write an ALP to store data of register B into memory location 2050H.**

```
MVI B, 67H

MOV A, B

STA 2050H ; Store data of  Accumulator at memory location 2050H

HLT
```

4. **write an ALP which directly store data 56H into memory location 2050H.**

```
LXI H, 2050H

MVI M, 56H

HLT
```

5. **Write an 8085 assembly language program for exchanging two 8-bit numbers stored in memory locations 2050h and 2051h.**

```
LDA 2050H

MOV B, A

LDA 2051H

STA 2050H

MOV A, B

STA 2051H

HLT
```

6. **Write an ALP to interchange 16-bit data stored in register BC and DE.**

WITHOUT XCHG INSTRUCTION

```
MOV H, B
MOV L, C

MOV B, D

MOV C, E

MOV D, H

MOV E, L

HLT
```

WITH XCHG INSTRUCTION

```
MOV H, B

MOV L, C

XCHG        ; The contents of register H are exchanged with the contents of register D, and the

            ; contents of register L are exchanged with the contents of register E.

MOV B, H

MOV C, L

HLT
```

7. **Write the set of 8085 assembly language instructions to store the contents of B and C registers on the stack.**

```
MVI B, 50H

MVI C, 60H

PUSH B

PUSH C

HLT
```

8. **Write an ALP to delete (Make 00H) the data byte stored at memory location from address stores in register DE.**

```
MVI A, 00H

STAX D

HLT
```

9. **Write an 8085 assembly language program to add two 8-bit numbers stored in memory locations 2050h and 2051h. Store result in location 2052h.**

```
LXI H 2050H

MOV A M

INX H

ADD M
```

```
INX H

MOV M A

HLT
```

10. Subtract 8 bit data stored at memory location 2050H from data stored at memory location 2051H and store result at 2052H.

```
LXI H 2050H

MOV A M

INX H

SUB M ;  A = A - M

INX H

MOV M A

HLT
```

11. Write an 8085 assembly language program to add two 16-bit numbers stored in memory.

```
LHLD 2050H

XCHG        ; The contents of register H are exchanged with the contents of register D, and the

            ; contents of register L are exchanged with the contents of register E.

LHLD 2052H

MOV A E

ADD L

MOV L A

MOV A D

ADC H

MOV H A

SHLD 2054H ; Store Value of L Register at 2054 and value of H register at 2055.

HLT
```

12. Write an 8085 assembly language program to find the number of 1's binary representation of given 8-bit number.

```
MVI B 00H

MVI C 08H

MOV A D
```

```
BACK: RAR ; Rotate Accumulator Right through carry flag.

JNC SKIP

INR B

SKIP: DCR C ; Increment of B will skip.

JNZ BACK

HLT
```

13. Implement the Boolean equation D= (B+C) · E, where B, C, D and E represents data in various registers of 8085.

```
MOV A B

ORA C

ANA E

MOV D A

HLT
```

14. Write an 8085 assembly language program to add two decimal numbers using DAA instruction.

```
LXI H 2050H

MOV A M

INX H

MOV B M

MVI C 00H

ADD B

DAA ; Decimal adjustment of accumulator.

JNC SKIP

INR C

SKIP: INX H ; Increment of C will skip.

MOV M A

INX H

MOV M C

HLT
```

15. Write an 8085 assembly language program to find the minimum from two 8-bit numbers.

```
LDA 2050H

MOV B A

LDA 2051H

CMP B

JNC SMALL

STA 2052H

HLT

SMALL: MOV A B

STA 2052H

HLT
```

16. Write an 8085 program to copy block of five numbers starting from location 2001h to locations starting from 3001h.

```
LXI D 3100H

MVI C 05H

LXI H 2100

LOOP: MOV A M

STAX D

INX D

INX H

DCR C

JNZ LOOP

HLT
```

17. An array of ten data bytes is stored on memory locations 2100H onwards. Write an 8085 assembly language program to find the largest number and store it on memory location 2200H.

```
LXI H 2100H

MVI C 0AH

MOV A M

DCR C

LOOP: INX H
```

```
CMP M       ; Compare Data of accumulator with the data of memory location specified by HL pair and
            ; set flags accordingly.
JNC AHED
MOV A M
AHED: DCR C
JNZ LOOP
STA 2200H
HLT
```

18. Write an 8085 assembly language program to add block of 8-bit numbers.

```
LXI H 2000H
LXI B 3000H
LXI D 4000H
BACK: LDAX B
ADD M
STAX D
INX H
INX B
INX D
MOV A L
CPI 0A
JNZ BACK
HLT
```

19. Write an 8085 assembly language program to count the length of string ended with 0dh starting from location 2050h (Store length in register B).

```
LXI H 2050H
MVI B 00H
BACK: MOV A M
INR B
INX H
CPI 0DH
```

```
JNZ BACK

DCR B

HLT
```

20. An array of ten numbers is stored from memory location 2000H onwards. Write an 8085 assembly language program to separate out and store the EVEN and ODD numbers on new arrays from 2100H and 2200H, respectively.

```
LXI H 2000H

LXI D 2100H

LXI B 2200H

MVI A 0AH

COUNTER: STA 3000H

MOV A M

ANI 01H

JNZ CARRY

MOV A M

STAX B

INX B

JMP JUMP

CARRY: MOV A M ; This block will store Odd numbers.

STAX D

INX D

JUMP: LDA 3000H

DCR A

INX H

JNZ COUNTER

HLT
```

21. An array of ten data bytes is stored on memory locations 2100H onwards. Write an 8085 assembly language program to find the bytes having complemented nibbles (e.g. 2DH, 3CH, 78H etc.) and store them on a new array starting from memory locations 2200H onwards.

```
LXI H 2100H

LXI D 2200H
```

```
MVI C 0AH

LOOP: MOV A M

ANI 0FH

MOV B A

MOV A M

ANI F0H

RRC

RRC

RRC

RRC

CPM B

JNZ NEXT

MOV A M

STAX D

INX D

NEXT: INX H

DCR C

JNZ LOOP

HLT
```

22. Write an 8085 assembly language program to count the positive numbers, negative numbers, zeros, and to find the maximum number from an array of twenty bytes stored on memory locations 2000H onwards. Store these three counts and the maximum number on memory locations 3001H to 3004H, respectively.

```
LXI H 2000

MVI C 14

MVI D 00

MVI B 00

MVI E 00

LOOP: MOV A M

CMP B
```

```
JC NEG

JNZ POS

INX H

DCR C

JNZ LOOP

JMP STORE


NEG: INR D ; Count Negative number

INX H

DCR C

JNZ LOOP

JMP STORE


POS: INR E ; Count Positive number

INX H

DCR C

JNZ LOOP

JMP STORE


STORE: MOV A E

STA 3001


MOV A D

STA 3002


LXI H 2000

MVI C 14

MVI D 00

MVI B 00
MVI E 00

LOOP1: MOV A M ; Main Program for count Zero And Find Maximum.
CMP B
JZ ZERO
JNC MAX
INX H
DCR C
JNZ LOOP1
JMP STORE1

ZERO: INR D ; For count Zero
INX H
DCR C
JNZ LOOP1
JMP STORE1

MAX: CMP E ; Find Maximum.
JC SKIP
MOV E A
SKIP: INX H
DCR C
JNZ LOOP1
JMP STORE1

STORE1: MOV A D ; Store Number of zeros
STA 3003
MOV A E
STA 3004 ; Store maximum.
```

```
HLT
```

23. Write an 8085 assembly language program to separate out the numbers between 20_{10} and 40_{10} from an array of ten numbers stored on memory locations 2000H onwards. Store the separated numbers on a new array from 3000H onwards.

```
LXI H 2000

LXI D 3000

MVI C 0A

LOOP: MOV A M

CPI 14

JZ NEXT

JC NEXT

CPI 28

JNC NEXT

STAX D

INX D

NEXT: INX H ; Skip Storing of Number.

DCR C

JNZ LOOP

HLT
```

24. Write an 8085 assembly language program sort an array of twenty bytes stored on memory locations 2000H onwards in descending order.

```
MVI B 14

L2: LXI H 2000

MVI C 13

L1: MOV A M

INX H

CMP M

JC SWAP

bACK: DCR C

JNZ L1
```

```
DCR B

JNZ L2

HLT


SWAP: MOV D M; This block swap values.

MOV M A

DCX H

MOV M D

INX H

JMP BACK
```

25. An array of twenty data bytes is stored on memory locations 4100H onwards. Write an 8085 assembly language program to remove the duplicate entries from the array and store the compressed array on a new array starting from memory locations 4200H onwards.

```
MVI B 14H

MVI C 01H

LXI H 4101H

SHLD 3000H

LDA 4100H

STA 4200H

; This program fetch one by one value from original array and sore it on new array if it is not duplicate.

L1: LHLD 3000H

MOV A M

INX H

DCR B

JZ OVER

SHLD 3000H

LXI H 4200H

MOV D C

L2: CMP M

JZ L1
```

```
INX H

DCR D

JNZ L2

MOV M A

INR C

JMP L1

OVER: HLT
```

26. Write an ALP to Pack the two unpacked BCD numbers stored in memory locations 2200H and 2201H and store result in memory location 2300H. Assume the least significant digit is stored at 2200H.

```
LDA 2201

RLC ; Rotate accumulator left 4 times without carry.

RLC

RLC

RLC

ANI F0

MOV C A

LDA 2200

ADD C

STA 2300

HLT
```

27. Write a set of 8085 assembly language instructions to unpack the upper nibble of a BCD number.

```
MVI A 98

MOV B A

ANI F0

RRC ; Rotate accumulator left 4 times without carry.

RRC

RRC

RRC

STA 2000
```

```
HLT
```

28. Write Assembly language program to subtract 2 16-bit BCD numbers.

```
LXI H 3040

LXI D 1020

MOV A L

SUB E

DAA

STA 2000

MOV A H

SBB D

DAA

STA 2001

HLT
```

29. Write an 8085 assembly language program to continuously read an input port with address 50H. Also write an ISR to send the same data to output port with address A0H when 8085 receives an interrupt request on its RST 5.5 pin.

```
LOOP: IN 50

EI

CALL DELAY

JMP LOOP

HLT

DELAY: NOP

NOP

NOP

NOP

RET


; This code must be write at memory location 002C onwards.

OUT A0

JMP LOOP
```

30. Write an ALP to generate a square wave of 2.5 kHz frequency. Use D_0 bit of output port ACH to output the square wave.

```
MVI A 01H

REPEAT: OUT AC

MVI C Count

AGAIN: DCR C

JNZ AGAIN

CMA

JMP REPEAT
```

Calculation:

$$Time\ period\ of\ square\ wave = \frac{1}{2.5 * 10^3} = 0.4 * 10^{-3} s.$$

$$Time\ period\ of\ upper\ half\ and\ lower\ half\ of\ square\ wave = \frac{0.4 * 10^{-3} s}{2}. = 0.2 * 10^{-3} s.$$

$$let\ processor\ time\ period = 0.3 * 10^{-6} s.$$

$$Delay\ required\ beween\ transition\ of\ square\ wave = \frac{0.2 *\ 10^{-3}}{0.3 * 10^{-6}} \approx 666 Tstates$$

Now

$$666 = 7 + (14 * Count) - 3 + 4$$

$$658 = 14 * Count$$

$$Count = 47$$

$$Count = 2FH$$

Final Program:

```
MVI A 01H

REPEAT: OUT AC

MVI C 2F

AGAIN: DCR C

JNZ AGAIN

CMA

JMP REPEAT
```

1. Stack

- Stack is a group of memory location in the R/W memory that is used for temporary storage of binary information during execution of a program.
- The starting memory location of the stack is defined in program and space is reserved usually at the high end of memory map.
- The beginning of the stack is defined in the program by using instruction **LXI SP, 16-bit memory address**. Which loads a 16-bit memory address in stack pointer register of microprocessor.
- Once stack location is defined storing of data bytes begins at the memory address that is one less then address in stack pointer register. LXI SP, 2099h the storing of data bytes begins at 2098H and continues in reversed numerical order.

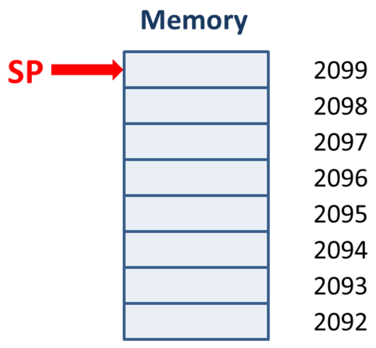
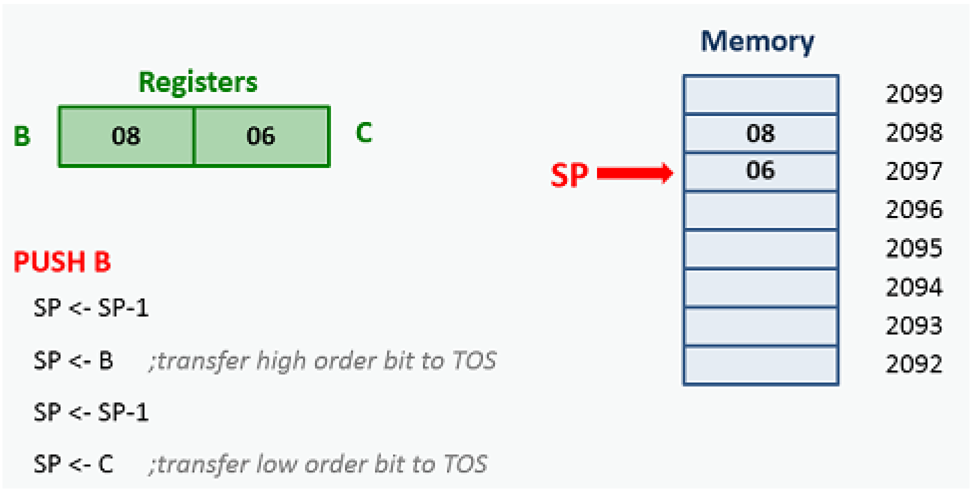


Fig. Stack

- Data bytes in register pair of microprocessor can be stored on the stack in reverse order by using the PUSH instruction.
- PUSH B instruction sore data of register pair BC on sack.



```
PUSH B
  SP <- SP-1
  SP <- B     ;transfer high order bit to TOS
  SP <- SP-1
  SP <- C     ;transfer low order bit to TOS
```

Fig. PUSH operation on stack

- Data bytes can be transferred from the stack to respective registers by using instruction POP.

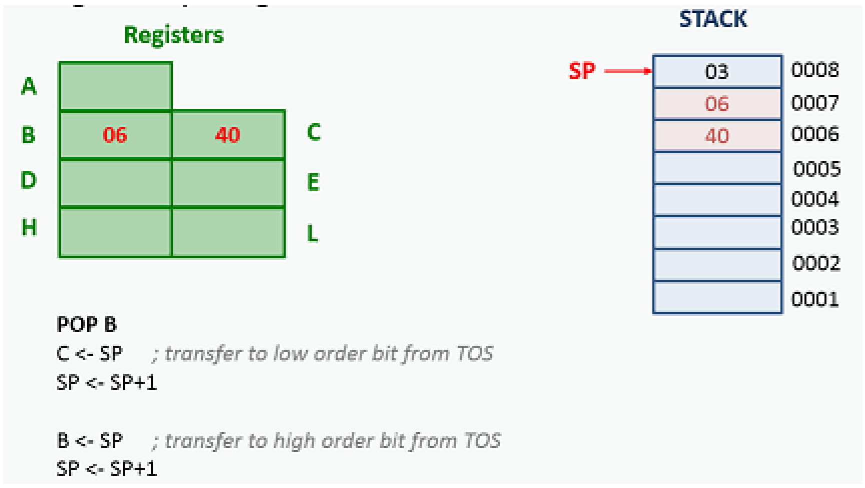


Fig. POP operation on stack

Instruction necessary for stack in 8085

LXI SP, 2095	Load the stack pointer register with a 16-bit address.
PUSH B/D/H	It copies contents of B-C/D-E/H-L register pair on the stack.
PUSH PSW	Operand PSW represents Program status word meaning contents of accumulator and flags.
POP B/D/H	It copies content of top two memory locations of the stack in to specified register pair.
POP PSW	It copies content of top two memory locations of the stack in to B-C accumulator and flags respectively.

2. Subroutine

- A subroutine is a group of instruction that performs a subtask of repeated occurrence.
- A subroutine can be used repeatedly in different locations of the program.

Advantage of using Subroutine

- Rather than repeat the same instructions several times, they can be grouped into a subroutine that is called from the different locations.

Where to write Subroutine?

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

Instructions for dealing with subroutines in 8085.

- The **CALL** instruction is used to redirect program execution to the subroutine.
  - When CALL instruction is fetched, the Microprocessor knows that the next two **new** Memory location contains 16bit subroutine address.
  - Microprocessor Reads the subroutine address from the next two memory location and stores the higher order 8bit of the address in the **W** register and stores the lower order 8bit of the address in the **Z** register.
  - Push the **Older** address of the instruction immediately following the CALL onto the stack [Return address]
  - Loads the program counter (**PC**) with the **new** 16-bit address supplied with the CALL instruction from **WZ** register.
- The **RET** instruction is used to return.

- Number of PUSH and POP instruction used in the subroutine must be same, otherwise, RET instruction will pick wrong value of the return address from the stack and program will fail.

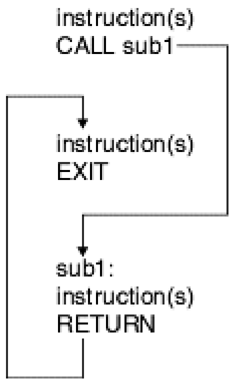


Fig. Subroutine

- Example: write ALP to add two numbers using call and subroutine.

```
LXI H 2000 ; Load memory address of operand
MOV B M ; Store first operand in register B
INX H ;Increment H-L pair
MOV A M ; Store second operand in register A
CALL ADDITION ; Call subroutine ADDITION
STA 3000 ; Store answer
HLT

ADDITION: ADD B ; Add A and B
RET ; Return
```

Conditional call and return instruction available in 8085

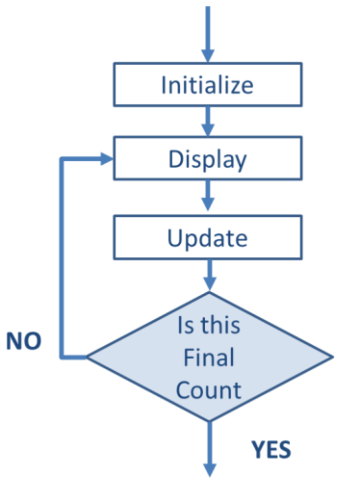
CC 16-bit address	Call on Carry, Flag Status: CY=1
CNC 16-bit address	Call on no Carry, Flag Status: CY=0
CP 16-bit address	Call on positive, Flag Status: S=0
CM 16-bit address	Call on minus, Flag Status: S=1
CZ 16-bit address	Call on zero, Flag Status: Z=1
CNZ 16-bit address	Call on no zero, Flag Status: Z=0
CPE 16-bit address	Call on parity even, Flag Status: P=1
CPO 16-bit address	Call on parity odd, Flag Status: P=0
RC	Return on Carry, Flag Status: CY=1
RNC	Return on no Carry, Flag Status: CY=0
RP	Return on positive, Flag Status: S=0
RM	Return on minus, Flag Status: S=1
RZ	Return on zero, Flag Status: Z=1
RNZ	Return on no zero, Flag Status: Z=0
RPE	Return on parity even, Flag Status: P=1
RPO	Return on parity odd, Flag Status: P=0

3. Applications of Counters and Time Delays

1. Traffic Signal
2. Digital Clocks
3. Process Control
4. Serial data transfer

4. Counters

- A counter is designed simply by loading appropriate number into one of the registers and using INR or DNR instructions.
- Loop is established to update the count.
- Each count is checked to determine whether it has reached final number; if not, the loop is repeated.



```
1. MVI C,05
2. LOOP: MOV A,C
3.       OUT 01
4.       DCR C
5.     JNZ LOOP
6. HLT
```

Fig. Counter

5. Time Delay

- Each instruction passes through different combinations of Fetch, Memory Read, and Memory Write cycles.
- Knowing the combinations of cycles, one can calculate how long such an instruction would require to complete.
- It is counted in terms of number of T–states required.
- Calculating this time we generate require software delay.

Time Delay Using Single Register

Label	Opcode	Operand	Comment	T-states
	MVI	C,05h	; Load Counter	**7**
LOOP:	DCR	C	; Decrement Counter	**4**
	JNZ	LOOP	; Jump back to Decr. C	**10/7**

MVI C 05 Mchine Cycle: F + R = 2 T-States: 4T + 3T = 7T	DCR C Mchine Cycle: F = 1 T-States: 4T = 4T	JNZ LOOP (true) Mchine Cycle: F + R + R = 3 T-States: 4T + 3T + 3T = 10T	JNZ LOOP (false) Mchine Cycle: F + R = 3 T-States: 4T + 3T = 7T

- Instruction MVI C, 05h requires 7 T-States to execute. Assuming, 8085 Microprocessor with 2MHz clock frequency. How much time it will take to execute above instruction?
  Clock frequency of the system (f) = 2 MHz
  Clock period (T) = 1/f = ½ * 10-6 = 0.5 μs
  Time to execute MVI = 7 T-states * 0.5 μs
  = 3.5 μs
- Now to calculate time delay in loop, we must account for the T-states required for each instruction, and for the number of times instructions are executed in the loop. There for the next two instructions:
  DCR: 4 T-States
  JNZ: 10 T-States
  14 T-States
- Here, the loop is repeated for 5 times.
- Time delay in loop T_L with 2MHz clock frequency is calculated as:
  T_L= T * Loop T-sates * N_{10} -----------------(1)
  T_L : Time Delay in Loop
  T : Clock Frequency
  N_{10} : Equivalent decimal number of hexadecimal count loaded in the delay register.
- Substituting value in equation (1)
  T_L= $(0.5 * 10^{-6} * 14 * 5)$
  = 35 μs
- If we want to calculate delay more accurately, we need to accurately calculate execution of JNZ instruction i.e
  If **JNZ = true**, then **T-States = 10**
  Else if **JNZ =false**, then **T-States = 7**
- Delay generated by last clock cycle:
  = 3T * Clock Period
  = 3T * $(1/2 * 10^{-6})$
  = 1.5 μs
- Now, the accurate loop delay is:
  T_{LA}=T_L - Delay generated by last clock cycle
  T_{LA}= 35 μs - 1.5 μs
  T_{LA}= 33.5 μs
- Now, to calculate total time delay
  Total Delay = Time taken to execute instruction outside loop + Time taken to execute loop instructions
  $T_D = T_O + T_{LA}$
  = (7 * 0.5 μs) + 33.5 μs
  = 3.5 μs + 33.5 μs
  = 37 μs
- In most of the case we are given time delay and need to find value of the counter register which decide number of times loop execute.
- For example: write ALP to generate 37 μs delay given that clock frequency if 2 MHz.
- Single register loop can generate small delay only for large delay we use other technique.

Time Delay Using a Register Pair

- Time delay can be considerably increased by setting a loop and using a register pair with a 16-bit number (FFFF h).
- A 16-bit is decremented by using DCX instruction.
- Problem with DCX instruction is DCX instruction doesn't set **Zero** flag.
- Without test flag, Jump instruction can't check desired conditions.
- Additional technique must be used to set Zero flag.

Label	Opcode	Operand	Comment	T-states
	LXI	B,2384 h	; Load BC with 16-bit counter	**10**
LOOP:	DCX	B	; Decrement BC by 1	**6**
	MOV	A, C	; Place contents of C in A	**4**
	ORA	B	; OR B with C to set Zero flag	**4**
	JNZ	LOOP	; if result not equal to 0, 10/7 jump back to loop	**10/7**

- Here the loop includes four instruction:
  Total T-States = 6T + 4T + 4T + 10T
  = 24 T-states
- The loop is repeated for 2384 h times.
- Converting $(2384)_{16}$ into decimal.
  2384 h = $(2 * 16^3) + (3 * 16^2) + (8 * 16^1) + (4 * 16^0)$
  = 8192 + 768 + 128 + 4 = **9092**
- Clock frequency of the system (f)= 2 MHz
- Clock period (T) = 1/f = ½ * 10^{-6} = 0.5 μs
- Now, to find delay in the loop
  T_L= T * Loop T-sates * N_{10}
  = 0.5 * 24 * 9092
  = 109104 μs = 109 ms (without adjusting last cycle)

Time Delay Using a LOOP within a LOOP

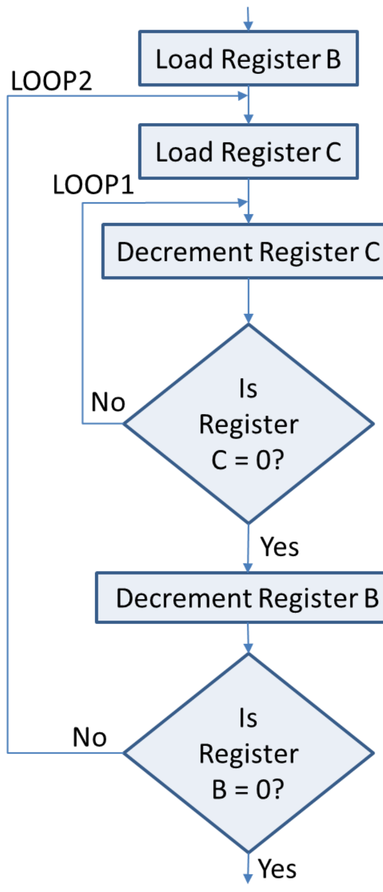


Fig. Time Delay Using a LOOP within a LOOP

Label	Opcode	Operand	T-states
	MVI	B,38h	7T
LOOP2:	MVI	C,FFh	7T
LOOP1:	DCR	C	4T
	JNZ	LOOP1	10/7 T
	DCR	B	4T
	JNZ	LOOP2	10/7 T

- Calculating delay of inner LOOP1: T_{L1}

  T_L= T * Loop T-states * N_{10}

  = 0.5 * 14* 255

  = 1785 µs = 1.8 ms

  T_{L1}= TL – (3T states* clock period)

  = 1785 – (3 * ½ * 10^{-6})

  = 1785-1.5=**1783.5 µs**
- Now, Calculating delay of outer LOOP2: T_{L2}
- Counter B : $(38)_{16}$ = **$(56)_{10}$** So loop2 is executed for 56 times.

  T-States = 7 + 4 + 10 = 21 T-States

  T_{L2} = 56 (T_{L1} + 21 T-States * 0.5)

  = 56(1783.5 µs + 10.5)

= 100464 µs

T_{L2} = 100.46 ms

Disadvantage of using software delay

- Accuracy of time delay depends on the accuracy of system clock.
- The Microprocessor is occupied simply in a waiting loop; otherwise it could be employed to perform other functions.
- The task of calculating accurate time delays is tedious.
- In real time applications timers (integrated timer circuit) are commonly used.
- Intel 8254 is a programmable timer chip that can be interfaced with microprocessor to provide timing accuracy.
- The disadvantage of using hardware chip include the additional expense and the need for extra chip in the system.

6. Counter design with time delay

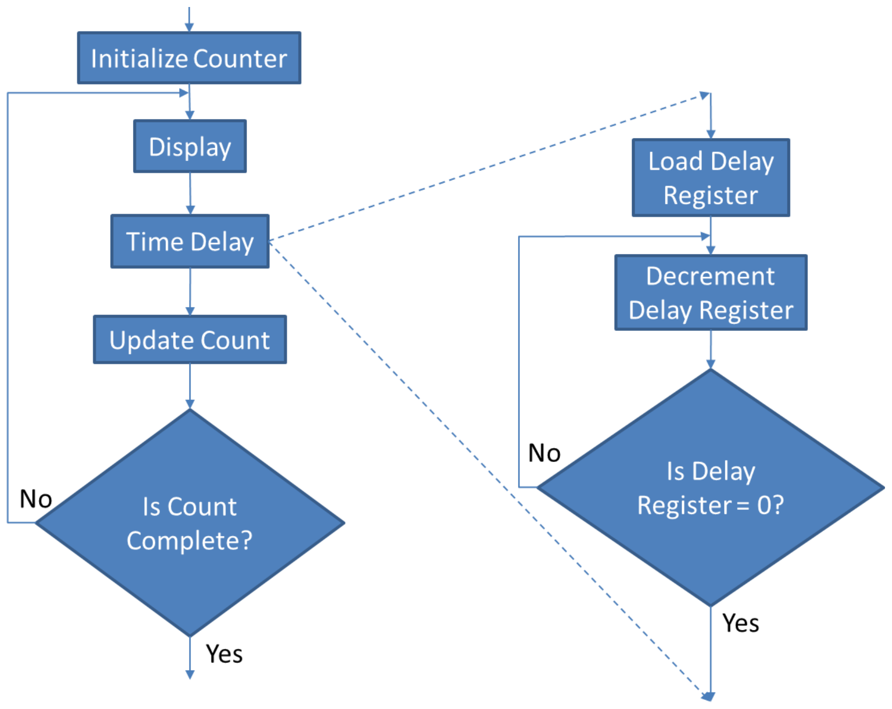


Fig. 6. Counter design with time delay

- It is combination of counter and time delay.
- I consist delay loop within counter program.

7. Hexadecimal counter program

- Write a program to count continuously in hexadecimal from **FFh** to **00h** with **0.5 µs** clock period. Use register C to set up 1 ms delay between each count and display the number at one of the output port.
- **Given:**
- Counter= FF h
- Clock Period T=0.5 µs

- Total Delay = 1ms
- **Output:**
- To find value of delay counter
- **Program**

```
MVI B,FF
LOOP:MOV A,B
    OUT 01
    MVI C, COUNT; need to calculate delay count
    DELAY: DCR C
            JNZ DELAY
    DCR B
    JNZ LOOP
HLT
```

- Calculate Delay for Internal Loop

  TI = T-States * Clock Period * COUNT

  = 14 * 0.5 * 10-6 * COUNT

  TI = (7.0 * 10-6)* COUNT
- Calculate Delay for Outer Loop:

  TO = T-States * Clock Period

  = 35 * 0.5 * 10-6

  TO = 17.5 s
- Calculate Total Time Delay:

  TD = TO + TL

  1 ms = 17.5 * 10-6 + (7.0 * 10-6)* COUNT

  1 * 10-3 = 17.5 * 10-6 + (7.0 * 10-6)* COUNT

  COUNT="1 ∗ 10−3 − 17.5 ∗ 10−6" /"7.0 ∗ 10−6"

  COUNT= $(140)_{10}$ = $(8C)_{16}$

8. 0-9 up/down counter program

- Write an 8085 assembly language program to generate a decimal counter (which counts 0 to 9 continuously) with a one second delay in between. The counter should reset itself to zero and repeat continuously. Assume a crystal frequency of 1MHz.
- **Program**

```
    START: MVI B,00H
    DISPLAY: OUT 01
            LXI H, COUNT
    LOOP: DCX H
            MOV A, L
            ORA H
            JNZ LOOP
            INR B
            MOV A,B
            CPI 0A
            JNZ DISPLAY
```

```
JZ START
```

9. Code Conversion

Two Digit BCD Number to Binary Number

1. Initialize memory pointer to given address (2000).
2. Get the Most Significant Digit (MSD).
3. Multiply the MSD by ten using repeated addition.
4. Add the Least Significant Digit (LSD) to the result obtained in previous step.
5. Store the HEX data in Memory.

- **Program**

```
LXI H 2000
MOV C M
MOV A C
ANI 0F ; AND operation with 0F (00001111)
MOV E A
MOV A C
ANI F0 ; AND operation with F0 (11110000)
JZ SB1 ; If zero skip further process and directly add LSD
RRC ; Rotate 4 times right
RRC
RRC
RRC
MOV D A
MVI A 00
L1: ADI 0A ; Loop L1 multiply MSD with 10
DCR D
JNZ L1
SB1: ADD E
STA 3000 ; Store result
HLT
```

8-bit Binary Number to Decimal Number

1. Load the binary data in accumulator
2. Compare ‘A’ with 64 (Dicimal 100) if cy = 01, go step 5 otherwise next step
3. Subtract 64H from ‘A’ register
4. Increment counter 1 register
5. Go to step 2
6. Compare the register ‘A’ with ‘0A’ (Dicimal 10), if cy=1, go to step 10, otherwise next step
7. Subtract 0AH from ‘A’ register
8. Increment Counter 2 register
9. Go to step 6
10. Combine the units and tens to from 8 bit result
11. Save the units, tens and hundred’s in memory
12. Stop the program execution

- **Program**

```
MVI B 00
```

```
LDA 2000
LOOP1: CPI 64 ; Compare with 64H
JC NEXT1 : If A is less than 64H then jump on NEXT1
SUI 64 ; subtract 64H
INR B
JMP LOOP1
NEXT1:  LXI H 2001
MOV M B ; Store MSD into memory
MVI B 00
LOOP2: CPI 0A ; Compare with 0AH
JC NEXT2 ; If A is less than 0AH then jump on NEXT2
SUI 0A ; subtract 0AH
INR B
JMP LOOP2
NEXT2:  MOV D A
MOV A B
RLC
RLC
RLC
RLC
ADD D
STA 2002 ; Store packed number formed with two leas significant digit
HLT
```

Binary Number to ASCII Number

- Load the given data in A - register and move to B - register
- Mask the upper nibble of the Binary decimal number in A - register
- Call subroutine to get ASCII of lower nibble
- Store it in memory
- Move B - register to A - register and mask the lower nibble
- Rotate the upper nibble to lower nibble position
- Call subroutine to get ASCII of upper nibble
- Store it in memory
- Terminate the program.

```
LDA 5000 Get Binary Data
    MOV B, A
    ANI 0F        ; Mask Upper Nibble
    CALL SUB1       ; Get ASCII code for upper nibble
    STA 5001
    MOV A, B
    ANI F0        ; Mask Lower Nibble
    RLC
    RLC
    RLC
    RLC
    CALL SUB1       ; Get ASCII code for lower nibble
    STA 5002
```

```
        HLT           ; Halt the program.

SUB1:   CPI 0A
        JC SKIP
        ADI 07
SKIP:   ADI 30
        RET           ; Return Subroutine
```

ASCII Character to Hexadecimal Number

1. Load the given data in A - register
2. Subtract 30H from A - register
3. Compare the content of A - register with 0AH
4. If A < 0AH, jump to step6. Else proceed to next step
5. Subtract 07H from A - register
6. Store the result
7. Terminate the program

- **Program**

```
LDA 2000
CALL ASCTOHEX
STA 2001
HLT

ASCTOHEX: SUI 30 ; This block Convert ASCII to Hexadecimal.
CPI 0A
RC
SUI 07
RET
```

10. BCD Arithmetic

Add 2 8-bit BCD Numbers

1. Load firs number into accumulator.
2. Add second number.
3. Apply decimal adjustment to accumulator.
4. Store result.

- **Program**

```
LXI H, 2000H
MOV A, M
INX H
ADD M
DAA
INX H
MOV M, A
HLT
```

Subtract the BCD number stored in E register from the number stored in the D register

1. Find 99’s complement of data of register E
2. Add 1 to find 100’s complement of data of register E
3. Add Data of Register D
4. Apply decimal adjustment

- **Program**

```
MVI A, 99H
SUB E          : Find the 99's complement of subtrahend
INR A          : Find 100's complement of subtrahend
ADD D          : Add minuend to 100's complement of subtrahend
DAA            : Adjust for BCD
HLT            : Terminate program execution
```

11. 16-Bit Data operations

Add Two 16 Bit Numbers

1. Initialize register C for using it as a counter for storing carry value.
2. Load data into HL register pair from one memory address (9000H).
3. Exchange contents of register pair HL with DE.
4. Load second data into HL register pair (from 9002H).
5. Add register pair DE with HL and store the result in HL.
6. If carry is present, go to 7 else go to 8.
7. Increment register C by 1.
8. Store value present in register pair HL to 9004H.
9. Move content of register C to accumulator A.
10. Store value present in accumulator (carry) into memory (9006H).
11. Terminate the program.

- **Program**

```
MVI C, 00H
LHLD 9000H
XCHG ; Exchange contents of register pair HL with DE
LHLD 9002H
DAD D ; Add register pair DE with HL and store the result in HL
JNC AHEAD ; If carry is present, go to AHEAD
INR C
AHEAD: SHLD 9004H ; Store value present in register pair HL to 9004H
MOV A, C
STA 9006H ; Store value present in accumulator (carry) into memory (9006H)
HLT
```

Subtract Two 16 Bit Numbers

1. Load first data from Memory (9000H) directly into register pair HL.
2. Exchange contents of register pair DE and HL.
3. Load second data from memory location (9002H) directly into register pair HL.

4. Move contents of register E into accumulator A.
5. Subtract content of register L from A.
6. Move contents of accumulator A into register L.
7. Move contents of register D into accumulator A.
8. Subtract with borrow contents of register H from accumulator A.
9. Move contents of accumulator A into register H.
10. Store data contained in HL register pair into memory (9004H).
11. Terminate the program.

- **Program**

```
LHLD 9000H ; Load first data from Memory (9000H) directly into register pair HL
XCHG ; Exchange contents of register pair DE and HL.
LHLD 9002H ; Load second data from memory location (9002H) directly into register pair HL
MOV A, E
SUB L
MOV L, A
MOV A, D
SBB H ; Subtract with borrow contents of register H from accumulator A
MOV H, A
SHLD 9004H ; Store data contained in HL register pair into memory (9004H)
HLT
```

1. Draw and explain the block diagram of the programmable peripheral interface (8255A).

Ans.

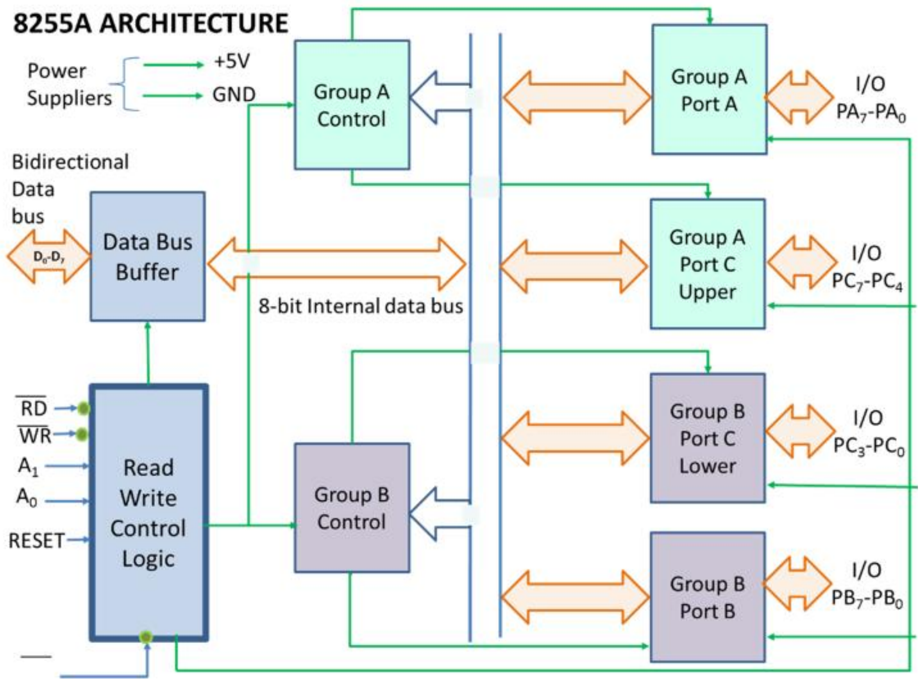


Figure: 8255A Architecture

Read Write Control Logic

RD (READ)	This is an active low signal that enables Read operation. When signal is low MPU reads data from selected I/O port of 8255A
WR (WRITE)	This is an active low signal that enables Write operation. When signal is low MPU writes data into selected I/O port or control register
RESET	This is an active high signal, used to reset the device. That means clear control registers
CS	This is Active Low signal. When it is low, then data is transfer from 8085 CS signal is the master Chip Select. A_0 and A_1 specify one of the I/O ports or control register

CS	A1	A0	Selected
0	0	0	PORT A
0	0	1	PORT B
0	1	0	PORT C
0	1	1	Control Register
1	X	X	8255A is not selected

Data Bus Buffer

- This three-state bi-directional 8-bit buffer is used to interface the 8255 to the system data bus.
- Data is transmitted or received by the buffer upon execution of input or output instructions by the CPU.
- Control words and status information are also transferred through the data bus buffer.

Group A and Group B Controls

- The functional configuration of each port is programmed by the systems software. In essence, the CPU "outputs" a control word to the 8255.
- The control word contains information such as "mode", "bit set", "bit reset", etc., that initializes the functional configuration of the 8255.
- Each of the Control blocks (Group A and Group B) accepts "commands" from the Read/Write Control logic, receives "control words" from the internal data bus and issues the proper commands to its associated ports.

Ports A, B, and C

- The 8255 contains three 8-bit ports (A, B, and C).
- All can be configured to a wide variety of functional characteristics by the system software but each has its own special features or "personality" to further enhance the power and flexibility of the 8255.
- Port A One 8-bit data output latch/buffer and one 8-bit data input latch.
- Both "pull-up" and "pull-down" bus-hold devices are present on Port A.
- Port B One 8-bit data input/output latch/buffer and one 8-bit data input buffer.
- Port C One 8-bit data output latch/buffer and one 8-bit data input buffer (no latch for input). This port can be divided into two 4-bit ports under the mode control.
- Each 4-bit port contains a 4-bit latch and it can be used for the control signal output and status signal inputs in conjunction with ports A and B.

2. Explain 8255A I/O Operating Modes

Ans. 8255A has three different I/O operating modes:

1. Mode 0
2. Mode 1
3. Mode 2

Mode 0

- Simple I/O for port A,B and C
- In this mode, Port A and B is used as two 8-bit ports and Port C as two 4-bit ports.
- Each port can be programmed in either input mode or output mode where outputs are latched and inputs are not latched.
- Ports do not have handshake or interrupt capability.

Mode 1: Input or Output with Handshake

- Handshake signal are exchanged between MPU and peripheral prior to data transfer.
- In this mode, Port A and B is used as 8-bit I/O ports.
- Mode 1 is a handshake Mode whereby ports A and/or B use bits from port C as handshake signals.
- In the handshake mode, two types of I/O data transfer can be implemented: status check and interrupt.
- Port A uses upper 3 signals of Port C: PC3, PC4, PC5
- Port B uses lower 3 signals of Port C : PC0, PC1, PC2
- PC6 and PC7 are general purpose I/O pins

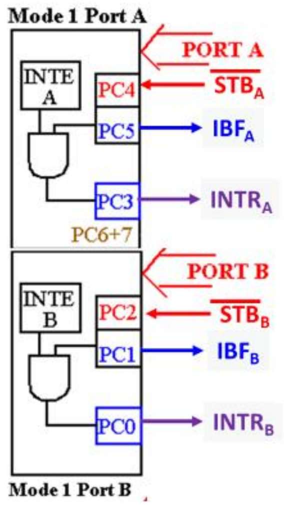


Figure: Mode1 Input Handshake

STB (Strobe Input):

- This active low signal is generated by a peripheral device to indicate that, it has transmitted a byte of data. The 8255A, in response to **STB**, generates **IBF** and **INTR**.

IBF (Input Buffer Full)

This signal is acknowledged by 8255A to indicate that the input latch has received the data byte. It will get reset when the MPU reads the data.

INTR(Interrupt Request)

This is an output signal that may be used to interrupt the MPU. This signal is generated if STB, IBF and INTE (internal flip-flop) are all at logic 1. It will get reset by the falling edge of RD

INTE(Interrupt Enable)

- This signal is an internal flip-flop, used to enable or disable the generation of INTR signal.
- The interrupt enable signal is neither an input nor an output; it is an internal bit programmed via the PC4 (port A) or PC2 (port B) bits.

Mode 2

- In this mode, Port A can be configured as the bidirectional port and Port B either in Mode 0 or Mode 1.
- Port A uses five signals from Port C as handshake signals for data transfer.
- The remaining three signals from Port C can be used either as simple I/O or as handshake for port B.

3. Explain BSR Mode of the programmable peripheral interface (8255A) with necessary diagrams.

Ans.

- These are two basic modes of operation of 8255.
  I/O mode and Bit Set-Reset mode (BSR).
- In I/O mode, the 8255 ports work as programmable I/O ports, while in BSR mode only port C (PC0-PC7) can be used to set or reset its individual port bits.
- Under the I/O mode of operation, further there are three modes of operation of 8255, so as to support different types of applications, mode 0, mode 1 and mode 2.

8255A: BSR(Bit Set/Reset) Mode

- In this mode any of the 8-bits of port C can be set or reset depending on D_0 of the control word.
- The bit to be set or reset is selected by bit select flags D_3, D_2 and D_1 of the CWR (Control Word Register).
- BSR Control Word affects one bit at a time
- It does not affect the I/O mode

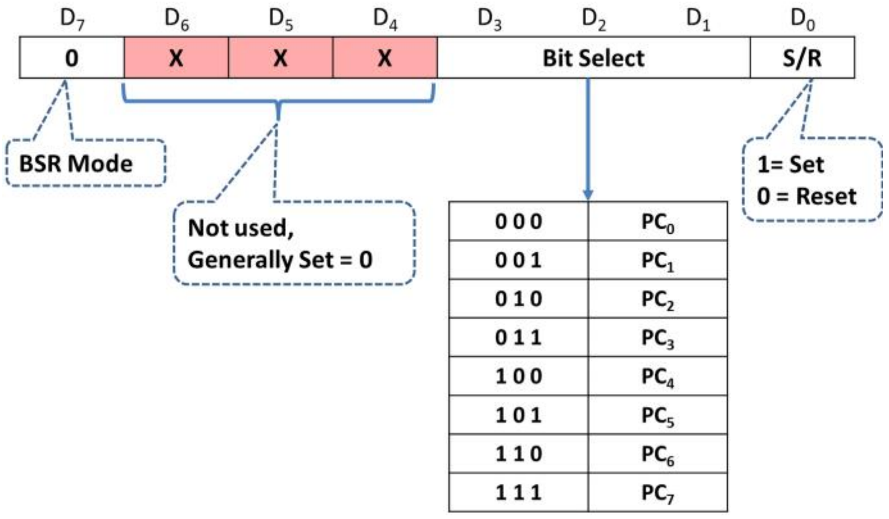


Figure: BSR Mode Control Word

4. Explain 8255A Control Word and Control Register with necessary diagram.

Ans. Control Register

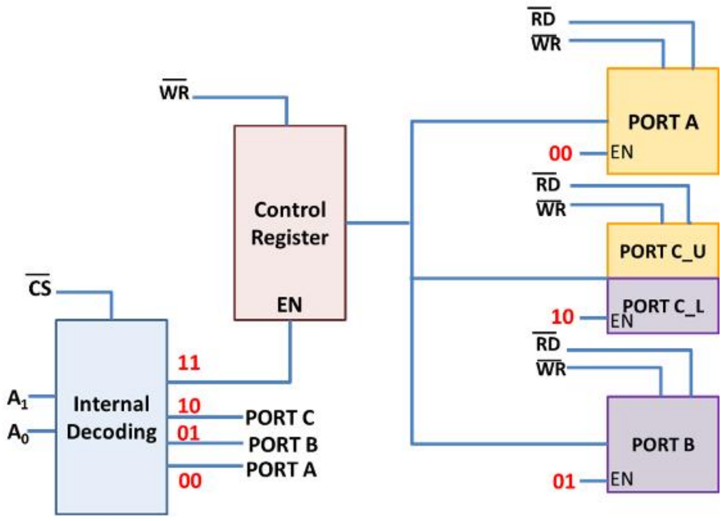


Figure: Control Register 8255A

Control Word: Content of Control register is known as Control Word.

- Control word specify an I/O function for each port this register can be.

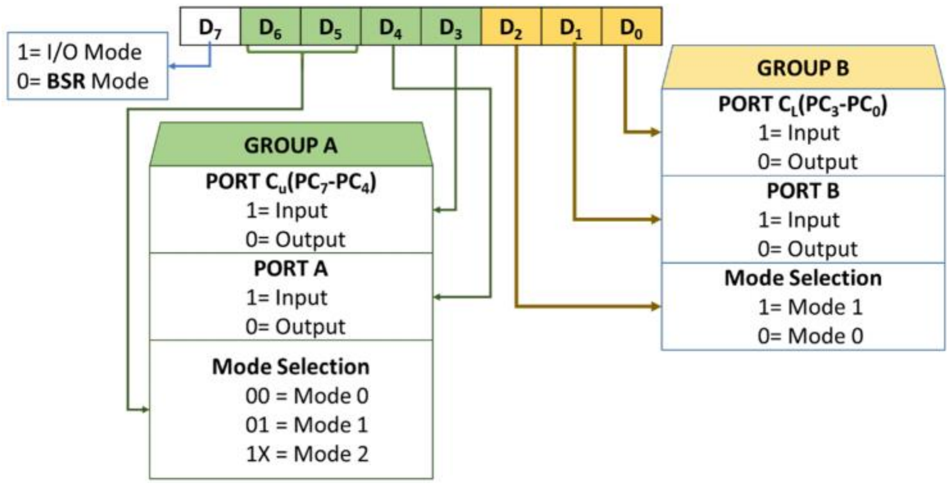


Figure:8255A Control Word

- Accessed to write a control word when A0 and A1 are at logic1, the register is not accessible for a read operation.
- Bit D7 of the control register either specifies the I/O function or the bit Set/Reset function, as classified in figure 1.
- If bit D7=0, bits D6-D0 determine I/O function in various mode, as shown in figure 4.
- If bit D7=0 port C operates in the bit Set/Reset (BSR) mode.
- The BSR control word does not affect the function of port A and B.

5. What is the need of the programmable interrupt controller (8259A)? Draw and explain the block diagram of 8259A.

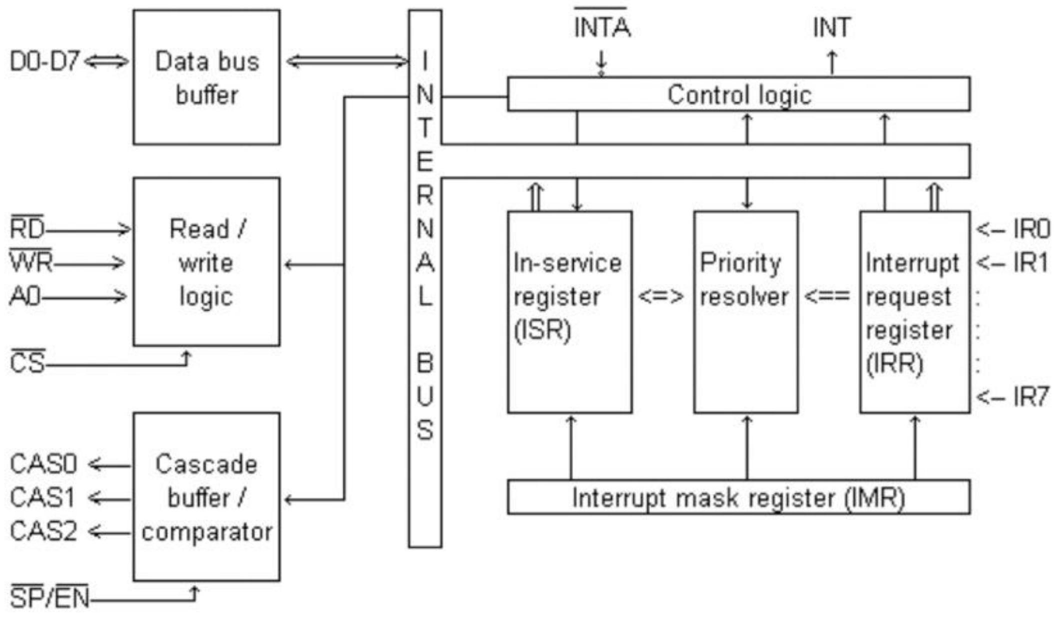
Ans.

- The Intel 8259 is a Programmable Interrupt Controller (PIC) designed for use with the 8085 and 8086 microprocessors.
- The 8259 can be used for applications that use more than five numbers of interrupts from multiple sources.

The main features of 8259 are listed below

- Manage eight levels of interrupts.
- Eight interrupts are spaced at the interval of four or eight locations.
- Resolve eight levels of priority in fully nested mode, automatic rotation mode or specific rotation mode.
- Mask each interrupt individually.
- Read the status of pending interrupt, in-service interrupt, and masked interrupt.
- Accept either the level triggered or edge triggered interrupt

8259 Internal Block Diagram



Read/Write Logic

- It is typical R/W logic.
- When address line A0 is at logic 0, the controller is selected to write a command word or read status.
- The Chip Select logic and A0 determine the port address of controller.

Control Logic

- It has two pins: INT as output and INTA as input.
- The INT is connected to INTR pin of MPU

Interrupt Registers and Priority Resolver

1. Interrupt Request Register (IRR)
2. Interrupt In-Service Register (ISR)
3. Priority Resolver
4. Interrupt Mask Register (IMR)

Interrupt Request Register (IRR) and Interrupt In-Service Register (ISR)

- Interrupt input lines are handled by two registers in cascade – IRR and ISR
- IRR is used to store all interrupt which are requesting service.
- ISR is used to store all interrupts which are being serviced.

Priority Resolver

- This logic block determines the priorities of the bit set in IRR.
- IR_0 is having highest priority, IR_7 is having lowest priority

Interrupt Mask Register

- It stores bits which mask the interrupt lines to be masked
- IMR operates on the IRR.
- Masking of high priority input will not affect the interrupt request lines.

Cascade Buffer / Comparator

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

6. State the difference between the vectored and non-vectored interrupts. Explain vectored interrupts of the 8085 microprocessor.

Ans. Difference between the vectored and non-vectored interrupts

VECTORED INTERRUPT

- In vectored interrupts, the processor automatically branches to the specific address in response to an interrupt.
- In vectored interrupts, the manufacturer fixes the address of the ISR to which the program control is to be transferred.
- The TRAP, RST 7.5, RST 6.5 and RST 5.5 are vectored interrupts.
- TRAP is the only non-maskable interrupt in the 8085.

NON-VECTORED INTERRUPT

- In non-vectored interrupts the interrupted device should give the address of the interrupt service routine (ISR).
- The INTR is a non-vectored interrupt.
- Hence when a device interrupts through INTR, it has to supply the address of ISR after receiving interrupt acknowledge signal.

Interrupt	Maskable	Vectored
INTR	Yes	No
RST 5.5	Yes	Yes
RST 6.5	Yes	Yes
RST 7.5	Yes	Yes
TRAP	No	Yes

Explain vectored interrupts of the 8085 microprocessor

The vector addresses of 8085 interrupts are given below:

Software Interrupt

RST 0	0000H
RST 1	0008H
RST 2	0010H
RST 3	0018H
RST 4	0020H
RST 5	0028H
RST 6	0030H
RST 7	0038H

Hardware Interrupt

RST 7.5	003CH
RST 6.5	0034H
RST 5.5	002CH
TRAP	0024H

Software Interrupt

- The software interrupts of 8085 are RST 0, RST 1, RST 2, RST 3, RST 4, RST 5, RST 6 and RST 7.
- The software interrupts cannot be masked and they cannot be disabled.

Hardware Interrupt

- The vectored hardware interrupts of 8085 are TRAP, RST 7.5, RST 6.5, RST 5.5.
- An external device, initiates the hardware interrupts of 8O85 by placing an appropriate signal at the interrupt pin of the processor.
- The processor keeps on checking the interrupt pins at the second T -state of last machine cycle of every instruction.
- If the processor finds a valid interrupt signal and if the interrupt is unmasked and enabled, then the processor accepts the interrupt.
- The acceptance of the interrupt is acknowledged by sending an INTA signal to the interrupted device.
- The processor saves the content of PC (program Counter) in stack and then loads the vector address of the interrupt in PC. (If the interrupt is non-vectored, then the interrupting device has to supply the address of ISR when it receives INTA signal).
- It starts executing ISR in this address.
- At the end of ISR, a return instruction, RET will be placed.
- When the processor executes the RET instruction, it POP the content of top of stack to PC.
- Thus the processor control returns to main program after servicing interrupt.

7. Explain Interfacing Seven-Segment LEDs as an Output

Ans.

- Interface the 8085 Microprocessor System with seven segment display through its programmable I/O port 8255.
- Seven segment displays is often used in the digital electronic equipment to display information regarding certain process.
- I/O devices (or peripherals) such as LEDs and keyboards are essential components of the microprocessor-based or microcontroller-based systems.
- Seven-segment LEDs Often used to display BCD numbers (1 through 9) and a few alphabets.
- A group of eight LEDs physically mounted in the shape of the number eight plus a decimal point.
- Each LED is called a segment and labeled as ‘a’ through ‘g’.

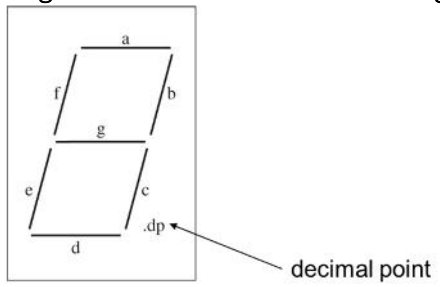
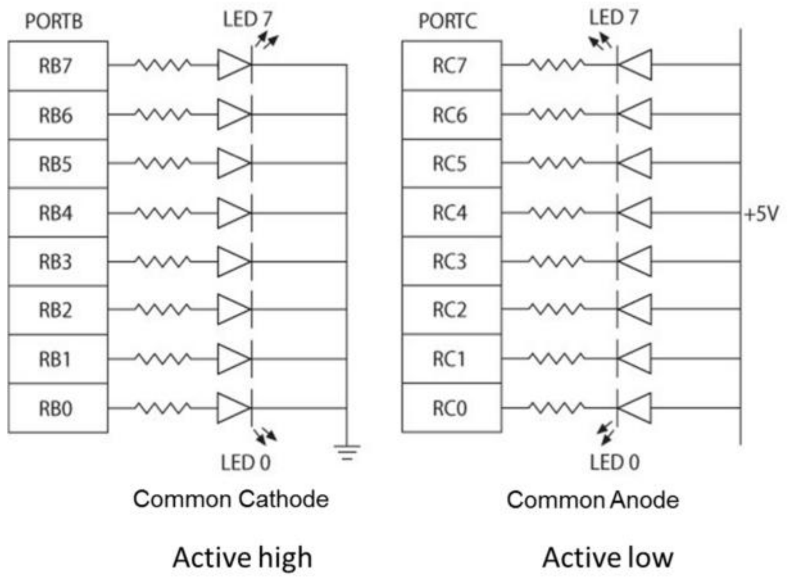


Figure: Seven Segment LED

- Commonly used output peripherals in embedded systems are LEDs, seven-segment LEDs, and LCDs; the simplest is LED

Two ways of connecting LEDs to I/O ports:

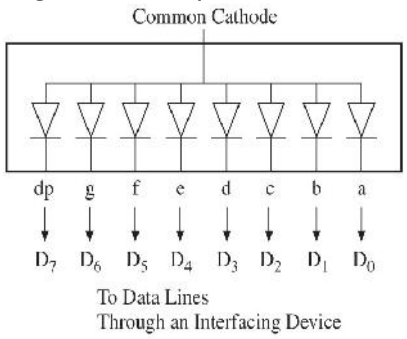
1. LED cathodes are grounded and logic 1 from the I/O port turns on the LEDs - The current is supplied by the I/O port called current sourcing.
2. LED anodes are connected to the power supply and logic 0 from the I/O port turns on the LEDs - The current is received by the chip called current sinking.



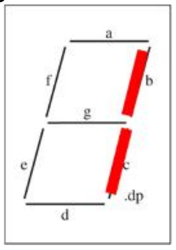
- In a common anode seven-segment LED All anodes are connected together to a power supply and cathodes are connected to data lines
- Logic 0 turns on a segment.

Example:

To display digit 1, so all segments except b and c should be off.


Common Cathode
dp g f e d c b a
D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0
To Data Lines
Through an Interfacing Device

Byte 11111001 = F9H will display digit 1.



8. Explain I/O interfacing Methods

Ans. There are two method of interfacing memory or I/O devices with the microprocessor are as follows:

1) I/O mapped I/O

2) Memory mapped I/O

1) I/O MAPPED I/O

- In this technique, I/O device is treated as an I/O device and memory as memory. Each I/O device uses eight address lines.
- If eight address lines are used to interface to generate the address of the I/O port, then 256 Input/output devices can be interfaced with the microprocessor.
- The 8085 microprocessor has 16 bit address bus, so we can either use lower order address lines (A0 – A7) or higher order address lines(A8 – A15) to address I/O devices. We used lower order address bus & address available on A0 – A7 will be copied on the address lines A8 – A15.
- In I/O mapped I/O, the complete 64 Kbytes of memory can be used to address memory locations separately as the address space is not shared with I/O devices.
- In this interface type, the data transfer is possible between accumulator (A) and I/O devices only. Arithmetic and logical operation are not possible directly.
- As 8 bit device address used, Address decoding is simple so less hardware is required.
- The separate control signals are used to access I/O devices and memory such as IOR, IOW for I/O port and MEMR, MEMW for memory hence memory location are protected from the I/O access.

2) MEMORY MAPPED I/O

- In this technique, I/O devices are treated as memory and memory as memory, hence the address of the I/O devices are as same as that of memory i.e. 16 bit for 8085 microprocessor.
- So, the address space of the memory i.e. 64 Kbytes will be shared by the I/O devices as well as by memory. All 16 address lines i.e. A0-A15 is used to address memory locations as well as I/O devices.
- The control signals MEMR and MEMW are used to access memory devices as well as I/O devices.

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

No	Characteristics	Memory mapped I/O	I/O mapped I/O
1	Device Address	16 bit	8 Bit
2	Control signals for inputs	MEMR & MEMW	IOR & IOW
3	Instruction Available	All memory related instruction : LDA; STA; LDAX; STAX; MOV M,R; ADD M; SUB M	IN and OUT instructions only
4	Data Transfer	Between any register and I/O devices.	Between I/O device and Accumulator only.
5	Maximum Numbers of I/Os Possible	Memory Map (64K) is shared between I/Os and System memory.	I/O Mapped is independent of memory map; 256 Input and 256 output devices can be connected.
6	Execution Speed	13 T-State (LDA, STA, ..) 7 T-State (MOV M,R)	10 T-State
7	Hardware Requirement	More hardware is needed to decode 16 bit address	Less hardware is needed to decode 8 bit address
8	Other Feature	Arithmetic and logical operations are directly performed with I/O devices.	Not available

MODULE 3

INTRODUCTION TO EMBEDDED SYSTEM

System

A system is an arrangement in which all its unit assemble work together according to a set of rules. It can also be defined as a way of working, organizing or doing one or many tasks according to a fixed plan. For example, a watch is a time displaying system. Its components follow a set of rules to show time. If one of its parts fails, the watch will stop working. So we can say, in a system, all its subcomponents depend on each other.

Embedded System

As its name suggests, Embedded means something that is attached to another thing. An embedded system can be thought of as a computer hardware system having software embedded in it. An embedded system can be an independent system or it can be a part of a large system. An embedded system is a microcontroller or microprocessor based system which is designed to perform a specific task. For example, a fire alarm is an embedded system; it will sense only smoke.

An embedded system has three components −

- It has hardware.
- It has application software.
- It has Real Time Operating system (RTOS) that supervises the application software and provide mechanism to let the processor run a process as per scheduling by following a plan to control the latencies. RTOS defines the way the system works. It sets the rules during the execution of application program. A small scale embedded system may not have RTOS.

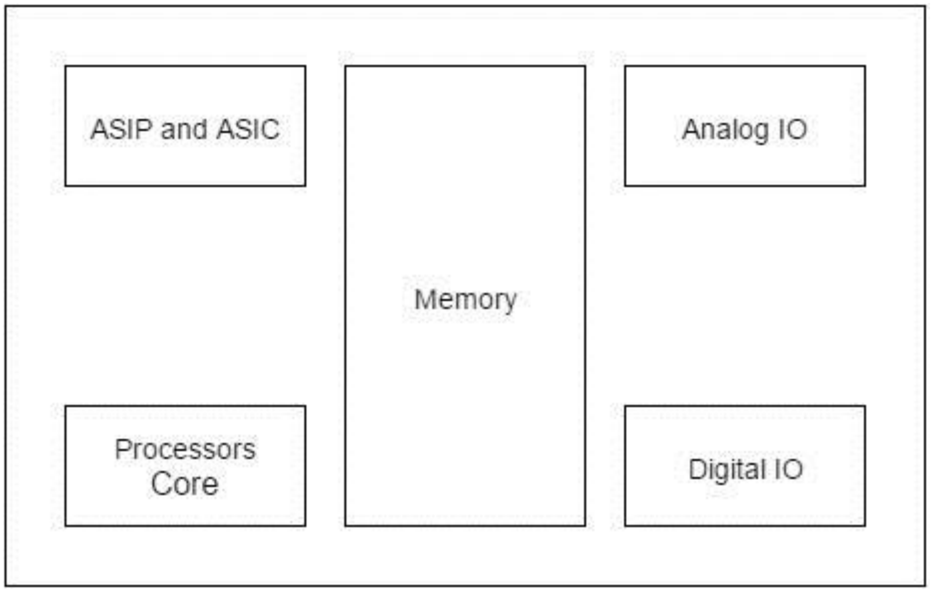
So we can define an embedded system as a Microcontroller based, software driven, reliable, real-time control system.

Characteristics of an Embedded System

- **Single-functioned** − An embedded system usually performs a specialized operation and does the same repeatedly. For example: A pager always functions as a pager.
- **Tightly constrained** − All computing systems have constraints on design metrics, but those on an embedded system can be especially tight. Design metrics is a measure of an implementation's features such as its cost, size, power, and performance. It must be of a

size to fit on a single chip, must perform fast enough to process data in real time and consume minimum power to extend battery life.

- **Reactive and Real time** − Many embedded systems must continually react to changes in the system's environment and must compute certain results in real time without any delay. Consider an example of a car cruise controller; it continually monitors and reacts to speed and brake sensors. It must compute acceleration or de-accelerations repeatedly within a limited time; a delayed computation can result in failure to control of the car.
- **Microprocessors based** − It must be microprocessor or microcontroller based.
- **Memory** − It must have a memory, as its software usually embeds in ROM. It does not need any secondary memories in the computer.
- **Connected** − It must have connected peripherals to connect input and output devices.
- **HW-SW systems** − Software is used for more features and flexibility. Hardware is used for performance and security.



Advantages

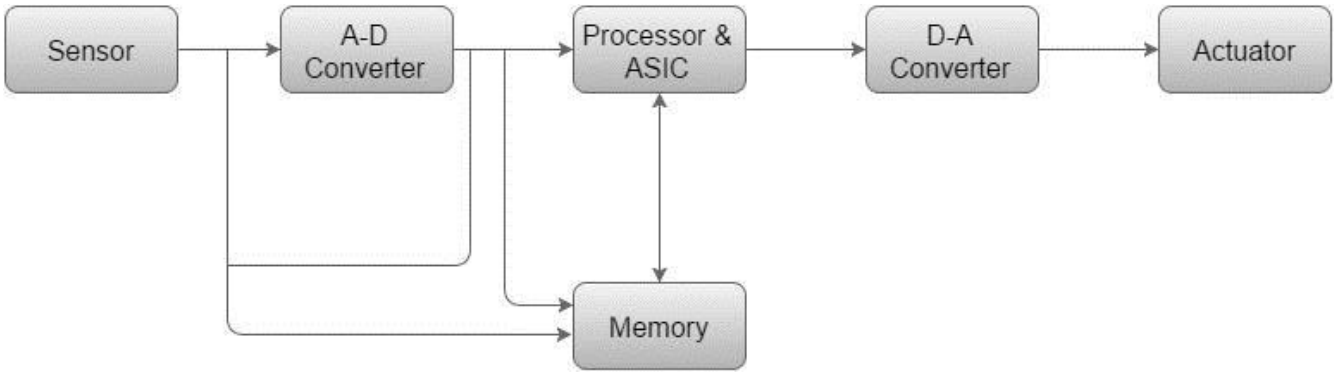
- Easily Customizable
- Low power consumption
- Low cost
- Enhanced performance

Disadvantages

- High development effort
- Larger time to market

Basic Structure of an Embedded System

The following illustration shows the basic structure of an embedded system −



- **Sensor** − It measures the physical quantity and converts it to an electrical signal which can be read by an observer or by any electronic instrument like an A2D converter. A sensor stores the measured quantity to the memory.
- **A-D Converter** − An analog-to-digital converter converts the analog signal sent by the sensor into a digital signal.
- **Processor & ASICs** − Processors process the data to measure the output and store it to the memory.
- **D-A Converter** − A digital-to-analog converter converts the digital data fed by the processor to analog data
- **Actuator** − An actuator compares the output given by the D-A Converter to the actual (expected) output stored in it and stores the approved output.

Microprocessor vs Microcontroller

Three Key Differences Between Microcontrollers and Microprocessors

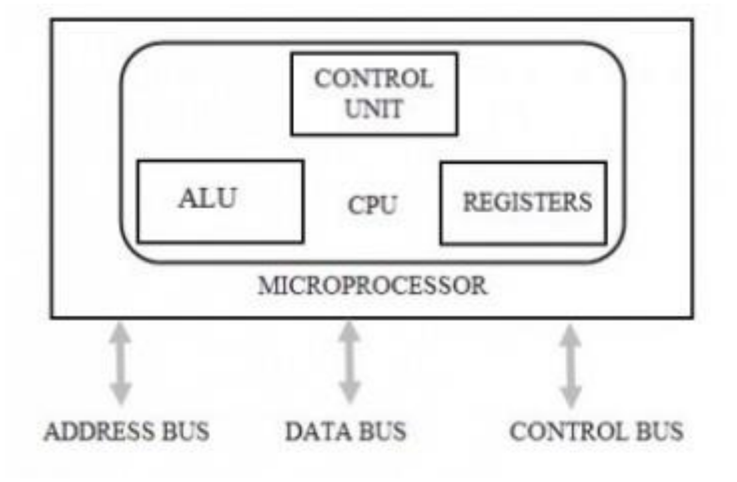
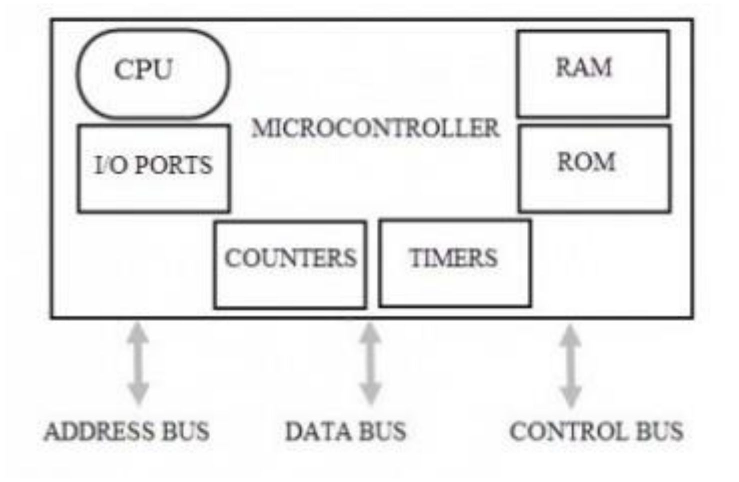
- **Cost:** Generally, microcontrollers cost less than microprocessors. Microprocessors are typically manufactured for use with more expensive devices. They are also significantly more complex, as they are meant to perform a variety of computational tasks while microcontrollers usually perform a dedicated function. With a microcontroller, engineers write and compile the code intended for the specific application and upload it into the microcontroller, which internally houses all of the necessary computing features and components to execute the code.

- **Speed:** When it comes to clock speed, there is a significant difference. This relates back to the idea that microcontrollers are meant to handle a specific task or application, while a microprocessor is meant for more complex, robust, and unpredictable computing tasks. That means using just the right amount of speed and power to get the job done – no more and no less. As a result, many microprocessors are clocking speeds of up to 4 GHz while microcontrollers can operate with much slower speeds of 200 MHz or less.

- **Power Consumption:** One of the key advantages associated with microcontrollers is their low power consumption. A computer processor that performs a dedicated task requires less speed, and therefore less power, than a processor with robust computational capacity. Power consumption plays an important role in implementation design: a processor that consumes a lot of power may need to be plugged in or supported by an external power supply, whereas a processor that consumes limited power could be powered for a long time by just a small battery.

Basic Difference

	Microprocessor	Microcontroller
	Microprocessor acts as the heart of computer system.	The microcontroller acts as the heart of the embedded system.
	It is a processor in which memory and I/O output component is connected externally.	It is a controlling device in which memory and I/O output component are present internally.
	Since memory and I/O output is to be connected externally. Therefore the circuit is more complex.	Since on-chip memory and I/O output component is available. Therefore the circuit is less complex.
	It cannot be used in a compact system. Therefore microprocessor is inefficient.	It can be used in a compact system. Therefore microcontroller is more efficient.
	The microprocessor has fewer registers. Therefore most of the operations are memory-based.	The microcontroller has more registers. Therefore a program is easier to write.
	A microprocessor having a zero status flag.	A microcontroller has no zero flag.
	It is mainly used in personal computers.	It is mainly used in washing machines, air conditioners etc.

Microprocessor	Microcontroller
 CONTROL UNIT ALU CPU REGISTERS MICROPROCESSOR ADDRESS BUS DATA BUS CONTROL BUS	 CPU RAM MICROCONTROLLER I/O PORTS ROM COUNTERS TIMERS ADDRESS BUS DATA BUS CONTROL BUS
Microprocessor assimilates the function of a central processing unit (CPU) on to a single integrated circuit (IC).	A microcontroller can be considered as a small computer that has a processor and some other components in order to make it a computer.
Microprocessors are mainly used in designing general-purpose systems from small to large and complex systems like supercomputers.	Microcontrollers are used in automatically controlled devices.
Microprocessors are basic components of personal computers.	Microcontrollers are generally used in embedded systems
The computational capacity of the microprocessor is very high. Hence can perform complex tasks.	Less computational capacity when compared to microprocessors. Usually used for simpler tasks.
A microprocessor-based system can perform numerous tasks.	A microcontroller based system can perform single or very few tasks.
Microprocessors have integrated Math Coprocessor. Complex mathematical calculations which involve floating point can be performed with great ease.	Microcontrollers do not have math coprocessors. They use software to perform floating-point calculations which slows down the device.
The main task of the microprocessor is to perform the instruction cycle repeatedly. This includes fetch, decode and execute.	In addition to performing the tasks of fetch, decode and execute, a microcontroller also controls its environment based on the output of the instruction cycle.
In order to build or design a system (computer), a microprocessor has to be connected externally to some other components like Memory (RAM and ROM) and Input / Output ports.	The IC of a microcontroller has memory (both RAM and ROM) integrated into it along with some other components like I / O devices and timers.
The overall cost of a system built using a microprocessor is high. This is because of the requirement of external components.	The cost of a system built using a microcontroller is less as all the components are readily available.
Generally, power consumption and dissipation	Power consumption is less.

are high because of the external devices. Hence it requires an external cooling system.	
The clock frequency is very high usually in the order of Giga Hertz.	The clock frequency is less usually in the order of MegaHertz.
Instruction throughput is given higher priority than interrupt latency.	In contrast, microcontrollers are designed to optimize interrupt latency.
Have few bit manipulation instructions	Bit manipulation is powerful and widely used feature in microcontrollers. They have numerous bit manipulation instructions.
Generally, microprocessors are not used in real-time systems as they are severely dependent on several other components.	Microcontrollers are used to handle real-time tasks as they are single programmed, self-sufficient and task-oriented devices.

Current Trends in Embedded Systems

An embedded system is an application-specific system designed with a combination of hardware and software to meet real-time constraints. The key characteristics of embedded industrial systems include speed, security, size, and power. The major trends in the embedded systems market revolve around the improvement of these characteristics.

To give context into how large the embedded systems industry is, here are a few statistics:

- The global market for the embedded systems industry was valued at $68.9 billion in 2017 and is expected to rise to $105.7 billion by the end of 2025.
- 40% of the industrial share for embedded systems market is shared by the top 10 vendors.
- In 2015, embedded hardware contributed to 93% of the market share and it is expected to dominate the market over embedded software in the upcoming years as well.

CHALLENGES IN EMBEDDED SYSTEM DESIGN

→ The challenges that are encounted during the desig process are not Computer related, rather they are mechanical or electrical.

→ of these, the most challenging areas are.

(a) Hardware. (b) deadlines. (c) power Consumptio

(d) Upgradeability (e) Reliability.

(a) Hardware : (How much hardware do we need ?)
→ The choice of The hardware plays a major rol
in meeting manufacturing cost constraints and
Performance deadlines.
→ The choice shouldn't be too expensive or too chea
rather it should be exact to meet the deadlines.

(b) Deadlines : (How do we meet deadlines)
→ Meeting deadlines is another challenge in designi-
an embedded Computing system.
→ One method of meeting deadlines is the "brute
force method". In this method the speed of the
h/w is increased so that the program runs faster
of course, that makes the system more expensive

(C) Power Consumption : (How do we minimize Power Consum-ption
→ In battery-powered applications, power consumptio
is extremely important, even in non-battery
applications, excess power consumption can increase
heat dissipation. One way to make it consume less

power is to run slowly. but slowing down leads to missing deadlines. Therefore careful design must be done to meet performance goals.

(d) <u>Design for Upgradability</u>: (How do we design for Upgradability?)

→ Designing a machine that can match with Upgrades in software is another challege.

→ The same hardware may be used with different versions of the software.

(e) <u>Reliability</u>: (Does it really works ?)

→ Reliability plays a vital role, if products are safety-critical products. if these products lack reliability then the consequences can be very dangerous.

→ To ensure reliability, the designer must maintain an equilibrium between cost and the time.

eal time system is defined as a system in which job has deadline, job has to finished by the deadline (strictly finished). If a result is delayed, huge loss may happen.

1. Hard Real Time System :
Hard real time is a system whose operation is incorrect whose result is not produce according to time constraint.
For example,

1. Air Traffic Control

2. Medical System

2. Soft Real Time System :
Soft real time system is a system whose operation is degrade if results are not produce according to the specified timing requirement.
For example<

1. Multimedia Transmission and Reception

2. Computer Games

Difference between Hard real time and Soft real time system :

HARD REAL TIME SYSTEM	SOFT REAL TIME SYSTEM
In hard real time system, the size of data file is small or medium.	In soft real time system, the size of data file is large.
In this system response time is in millisecond.	In this system response time are higher.
Peak load performance should be predictable.	In soft real time system, peak load can be tolerated.
In this system safety is critical.	In this system safety is not critical.

HARD REAL TIME SYSTEM	SOFT REAL TIME SYSTEM
A hard real time system is very restrictive.	A Soft real time system is less restrictive.
In case of an error in a hard real time system, the computation is rolled back.	In case of an soft real time system, computation is rolled back to previously established a checkpoint.
Satellite launch, Railway signaling system etc.	DVD player, telephone switches, electronic games etc.

EMBEDDED SYSTEM DESIGN PROCESSES :

The major steps in the design process are summarized in the top down view as follows .

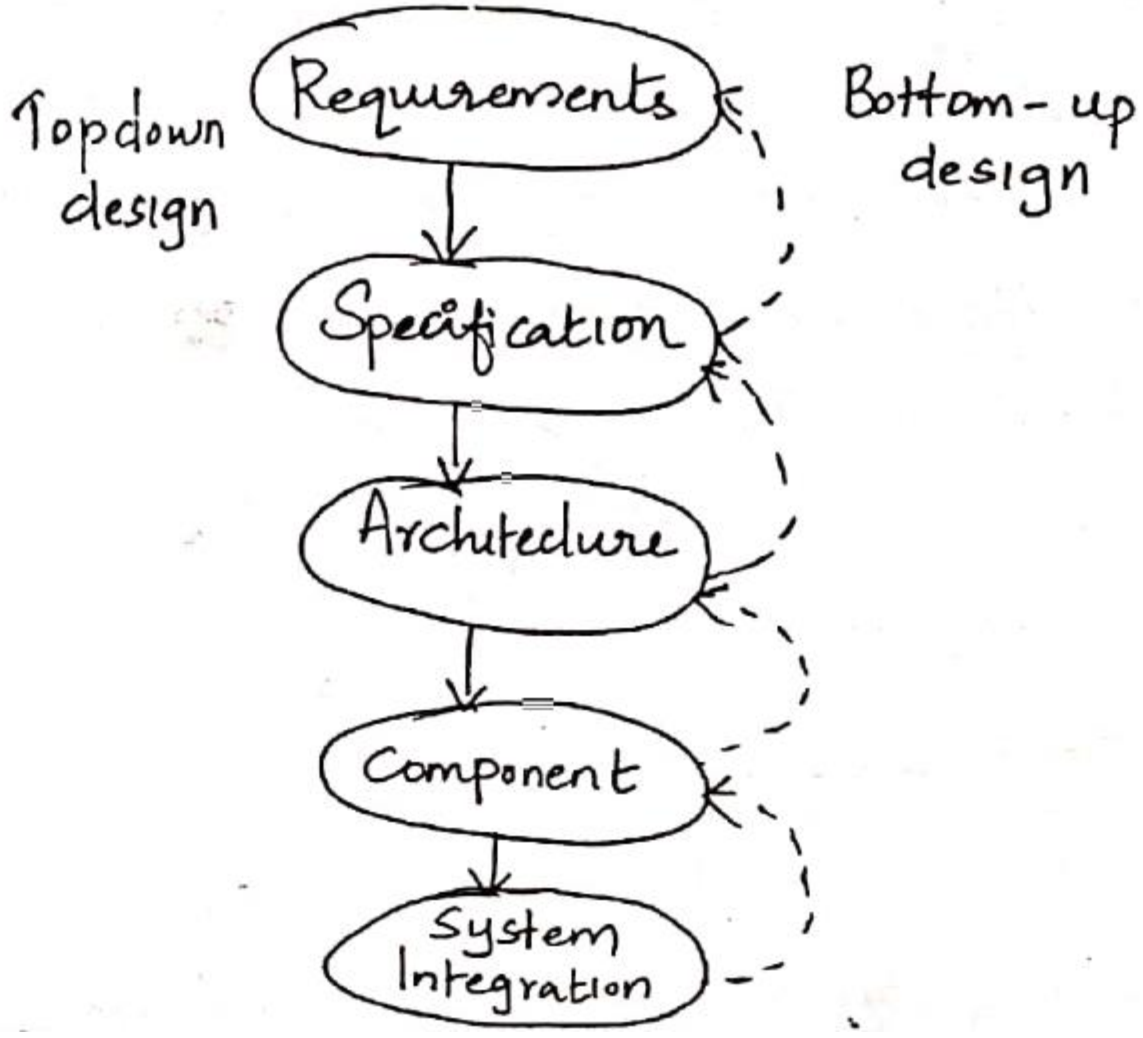


Fig : Major levels of abstraction in the design Process .

Requirements :

→ The requirement phase of the design process Capture "what to design".

→ Informal description gathered from customers is Known as "requirement".

→ Requirement can be of 2 types ie,

(a) Functional requirements

(b) Non-functional requirements

→ Some of the non-functional requirement are

① Power Consumption :

→ In the requiremet stage, power can be Specifiec interms of battery life.

→ However, the allowable ~~Voltage~~ wattage can't be defined by the customer.

(2) Physical size and weight :

→ Depending on the application, the physical size and weight of the final system can vary a lot.

⇒ if the application involves a handheld device then there are constraint on both the size and weight of the device. However, if it is an Industrial control system then there is no constraint on the size and weight.

(3) Performance

→ the cost and usability of the system are effected by it speed. The performance metries can be combination of soft metrics and hard metrics

(4) Cost :

→ The system purchase price or the target cost is very important. Complete cost or simply cost includes the following two components

a) Manufacturing cost

b) NRE (Non-Recurring Engineering) cost.

- Manufacturing cost is the cost of the Component
- NRE costs are the cost of hiring personnel and other design related costs.

Requirements Validation :

→ Requirements Validation requires phychological skills, as it deals with understanding what custome

(5)

→ In the System requirement, the user interface par can be done by creating "Mock-up".

→ In order to stimulate functionality in a restricte area the mock-up make use of scanned data.

→ This mock up can be executed either by PC o work-station.

Requirement form / Requirement chart

→ The requirement form / chart acts like a checkli when the project is in initial stages. A sample form is given below.

Name :	___	
Purpose :	___	(what the System Supposed to do
Inputs :	___	
Outputs :	___	
Functions :	___	(functionality of the System).
Performance :	___	
Manufacturing Cost :	___	
Power :	___	
Physical size & wt :	___	

Fig : Sample Requirements form .

Name :- Name not only describe the purpose of the machine , but also helpful while Commiting about the project .

Purpose : This should be a brief one or two line description of what the system is supposed to do .

Inputs and Outputs : this field requires information about type of I/o devices , data characteristics and type af data .

Functions : A detailed description about the functionality of the machine is described in this field .

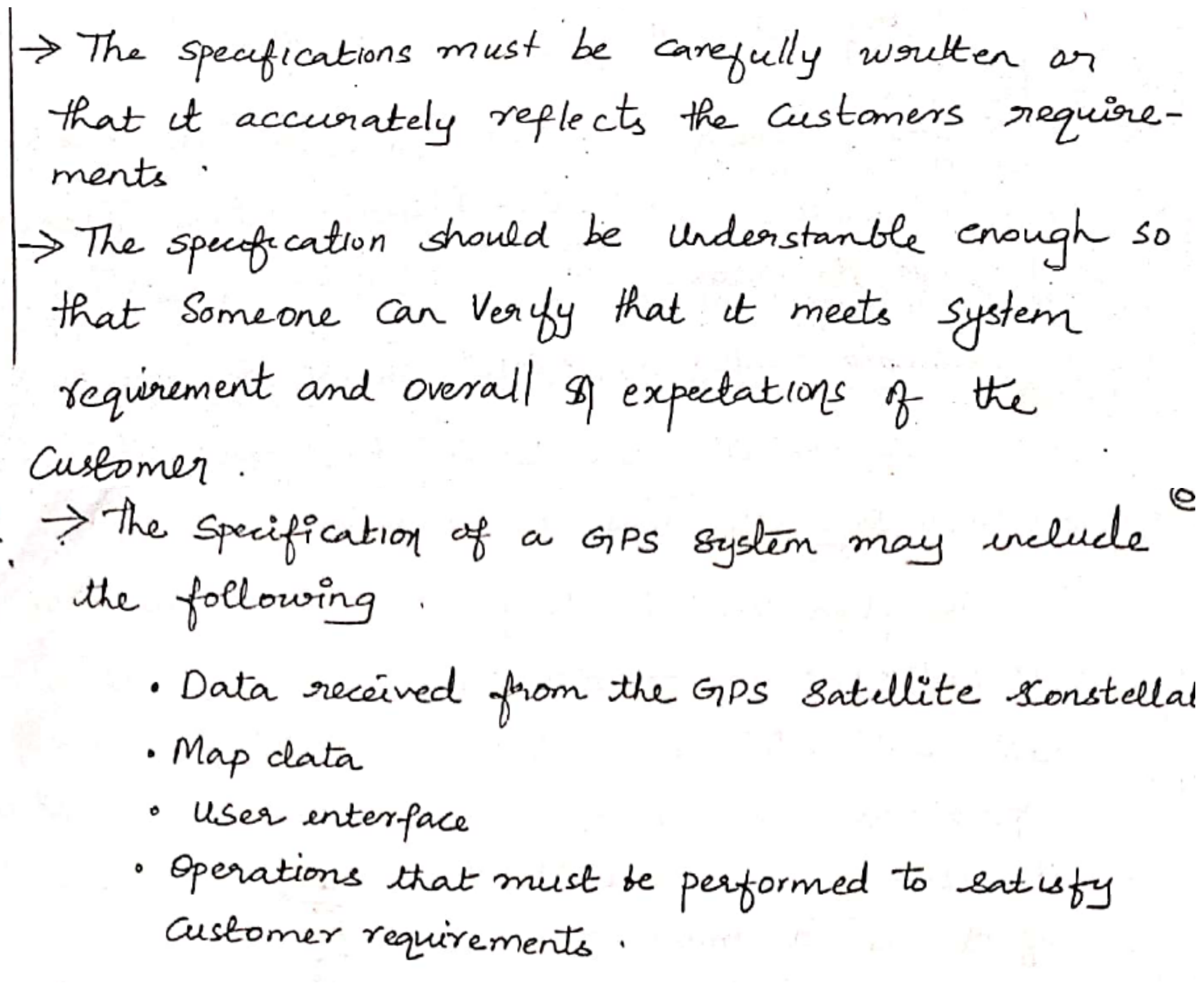
Performance : Inorder to assure proper functionalit performance requirements should be Identified before hand and they must be measured carefully .

Physical Size and weight : Inorder to take archi tedural decisions , the approximate physical size & weight of the system is important .

Power : The rough idea about power consumptio can be very helpful . Decision about whether the System is battery based or non-battery is important here .

SPECIFICATIONS

→ Specifications serves as the contract between the customer and the architect .



→ The specifications must be carefully written so that it accurately reflects the customers requirements.

→ The specification should be understanble enough so that Someone can Verify that it meets system requirement and overall ~~S~~ expectations of the customer.

→ The specification of a GPS system may include the following.

- Data received from the GPS Satellite Constellat
- Map data
- User enterface
- Operations that must be performed to satisfy customer requirements.

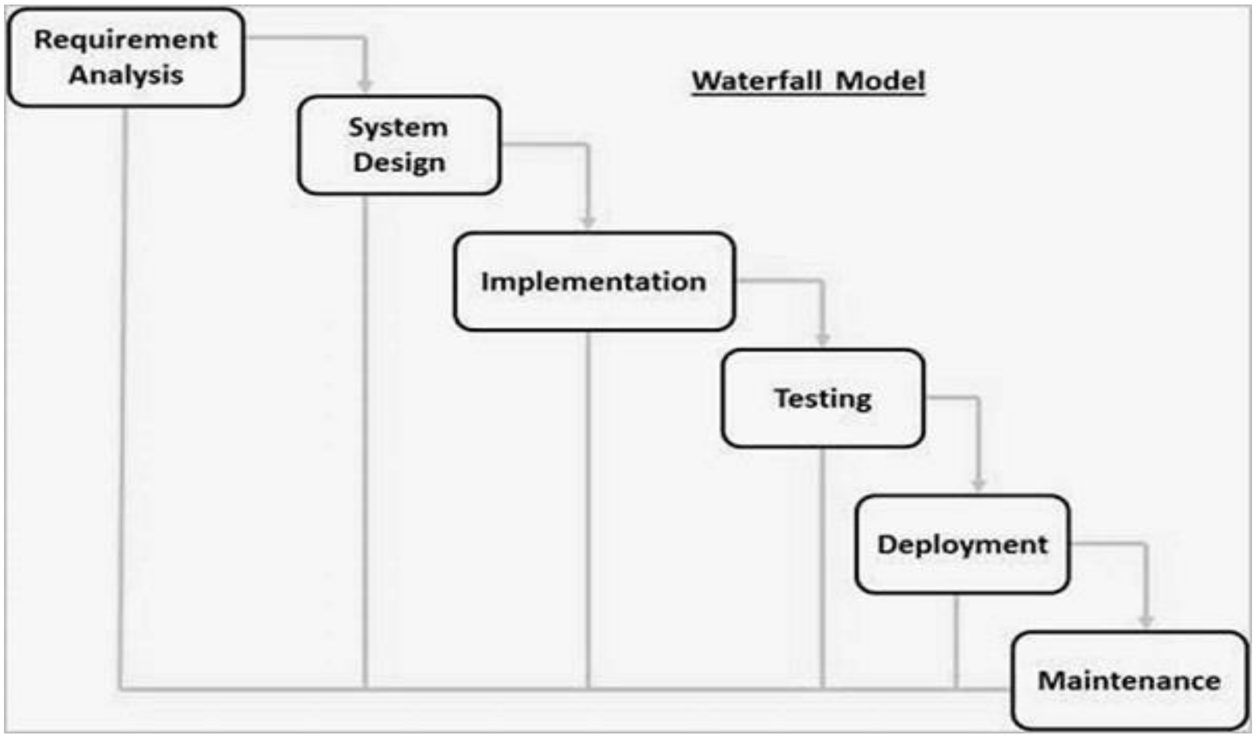
<u>LIFE CYCLE MANAGEMENT</u>

WATER FALL MODEL

- The Waterfall Model was the first Process Model to be introduced. It is also referred to as a **linear-sequential life cycle model**. It is very simple to understand and use. In a waterfall model, each phase must be completed before the next phase can begin and there is no overlapping in the phases.
- The Waterfall model is the earliest SDLC approach that was used for software development.
- The waterfall Model illustrates the software development process in a linear sequential flow. This means that any phase in the development process begins only if the previous phase is complete. In this waterfall model, the phases do not overlap.

- Waterfall approach was first SDLC Model to be used widely in Software Engineering to ensure success of the project.
- In "The Waterfall" approach, the whole process of software development is divided into separate phases.
- In this Waterfall model, typically, the outcome of one phase acts as the input for the next phase sequentially.

The following illustration is a representation of the different phases of the Waterfall Model.



The sequential phases in Waterfall model are

- **Requirement Gathering and analysis** − All possible requirements of the system to be developed are captured in this phase and documented in a requirement specification document.
- **System Design** − The requirement specifications from first phase are studied in this phase and the system design is prepared. This system design helps in specifying hardware and system requirements and helps in defining the overall system architecture.
- **Implementation** − With inputs from the system design, the system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality, which is referred to as Unit Testing.

- **Integration and Testing** − All the units developed in the implementation phase are integrated into a system after testing of each unit. Post integration the entire system is tested for any faults and failures.
- **Deployment of system** − Once the functional and non-functional testing is done; the product is deployed in the customer environment or released into the market.
- **Maintenance** − There are some issues which come up in the client environment. To fix those issues, patches are released. Also to enhance the product some better versions are released. Maintenance is done to deliver these changes in the customer environment.
- All these phases are cascaded to each other in which progress is seen as flowing steadily downwards (like a waterfall) through the phases.
- The next phase is started only after the defined set of goals are achieved for previous phase and it is signed off, so the name "Waterfall Model".
- In this model, phases do not overlap.

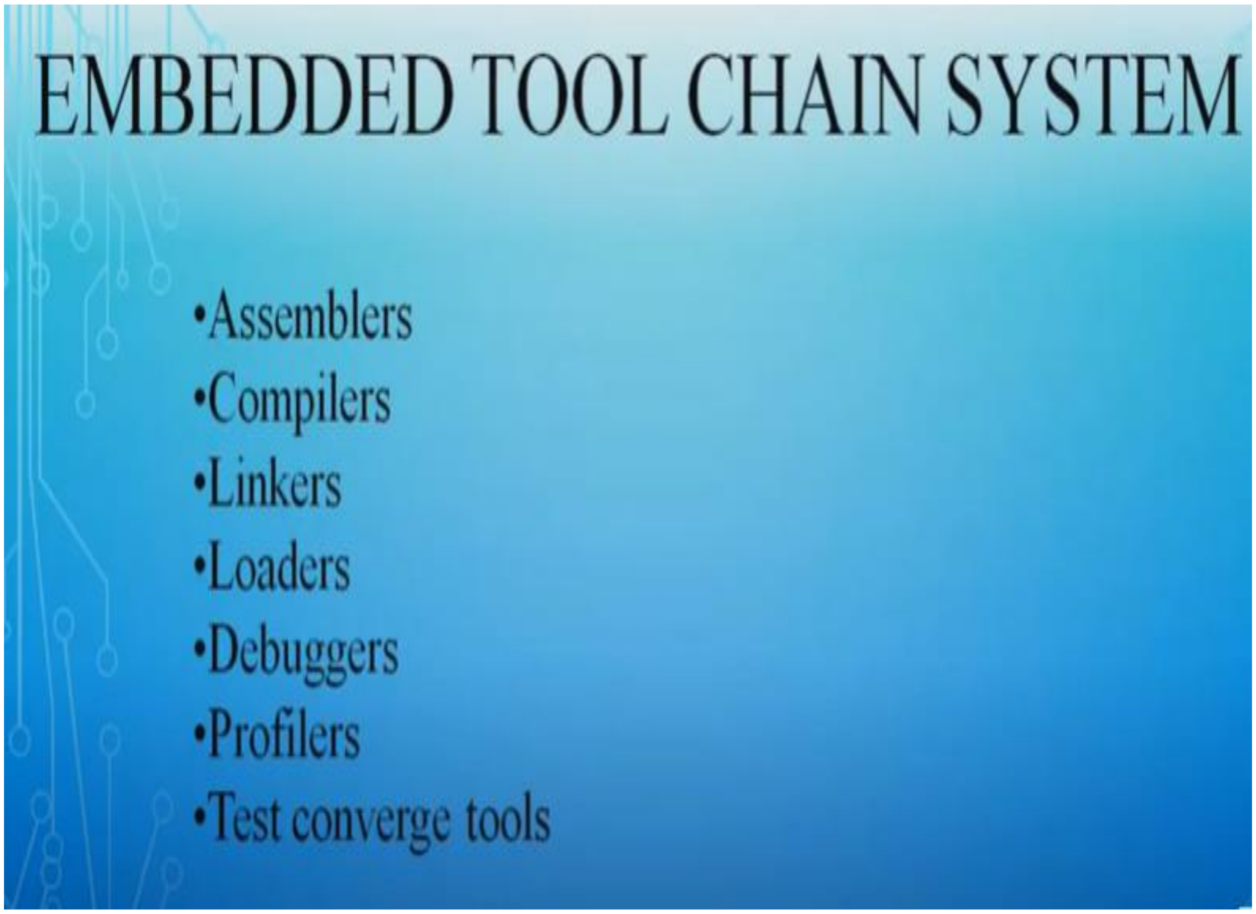
Waterfall Model – Advantages

- Simple and easy to understand and use
- Easy to manage due to the rigidity of the model. Each phase has specific deliverables and a review process.
- Phases are processed and completed one at a time.
- Works well for smaller projects where requirements are very well understood.
- Clearly defined stages.
- Well understood milestones.
- Easy to arrange tasks.
- Process and results are well documented.

Waterfall Model – Disadvantages

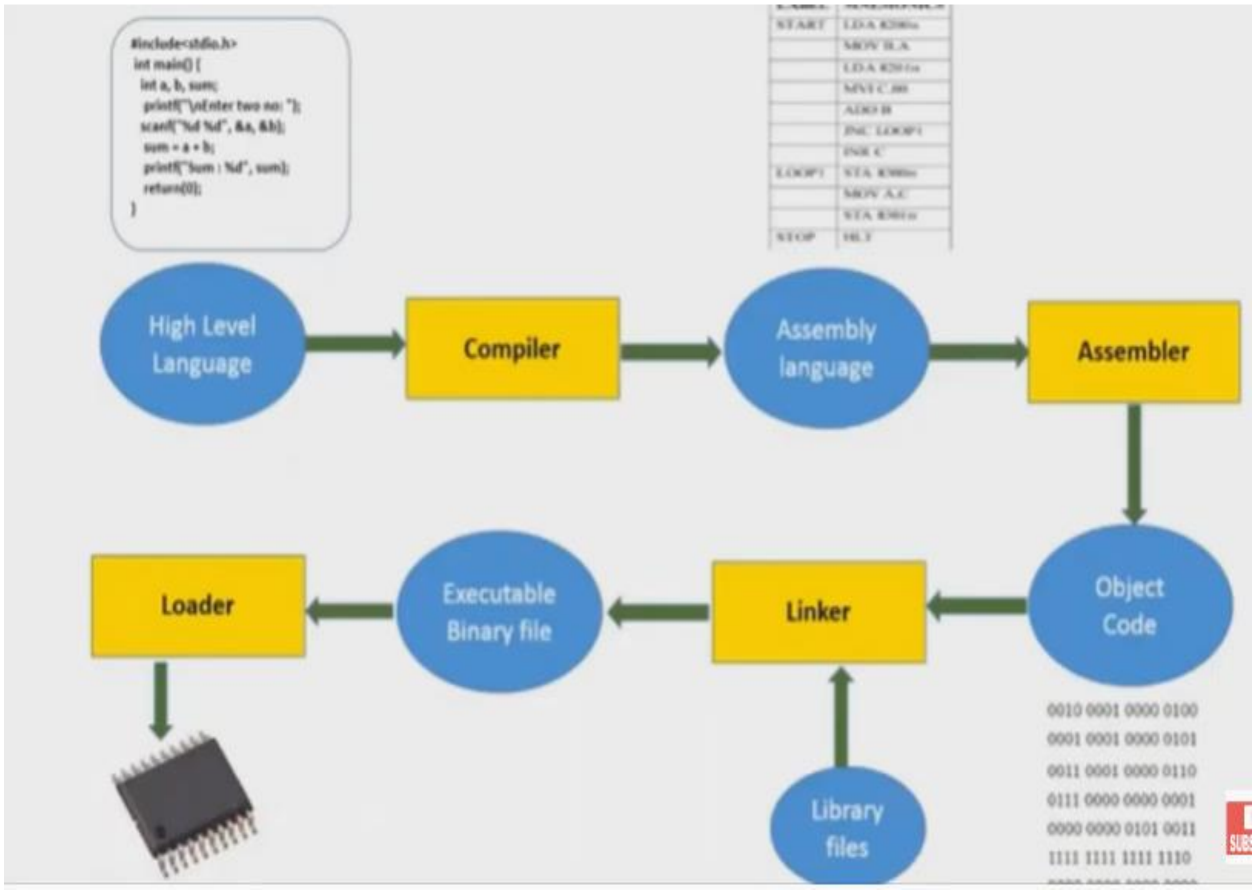
- No working software is produced until late during the life cycle.
- High amounts of risk and uncertainty.
- Not a good model for complex and object-oriented projects.
- Poor model for long and ongoing projects.

- Not suitable for the projects where requirements are at a moderate to high risk of changing. So, risk and uncertainty is high with this process model.
- It is difficult to measure progress within stages.
- Cannot accommodate changing requirements.
- Adjusting scope during the life cycle can end a project.
- Integration is done as a "big-bang. at the very end, which doesn't allow identifying any technological or business bottleneck or challenges early.



EMBEDDED TOOL CHAIN SYSTEM

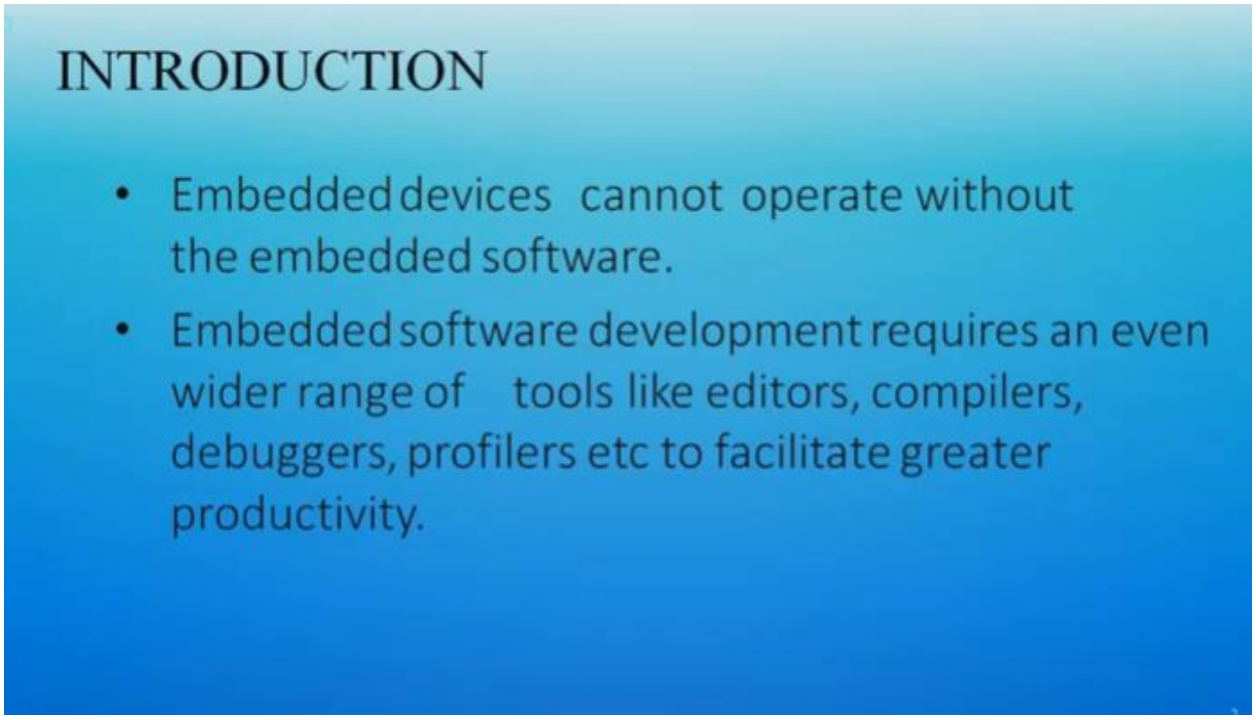
- Assemblers
- Compilers
- Linkers
- Loaders
- Debuggers
- Profilers
- Test converge tools



```
#include<stdio.h>
 int main() {
  int a, b, sum;
   printf("\nEnter two no: ");
  scanf("%d %d", &a, &b);
   sum = a + b;
   printf("Sum : %d", sum);
   return(0);
}
```

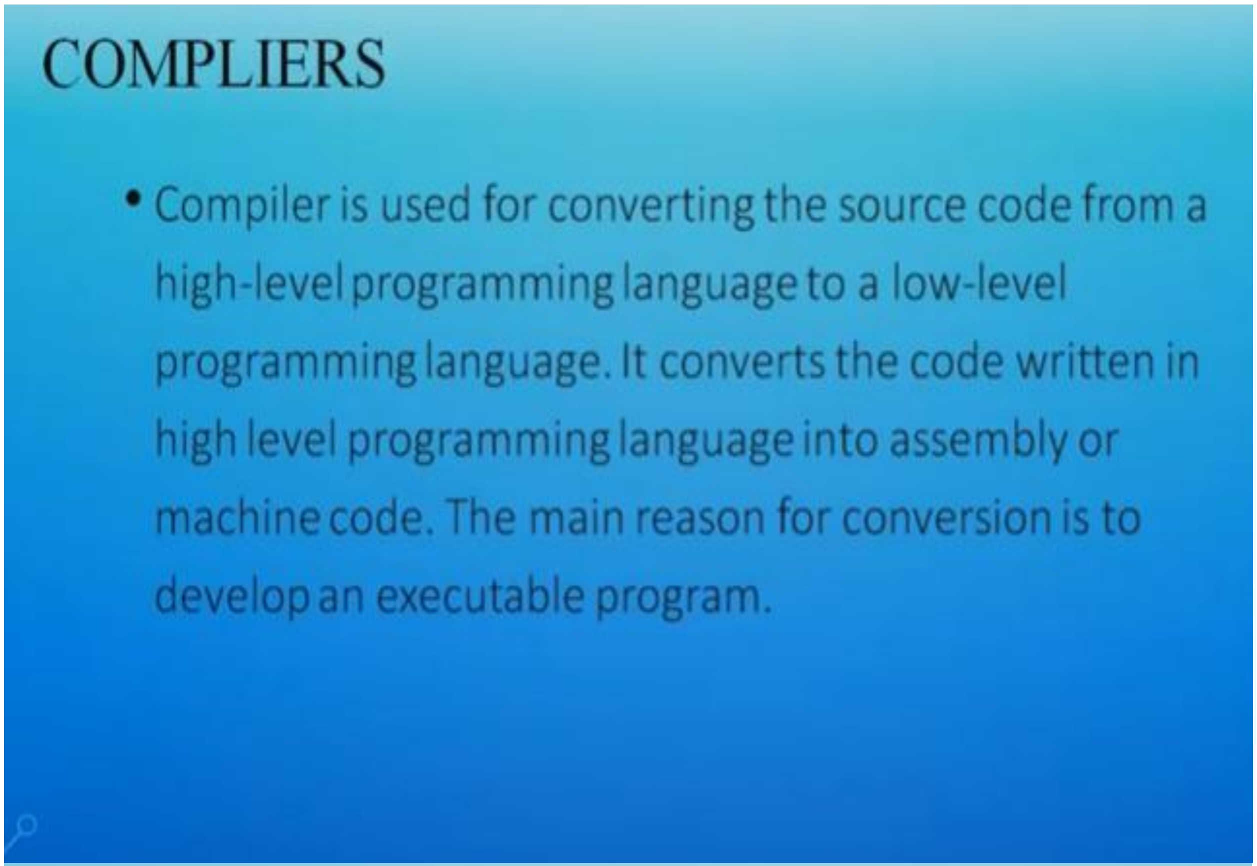
START	LDA 8200H
	MOV B,A
	LDA 8201H
	MVI C,00
	ADD B
	JNC LOOP1
	INR C
LOOP1	STA 8300H
	MOV A,C
	STA 8301H
STOP	HLT

```
0010 0001 0000 0100
0001 0001 0000 0101
0011 0001 0000 0110
0111 0000 0000 0001
0000 0000 0101 0011
1111 1111 1111 1110
```



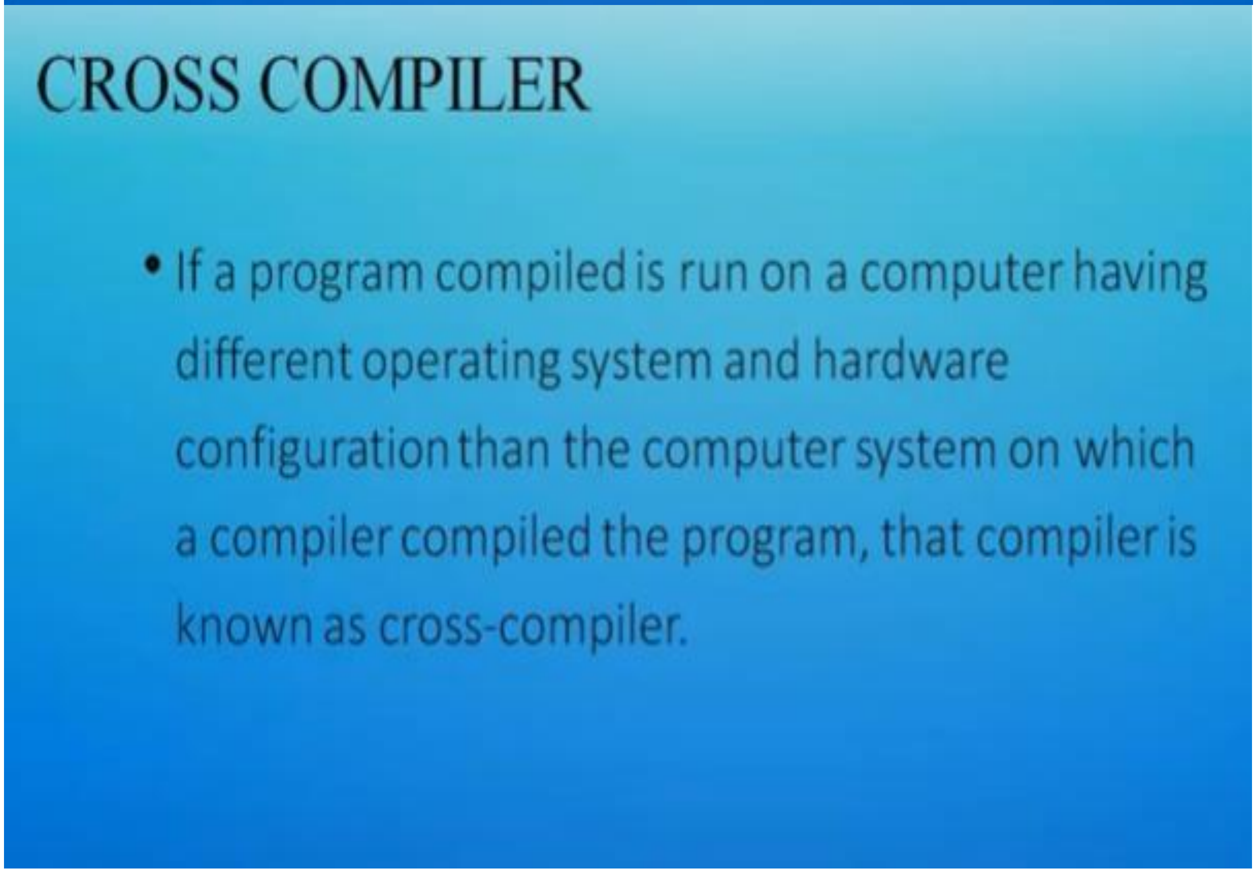
INTRODUCTION

- Embedded devices cannot operate without the embedded software.
- Embedded software development requires an even wider range of tools like editors, compilers, debuggers, profilers etc to facilitate greater productivity.



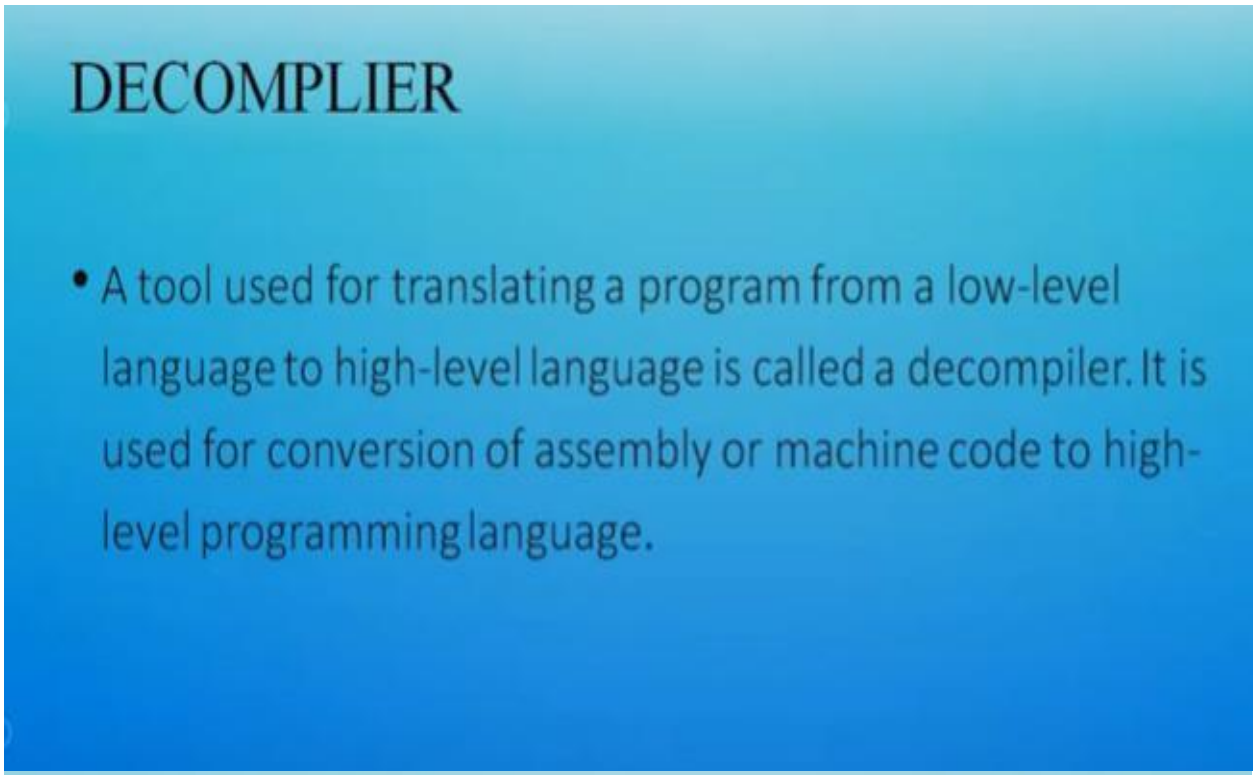
COMPLIERS

- Compiler is used for converting the source code from a high-level programming language to a low-level programming language. It converts the code written in high level programming language into assembly or machine code. The main reason for conversion is to develop an executable program.



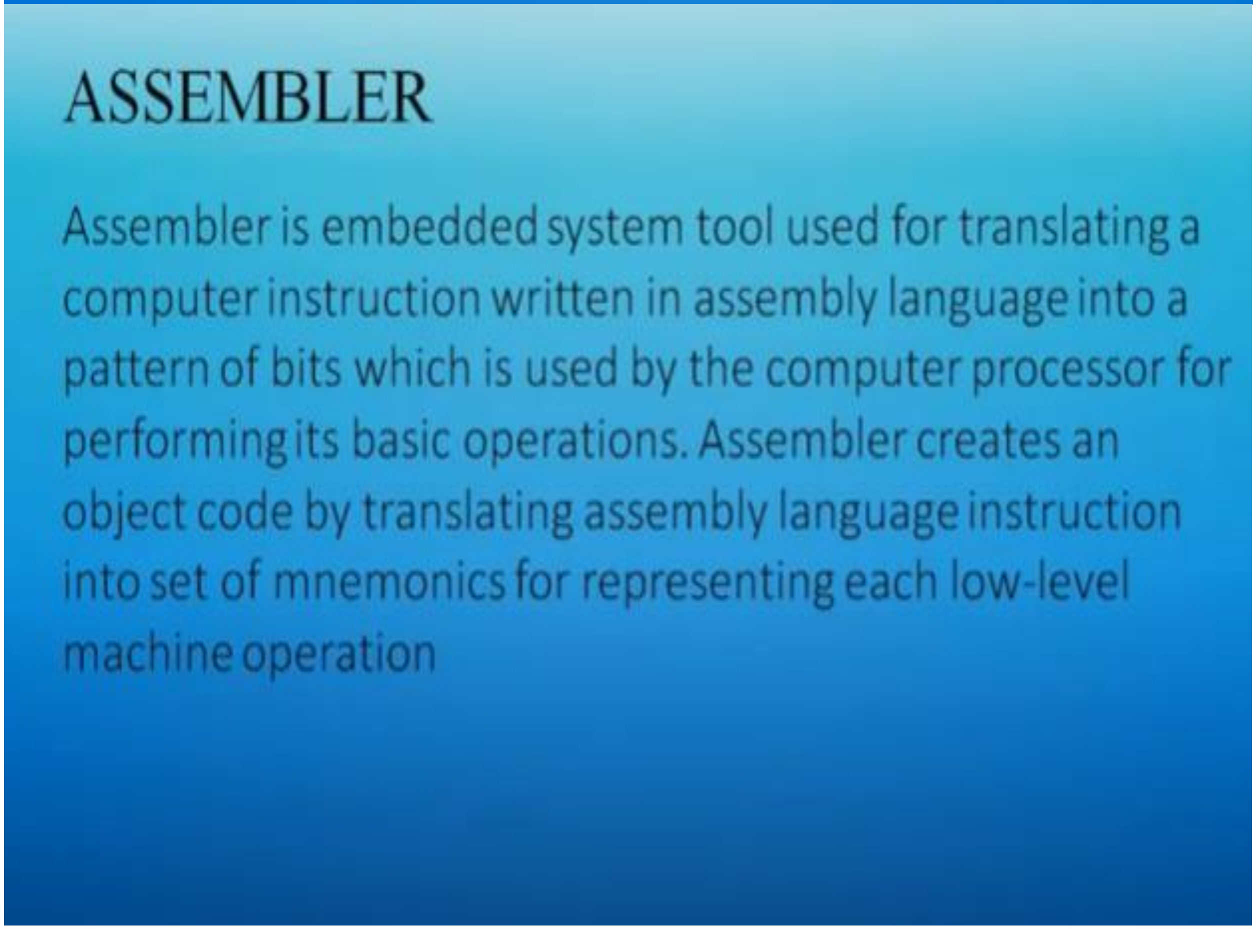
CROSS COMPILER

- If a program compiled is run on a computer having different operating system and hardware configuration than the computer system on which a compiler compiled the program, that compiler is known as cross-compiler.



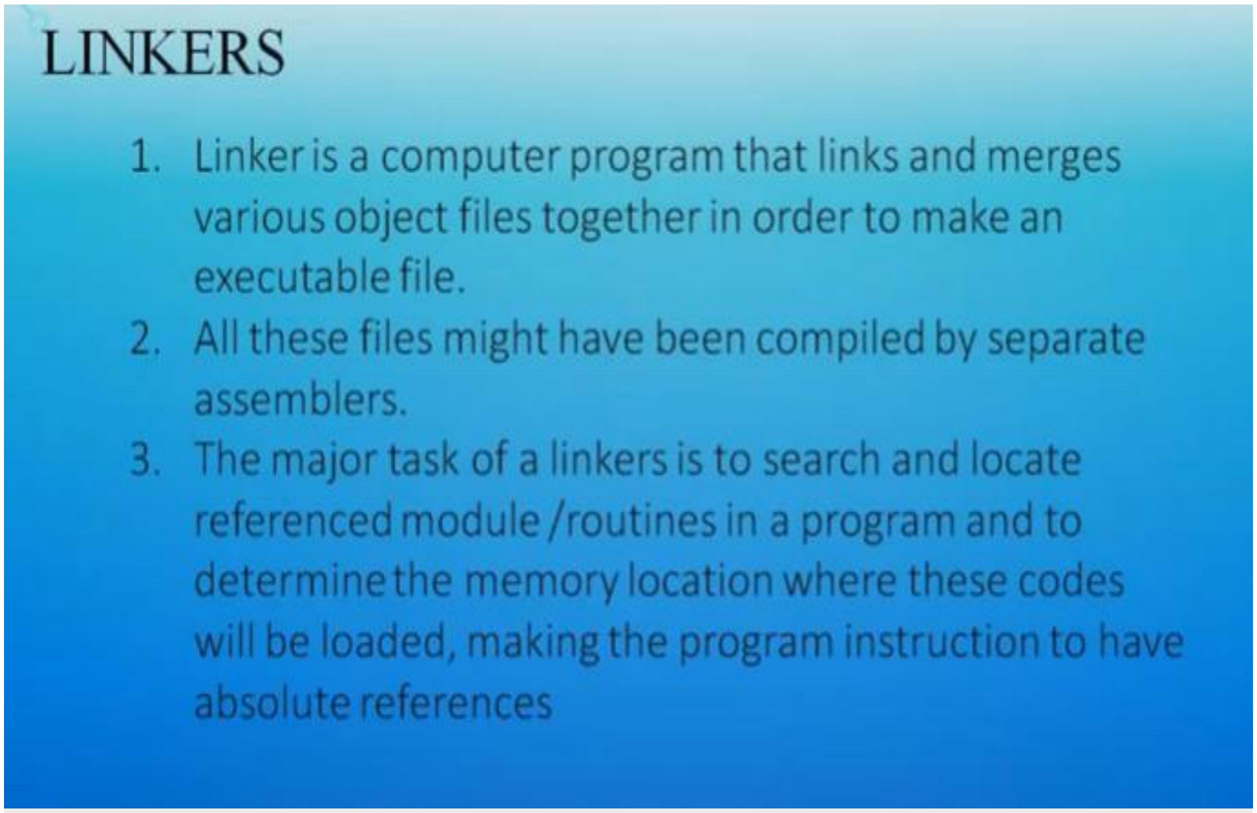
DECOMPLIER

- A tool used for translating a program from a low-level language to high-level language is called a decompiler. It is used for conversion of assembly or machine code to high-level programming language.



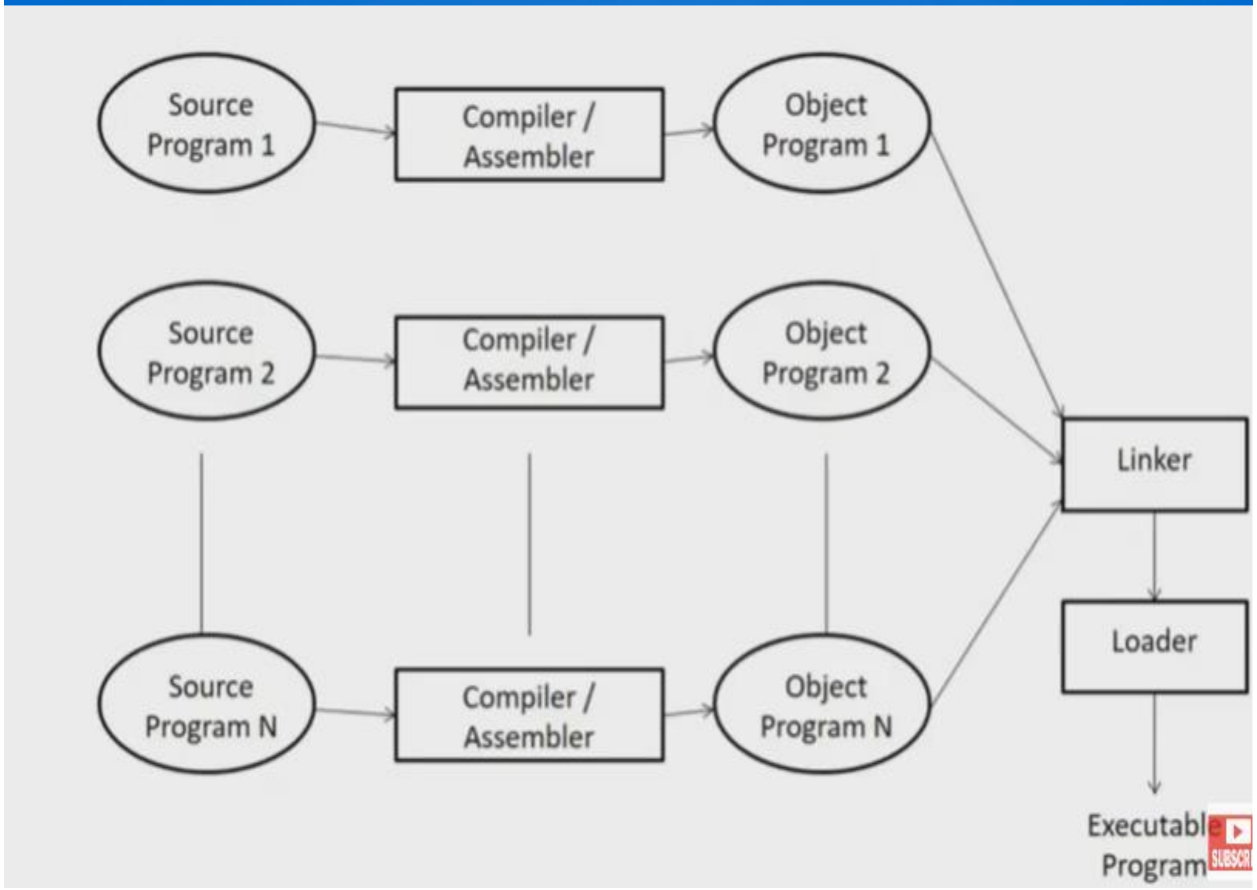
ASSEMBLER

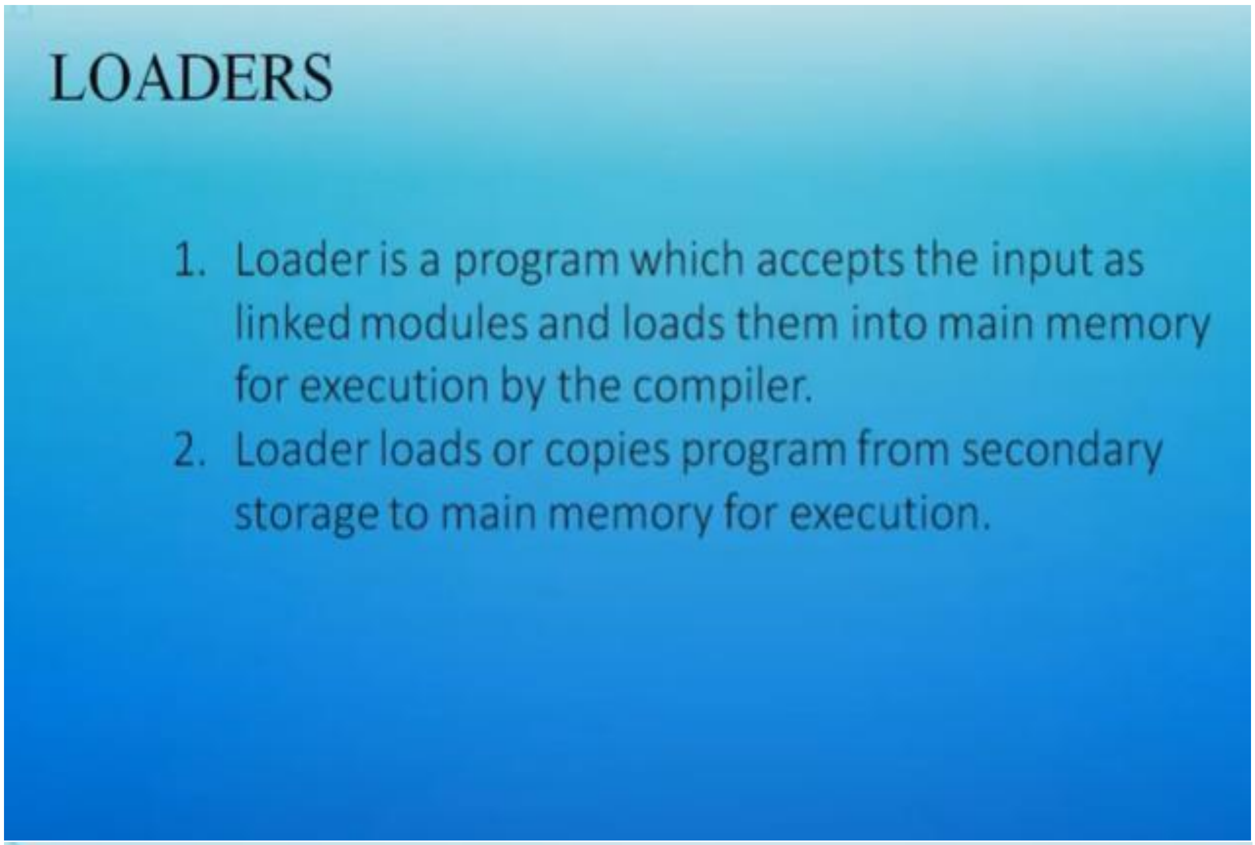
Assembler is embedded system tool used for translating a computer instruction written in assembly language into a pattern of bits which is used by the computer processor for performing its basic operations. Assembler creates an object code by translating assembly language instruction into set of mnemonics for representing each low-level machine operation



LINKERS

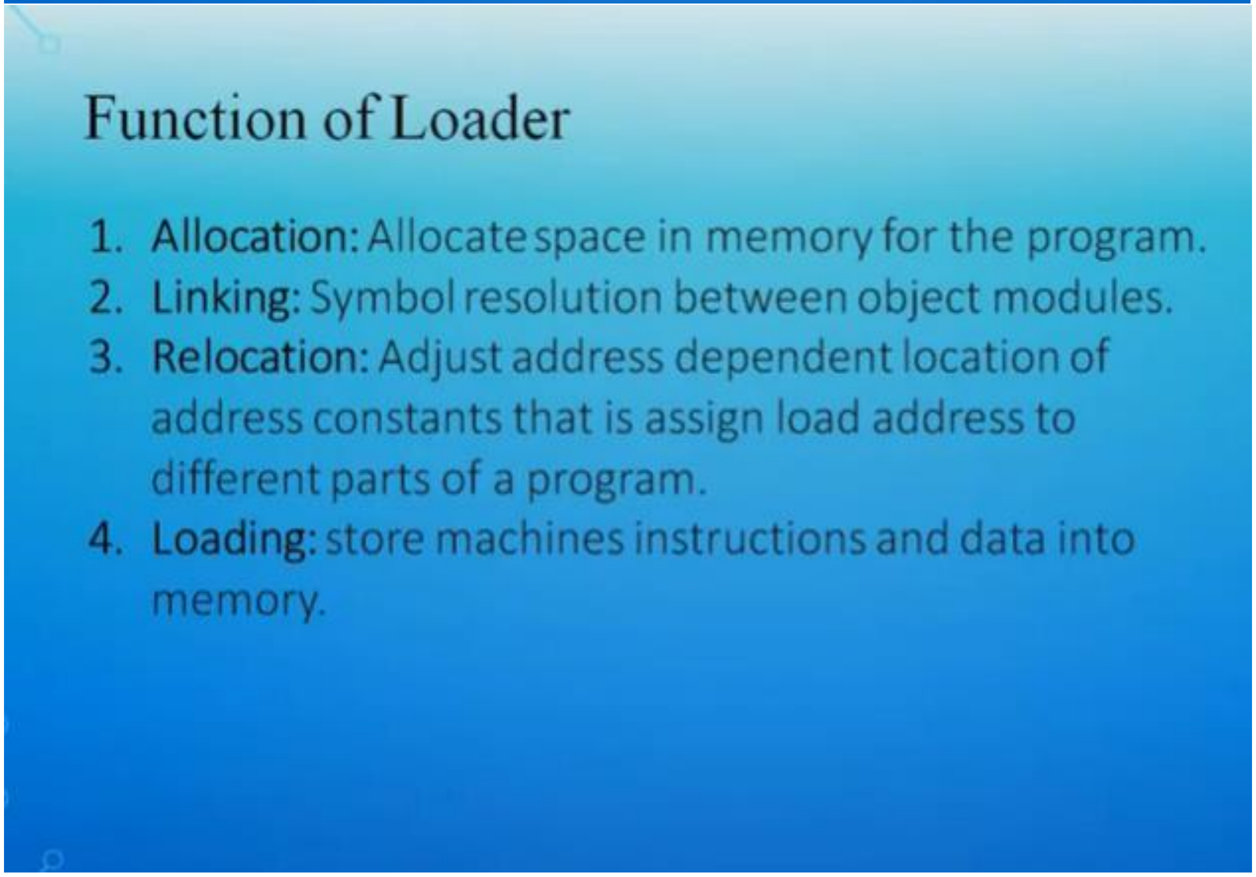
1. Linker is a computer program that links and merges various object files together in order to make an executable file.
2. All these files might have been compiled by separate assemblers.
3. The major task of a linkers is to search and locate referenced module /routines in a program and to determine the memory location where these codes will be loaded, making the program instruction to have absolute references





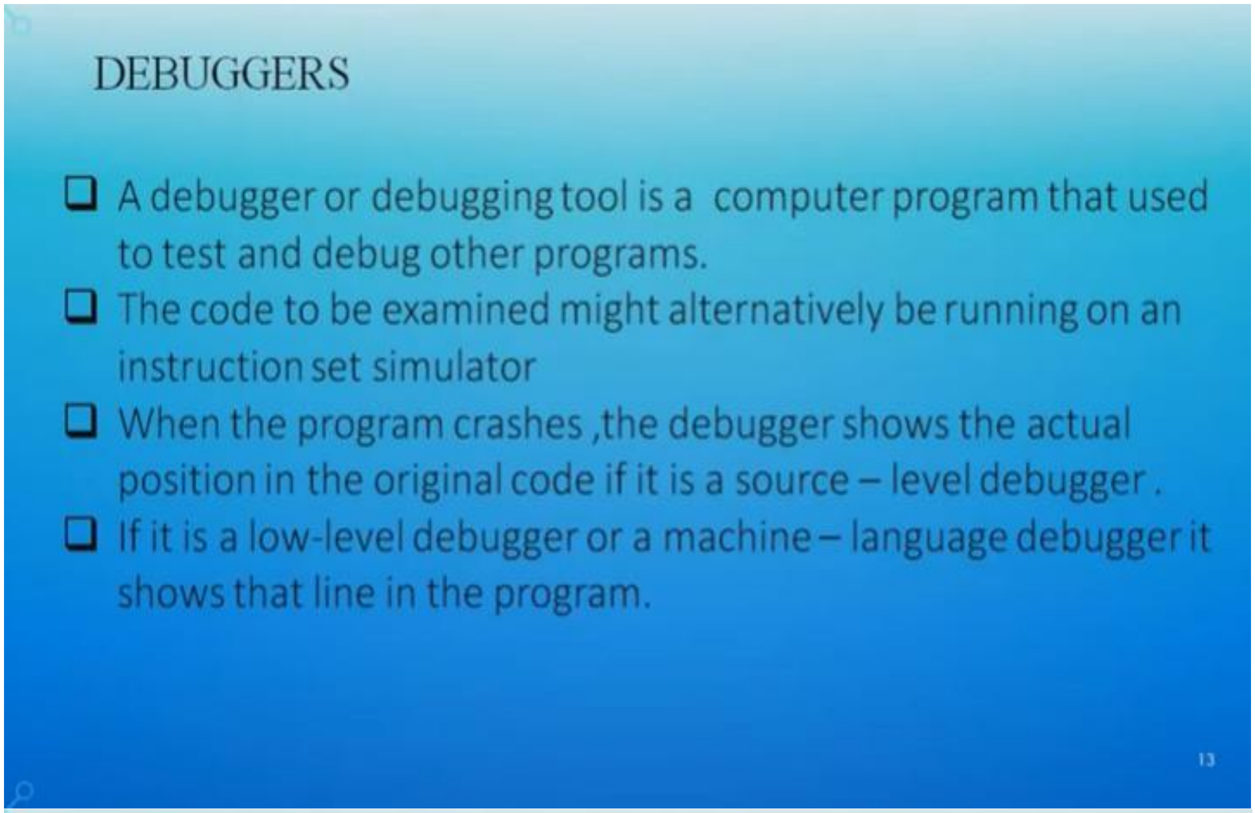
LOADERS

1. Loader is a program which accepts the input as linked modules and loads them into main memory for execution by the compiler.
2. Loader loads or copies program from secondary storage to main memory for execution.



Function of Loader

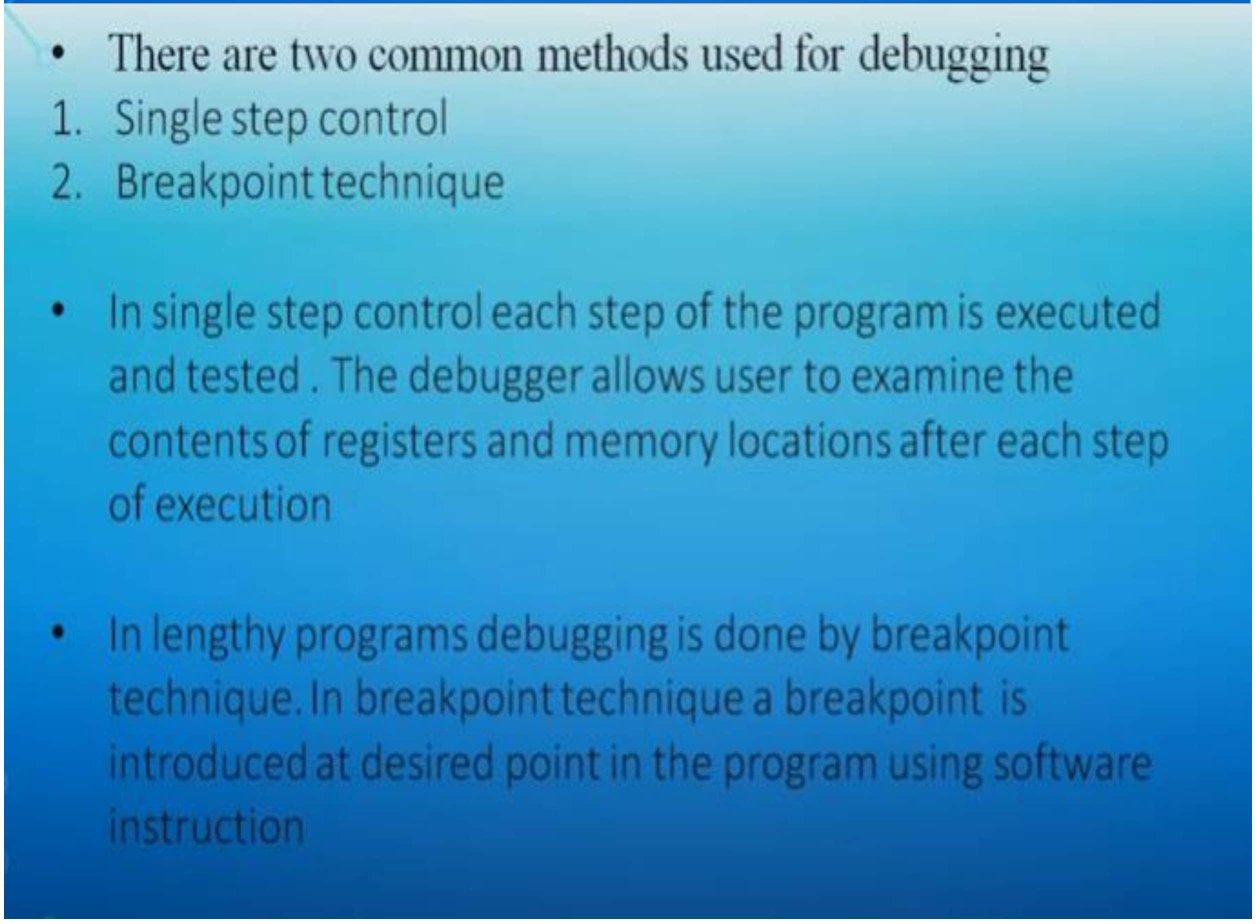
1. **Allocation:** Allocate space in memory for the program.
2. **Linking:** Symbol resolution between object modules.
3. **Relocation:** Adjust address dependent location of address constants that is assign load address to different parts of a program.
4. **Loading:** store machines instructions and data into memory.



DEBUGGERS

- ❑ A debugger or debugging tool is a computer program that used to test and debug other programs.
- ❑ The code to be examined might alternatively be running on an instruction set simulator
- ❑ When the program crashes ,the debugger shows the actual position in the original code if it is a source – level debugger .
- ❑ If it is a low-level debugger or a machine – language debugger it shows that line in the program.

13

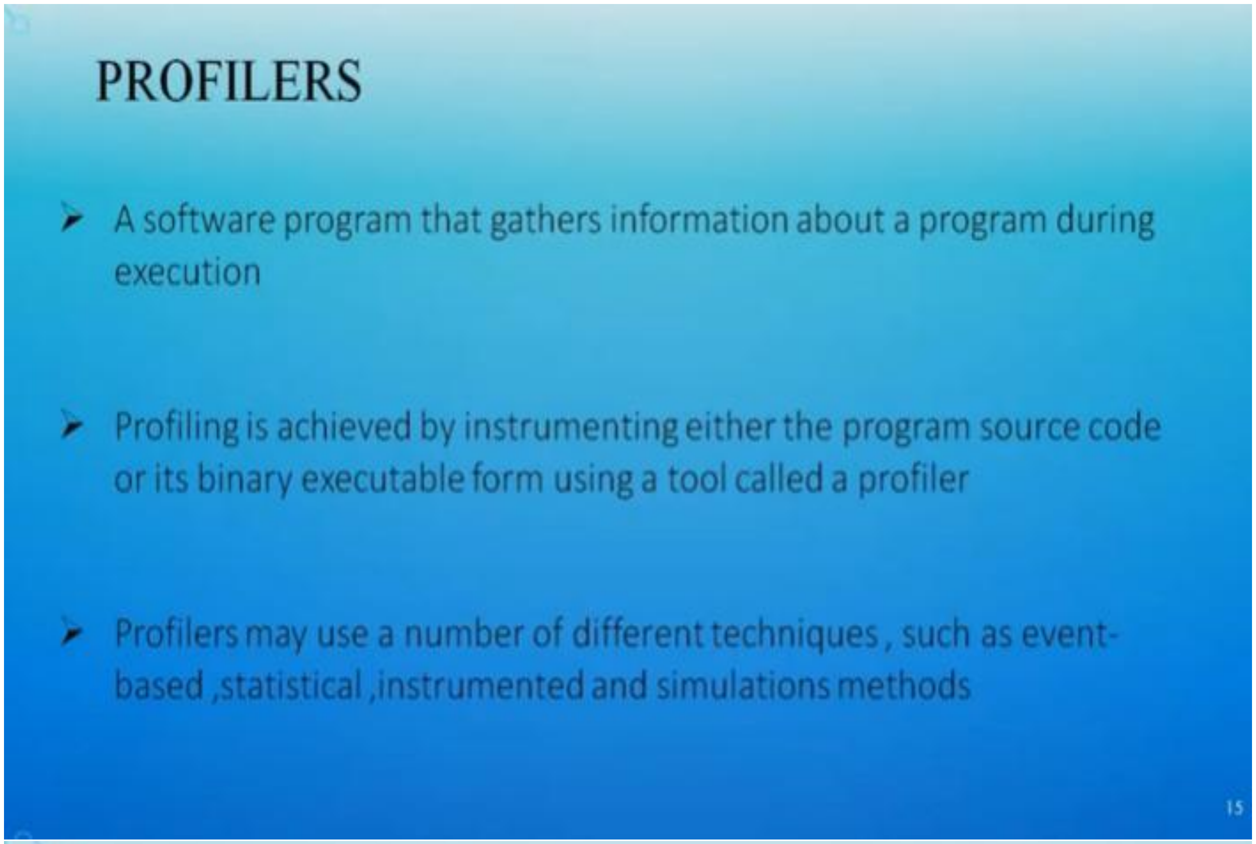


- There are two common methods used for debugging

1. Single step control
2. Breakpoint technique

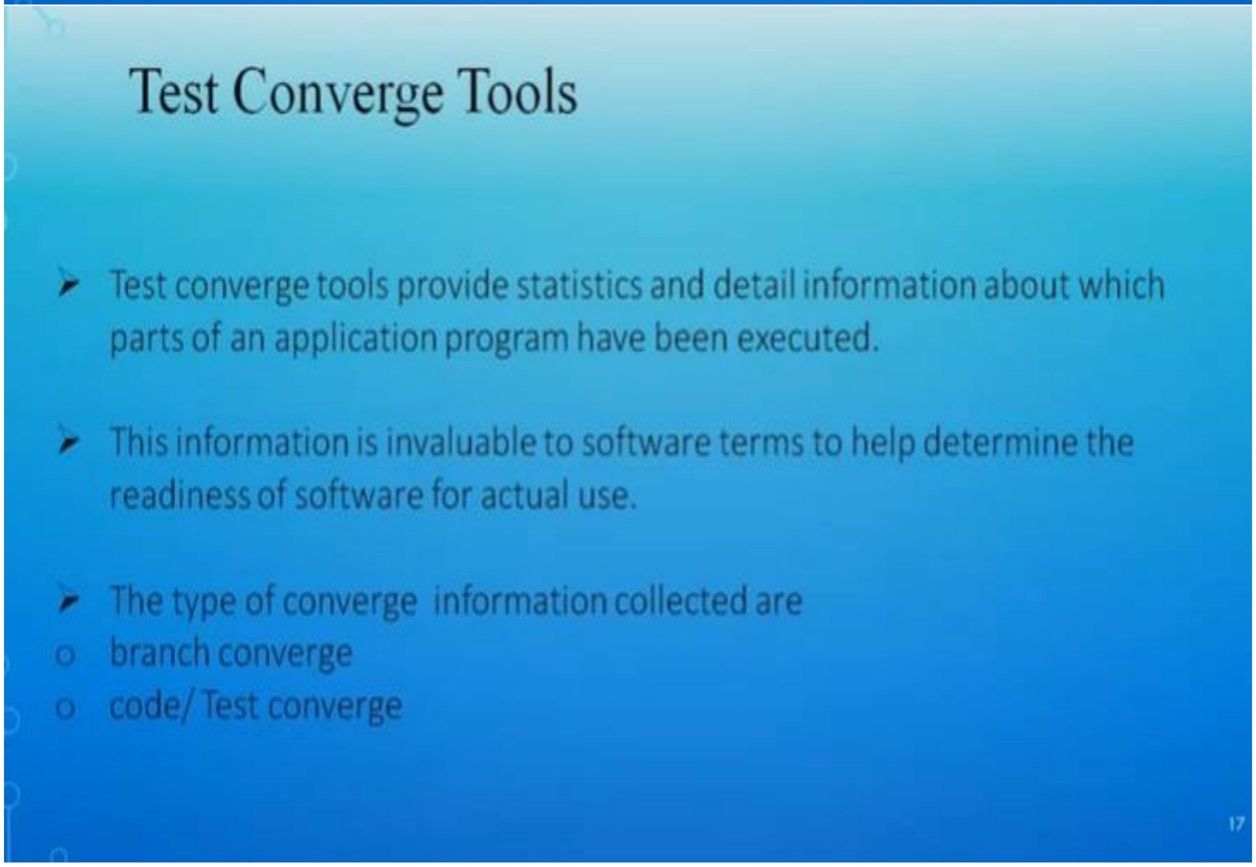
- In single step control each step of the program is executed and tested . The debugger allows user to examine the contents of registers and memory locations after each step of execution

- In lengthy programs debugging is done by breakpoint technique. In breakpoint technique a breakpoint is introduced at desired point in the program using software instruction



PROFILERS

- A software program that gathers information about a program during execution
- Profiling is achieved by instrumenting either the program source code or its binary executable form using a tool called a profiler
- Profilers may use a number of different techniques , such as event-based ,statistical ,instrumented and simulations methods



Test Converge Tools

- Test converge tools provide statistics and detail information about which parts of an application program have been executed.
- This information is invaluable to software terms to help determine the readiness of software for actual use.
- The type of converge information collected are
  - branch converge
  - code/ Test converge

Salient Features of 8051:

1) A Microcontroller is a **complete computer system** built on a **single chip**.

2) It contains all components like Processor (CPU), RAM, ROM, Serial port, Parallel port, Interrupt logic, Timers etc.. on chip.

3) A Microcontroller **saves cost**, saves **power** consumption and makes the circuit **compact**.

4) Microcontrollers are ideally suited for **appliances** like **remote controllers**, **refrigerators**, microwave ovens, modems etc.

5) **8051** is an **8-bit Microcontroller,** it has an 8 bit ALU. This means all arithmetic and logic operations are of 8 bits.

6) **8051** has an **8-bit data bus,** so all external Data Transfers will be of 8-bits in one cycle.

7) It has **internal ROM of 4KB** used for storing **programs**.

8) It has **internal RAM of 128 bytes** used for storing **data**.

9) Since **program memory (ROM) and data memory (RAM) are separate**, 8051 follows **Harvard Model**. In contrast, Processors based on Von Neumann Model store programs and data in a common memory space.

10) There are **4, 8bit, bidirectional I/O ports** for interfacing external devices like **keyboards, displays** etc. These ports can also be used for their **alternate functions** like multiplexed **address data buses** and **control signals**.

11) It has a **serial port for long distance communication**.
The serial port can perform synchronous and asynchronous transfers.

12) 8051 has **two, 16bit Timers**, which act as 'up' counters.
They are used to produce hardware delays and for counting external events.

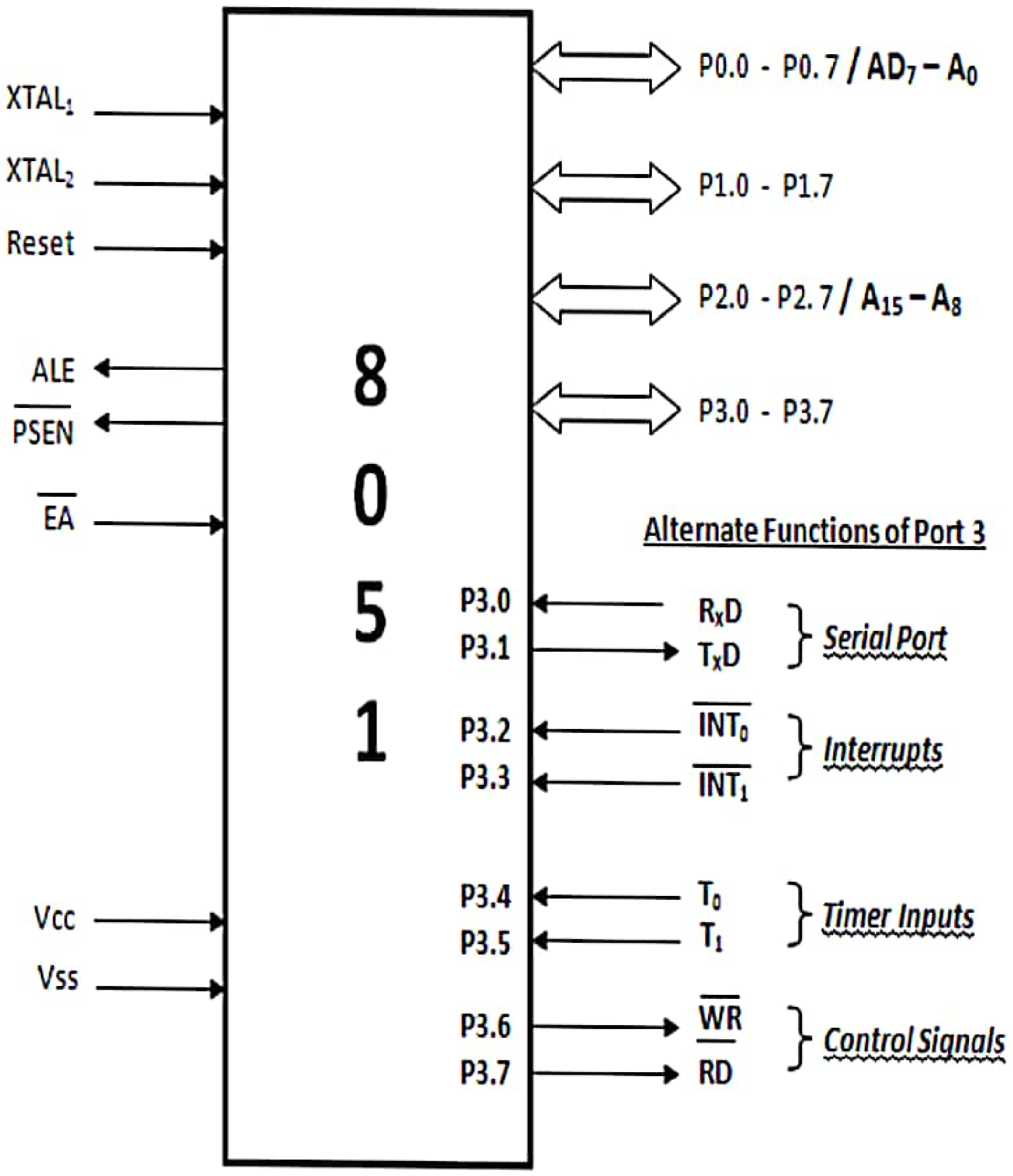
13) There are **5 interrupts**, operating at two priority levels.

14) 8051 has **two power saving modes** called "Idle mode" and "Power Down mode".

15) In addition to internal memory, **up to 64 KB** of **external RAM and External ROM** can be connected, as per user requirement. The figure 64 KB is due to the **16-bit address bus**.

16) 8051 is a **40-pin IC** and typically **operates at 12 MHz frequency.**

PIN DIAGRAM OF 8051



8051 has 40 pins.
The function of these pins is briefly explained as follows.

**XTAL1
&
XTAL2**

These are connected to the **crystal oscillator**.
The **typical operating frequency is 12 MHz**.
In **Serial communication** based applications, the operating frequency is chosen to be **11.0592 MHz**, in order to derive the standard universal baud rates. This will be discussed in detail in the further chapters.

Reset

It is used to **reset** the 8051 microcontroller.
On reset **PC becomes 0000H**.
This address is called the **reset vector address**.
From here, 8051 executes the **BIOS program** also called the Booting program or the monitor program. It is used to **set-up the system** and make it **ready**, to be used by the **end-user**.

ALE
Address
Latch Enable

It is used to **enable the latching of the address**.
The **address and data buses are multiplexed**.
This is done to **reduce the number of pins** on the 8051 IC.
Once out of the chip, address and data **have to be separated** that is called **de-multiplexing**.
This is done by **a latch**, with the **help of ALE signal**.
ALE is **"1"** when the bus carries **address** and **"0"** when the bus carries **data**.
This informs the latch, when the bus is carrying address so that the latch captures only address and not the data.

$\overline{\text{EA}}$
Enable External
Access

It decides whether the first 4 KB of program memory space (0000H... 0FFFH) will be assigned to internal ROM or External ROM.

If $\overline{\text{EA}}$ = 0, the External ROM begins from 0000H.

In this case the Internal ROM is discarded.
8051 now uses only External ROM.

If $\overline{\text{EA}}$ = 1, the External ROM begins from 1000H.

In this case the Internal ROM is used. It occupies the space 0000H... 0FFFH.
In modern **FLASH ROM versions**, this pin also acts as **VPP** (12 Volt programming voltage) to write into the FLASH ROM.

$\overline{\text{PSEN}}$
Program Status Enable

8051 has a **16-bit address bus** (A_{15} – A_0).
This should allow 8051 to access **64 KB of external Memory** as 2^{16} = 64 KB.
Interestingly though, 8051 can access **64 KB of External ROM and 64 KB of External RAM,** making a total of 128 KB.
Both have the same address range **0000H to FFFFH.**
This does not lead to any confusion because there are separate control signals for External RAM and External ROM.

$\overline{\text{RD}}$ and $\overline{\text{WR}}$ are control signals for External RAM.

$\overline{\text{PSEN}}$ is the READ signal for External ROM.

It is called Program Status Enable as it allows reading from ROM also known as Program Memory.
Having separate control signals for External RAM and External ROM actually **allows us to double the size of the external memory** to a total of 128 KB from the original 64 KB.

Vcc
&
GND

These are **power supply** pins.
8051 works at **+5V / 0V** power supply.

P0.0... P0.7

These are **8 pins** of Port 0.
We can perform a **byte operation** (8-bit) on the whole port 0.
We can also **access every bit of port 0 individually** by performing **bit operations like set, clear, complement** etc.
The bits are called **P0.0... P0.7**.
Additionally, Port 0 also has an **alternate function**.
It carries the **multiplexed address data lines**.
A0-A7 (the lower 8 bits of address) and **D0-D7** (8 bits of data) are **multiplexed into AD0-AD7**.
In any operation address and data are not issued simultaneously.
First, address is given, then data is transferred.
Using a common bus for both, **reduces the number of pins**.
To identify if the bus is carrying address or data, we look at the ALE signal.
If **ALE = 1**, the bus carries **address**,
If **ALE = 0**, the bus carries **data**.

P1.0... P1.7

These are **8 pins** of Port 1.
We can perform a **byte operation** (8-bit) on the whole port 1.
We can also **access every bit of port 1 individually** by performing bit operations like set, clear, complement etc. on **P1.0... P1.7**.
Port 1 also has NO alternate function.

P2.0... P2.7

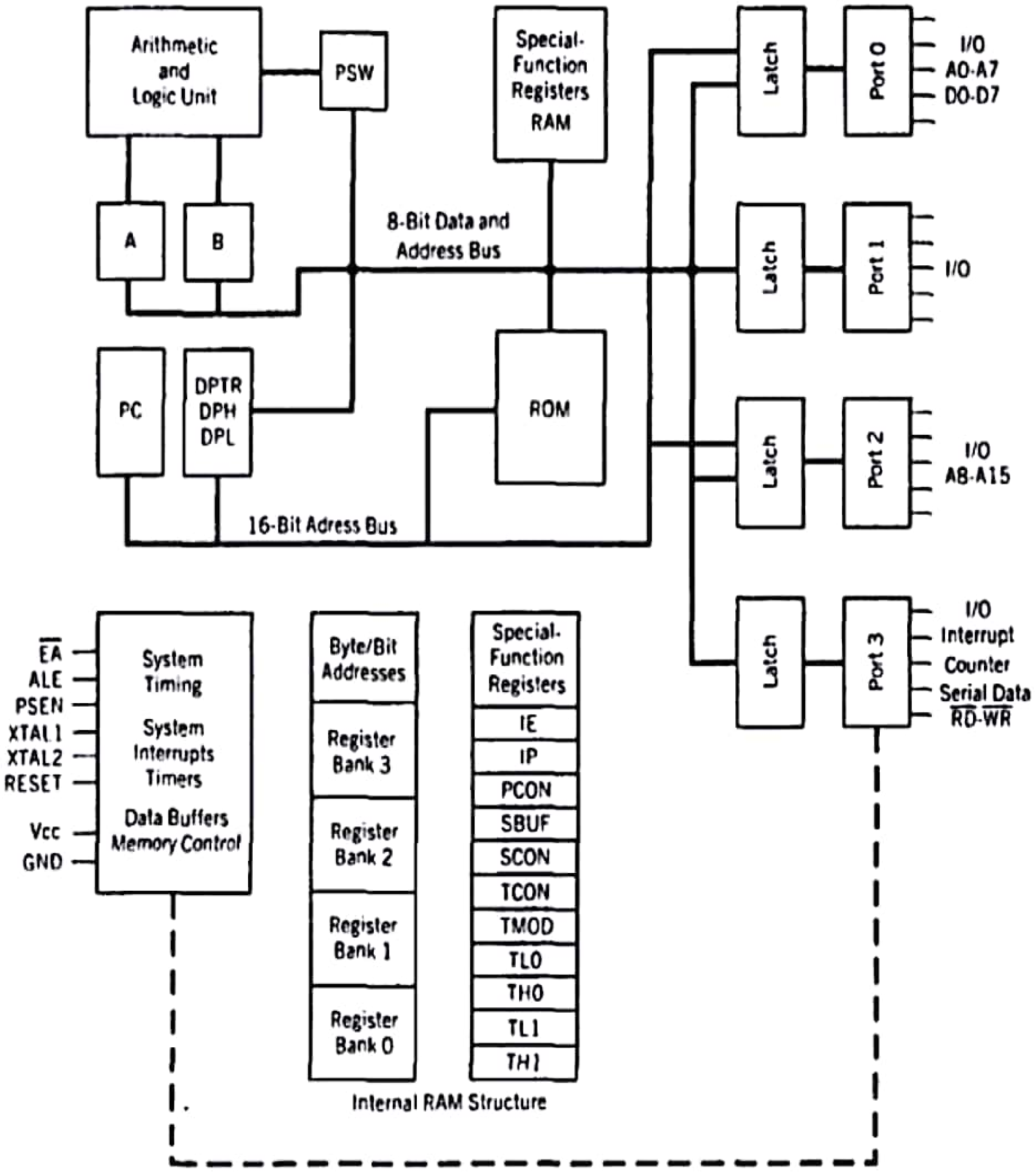
These are **8 pins** of Port 2.
We can perform a **byte operation** (8-bit) on the whole port 2.
We can also **access every bit of port 2 individually** by performing bit operations like set, clear, complement etc. on **P2.0... P2.7**.
Additionally, Port 2 also has an **alternate function**.
It carries the **higher order address lines A8-A15**.

P3.0... P3.7

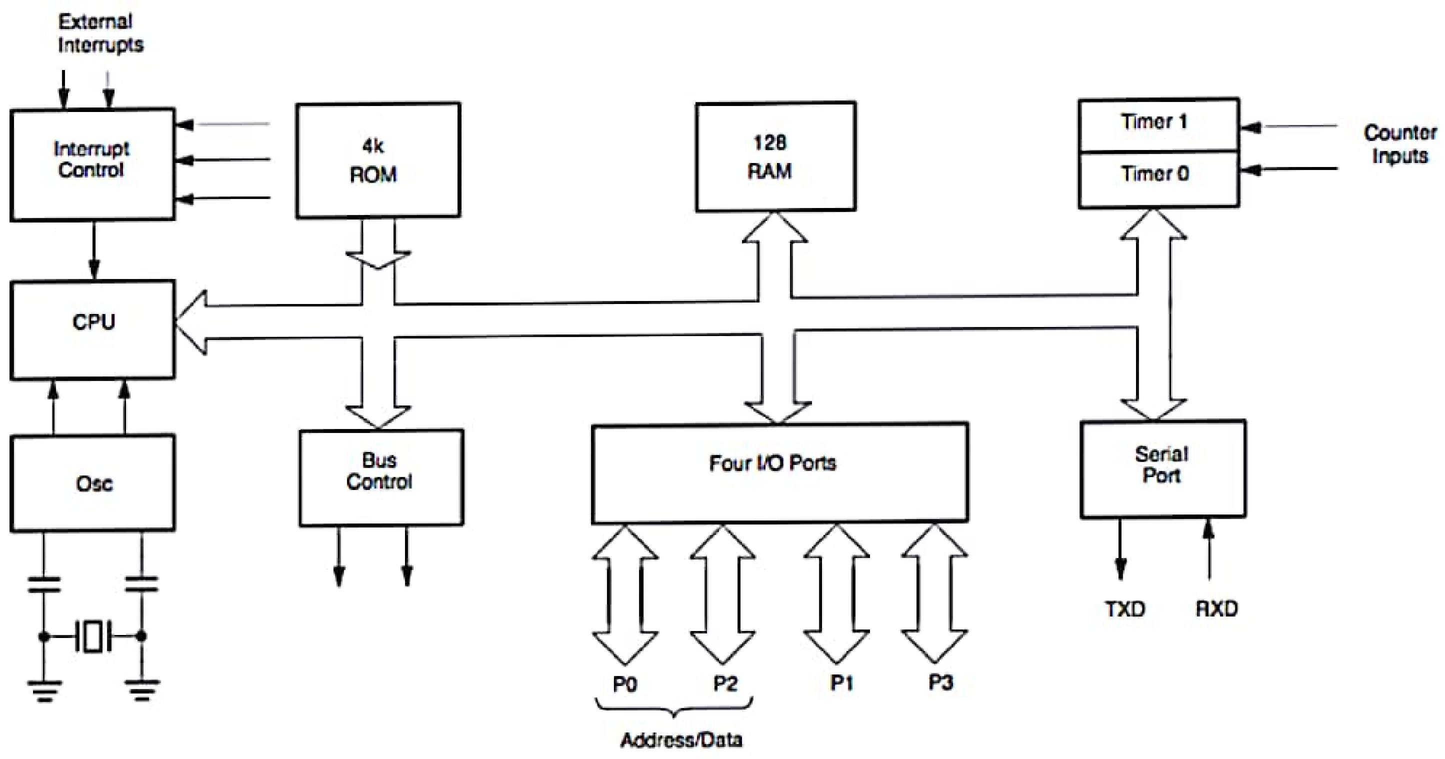
These are **8 pins** of Port 3.
We can perform a **byte operation** (8-bit) on the whole port 3.
We can also **access every bit of port 3 individually.**
The bits are called **P0.0... P0.7**.
The various pins of Port 3 have a lot of alternate functions.

P3.0 (**Rxd**) and P3.1 (**Txd**):	They are used to **receive and transmit serial data**. This forms the **serial port of 8051**.
P3.2 ($\overline{\textbf{INT0}}$) and P3.3 ($\overline{\textbf{INT1}}$):	They are external **hardware interrupts of 8051**. If they occur simultaneously, INTO is by default higher priority.
P3.4 (**T0**) and P3.5 (**T1**):	They are used **timer clock inputs**. They provide external clock inputs to Timer 0 and Timer 1.
P3.6 ($\overline{\textbf{WR}}$) and P3.7 ($\overline{\textbf{RD}}$):	They are used as **control signals for External RAM**. 8051 can access 64 KB External RAM from 0000H to FFFFH..

8051 Block Diagram



ALTERNATE DIAGRAM FOR 8051 ARCHITECTURE / BLOCK DIAGRAM



8051 is a microcontroller. This means it has an internal processor, internal memory and an I/O section. The architecture of 8051 is thus divided into three main sections:

- The CPU
- Internal Memory
- I/O components.

CPU

8051 has an 8 bit CPU.
This is where all 8-bot arithmetic and logic operations are performed.
It has the following components.

ALU – Arithmetic Logic Unit

It performs **8-bit arithmetic and logic operations**.
It can also perform some bit operations.

Example:
```
ADD A, R0        ; Adds contents of A register and R0 register and stores the result in A register.
ANL A, R0        ; Logically ANDs contents of A register and R0 register and stores the result in A register.
CPL P0.0         ; Complements the value of P0.0 pin.
```

A – Register (Accumulator)

It is an **8-bit register**.
In most arithmetic and logic operations, **A register hold the first operand and also gets the result of the operation.**
Moreover, it is the only register to be used for **data transfers** to and from **external memory**.

Example:
```
ADD A, R1        ; Adds contents of A register and R1 register and stores the result in A register.
MOVX A, @DPTR    ; A gets the data from External RAM location pointed by DPTR
```

B – Register

It is an **8-bit register**.
It is dedicated for **Multiplication and Division**.
It can also be used in other operations.

Example:
```
MUL AB           ; Multiplies contents of A and B registers. Stores 16-bit result in B and A registers.
DIV AB           ; Divides contents of A by those of B. Stores quotient in A and remainder in B.
```

PC – Program Counter

It is an **16-bit register.**
It holds **address of the next instruction** in program memory (ROM).
PC gets automatically **incremented** as soon as any **instruction is fetched.**
That's what makes the program move ahead in a **sequential manner.**
In the case of a **branch**, a **new address is loaded into PC.**

DPTR – Data Pointer

It is an **16-bit register.**
It holds **address data** in data memory (RAM).
DPTR is divided into two registers DPH (higher byte) and DPL (lower byte).
It is typically used by the programmer **to transfer data from External RAM.**
It can also be used as a pointer to a **look up table** in ROM, using **Indexed addressing mode.**

Example:
```
MOVX A, @DPTR       ; A gets the data from External RAM location pointed by DPTR
MOVC A, @A+DPTR     ; A gets the data from ROM location pointed by A + DPTR
```

SP – Stack Pointer

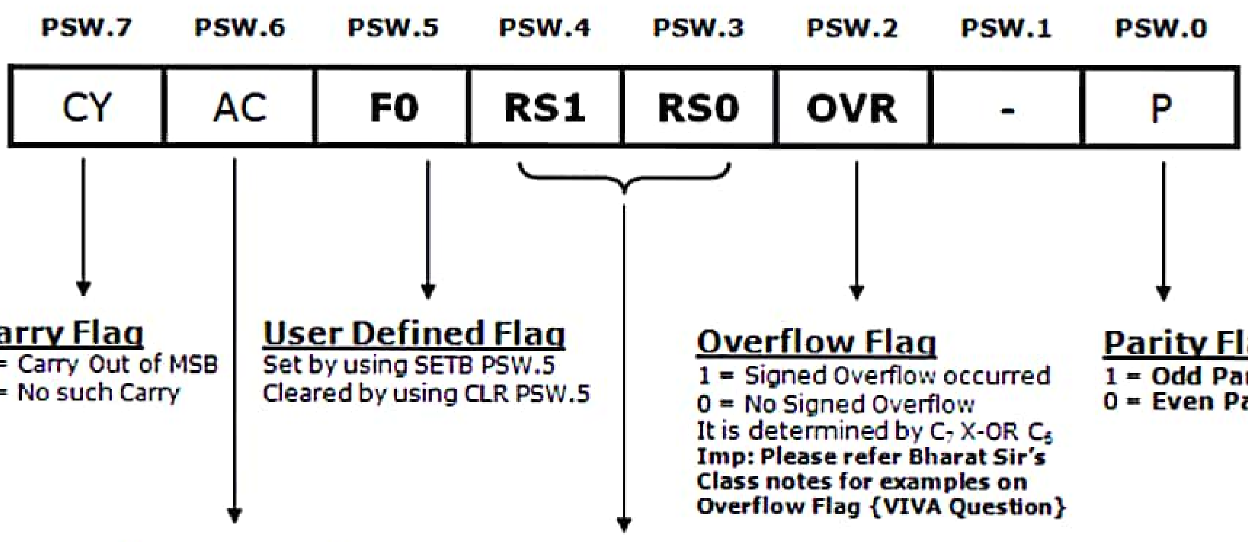
It is an **8-bit register.**
It contains **address of the top of stack.**
The Stack is present in the Internal RAM.
Internal RAM has 8-bit addresses from 00H... 7FH.
Hence SP is an 8-bit register.
It is affected during Push and Pop operations.
During a **Push**, SP gets **incremented.**
During a **Pop**, SP gets **decremented.**

PSW – Program Status Word

It is an **8-bit register.**
It is also called the **"Flag register"**, as it mainly contains the status flags.
These flags indicate **status of the current result.**
They are **changed by the ALU after every arithmetic or logic operation.**
The flags can also be **changed by the programmer.**
PSW is a **bit addressable** register.
Each bit can be individually set or reset by the programmer.
The bits can be referred to by their bit numbers (**PSW.4**) or by their name (**RS1**).

Example:
```
SETB PSW.3          ; Makes PSW.3 ← 1
CLR PSW.4           ; Makes PSW.4 ← 0
```

FLAG REGISTER (PSW) OF 8051



Carry Flag
1 = Carry Out of MSB
0 = No such Carry

User Defined Flag
Set by using SETB PSW.5
Cleared by using CLR PSW.5

Overflow Flag
1 = Signed Overflow occurred
0 = No Signed Overflow
It is determined by C_7 X-OR C_5
Imp: Please refer Bharat Sir's Class notes for examples on Overflow Flag {VIVA Question}

Parity Flag
1 = Odd Parity
0 = Even Parity

Auxiliary Carry Flag
1 = Carry from Lower Nibble to Higher Nibble
0 = No such Carry

Register Bank Select
00 --- Register Bank 0 {default}
01 --- Register Bank 1
10 --- Register Bank 2
11 --- Register Bank 3

RS1 RS0	REGISTER BANK	RAM ADDRESS	SELECTED BY INSTRUCTIONS
0 0	Bank 0	00H ... 07H	CLR PSW.4, CLR PSW.3
0 1	Bank 1	08H ... 0FH	CLR PSW.4, SETB PSW.3
1 0	Bank 2	10H ... 17H	SETB PSW.4, CLR PSW.3
1 1	Bank 3	18H ... 1FH	SETB PSW.4, SETB PSW.3

INTERNAL MEMORY

8051 has two forms of internal memories.
It has 128 bytes of Internal RAM and 4 KB of Internal ROM.

INTERNAL RAM

8051 has **128 bytes of Internal RAM**.
RAM is used to store data, hence is also called **Data Memory**.
The are **128 locations** each containing one byte information.
The **address range** is **00H... 7FH**.
It contains **Register banks**, a **Bit addressable area** and a **General purpose area**.

INTERNAL ROM

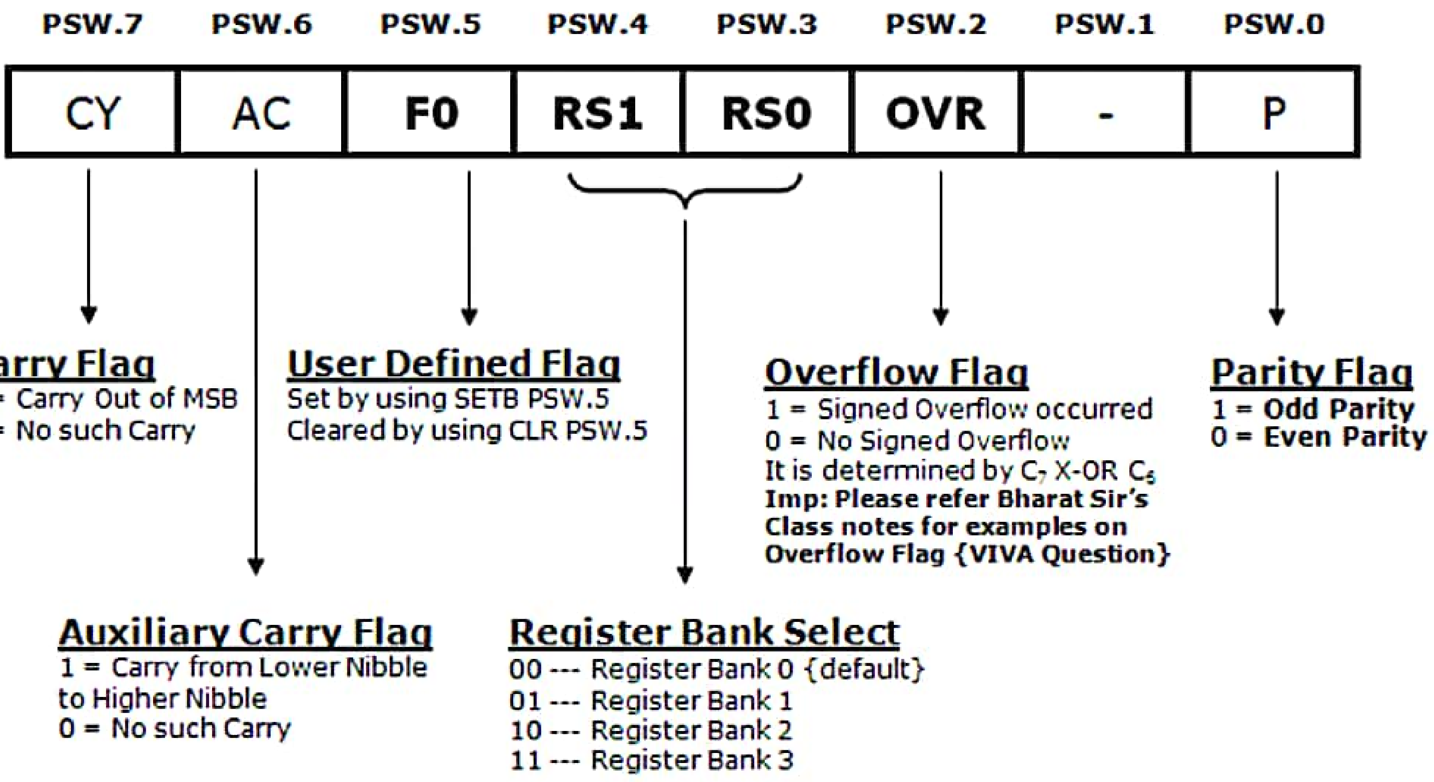
8051 has **4 KB of Internal ROM**.
ROM is used to store programs, hence is also called **Program Memory or Code Memory**.
The are **4 K locations** each containing one byte information.
The **address range** is **0000H... 0FFFH**.
It mainly contains **programs.**
It may also contains some **permanent data** stored in the form of **look up tables**.
To access **programs**, the address is given by **PC** – Program Counter.
To access **data**, the address is given by **DPTR** – Data Pointer.

I/O COMPONENTS

Like any other typical microcontroller, 8051 has several I/O components.
They include I/O ports, Timers, Serial port etc.
8051 has **4, 8-bit I/O ports: P0, P1, P2 and P3.**
They support **bit and byte operations.**
They also have several **alternate functions**.
There are **two 16-bit timers**, which operate as down counters.
There is a **serial port** having pins **Rxd and Txd** to receive and transmit data serially.
There are **two external interrupt pins**.
Additionally there are **address, data and control signals** for transfers with **External RAM and External ROM**.

Finally, 8051 has **21, 8-bit internal SFRs** (Special Function Registers).
These are used to **control operations of the various I/O components** mentioned above.

FLAG REGISTER (PSW) OF 8051



Carry Flag
1 = Carry Out of MSB
0 = No such Carry

User Defined Flag
Set by using SETB PSW.5
Cleared by using CLR PSW.5

Overflow Flag
1 = Signed Overflow occurred
0 = No Signed Overflow
It is determined by C_7 X-OR C_6
Imp: Please refer Bharat Sir's Class notes for examples on Overflow Flag {VIVA Question}

Parity Flag
1 = Odd Parity
0 = Even Parity

Auxiliary Carry Flag
1 = Carry from Lower Nibble to Higher Nibble
0 = No such Carry

Register Bank Select
00 --- Register Bank 0 {default}
01 --- Register Bank 1
10 --- Register Bank 2
11 --- Register Bank 3

PSW – Program Status Word

It is an **8-bit register**.
It is also called the **"Flag register"**, as it mainly contains the status flags.
These flags indicate **status of the current result.**
They are **changed by the ALU after every arithmetic or logic operation.**
The flags can also be **changed by the programmer**.
PSW is a **bit addressable** register.
Each bit can be individually set or reset by the programmer.
The bits can be referred to by their bit numbers (**PSW.4**) or by their name (**RS1**).

CY - Carry Flag

It indicates the carry out of the MSB, after any arithmetic operation.

If CY = 1 : There was a carry out of the MSB
If CY = 0 : There was no carry out of the MSB

AC – Auxiliary Carry Flag

It indicates the carry from lower nibble (4-bits) to higher nibble.
If the 8bits are numbered Bit 7 --- Bit 0, this is the carry from Bit 3 to Bit 4.

If AC = 1 : There was an auxiliary carry
If AC = 0 : There was no auxiliary carry

Note: It is particularly useful in an operation called DA A (Decimal Adjust after Addition).

OVR - Overflow Flag

It indicates if there was an overflow during a signed operation.
An 8-bit signed number has the range -80H... 00H... +7FH.
Any result, out of this range causes an overflow.

If OVR = 1 : There was an overflow in the result
If OVR = 0 : There was no overflow in the result

Overflow is determined by doing an Ex-Or between the 2nd last carry (C_6) and the last carry (C_7)

Note: After an overflow, the Sign (MSB) of the result becomes wrong.

P - Parity Flag

It indicates the Parity of the result.
Parity is determined by the number of 1's in the result.

If PF = 1 : The result has ODD parity
If PF = 0 : The result has EVEN parity

F0 – USER DEFINED FLAG

This flag is **available to the programmer.**
It can be used by us to store any **user defined information.**
For example: In an Air Conditioning unit, programmer can use this flag indicate whether the compressor is ON or OFF (1 or 0).
This flag can be changed by simple instructions like SETB and CLR.

SETB PSW.5; This makes F0 bit ← 1
CLR PSW.5; This makes F0 bit ← 0

RS1, RS0 – REGISTER BANK SELECT

The initial 32 locations (bytes) of the Internal RAM are available to the programmer as registers.
Having so many registers makes programming easier and faster.
Naming R0... R31, would tremendously increase the number of opcodes.
Hence the registers are divided into 4 banks: Bank0... Bank3.
Each bank has 8 registers named R0... R7.
At a time, only of the four banks is the "active bank".
RS1 and RS0 are used by the programmer to select the active bank.

RS1 RS0	REGISTER BANK	SELECTED BY INSTRUCTIONS
0 0	Bank 0	CLR PSW.4 CLR PSW.3
0 1	Bank 1	CLR PSW.4 SETB PSW.3
1 0	Bank 2	SETB PSW.4 CLR PSW.3
1 1	Bank 3	SETB PSW.4 SETB PSW.3

MEMORY ORGANIZATION OF 8051

8051 operates with 4 different memories:

Internal ROM
External ROM

Internal RAM
External RAM

Being based on **Harvard Model**, 8051 stores **programs and data in separate memory spaces**. Programs are stored in ROM, whereas data is stored in RAM.

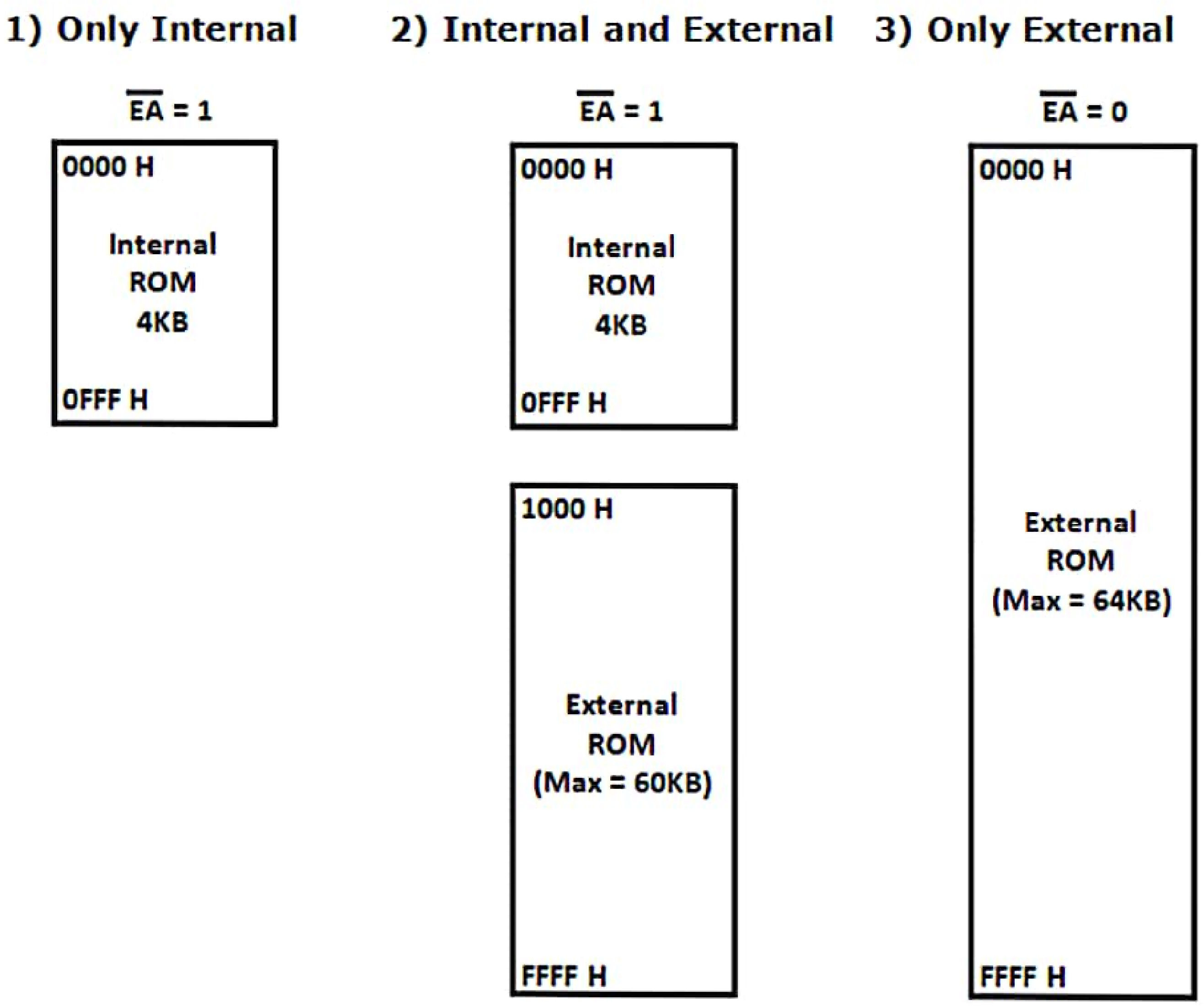
Microcontrollers are used in **appliances**.
Washing machines, remote controllers, microwave ovens are some of the examples.
Here **programs are generally permanent** in nature and very rarely need to be modified.
Moreover, the programs must be **retained** even after the device is completely **switched off**.
Hence programs are stored in permanent (non-volatile) memory like ROM.

Data on the other hand is continuously **changed at runtime.**
For example current temperature, cooking time etc. in an oven.
Such data is not permanent in nature and will certainly be modified in every usage of the device.
Hence Data is stored in writeable memory like RAM.

However, sometimes there is **permanent data**, such as ASCII codes or 7-segment display codes. Such data is stored in **ROM**, in the form of **Look up tables** and is accessed using a dedicated addressing mode called **Indexed** Addressing mode. We will discover this in more depth in further topics.

We are now going to take a closer look at all four memories.

ROM Organization / Code Memory / Program Memory



We can implement ROM in three different ways in 8051.

Only Internal ROM

8051 has 4 KB internal ROM.
In many cases this size is sufficient and there is no need for connecting External ROM.
Such systems use only Internal ROM of 8051.
All addresses from 0000H... 0FFFH will be accessed from Internal ROM.
Any address beyond that will be invalid.
In such systems $\overline{\textbf{EA}}$ will be "1" as Internal ROM is being used.

(PS: Read the whole answer to understand $\overline{\text{EA}}$ clearly)

Internal And External ROM

8051 has 4 KB internal ROM.
In many cases this size is may be insufficient and we may need to add some External ROM.
Such systems use a combination of Internal ROM and External ROM.
The **"total"** ROM that can be accessed is 64 KB.
Since we are using the Internal ROM of 4 KB, the maximum amount of External ROM that can be connected is 60 KB.
All addresses from 0000H... 0FFFH will be accessed from Internal ROM.
Addresses 1000H... FFFH will be accessed from External ROM.
In such systems $\overline{\textbf{EA}}$ will be "1" as Internal ROM is being used.

(PS: Read the whole answer to understand $\overline{\text{EA}}$ clearly)

Only External ROM

This is the most interesting case.
Though 8051 has 4 KB of Internal ROM, the user may choose the discard it and connect only External ROM.
This may happen due to several reasons.
The program stored in the Internal ROM may have become invalid or outdated, or the system may need to be upgraded etc.
Such systems use only External ROM, and the Internal ROM is discarded.
Here we can connect up to 64 KB of External ROM.
All addresses from 0000H... FFFFH will be accessed from External ROM.
But do keep in mind, that the Internal ROM is still present in 8051.
We need to clearly indicate to 8051 that the Internal ROM must be ignored and every address from 0000H... FFFFH must be accessed externally. This is indicated by us to 8051 using $\overline{\textbf{EA}}$ **.**

By making $\overline{\textbf{EA}}$ **= 0**, we inform 8051 that the Internal ROM must be discarded and all ROM must be accessed externally.

Note: Use of $\overline{\textbf{EA}}$ pin of 8051.

The $\overline{\textbf{EA}}$ pin of 8051 decides whether the Internal ROM will be used or not.

If the Internal ROM has to be used we must make $\overline{\textbf{EA}} = 1$.

Now 8051 will Access the internal ROM for all addresses from 0000H to 0FFFH and will only access external ROM for addresses 1000H and beyond.

But if $\overline{\textbf{EA}} = 0$, then the Internal ROM is completely discarded.

Now 8051 will access the External ROM for all addresses from 0000H to FFFFH, hence discarding the internal ROM.

8051 checks $\overline{\textbf{EA}}$ pin during every ROM operation where the address is 0000H... 0FFFH.

If $\overline{\textbf{EA}}$ **= 1,** this location is accessed from internal ROM.

If $\overline{\textbf{EA}}$ **= 0,** this location is accessed from external ROM.

If the address is 1000H or more, 8051 does not check $\overline{\textbf{EA}}$ as this location can only be present in External ROM.

SPECIAL FUNCTION REGISTERS (SFRs) OF 8051

8051 has 21, 8-bit Special Function registers.

	NAME	FUNCTION	BYTE ADDRESS	BIT ADDRESS
Used for holding data and status during Programming	A*	Accumulator	0E0H	0E7H...0E0H
	B*	Arithmetic	0F0H	0F7H...0F0H
	PSW*	Program Status Word	0D0H	0D7H...0D0H
Used in instructions to point to memory	SP	Stack Pointer	81H	NA
	DPL	Address External Memory	82H	NA
	DPH	Address External Memory	83H	NA
Used by the respective I/O Ports	P0*	I/O Port latch	80H	87H...80H
	P1*	I/O Port latch	90H	97H...90H
	P2*	I/O Port latch	0A0H	0A7H...0A0H
	P3*	I/O Port latch	0B0H	0B7H...0B0H
Used by the Serial Port	SCON*	Serial Port Control	98H	9FH...98H
	SBUF	Serial Port Data Buffer	99H	NA
Used for Timer Control	TCON*	Timer/Counter Control	88H	8FH...88H
	TMOD	Timer/Counter Mode Control	89H	NA
	TL0	Timer 0 Low Byte	8AH	NA
	TL1	Timer 1 Low Byte	8BH	NA
	TH0	Timer 0 High Byte	8CH	NA
	TH1	Timer 1 High Byte	8DH	NA
Used for Interrupt Control	IE*	Interrupt Enable	0A8H	0AFH...0A8H
	IP*	Interrupt Priority	0B8H	0BFH...0B8H
Used for Power Control	PCON	Power Control	87H	NA

*Means the SFR is Bit Addressable

1) SFRs are **8-bit** registers.
Each SFR has its own **special function**.

2) **They are** placed **inside** the **Microcontroller.** #Please refer Bharat Sir's Lecture Notes for this ...

3) They are used by the programmer to perform special functions like controlling the timers, the serial port, the I/O ports etc.

4) As SFRs are available to the programmer, we will use them in instructions.
This causes another problem.
SFRs are registers after all, and hence using them would tremendously increase the number of opcodes. ***(Refer to Bharat Sir's Lecture notes for more on this)***

5) To reduce the number of opcodes, **SFRs are allotted addresses.**
These addresses must not clash with any other addresses of the existing memories.

6) Incidentally, the internal RAM is of 128 bytes and uses addresses only from 00H... 7FH.
This gives an entire range of addresses from 80H... FFH completely unused and can be freely allotted to the SFRs.

7) **Hence SFRs are allotted addresses from 80H... FFH.**
It is not a co-incidence that these addresses are free. It is how 8051 design was planned. The Internal RAM was restricted to 128 bytes instead of 256 bytes so that these addresses are free for SFRs.

8) Moreover, some SFRs are bit addressable, like Port 0.
All 8-bits can be individually accessed from P0.0... P0.7, by instructions like SETB, CLR etc.
But again, this will again tremendously increase the number of opcodes.

9) To avoid this problem, **even the bits of the SFRs are allotted addresses.**
These are bit addresses, which are different from byte addresses.
These bit addresses must not clash with those of the bit addressable area of the Internal RAM.
Amazingly, even the bit addresses in the Internal RAM are 00H... 7FH (again 128 bits), keeping bit addresses 80H... FFH free to be used by the SFR bits.

10) **So bit addresses 80H... FFH are allotted to the bits of various SFRs.**
(Watch Bharat Acharya Education, videos on YouTube for more on this)

11) Port 0 has a byte address of 80H and its bit addresses are from 80H... 87H.
A byte operation at address 80H will affect entire Port0.
E.g.:: MOV A, P0; this refers to Byte address 80H that's whole Port 0.

12) **A bit operation at 80H will affect only P0.0.**
E.g.:: SETB P0.0; this refers to bit address 80H that's Port0.0

ADDRESSING MODES OF 8051

Addressing Modes is the manner in which operands are given in the instruction.
8051 supports the following 5 addressing modes:

1) IMMEDIATE ADDRESSING MODE

In this addressing mode, the **Data** is given **in** the **Instruction** itself.
We put a "#" symbol, before the data, to identify it as a data value and not as an address.

Eg:

```
MOV A, #35H         ; A ← 35H

MOV DPTR, #3000H    ; DPTR ← 3000H
```

2) REGISTER ADDRESSING MODE

In this addressing mode, **Data** is given by **a Register** in the instruction.
The permitted registers are **A, R7 ... R0** of each memory bank.
Note: Data transfer between two RAM registers is not allowed.

Eg:

```
MOV A, R0     ; A ← R0 ...  If R0 = 25H, then A gets the Value 25H.

MOV R5, A     ; R5 ← A

MOV Rx, Ry    ; NOT ALLOWED. That's because this would allow 64 combinations of registers
              ; As registers invite opcodes, this would need 64 opcodes!
```

3) DIRECT ADDRESSING MODE

Here, the **address** of the operand is given **in** the **instruction**.
Only Internal RAM addresses (**00H...7FH**) and **SFR** addresses (from **80H to FFH**) allowed.

Eg:

```
MOV A, 35H      ; A ← Contents of RAM location 35H

MOV A, 80H      ; A ← Contents of Port 0 (SFR at address 80H)

MOV 20H, 30H    ; [20H] ← [30H]
                ; i.e. Location 20H gets the contents of Location 30H
```

4) Indirect Addressing Mode

Here, the **address** of the operand is given **in** a **register.**
Internal RAM and External RAM can be accessed using this mode.
The advantage of giving an address using a register is that we can increment the address in a loop, by simply incrementing the register, and hence access a series of locations.

Internal RAM: (8-bit address given by R0 or R1)

ONLY R1 or R0, called as *Data Pointers*, can be used to specify **address** (00H ... 7FH).
An "@" sign is present before the register to indicate that the register is giving an address.

```
Eg:   MOV A, @R0        ; A ← [R0]
                        ; i.e. A ← Contents of Internal RAM Location whose address is given by R0.
                        ; if R0 = 25H, then A gets the contents of Location 25H from Internal RAM

      MOV @R1, A        ; [R1] ← A i.e. Internal RAM Location pointed by R1 gets value of A.
```

External RAM: (16 Bit address given by DPTR)

For the **External RAM, address** is provided by **R1 or R0,** or by **DPTR**.
If DPTR is used to give an address, then the full 64KB range of External RAM from 0000H... FFFFH is available. This is because DPTR is 16-bit and $2^{16} = 65536$.
An "**X**" is present in the instruction, to indicate External RAM.

```
Eg:   MOVX A, @DPTR     ; A ← [DPTR]^
                        ; A gets the contents of External RAM Location whose address is given by DPTR
                        ; If DPTR = 2000H, then A gets contents of Location 2000H from External RAM

      MOVX @DPTR, A     ; [DPTR]^ ← A
                        ; i.e. A is stored at the External RAM Location whose address is given by DPTR
```

External RAM: (8 Bit address given by R0 or R1)

If R0 or R1 is used to give an address, then only the first 256 locations of External RAM is available from 0000 H to 00FF H. This is because R0 or R1 are 8-bit and 2^8 = only 256.

```
Eg:   MOVX A, @R0       ; A ← [R0]^
                        ; i.e. A gets the contents of External RAM Location whose address is given by R0
                        ; If R0 = 25H, then A gets contents of Location 0025H from the External RAM

      MOVX @R1, A       ; [R1]^ ← A
                        ; i.e. A is stored at the External RAM Location whose address is given by R1
```

5) Indexed Addressing Mode

This mode is used to access data from the Code memory (**Internal ROM or External ROM**).
In this addressing mode, address is indirectly specified as a "SUM" of (A and DPTR) or (A and PC).
This is very useful because ROM contains permanent data which is stored in the form of Look Up tables. To access a Look Up table, address is given as a SUM or two registers, where one acts as the base and the other acts as the index within the table.
A "**C**" is present in such instructions, to indicate Code Memory.

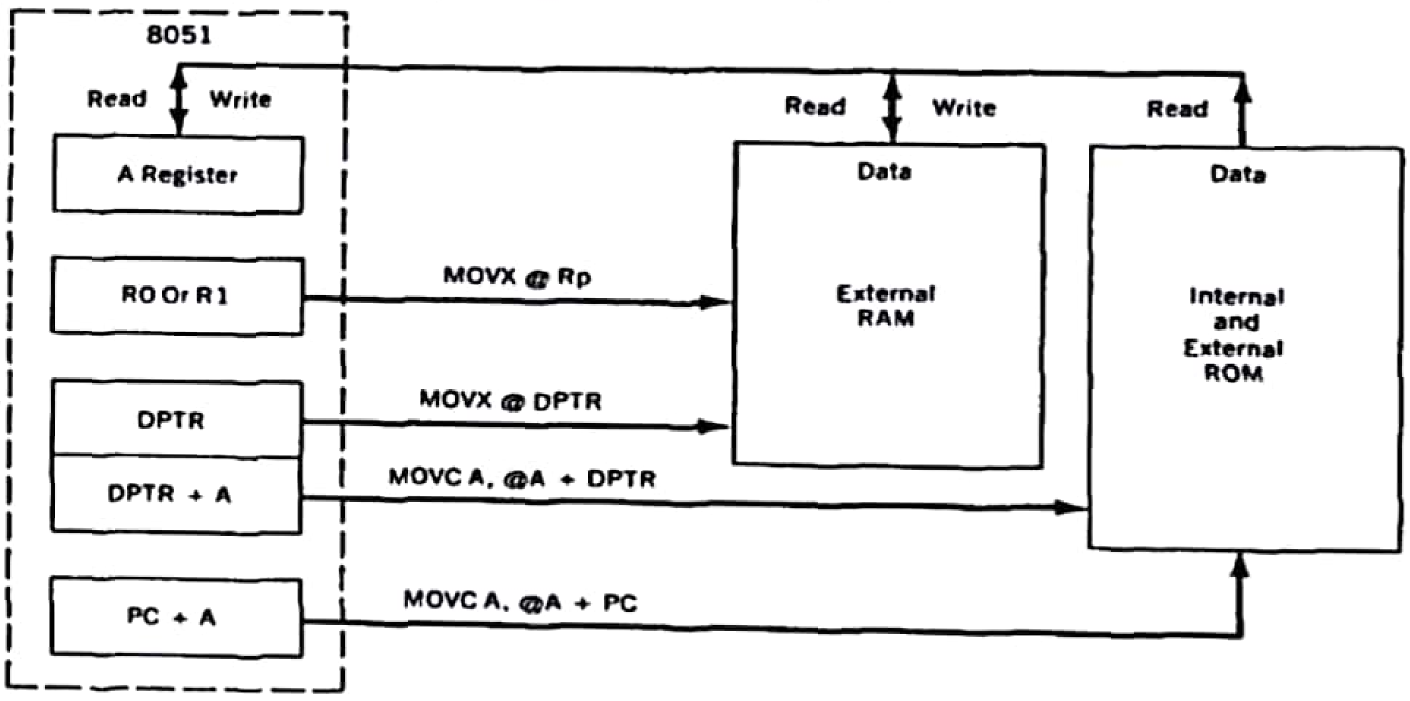
Eg: MOV**C** A, @A+DPTR ; *A ← Contents of a ROM Location pointed by A+DPTR.*
; If DPTR = 0400H and A = 05H,
; then A gets the contents of ROM Location whose address is 0405 H.

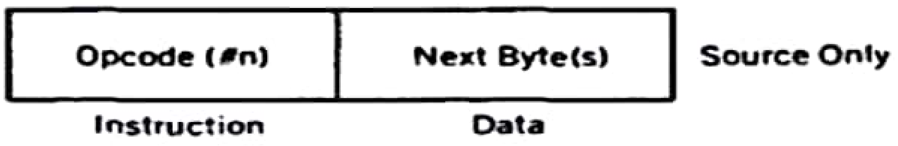
MOV**C** A, @A+PC ; *A ← Contents of a ROM Location pointed by A+PC.*

The same instruction may operate on Internal or External ROM, depending upon the address and on the value of $\overline{\textbf{EA}}$ pin of 8051.

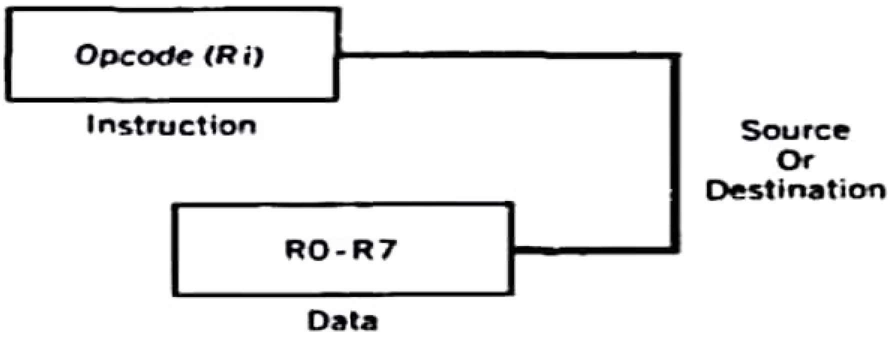
If the address is in the range of 0000... 0FFFH, then $\overline{\textbf{EA}}$ pin will decide if it operates on Internal ROM or External ROM. IF $\overline{\textbf{EA}}$ = 0, External ROM else Internal ROM.
If Address is 1000H and more, it will certainly be External ROM.

External Addressing using MOVX and MOVC

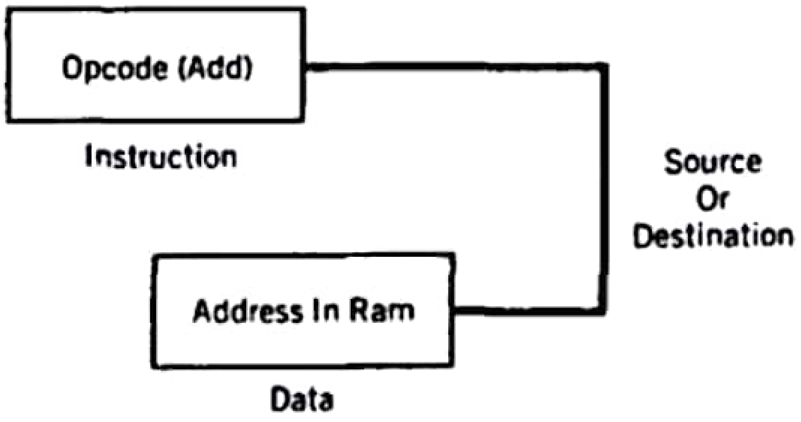




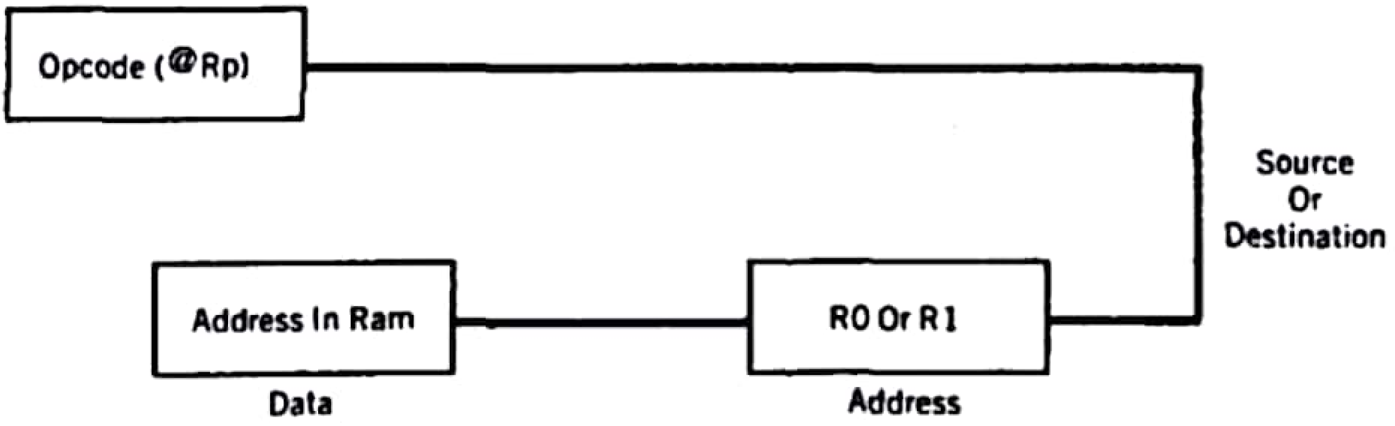
Immediate Addressing Mode



Register Addressing Mode



Direct Addressing Mode



Indirect Addressing Mode

ARITHMETIC INSTRUCTIONS

Here you will see operations like:

- Addition
- Addition with Carry
- Subtraction with Borrow
- Increment
- Decrement
- Multiply
- Divide
- Decimal Adjustment after addition.

1) ADD A, #n | "Add"

Example:

ADD A, #25H; A ← A + 25H

Operation:

Adds A Register with Immediate data.
Stores the result in A Register.

No of cycles required: **1**

IMPORTANT TIP FROM BHARAT ACHARYA
Please do remember to check for carry after performing addition.
Carry flag is checked by instructions like JC and JNC.

---

2) ADD A, Rr | "Add"

Example:

ADD A, R0; A ← A + R0

Operation:

Adds A Register with the value of a RAM register.
Stores the result in A Register.

No of cycles required: **1**

---

3) ADD A, addr | "Add"

Example:

ADD A, 25H; A ← A + [25H]

Operation:

Adds A Register with the contents of the address.
Stores the result in A Register.

No of cycles required: **1**

---

4) ADD A, @Rp | "Add"

Example:

ADD A, @R0; A ← A + [R0]

Operation:

Adds A Register with the contents of the location pointed by the register.
Result is stored in A Register.

If R0 = 20H and Location 20H contains value 35H, then 35H will be added to A register.

No of cycles required: **1**

5) ADDC A, #n | "Add with carry"

Example:

ADDC A, #25H; A ← A + 25H + Carry Flag

Operation:

Adds A Register with Immediate data along with the Carry of the previous addition which is present in the Carry Flag. Stores the result in A Register.

No of cycles required: **1**

IMPORTANT TIP FROM BHARAT ACHARYA

ADDC is used when we want to ADD two large numbers like 16 bit numbers.
First we add the lower bytes using ADD instruction.
Then we add the higher bytes using ADDC instruction.
If the Lower byte has produced a Carry, then CF will be 1.
This Carry will be added into the higher bytes.
Please refer to our classroom example of adding 12FFH + 0001H = 1300H.

---

6) ADDC A, Rr | "Add with carry"

Example:

ADDC A, R0; A ← A + R0 + Carry Flag

Operation:

Adds A Register with the value of a RAM register along with the Carry of the previous addition. Stores the result in A Register.

No of cycles required: **1**

---

7) ADDC A, addr | "Add with carry"

Example:

ADDC A, 25H; A ← A + [25H] + Carry Flag

Operation:

Adds A Register with the contents of the address along with the Carry of the previous addition. Stores the result in A Register.

No of cycles required: **1**

---

8) ADDC A, @Rp | "Add with carry"

Example:

ADDC A, @R0; A ← A + [R0] + Carry Flag

Operation:

Adds A Register with the contents of the location pointed by the register along with the Carry of the previous addition. Result is stored in A Register.

No of cycles required: **1**

9) SUBB A, #n | "Subtract with borrow"

Example:

SUBB A, #25H; A ← A - 25H - Carry Flag

Operation:

Performs A Register – Immediate data – Carry Flag (Carry Flag holds the borrow of the previous subtraction). Stores the result in A Register.

No of cycles required: **1**

IMPORTANT TIP FROM BHARAT ACHARYA

SUBB is used when we want to Subtract two large numbers like 16 bit numbers.
First we Subtract the lower bytes.
If the Lower byte subtraction needs a borrow, then CF will be 1.
This Carry will be subtracted from the higher bytes.
It is important to realize that there is no ordinary SUB instruction.
Hence if we want to perform simple 8-bit subtraction, we still have to use SUBB instruction.
If we don't want Carry flag to interfere with the operation,
We must first clear the carry flag using CLR C instruction before using SUBB.

---

10) SUBB A, Rr | "Subtract with borrow"

Example:

SUBB A, R0; A ← A - R0 - Carry Flag

Operation:

Performs A Register – value of RAM register – Carry Flag (Carry Flag holds the borrow of the previous subtraction). Stores the result in A Register.

No of cycles required: **1**

---

11) SUBB A, addr | "Subtract with borrow"

Example:

SUBB A, 25H; A ← A - [25H] - Carry Flag

Operation:

Performs A Register – contents of memory address – Carry Flag (Carry Flag holds the borrow of the previous subtraction). Stores the result in A Register.

No of cycles required: **1**

---

12) SUBB A, @Rp | "Subtract with borrow"

Example:

SUBB A, @R0; A ← A - [R0] - Carry Flag

Operation:

Performs A Register – Contents of the memory location pointed by the register – Carry Flag. Result is stored in A Register.

No of cycles required: **1**

13) INC A

| "Increment"

Example:

INC A; A ← A + 1

Operation:

Increments the value of A register. Stores the result in A Register.

No of cycles required: **1**

Important tip from Bharat Acharya

During INC A, if A was FFH, it will roll over to 00H.

---

14) INC Rr

| "Increment"

Example:

INC R0; R0 ← R0 + 1

Operation:

Increments the value of a RAM register. Stores the result in the same RAM Register.

No of cycles required: **1**

---

15) INC addr

| "Increment"

Example:

INC 25H; [25H] ← [25H] + 1

Operation:

Increments the contents of a memory address. Stores the result back in the same location.

No of cycles required: **1**

---

16) INC @Rp

| "Increment"

Example:

INC @R0; [@R0] ← [@R0] + 1

Operation:

Increments the contents of a memory location pointed by R0. Will store the result back at the same location.

No of cycles required: **1**

---

17) INC DPTR

| "Increment"

Example:

INC DPTR; DPTR ← DPTR + 1

Operation:

Increments the 16-bit value of DPTR register. Stores the result in DPTR Register.

No of cycles required: **2**

18) DEC A

| "Decrement"

Example:

DEC A; A ← A – 1

Operation:

Decrements the value of A register.
Stores the result in A Register.

No of cycles required: **1**

Important tip from Bharat Acharya
During DEC A, if A was 00H, it will roll back to FFH.
That's because 00H – 1 = – (01H) which is FFH in 2's complement form.
Please refer to classroom explanation for more clarity on this.
Also, do remember, **DEC DPTR does not exists.**
If you need to decrement DPTR, you can do it by individually decrementing DPL and DPH.
First decrement DPL. If it rolls back from 00H to FFH, then also decrement DPH.
E.g.:: 1300H – 1 = 12FFH.

---

19) DEC Rr

| "Decrement"

Example:

DEC R0; R0 ← R0 – 1

Operation:

Decrements the value of a RAM register.
Stores the result in the same RAM Register.

No of cycles required: **1**

---

20) DEC addr

| "Decrement"

Example:

DEC 25H; [25H] ← [25H] – 1

Operation:

Decrements the contents of a memory address.
Stores the result back in the same location.

No of cycles required: **1**

---

21) DEC @Rp

| "Decrement"

Example:

DEC @R0; [@R0] ← [@R0] – 1

Operation:

Decrements the contents of a memory location pointed by R0.
Will store the result back at the same location.

No of cycles required: **1**

22) MUL AB | "Multiply A and B"

Example:

MUL AB; B $_{(Higher)}$. A $_{(Lower)}$ ← A x B

Operation:

Multiples the 8-bit values of A register and B register.
Stores the 16 bit result in B and A Registers.
B register gets the Higher Byte, A register gets the Lower Byte.

No of cycles required: **4**

---

23) DIV AB | "Divide A by B"

Example:

DIV AB; B $_{(Remainder)}$. A $_{(Quotient)}$ ← A ÷ B

Operation:

Divides the 8-bit value of A register by the 8-bit value of B register.
Stores the result in B and A Registers.
B register gets the Remainder, A register gets the Quotient.

No of cycles required: **4**

IMPORTANT TIP FROM BHARAT ACHARYA
During DIV AB, if B register is 00H, then the instruction will be aborted.
A and B registers will contain garbage values after the operation.
This is indicated to the programmer by the **OVERFLOW FLAG**.
If Overflow Flag becomes "1" after the operation, it simply means division by 0 was attempted.

---

24) DA A

| "Decimal Adjust after Addition"

Example:

DA A;

Operation:

It is used when we want to add two decimal numbers (BCD numbers).
We first enter the decimal numbers, as if they are Hexadecimal.
We add them by normal ADD instruction.
The answer is then adjusted using DA A instruction.

DA A always works on A Register only.

It first checks the Lower nibble of A Register.
    If Lower nibble is > 9 or Aux Carry is 1, then ADD 06H

It then checks the Higher nibble of A Register.
    If Higher nibble is > 9 or Carry Flag is 1, then ADD 60H

The final answer will be stored in A and Carry flag.

Please refer numerous examples discussed in the class. For doubts call #BharatSir @9820408217

Assume we want to add 25d + 25d, the result must obviously be 50d.

We enter the numbers as if they are hexadecimal, and add them by normal ADD instruction.

```
MOV A, #25H   ; A ← 25
ADD A, #25H   ; A ← 4A
```

Now we perform the adjustment using DAA instruction.

```
DA A          ; A ← 50
```

	24H			26H		28H		50H		80H		99H
	+ 25H			+ 26H		+ 28H		+ 50H		+ 80H		+ 99H
After ADD	= 49H		After ADD	= 4CH	After ADD	= 50H	After ADD	= A0H	After ADD	= 100H	After ADD	= 132H
After DA A	→ 49		After DA A	→ 52	After DA A	→ 56	After DA A	→ 100	After DA A	→ 160	After DA A	→ 198

No of cycles required: **1**

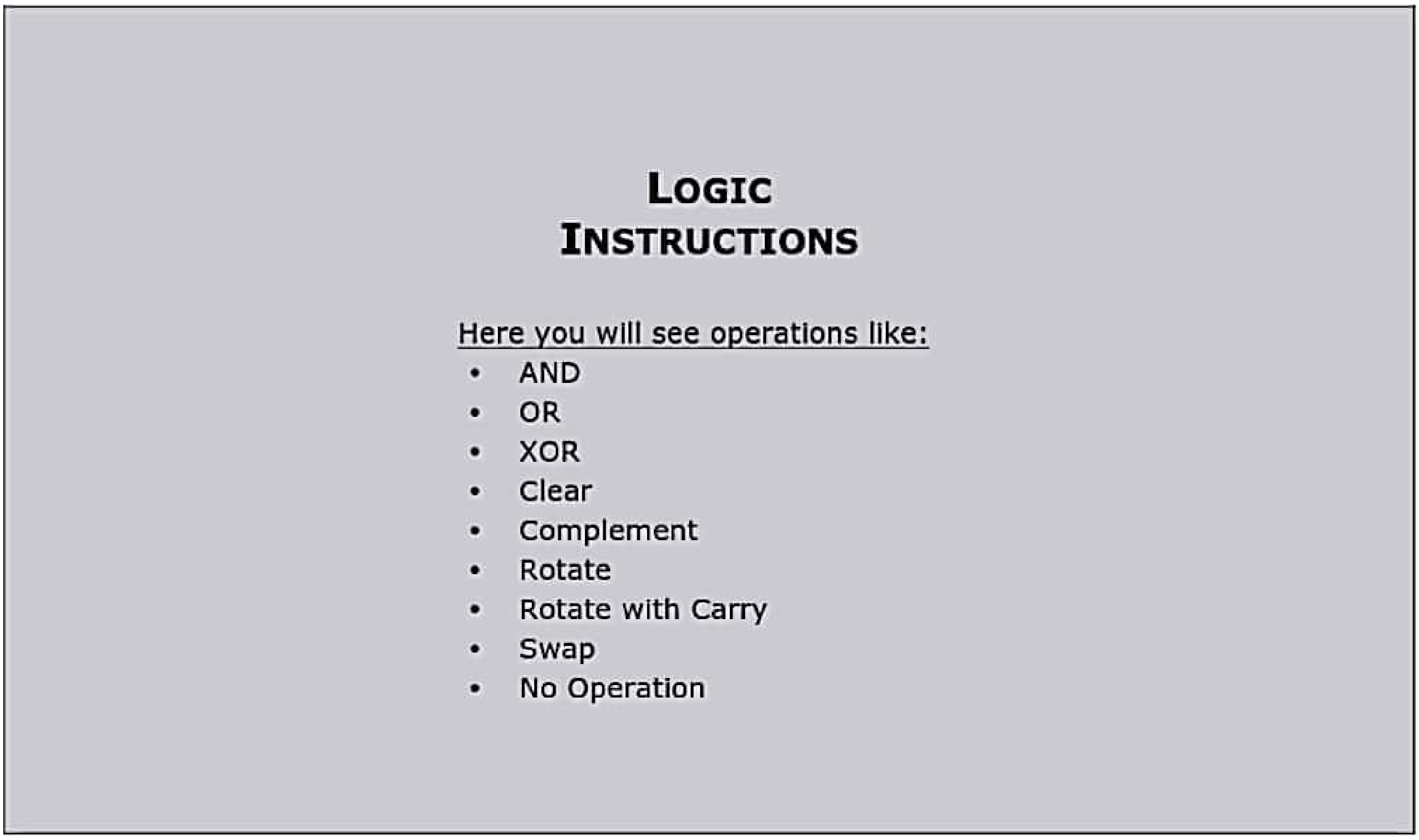
IMPORTANT TIP FROM BHARAT ACHARYA
Please get this clear, "DA A does not convert any number from Hexadecimal to Decimal".
It simply makes the addition work like decimal addition.

DA A can only be used "After performing an Addition" operation.

VIVA question: Put 25H in A register and show the working of DA A.
Reply: Invalid question! We must first perform addition. Simply putting 25H in A and doing DA A is absurd.

VIVA question: 29H and 3CH and show the working of DA A.
Reply: Invalid question! DA A is used to Add decimal numbers. 3C is not decimal!

Also, do remember, DA A is an adjustment for Addition. So there is always a chance of a carry.
If the answer exceeds 99, the lower two digits will be in A and the highest digit will be Carry Flag.
If the answer is 160, A will contain 60 and Carry Flag will get 1.



Logic Instructions

Here you will see operations like:

- AND
- OR
- XOR
- Clear
- Complement
- Rotate
- Rotate with Carry
- Swap
- No Operation

25) ANL A, #n | "AND logically"

Example:

ANL A, #25H; A ← A AND 25H

Operation:

Logically ANDs the value of A register with the immediate data.
Stores the result in A Register.

No of cycles required: **1**

IMPORTANT TIP FROM BHARAT ACHARYA

AND is used to CLEAR any bit of a register.
If we want to clear any bit, we must AND that particular bit with "0" and the remaining bits with "1".
This is because, if we AND anything (0 or 1) with 0, it becomes 0.
But if we AND anything (0 or 1) with 1, it remains the same.

Suppose we want to CLEAR the lower nibble of A register.
Assume: A register is 95H → 1001 0101
AND this register with the number F0H → 1111 0000
As a result A will become 90H → 1001 0000

Please refer examples form Bharat Academy lecture notes for more clarity on this.

---

26) ANL A, Rr | "AND logically"

Example:

ANL A, R0; A ← A AND R0

Operation:

Logically ANDs the value of A register with the value of RAM register.
Stores the result in A Register.

No of cycles required: **1**

---

27) ANL A, addr | "AND logically"

Example:

ANL A, 25H; A ← A AND [25H]

Operation:

Logically ANDs the value of A register with contents of the address.
Stores the result in A Register.

No of cycles required: **1**

28) ANL A, @Rp | "AND logically"

Example:

ANL A, @R0; A ← A AND [R0]

Operation:

Logically ANDs the value of A reg. with contents of the location pointed by the reg. Stores the result in A Register.

No of cycles required: **1**

---

29) ANL addr, A | "AND logically"

Example:

ANL 25H, A; [25H] ← [25H] AND A

Operation:

Logically ANDs contents of the address with the value of A register. Stores the result at the address.

No of cycles required: **1**

---

30) ANL addr, #n | "AND logically"

Example:

ANL 25H, #30H; [25H] ← [25H] AND 30H

Operation:

Logically ANDs contents of the address with the immediate data. Stores the result at the address.

No of cycles required: **2**

31) ORL A, #n

| "OR logically"

Example:

ORL A, #25H; A ← A OR 25H

Operation:

Logically ORs the value of A register with the immediate data.
Stores the result in A Register.

No of cycles required: **1**

IMPORTANT TIP FROM BHARAT ACHARYA

OR is used to SET any bit of a register.
If we want to set any bit, we must OR that particular bit with "1" and the remaining bits with "0".
This is because, if we OR anything (0 or 1) with 1, it becomes 1.
But if we OR anything (0 or 1) with 0, it remains the same.

Suppose we want to SET the lower nibble of A register.
Assume: A register is 95H → 1001 0101
OR this register with the number 0FH → 0000 1111
As a result A will become 9FH → 1001 1111

Please refer examples form Bharat Academy lecture notes for more clarity on this.

---

32) ORL A, Rr

| "OR logically"

Example:

ORL A, R0; A ← A OR R0

Operation:

Logically ORs the value of A register with the value of RAM register.
Stores the result in A Register.

No of cycles required: **1**

---

33) ORL A, addr

| "OR logically"

Example:

ORL A, 25H; A ← A OR [25H]

Operation:

Logically ORs the value of A register with contents of the address.
Stores the result in A Register.

No of cycles required: **1**

34) ORL A, @Rp

| "OR logically"

Example:

ORL A, @R0; A ← A OR [R0]

Operation:

Logically ORs the value of A reg. with contents of the location pointed by the reg.
Stores the result in A Register.

No of cycles required: **1**

---

35) ORL addr, A

| "OR logically"

Example:

ORL 25H, A; [25H] ← [25H] OR A

Operation:

Logically ORs contents of the address with the value of A register.
Stores the result at the address.

No of cycles required: **1**

---

36) ORL addr, #n

| "OR logically"

Example:

ORL 25H, #30H; [25H] ← [25H] OR 30H

Operation:

Logically ORs contents of the address with the immediate data.
Stores the result at the address.

No of cycles required: **2**

37) XRL A, #n

| "Ex-OR logically"

Example:

XRL A, #25H; A ← A XOR 25H

Operation:

Logically XORs the value of A register with the immediate data.
Stores the result in A Register.

No of cycles required: **1**

IMPORTANT TIP FROM BHARAT ACHARYA

XOR is used to COMPLEMENT any bit of a register.
If we want to complement any bit, we must XOR that bit with "1" and the remaining bits with "0".
This is because, if we XOR anything (0 or 1) with 1, it gets complemented.
But if we XOR anything (0 or 1) with 0, it remains the same.

Suppose we want to COMPLEMENT the lower nibble of A register.
Assume: A register is 95H → 1001 0101
XOR this register with the number 0FH → 0000 1111
As a result A will become 9AH → 1001 1010

Please refer examples form Bharat Academy lecture notes for more clarity on this.

---

38) XRL A, Rr

| "Ex-OR logically"

Example:

XRL A, R0; A ← A XOR R0

Operation:

Logically XORs the value of A register with the value of RAM register.
Stores the result in A Register.

No of cycles required: **1**

---

39) XRL A, addr

| "Ex-OR logically"

Example:

XRL A, 25H; A ← A XOR [25H]

Operation:

Logically XORs the value of A register with contents of the address.
Stores the result in A Register.

No of cycles required: **1**

40) XRL A, @Rp

| "Ex-OR logically"

Example:

XRL A, @R0; A ← A XOR [R0]

Operation:

Logically XORs the value of A reg. with contents of the location pointed by the reg.
Stores the result in A Register.

No of cycles required: **1**

---

41) XRL addr, A

| "Ex-OR logically"

Example:

XRL 25H, A; [25H] ← [25H] XOR A

Operation:

Logically XORs contents of the address with the value of A register.
Stores the result at the address.

No of cycles required: **1**

---

42) XRL addr, #n

| "Ex-OR logically"

Example:

XRL 25H, #30H; [25H] ← [25H] XOR 30H

Operation:

Logically XORs contents of the address with the immediate data.
Stores the result at the address.

No of cycles required: **2**

43) RL A | "Rotate Left"

Example:

RL A; A ← A register rotated left by one position

Operation:

Rotates the bits of A register in the left direction by one position.
Each bit goes one position to the left.
MSB goes to the Carry flag as well as to the LSB.

No of cycles required: **1**

IMPORTANT TIP FROM BHARAT ACHARYA
Rotates are used to determine the value of any bit, in A register.
To know the value of a bit, Rotate the register as many times, so that the bit comes into Carry Flag.
Now check the carry flag to know if your desired bit was 0 or 1.

---

44) RR A | "Rotate Right"

Example:

RR A; A ← A register rotated right by one position

Operation:

Rotates the bits of A register in the right direction by one position.
Each bit goes one position to the right.
LSB goes to the Carry flag as well as to the MSB.

No of cycles required: **1**

---

45) RLC A | "Rotate Left, with Carry"

Example:

RLC A; A ← A register rotated left by one position along with the carry flag

Operation:

Rotates the bits of A register in the left direction by one position along with the carry flag.
Each bit goes one position to the left.
MSB goes to the Carry flag and Carry Flag goes to the LSB.

No of cycles required: **1**

---

46) RRC A | "Rotate Right, with Carry"

Example:

RRC A; A ← A register rotated right by one position along with the carry flag

Operation:

Rotates the bits of A register in the right direction by one position along with the carry flag.
Each bit goes one position to the right.
LSB goes to the Carry flag and Carry Flag goes to the MSB.

No of cycles required: **1**

47) CPL A

| "Complement A"

Example:

CPL A; A ← One's complement of A.

Operation:

Complements the value of A register.
Works just like a Not gate.

No of cycles required: **1**

<u>**IMPORTANT TIP FROM BHARAT ACHARYA**</u>
CPL is used as a NOT operation.
If we do CPL after AND, it works like **NAND**.
CPL after OR, it works like **NOR**.
CPL after XOR, it works like **XNOR**.

---

48) CLR A

| "Clear A"

Example:

CLR A; A ← 00H.

Operation:

Clears the entire A register and makes it 00H.

No of cycles required: **1**

---

49) SWAP A

| "SWĀP A"

Example:

SWAP A; A $_{\text{Lower Nibble}}$ **←→ A** $_{\text{Higher Nibble}}$

Operation:

Interchanges the Nibbles of A register.
If A register was 35H it will become 53H.
It is as good as rotating A register four times.

No of cycles required: **1**

---

50) NOP

| "No operation"

Example:

NOP; No operation. PC simply becomes PC + 1.

Operation:

This instruction performs no operation.
It is typically used to produce a delay.

No of cycles required: **1**

Branch operations of 8051 (SJMP, AJMP and LJMP)

Short Jump

Syntax: **SJMP radd**; // Short Jump using the relative address

Range: **(-128 … +127) locations** because "radd" is an 8-bit signed number

Size of instruction: **2 Bytes** (Opcode of SJMP= 1Byte, radd = 1Byte)

New address calculation: **PC ← PC (address of next instruction) + radd**

Usage: **SJMP** (unconditional) and ALL Conditional jumps like JC, JNC, CJNE, DJNZ etc. ← Important for VIVA

Description: *Here the branch address (radd) is calculated as a relative distance from the next instruction to the branch location. In simple terms, instead of telling where we want to jump, we are telling how far we want to jump. This "radd" is then added to PC which normally contains address of the next instruction.* ***For examples of SJMP, please refer #BharatSir Lecture Notes***

Absolute Jump

Syntax: **AJMP sadd**;// Absolute Jump using the short address

Range: **max 2KB** as long as the Jump is within the **Same Page**

Size of instruction: **2 Bytes** (Opcode of AJMP= 1Byte, sadd = 1Byte)

New address calculation:

PC ←	PC	Opcode of AJMP	Sadd
(16)	5 bits Remains the same as branch is in the same page	3 bits Hence AJMP has 8 opcodes	8 bits Lower 8 bits of the jump location

Usage: **AJMP and ACALL.**

Description: *Here the entire program memory (64 KB), is divided into 32 pages, each page being of 2KB. We can jump to any location of the same page, giving a max range of 2 KB. As the jump is in the same page, only the lower 11 bits of the address will change. Out of them, lower 8 bits are given by "sadd" and the higher 3 bits are given by the opcode of AJMP. 3 bits have 8 combinations, hence AJMP has 8 opcodes.* ***For examples of AJMP, please refer #BharatSir Lecture Notes***

Long Jump

Syntax: **LJMP ladd;** // Long Jump using the long (full) address

Range: **64 KB** because "ladd" is a 16-bit address so can be any value from 0000H… FFFFH.

Size of instruction: **3 Bytes** (Opcode of LJMP= 1Byte, ladd = 2Bytes)

New address calculation: **PC ← ladd**

Usage: **LJMP, LCALL.**

Description: *This is the simplest type of Jump. Here we simply give the address where we wish to jump using "ladd". This "ladd" is then simply put into PC.* ***For examples of LJMP, please refer #BharatSir Lecture Notes***

The flow of program proceeds in a sequential manner, from one instruction to the next instruction, unless a control transfer instruction is executed. The various types of control transfer instruction in assembly language include conditional or unconditional jumps and call instructions.

Loop and Jump Instructions

Looping in the 8051

Repeating a sequence of instructions a certain number of times is called a **loop**. An instruction **DJNZ reg, label** is used to perform a Loop operation. In this instruction, a register is decremented by 1; if it is not zero, then 8051 jumps to the target address referred to by the label.

The register is loaded with the counter for the number of repetitions prior to the start of the loop. In this instruction, both the registers decrement and the decision to jump are combined into a single instruction. The registers can be any of R0–R7. The counter can also be a RAM location.

Other Conditional Jumps

The following table lists the conditional jumps used in 8051 –

Instruction	Action
JZ	Jump if A = 0
JNZ	Jump if A ≠ 0
DJNZ	Decrement and Jump if register ≠ 0
CJNE A, data	Jump if A ≠ data
CJNE reg, #data	Jump if byte ≠ data
JC	Jump if CY = 1
JNC	Jump if CY ≠ 1
JB	Jump if bit = 1
JNB	Jump if bit = 0
JBC	Jump if bit = 1 and clear bit

- **JZ (jump if A = 0)** – In this instruction, the content of the accumulator is checked. If it is zero, then the 8051 jumps to the target address. JZ instruction can be used only for the accumulator, it does not apply to any other register.

- **JNZ (jump if A is not equal to 0)** – In this instruction, the content of the accumulator is checked to be non-zero. If it is not zero, then the 8051 jumps to the target address.

- **JNC (Jump if no carry, jumps if CY = 0)** – The Carry flag bit in the flag (or PSW) register is used to make the decision whether to jump or not "JNC label". The CPU looks at the carry flag to see if it is raised (CY = 1). If it is not raised, then the CPU starts to fetch and execute instructions from the address of the label. If CY = 1, it will not jump but will execute the next instruction below JNC.

- **JC (Jump if carry, jumps if CY = 1)** – If CY = 1, it jumps to the target address.
- **JB (jump if bit is high)**
- **JNB (jump if bit is low)**

Note – It must be noted that all conditional jumps are short jumps, i.e., the address of the target must be within –128 to +127 bytes of the contents of the program counter.

Unconditional Jump Instructions

There are two unconditional jumps in 8051 –

- **LJMP (long jump)** – LJMP is 3-byte instruction in which the first byte represents opcode, and the second and third bytes represent the 16-bit address of the target location. The 2-byte target address is to allow a jump to any memory location from 0000 to FFFFH.

- **SJMP (short jump)** – It is a 2-byte instruction where the first byte is the opcode and the second byte is the relative address of the target location. The relative address ranges from 00H to FFH which is divided into forward and backward jumps; that is, within −128 to +127 bytes of memory relative to the address of the current PC (program counter). In case of forward jump, the target address can be within a space of 127 bytes from the current PC. In case of backward jump, the target address can be within −128 bytes from the current PC.

Calculating the Short Jump Address

All conditional jumps (JNC, JZ, and DJNZ) are short jumps because they are 2-byte instructions. In these instructions, the first byte represents opcode and the second byte represents the relative address. The target address is always relative to the value of the program counter. To calculate the target address, the second byte is added to the PC of the instruction immediately below the jump.

Backward Jump Target Address Calculation

In case of a forward jump, the displacement value is a positive number between 0 to 127 (00 to 7F in hex). However, for a backward jump, the displacement is a negative value of 0 to −128.

CALL Instructions

CALL is used to call a subroutine or method. Subroutines are used to perform operations or tasks that need to be performed frequently. This makes a program more structured and saves memory space. There are two instructions – LCALL and ACALL.

LCALL (Long Call)

LCALL is a 3-byte instruction where the first byte represents the opcode and the second and third bytes are used to provide the address of the target subroutine. LCALL can be used to call subroutines which are available within the 64K-byte address space of the 8051.

To make a successful return to the point after execution of the called subroutine, the CPU saves the address of the instruction immediately below the LCALL on the stack. Thus, when a subroutine is called, the control is transferred to that subroutine, and the processor saves the PC (program counter) on the stack and begins to fetch instructions from the new location. The instruction RET (return) transfers the control back to the caller after finishing execution of the subroutine. Every subroutine uses RET as the last instruction.

ACALL (Absolute Call)

ACALL is a 2-byte instruction, in contrast to LCALL which is 3 bytes. The target address of the subroutine must be within 2K bytes because only 11 bits of the 2 bytes are used for address. The difference between the ACALL and LCALL is that the target address for LCALL can be anywhere within the 64K-bytes address space of the 8051, while the target address of CALL is within a 2K-byte range.

TIMER SECTION OF 8051

8051 has **2, 16-bit Up Counters** T1 and T0.
If the counter **counts internal clock** pulses it is known as **timer**.
If it counts **external clock** pulses it is known as **counter**.
Each counter is divided into 2, 8-bit registers TH1 - TL1 and TH0 - TL0.
The **timer** action is **controlled** mainly by the **TCON** and the **TMOD** registers.

TCON - Timer Control (SFR) [Bit-Addressable As TCON.7 to TCON.0]

TF1	**TR1**	**TF0**	**TR0**	IE1	IT1	IE0	IT0

TF1 and TF0: (Timer Overflow Flag)
Set (1) when Timer 1 or Timer 0 overflows respectively i.e. its bits roll over from all 1's to all 0's.
Cleared (0) when the processor executes ISR (address 001BH for Timer 1 and 000BH for Timer 0).

TR1 and TR0: (Timer Run Control Bit)
Set (1) - Starts counting on Timer 1 or Timer 0 respectively.
Cleared (0) - Halts Timer 1 or Timer 0 respectively.

IE1 and IE0: (External Interrupt Edge Flag)
Set (1) when external interrupt signal received at $\overline{\textbf{INT1}}$ or $\overline{\textbf{INT0}}$ respectively.
Cleared (0) when ISR executed (address 0013H for Timer 1 and 0003H for Timer 0).

IT1 and IT0: (External Interrupt Type Cobtrol Bit)
Set (1) - Interrupt at $\overline{\textbf{INT1}}$ or $\overline{\textbf{INT0}}$ must be -ve edge triggered.
Cleared (0) - Interrupt at $\overline{\textbf{INT1}}$ or $\overline{\textbf{INT0}}$ must be low-level triggered.

TMOD - Timer Mode Control (SFR) [NOT Bit-Addressable]

Gate	C/T	M1	M0	Gate	C/T	M1	M0
Timer 1				Timer 0			

C/T: (Counter/Timer)
Set (1) - Acts as Counter (Counts external frequency on T1 and T0 pin inputs).
Cleared (0) - Acts as Timer (Counts internal clock frequency, fosc/12).

Gate: (Gate Enable Control bit)
Set (1) - Timer controlled by hardware i.e. INTX signal.
Cleared (0) – Counting independent of INTX signal.

M1, M0: (Mode Selection bits)
Used to select the operational modes of the respective Timer.

M1	M0	Timer Mode
0	0	Mode 0
0	1	Mode 1
1	0	Mode 2
1	1	Mode 3

Timer Counter Interrupts

To use the timer, a certain **count value** is placed **in** the **Count Register**.

This value is the **Max Count - Desired Count + 1**.

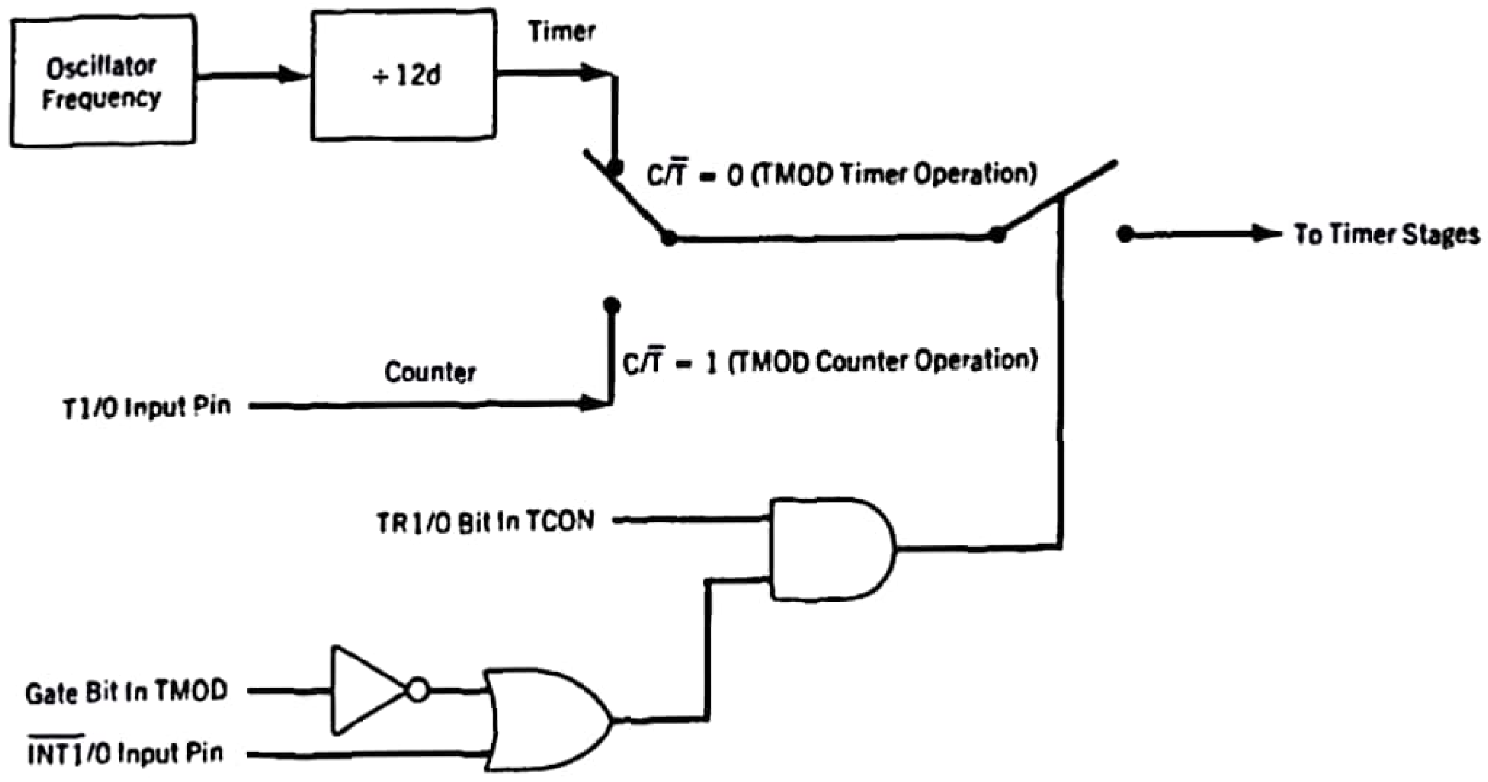
On **each count** (rising edge of the input clock) the **counter increments** its value.

When the counter rolls over (i.e. form **all 1's to all 0's**) it is said to **overflow**.

Thus the Timer Overflow Flag, **TFX** (TF1 or TF0) is **set**.

If timer **interrupt** is **enabled** then the **Timer Interrupt will occur** on overflow.

Timer Counter Logic


Oscillator Frequency
÷12d
Timer
C/$\overline{T}$ = 0 (TMOD Timer Operation)
To Timer Stages
Counter
C/$\overline{T}$ = 1 (TMOD Counter Operation)
T1/0 Input Pin
TR1/0 Bit In TCON
Gate Bit In TMOD
$\overline{INT1}$/0 Input Pin

As shown above, based on **C/T** bit the timer functions as a **Counter** or as a **Timer.**

If it is a **Timer,** It will count the **internal clock frequency** of 8051 **divided by 12_d** (f/12).

If it is a **Counter,** the **input clock signal** is applied at the **TX** (T1 or T0) input pins for Timer1 or Timer0 respectively. #Please refer Bharat Sir's Lecture Notes for this ...

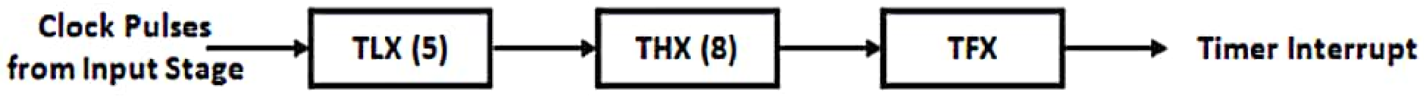
As shown the **Timer** is **running only if** the **TRX** bit (TR1 or TR0) **is set.**

Also **if** the **Gate** bit is **set** in the TMOD then the **INTX** (**$\overline{\text{INT1}}$** or **$\overline{\text{INT0}}$**) pin **must be "high (1)"** for the timer to count.

Scanned by CamScanner

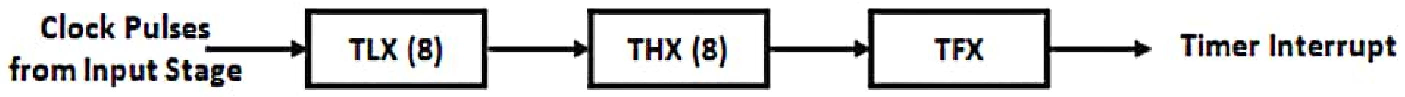
Timer Modes

a) **Timer Mode 0** (13-bit Timer/Counter)



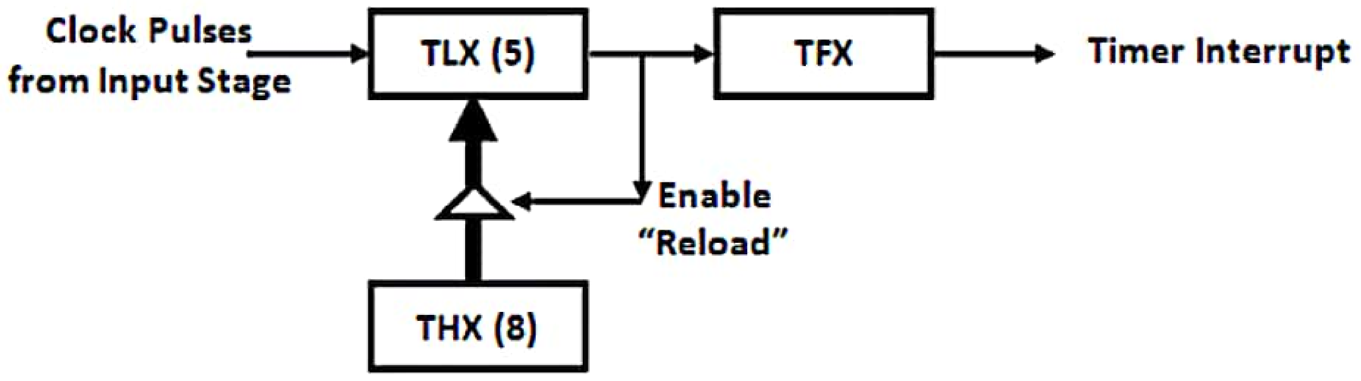
THX is used as an **8-bit counter.**
TLX is used as a **5-bit pre-set**. Hence **13-bits** are used for counting.
On **each count** the **TLX increments.**
Each time **TLX rolls-over, THX increments.**
Thus the **input frequency** is **divided by 32** (5-bits of TLX and **2^5 = 32**).
The timer overflow flag **TFX** is **set** only **when THX overflows** i.e. rolls from FFH to 00H.
Max Count = 2^{13} = 8K = 8192 (1FFFH). Hence **Max Delay → 8192(12/f)**

b) **Timer Mode 1** (16-bit Timer/Counter)



All **16-bits** of the **Counter** are used (8 bits of THX and 8 bits of TLX).
On **each count** the 16-bit **Timer increments.**
The timer overflow flag **TFX** is **set when** the **Timer rolls-over from FFFFH to 0000H**.
Max Count => 2^{16} = 16K = 65536 (FFFFH). Hence **Max Delay → 65536(12/f).**

c) **Timer Mode 2** (Auto reload TL from TH)

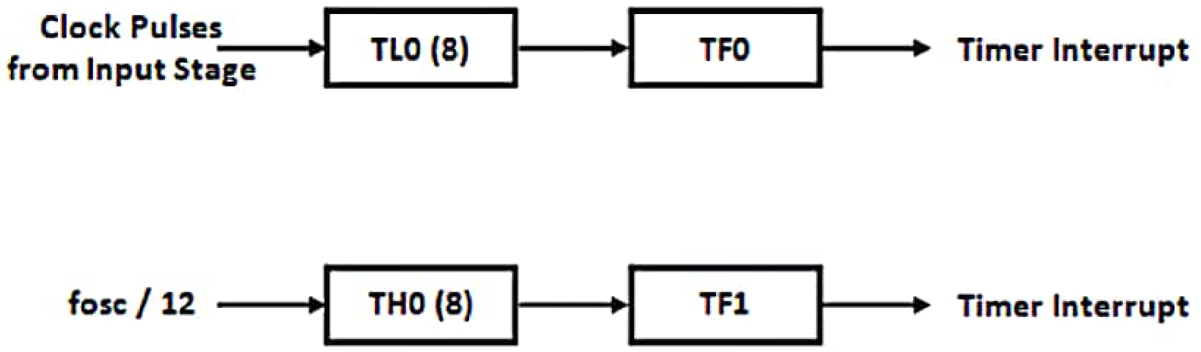


TLX is used as an **8-bit counter**.
THX holds the **count value** to be **reloaded**.
On **each count TLX increments**.
When **TLX rolls-over** (i.e. from FFH to 00H), the following events take place:

1. Timer overflow flag **TFX** is **set**, hence timer interrupt occurs.
2. The value of **THX** is **copied into TLX**. Hence TLX is **auto-reloaded** form THX, and the process repeats.

Thus the timer interrupt occurs at regular intervals "**Continuously**".
This mode is used to generate a desired frequency using the Timer Flag.
Max Count → 2^8 = 256 (FFH). Hence **Max Delay → 256(12/f)**.

d) **Timer Mode 3** (Two 8-bit Timers Using Timer0)



Timer 0 is used **as 2 separate 8-bit timers TH0 and TL0**.
TL0 uses the control bits (**TR0 and TF0**) of Timer 0.
It can work as a **Timer or** a **Counter**.
TH0 uses the control bits (**TR1 and TF1**) of Timer 1.
It can work only as a **Timer**. #Please refer Bharat Sir's Lecture Notes for this ...
Timer 1 can be in Mode 0, Mode 1, or Mode 2, but will not generate an interrupt.

8051 TIMER/COUNTER (HARDWARE DELAY) PROGRAMMING

Q10

WAP to generate a delay of 20 µsec using internal timer-0 of 8051. After the delay send a "1" through Port3.1. Assume Suitable Crystal Frequency

NOTE: In 8051, if we select a Crystal of 12 MHz, then Timer freq will be $f_{osc}/12$ ➔ 1MHz. Hence each count will require 1/1MHz ➔ 1 µsec. Thus for 20 µsec, the Desired Count will be 20_d ➔ 14H. For an Up-Counter (Mode 1):

Count = Max Count – Desired Count + 1
Count = FFFF - 14 + 1
Count = FFECH #Please refer Bharat Sir's Lecture Notes for this ...

```
SOLN:  MOV TMOD, #01H       ; Program TMOD ➔ (0000 0001)2... Timer0 Mode1
       MOV TL0, #0ECH       ; Load lower byte of Count
       MOV TH0, #0FFH       ; Load upper byte of Count
       MOV TCON, #10H       ; Program TCON ➔ (0001 0000)2 ... start Timer0
WAIT:  JNB TCON.5, WAIT     ; Wait for overflow
       SETB P3.1            ; Send a "1" through Port3.1
       MOV TCON, #00H       ; Stop Timer0
HERE:  SJMP HERE            ; End of program
```

Q11

WAP to generate a Square wave of 1 KHz from the TxD pin of 8051, using Timer1. Assume Clock Frequency of 12 MHz.

NOTE: For a Square wave of 1 KHz, the delay required is .5 msec.
We know, each count will require 1/1MHz ➔ 1 µsec.
Thus for 500 µsec, the Desired Count will be 500_d ➔ 01F4H. For an Up-Counter (Mode 1):

Count = Max Count - Desired Count + 1
Count = FFFF - 01F4 + 1
Count = FE0CH

```
SOLN:    CLR P3.1             ; Clear Txd Line initially
         MOV TMOD, #10H       ; Program TMOD ➔ (0001 0000)2... Timer1 Mode1
REPEAT:  MOV TL1, #0CH        ; Load lower byte of Count
         MOV TH1, #0FEH       ; Load upper byte of Count
         MOV TCON, #40H       ; Program TCON ➔ (0100 0000)2 ... start Timer1
WAIT:    JNB TCON.7, WAIT     ; Wait for overflow
         CPL P3.1             ; Toggle Txd pin after the delay
         MOV TCON, #00H       ; Stop Timer1
         SJMP REPEAT          ; Repeat the process
```

Q12 **WAP to generate a Rectangular wave of 1 KHz, having a 25% Duty Cycle from the TxD pin of 8051, using Timer1. Assume XTAL of 12 MHz.**

NOTE: For a Rectangular wave of 1 KHz, having 25% Duty Cycle:
T_{ON} = 250 µsec; T_{OFF} = 750 µsec.

For T_{ON}: Desired Count = 250 $_d$ → 00FAH
$Count_{ON}$ = Max Count − Desired Count + 1
$Count_{ON}$ = FFFF − 00FA + 1
$Count_{ON}$ = FF06H

For T_{OFF}: Desired Count = 750 $_d$ → 02EEH
$Count_{OFF}$ = Max Count − Desired Count + 1
$Count_{OFF}$ = FFFF - 02EE + 1
$Count_{OFF}$ = FD12H

```
SOLN:    MOV TMOD, #10H        ; Program TMOD → (0001 0000)2 ... Timer1 Mode1

REPEAT:  MOV TL1, #06H         ; Load lower byte of CountON
         MOV TH1, #0FFH        ; Load upper byte of CountON
         SETB P3.1             ; Display "1" at Txd
         MOV TCON, #40H        ; Program TCON → (0100 0000)2 ... start Timer1
   ON:   JNB TCON.7, ON        ; Maintain "1" at Txd
         CLR P3.1              ; Clear Txd
         MOV TCON, #00H        ; Stop Timer1

         MOV TL1, #12H         ; Load lower byte of CountOFF
         MOV TH1, #0FDH        ; Load upper byte of CountOFF
         MOV TCON, #40H        ; Program TCON → (0100 0000)2 ... start Timer1
  OFF:   JNB TCON.7, OFF       ; Maintain "0" at Txd
         MOV TCON, #00H        ; Stop Timer1

         SJMP REPEAT           ; Repeat the process
```

Note: If System Freq = 12MHz, it is clear that 1 Count requires 1 msec.
In Mode 1, we have a 16bit Count.
Hence max pulses that can be desired is 2^{16} = 65536.
Count = Max Count − Desired Count + 1
= 65535 − 65536 + 1
= 0.
Thus we will get max delay if we load the count as 0000H, as it will have to "roll-over" back to 0000H to overflow.
Hence Max delay if XTAL is of 12 MHz ... is 65536 µsec → 65.536 msec.
Similarly Max delay if XTAL is of 11.0592 MHz ... is 71106 µsec → 71.106 msec.

Q13 **WAP to generate a delay of 1 SECOND using Timer1. Assume Clock Frequency of 12 MHz. (Popular Question in College!)**

NOTE: Max delay if XTAL is of 12 MHz … is 65536 μsec → 65.536 msec.
Hence to get a delay of 1 second, we will have to perform the counting repeatedly in a loop.
Lets keep the Desired Count 50000. (50 msec delay)
Now 50000_d = C350H
Count = Max Count – Desired Count + 1
Count = FFFF – C350 + 1
Count = 3CB0H #Please refer Bharat Sir's Lecture Notes for this ...
We will have to perform this counting 1sec/50msec times → **20 times**

```
 SOLN:  MOV TMOD, #10H        ; Program TMOD → (0001 0000)2 … Timer1 Mode1
        MOV R0, #14H          ; Load count 20 in R0
REPEAT: MOV TL1, #0B0H        ; Load lower byte of CountON
        MOV TH1, #3CH         ; Load upper byte of CountON
        MOV TCON, #40H        ; Program TCON → (0100 0000)2 … start Timer1
 WAIT:  JNB TCON.1, WAIT      ; Wait for an overflow
        MOV TCON, #00H        ; Stop Timer1
        DJNZ R0, REPEAT       ; repeat the process 20 times
 HERE:  SJMP HERE:            ; End of program
```

Q14 **WAP to read the data from Port1, 10 times, each after a 1 sec delay. Store the data from RAM locations 20H onwards. When the operation is complete, ring an "Alarm" connected at Port3.1. Assume CLK = 12 MHz.**

NOTE: As seen from the previous program, for a delay of 1 second, we have **Count = 3CB0H**. Counting has to be performed **20 times.**
Also note that all ports of 8051 are o/p ports by default.
To program a port as i/p ports, **all "1"s** must be sent though it.

```
   SOLN:  CLR P3.1             ; Clear Port3.1 line
          MOV TMOD, #10H       ; Program TMOD → (0001 0000)2 ... Timer1 Mode1
          MOV 90H, #0FFH       ; Program Port1 as i/p by sending all "1"s through it

 REPEAT:  MOV R0, #0AH         ; Load Data Count of 10 in R0
          MOV R1, #20H         ; Load Storage address in R1
          MOV @R1, 90H         ; Read data from Port
          INC R1               ; Increment data storage address from next Iteration
          ACALL DELAY          ; Call delay of 1 sec before going into next Iteration
          DJNZ R0, REPEAT      ; Repeat till all 10 bytes are read
          SETB P3.1            ; Ring "Alarm" at Port3.1
   HERE:  SJMP HERE:           ; End of program

  DELAY:  MOV R2, #14H         ; Load count 20 in R0
 REPEAT:  MOV TL1, #0B0H       ; Load lower byte of CountON
          MOV TH1, #3CH        ; Load upper byte of CountON
          MOV TCON, #40H       ; Program TCON → (0100 0000)2 ... start Timer1
   WAIT:  JNB TCON.1, WAIT     ; Wait for an overflow
          MOV TCON, #00H       ; Stop Timer1
          DJNZ R2, REPEAT      ; Repeat the process 20 times
          RET                  ; End of delay routine
```

INTERRUPTS OF 8051

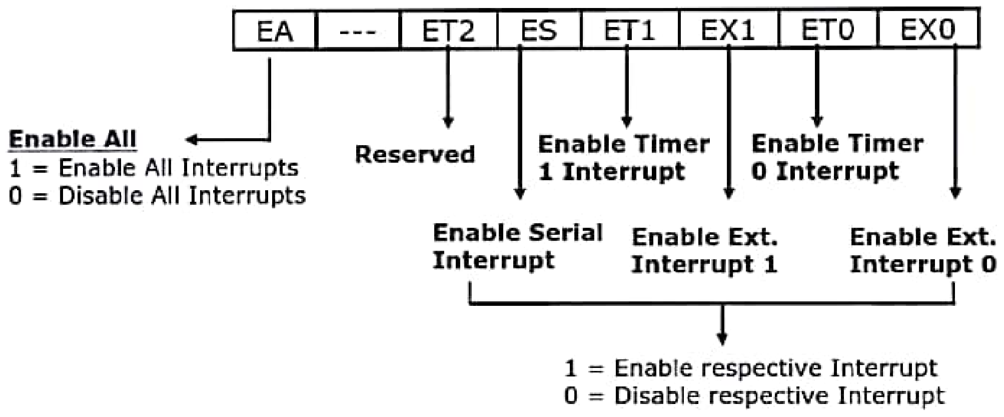
8051 supports **5** interrupts.
2 External Interrupts are on the following pins

$\overline{\textbf{INT1}}$

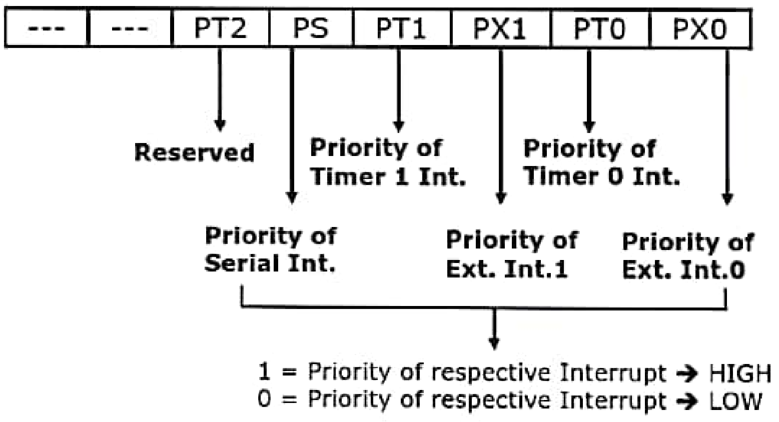
$\overline{\textbf{INT0}}$

2 Internal Timer interrupts are:
Timer 1 Overflow Interrupt
Timer 0 Overflow Interrupt
1 Serial Port Interrupt (Common for RI or TI)
All interrupts are **vectored** i.e. they cause the program to execute an ISR from a pre-determined address in the Program Memory. #Please refer Bharat Sir's Lecture Notes for this ...
Interrupts are controlled mainly by **IE** and **IP** SFR's and also by some bits of **TCON** SFR.

IE - Interrupt Enable (SFR) [Bit-Addressable As IE.7 to IE.0]



IP - Interrupt Priority (SFR) [Bit-Addressable As IP.7 to IP.0]



Timer Overflow Interrupts (TF1 and TF0)

When any of the 2 **Timers overflow**, their respective bit **TFX** (TF1 or TF0) is **set** in **TCON** SFR.
If Timer Interrupts are enabled then the **timer interrupt occurs**.
The **TFX** bits are **cleared** when their respective **ISR** is executed.

Serial Port Interrupt (RI or TI)

While receiving serial data, when a **complete byte** is **received** the **RI** (receive interrupt) bit is set in the **SCON.**
During transmission, when a **complete byte** is **transmitted** the **TI** (transmit interrupt) bit is set in the **SCON**.
ANY of these events can cause the **Serial Interrupt** (provided Serial Interrupt is enabled).
The **RI/TI** bit is **not cleared** automatically on **executing** the **ISR**. The program should **explicitly clear** this bit to allow further Serial Interrupts.

External Interrupts ($\overline{\textbf{INT1}}$ and $\overline{\textbf{INT0}}$)

Pins $\overline{\textbf{INT1}}$ and $\overline{\textbf{INT0}}$ are inputs for external interrupts.
These interrupts can be -ve **edge** or low-**level triggered** depending upon the **IT0 and IT1** bit in **TCON** SFR. (ITX = 1 → -ve edge triggered)
Whew any of these interrupts occur the respective bits **TE1** or **IE0** are **set** in the **TCON** SFR.
If External Interrupts are enabled then the **ISR** is **executed** from the respective address.

Interrupt Sequence

The following sequence is executed to service an interrupt:
Address of next instruction of the main program i.e. **PC** is **Pushed** into the **Stack**.
All **interrupts** are **disabled**, by making EA bit in IE SFR ← 0.
Program Control is shifted to the **Vector Address** (location) of the **ISR**.
The **ISR begins**.

Returning Sequence

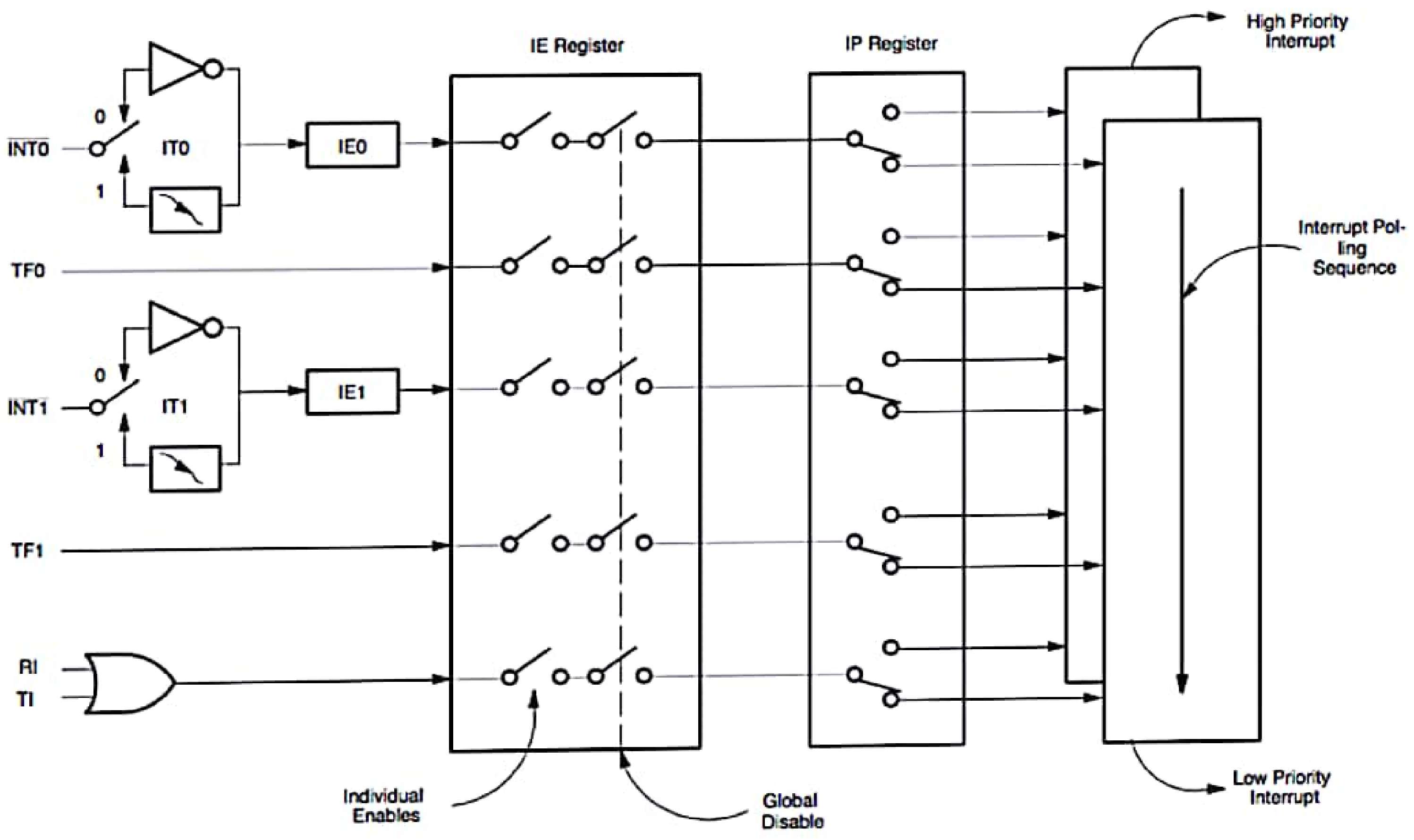
RETI instruction denotes the **end** of the **ISR**.
It causes the processor to **POP** the contents of the Stack Top into the **PC.**
It also re-enables interrupts by making EA bit in IE SFR ← 1.
The **main program resumes**.

Interrupt Priorities

8051 has only **two priority levels** for the interrupts: **Low** and **High**.
Interrupt priorities are set using the **IP** SFR.
As the name suggests, a high priority interrupt can interrupt a low priority interrupt.
It two or more interrupts at the same level occur simultaneously then priorities are decided as follows:

INTERRUPT	PRIORITY	VECTOR ADDRESS
$\overline{\text{INT0}}$	1	0003H
TF0	2	000BH
$\overline{\text{INT1}}$	3	0013H
TF1	4	001BH
Serial (RI or TI)	5	0023H

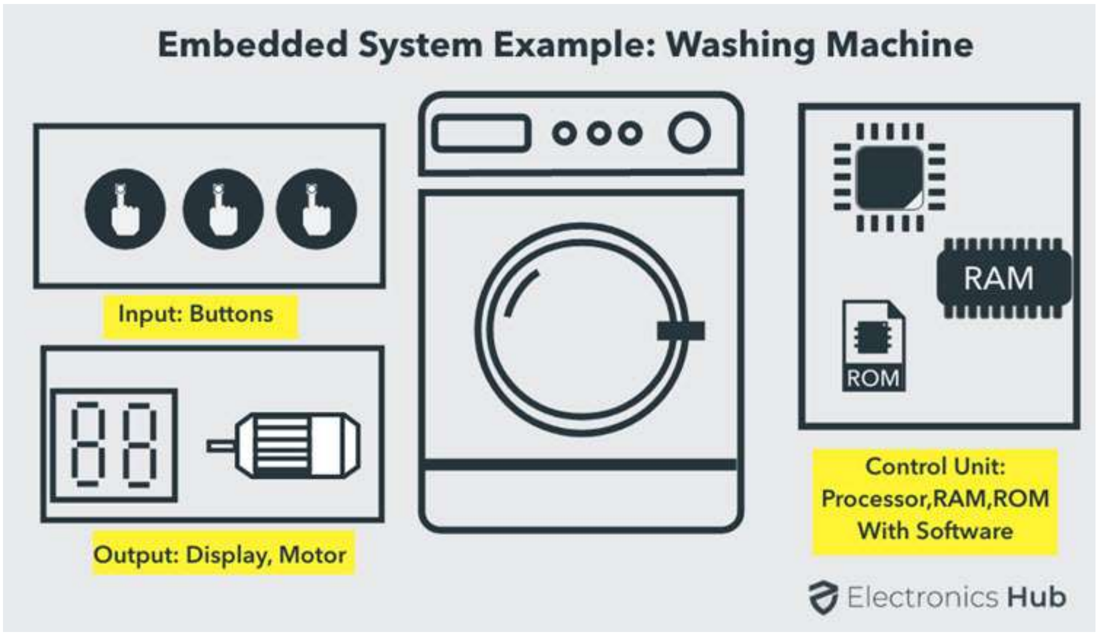
DIAGRAM FOR INTERRUPTS... OPTIONAL
DRAW ONLY IF ASKED



What is an Embedded System?

An Embedded System can be best described as a system which has both the hardware and software and is designed to do a specific task. A good example for an Embedded System, which many households have, is a Washing Machine.

We use washing machines almost daily but wouldn't get the idea that it is an embedded system consisting of a Processor (and other hardware as well) and software.


Embedded System Example: Washing Machine
Input: Buttons
Output: Display, Motor
RAM
ROM
Control Unit: Processor,RAM,ROM With Software
Electronics Hub

It takes some inputs from the user like wash cycle, type of clothes, extra soaking and rinsing, spin rpm, etc., performs the necessary actions as per the instructions and finishes washing and drying the clothes. If no new instructions are given for the next wash, then the washing machines repeats the same set of tasks as the previous wash.

Embedded Systems can not only be stand-alone devices like Washing Machines but also be a part of a much larger system. An example for this is a Car. A modern day Car has several individual embedded systems that perform their specific tasks with the aim of making a smooth and safe journey.

Some of the embedded systems in a Car are Anti-lock Braking System (ABS), Temperature Monitoring System, Automatic Climate Control, Tire Pressure Monitoring System, Engine Oil Level Monitor, etc.

Programming Embedded Systems

As mentioned earlier, Embedded Systems consists of both Hardware and Software. If we consider a simple Embedded System, the main Hardware Module is the Processor. The Processor is the heart of the Embedded System and it can be anything like a Microprocessor, Microcontroller, DSP, CPLD (Complex Programmable Logic Device) or an FPGA (Field Programmable Gated Array).

All these devices have one thing in common: they are programmable i.e., we can write a program (which is the software part of the Embedded System) to define how the device actually works.

Embedded Software or Program allow Hardware to monitor external events (Inputs / Sensors) and control external devices (Outputs) accordingly. During this process, the program for an Embedded System may have to directly manipulate the internal architecture of the Embedded Hardware (usually the processor) such as Timers, Serial Communications Interface, Interrupt Handling, and I/O Ports etc.

From the above statement, it is clear that the Software part of an Embedded System is equally important as the Hardware part. There is no point in having advanced Hardware Components with poorly written programs (Software).

There are many programming languages that are used for Embedded Systems like Assembly (low-level Programming Language), C, C++, JAVA (high-level programming languages), Visual Basic, JAVA Script (Application level Programming Languages), etc.

In the process of making a better embedded system, the programming of the system plays a vital role and hence, the selection of the Programming Language is very important.

Factors for Selecting the Programming Language

The following are few factors that are to be considered while selecting the Programming Language for the development of Embedded Systems.

- **Size:** The memory that the program occupies is very important as Embedded Processors like Microcontrollers have a very limited amount of ROM (Program Memory).
- **Speed**: The programs must be very fast i.e., they must run as fast as possible. The hardware should not be slowed down due to a slow running software.
- **Portability:** The same program can be compiled for different processors.
- Ease of Implementation
- Ease of Maintenance
- Readability

Earlier Embedded Systems were developed mainly using Assembly Language. Even though Assembly Language is closest to the actual machine code instructions and produces small size hex files, the lack of portability and high amount of resources (time and man power) spent on developing the code, made the Assembly Language difficult to work with.

There are other high-level programming languages that offered the above mentioned features but none were close to C Programming Language. Some of the benefits of using Embedded C as the main Programming Language:

- Significantly easy to write code in C
- Consumes less time when compared to Assembly
- Maintenance of code (modifications and updates) is very simple
- Make use of library functions to reduce the complexity of the main code
- You can easily port the code to other architecture with very little modifications

Introduction to Embedded C Programming Language

Before going in to the details of Embedded C Programming Language and basics of Embedded C Program, we will first talk about the C Programming Language.

The C Programming Language, developed by Dennis Ritchie in the late 60’s and early 70’s, is the most popular and widely used programming language. The C Programming Language provided low level memory access using an uncomplicated compiler (a software that converts programs to machine code) and achieved efficient mapping to machine instructions.

The C Programming Language became so popular that it is used in a wide range of applications ranging from Embedded Systems to Super Computers.

Embedded C Programming Language, which is widely used in the development of Embedded Systems, is an extension of C Program Language. The Embedded C Programming Language uses the same syntax and semantics of the C Programming Language like main function, declaration of datatypes, defining variables, loops, functions, statements, etc.

The extension in Embedded C from standard C Programming Language include I/O Hardware Addressing, fixed point arithmetic operations, accessing address spaces, etc.

Difference between C and Embedded C

There is actually not much difference between C and Embedded C apart from few extensions and the operating environment. Both C and Embedded C are ISO Standards that have almost same syntax, datatypes, functions, etc.

Embedded C is basically an extension to the Standard C Programming Language with additional features like Addressing I/O, multiple memory addressing and fixed-point arithmetic, etc.

C Programming Language is generally used for developing desktop applications, whereas Embedded C is used in the development of Microcontroller based applications.

Basics of Embedded C Program

Now that we have seen a little bit about Embedded Systems and Programming Languages, we will dive in to the basics of Embedded C Program. We will start with two of the basic features of the Embedded C Program: Keywords and Datatypes.

Why program the 8051 in C?

Compilers produce hex files that we download into the ROM of the microcontroller. The size of the hex file produced by the compiler is one of the main concerns of microcontroller programmers, for two reasons:

1. Microcontrollers have limited on-chip ROM.
2. The code space for the 8051 is limited to 64K bytes.

How does the choice of programming language affect the compiled program size? While Assembly language produces a hex file that is much smaller than C, programming in Assembly language is tedious and time consuming. C programming, on the other hand, is less time consuming and much easier to write, but the hex file size produced is much larger than if we used Assembly language. The following are some of the major reasons for writing programs in C instead of Assembly:

1. It is easier and less time consuming to write in C than Assembly.
2. C is easier to modify and update.
3. You can use code available in function libraries.
4. C code is portable to other microcontrollers with little or no modification.

Keywords in Embedded C

A Keyword is a special word with a special meaning to the compiler (a C Compiler for example, is a software that is used to convert program written in C to Machine Code). For example, if we take the Keil's Cx51 Compiler (a popular C Compiler for 8051 based Microcontrollers) the following are some of the keywords:

- bit
- sbit
- sfr
- small
- large

The following table lists out all the keywords associated with the Cx51 C Compiler.

_at_	alien	bdata
bit	code	compact
data	far	idata
interrupt	large	pdata
priority	reentrant	sbit

sfr	sfr16	small
task	using	xdata

SECTION 7.1: DATA TYPES AND TIME DELAY IN 8051 C

In this section we first discuss C data types for the 8051 and then provide code for time delay functions.

C data types for the 8051

Since one of the goals of 8051 C programmers is to create smaller hex files, it is worthwhile to re-examine C data types for 8051 C. In other words, a good understanding of C data types for the 8051 can help programmers to create smaller hex files. In this section we focus on the specific C data types that are most useful and widely used for the 8051 microcontroller.

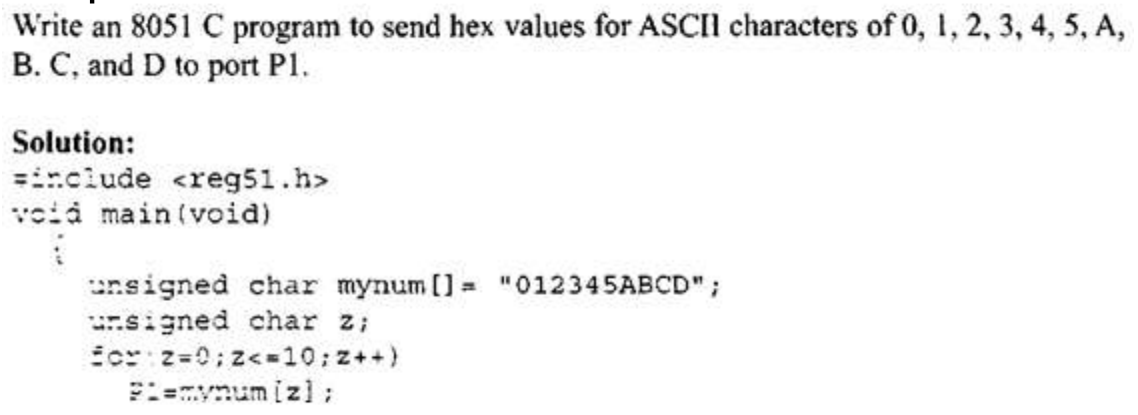
Unsigned char

Since the 8051 is an 8-bit microcontroller, the character data type is the most natural choice for many applications. The unsigned char is an 8-bit data type that takes a value in the range of 0 – 255 (00 – FFH). It is one of the most widely used data types for the 8051. In many situations, such as setting a counter value.

where there is no need for signed data we should use the unsigned char instead of the signed char. Remember that C compilers use the signed char as the default if we do not put the keyword *unsigned* in front of the char (see Example 7-1). We can also use the unsigned char data type for a string of ASCII characters, including extended ASCII characters. Example 7-2 shows a string of ASCII characters. See Example 7-3 for toggling ports.

In declaring variables, we must pay careful attention to the size of the data and try to use unsigned char instead of int if possible. Because the 8051 has a limited number of registers and data RAM locations, using the int in place of the char data type can lead to a larger size hex file. Such a misuse of the data types in compilers such as Microsoft Visual C++ for x86 IBM PCs is not a significant issue.

Example 7-2



Write an 8051 C program to send hex values for ASCII characters of 0, 1, 2, 3, 4, 5, A, B, C, and D to port P1.

Solution:

```
#include <reg51.h>
void main(void)
  {
    unsigned char mynum[]= "012345ABCD";
    unsigned char z;
    for(z=0;z<=10;z++)
      P1=mynum[z];
```

Example 7-1

Write an 8051 C program to send values 00 - FF to port P1.

Solution:

```
#include <reg51.h>
void main(void)
  {
    unsigned char z;
    for(z=0;z<=255;z++)
      P1=z;
  }
```

Run the above program on your simulator to see how P1 displays values 00 - FFH in binary.

Run the above program on your simulator to see how PI displays values 30H, 31H, 32H. 33H. 34H. 35H. 41H. 42H, 43H, and 44H, the hex values for ASCII 0, 1, 2, and so on.

Example 7-3

Write an 8051 C program to toggle all the bits of PI continuously. **Solution:**

```
// Toggle PI forever ^include <reg51.h> void main(void)
  {
    for(;;)         //repeat forever
     {
       P1=0x55;    //0x indicates the data is in hex (binary)
       P1=0xAA;
     }
  }
```

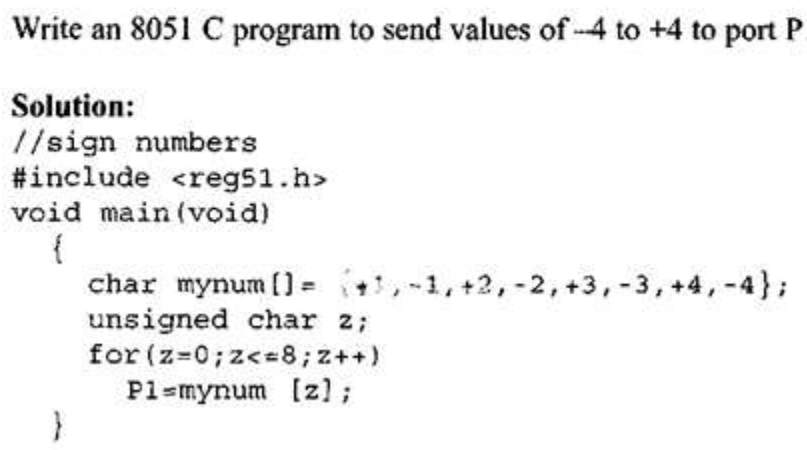
Run the above program on your simulator to see how PI toggles continuously. Examine the asm code generated by the C compiler.

Signed char

The signed char is an 8-bit data type that uses the most significant bit (D7 of D7 – DO) to represent the – or + value. As a result, we have only 7 bits for the magnitude of the signed number, giving us values from -128 to +127. In situations where + and – are needed to represent a given quantity such as temperature, the use of the signed char data type is a must.

Again notice that if we do not use the keyword *unsigned,* the default is the signed value. For that reason we should stick with the unsigned char unless the data needs to be represented as signed numbers.

Example 7-4


Write an 8051 C program to send values of –4 to +4 to port P1.

Solution:

```
//sign numbers
#include <reg51.h>
void main(void)
  {
    char mynum[]= {+1,-1,+2,-2,+3,-3,+4,-4};
    unsigned char z;
    for(z=0;z<=8;z++)
      P1=mynum [z];
  }
```

Run the above program on your simulator to see how PI displays values of 1, FFH, 2, FEH, 3, FDH, 4, and FCH, the hex values for +!,-!, +2, -2, and so on.

Unsigned int

The unsigned int is a 16-bit data type that takes a value in the range of 0 to 65535 (0000 – FFFFH). In the 8051, unsigned int is used to define 16-bit variables such as memory addresses. It is also used to set counter values of more than 256. Since the 8051 is an 8-bit microcontroller and the int data type takes two bytes of RAM, we must not use the int data type unless we have to. Since registers and memory accesses are in 8-bit chunks, the misuse of int variables will result in a larger hex file. Such misuse is not a big deal in PCs with 256 megabytes of memory, 32-bit Pentium registers and memory accesses, and a bus speed of 133 MHz. However, for 8051 programming do not use unsigned int in places where unsigned char will do the job. Of course the compiler will not generate an error for this misuse, but the overhead in hex file size is noticeable. Also in situations where there is no need for signed data (such as setting counter values), we should use unsigned int instead of signed int. This gives a much wider range for data declaration. Again, remember that the C compiler uses signed int as the default if we do not use the keyword *unsigned.*

Signed int

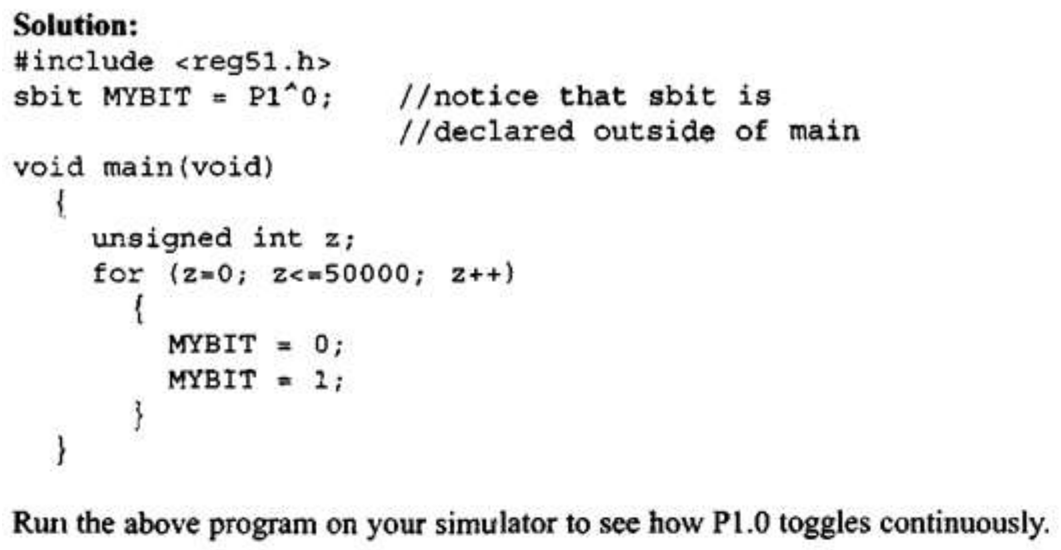
Signed int is a 16-bit data type that uses the most significant bit (015 of D15 – DO) to represent the – or + value. As a result, we have only 15 bits for the magnitude of the number, or values from -32,768 to +32,767.

Sbit (single bit)

The sbit keyword is a widely used 8051 C data type designed specifically to access single-bit addressable registers. It allows access to the single bits of the SFR registers. As we saw in Chapter 5, some of the SFRs are bit^addressable. Among the SFRs that are widely used and are also bit-addressable are ports PO -P3. We can use sbit to access the individual bits of the ports as shown in Example 7-5.

Example 7-5

Write an 8051 C program to toggle bit DO of the port PI (Pl.O) 50,000 times.


Solution:

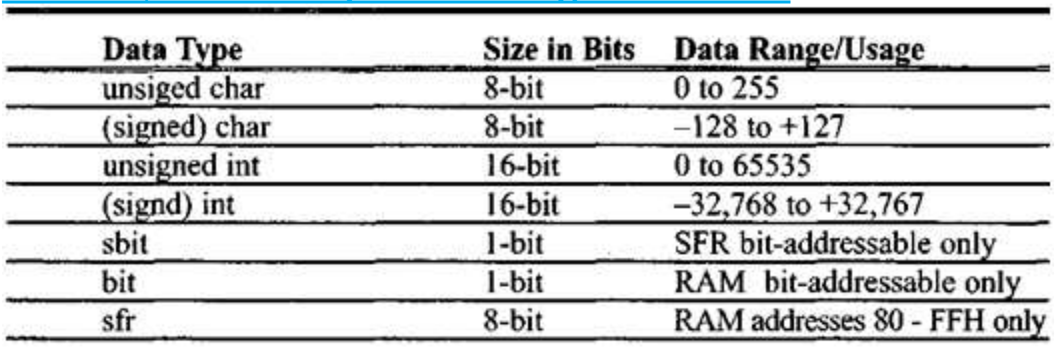
```
#include <reg51.h>
sbit MYBIT = P1^0;      //notice that sbit is
                        //declared outside of main
void main(void)
  {
    unsigned int z;
    for (z=0; z<=50000; z++)
      {
        MYBIT = 0;
        MYBIT = 1;
      }
  }
```

Run the above program on your simulator to see how P1.0 toggles continuously.

Bit and sfr

The bit data type allows access to single bits of bit-addressable memory spaces 20 – 2FH. Notice that while the sbit data type is used for bit-addressable SFRs, the bit data type is used for the bit-addressable section of RAM space 20 -2FH. To access the byte-size SFR registers, we use the sfr data type. We will see the use of sbit, bit, and sfr data types in the next section.

Table 7-1; Some Widely Used Data Types for 8051 C



Data Type	Size in Bits	Data Range/Usage
unsiged char	8-bit	0 to 255
(signed) char	8-bit	–128 to +127
unsigned int	16-bit	0 to 65535
(signd) int	16-bit	–32,768 to +32,767
sbit	1-bit	SFR bit-addressable only
bit	1-bit	RAM bit-addressable only
sfr	8-bit	RAM addresses 80 - FFH only

Time Delay

There are two ways to create a time delay in 8051 C:

1. Using a simple for loop
2. Using the 8051 timers

In either case, when we write a time delay we must use the oscilloscope to measure the duration of our time delay. Next, we use the for loop to create time delays. Discussion of the use of the 8051 timer to create time delays is postponed until Chapter 9.

In creating a time delay using a for loop, we must be mindful of three factors that can affect the accuracy of the delay.

1. The 8051 design. Since the original 8051 was designed in 1980, both the fields of 1C technology and microprocessor architectural design have seen great advancements. As we saw in Chapter 3, the number of machine cycles and the number of clock periods per machine cycle vary among different versions of the 8051/52 microcontroller. While the original 8051/52 design used 12 clock

periods per machine cycle, many of the newer generations of the 8051 use fewer clocks per machine cycle. For example, the DS5000 uses 4 clock peri ods per machine cycle, while the DS89C420 uses only one clock per machine cycle.

2. The crystal frequency connected to the XI – X2 input pins. The duration of the clock period for the machine cycle is a function of this crystal frequency.
3. Compiler choice. The third factor that affects the time delay is the compiler used to compile the C program. When we program in Assembly language, we can control the exact instructions and their sequences used in the delay sub routine. In the case of C programs, it is the C compiler that converts the C statements and functions to Assembly language instructions. As a result, dif ferent compilers produce different code. In other words, if we compile a given 8051 C programs with different compilers, each compiler produces different hex code.

For the above reasons, when we write time delays for C, we must use the oscilloscope to measure the exact duration. Look at Examples 7-6 through 7-8.

Example 7-6

Write an 8051 C program to toggle bits of PI continuously forever with some delay. **Solution:**

```
// Toggle PI forever with some delay in between “on” and “off”,
^include <reg51.h>
void main(void)
  {
    unsigned int x;
    for(;;)                    //repeat forever
      {
        P1=0x55;
        for(x=0;x<40000;x++);  //delay size unknown
        P1=0xAA;
        for(x=0;x<40000;x++);
      }
  }
```

Example 7-7

Write an 8051 C program to toggle the bits of PI ports continuously with a 250 ms delay.

Solution:

The program below is tested for the DS89C420 with XTAL = 11.0592 MHz.

```
#include <reg51.h>
void MSDelay(unsigned int);
void main(void)
  {
    while(1)    //repeat forever
      {
        P1=0x55;
        MSDelay(250);
        P1=0xAA;
        MSDelay(250);
      }
  }

void MSDelay(unsigned int itime)
  {
    unsigned int i, j;
    for(i=0;i<itime;i++)
      for(j=0;j<1275;j++);
  }
```

Run the above program on your Trainer and use the oscilloscope to measure the delay.

Example 7-8

Write a 8051 C program to toggle all the bits of P0 and P2 continuously with a 250 ms delay.

Solution:

```
//This program is tested for the DS89C420 with XTAL = 11.0592 MHz
#include <reg51.h>
void MSDelay(unsigned int);
void main(void)
  {
    while(1)    //another way to do it forever
      {
        P0=0x55;
        P2=0x55;
        MSDelay(250);
        P0=0xAA;
        P2=0xAA;
        MSDelay(250);
      }
  }
void MSDelay(unsigned int itime)
  {
    unsigned int i, j;
    for(i=0;i<itime;i++)
      for(j=0;j<1275;j++);
  }
```

Basic Structure of an Embedded C Program (Template for Embedded C Program)

The next thing to understand in the Basics of Embedded C Program is the basic structure or Template of Embedded C Program. This will help us in understanding how an Embedded C Program is written.

The following part shows the basic structure of an Embedded C Program.

-
  - Multiline Comments . . . . . Denoted using /*……*/
  - Single Line Comments . . . . . Denoted using //
  - Preprocessor Directives . . . . . #include<…> or #define
  - Global Variables . . . . . Accessible anywhere in the program
  - Function Declarations . . . . . Declaring Function
  - Main Function . . . . . Main Function, execution begins here
    {
    Local Variables . . . . . Variables confined to main function
    Function Calls . . . . . Calling other Functions
    Infinite Loop . . . . . Like while(1) or for(;;)
    Statements . . . . .
    ….
    ….
    }
  - Function Definitions . . . . . Defining the Functions
    {
    Local Variables . . . . . Local Variables confined to this Function
    Statements . . . . .
    ….
    ….
    }

Before seeing an example with respect to 8051 Microcontroller, we will first see the different components in the above structure.

Different Components of an Embedded C Program

Comments: Comments are readable text that are written to help us (the reader) understand the code easily. They are ignored by the compiler and do not take up any memory in the final code (after compilation).

There are two ways you can write comments: one is the single line comments denoted by // and the other is multiline comments denoted by /*….*/.

Preprocessor Directive: A Preprocessor Directive in Embedded C is an indication to the compiler that it must look in to this file for symbols that are not defined in the program.

In C Programming Language (also in Embedded C), Preprocessor Directives are usually represented using # symbol like #include… or #define….

In Embedded C Programming, we usually use the preprocessor directive to indicate a header file specific to the microcontroller, which contains all the SFRs and the bits in those SFRs.

In case of 8051, Keil Compiler has the file "reg51.h", which must be written at the beginning of every Embedded C Program.

Global Variables: Global Variables, as the name suggests, are Global to the program i.e., they can be accessed anywhere in the program.

Local Variables: Local Variables, in contrast to Global Variables, are confined to their respective function.

Main Function: Every C or Embedded C Program has one main function, from where the execution of the program begins.

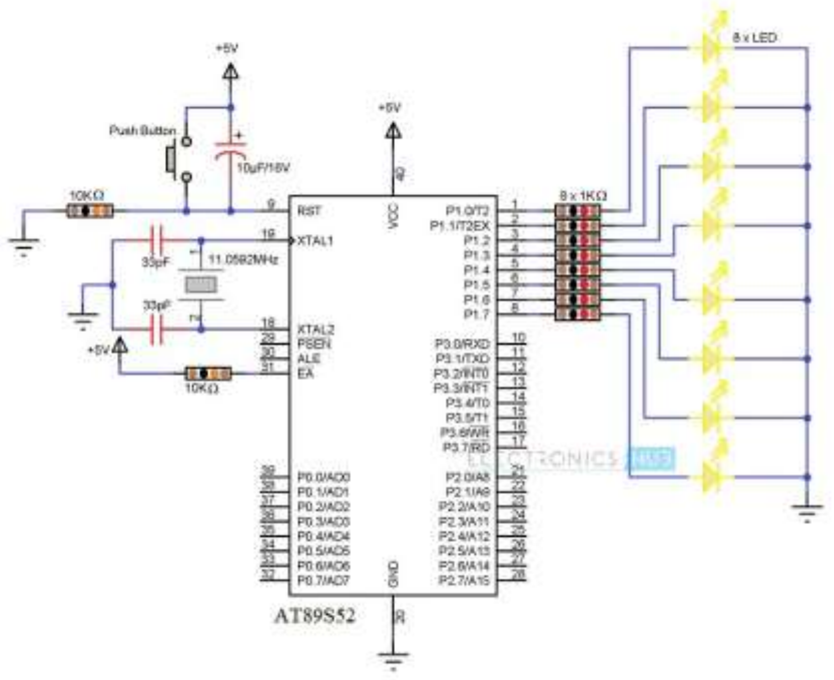
Basic Embedded C Program

Till now, we have seen a few Basics of Embedded C Program like difference between C and Embedded C, basic structure or template of an Embedded C Program and different components of the Embedded C Program.

Continuing further, we will explore in to basics of Embedded C Program with the help of an example. In this example, we will use an 8051 Microcontroller to blink LEDs connected to PORT1 of the microcontroller.

Example of Embedded C Program

The following image shows the circuit diagram for the example circuit. It contains an 8051 based Microcontroller (AT89S52) along with its basic components (like RESET Circuit, Oscillator Circuit, etc.) and components for blinking LEDs (LEDs and Resistors).



In order to write the Embedded C Program for the above circuit, we will use the Keil C Compiler. This compiler is a part of the Keil µVision IDE. The program is shown below.

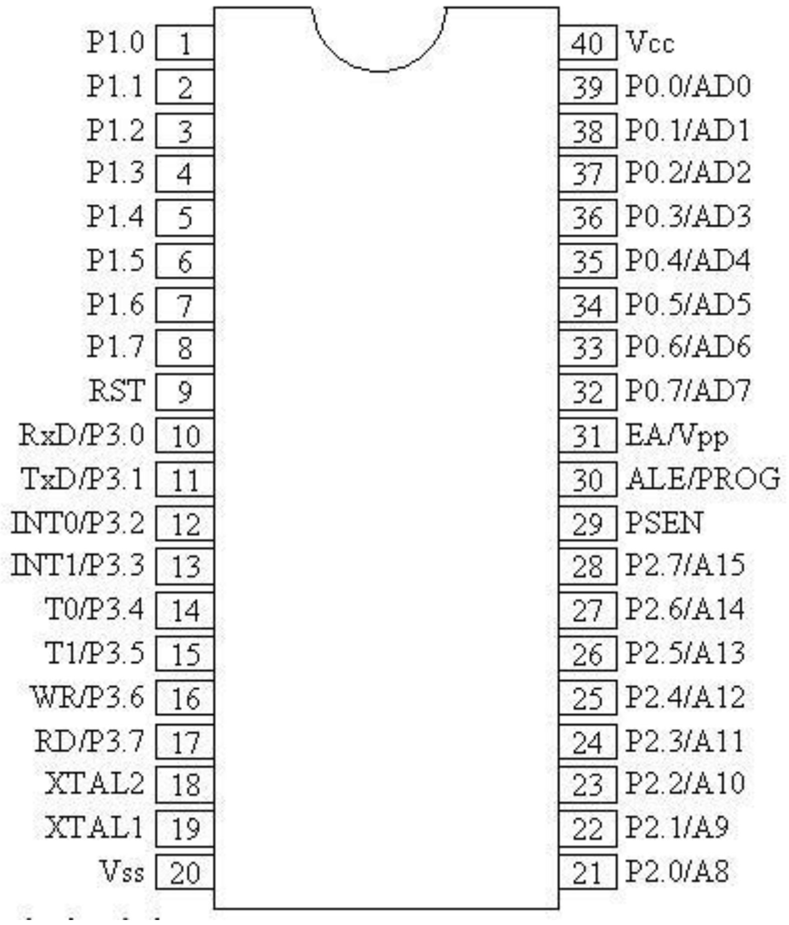
Sample Code

```
#include<reg51.h> // Preprocessor Directive
void delay (int); // Delay Function Declaration
void main(void) // Main Function
{
P1 = 0x00;
/* Making PORT1 pins LOW. All the LEDs are OFF.
 * (P1 is PORT1, as defined in reg51.h) */


while(1) // infinite loop
{
P1 = 0xFF; // Making PORT1 Pins HIGH i.e. LEDs are ON.
delay(1000);
/* Calling Delay function with Function parameter as 1000.
 * This will cause a delay of 1000mS i.e. 1 second */


P1 = 0x00; // Making PORT1 Pins LOW i.e. LEDs are OFF.
delay(1000);
}
}


void delay (int d) // Delay Function Definition
{
unsigned int i=0; // Local Variable. Accessible only in this function.

/* This following step is responsible for causing delay of 1000mS
 * (or as per the value entered while calling the delay function) */


for(; d>0; d–)
{
for(i=250; i>0; i – -);
for(i=248; i>0; i – -);
}
}
```

I/O PORT PROGRAMMING

In 8051, I/O operations are done using four ports and 40 pins. The following pin diagram shows the details of the 40 pins. I/O operation port reserves 32 pins where each port has 8 pins. The other 8 pins are designated as V_{cc}, GND, XTAL1, XTAL2, RST, EA (bar), ALE/PROG (bar), and PSEN (bar).

It is a 40 Pin PDIP (Plastic Dual Inline Package)



Note − In a DIP package, you can recognize the first pin and the last pin by the cut at the middle of the IC. The first pin is on the left of this cut mark and the last pin (i.e. the 40th pin in this case) is to the right of the cut mark.

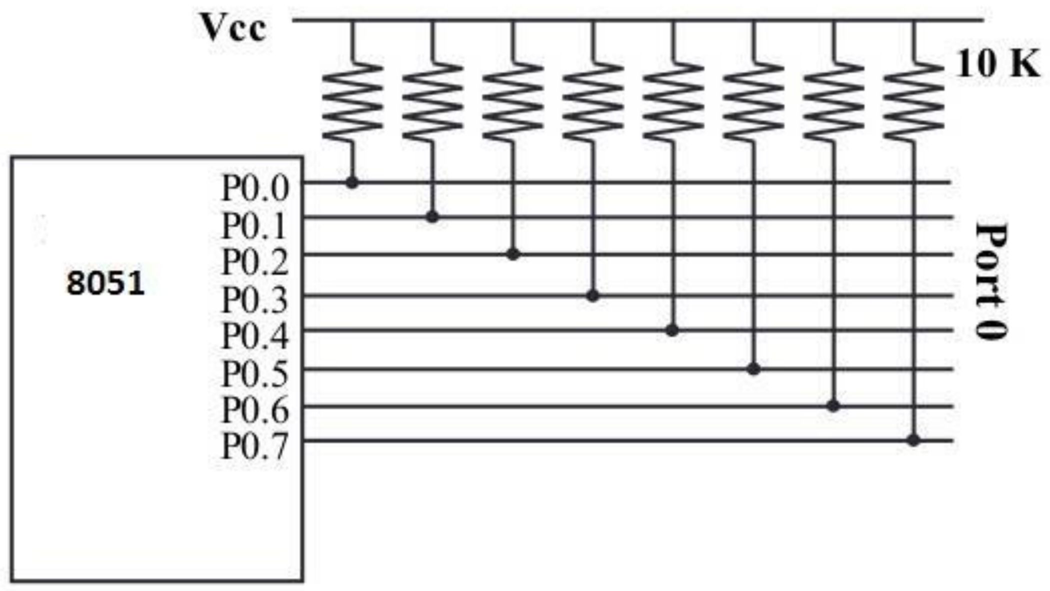
I/O Ports and their Functions

The four ports P0, P1, P2, and P3, each use 8 pins, making them 8-bit ports. Upon RESET, all the ports are configured as inputs, ready to be used as input ports. When the first 0 is written to a port, it becomes an output. To reconfigure it as an input, a 1 must be sent to a port.

Port 0 (Pin No 32 – Pin No 39)

It has 8 pins (32 to 39). It can be used for input or output. Unlike P1, P2, and P3 ports, we normally connect P0 to 10K-ohm pull-up resistors to use it as an input or output port being an open drain.

It is also designated as AD0-AD7, allowing it to be used as both address and data. In case of 8031 (i.e. ROMless Chip), when we need to access the external ROM, then P0 will be used for both Address and Data Bus. ALE (Pin no 31) indicates if P0 has address or data. When ALE = 0, it provides data D0-D7, but when ALE = 1, it has address A0-A7. In case no external memory connection is available, P0 must be connected externally to a 10K-ohm pull-up resistor.



```
MOV A,#0FFH   ;(comments: A=FFH(Hexadecimal  i.e. A=1111 1111)

MOV P0,A      ;(Port0 have 1's on every pin so that it works as
Input)
```

Port 1 (Pin 1 through 8)

It is an 8-bit port (pin 1 through 8) and can be used either as input or output. It doesn't require pull-up resistors because they are already connected internally. Upon reset, Port 1 is configured as an input port. The following code can be used to send alternating values of 55H and AAH to Port 1.

```
;Toggle all bits of continuously
MOV     A,#55
BACK:

MOV     P2,A
ACALL   DELAY
CPL     A      ;complement(invert) reg. A
SJMP    BACK
```

If Port 1 is configured to be used as an output port, then to use it as an input port again, program it by writing 1 to all of its bits as in the following code.

```
;Toggle all bits of continuously

MOV     A ,#0FFH    ;A = FF hex
MOV      P1,A         ;Make P1 an input port
MOV      A,P1         ;get data from P1
MOV      R7,A         ;save it in Reg R7
ACALL    DELAY        ;wait

MOV      A,P1         ;get another data from P1
MOV      R6,A         ;save it in R6
ACALL    DELAY        ;wait

MOV      A,P1         ;get another data from P1
MOV      R5,A         ;save it in R5
```

Port 2 (Pins 21 through 28)

Port 2 occupies a total of 8 pins (pins 21 through 28) and can be used for both input and output operations. Just as P1 (Port 1), P2 also doesn't require external Pull-up resistors because they are already connected internally. It must be used along with P0 to provide the 16-bit address for the external memory. So it is also designated as (A0–A7), as shown in the pin diagram. When the 8051 is connected to an external memory, it provides path for upper 8-bits of 16-bits address, and it cannot be used as I/O. Upon reset, Port 2 is configured as an input port. The following code can be used to send alternating values of 55H and AAH to port 2.

```
;Toggle all bits of continuously
MOV      A,#55
BACK:
MOV      P2,A
ACALL    DELAY
CPL      A          ; complement(invert) reg. A
SJMP     BACK
```

If Port 2 is configured to be used as an output port, then to use it as an input port again, program it by writing 1 to all of its bits as in the following code.

```
;Get a byte from P2 and send it to P1
MOV     A,#0FFH     ;A = FF hex
MOV     P2,A        ;make P2 an input port
BACK:
MOV     A,P2        ;get data from P2
MOV     P1,A        ;send it to Port 1
SJMP    BACK        ;keep doing that
```

Port 3 (Pins 10 through 17)

It is also of 8 bits and can be used as Input/Output. This port provides some extremely important signals. P3.0 and P3.1 are RxD (Receiver) and TxD (Transmitter) respectively and are collectively used for Serial Communication. P3.2 and P3.3 pins are used for external interrupts. P3.4 and P3.5 are used for timers T0 and T1 respectively. P3.6 and P3.7 are Write (WR) and Read (RD) pins. These are active low pins, means they will be active when 0 is given to them and these are used to provide Read and Write operations to External ROM in 8031 based systems.

P3 Bit	Function	Pin
P3.0	RxD	10
P3.1 <	TxD	11
P3.2 <	Complement of INT0	12
P3.3 <	INT1	13
P3.4 <	T0	14
P3.5 <	T1	15
P3.6 <	WR	16
P3.7 <	Complement of RD	17

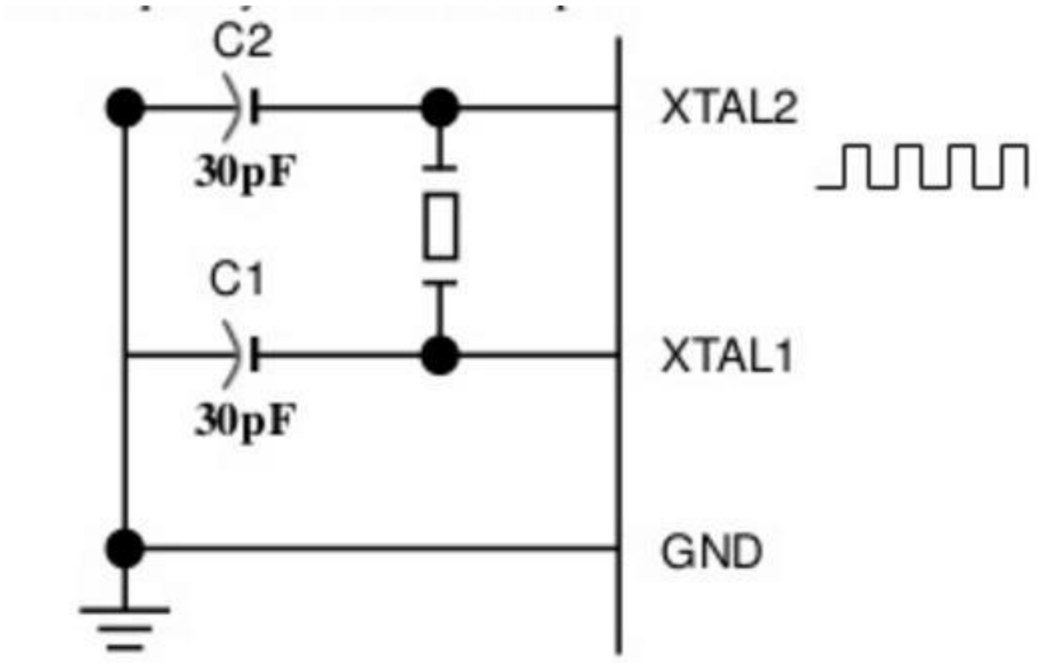
Dual Role of Port 0 and Port 2

- **Dual role of Port 0 −** Port 0 is also designated as AD0–AD7, as it can be used for both data and address handling. While connecting an 8051 to external memory, Port 0 can provide both address and data. The 8051 microcontroller then multiplexes the input as address or data in order to save pins.
- **Dual role of Port 2 −** Besides working as I/O, Port P2 is also used to provide 16-bit address bus for external memory along with Port 0. Port P2 is also designated as (A8– A15), while Port 0 provides the lower 8-bits via A0–A7. In other words, we can say that when an 8051 is connected to an external memory (ROM) which can be maximum up to 64KB and this is possible by 16 bit address bus because we know 216 = 64KB. Port2 is used for the upper 8-bit of the 16 bits address, and it cannot be used for I/O and this is the way any Program code of external ROM is addressed.

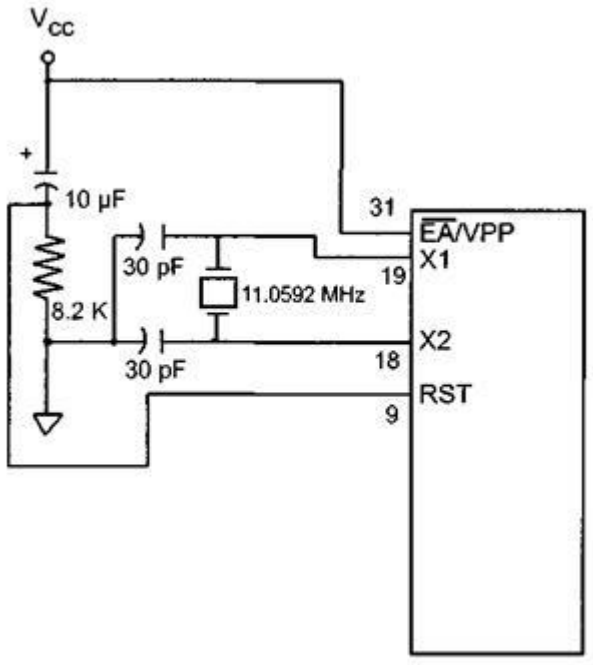
Hardware Connection of Pins

- **V_{cc} −** Pin 40 provides supply to the Chip and it is +5 V.
- **Gnd −** Pin 20 provides ground for the Reference.

- **XTAL1, XTAL2 (Pin no 18 & Pin no 19) −** 8051 has on-chip oscillator but requires external clock to run it. A quartz crystal is connected between the XTAL1 & XTAL2 pin of the chip. This crystal also needs two capacitors of 30pF for generating a signal of desired frequency. One side of each capacitor is connected to ground. 8051 IC is available in various speeds and it all depends on this Quartz crystal, for example, a 20 MHz microcontroller requires a crystal with a frequency no more than 20 MHz.



- **RST (Pin No. 9) −** It is an Input pin and active High pin. Upon applying a high pulse on this pin, that is 1, the microcontroller will reset and terminate all activities. This process is known as **Power-On Reset**. Activating a power-on reset will cause all values in the register to be lost. It will set a program counter to all 0's. To ensure a valid input of Reset, the high pulse must be high for a minimum of two machine cycles before it is allowed to go low, which depends on the capacitor value and the rate at which it charges. (**Machine Cycle** is the minimum amount of frequency a single instruction requires in execution).
- **EA or External Access (Pin No. 31) −** It is an input pin. This pin is an active low pin; upon applying a low pulse, it gets activated. In case of microcontroller (8051/52) having on-chip ROM, the EA (bar) pin is connected to V_{cc}. But in an 8031 microcontroller which does not have an on-chip ROM, the code is stored in an external ROM and then fetched by the microcontroller. In this case, we must connect the (pin no 31) EA to Gnd to indicate that the program code is stored externally.



- **PSEN or Program store Enable (Pin No 29) −** This is also an active low pin, i.e., it gets activated after applying a low pulse. It is an output pin and used along with the EA pin in 8031 based (i.e. ROMLESS) Systems to allow storage of program code in external ROM.
- **ALE or (Address Latch Enable) −** This is an Output Pin and is active high. It is especially used for 8031 IC to connect it to the external memory. It can be used while deciding whether P0 pins will be used as Address bus or Data bus. When ALE = 1, then the P0 pins work as Data bus and when ALE = 0, then the P0 pins act as Address bus.

I/O Ports and Bit Addressability

It is a most widely used feature of 8051 while writing code for 8051. Sometimes we need to access only 1 or 2 bits of the port instead of the entire 8-bits. 8051 provides the capability to access individual bits of the ports.

While accessing a port in a single-bit manner, we use the syntax "SETB X. Y" where X is the port number (0 to 3), and Y is a bit number (0 to 7) for data bits D0-D7 where D0 is the LSB and D7 is the MSB. For example, "SETB P1.5" sets high bit 5 of port 1.

The following code shows how we can toggle the bit P1.2 continuously.

```
AGAIN:
SETB    P1.2
ACALL   DELAY
CLR     P1.2
ACALL   DELAY
SJMP    AGAIN
```

Single-Bit Instructions

Instructions	Function
SETB bit	Set the bit (bit = 1)
CLR bit	clear the bit (bit = 0)
CPL bit	complement the bit (bit = NOT bit)
JB bit, target	jump to target if bit = 1 (jump if bit)
JNB bit, target	jump to target if bit = 0 (jump if no bit)
JBC bit, target	jump to target if bit = 1, clear bit (jump if bit, then clear)

SERIAL COMMUNICATION PROGRAMMING

Transmitting and receiving data in 8051 C

SFR registers of the 8051 are accessible directly in 8051 C compilers by using the reg SI.h file. Examples 10-15 through 10-19 show how to program the serial port in 8051 C. Connect your 8051 Trainer to the PC’s COM port and use HyperTerminal to test the operation of these examples.

Example 10-15

Write a C program for the 8051 to transfer the letter "A" serially at 4800 baud continuously. Use 8-bit data and 1 stop bit.

Solution:

```
#include <reg51.h>
void main(void)
  {
    TMOD=0x20;              //use Timer 1,8-BIT auto-reload
    TH1=0xFA;               //4800 baud rate
    SCON=0x50;
    TR1=1;
    while(1)
      {
        SBUF='A';           //place value in buffer
        while(TI==0);
        TI=0;
      }
  }
```

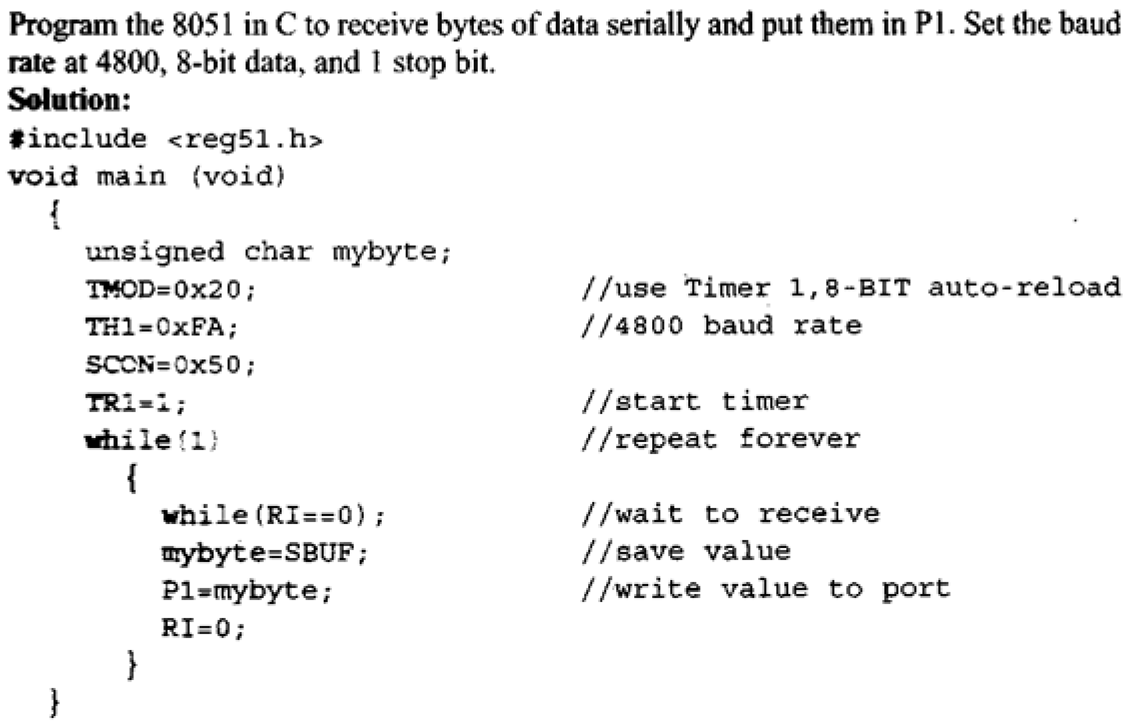
Example 10-16

Write an 8051 C program to transfer the message "YES" serially at 9600 baud, 8-bit data, 1 stop bit. Do this continuously.

Solution:

```
#include <reg51.h>
void SerTx(unsigned char);
void main(void)
  {
    TMOD=0x20;          //use Timer 1,8-BIT auto-reload
    TH1=0xFD;           //9600 baud rate
    SCON=0x50;
    TR1=1;              //start timer
    while(1)
      {
        SerTx('Y');
        SerTx('E');
        SerTx('S');
      }
  }
void SerTx(unsigned char x)
  {
    SBUF=x;             //place value in buffer
    while(TI==0);       //wait until transmitted
    TI=0;
  }
```

Example 10-17



Program the 8051 in C to receive bytes of data serially and put them in P1. Set the baud rate at 4800, 8-bit data, and 1 stop bit.

Solution:

```
#include <reg51.h>
void main (void)
  {
    unsigned char mybyte;
    TMOD=0x20;                  //use Timer 1,8-BIT auto-reload
    TH1=0xFA;                   //4800 baud rate
    SCON=0x50;
    TR1=1;                      //start timer
    while(1)                    //repeat forever
      {
        while(RI==0);           //wait to receive
        mybyte=SBUF;            //save value
        P1=mybyte;              //write value to port
        RI=0;
      }
  }
```

Example 10-18

Write an 8051 C program to send two different strings to the serial port. Assuming that

SW is connected to pin P2.0, monitor its status and make a decision as follows:

SW = 0: send your first name

SW = 1: send your last name

Assume XTAL = 11.0592 MHz, baud rate of 9600, 8-bit data, 1 stop bit.

Solution:

```
#include <reg51.h>
sbit MYSW=P2^0;                    //input switch
void main(void)
  {
    unsigned char z;
    unsigned char fname[]="ALI";
    unsigned char lname[]="SMITH";
    TMOD=0x20;                     //use Timer 1,8-BIT auto-reload
    TH1=0xFD;                      //9600 baud rate
    SCON=0x50;
    TR1=1;                         //start timer
    if(MYSW==0)                    //check switch
      {
        for(z=0;z<3;z++)           //write name
          {
            SBUF=fname[z];         //place value in buffer
            while(TI==0);          //wait for transmit
            TI=0;
          }
      }
    else
      {
        for(z=0;z<5;z++)           //write name
          {
            SBUF=lname[z];         //place value in buffer
            while(TI==0);          //wait for transmit
            TI=0;
          }
      }
  }
```

Example 10-19

Write an 8051 C program to send the two messages “Normal Speed” and “High Speed” to the serial port. Assuming that SW is connected to pin P2.0, monitor its status and set the baud rate as follows:

SW = 0 28,800 baud rate

SW = 1 56K baud rate

Assume that XTAL = 11.0592 MHz for both cases.

Solution:

```
#include <reg51.h>
sbit MYSW=P2^0;                    //input switch
void main(void)
  {
    unsigned char z;
    unsigned char Mess1[]="Normal Speed";
    unsigned char Mess2[]="High Speed";
    TMOD=0x20;                     //use Timer 1,8-BIT auto-reload
    TH1=0xFF;                      //28,800 for normal speed
    SCON=0x50;
    TR1=1;                         //start timer
    if(MYSW==0)
      {
        for(z=0;z<12;z++)
          {
            SBUF=Mess1[z];         //place value in buffer
            while(TI==0);          //wait for transmit
            TI=0;
          }
      }
    else
      {
        PCON=PCON|0x80;            //for high speed of 56K
        for(z=0;z<10;z++)
          {
            SBUF=Mess2[z];         //place value in buffer
            while(TI==0);          //wait for transmit
            TI=0;
          }
      }
  }
```

8051 C compilers and the second serial port

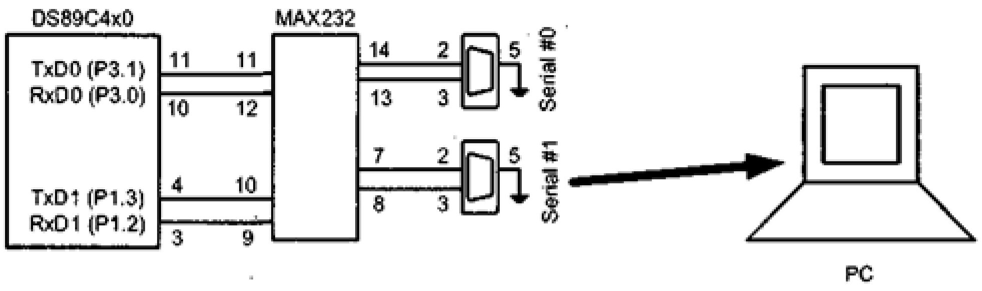
Since many C compilers do not support the second serial port of the DS89C4xO chip, we have to declare the byte addresses of the new SFR registers using the sfr keyword. Table 10-6 and Figure 10-12 provide the SFR byte and bit addresses for the DS89C4xO chip. Examples 10-20 and 10-21 show C versions of Examples 10-11 and 10-13 in Section 10.4.

Notice in both Examples 10-20 and 10-21 that we are using Tinier 1 to set the baud rate for the second serial port. Upon reset, Timer 1 is the default for the second serial port of the DS89C4xO chip.

Example 10-20

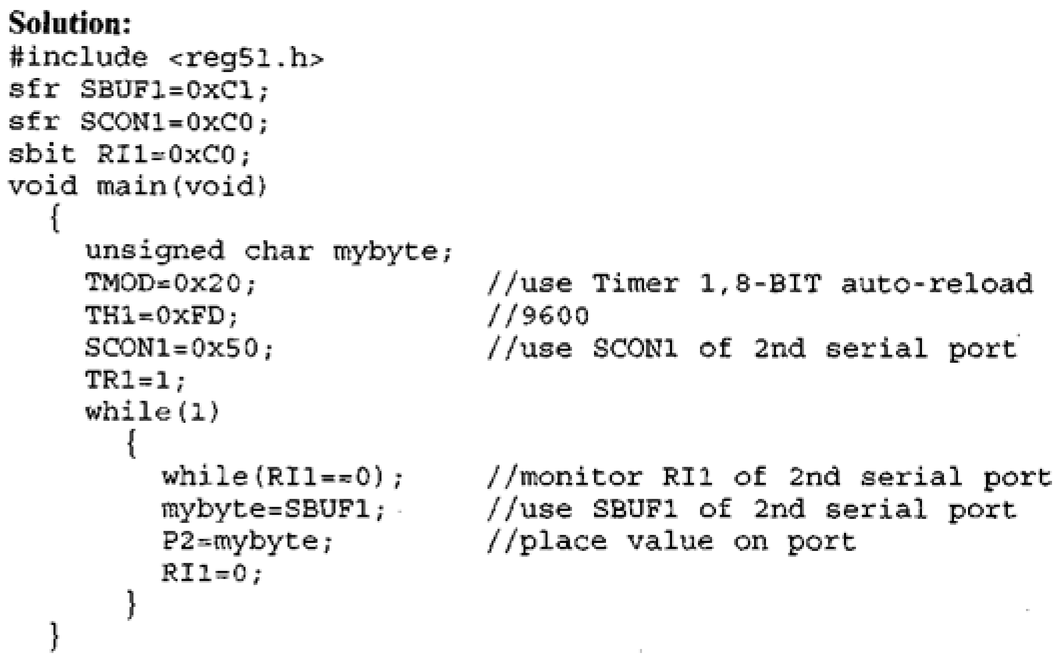
Write a C program for the DS89C4xO to transfer letter "A" serially at 4800 baud continuously. Use the second serial port with 8-bit data and 1 stop bit. We can only use Timer 1 to set the baud rate. **Solution:**

```
#include <reg51.h>
sfr SBUF1=0xC1;
sfr SCON1=0xC0;
sbit TI1=0xC1;
void main(void)
  {
    TMOD=0x20;          //use Timer 1 for 2nd serial port
    TH1=0xFA;           //4800 baud rate
    SCON1=0x50;         //use 2nd serial port SCON1 register
    TR1=1;              //start timer
    while(1)
      {
        SBUF1='A';      //use 2nd serial port SBUF1 register
        while(TI1==0);  //wait for transmit
        TI1=0;
      }
  }
```



Example 10-21

Program the DS89C4xO in C to receive bytes of data serially via the second serial port and put them in PI. Set the baud rate at 9600, 8-bit data, and 1 stop bit. Use Timer 1 for baud rate generation.


Solution:

```
#include <reg51.h>
sfr SBUF1=0xC1;
sfr SCON1=0xC0;
sbit RI1=0xC0;
void main(void)
  {
    unsigned char mybyte;
    TMOD=0x20;              //use Timer 1,8-BIT auto-reload
    TH1=0xFD;               //9600
    SCON1=0x50;             //use SCON1 of 2nd serial port
    TR1=1;
    while(1)
      {
        while(RI1==0);      //monitor RI1 of 2nd serial port
        mybyte=SBUF1;       //use SBUF1 of 2nd serial port
        P2=mybyte;          //place value on port
        RI1=0;
      }
  }
```

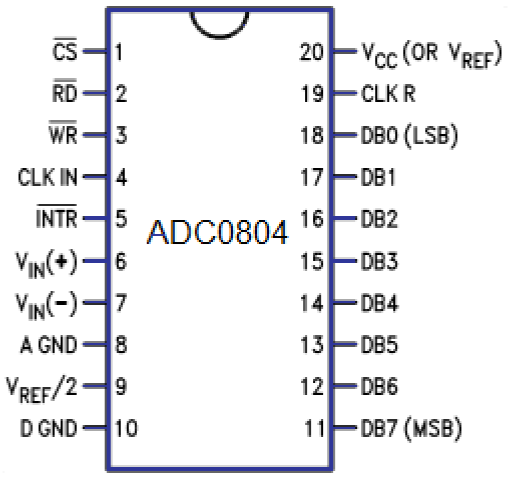
PROGRAMMING OF A/D AND D/A CONVERTERS

Interfacing ADC to 8051

ADC (Analog to digital converter) forms a very essential part in many embedded projects and this article is about interfacing an ADC to 8051 embedded controller. ADC 0804 is the ADC used here and before going through the interfacing procedure, we must neatly understand how the ADC 0804 works.

ADC 0804.

ADC0804 is an 8 bit successive approximation analogue to digital converter from National semiconductors. The features of ADC0804 are differential analogue voltage inputs, 0-5V input voltage range, no zero adjustment, built in clock generator, reference voltage can be externally adjusted to convert smaller analogue voltage span to 8 bit resolution etc. The pin out diagram of ADC0804 is shown in the figure below.



The voltage at Vref/2 (pin9) of ADC0804 can be externally adjusted to convert smaller input voltage spans to full 8 bit resolution. Vref/2 (pin9) left open means input voltage span is 0-5V and step size is 5/255=19.6V. Have a look at the table below for different Vref/2 voltages and corresponding analogue input voltage spans.

Vref/2 (pin9) (volts)	Input voltage span (volts)	Step size (mV)
Left open	0 – 5	5/255 = 19.6
2	0 – 4	4/255 = 15.69
1.5	0 – 3	3/255 = 11.76
1.28	0 – 2.56	2.56/255 = 10.04
1.0	0 – 2	2/255 = 7.84
0.5	0 – 1	1/255 = 3.92

Steps for converting the analogue input and reading the output from ADC0804.

- Make CS=0 and send a low to high pulse to WR pin to start the conversion.
- Now keep checking the INTR pin. INTR will be 1 if conversion is not finished and INTR will be 0 if conversion is finished.
- If conversion is not finished (INTR=1) , poll until it is finished.
- If conversion is finished (INTR=0), go to the next step.
- Make CS=0 and send a high to low pulse to RD pin to read the data from the ADC.

The circuit initiates the ADC to convert a given analogue input , then accepts the corresponding digital data and displays it on the LED array connected at P0. For example, if the analogue input voltage Vin is 5V then all LEDs will glow indicating 11111111 in binary which is the equivalent of 255 in decimal. AT89s51 is the microcontroller used here. Data out pins (D0 to D7) of the ADC0804 are connected to the port pins P1.0 to P1.7 respectively. LEDs D1 to D8 are connected to the port pins P0.0 to P0.7 respectively. Resistors R1 to R8 are current limiting resistors. In simple words P1 of the microcontroller is the input port and P0 is the output port. Control signals for the ADC (INTR, WR, RD and CS) are available at port pins P3.4 to P3.7 respectively. Resistor R9 and capacitor C1 are associated with the internal clock circuitry of the ADC. Preset resistor R10 forms a voltage divider which can be used to apply a particular input analogue voltage to the ADC. Push button S1, resistor R11 and capacitor C4 forms a debouncing reset mechanism. Crystal X1 and capacitors C2,C3 are associated with the clock circuitry of the microcontroller.

Program.

```
ORG 00H

MOV P1,#11111111B // initiates P1 as the input port

MAIN: CLR P3.7 // makes CS=0

      SETB P3.6 // makes RD high

      CLR P3.5 // makes WR low

      SETB P3.5 // low to high pulse to WR for starting convers
ion
```

```
WAIT: JB P3.4,WAIT // polls until INTR=0

      CLR P3.7 // ensures CS=0

      CLR P3.6 // high to low pulse to RD for reading the data
from ADC

      MOV A,P1 // moves the digital data to accumulator

      CPL A // complements the digital data (*see the notes)

      MOV P0,A // outputs the data to P0 for the LEDs

      SJMP MAIN // jumps back to the MAIN program

      END
```

Notes.

- The entire circuit can be powered from 5V DC.
- ADC 0804 has active low outputs and the instruction CPL A complements it t0 have a straight forward display. For example, if input is 5V then the output will be 11111111 and if CPL A was not used it would have been 00000000 which is rather awkward to see.

Interfacing DAC with 8051

n this section we will see how DAC (Digital to Analog Converter) using Intel 8051 Microcontroller. We will also see the sinewave generation using DAC.

The Digital to Analog converter (DAC) is a device, that is widely used for converting digital pulses to analog signals. There are two methods of converting digital signals to analog signals. These two methods are binary weighted method and R/2R ladder method. In this article we will use the MC1408 (DAC0808) Digital to Analog Converter. This chip uses R/2R ladder method. This method can achieve a much higher degree of precision. DACs are judged by its resolution. The resolution is a function of the number of binary inputs. The most common input counts are 8, 10, 12 etc. Number of data inputs decides the resolution of DAC. So if there are n digital input pin, there are 2^n analog levels. So 8 input DAC has 256 discrete voltage levels.

The MC1408 DAC (or DAC0808)

In this chip the digital inputs are converted to current. The output current is known as I_{out} by connecting a resistor to the output to convert into voltage. The total current provided by the I_{out} pin is basically a function of the binary numbers at the input pins D_0 - D_7 (D_0 is the LSB and D_7 is the MSB) of DAC0808 and the reference current I_{ref}. The following formula is showing the function of I_{out}

IOut=Iref(D72+D64+D58+D416+D332+D264+D1128+D0256)IOut=Iref(D72+D64 +D58+D416+D332+D264+D1128+D0256)

The I_{ref} is the input current. This must be provided into the pin 14. Generally 2.0mA is used as I_{ref}

We connect the I_{out} pin to the resistor to convert the current to voltage. But in real life it may cause inaccuracy since the input resistance of the load will also affect the output voltage. So practically I_{ref} current input is isolated by connecting it to an Op-Amp with R_f = 5KΩ as feedback resistor. The feedback resistor value can be changed as per requirement.

Generating Sinewave using DAC and 8051 Microcontroller

For generating sinewave, at first we need a look-up table to represent the magnitude of the sine value of angles between 0° to 360°. The sine function varies from -1 to +1. In the table only integer values are applicable for DAC input. In this example we will consider 30° increments and calculate the values from degree to DAC input. We are assuming full-scale voltage of 10V for DAC output. We can follow this formula to get the voltage ranges.

$V_{out} = 5V + (5 \times \sin\theta)$

Let us see the lookup table according to the angle and other parameters for DAC.

Angle(in θ)	**sinθ**	**V_{out} (Voltage Magnitude)**	**Values sent to DAC**
0	0	5	128
30	0.5	7.5	192
60	0.866	9.33	238
90	1.0	10	255
120	0.866	9.33	238
150	0.5	7.5	192

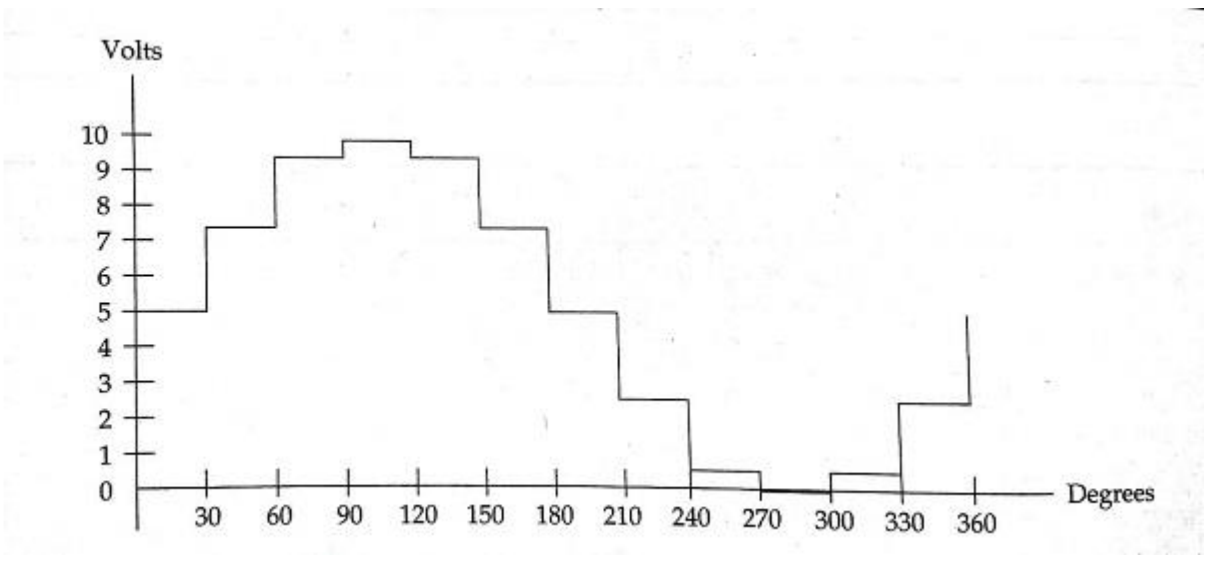
Angle(in θ)	sinθ	V_{out} (Voltage Magnitude)	Values sent to DAC
180	0	5	128
210	-0.5	2.5	64
240	-0.866	0.669	17
270	-1.0	0	0
300	-0.866	0.669	17
330	-0.5	2.5	64
360	0	5	128

Source Code

```
#include<reg51.h>
sfr DAC = 0x80; //Port P0 address
void main(){
   int sin_value[12] =
{128,192,238,255,238,192,128,64,17,0,17,64};
   int i;
   while(1){
      //infinite loop for LED blinking
      for(i = 0; i<12; i++){
         DAC = sin_value[i];
      }
   }
}
```

Output

The output will look like this −

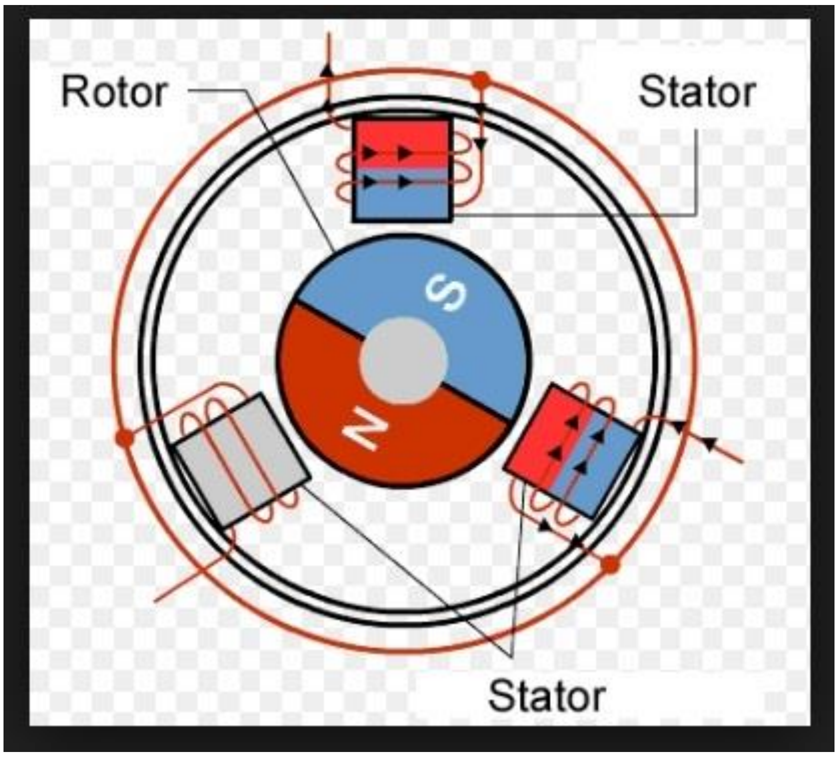


<u>Interfacing Stepper Motor with 8051Microcontroller</u>

In this section, we will see how to connect a stepper motor with Intel 8051 Microcontroller. Before discussing the interfacing techniques, we will see what are the stepper motors and how they work.

Stepper Motor

Stepper motors are used to translate electrical pulses into mechanical movements. In some disk drives, dot matrix printers, and some other different places the stepper motors are used. The main advantage of using the stepper motor is the position control. Stepper motors generally have a permanent magnet shaft (rotor), and it is surrounded by a stator.



Normal motor shafts can move freely but the stepper motor shafts move in fixed repeatable increments.

Some parameters of stepper motors −

- **Step Angle** − The step angle is the angle in which the rotor moves when one pulse is applied as an input of the stator. This parameter is used to determine the positioning of a stepper motor.
- **Steps per Revolution** − This is the number of step angles required for a complete revolution. So the formula is 360° /Step Angle.
- **Steps per Second** − This parameter is used to measure a number of steps covered in each second.
- **RPM** − The RPM is the Revolution Per Minute. It measures the frequency of rotation. By this parameter, we can measure the number of rotations in one minute.

The relation between RPM, steps per revolution, and steps per second is like below:

```
Steps per Second = rpm x steps per revolution / 60
```

Interfacing Stepper Motor with 8051 Microcontroller

Weare using Port P0 of 8051 for connecting the stepper motor. HereULN2003 is used. This is basically a high voltage, high current Darlington transistor array. Each ULN2003 has seven NPN Darlington pairs. It can provide high voltage output with common cathode clamp diodes for switching inductive loads.

The Unipolar stepper motor works in three modes.

- **Wave Drive Mode −** In this mode, one coil is energized at a time. So all four coils are energized one after another. This mode produces less torque than full step drive mode.

The following table is showing the sequence of input states in different windings.

Steps	Winding A	Winding B	Winding C	Winding D
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

- **Full Drive Mode −** In this mode, two coils are energized at the same time. This mode produces more torque. Here the power consumption is also high

The following table is showing the sequence of input states in different windings.

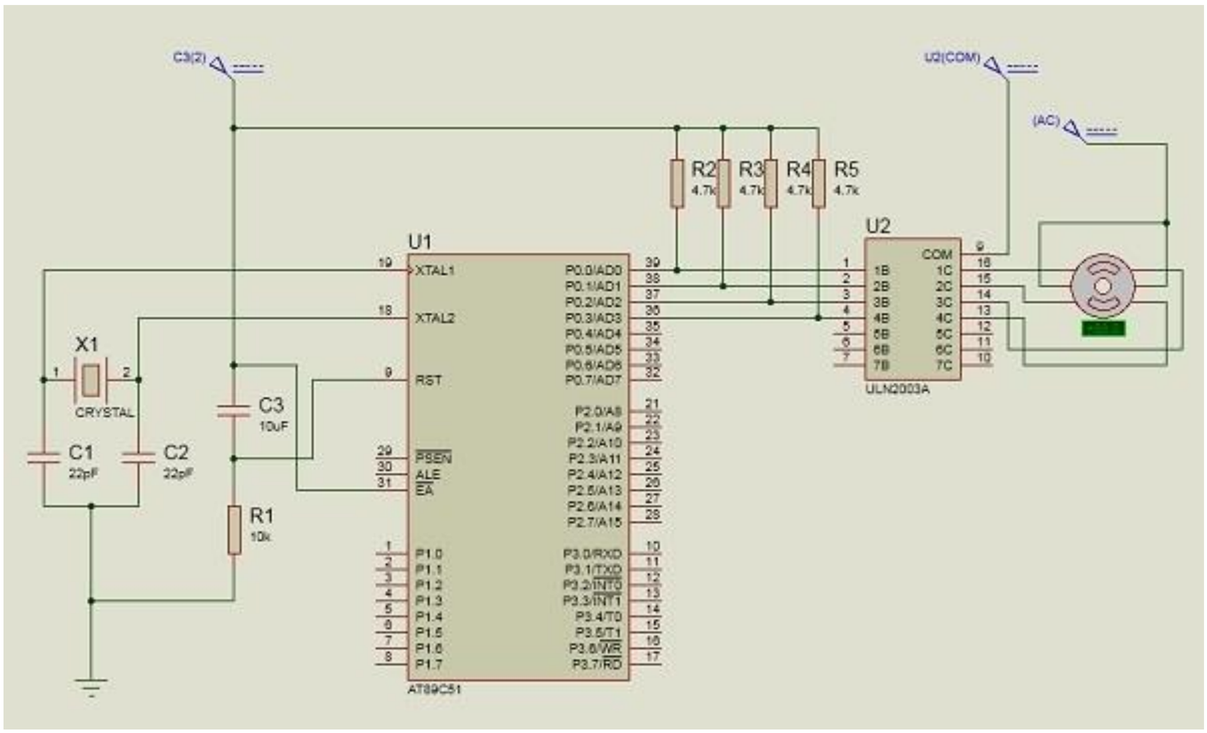
Steps	Winding A	Winding B	Winding C	Winding D
1	1	1	0	0
2	0	1	1	0
3	0	0	1	1
4	1	0	0	1

- **Half Drive Mode −** In this mode, one and two coils are energized alternately. At first, one coil is energized then two coils are energized. This is basically a combination of wave and full drive mode. It increases the angular rotation of the motor

The following table is showing the sequence of input states in different windings.

Steps	Winding A	Winding B	Winding C	Winding D
1	1	0	0	0
2	1	1	0	0
3	0	1	0	0
4	0	1	1	0
5	0	0	1	0
6	0	0	1	1
7	0	0	0	1
8	1	0	0	1

The circuit diagram is like below: We are using the full drive mode.



Example

```
#include<reg51.h>
sbit LED_pin = P2^0; //set the LED pin as P2.0
void delay(int ms){
   unsigned int i, j;
   for(i = 0; i<ms; i++){ // Outer for loop for given
milliseconds value
      for(j = 0; j< 1275; j++){
         //execute in each milliseconds;
       }
   }
}
void main(){
   int rot_angle[] = {0x0C,0x06,0x03,0x09};
   int i;
   while(1){
      //infinite loop for LED blinking
      for(i = 0; i<4; i++){
         P0 = rot_angle[i];
         delay(100);
      }
   }
}
```

8051 TIMER & Interrupt PROGRAMMING IN ASSEMBLY AND C

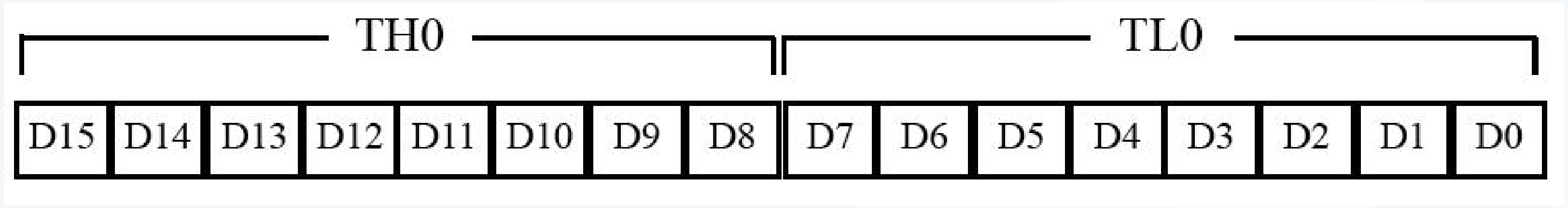
PROGRAMMING 8051 TIMERS

- **Basic registers of the timer**
  - **Timer 0 and Timer 1 are 16 bits wide**
  - **each 16-bit timer is accessed as two separate registers of low byte and high byte.**

PROGRAMMING 8051 TIMERS

- **Timer 0 registers**
  - **low byte register is called TL0 (Timer 0 low byte) and the high byte register is referred to as TH0 (Timer 0 high byte)**
  - **can be accessed like any other register, such as A, B, R0, R1, R2, etc.**
  - **"MOV TL0, #4 FH" moves the value 4FH into TL0**
  - **"MOV R5, TH0" saves TH0 (high byte of Timer 0) in R5**

PROGRAMMING 8051 TIMERS

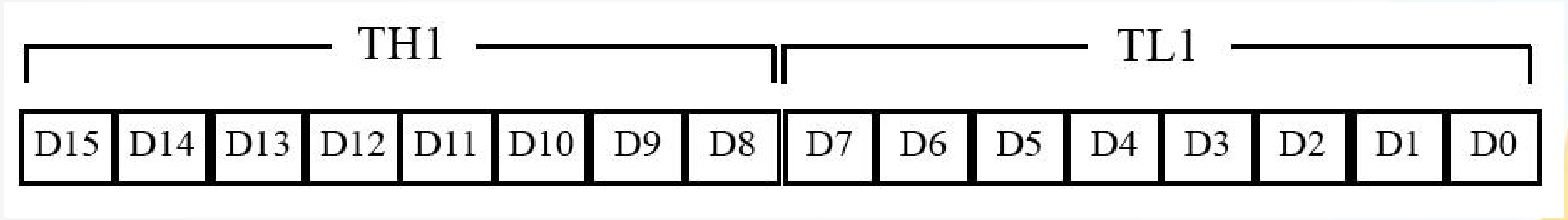


Timer 0 Registers

PROGRAMMING 8051 TIMERS

- **Timer 1 registers**
  - **also 16 bits**
  - **split into two bytes TL1 (Timer 1 low byte) and TH1 (Timer 1 high byte)**
  - **accessible in the same way as the registers of Timer 0.**

SECTION 9.1: PROGRAMMING 8051 TIMERS

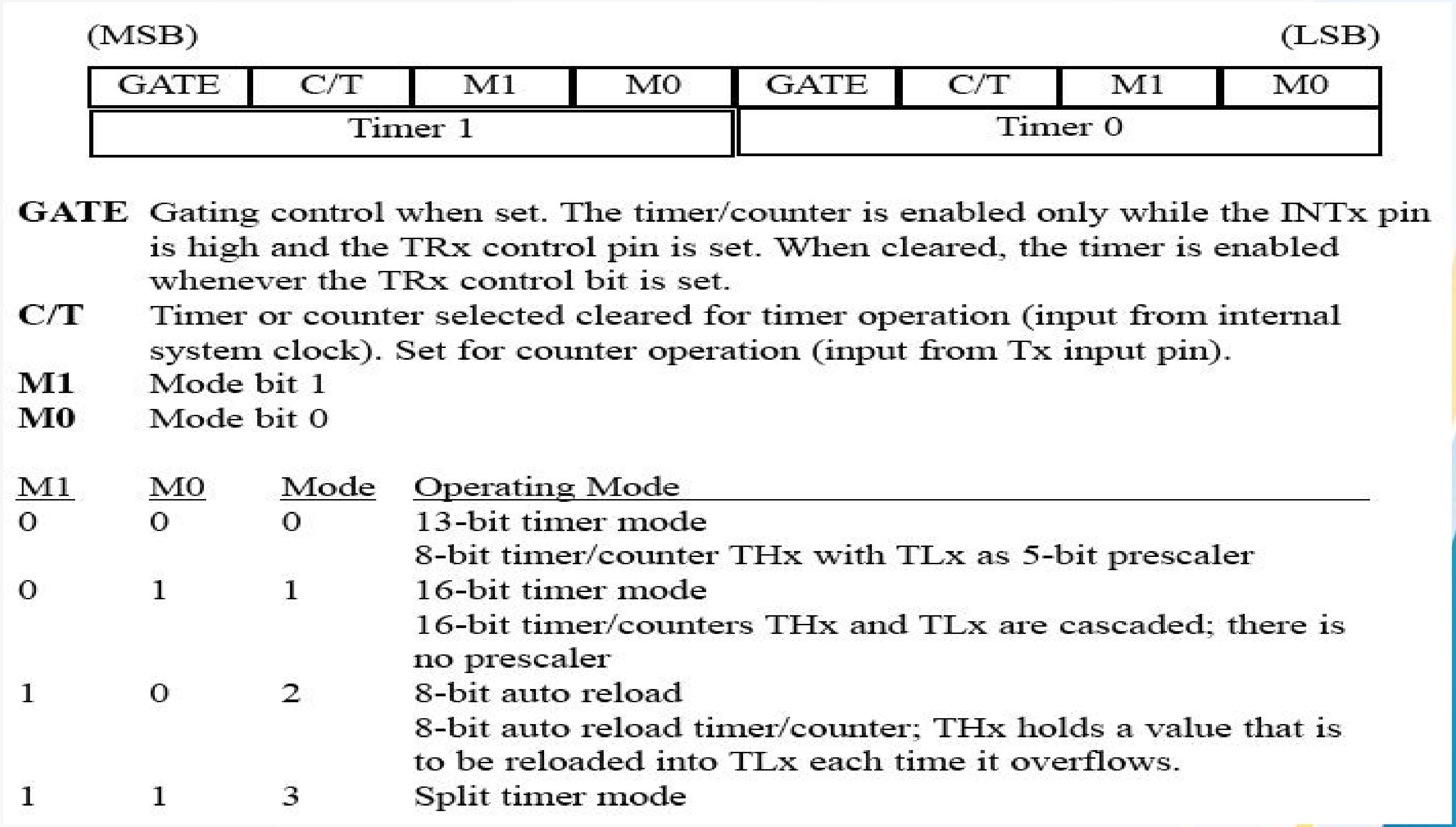


Timer 1 Registers

PROGRAMMING 8051 TIMERS

- **TMOD (timer mode) register**
  - **timers 0 and 1 use TMOD register to set operation modes (only learn Mode 1 and 2)**
  - **8-bit register**
  - **lower 4 bits are for Timer 0**
  - **upper 4 bits are for Timer 1**
  - **lower 2 bits are used to set the timer mode**
    - **(only learn Mode 1 and 2)**
  - **upper 2 bits to specify the operation**
    - **(only learn timer operation)**

PROGRAMMING 8051 TIMERS



(MSB) (LSB)

GATE	C/T	M1	M0	GATE	C/T	M1	M0
Timer 1				Timer 0			

GATE	Gating control when set. The timer/counter is enabled only while the INTx pin is high and the TRx control pin is set. When cleared, the timer is enabled whenever the TRx control bit is set.
C/T	Timer or counter selected cleared for timer operation (input from internal system clock). Set for counter operation (input from Tx input pin).
M1	Mode bit 1
M0	Mode bit 0

M1	M0	Mode	Operating Mode
0	0	0	13-bit timer mode 8-bit timer/counter THx with TLx as 5-bit prescaler
0	1	1	16-bit timer mode 16-bit timer/counters THx and TLx are cascaded; there is no prescaler
1	0	2	8-bit auto reload 8-bit auto reload timer/counter; THx holds a value that is to be reloaded into TLx each time it overflows.
1	1	3	Split timer mode

TMOD Register

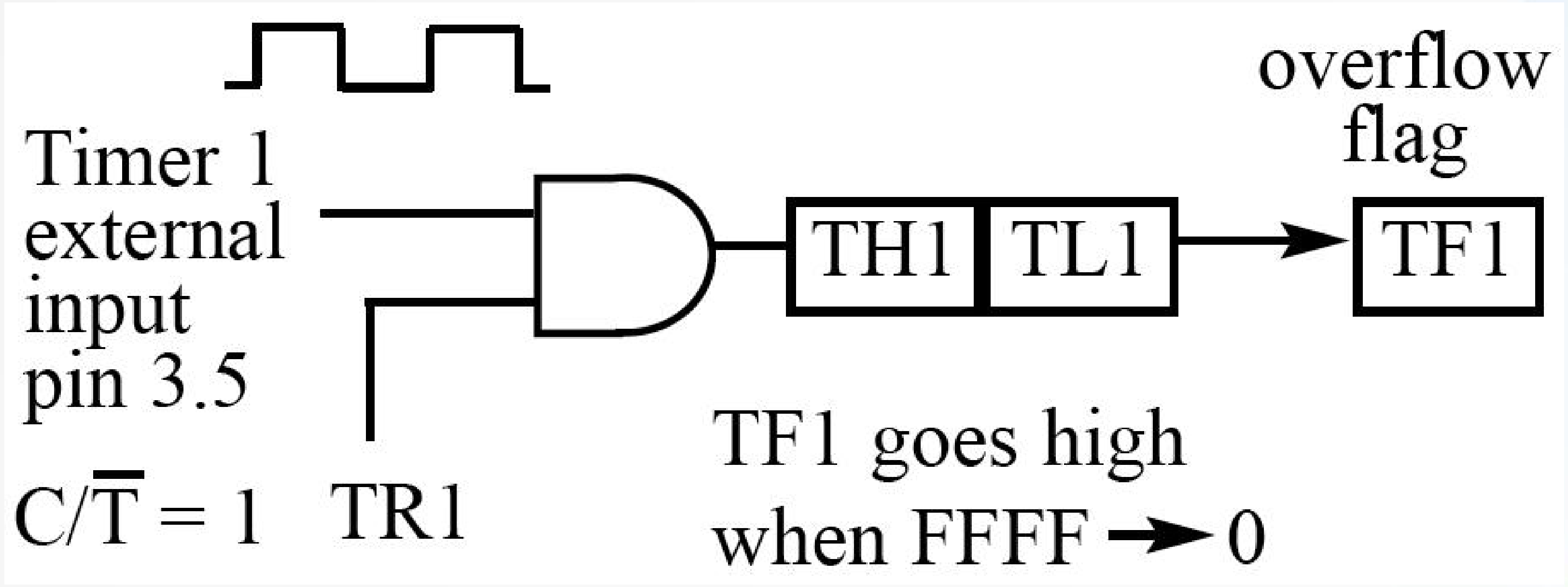
PROGRAMMING 8051 TIMERS

- Clock source for timer
  - timer needs a clock pulse to tick
  - if C/T = 0, the crystal frequency attached to the 8051 is the source of the clock for the timer
  - frequency for the timer is always 1/12th the frequency of the crystal attached to the 8051
  - XTAL = 11.0592 MHz allows the 8051 system to communicate with the PC with no errors
  - In our case, the timer frequency is 1MHz since our crystal frequency is 12MHz

PROGRAMMING 8051 TIMERS

- Mode 1 programming
  - 16-bit timer, values of 0000 to FFFFH
  - TH and TL are loaded with a 16-bit initial value
  - timer started by "SETB TR0" for Timer 0 and "SETB TR1" for Timer l
  - timer count ups until it reaches its limit of FFFFH
  - rolls over from FFFFH to 0000H
  - sets TF (timer flag)
  - when this timer flag is raised, can stop the timer with "CLR TR0" or "CLR TR1“
  - after the timer reaches its limit and rolls over, the registers TH and TL must be reloaded with the original value and TF must be reset to 0

PROGRAMMING 8051 TIMERS

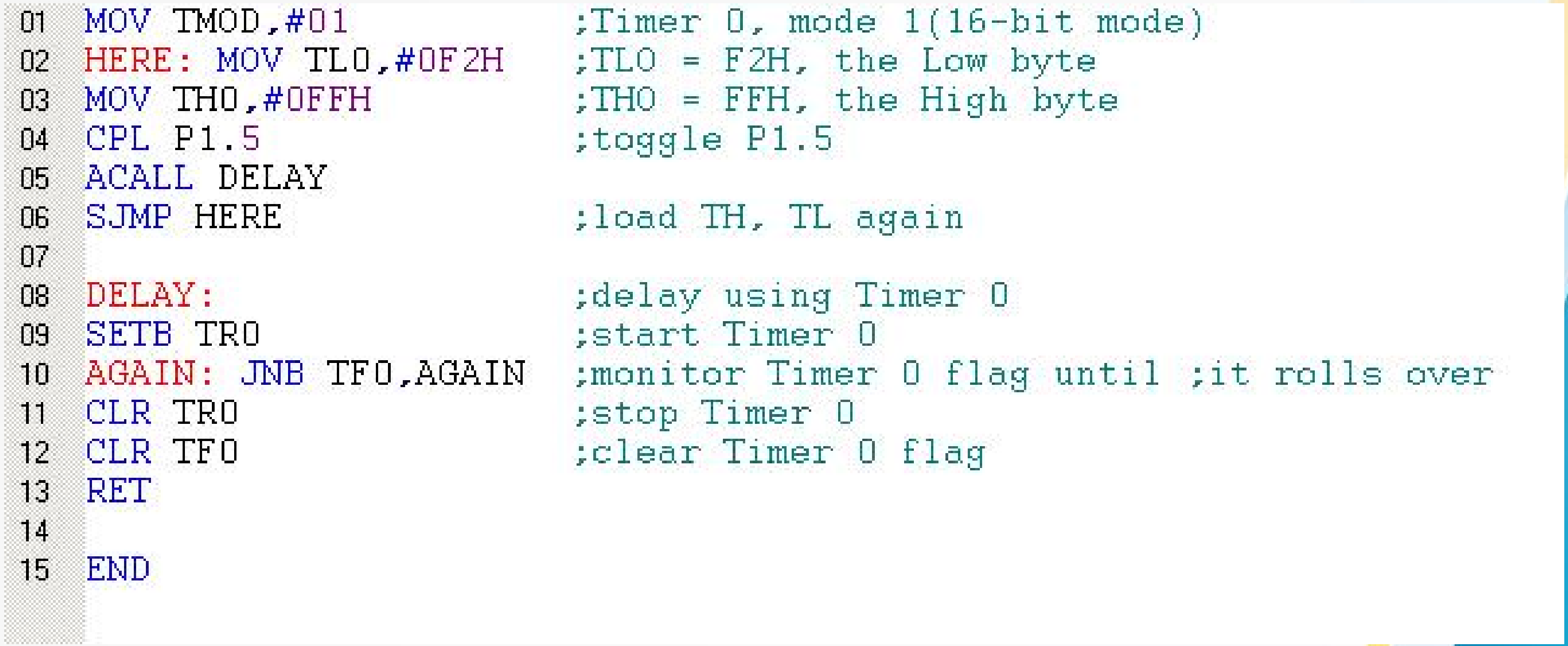


Timer 1 with External Input (Mode 1)

PROGRAMMING 8051 TIMERS

- **Steps to program in mode 1**
  - **Set timer mode 1 or 2**
  - **Set TL0 and TH0 (for mode 1 16 bit mode)**
  - **Set TH0 only (for mode 2 8 bit auto reload mode)**
  - **Run the timer**
  - **Monitor the timer flag bit**

In the following program, we are creating a square wave of 50% duty cycle (with equal portions high and low) on the P1.5 bit.
Timer 0 is used to generate the time delay



```
MOV TMOD,#01              ;Timer 0, mode 1(16-bit mode)
HERE: MOV TL0,#0F2H       ;TL0 = F2H, the Low byte
MOV TH0,#0FFH             ;TH0 = FFH, the High byte
CPL P1.5                  ;toggle P1.5
ACALL DELAY
SJMP HERE                 ;load TH, TL again

DELAY:                    ;delay using Timer 0
SETB TR0                  ;start Timer 0
AGAIN: JNB TF0,AGAIN      ;monitor Timer 0 flag until ;it rolls over
CLR TR0                   ;stop Timer 0
CLR TF0                   ;clear Timer 0 flag
RET

END
```

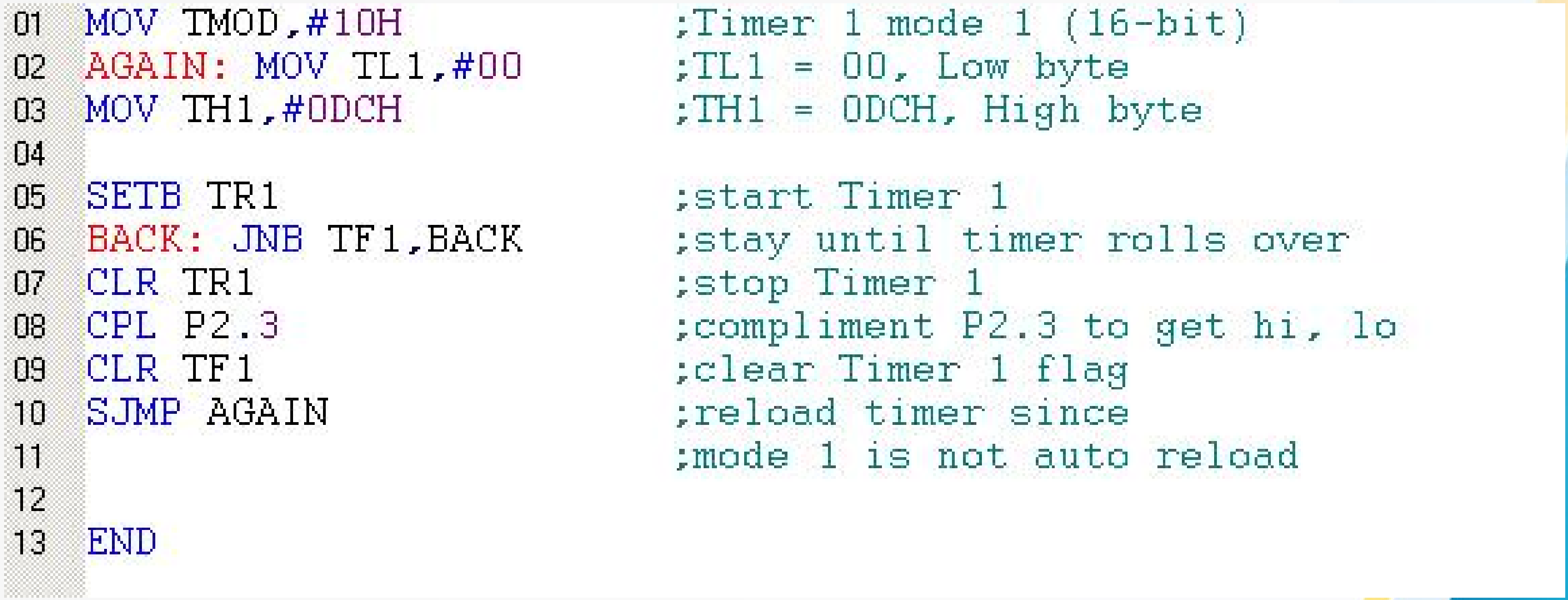
PROGRAMMING 8051 TIMERS

- **Finding values to be loaded into the timer**
  - **XTAL = 11.0592 MHz (12MHz)**
  - **divide the desired time delay by 1.085∓ s (1∓ s) to get *n***
  - **65536 – *n* = *N***
  - **convert *N* to hex yyxx**
  - **set TL = xx and TH = yy**

Assuming XTAL = 11.0592 MHz, write a program to generate a square wave of 50 Hz frequency on pin P2.3.

- T = 1/50 Hz = 20 ms
- 1/2 of it for the high and low portions of the pulse = 10 ms
- 10 ms / 1.085 us = 9216
- 65536 - 9216 = 56320 in decimal = DC00H
- TL = 00 and TH = DCH
- The calculation for 12MHz crystal uses the same steps

Assuming XTAL = 11.0592 MHz, write a program to generate a square wave of 50 Hz frequency on pin P2.3.

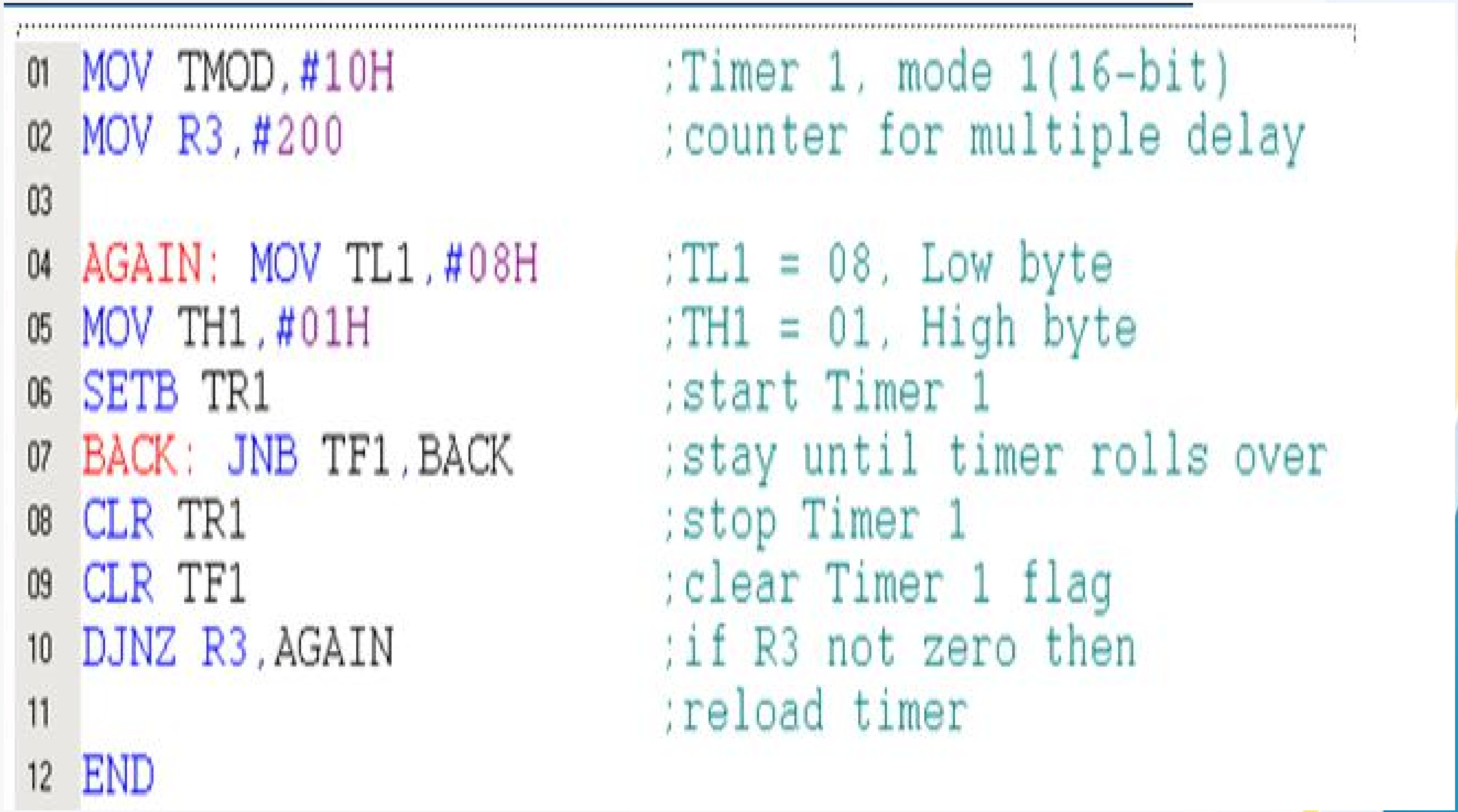


```
MOV TMOD,#10H            ;Timer 1 mode 1 (16-bit)
AGAIN: MOV TL1,#00       ;TL1 = 00, Low byte
MOV TH1,#0DCH            ;TH1 = 0DCH, High byte

SETB TR1                 ;start Timer 1
BACK: JNB TF1,BACK       ;stay until timer rolls over
CLR TR1                  ;stop Timer 1
CPL P2.3                 ;compliment P2.3 to get hi, lo
CLR TF1                  ;clear Timer 1 flag
SJMP AGAIN               ;reload timer since
                         ;mode 1 is not auto reload


END
```

PROGRAMMING 8051 TIMERS

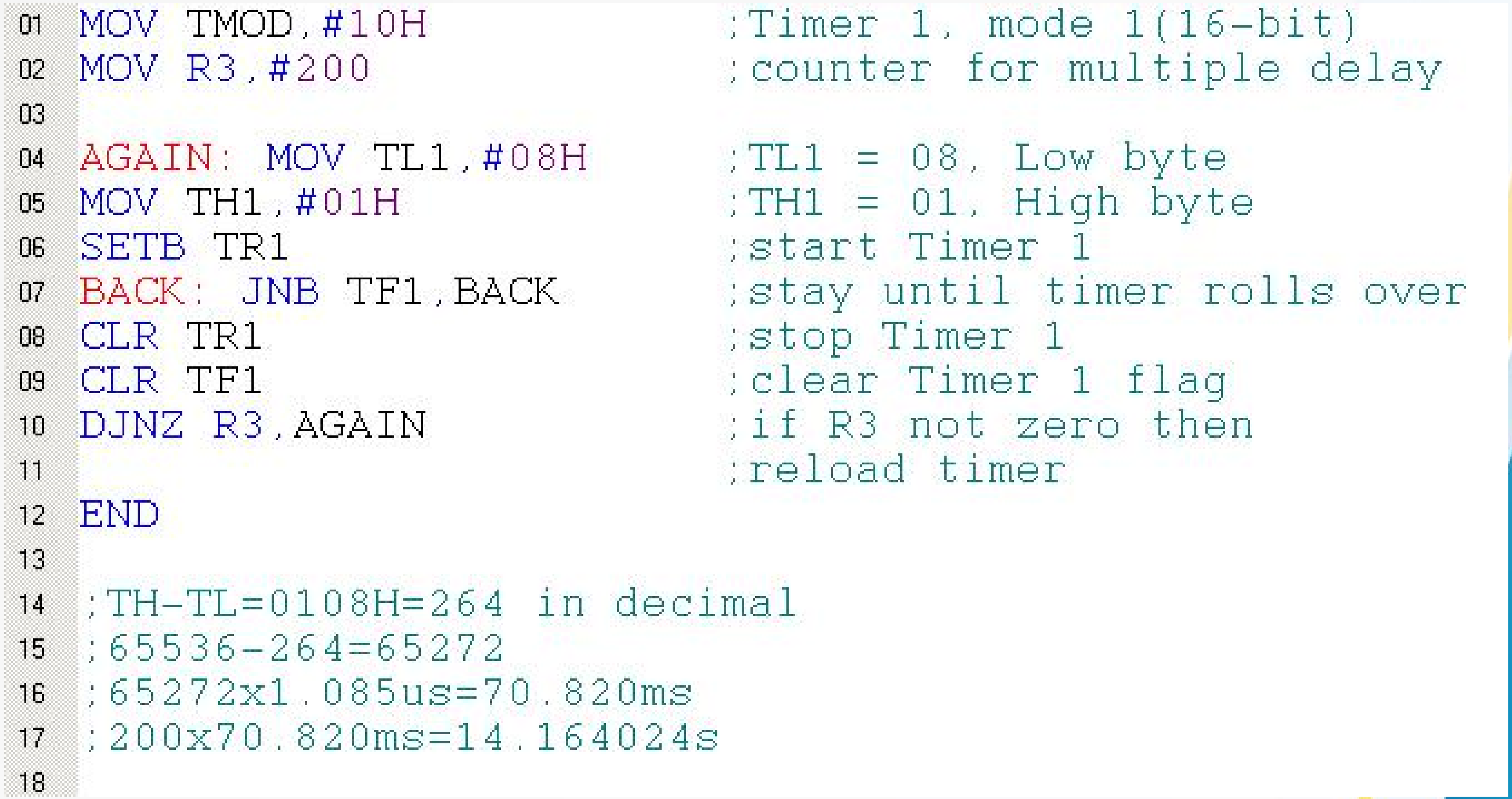
- **Generating a large time delay**
  - **size of the time delay depends**
    - **crystal frequency**
    - **timer's 16-bit register in mode 1**
  - **largest time delay is achieved by making both TH and TL zero**

Examine the following program and find the time delay in seconds. Exclude the time delay due to the instructions in the loop.



```
MOV TMOD,#10H              ;Timer 1, mode 1(16-bit)
MOV R3,#200                ;counter for multiple delay

AGAIN: MOV TL1,#08H        ;TL1 = 08, Low byte
MOV TH1,#01H               ;TH1 = 01, High byte
SETB TR1                   ;start Timer 1
BACK: JNB TF1,BACK         ;stay until timer rolls over
CLR TR1                    ;stop Timer 1
CLR TF1                    ;clear Timer 1 flag
DJNZ R3,AGAIN              ;if R3 not zero then
                           ;reload timer
END
```

Examine the following program and find the time delay in seconds. Exclude the time delay due to the instructions in the loop.



```
MOV TMOD,#10H              ;Timer 1, mode 1(16-bit)
MOV R3,#200                ;counter for multiple delay

AGAIN: MOV TL1,#08H        ;TL1 = 08, Low byte
MOV TH1,#01H               ;TH1 = 01, High byte
SETB TR1                   ;start Timer 1
BACK: JNB TF1,BACK         ;stay until timer rolls over
CLR TR1                    ;stop Timer 1
CLR TF1                    ;clear Timer 1 flag
DJNZ R3,AGAIN              ;if R3 not zero then
                           ;reload timer
END

;TH-TL=0108H=264 in decimal
;65536-264=65272
;65272x1.085us=70.820ms
;200x70.820ms=14.164024s
```

PROGRAMMING 8051 TIMERS (for information only)

- **Mode 0**
  - **works like mode 1**
  - **13-bit timer instead of 16-bit**
  - **13-bit counter hold values 0000 to 1FFFH**
  - **when the timer reaches its maximum of 1FFFH, it rolls over to 0000, and TF is set**

PROGRAMMING 8051 TIMERS

- **Mode 2 programming**
  - **8-bit timer, allows values of 00 to FFH**
  - **TH is loaded with the 8-bit value**
  - **a copy is given to TL**
  - **timer is started by ,"SETB TR0" or "SETB TR1“**
  - **starts to count up by incrementing the TL register**
  - **counts up until it reaches its limit of FFH**
  - **when it rolls over from FFH to 00, it sets high TF**
  - **TL is reloaded automatically with the value in TH**
  - **To repeat, clear TF**
  - **mode 2 is an auto-reload mode**

PROGRAMMING 8051 TIMERS

- **Steps to program in mode 2**
  1. **load TMOD, select mode 2**
  2. **load the TH**
  3. **start timer**
  4. **monitor the timer flag (TF) with "JNB”**
  5. **get out of the loop when TF=1**
  6. **clear TF**
  7. **go back to Step 4 since mode 2 is auto-reload**

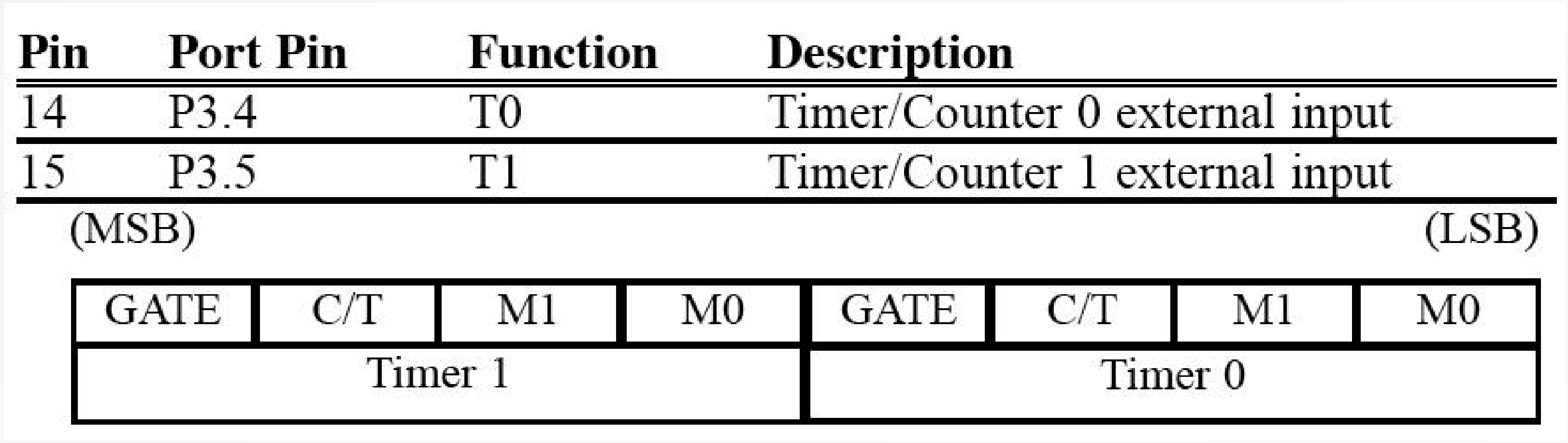
PROGRAMMING 8051 TIMERS

- **Assemblers and negative values**
  - **can let the assembler calculate the value for TH and TL which makes the job easier**
  - **"MOV TH1, # -100", the assembler will calculate the -100 = 9CH**
  - **"MOV TH1,#high(-10000) "**
  - **"MOV TL1,#low(-10000) "**

COUNTER PROGRAMMING

- **C/T bit in TMOD register**
  - C/T bit in the TMOD register decides the source of the clock for the timer
  - C/T = 0, timer gets pulses from crystal
  - C/T = 1, the timer used as counter and gets pulses from outside the 8051
  - C/T = 1, the counter counts up as pulses are fed from pins 14 and 15
  - pins are called T0 (Timer 0 input) and T1 (Timer 1 input)
  - these two pins belong to port 3
  - Timer 0, when C/T = 1, pin P3.4 provides the clock pulse and the counter counts up for each clock pulse coming from that pin
  - Timer 1, when C/T = 1 each clock pulse coming in from pin P3.5 makes the counter count up

COUNTER PROGRAMMING



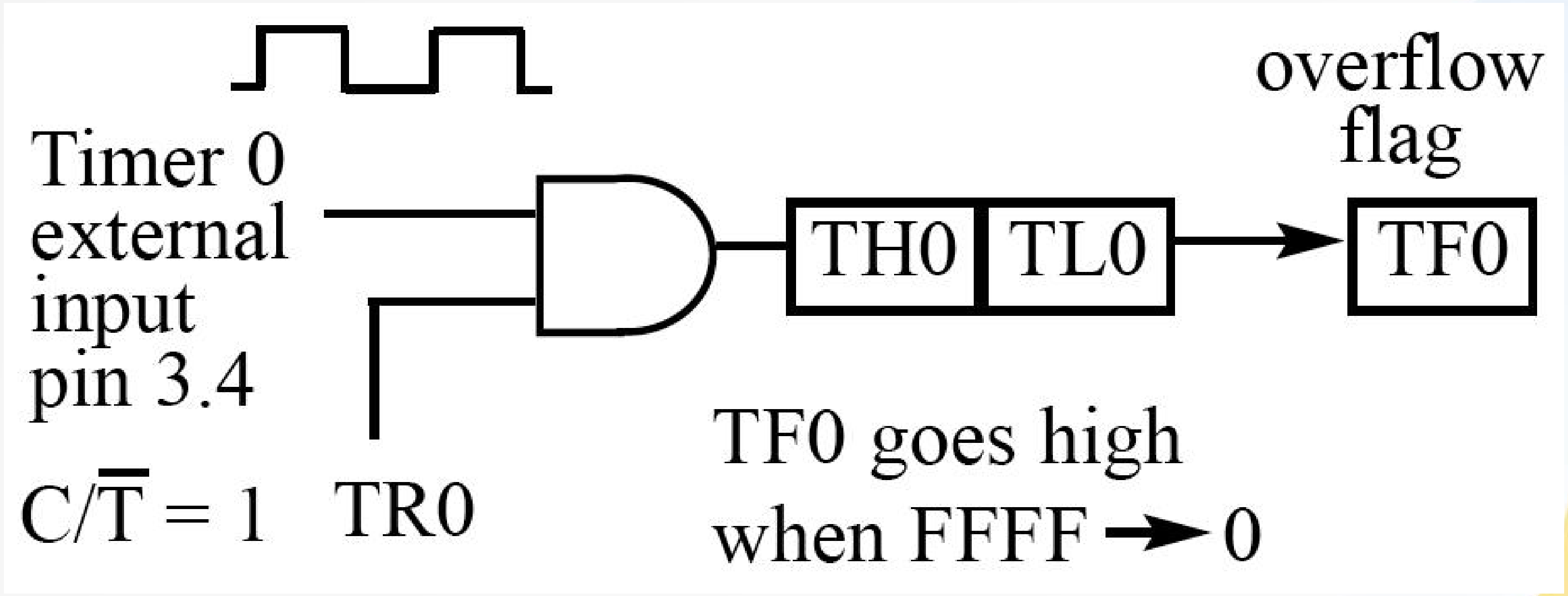
Pin	Port Pin	Function	Description
14	P3.4	T0	Timer/Counter 0 external input
15	P3.5	T1	Timer/Counter 1 external input

(MSB) (LSB)

GATE	C/T	M1	M0	GATE	C/T	M1	M0
Timer 1				Timer 0			

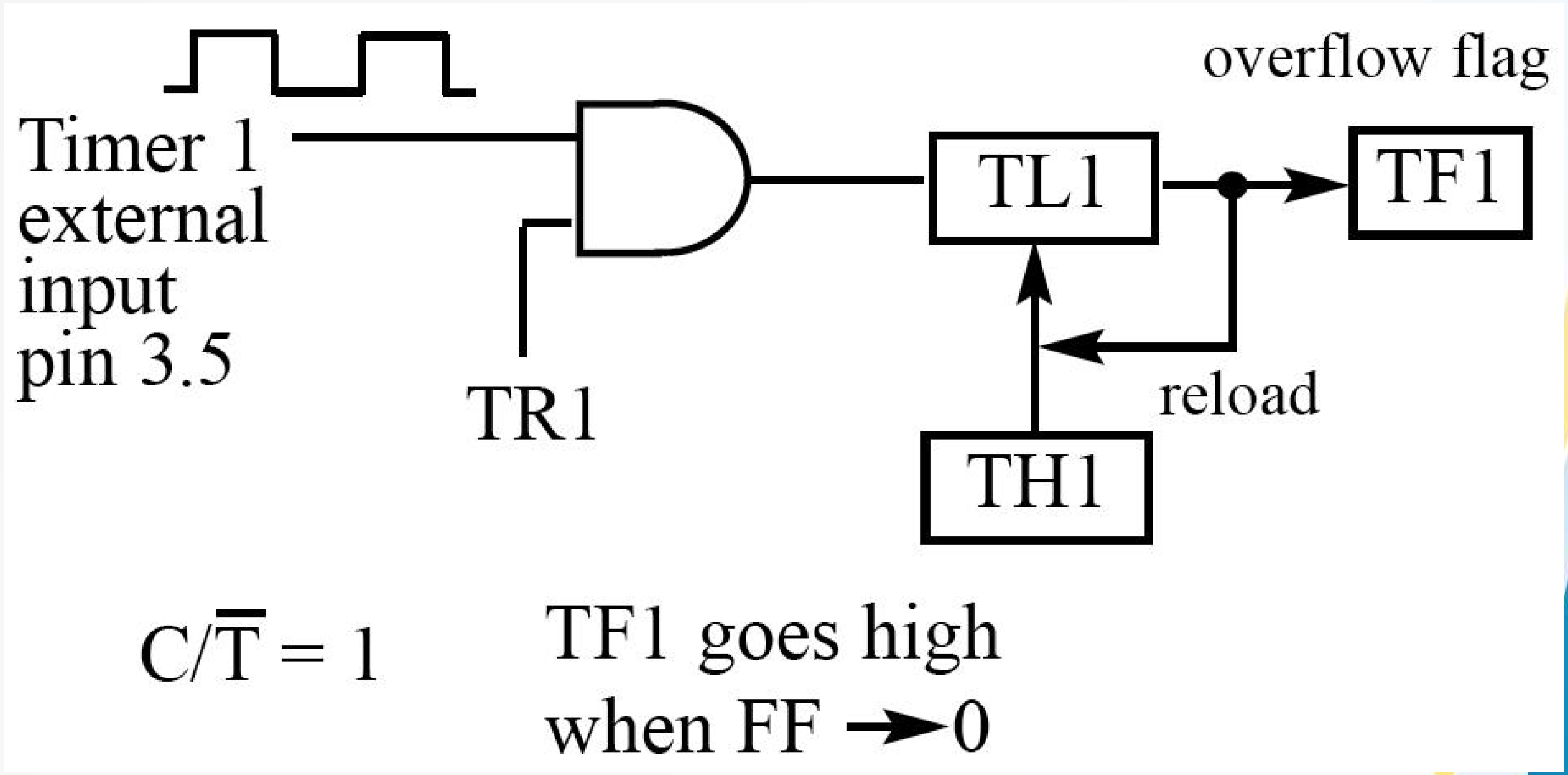
Port 3 Pins Used For Timers 0 and 1

PROGRAMMING 8051 TIMERS


Timer 0 external input pin 3.4
C/$\overline{T}$ = 1
TR0
TH0
TL0
overflow flag
TF0
TF0 goes high when FFFF $\rightarrow$ 0

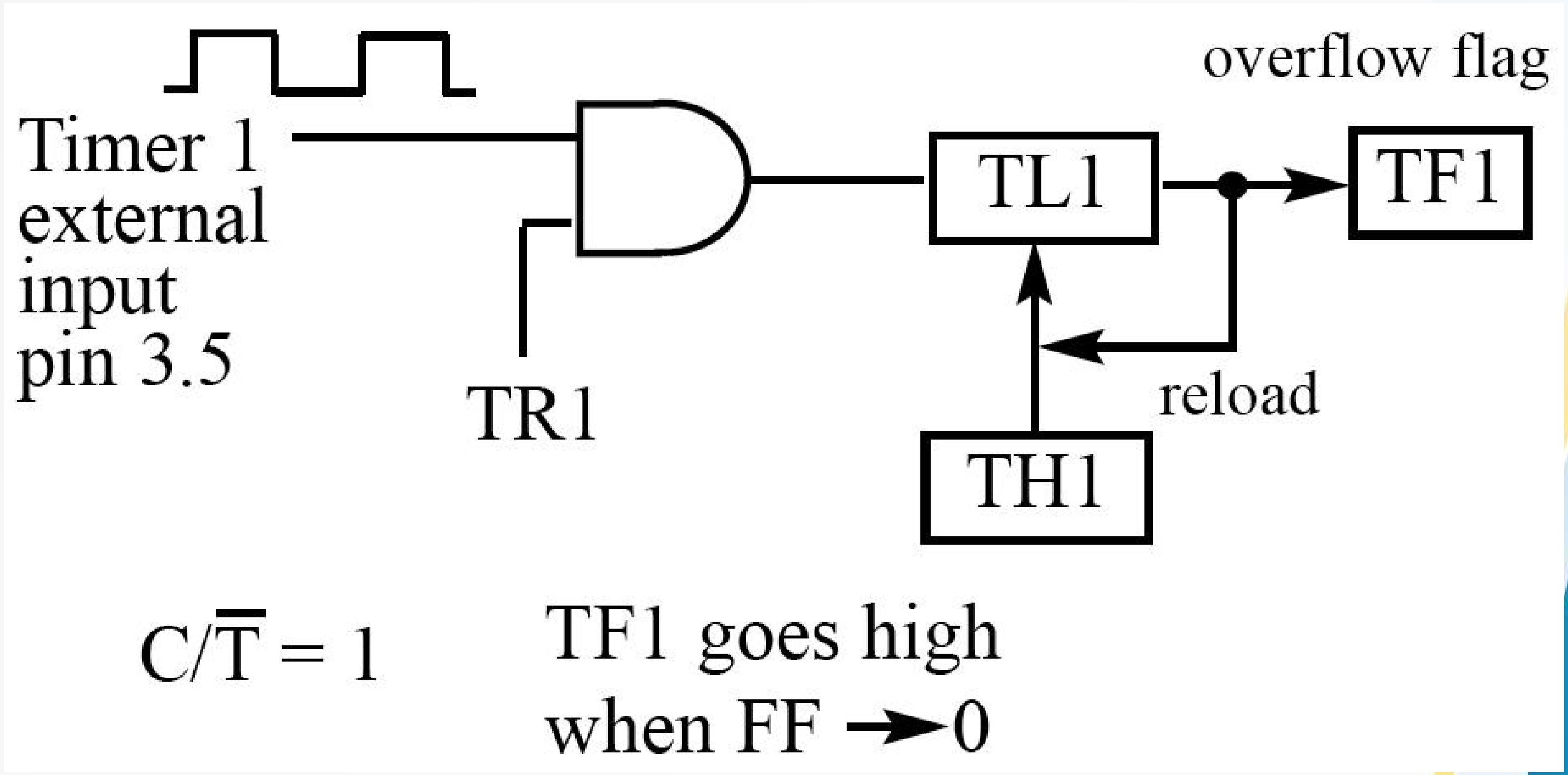
Timer 0 with External Input (Mode 1)

COUNTER PROGRAMMING

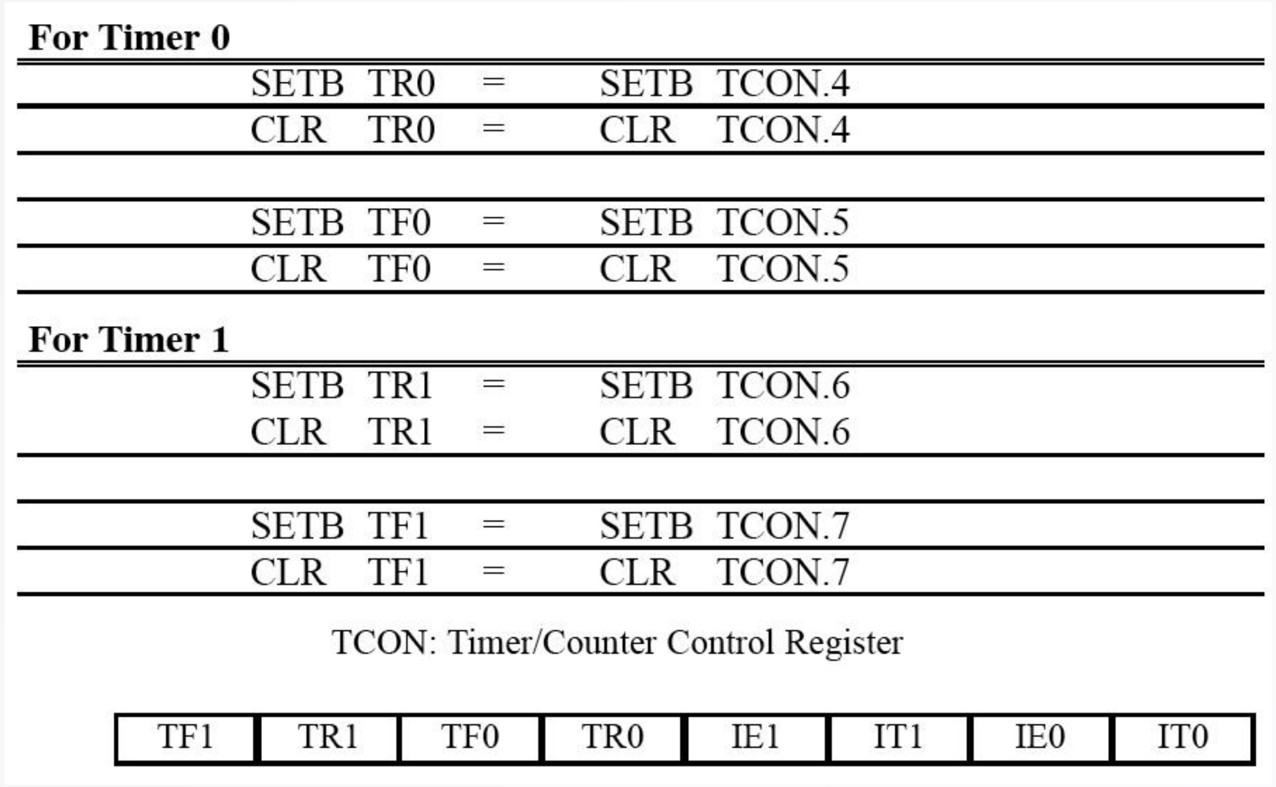

overflow flag
Timer 1 external input pin 3.5
TR1
TL1
TF1
reload
TH1
$C/\overline{T} = 1$
TF1 goes high when FF $\rightarrow$ 0

Timer 1 with External Input (Mode 2)

SECTION 9.2: COUNTER PROGRAMMING


overflow flag
Timer 1 external input pin 3.5
TR1
TL1
TF1
reload
TH1
$C/\overline{T} = 1$
TF1 goes high when FF $\rightarrow$ 0

COUNTER PROGRAMMING



For Timer 0

SETB	TR0	=	SETB	TCON.4
CLR	TR0	=	CLR	TCON.4
SETB	TF0	=	SETB	TCON.5
CLR	TF0	=	CLR	TCON.5

For Timer 1

SETB	TR1	=	SETB	TCON.6
CLR	TR1	=	CLR	TCON.6
SETB	TF1	=	SETB	TCON.7
CLR	TF1	=	CLR	TCON.7

TCON: Timer/Counter Control Register

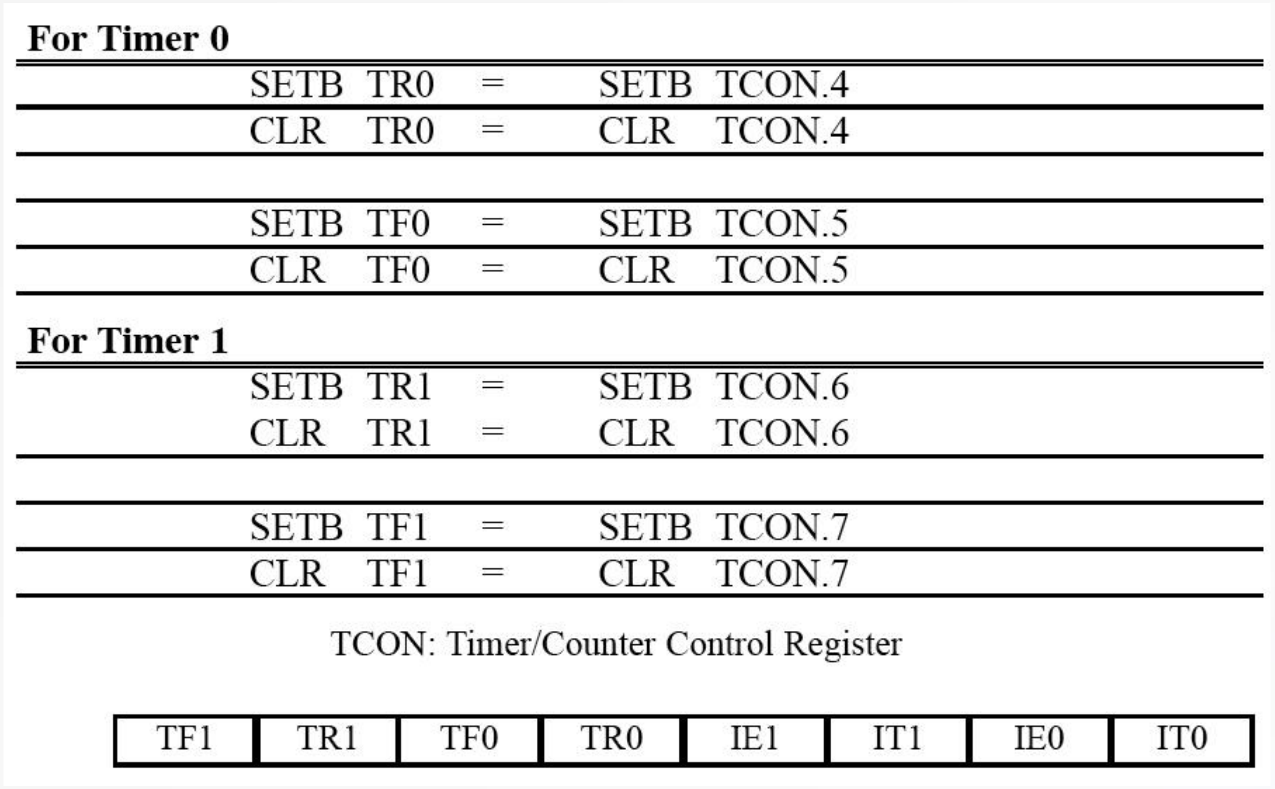
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

Port 3 Pins Used For Timers 0 and 1

COUNTER PROGRAMMING

- **TCON register**
  - **TR0 and TR1 flags turn on or off the timers**
  - **bits are part of a register called TCON (timer control)**
  - **upper four bits are used to store the TF and TR bits of both Timer 0 and Timer 1**
  - **lower four bits are set aside for controlling the interrupt bits**
  - **"SETB TRl" and "CLR TRl“**
  - **"SETB TCON. 6" and "CLR TCON. 6“**

COUNTER PROGRAMMING



For Timer 0

SETB	TR0	=	SETB	TCON.4
CLR	TR0	=	CLR	TCON.4
SETB	TF0	=	SETB	TCON.5
CLR	TF0	=	CLR	TCON.5

For Timer 1

SETB	TR1	=	SETB	TCON.6
CLR	TR1	=	CLR	TCON.6
SETB	TF1	=	SETB	TCON.7
CLR	TF1	=	CLR	TCON.7

TCON: Timer/Counter Control Register

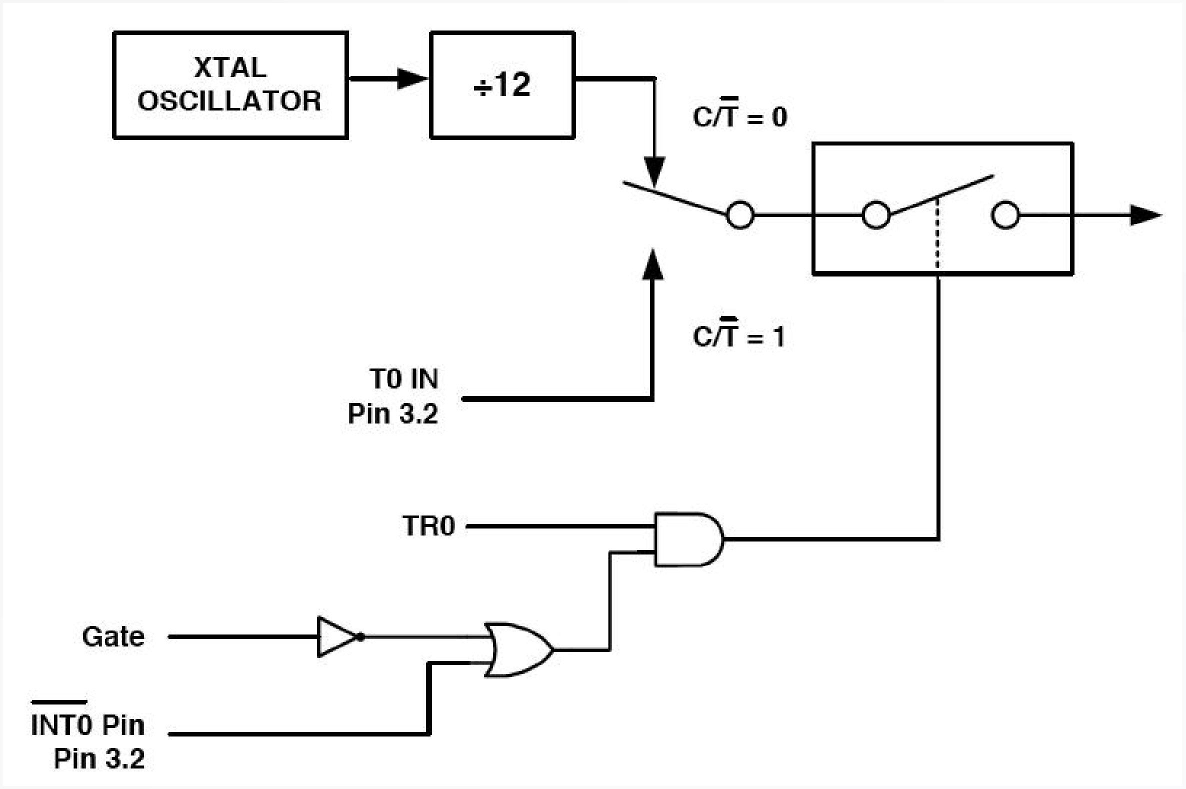
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

Equivalent Instructions for the Timer Control Register (TCON)

COUNTER PROGRAMMING

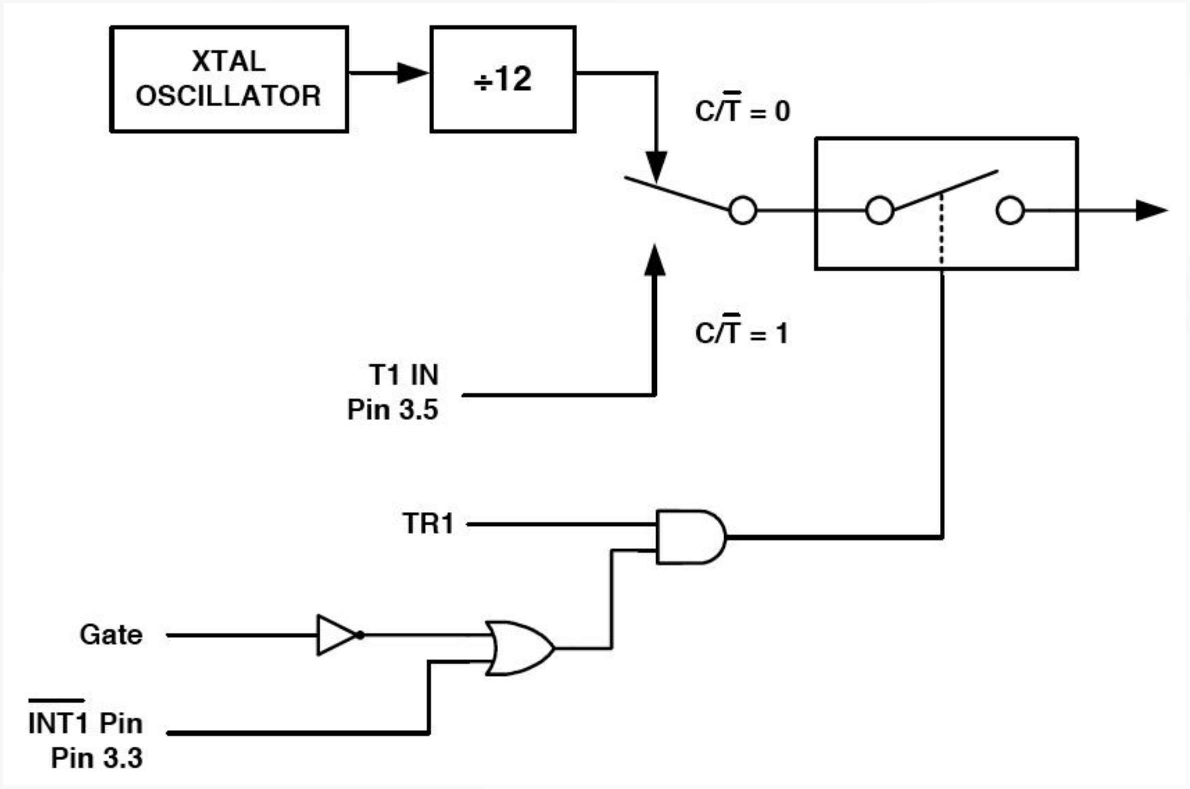
- **The case of GATE = 1 in TMOD**
  - **GATE = 0, the timer is started with instructions** "SETB TR0" and "SETB TR1“
  - **GATE = 1, the start and stop of the timers are done externally through pins P3.2 and P3.3**
  - **allows us to start or stop the timer externally at any time via a simple switch**

COUNTER PROGRAMMING



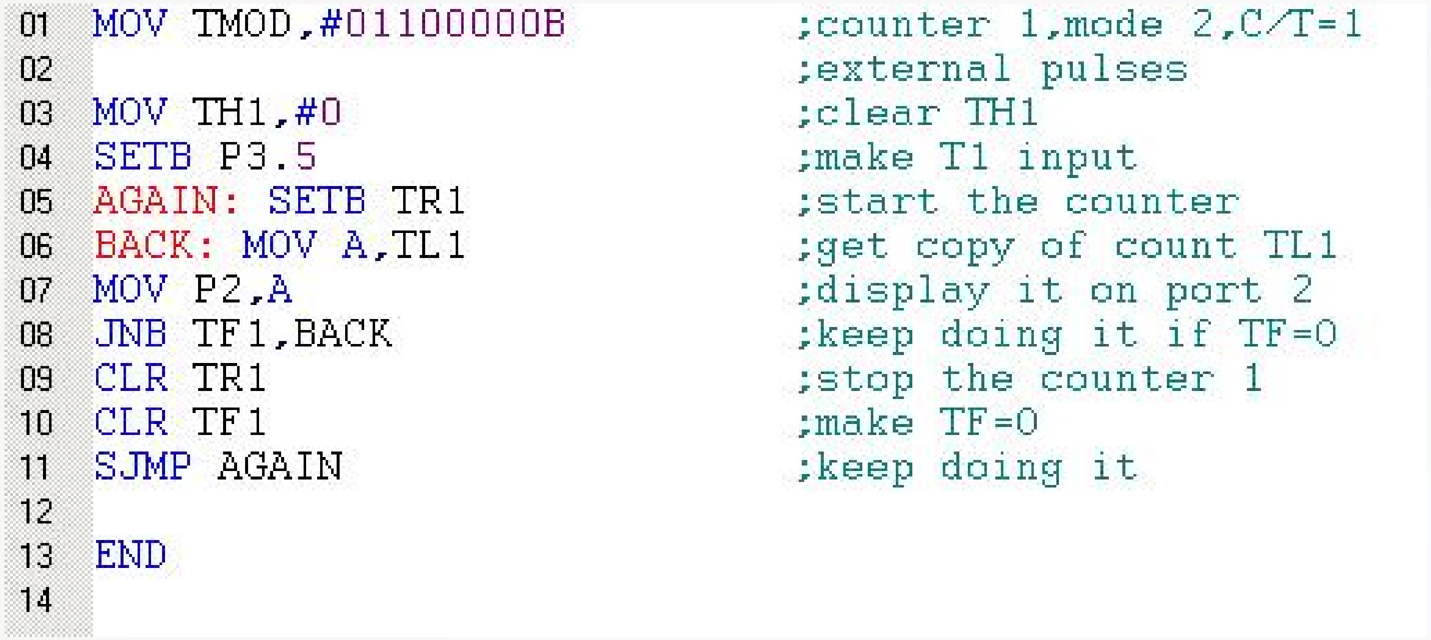
Timer/Counter 0

COUNTER PROGRAMMING



Timer/Counter 1

Assuming that clock pulses are fed into pin T1, write a program for counter 1 in mode 2 to count the pulses and display the state of the TL1 count on P2. (for information only)



```
MOV TMOD,#01100000B                   ;counter 1,mode 2,C/T=1
                                      ;external pulses
MOV TH1,#0                            ;clear TH1
SETB P3.5                             ;make T1 input
AGAIN: SETB TR1                       ;start the counter
BACK: MOV A,TL1                       ;get copy of count TL1
MOV P2,A                              ;display it on port 2
JNB TF1,BACK                          ;keep doing it if TF=0
CLR TR1                               ;stop the counter 1
CLR TF1                               ;make TF=0
SJMP AGAIN                            ;keep doing it

END

```

Interrupts vs. Polling

- An interrupt is an external or internal event that interrupts the microcontroller
  - To inform it that a device needs its service
- A single microcontroller can serve several devices by two ways
  - Interrupts
    - Whenever any device needs its service, the device notifies the microcontroller by sending it an interrupt signal
    - Upon receiving an interrupt signal, the microcontroller interrupts whatever it is

Interrupts vs. Polling (cont.)

- The program which is associated with the interrupt is called the interrupt service routine (ISR) or interrupt handler

- Polling
  - The microcontroller continuously monitors the status of a given device
  - ex. JNB TF, target
  - When the conditions met, it performs the service
  - After that, it moves on to monitor the next device until every one is serviced
  - Polling can monitor the status of several devices and serve each of them as certain conditions are met
  - The polling method is not efficient, since it wastes much of the microcontroller's

Interrupts vs. Polling (cont.)

- The advantage of interrupts is:
  - The microcontroller can serve many devices (not all at the same time)
    - Each device can get the attention of the microcontroller based on the assigned priority
    - For the polling method, it is not possible to assign priority since it checks all devices in a round-robin fashion
  - The microcontroller can also ignore (mask) a device request for service
    - This is not possible for the polling

Interrupt Service Routine

- For every interrupt, there must be an interrupt service routine (ISR), or interrupt handler
  - When an interrupt is invoked, the microcontroller runs the interrupt service routine
  - There is a fixed location in memory that holds the address of its ISR
    - The group of memory locations set aside to hold the addresses of ISRs is called interrupt vector table

Steps in Executing an Interrupt

- Upon activation of an interrupt, the microcontroller goes through:
  - It finishes the instruction it is executing and saves the address of the next instruction (PC) on the stack
  - It also saves the current status of all the registers internally (not on the stack)
  - It jumps to a fixed location in memory, called the interrupt vector table, that holds the address of the ISR

Steps in Executing an Interrupt (cont.)

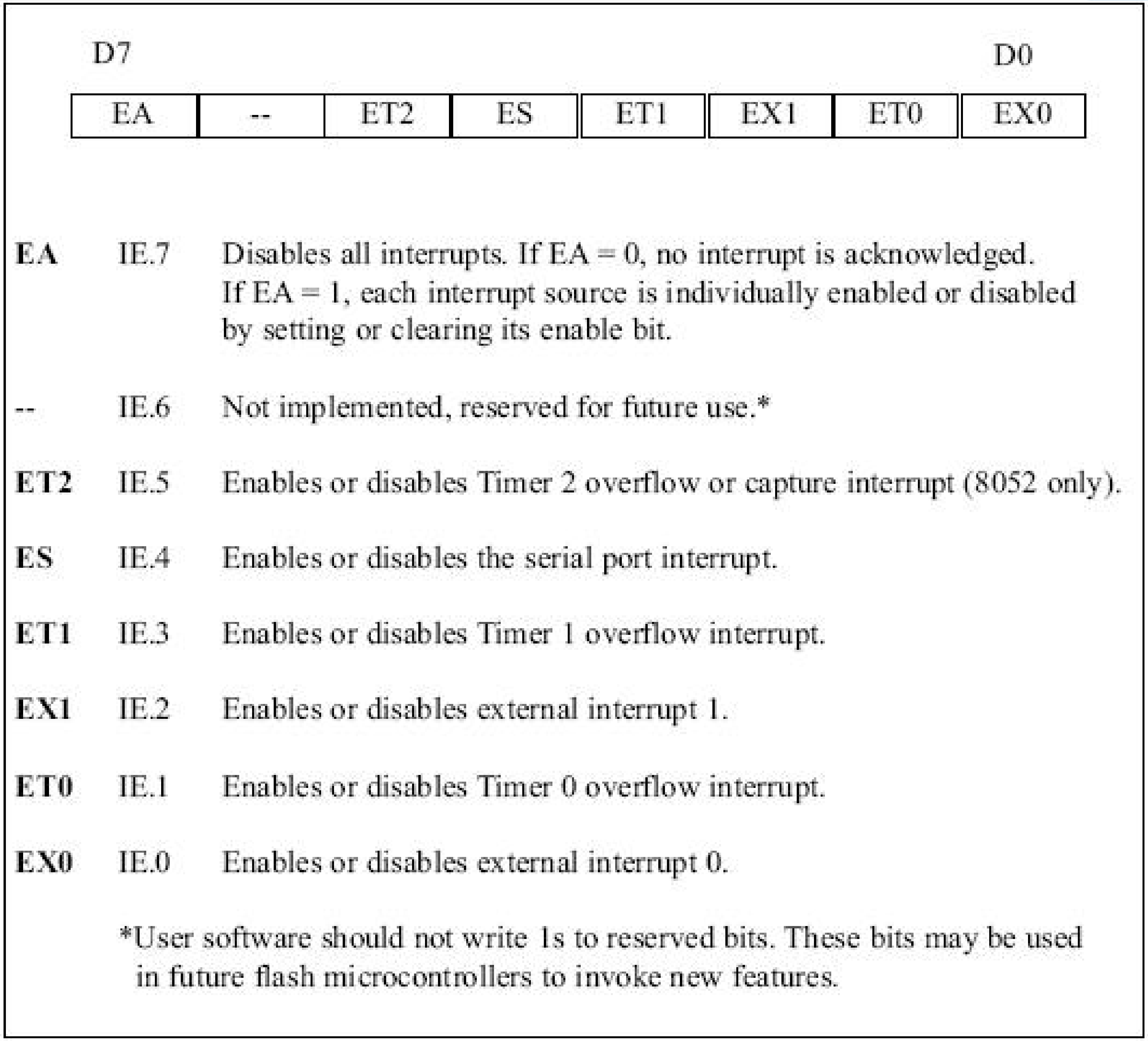
- It gets the address of the ISR from the interrupt vector table and jumps to ISR
    - It starts to execute the interrupt service subroutine until it reaches the last instruction of the subroutine which is RETI (return from interrupt)
- Upon executing the RETI instruction, the microcontroller returns to the place where it was interrupted
    - It gets the program counter (PC) address from the stack by popping the top two bytes of the stack into the PC
    - It starts to execute from that address

Six Interrupts in 8051

- Six interrupts are allocated as follows
  - Reset – power-up reset
  - Two interrupts are set aside for the timers:
    - One for timer 0 and one for timer 1
  - Two interrupts are set aside for hardware external interrupts
    - P3.2 and P3.3 are for the external hardware interrupts INT0 (or EX1), and INT1 (or EX2)
  - Serial communication has a single interrupt that belongs to both receive and transfer

Enabling and Disabling an Interrupt

- Upon reset, all interrupts are disabled (masked)
  - None will be responded to by the microcontroller if they are activated
    - The interrupts must be enabled by software in order for the microcontroller to respond to them
  - There is a register called IE (interrupt enable) that is responsible for enabling (unmasking) and disabling (masking) the interrupts



D7 D0

EA	--	ET2	ES	ET1	EX1	ET0	EX0

EA	IE.7	Disables all interrupts. If EA = 0, no interrupt is acknowledged. If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.
--	IE.6	Not implemented, reserved for future use.*
ET2	IE.5	Enables or disables Timer 2 overflow or capture interrupt (8052 only).
ES	IE.4	Enables or disables the serial port interrupt.
ET1	IE.3	Enables or disables Timer 1 overflow interrupt.
EX1	IE.2	Enables or disables external interrupt 1.
ET0	IE.1	Enables or disables Timer 0 overflow interrupt.
EX0	IE.0	Enables or disables external interrupt 0.

*User software should not write 1s to reserved bits. These bits may be used in future flash microcontrollers to invoke new features.

Figure 11-2. IE (Interrupt Enable) Register

Enabling and Disabling an Interrupt (cont.)

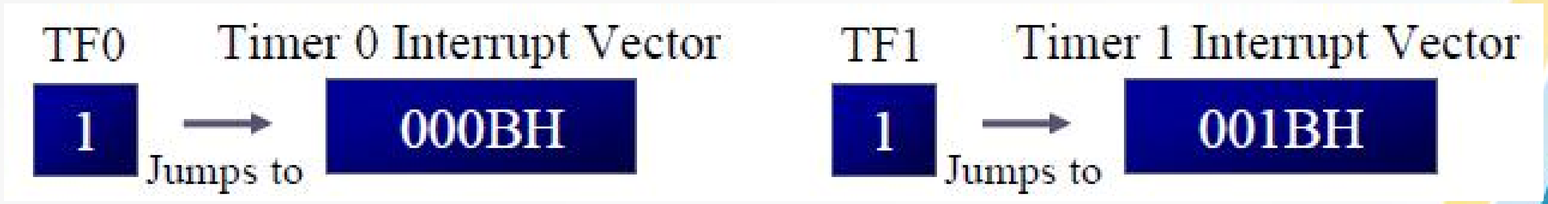
- To enable an interrupt, we take the following steps:
  - Bit D7 of the IE register (EA) must be set to high to allow the rest of register to take effect
  - The value of EA
    - If EA = 1, interrupts are enabled and will be responded to if their corresponding bits in IE are high
    - If EA = 0, no interrupt will be responded to, even if the associated bit in the IE register is high

Timer Interrupts

- The timer flag (TF) is raised when the timer rolls over
  - In polling TF, we have to wait until the TF is raised
    - The microcontroller is tied down while waiting for TF to be raised, and can not do anything else
  - Using interrupts to avoid tying down the controller
    - If the timer interrupt in the IE register is enabled, whenever the timer rolls over, TF is raised

Timer Interrupts (cont.)

- The microcontroller is interrupted in whatever it is doing, and jumps to the interrupt vector table to service the ISR
- In this way, the microcontroller can do other until it is notified that the timer has rolled over



External Hardware Interrupts

- The 8051 has two external hardware interrupts
  - Pin 12 (P3.2) and pin 13 (P3.3) of the 8051
    - Designated as INT0 and INT1
    - Used as external hardware interrupts
  - The interrupt vector table locations 0003H and 0013H are set aside for INT0 and INT1
  - There are two activation levels for the external hardware interrupts
    - Level trigged

Level-Triggered Interrupt

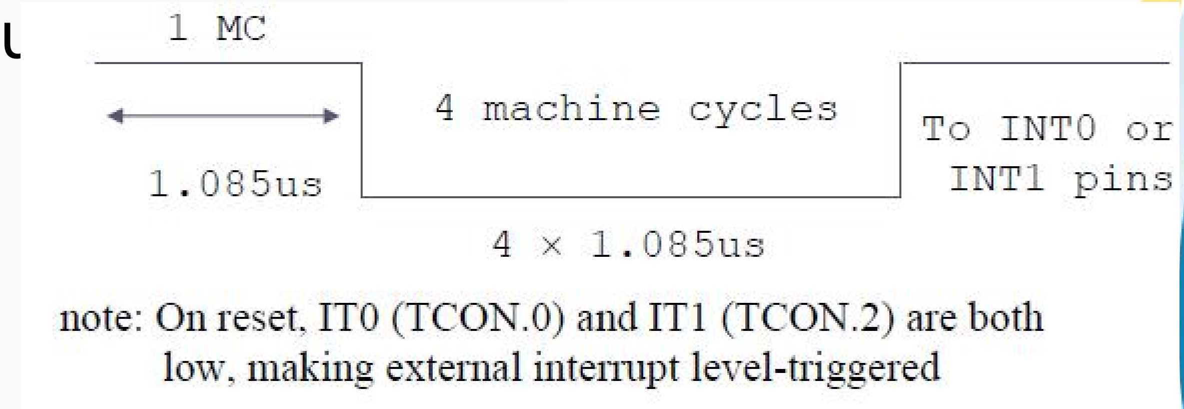
- INT0 and INT1 pins are normally high
  - If a low-level signal is applied to them, it triggers the interrupt
    - The microcontroller stops whatever it is doing and jumps to the interrupt vector table to service that interrupt
    - The low-level signal at the INT pin must be removed before the execution of the last instruction of the ISR, RETI
      - Otherwise, another interrupt will be generated
    - This is called a level-triggered or level-activated interrupt and is the default

Sampling Low Level-Triggered Interrupt

- P3.2 and P3.3 are used for normal I/O
  - Unless the INT0 and INT1 bits in the IE register are enabled
    - After the hardware interrupts are enabled, the controller keeps sampling the INTn pin for a low-level signal once each machine cycle
    - The pin must be held in a low state until the start of the execution of ISR
      - If the INTn pin is brought back to a logic high before the start of the execution of ISR, there will be no interrupt
    - If INTn pin is left at a logic low after the

Sampling Low Level-Triggered Interrupt (cont.)

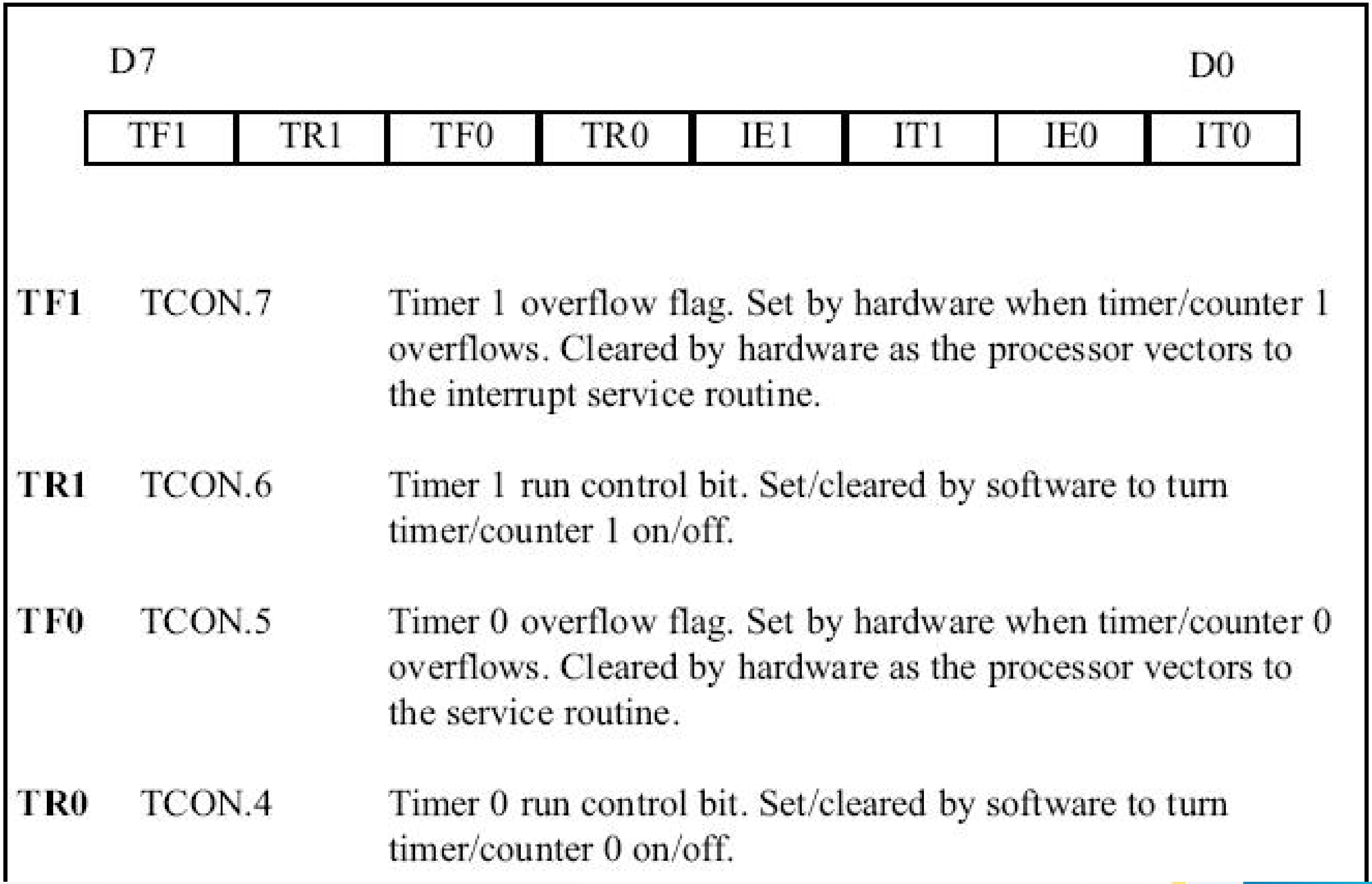
- To ensure the activation of the hardware interrupt at the INTn pin,
  - The duration of the low-level signal is around 4 machine cycles, but no more
    - This is due to the fact that the level-triggered interrupt is not latched
    - Thus the pin must be held in a low state u



note: On reset, IT0 (TCON.0) and IT1 (TCON.2) are both low, making external interrupt level-triggered

Edge-Triggered Interrupt

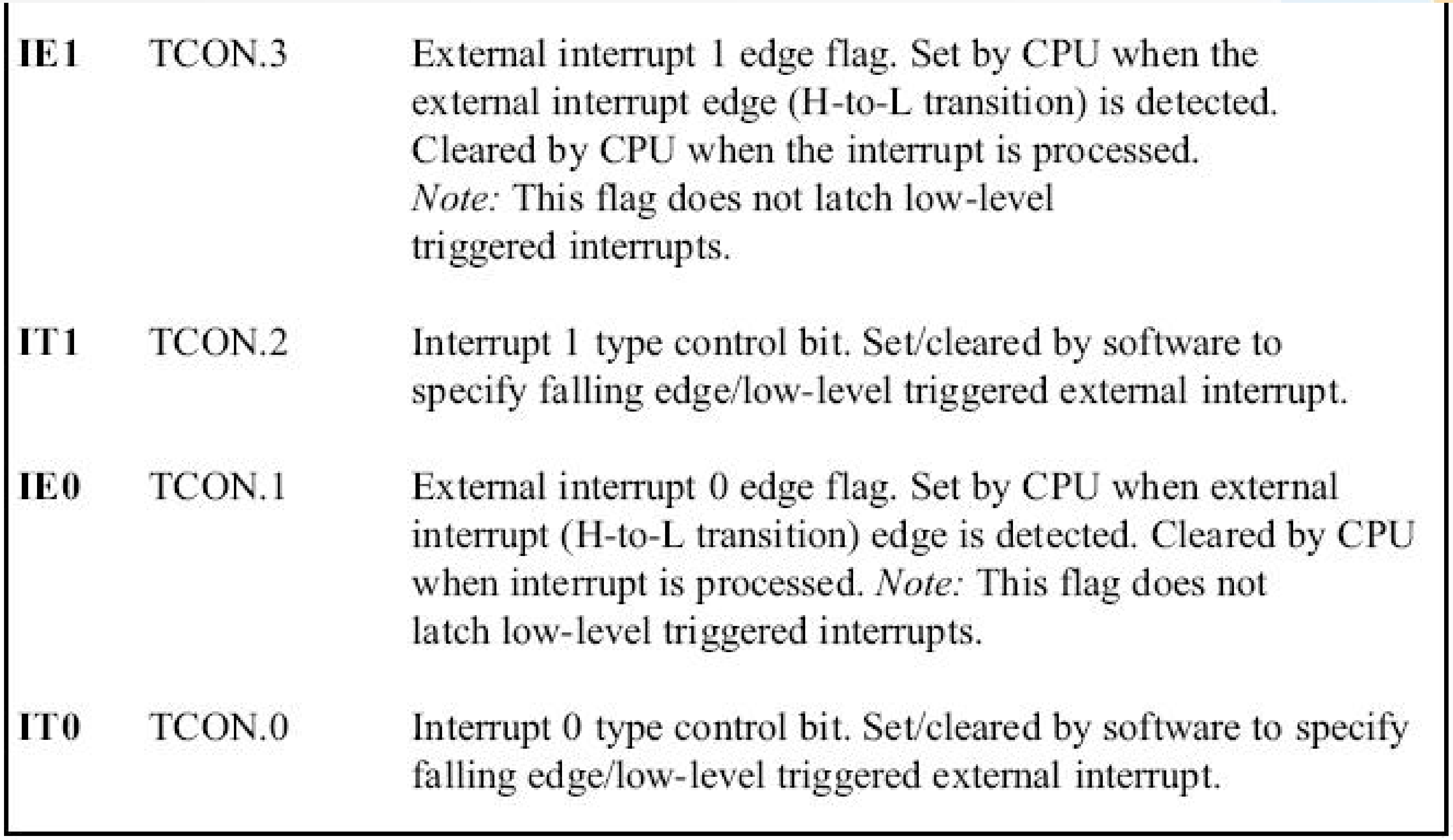
- To make INT0 and INT1 edge-triggered interrupts, we must program the bits of the TCON register
  - The TCON register holds the IT0 and IT1 flag bits that determine level- or edge-triggered mode of the hardware interrupt
    - IT0 and IT1 are bits D0 and D2 of TCON
      - They are also referred to as TCON.0 and TCON.2 since the TCON register is bit-addressable



D7 D0

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TF1	TCON.7	Timer 1 overflow flag. Set by hardware when timer/counter 1 overflows. Cleared by hardware as the processor vectors to the interrupt service routine.
TR1	TCON.6	Timer 1 run control bit. Set/cleared by software to turn timer/counter 1 on/off.
TF0	TCON.5	Timer 0 overflow flag. Set by hardware when timer/counter 0 overflows. Cleared by hardware as the processor vectors to the service routine.
TR0	TCON.4	Timer 0 run control bit. Set/cleared by software to turn timer/counter 0 on/off.

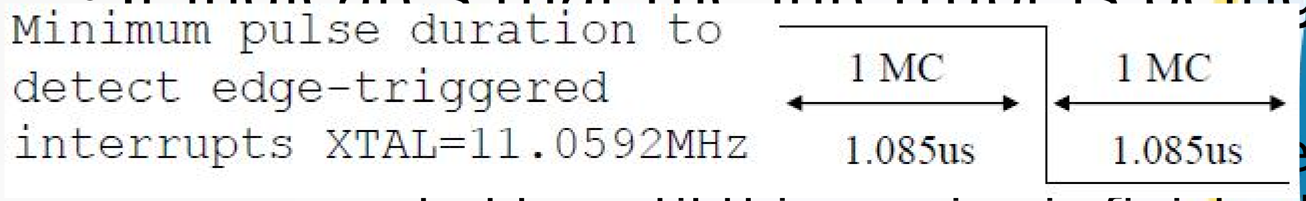


IE1	TCON.3	External interrupt 1 edge flag. Set by CPU when the external interrupt edge (H-to-L transition) is detected. Cleared by CPU when the interrupt is processed. *Note:* This flag does not latch low-level triggered interrupts.
IT1	TCON.2	Interrupt 1 type control bit. Set/cleared by software to specify falling edge/low-level triggered external interrupt.
IE0	TCON.1	External interrupt 0 edge flag. Set by CPU when external interrupt (H-to-L transition) edge is detected. Cleared by CPU when interrupt is processed. *Note:* This flag does not latch low-level triggered interrupts.
IT0	TCON.0	Interrupt 0 type control bit. Set/cleared by software to specify falling edge/low-level triggered external interrupt.

Figure 11-6. TCON (Timer/Counter) Register (Bit-addressable)

Sampling Edge-Triggered Interrupt

- The external source must be held high for at least one machine cycle, and then held low for at least one machine cycle
  - The falling edge of pins INT0 and INT1 are latched by the 8051 and are held by the TCON.1 and TCON.3 bits of TCON register
    - Function as interrupt-in-service flags
    - It indicates that the interrupt is being


Minimum pulse duration to detect edge-triggered interrupts XTAL=11.0592MHz
1 MC
1.085us
1 MC
1.085us

e

responded to until this service is finished

Sampling Edge-Triggered Interrupt (cont.)

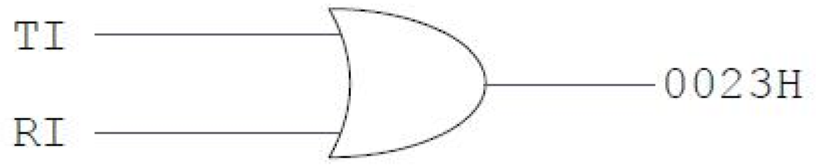
- When the ISRs are finished, TCON.1 and TCON.3 are cleared
  - The interrupt is finished and the 8051 is ready to respond to another interrupt on that pin
    - During the time that the interrupt service routine is being executed, the INTn pin is ignored, no matter how many times it makes a high-to-low transition
  - RETI clears the corresponding bit in TCON register (TCON.1 or TCON.3)
    - There is no need for instruction CLR

Serial Communication Interrupt

- TI (transfer interrupt) is raised when the stop bit is transferred
  - Indicating that the SBUF register is ready to transfer the next byte
- RI (received interrupt) is raised when the stop bit is received
  - Indicating that the received byte needs to be picked up before it is lost (overrun) by new incoming serial data

RI and TI Flags and Interrupts

- In the 8051 there is only one interrupt set aside for serial communication
  - Used to both send and receive data
  - If the interrupt bit in the IE register (IE.4) is enabled, when RI or TI is raised the 8051 gets interrupted and jumps to memory location 0023H to execute the ISR
    - In that ISR we must examine the TI and



Serial interrupt is invoked by TI or RI flags

Use of Serial COM in 8051

- The serial interrupt is used mainly for receiving data and is never used for sending data serially
  - This is like getting a telephone call in which we need a ring to be notified
  - If we need to make a phone call there are other ways to remind ourselves and there is no need for ringing
  - However in receiving the phone call, we must respond immediately no

Interrupt Flag Bits

- The TCON register holds four of the interrupt flags in the 8051
- The SCON register has the RI and TI flags

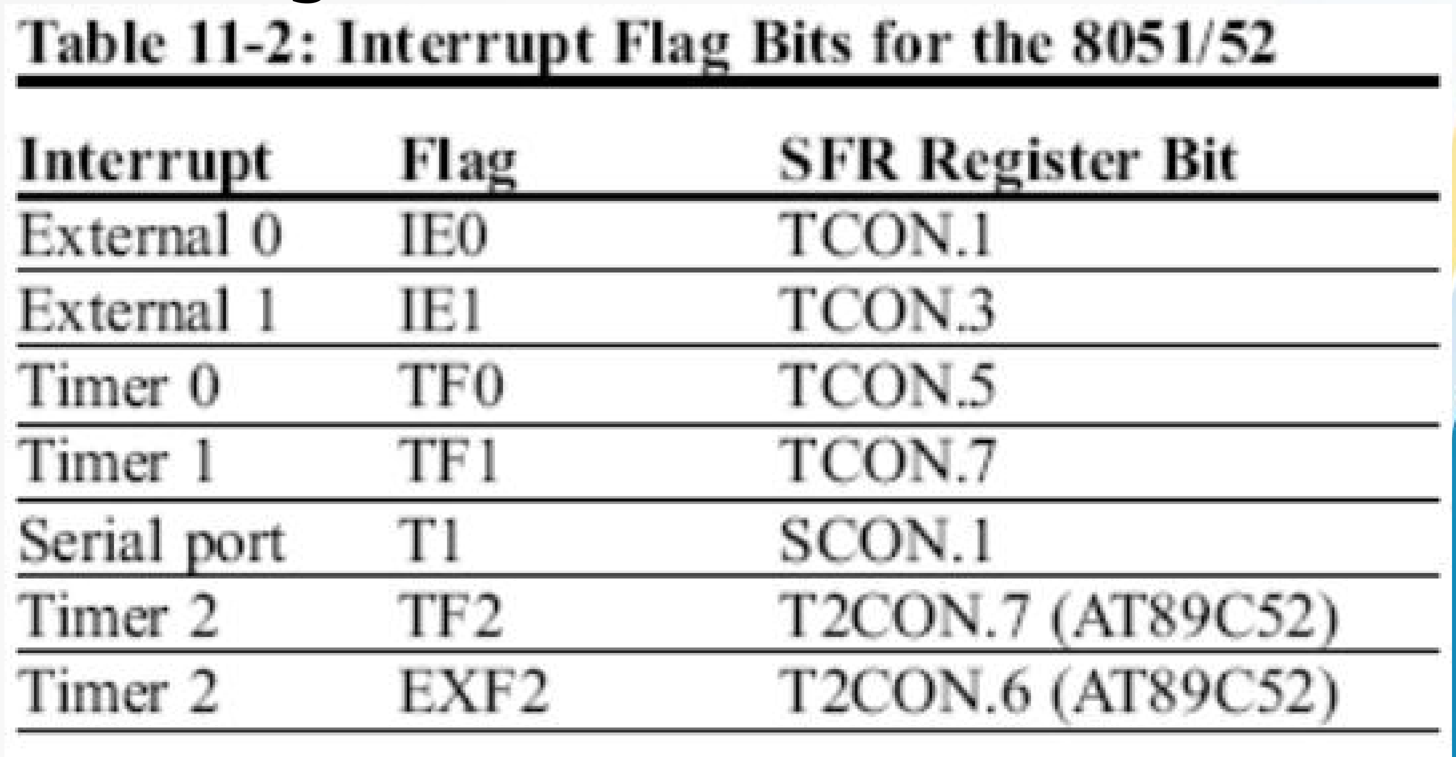


Table 11-2: Interrupt Flag Bits for the 8051/52

Interrupt	Flag	SFR Register Bit
External 0	IE0	TCON.1
External 1	IE1	TCON.3
Timer 0	TF0	TCON.5
Timer 1	TF1	TCON.7
Serial port	T1	SCON.1
Timer 2	TF2	T2CON.7 (AT89C52)
Timer 2	EXF2	T2CON.6 (AT89C52)

Interrupt Priority

- When the 8051 is powered up, the priorities are assigned
  - In reality, the priority scheme is nothing but an internal polling sequence in which the 8051 polls the interrupts in the sequence listed

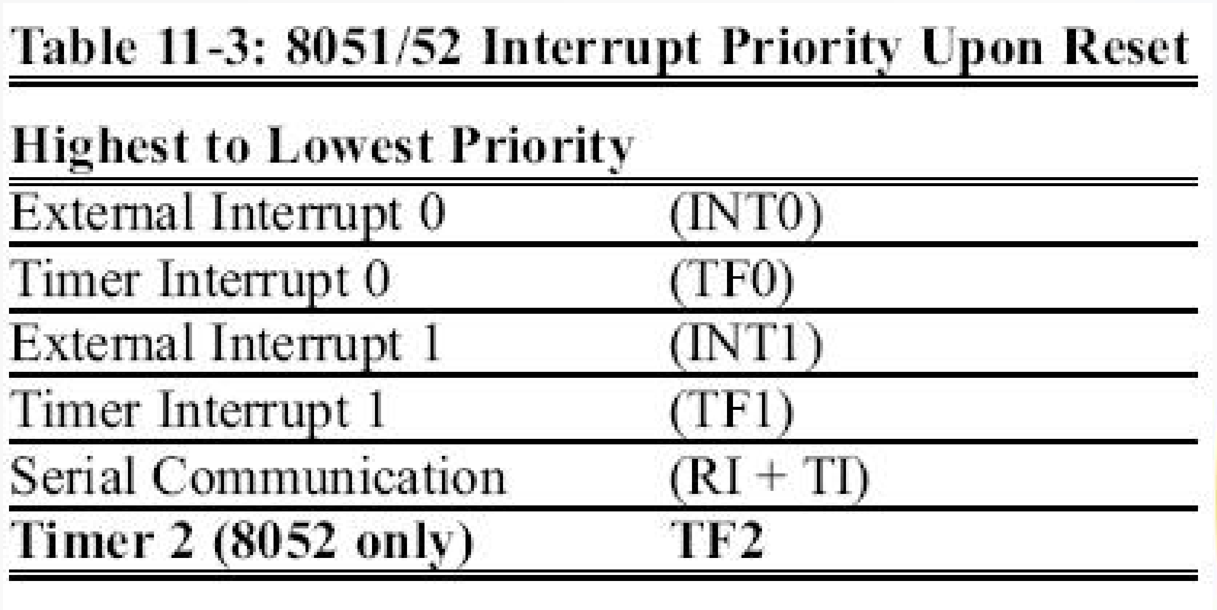


Table 11-3: 8051/52 Interrupt Priority Upon Reset

Highest to Lowest Priority	
External Interrupt 0	(INT0)
Timer Interrupt 0	(TF0)
External Interrupt 1	(INT1)
Timer Interrupt 1	(TF1)
Serial Communication	(RI + TI)
Timer 2 (8052 only)	**TF2**

Interrupt Priority Register (Bit-addressable)

D7							D0
--	--	PT2	PS	PT1	PX1	PT0	PX0

--	IP.7	Reserved
--	IP.6	Reserved
PT2	IP.5	Timer 2 interrupt priority bit (8052 only)
PS	IP.4	Serial port interrupt priority bit
PT1	IP.3	Timer 1 interrupt priority bit
PX1	IP.2	External interrupt 1 priority bit
PT0	IP.1	Timer 0 interrupt priority bit
PX0	IP.0	External interrupt 0 priority bit

Priority bit=1 assigns high priority
Priority bit=0 assigns low priority

Triggering Interrupt by Software

- To test an ISR by way of simulation can be done with simple instructions to set the interrupts high
  - Thereby cause the 8051 to jump to the interrupt vector table
  - ex. If the IE bit for timer 1 is set, an instruction such as SETB TF1 will interrupt the 8051 in whatever it is doing and will force it to jump to the interrupt vector table
    - We do not need to wait for timer 1 go roll over to have an interrupt

CONTENT BEYOND SYLLABUS

SL. NO.	**TOPIC**	**PO-PSO MAPPING**
1	80286 MICROPROCESSOR	PO1,PO2,PO3,PO4,PO5,PO6,PO7,PO8,PO9,PO10,PO12,PSO1,PSO2
2	STM32H7 MICROCONTROLLERS	PO1,PO2,PO3,PO4,PO5,PO6,PO7,PO8,PO9,PO10,PO12,PSO1,PSO2

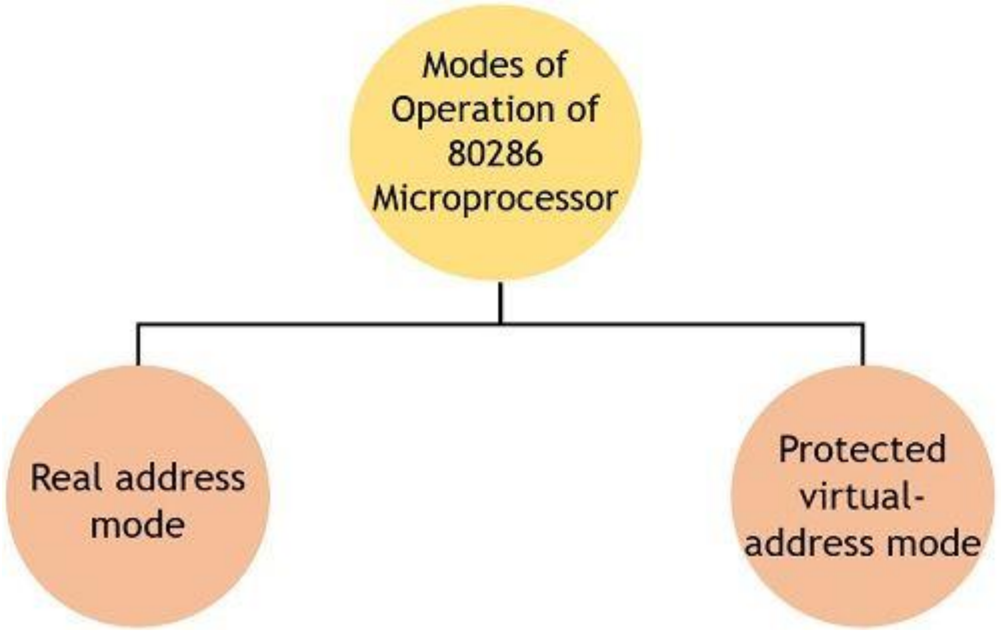
80286 MICROPROCESSOR

80286 Microprocessor is a 16-bit microprocessor that has the ability to execute 16-bit instruction at a time. It has non-multiplexed data and address bus. The size of data bus is 16-bit whereas the size of address bus is 24-bit.
It was invented in February 1982 by Intel. 80286 microprocessor was basically an advancement of 8086 microprocessor. Further in 1985, Intel produced upgraded version of 80286 which was a 32-bit microprocessor.
Now the question arises what are the factors that make 80286 more advantageous than 8086 microprocessor?

- It has non-multiplexed address and data bus that reduces operational speed.
- The addressable memory in case of 80286 is 16 MB.
- It offers an additional adder for address calculation.
- 80286 has faster multipliers that lead to quick operation.
- The performance per clock cycle of 80286 is almost twice when compared with 8086 or 8088.

Operating modes of 80286 microprocessor

80286 operates in two modes:


Modes of Operation of 80286 Microprocessor
Real address mode
Protected virtual-address mode

In real address mode, this microprocessor acts as a version of 8086 which is quite faster. Also without any special modification, the instruction programmed for 8086 can be executed in 80286. It offers memory addressability of 1 MB of physical memory.

The protected virtual-address mode of 80286 supports multitasking because multiple programs can be executed using virtual memory. This mode of 80286 offers memory addressability of 16 MB of physical memory along with 1 GB of virtual memory.
As using virtual memory, space for other programs can be saved. Sometimes bulky programs also do exist that cannot be stored in physical memory, so virtual memory is utilized in order to execute large programs.

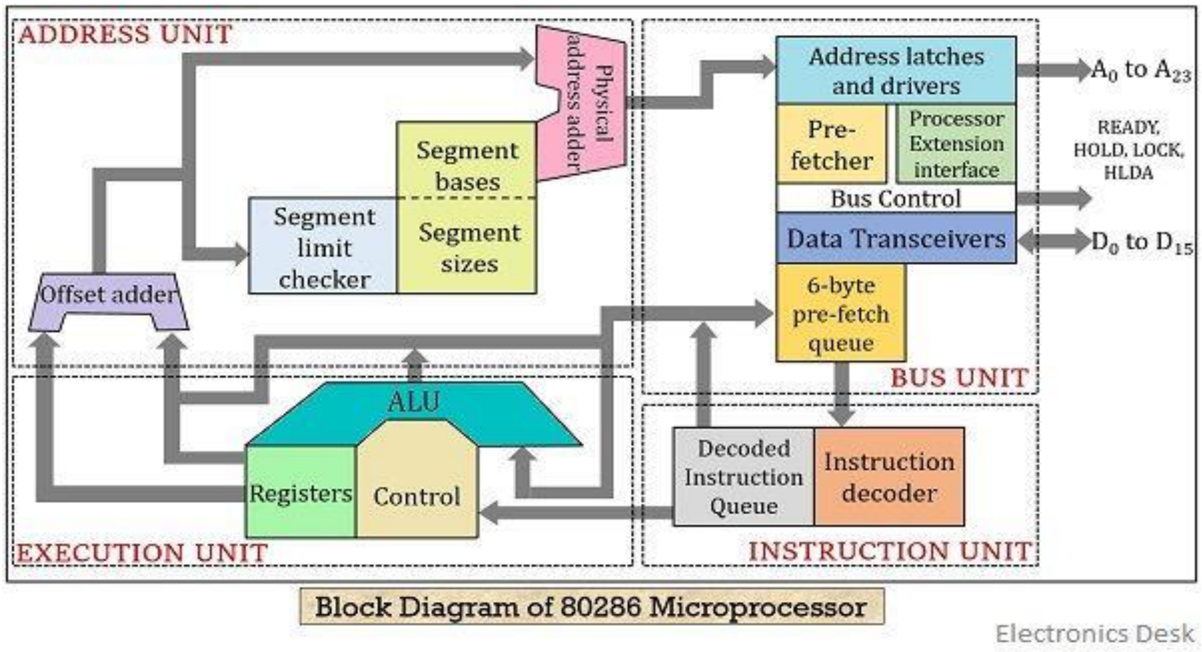
This mode is used in 80286, so that in case of memory failure in real address mode, it can stay in protected manner.

What is virtual memory?
Virtual memory is that part of hard disk which can be utilized for storing large instructions inside the system. This extra memory can be addressed by the computer other than the physical memory.

When there exists an instruction that is to be loaded in the memory but whose size is greater than the provided physical memory. Then some part of hard disk is used in order to store that instruction, which is known as virtual memory.

Architecture of 80286 Microprocessor

The figure below shows the architectural representation of 80286 microprocessor:



As

we have already mentioned earlier that it is a 16-bit microprocessor thus holds a 16-bit data bus and 24-bit address bus. Also, unlike the 8086 microprocessor, it offers non-multiplexed address and data bus, which increases the operating speed of the system.

80286 is composed of nearly around 125K transistors and the pin configuration has a total of 68 pins.

The CPU, central processing unit of 80286 microprocessor, consists of 4 functional block:

- Address Unit
- Bus Unit
- Instruction Unit
- Execution Unit

Firstly, the physical address from where the data or instruction is to be fetched is calculated, by the address unit. Once the physical address is calculated then the calculated address is handed over to the bus unit. More specifically we can say, that the calculated address is loaded on the address bus of the bus unit.
This address specifies the memory location from where the data or instruction is to be fetched. The fetching of data through the memory is done through the data bus.
For faster execution of instruction, the BU fetches the instructions in advanced from the memory and stores them in the queue.

This is done through the bus control module. As we have discussed that the prefetched instructions are stored in a 6-byte instruction queue. This instruction queue then further sends the instruction to the instruction unit.
The instruction unit on receiving the instructions now starts decoding the instruction. As instructions are stored in prefetched queue thus the decoder continuously decodes the fetched instructions and stores them into decoded instruction queue.

Now after the instructions gets decoded then further these are needed to be executed. So, the instructions from decoded instruction queue are fed to the execution unit. The main component of EU is ALU i.e., arithmetic and logic unit that performs the arithmetic and logic operations over the operand according to the decoded instruction.
Once the execution of the instruction is performed then the result of the operation i.e., the desired data is send to the register bank through the data bus.

As we have already discussed that 80286 is just a modified version of 8086. The register set in 80286 is same as that of 8086 microprocessor.

- It holds 8 general purpose registers of 16 bit each.
- It contains 4 segment register each of 16-bit.
- Also has status and control register and instruction pointer.

Interrupt of 80286 Microprocessor

We know that whenever an interrupt gets generated in a system, then the execution of the current program is stopped and the execution gets transferred to the new program location where the interrupt is generated.

But once the interrupt gets executed then then in order to get back to the original program, its address as well as machine state must be stored in the stack. Basically there exist 3 categories of interrupt in 80286 microprocessor:

- External interrupt (Hardware interrupt)
- INT instruction interrupt (Software interrupt)
- Internally generated interrupt due to some exceptions

External or hardware initiate interrupt are those interrupts that gets generated due to an external input. And are basically of two types:

1. Maskable interrupt
2. Non-maskable interrupt

Sometimes when multiple programs are allowed to be executed in a system, then this leads to generation of INT instruction, and such an interrupt is known as software interrupt.

Another interrupt in 80286 exist due to some unusual conditions or situations generated in the system that leads to prevention of further execution of the current instruction.

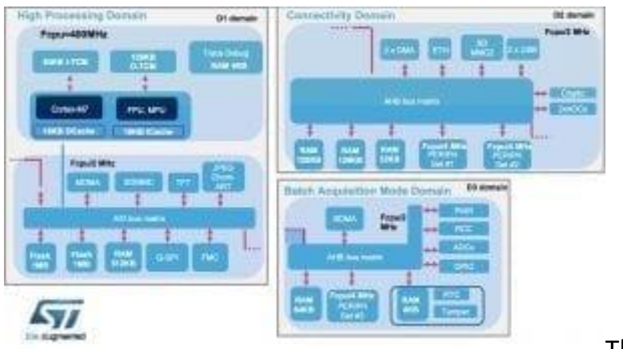
So, this is all about the modes of operation, architecture and interrupts of 80286 microprocessor.

STM32H7, the Most Powerful Cortex-M7 MCU, Breaks the 2000-point Threshold in CoreMark

The **STM32H7** series of microcontrollers (MCU) made history today by becoming the most powerful implementation of the ARM® Cortex®-M7 processor for the embedded market. It is more than twice as fast as the STM32F7 series, the previous STM32 flagship series, meaning that its core frequency of 400 MHz has enabled ST to become the first ever to reach 2010 points in CoreMark with a Cortex-M MCU.

This is possible because ST is the first to have shrunk its M7 implementation from a 90 nm process node to 40 nm. Media outlets have recently reported that some manufacturers have started or are about to start mass producing SoCs in 10 nm. However, it is important to understand that these components only have digital circuits, unlike the embedded MCU from ST, which includes digital circuitry, Flash memory, and analog components. Hence, these structures are much more complex than typical mainstream components and thus require more complex processes. Therefore, the 40 nm node used today is not only groundbreaking but the gateway to a masterful implementation of the Cortex-M7, and although we can’t enumerate all the great updates or optimizations found in the STM32H7 in one single post, we’ve decided to focus on some of the reasons why its performance sets new records.

Three Domains, Memory-Packed

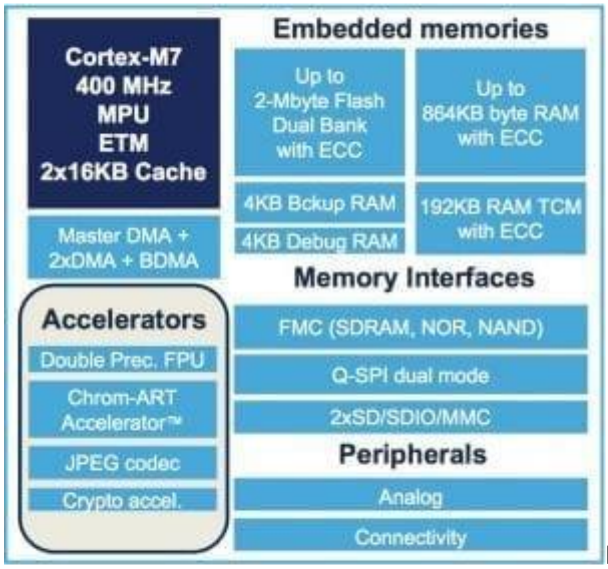


The Three Domains of the STM32H7 (Click to Enlarge)

To optimize the STM32H7, its architecture has been divided into three domains. Very simply, the first one (D1) includes the core with its cache, Flash memory and high bandwidth peripherals like the module to drive a screen or the Chrom-Art graphics engine. D2, the connectivity domain, groups low-speed peripherals like USB, the cryptographic accelerator and the SD/MMC2 unit for storage. Finally, D3, the batch acquisition mode domain, is responsible for some of the most fundamental aspects of the MCU like its reset and clock control as well as ADCs, GPIO, RTC, the chip's power management and a basic DMA (BDMA) controller.

This structure allowed ST to design a flexible and efficient architecture that packs a massive internal memory compared to some STM32F7 series. For instance, the L1 Cache is now four times bigger with 16 KB for instructions and the same amount for data. ST also included a total of 1 MB of SRAM and 2 MB of Flash, which is three times and twice as much respectively as the previous generation. However, instead of using a single block of SRAM, that would only benefit a certain domain, the STM32H7 placed various amounts at different locations to make the memory more versatile.

Concurrent Access



Memory Integration and Connectivity in the STM32H7

The D1 domain obviously holds the largest amount of SRAM. The core has a total of 192 KB of TCM SRAM (64 KB I-TCM, optimised for instructions and 128 KB D-TCM, optimised for data), which acts as an extension to the L1 cache. It has the same performance but is addressable. This means that TCM RAM can be accessed by the core with no latency and developers can specifically place information that needs to be deterministically retrieved to perform time-critical routines. The biggest chunk of SRAM (512 KB) is in the first domain because it contains the most computing intensive aspects of the architecture. Finally, D2 and D3 offer a quick access to their SRAM by the peripherals and other modules on the chip.

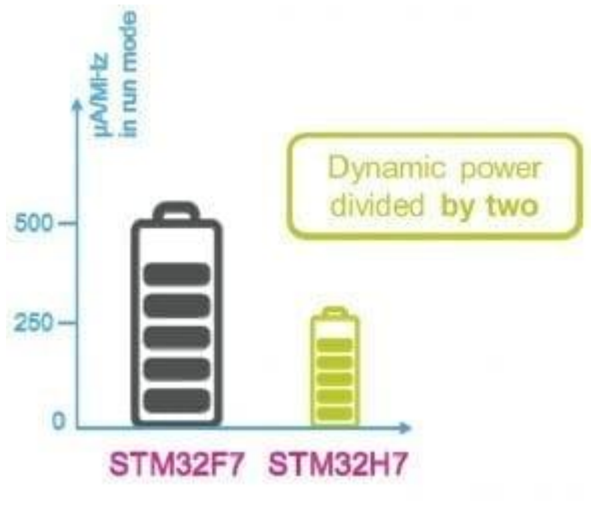
This organization has the great advantage of providing concurrent memory access, meaning that information can be fetched or stored in the different chunks of SRAM at the same time and by different domains, greatly improving the efficiency of the architecture. This is extremely important as embedded MCUs must often handle computationally intensive tasks, like running graphics and audio, while talking to an interface like a USB port to ensure that there is no disruption in the data transfer.

Optimized Memory and FPU

Another great feature stemming from the increased computational power of the STM32H7 series is the ability to use ECC SRAM and Flash. The speed increase compared to the STM32F7 series is so high that ST now has the computational resources to add error correction and still break performance records. By providing ECC, ST not only ensures data integrity, but also improves data retention in the Flash.

Another example of an architectural decision motivated by the needs of ST’s customers was the use of a double precision (FP64) floating point unit. The need for such a pipeline may not always be obvious, but some of the products that will benefit the most from the STM32H7 series need to perform DSP-type computations. For instance, an embedded system that monitors a power grid and will need to compute fast Fourier Transform algorithms, or a connected device that will run a precise GPS system will rely heavily on double precision computations.

Power Saving Features and So Much More



Power Savings in the STM32H7

It is impossible to offer a comprehensive list of all the features and optimizations brought by the STM32H7 series in a single blog post. We haven't even touched on the amazing power consumption optimizations that are offered by this three-domain architecture. For instance, it is possible to put D1 and D2 in a very low-powered standby mode (7µA) while D3 continues to capture data in its SRAM without needing to wake up the other domains, therefore greatly saving energy. There's also a complex and elaborate clock-control scheme to ensure that different parts of the architecture run at varying speeds in order to further improve the MCU's efficiency.

The STM32H7 series also builds on the previous generation by adding 10 more communication peripherals for a total of 35, it still offers cryptographic and hashing hardware acceleration, and remains pin to pin as well as software compatible with the STM32F7 series. The record-breaking STM32H7 series is sampling today to specific partners, and will be in mass production in Q2 2017. At this time, ST will have updated the mbed development platform to ensure developers can take full advantage of this groundbreaking architecture.