DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING

COURSE MATERIALS

ECT 206:COMPUTER ARCHITECTURE AND MICROCONTROLLER

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: 2002

♦ Course offered : B.Tech in Electronics and Communication Engineering

                M.Tech in VLSI

♦ Approved by AICTE New Delhi and Accredited by NAAC

♦ Affiliated to the University of Dr. A P J Abdul Kalam Technological University.

DEPARTMENT VISION

Providing Universal Communicative Electronics Engineers with corporate and social relevance towards sustainable developments through quality education.

DEPARTMENT MISSION

1)   Imparting Quality education by providing excellent teaching, learning environment.

2)   Transforming and adopting students in this knowledgeable era, where the electronic gadgets (things) are getting obsolete in short span.

3)   To initiate multi-disciplinary activities to students at earliest and apply in their respective fields of interest later.

4)   Promoting leading edge Research & Development through collaboration with academia & industry.

PROGRAMME EDUCATIONAL OBJECTIVES

PEOI. To prepare students to excel in postgraduate programmes or to succeed in industry / technical profession through global, rigorous education and prepare the students to practice and innovate recent fields in the specified program/ industry environment.

PEO2. To provide students with a solid foundation in mathematical, Scientific and engineering fundamentals required to solve engineering problems and to have strong practical knowledge required to design and test the system.

PEO3. To train students with good scientific and engineering breadth so as to comprehend, analyze, design, and create novel products and solutions for the real life problems.

PEO4. To provide student with an academic environment aware of excellence, effective communication skills, leadership, multidisciplinary approach, written ethical codes and the life-long learning needed for a successful professional career.

PROGRAM OUTCOMES (POS)

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
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.
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.
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.
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.
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.
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.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
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.
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.
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 OUTCOMES (PSO)

PSO1: Ability to Formulate and Simulate Innovative Ideas to provide software solutions for Real-time Problems and to investigate for its future scope.

PSO2: Ability to learn and apply various methodologies for facilitating development of high quality System Software Tools and Efficient Web Design Models with a focus on performance optimization.

PSO3: Ability to inculcate the Knowledge for developing Codes and integrating hardware/software products in the domains of Big Data Analytics, Web Applications and Mobile Apps to create innovative career path and for the socially relevant issues.

COURSE OUTCOMES
ECT 206

SUBJECT CODE: EC 308	
COURSE OUTCOMES	
ECT 206.1	Ability to understand basics of embedded system and to design an embedded system product.
ECT 206.2	Ability to understand the different standards and protocols used for communication with I/O devices.
ECT 206.3	Ability to distinguish different ways of communication with I/O devices.
ECT 206.4	Ability to understand basic programming concepts of Embedded Systems
ECT 206.5	Ability to understand about inter-process communication.

MAPPING OF COURSE OUTCOMES WITH PROGRAM OUTCOMES

CO'S	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
ECT 206.1	3											3
ECT 206.2	3											3
ECT 206.3	3											3
ECT 206.4	3				3							3
ECT 206.5	3	3			3							3
					3							3

CO'S	PSO1	PSO2	PSO3
ECT 206.1			
ECT 206.2	3		
ECT 206.3	3	3	2
ECT 206.4	3	3	2
ECT 206.5			
ECT 206	3	3	2

ECT 206	COMPUTER ARCHITECTURE AND MICROCONTROLLERS*	CATEGORY	L	T	P	CREDIT
		PCC	3	1	0	4

Preamble: This course aims to impart knowledge of basic computer architecture and modern microcontrollers.

Prerequisite: ECT203 Logic Circuit Design

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

CO 1	Explain the functional units, I/O and memory management w.r.t a typical computer architecture.
CO 2	Distinguish between microprocessor and microcontroller.
CO 3	Develop simple programs using assembly language programming.
CO 4	Interface 8051 microcontroller with peripheral devices using ALP/Embedded C
CO 5	Familiarize system software and Advanced RISC Machine Architecture.

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											3
CO 2	3											3
CO 3	3		3		3							3
CO 4	3	3	3		3							3
CO 5	3				3							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
Course project   : 15 marks

It is mandatory that a *course project* shall be undertaken by a student for this subject. The course project can be performed either as a hardware realization/simulation of a typical embedded system using Embedded C/ Assembly Language Programming. Instead of two assignments, two evaluations may be performed on the course project along with series tests, each carrying 5 marks. Upon successful completion of the project, a brief report shall be submitted by the student which shall be evaluated for 5 marks. The report has to be submitted for academic auditing. A few sample course projects are listed below:

Sample Course Projects
The below projects shall be done with the help of IDE for 8051/PIC/MSP/Arduino/Raspberry Pi-based interfacing boards/sensor modules.
1. Relay control
2. Distance measurement
3. Temperature measurement / Digital Thermometer
4. RF ID tags
5. Alphanumeric LCD display interface.
6. OLED display interfacing

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

SYLLABUS

Module 1: Computer Arithmetic and Processor Basics

Algorithms for binary multiplication and division. Fixed and floating-point number representation. Functional units of a computer, Von Neumann and Harvard computer architectures, CISC and RISC architectures. Processor Architecture – General internal architecture, Address bus, Data bus, control bus. Register set – status register, accumulator, program counter, stack pointer, general purpose registers. Processor operation – instruction cycle, instruction fetch, instruction decode, instruction execute, timing response, instruction sequencing and execution (basic concepts, datapath.

Module 2: 8051 Architecture

Microcontrollers and Embedded Processors. Architecture – Block diagram of 8051, Pin configuration, Registers, Internal Memory, Timers, Port Structures, Interrupts. Assembly Language Programming - Addressing Modes, Instruction set (Detailed study of 8051 instruction set is required).

Module 3: Programming and Interfacing of 8051

Simple programming examples in assembly language. Interfacing with 8051 using Assembly language programming: LED, Seven segment LED display. Programming in C - Declaring variables, Simple examples – delay generation, port programming, code conversion.

Interfacing of – LCD display, Keyboard, Stepper Motor, DAC and ADC -- with 8051 and its programming.

Module 4: Advanced Concepts

8051 Timers/Counters - Modes and Applications. Serial Data Transfer – SFRs of serial port, working, Programming the 8051 to transfer data serially. Introduction to ARM - ARM family, ARM 7 register architecture. ARM programmer's model. System software - Assembler, Interpreter, Compiler, Linker, Loader, Debugger.

Module 5: The Memory System

Types of memory - RAM, ROM. Memory Characteristics and Hierarchy. Cache memory – The basics of Caches, Mapping techniques, Improving Cache performance. Virtual memory – Overlay, Memory management, Address translation. Input/Output Organization – Introduction, Synchronous vs. asynchronous I/O, Programmed I/O, Interrupt driven I/O, Direct Memory Access.

Text Books

1. Muhammed Ali Mazidi & Janice Gilli Mazidi, R.D. Kinley, The 8051 microcontroller and Embedded System, Pearson Education, 2nd edition.

2. Subrata Ghoshal, Computer Architecture and Organization: From 8085 to Core2Duo and beyond, Pearson, 2011.

3. Steve Furber, ARM System - on-chip Architecture, Pearson Education

Reference Books

1. Mano M M, Computer System Architecture, 3rd Ed, Prentice Hall of India.

2. Computer organization and design: The Hardware/Software interface/David A. Patterson, John L. Hennessy. — 5th ed.

3. Computer Organisation V. Carl Hamacher, Zvonko G. Vranesic, Safwat G.Zaky.

4. John P Hayes, Computer Architecture and Organization, McGraw Hill.

5. Ramesh S Goankar, 8085 Microprocessor Architecture, Applications and Programming, Penram International, 5/e.

6. The 8051 Microcontrollers: Architecture Programming and Applications, K Uma Rao

Course Contents and Lecture Schedule

No	Topic	No. of Lectures

ELECTRONICS AND COMMUNICATION ENGINEERING

1	**Computer Arithmetic and Processor Basics**	
1.1	Algorithms for binary multiplication and division	2
1.2	Fixed- and floating-point number representation in computers.	1
1.3	Functional units of a computer, Von Neumann and Harvard computer architectures, CISC and RISC architectures.	1
1.4	Processor Architecture – General internal architecture, Address bus, Data bus, control bus. Register set – status register, accumulator, program counter, stack pointer, general purpose registers.	2

2	**8051 Architecture**	
2.1	Microcontrollers and Embedded Processors and Applications	1
2.2	Architecture – Block diagram of 8051, Pin configuration, Registers, Internal Memory, Timers, Port Structures, Interrupts.	3
2.3	Addressing Modes of 8051	1
2.4	Instruction sets (Detailed study of 8051 instructions)	4
3	**Programming and Interfacing of 8051**	
3.1	Simple programming examples in assembly language.	2
3.2	Interfacing programming in Assembly language	2
3.3	Programming in C - Declaring variables, Simple examples – delay generation, port programming, code conversion.	3
3.4	Interfacing of 7 segment LCD display	1
3.5	Interfacing of Keyboard and stepper motor	2
3.6	Interfacing of DAC and ADC	2
4	**Advanced Concepts**	
4.1	8051 Timers/Counters - Modes and Applications	2
4.2	Serial Data Transfer – SFRs of serial port, working, Programming the 8051 to transfer data serially	2
4.3	Introduction to ARM - ARM family, ARM 7 register architecture. ARM programmer's model	2
4.4	System software - Assembler, Interpreter, Compiler, Linker, Loader, Debugger.	2
5	**Memory System**	
5.1	Types of memory - RAM, ROM. Memory Characteristics and Hierarchy	2
5.2	Cache memory – The basics of Caches, Mapping techniques, Improving Cache performance	2
5.3	Virtual memory – Overlay, Memory management, Address translation	2
5. 4	Input/Output Organization – Introduction, Synchronous vs. asynchronous I/O, Programmed I/O, Interrupt driven I/O, Direct Memory Access.	3

Simulation assignments

The following examples may be solved in C program

1. Program to convert the ASCII number into unpacked BCD.
2. Program to swap a number 0 x ab to 0 x ba, where a and b are hex digits.
3. Program to find the number of 1's in an 8-bit data item.
4. Program to display 'M' and 'E' on the LCD connected to 8051 using the BUSY FLAG.
5. Program to rotate a stepper motor 50^0 in the clock wise direction.
6. Program to toggle pin P1.4 every second using interrupts for a frequency of 22 MHz. Use timer 1 in mode 1.
7. Program to generate a square wave of 1 kHz with duty cycle 33%. Use timer 1 in interrupt mode with a crystal frequency of 11.0592 MHz.

QUESTION BANK

MODULE I

Q:NO:	QUESTIONS	CO	KL	PAGE NO:
1	Explain the functional units of a computer system with diagram	CO	K2	2
2	Draw the general computer architecture explain shortly.	CO1	K3	4
3	Compare Von Newman and Harward architecture	CO1	K3	6
4	Discuss about typical characteristics RISC architecture	CO1	K2	10
5	Compare RISC & CISC architecture	CO1	K3	12
6	Explain the basic function of aprocessor	CO1	K2	13
7	Discuss about Advanced RISC MACHINE	CO1	K2	11
8	Explain about external signal used in a processor	CO1	K2	15
9	Addressing of memory location by the processor explain with neat diagram	CO1	K2	17
10	Explain different buses used in a processor	CO1	K2	18
11	Explain processor operation with appropriate diagrams	CO1	K2	13

MODULE II

1	List the components of 8051 microcontroller	CO2	K3	25
2	Which is more important for functioning of a basic processor,program counter,stack pointer.Justify your answer	CO2	K5	41
3	Explain the salient features of 8051 microcontroller	CO2	K2	35
4	Explain 8051 microcontroller with neat diagram	CO2	K2	27
5	Discuss detail about program status word register	CO2	K2	28

	with neat diagram			
6	Explain memory organization of 8051 microcontroller	CO2	K2	48
7	Write a short note about stack	CO2	K2	51
8	Draw the diagram of IE & IP register explain its briefly	CO2	K2	53
9	Explain about timer registers	CO2	K2	52
10	**Explain in detail about interrupts**	CO2	K2	53
11	Explain about timer mode operation	CO2	K2	54

MODULE III

1	Explain about assembler directives	CO3	K1	78
2	**Write a program to add the values of locations 50H and 51H and store the result in locations in 52h and 53H.**	CO3	K5	79
3	**Write a program to store data FFH into RAM memory locations 50H to 58H using indirect addressing mode.**	CO3	K5	79
4	**Write a program to clear 10 RAM locations starting at RAM address 1000H.**	CO3	K5	81
5	**Write a program to find the average of five 8 bit numbers. Store the result in H. (Assume that after adding five 8 bit numbers, the result is 8 bit only).**	CO3	K5	81
6	**Write a program to find the cube of an 8 bit number program is as follows**	CO3	K5	82
7	**Write a program to count the number of and o's of 8 bit data stored in location 6000H.**	CO3	K5	83
8	**Write a program to shift a 24 bit number stored at 57H-55H to the left logically four places. Assume that the**	CO3	K5	83

	least significant byte of data is stored in lower address.			
9	Explain how to interface seven segment display with 8051	CO3	K2	83
10	Explain about embedded c	CO3	K2	90
MODULE IV

1	Explain TMOD with detailed diagram	CO4	K3	127
2	With diagram explain /counter control logic	CO4	K2	128
3	Explain timer mode of operation	CO4	K2	128
4	Explain the types of serial communication	CO4	K2	131
5	Explain about seriel communication modes	CO4	K2	132
6	Explain about RS-232 Standards	CO4	K2	134
7	Write a program for the 8051 to transfer letter 'A' serially at 4800- baud rate, 8 bit data, 1 stop bit continuously	CO4	K5	135
8	Write a program for the 8051 to transfer the message 'EARTH' serially at 9600 baud, 8 bit data, 1 stop bit continuously.	CO4	K3	136
9	Explain the features of ARM Processors	CO4	K2	138
10	Draw the architecture of ARM processor with explain in detail.	CO4	K2	143
11	Explain about current program status register	CO4	K1	144
12	Explain about ARM instruction set	CO4	K2	146

MODULE V

1	Explain different types of memory system	CO5	K3	151
2	Compare RAM & ROM	CO5	K2	154
3	Compare SRAM & DRAM	CO5	K3	155

4	Explain about memory hierarchy	CO5	K3	156
5	Explain about cache memory	CO5	K2	157
6	Explain different levels of memory	CO5	K3	162
7	Explain about cache mapping	CO5	K3	163
8	Explain about types of cache	CO5	K2	163
9	Explain about virtual memory	CO5	K2	164

APPENDIX 1

CONTENT BEYOND THE SYLLABUS

S:NO;	TOPIC	PAGE NO:
1	Embedded system	179

.

.

MODULE 1
COMPUTER ARITHMETIC AND PROCESSOR BASICS

Syllabus:

Algorithms for binary multiplication and division. Fixed and floating-point number representation. Functional units of a computer, Von Neumann and Harvard computer architectures, CISC and RISC architectures. Processor Architecture – General internal architecture, Address bus, Data bus, control bus. Register set – status register, accumulator, program counter, stack pointer, general purpose registers. Processor operation – instruction cycle, instruction fetch, instruction decode, instruction execute, timing response, instruction sequencing and execution (basic concepts, data-path).

1.1 FUNCTIONAL UNITS

A computer consists of five functionally independent main parts: input, memory, arithmetic and logic, output, and control units, as shown in Figure 1.1.
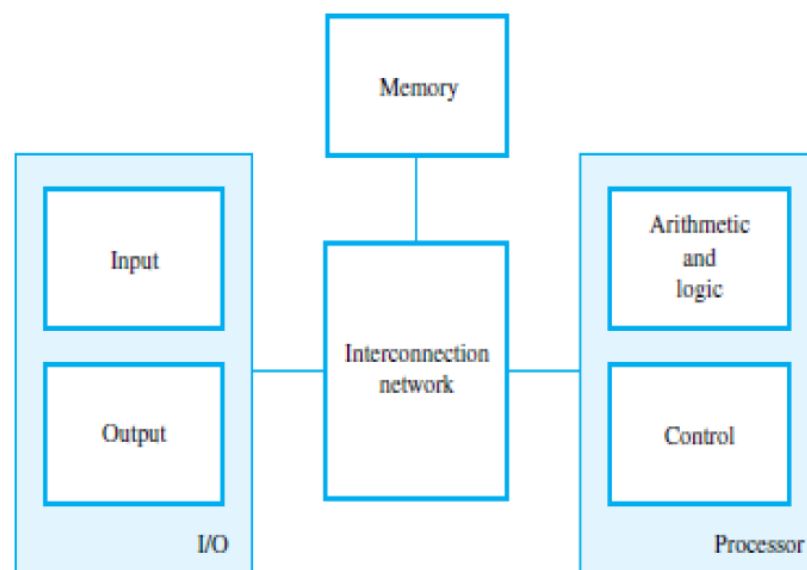
Fig 1.1 Basic Functional units of a computer

The input unit accepts coded information from human operators using devices such as keyboards, or from other computers over digital communication lines. The information received is stored in the computer‟s memory, either for later use or to be processed immediately by the arithmetic and logic unit. The processing steps are specified by a program that is also stored in the memory. Finally, the results are sent back to the outside world through the output unit. All of these actions are coordinated by the control unit. An interconnection network provides the means for the functional units to exchange information and coordinate

their actions. The arithmetic and logic circuits, in conjunction with the main control circuits, is the processor. Input and output equipment is often collectively referred to as the input-output (I/O) unit. A program is a list of instructions which performs a task. Programs are stored in the memory. The processor fetches the program instructions from the memory, one after another, and performs the desired operations. The computer is controlled by the stored program, except for possible external interruption by an operator or by I/O devices connected to it. Data are numbers and characters that are used as operands by the instructions. Data are also stored in the memory. The instructions and data handled by a computer must be encoded in a suitable format. Each instruction, number, or character is encoded as a string of binary digits called bits, each having one of two possible values, 0 or 1, represented by the two stable states.

Input Unit:

 Computers accept coded information through input units. The most common input device is the keyboard. Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted to the processor. Many other kinds of input devices for human-computer interaction are available, including the touchpad, mouse, joystick, and trackball. These are often used as graphic input devices in conjunction with displays. Microphones can be used to capture audio input which is then sampled and converted into digital codes for storage and processing. Similarly, cameras can be used to capture video input. Digital communication facilities, such as the Internet, can also provide input to a computer from other computers and database servers. **Memory Unit** The function of the memory unit is to store programs and data. There are two classes of storage, called primary and secondary. **Primary Memory** Primary memory, also called main memory, is a fast memory that operates at electronic speeds. Programs must be stored in this memory while they are being executed. The memory consists of a large number of semiconductor storage cells, each capable of storing one bit of information. These cells are rarely read or written individually. Instead, they are handled in groups of fixed size called words. The memory is organized so that one word can be stored or retrieved in one basic operation. The number of bits in each word is referred to as the word length of the computer, typically 16, 32, or 64 bits. To provide easy access to any word in the memory, a distinct address is associated with each word location. Addresses are consecutive numbers, starting from 0, that identify successive locations. Instructions and data can be written into or read from the memory under the control of the processor. A memory in which any location can be accessed in a short and fixed amount of time after specifying its address is called a random-access memory (RAM). The time required to access one word is called the memory access time. This time is independent of the location of the word being accessed. It typically ranges from a few nanoseconds (ns) to about 100 ns for current RAM units **Cache Memory** As an adjunct to the main memory, a smaller, faster RAM unit, called a cache, is used to hold sections of a program that are currently being executed, along with any associated data. The cache is tightly coupled with the processor and is usually contained on the same integrated-circuit chip. The purpose of the cache is to facilitate high instruction execution rates. At the start of program execution, the cache is empty. As execution proceeds, instructions are fetched into the processor chip, and a copy of each is placed in the cache. When the execution of an instruction requires data, located in the main

memory, the data are fetched and copies are also placed in the cache. If these instructions are available in the cache, they can be fetched quickly during the period of repeated use. **Secondary Storage** Although primary memory is essential, it tends to be expensive and does not retain information when power is turned off. Thus additional, less expensive, permanent secondarystorage is used when large amounts of data and many programs have to be stored, particularly for information that is accessed infrequently. Access times for secondary storage are longer than for primary memory. The devices available are including magnetic disks, optical disks (DVD and CD), and flash memory devices.

Arithmetic and Logic Unit Most computer operations are executed in the arithmetic and logic unit (ALU) of the processor. Any arithmetic or logic operation, such as addition, subtraction, multiplication division, or comparison of numbers, is initiated by bringing the required operands into the processor, where the operation is performed by the ALU. When operands are brought into the processor, they are stored in high-speed storage elements called registers. Each register can store one word of data. Access times to registers are even shorter than access times to the cache unit on the processor chip. **Output Unit** Output unit function is to send processed results to the outside world. A familiar example of such a device is a printer. Most printers employ either photocopying techniques, as in laser printers, or ink jet streams. Such printers may generate output at speeds of 20 or more pages per minute. However, printers are mechanical devices, and as such are quite slow compared to the electronic speed of a processor. Some units, such as graphic displays, provide both an output function, showing text and graphics, and an input function, through touchscreen capability. The dual role of such units is the reason for using the single name input/output (I/O) unit in many cases. **Control Unit** The memory, arithmetic and logic, and I/O units store and process information and perform input and output operations. The operation of these units must be coordinated in some way. This is the responsibility of the control unit. The control unit is effectively the nerve center that sends control signals to other units and senses their states. I/O transfers, consisting of input and output operations, are controlled by program instructions that identify the devices involved and the information to be transferred. Control circuits are responsible for generating the timing signals that govern the transfers. They determine when a given action is to take place. Data transfers between the processor and the memory are also managed by the control unit through timing signals. A large set of control lines (wires) carries the signals used for timing and synchronization of events in all units. The operation of a computer can be summarized as follows: • The computer accepts information in the form of programs and data through an input unit and stores it in the memory. • Information stored in the memory is fetched under program control into an arithmetic and logic unit, where it is processed. • Processed information leaves the computer through an output unit. • All activities in the computer are directed by the control unit. **Von Neumann architecture** In the 1940s, a mathematician called John Von Neumann described the basic arrangement (or architecture) of a computer. Most computers today follow the concept that he described although there are other types of architecture. A Von Neumann-based computer is a computer that: Uses a single processor. Uses one memory for both instructions and data. A von Neumann computer cannot distinguish between data and instructions in a memory location! It „knows" only because of the location of a particular bit

pattern in RAM. Executes programs by doing one instruction after the next in a serial manner using a fetch-decode-execute cycle.

1.2 COMPUTER ARCHITECTURE

Computer Architecture refers to the internal design of a computer with its CPU, which includes:

- ❖ Arithmetic and logic unit,
- ❖ Control unit,
- ❖ Registers,
- ❖ Memory for data and instructions,
- ❖ Input/output interface and
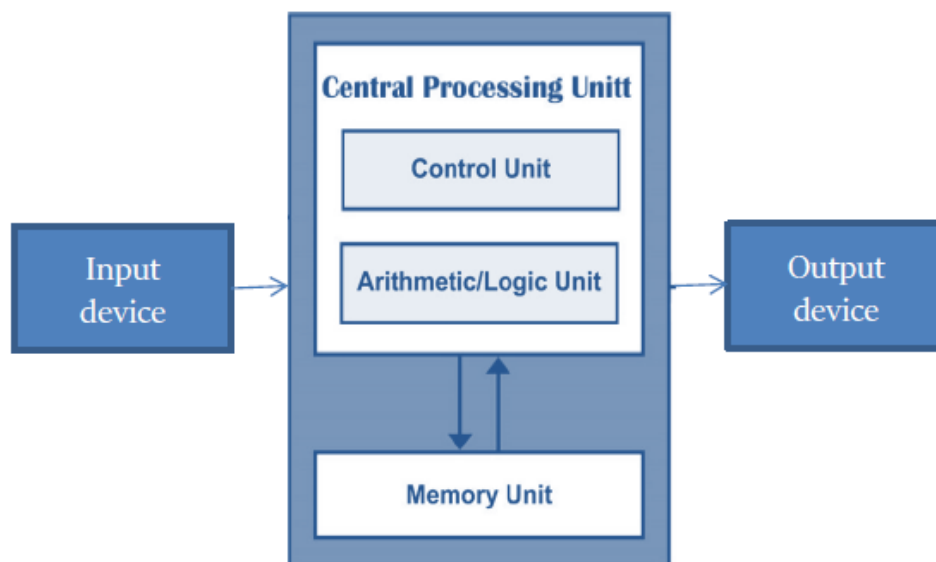- ❖ External storage functions.

Fig 1.2 : General Architecture

VON-NEUMANN ARCHITECTURE:

The same memory and bus are used to store both Data and Instructions.

The main drawback:
CPU is unable to access program memory and data memory simultaneously. This case is called the "**bottleneck**" that affects system performance.
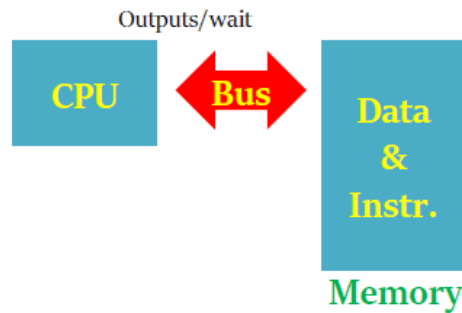
Fig 1.3: Von-Neumann architecture

The bottleneck

 ❖ If a Von-Neumann machine wants to perform an instruction (already fetched from the memory) on some data in memory, it has to move the data across the bus into the CPU.

 ❖ When the computation is done, it needs to move outputs of the computation to memory across the same bus; this operation will be completed if the bus is not used by another operation to fetch another instruction or data from the shared memory; otherwise the outputs of the computation has to wait.

HARVARD ARCHITECTURE:

The Harvard architecture stores machine instructions and data in separate memory units using different buses.

The main advantage:

 ❖ Computers designed with the Harvard architecture are able to run a program and access data independently, and therefore simultaneously.

Harvard architecture is more complicated but separate pipelines remove the **bottleneck** that Von-Neumann creates.

MODIFIED HARVARD ARCHITECTURE

The majority of modern computers have no physical separation between the memory spaces used by both data and instructions, therefore could be described technically as Von-Neumann. But as they have two separate address spaces, different buses and special instructions that keep data from being mistaken for code, this architecture is called "Modified Harvard Architecture".
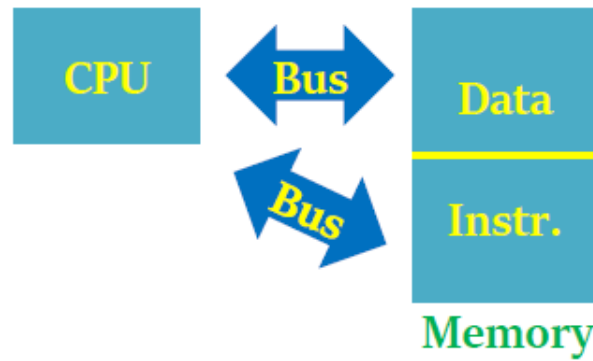
Fig 1.4: Harvard Architecture

Ex. some initial data values or constants can be accessed by the running program directly from instruction memory without taking up space in data memory.

1.3 RISC AND CISC ARCHITECTURES

General

The dominant architecture in the PC market belongs to the Complex Instruction Set Computer (CISC) design. The obvious reason for this classification is the "complex" nature of its Instruction Set Architecture (ISA). The motivation for designing such complex instruction sets is to provide an instruction set that closely supports the operations and data structures used by Higher-Level Languages (HLLs). However, the side effects of this design effort are far too serious to ignore.

Addressing Modes in CISC

The decision of CISC processor designers to provide a variety of addressing modes leads to variable-length instructions. For example, instruction length increases if an operand is in memory as opposed to in a register.

- ❖ This is because we have to specify the memory address as part of instruction encoding, which takes many more bits.
- ❖ This complicates instruction decoding and scheduling. The side effect of providing a wide range of instruction types is that the number of clocks required to execute instructions varies widely.
- ❖ This again leads to problems in instruction scheduling and pipelining.

Evolution of RISC

For these and other reasons, in the early 1980s designers started looking at simple ISAs. Because these ISAs tend to produce instruction sets with far fewer instructions, they coined the term Reduced Instruction Set Computer (RISC). Even though the main goal was not to reduce the number of instructions, but the complexity, the term has stuck.

There is no precise definition of what constitutes a RISC design. However, we can identify certain characteristics that are present in most RISC systems.

- ❖ We identify these RISC design principles after looking at why the designers took the route of CISC in the first place.
- ❖ Because CISC and RISC have their advantages and disadvantages, modern processors take features from both classes. For example, the PowerPC, which follows the RISC philosophy, has quite a few complex instructions.
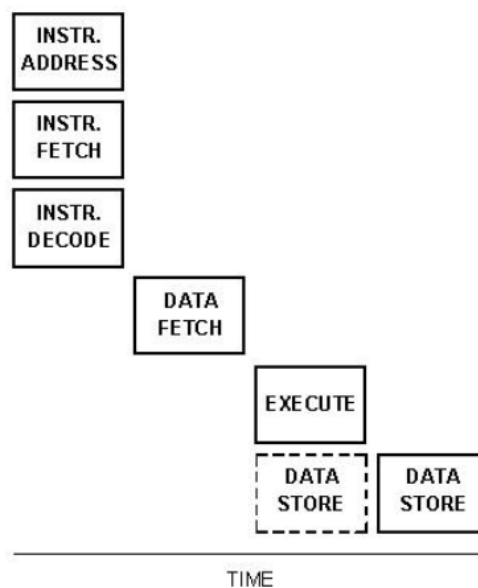
Fig 1.5: **Typical RISC Architecture based Machine - Instruction phase overlapping**

Definition of RISC

RISC, or Reduced Instruction Set Computer is a type of microprocessor architecture that utilizes a small, highly-optimized set of instructions, rather than a more specialized set of instructions often found in other types of architectures.

Evolution/History.

The first RISC projects came from IBM, Stanford, and UC-Berkeley in the late 70s and early 80s. The IBM 801, Stanford MIPS, and Berkeley RISC 1 and 2 were all designed with a similar philosophy which has become known as RISC. Certain design features have been characteristic of most RISC processors

- ➕ **One Cycle Execution Time**. RISC processors have a CPI (clock per instruction) of one cycle. This is due to the optimization of each instruction on the CPU and a technique called ;
- ➕ **Pipelining**. A technique that allows for simultaneous execution of parts, or stages, of instructions to more efficiently process instructions;
- ➕ **Large Number of Registers**. The RISC design philosophy generally incorporates a larger number of registers to prevent in large amounts of interactions with memory
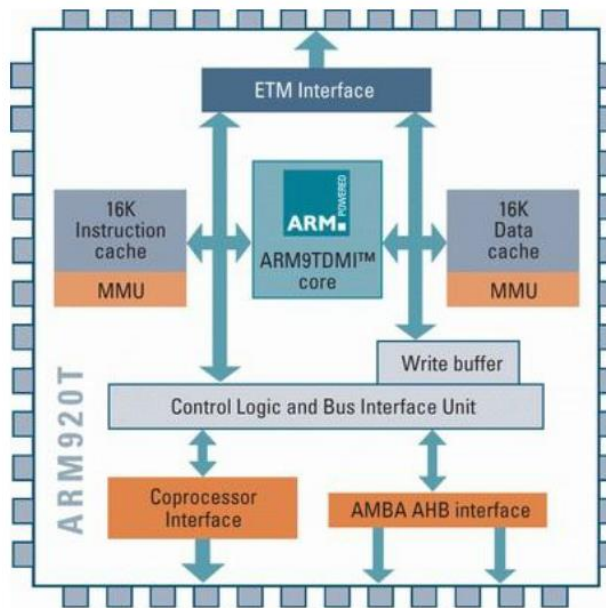
Fig 1.6 **Advanced RISC Machine (ARM)**

Non RISC Design or Pre RISC Design

In the early days of the computer industry, programming was done in assembly language or machine code, which encouraged powerful and easy to use instructions. CPU designers therefore tried to make instructions that would do as much work as possible. With the advent of higher level languages, computer architects also started to create dedicated instructions to directly implement certain central mechanisms of such languages.
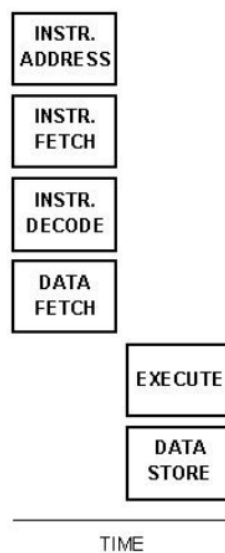

Fig 1.7: **Typical CISC Architecture** – Stack Design

Another general goal was to provide every possible addressing mode for every instruction, known as orthogonality, to ease compiler implementation. Arithmetic operations could therefore often have results as well as operands directly in memory (in addition to register or immediate).

The attitude at the time was that hardware design was more mature than compiler design so this was in itself also a reason to implement parts of the functionality in hardware and/or microcode rather than in a memory constrained compiler (or its generated code) alone. This design philosophy became retroactively termed Complex Instruction Set Computer (CISC) after the RISC philosophy came onto the scene.

An important force encouraging complexity was very limited main memories (on the order of kilobytes). It was therefore advantageous for the density of information held in computer programs to be high, leading to features such as highly encoded, variable length instructions, doing data loading as well as. These issues were of higher priority than the ease of decoding such instructions.

An equally important reason was that main memories were quite slow (a common type was ferrite core memory); by using dense information packing, one could reduce the frequency with which the CPU had to access this slow resource. Modern computers face similar limiting factors: main memories are slow compared to the CPU and the fast cache memories employed to overcome this are instead limited in size. This may partly explain why highly encoded instruction sets have proven to be as useful as RISC designs in modern computers.

TYPICAL CHARACTERISTICS OF RISC ARCHITECTURE

Designers make choices based on the available technology. As the technology, both hardware and software, evolves, design choices also evolve. Furthermore, as we get more experience in designing processors, we can design better systems. The RISC proposal was a response to the changing technology and the accumulation of knowledge from the CISC designs. CISC processors were designed to simplify compilers and to improve performance under constraints such as small and slow memories. The important observations that motivated designers to consider alternatives to CISC designs were

- **Simple Instructions**. The designers of CISC architectures anticipated extensive use of complex instructions because they close the semantic gap. In reality, it turns out that compilers mostly ignore these instructions. Several empirical studies have shown that this is the case. One reason for this is that different high-level languages use different semantics. For example, the semantics of the C for loop is not exactly the same as that in other languages. Thus, compilers tend to synthesize the code using simpler instructions.
- **Few Data Types**. CISC ISA tends to support a variety of data structures, from simple data types such as integers and characters to complex data structures such as records and structures. Empirical data suggest that complex data structures are used relatively infrequently. Thus, it is beneficial to design a system that supports a few simple data types efficiently and from which the missing complex data types can be synthesized.
- **Simple Addressing Modes**. CISC designs provide a large number of addressing modes. The main motivations are

  (1) To support complex data structures and

  (2) To provide flexibility to access operands.

(a) Problems Caused. Although this allows flexibility, it also introduces problems. First, it causes variable instruction execution times, depending on the location of the operands.

(b) Second, it leads to variable-length instructions. For example, the IA-32 instruction length can range from 1 to 12 bytes. Variable instruction lengths lead to inefficient instruction decoding and scheduling.

- **Identical General Purpose Registers**. Allowing any register to be used in any context, simplifying compiler design (although normally there are separate floating point registers).
- **Harvard Architecture Based**. RISC designs are also more likely to feature a Harvard memory model, where the instruction stream and the data stream are conceptually separated; this means that modifying the memory where code is held might not have any effect on the instructions executed by the processor (because the CPU has a separate instruction and data cache), at least until a special synchronization instruction is issued. On the upside, this allows both caches to be accessed simultaneously, which can often improve performance.

RISC VS CISC – AN EXAMPLE

The simplest way to examine the advantages and disadvantages of RISC architecture is by contrasting it with its predecessor, CISC (Complex Instruction Set Computers) architecture.

Multiplying Two Numbers in Memory.

The main memory is divided into locations numbered from (row) 1: (column) 1 to (row) 6: (column) 4. The execution unit is responsible for carrying out all computations. However, the execution unit can only operate on data that has been loaded into one of the six registers (A, B, C, D, E, or F). Let's say we want to find the product of two numbers - one stored in location 2:3 and another stored in location 5:2 - and then store the product back in the location 2:3
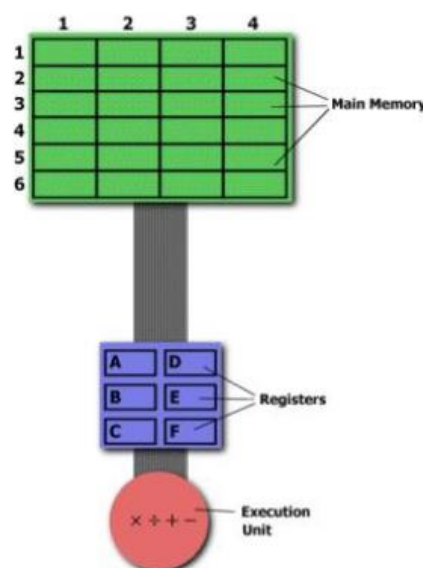

Fig 1.8: : Representation **of Storage Scheme for a Generic Computer**

The CISC Approach. The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible. This is achieved by building processor hardware that is capable of understanding and executing a series of operations. For this particular task, a CISC processor would come prepared with a specific instruction (say "MUL").

- When executed, this instruction loads the two values into separate registers, multiplies the operands in the execution unit, and then stores the product in the appropriate register.
- Thus, the entire task of multiplying two numbers can be completed with one instruction:

MUL 2:3, 5:2

- MUL is what is known as a "complex instruction."
- It operates directly on the computer's memory banks and does not require the programmer to explicitly call any loading or storing functions.
- It closely resembles a command in a higher level language. For instance, if we let "a" represent the value of 2:3 and "b" represent the value of 5:2, then this command is identical to the C statement "a = a x b."

Advantage.

One of the primary advantages of this system is that the compiler has to do very little work to translate a high-level language statement into assembly. Because the length of the code is relatively short, very little RAM is required to store instructions. The emphasis is put on building complex instructions directly into the hardware.

The RISC Approach. RISC processors only use simple instructions that can be executed within one clock cycle. Thus, the "MUL" command described above could be divided into three separate commands:

- "LOAD," which moves data from the memory bank to a register,
- "PROD," which finds the product of two operands located within the registers, and
- "STORE," which moves data from a register to the memory banks.

In order to perform the exact series of steps described in the CISC approach, a programmer would need to code four lines of assembly:

LOAD A, 2:3
LOAD B, 5:2
PROD A, B
STORE 2:3, A

Analysis. At first, this may seem like a much less efficient way of completing the operation. Because there are more lines of code, more RAM is needed to store the assembly level

instructions. The compiler must also perform more work to convert a high-level language statement into code of this form.

❖ Advantage of RISC. However, the RISC strategy also brings some very important advantages. Because each instruction requires only one clock cycle to execute, the entire program will execute in approximately the same amount of time as the multi-cycle "MUL" command. These RISC "reduced instructions" require less transistors of hardware space than the complex instructions, leaving more room for general purpose registers. Because all of the instructions execute in a uniform amount of time (i.e. one clock), pipelining is possible.

a) Separating the "LOAD" and "STORE" instructions actually reduces the amount of work that the computer must perform.

b) After a CISC-style "MUL" command is executed, the processor automatically erases the registers. If one of the operands needs to be used for another computation, the processor must re-load the data from the memory bank into a register. In RISC, the operand will remain in the register until another value is loaded in its place.

The following table will differentiate both the architectures and based on the analysis the overall advantage will be discussed.

| CISC	RISC
Emphasis on hardware	Emphasis on software
Includes multi-clock complex instructions	Single-clock, reduced instruction only
Memory-to-memory: "LOAD" and "STORE" incorporated in instructions	Register to register: "LOAD" and "STORE" are independent instructions
Small code sizes, high cycles per second	Low cycles per second, large code sizes
Transistors used for storing complex instructions	Spends more transistors on memory registers

Table **Comparison of CISC and RISC Architectures**

❖ CISC Approach. The CISC approach attempts to minimize the number of instructions per program, sacrificing the number of cycles per instruction.
❖ RISC Approach. RISC does the opposite, reducing the cycles per instruction at the cost of the number of instructions per program.

1.4 PROCESSOR ARCHITECTURE

GENERAL INTERNAL ARCHITECTURE

The general internal architecture of any processor is shown in Figure. Apart from the arithmetic logic unit (ALU), which performs all arithmetic and logical operations, every processor offers a set of general purpose registers for various storage and operations. Its status register accommodates the status of different arithmetic and logical operations, which might be necessary for conditional branching. Its program counter holds the address of next instruction word to be fetched from external memory. The stack pointer indicates the address of the stack-top.
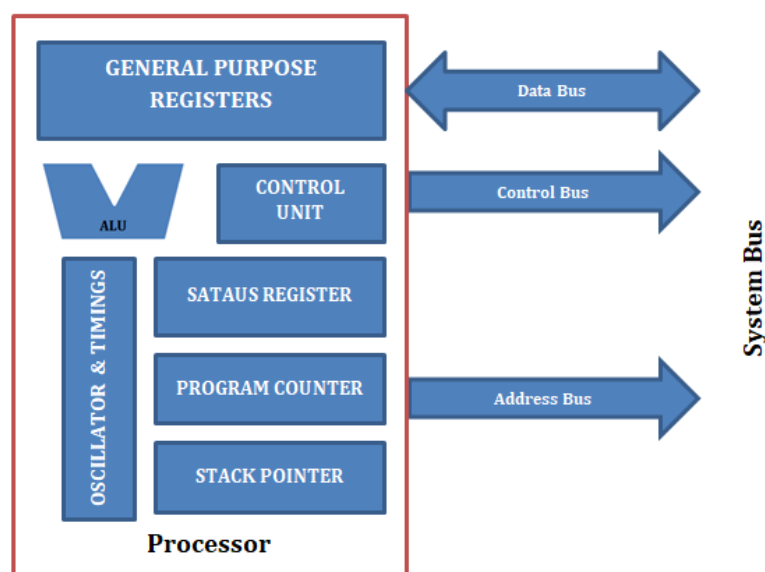
Fig 1.9 : General Internal Architecture of a processor

Two more architectural features of any processor are indicated in Figure. They are control unit and, oscillator and timing module. The control unit is responsible for generating all control signals and general working of the processor. This is achieved by the instructions with the help of internal clock, which is maintained by the oscillator unit.

BASIC FUNCTION

These architectural details of a processor are meant for executing a software. We should always remember that hardware and software must be dealt concurrently for any computer or any processor. Only hardware or only software would not be able to achieve any tangible outcome. ***The basic duty of any processor is to fetch, decode and execute instructions as long as it is powered on***. Unless it is a microcontroller, these instructions are available outside the physical boundary of the processor, within memory chips (ICs). These memory chips are electrically connected with the processor through a bunch of wires, designated as the bus.

| Fetch Instruction | Decode Instruction | Execute Instruction |
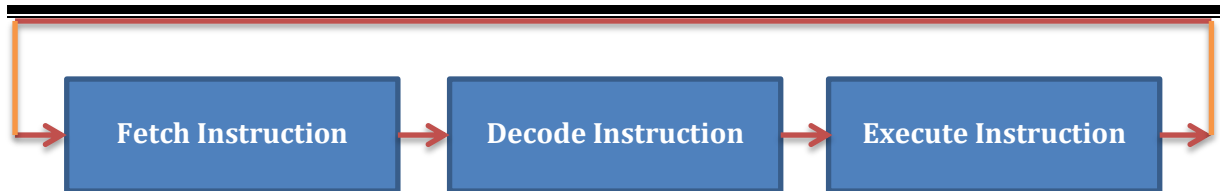
Fig 1.10: Basic Function of any Processor

Apart from fetching the opcode of executable instructions, sometimes it might be necessary for the processor to load or store operands in the external memory, if indicated so by the ongoing instruction Generally, the operation of a processor is sequential, which is diverted to another sequence due to conditional branching, subroutine calls and returning from subroutines, or to respond against any eventual external interrupt signal.

PERIPHERAL DEVICES AND EXTERNAL COMMUNICATION

Memory devices are not the only category of peripheral devices necessary for a processor to be functional. Later in this chapter, we shall see that a large number of peripheral devices are used to support different duties, as per the system requirements. These devices (non-memory devices) are, generally, referred as Input/output devices or I/O devices. Note that, just like memory devices, these I/O devices are also connected (or interfaced) with the processor through the bus.

Every processor offers three major types of bus.
- Address bus
- Data bus
- Control bus.

Out of these three bus, data bus is bi-directional, as data must come in and also go out of the processor, depending upon the specific requirements. Address bus is always unidirectional and it carries address signals from the processor to all external devices around it, memory and I/O. Most of the control signals also move out of the processor. Schematically, a generic processor's external signals are presented in Figure.
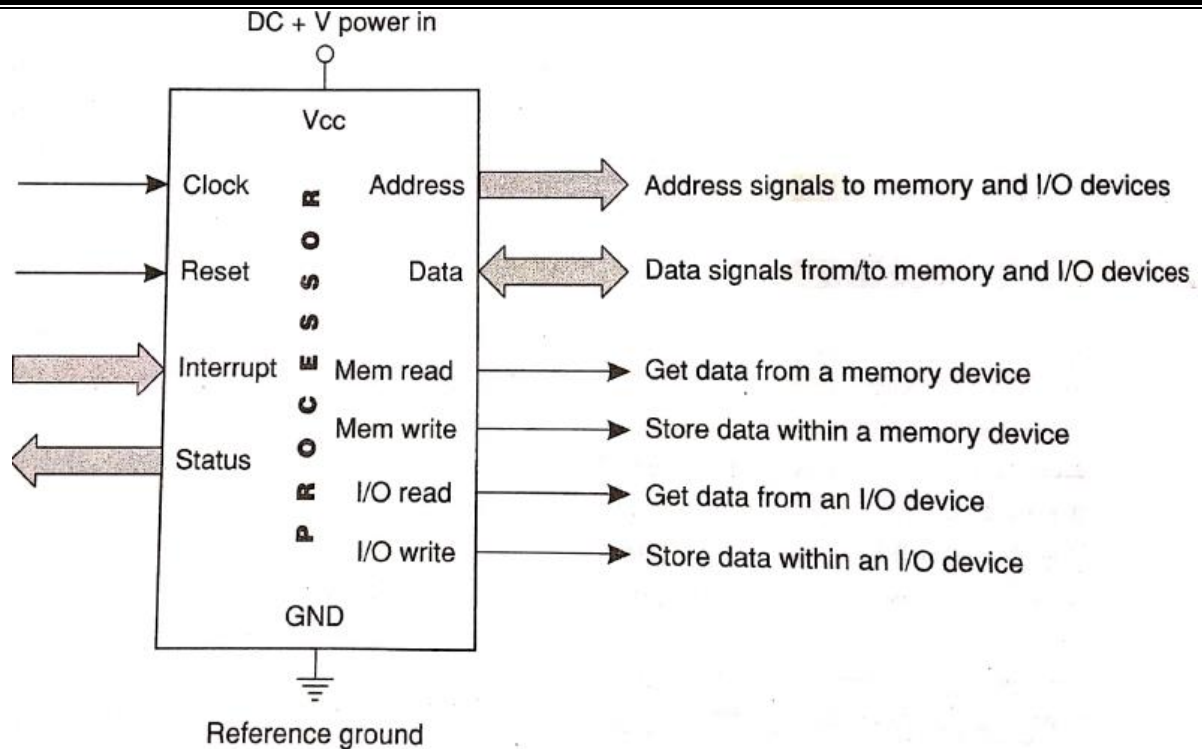
Fig 1.11; External signals of a generic processor

ADDRESS BUS AND ADDRESSING

The width of address bus or number of address lines available from any processor indicates its maximum memory size handling capability. The number of memory locations (bytes or words as the case might be) addressable by n address lines is 2^n. Therefore, if the processor offers 16 address lines then it can address 2^{16} or 64K locations (1 K = 2^{10} = 1,024). If it is offering 20 address lines then it can address 2^{20} or 1M locations and so on.

We have already discussed how the address bus helps in locating any desired data with any memory or I/O device. These address signals are decoded by a decoder inside the memory or I/O device to target the desired location.
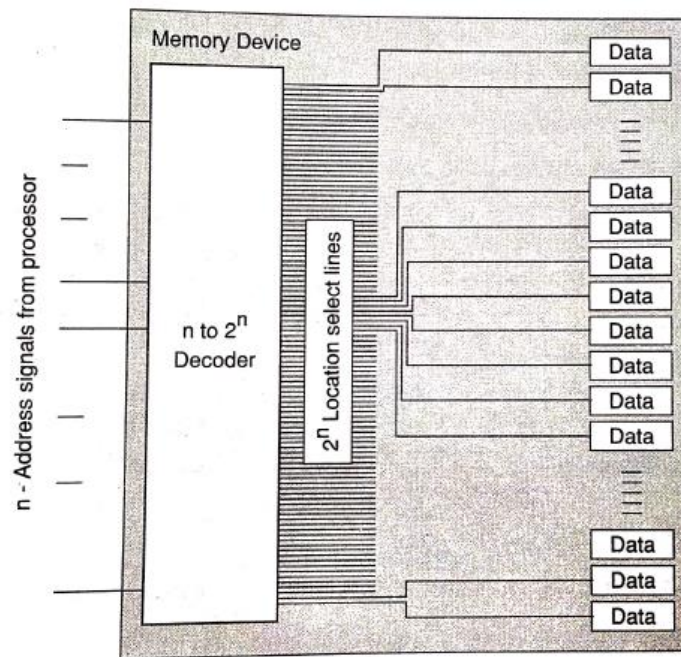
Fig 1.12 : Addressing of memory location by the processor

DATA BUS AND DATA FLOW CONTROL

The width of data bus of any processor indicates its simultaneous handling capability of the maximum number of bits. Generally, a processor is designated by its data bus width. For example, an 8-bit processor is capable of communicating 8-bit of data at the same time or having an 8-bit data bus. Similarly, a 16-bit processor has 16 parallel data lines for data communications.

The flow of data is bi-directional, depending upon whether the processor is interested in reading from or writing into the device (memory or I/O). This intension of the processor is expressed through its control signals (read and write). Depending upon this indication (read or write), the device (memory or I/O) enables the appropriate 3-state buffer to allow the flow of data signals from the data location already selected by address signals. The identical type of 3-state buffers is also present at the processor end in its data bus. This is illustrated through Figure, using 1-byte (8-bits) of storage area. Note that D0-D7 represents the data bus, interfaced with the processor. Eight flip-flops are for storage of data (8-bits) and at the input and output of each flip-flop, tri-state buffers are provided, whose control inputs are connected in parallel. The location-select signal from the decoder within the memory IC along with memory read or memory write signal enable these buffers. The clock signal acts in conjunction with memory write and select signals for the storage operation.
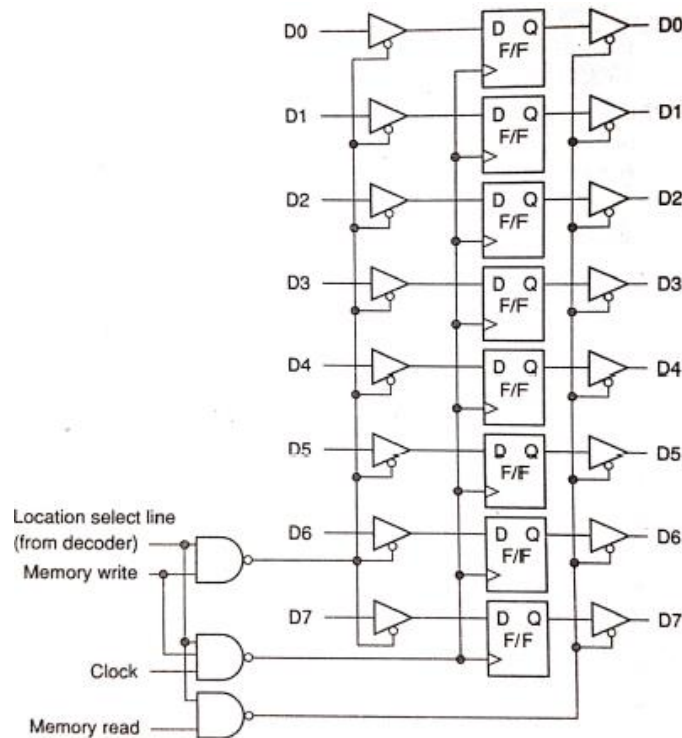
Fig 1.13 : Data flow mechanism between memory & processor

In Figure , the reader should note that although separate lines indicate input-to and output-from the tri-state buffers, the designation for any pair of buffers is the same, i.e., either both are DO or both are D1. Externally, these data line pairs are connected together to form a bus of 8 data lines, i.e., DO-D7.

CONTROL BUS

Number and functions of control signals, constituting the control bus varies widely with the processor itself. However, two of its important signals are READ and WRITE. Condition of these control signals indicate whether the present operation, intended by the processor is expecting the data in (READ) or sending the data out (WRITE). As the processor is to interact with two types of devices, memory and I/O, in general, four read/write signals are offered, as shown in Figure 1.11 . A few status signals are also available from the processor, apart from power input signals. Two more input signals are essential for all processors, namely clock and reset. Apart from these, a few external interrupt input signals are also provided in all processors.

The purpose of all these signals is to execute any program. The programs are composed of individual instructions. We shall now discuss how this program execution is implemented by the processor during its operational stage.

1.5 PROCESSOR OPERATION

The job of the processor is to execute programs, which are composed of multiple instructions. At this point, we should remember that instructions executable by the processor, are always in the machine code. Programs developed with high level language (HLL) instructions are first changed to this machine code, understandable by the processor. In general, the machine code instructions are extremely primitive and simple, e.g.,

❖ Copy a data byte from external memory to internal register or vice versa.
❖ Add two numbers available within the processor registers.
❖ If the result of subtraction is zero, then skip next three (or three thousand three hundred thirty three) instructions.

These instructions must be present in binary form within the memory of the system.

INSTRUCTION CYCLE
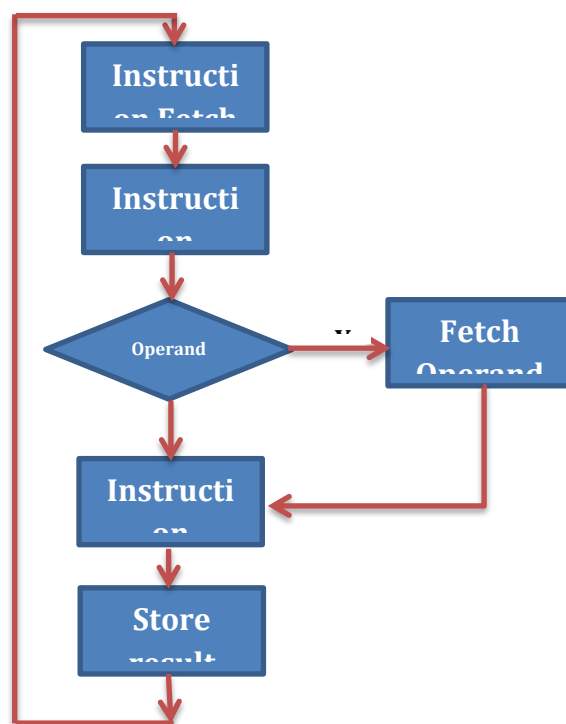
Fig 1.14 : Flow chart for simplified instruction cycle

To execute any type of instruction including those that are cited above, the processor should perform the following steps

↓ Fetch
↓ Decode

➕ Execute

Combination of these three steps is known as Instruction cycle. A flow chart of simplified form of instruction cycle of a generic processor is shown in figure

It may be observed from the flowchart that after fetching the instruction in the form of its opcode and decoding it, the processor checks for any eventual operand fetch, which might be necessary for some (not for all) instructions. If found necessary, then the operand is fetched from memory and then the instruction is executed. Finally, the result of the instruction is stored and the whole cycle is repeated. At this point, the reader may ask a question why this is designated as a simplified instruction cycle ? The answer is, we are avoiding many other details related with the instruction cycle, e.g., checking for any interrupt signal or looking for any direct memory access (DMA) request and so on. At a later stage, we shall consider all these details of the instruction cycle. We shall now discuss about the details of these three stages and some more related aspects.

INSTRUCTION FETCH

The first step, as indicated before, is to fetch the instruction byte(s) from external memory. This external memory is a vast area containing many bytes of instructions. Therefore, the processor must pin-point the correct location of this large memory area to extract the target byte.

It was already indicated that every memory location (byte in majority of cases) has a unique binary address. After receiving this address, the duty of the memory device is to decode the address to locate the target byte and place it on the data bus, so that the content of that address is available for the processor
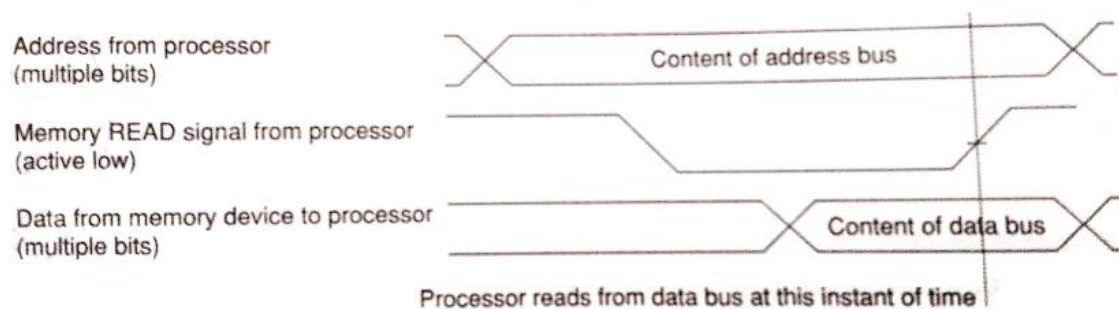
Fig 1.15 : Timing diagram for Instruction Fetch

Therefore, for the purpose of instruction fetch, the processor places an address, composed of multiple bits of binary information, on the address bus. Simultaneously, the processor also sends a memory read signal through its control bus. When these signals reach the memory device, the data are sent to the processor automat: by the memory device. Schematically, this transaction is depicted in Figure, which is known as timing diagram. Observe from Figure that data must be valid (stable) when the memory read signal goes from low to high. Address signals, generated by the processor, are stable at this stage to ensure a valid data transact

One question may arise here that how, out of so many devices interfaced with address, data and control bus, the correct device would pay attention to the processor's demand and that the other

devices would remain silent? The answer is, every device has a chip select input (generally, designated as CE or CS) and if this input 1s not activated, the device does not react with the system bus communications, Using a part of the address lines and a suitable decoder (we have studied this in Chapter 3), the processor activates only one device during any communication and that solves the problem. This technique is known as address decoding and device selection. A processor is assisted with a memory decoder and an VO decoder to target the correct device, which is of current interest.

INSTRUCTION DECODE

After receiving the instruction code byte within itself, the processor becomes busy in understanding it (what to do?). This part is known as instruction decode, carried out within the processor itself. After the completion of instruction decoding, the processor knows whether to fetch operands from external memory or to increment a register by one or to store a register content in external memory location.

This instruction decoding may be implemented through hardware. Instruction decoding may also be implemented through software, known as micro-programming. This demands a miniature processor within the processor itself, completely devoted for instruction decoding and its execution.
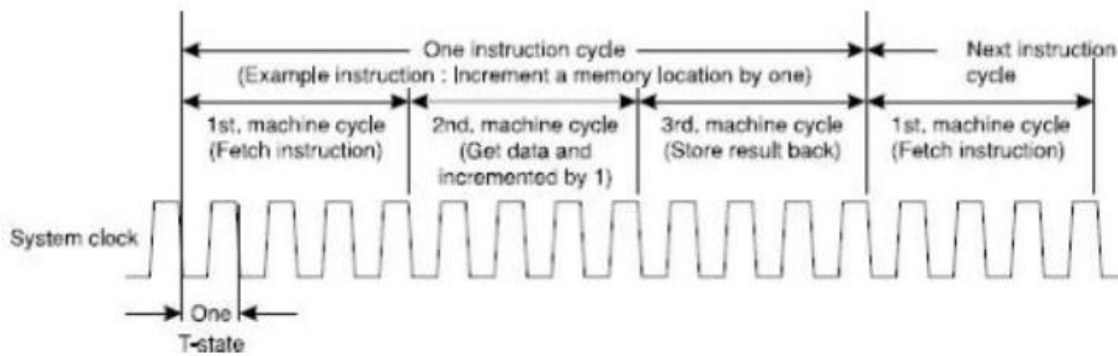
INSTRUCTION EXECUTE

This is the last and final phase of an instruction's execution. Depending upon the instruction, one or Several operations are implemented by the processor. Once this part is complete, the processor looks forward for the next instruction fetch-decode-execute, and the process continues.

1.6 MACHINE CYCLE AND T-STATES

An instruction cycle has one or more machine cycles and every machine cycle is composed of several T-States. These points need some elaboration. A machine cycle is the step or time-slice during which 1-byte (or one word) of data are transacted between the processor and some external device. Generally, this external device is the memory device. However, in exceptional cases it might be an I/O device also. To transact 1-byte of information,
information, one machine cycle must be executed by the processor. In Figure 1.15 , we have illustrated such a machine cycle. Note, that instead of reading, it might be a writing operation also. Each machine cycle is composed of several T-states. One complete oscillation of the processor clock is designated as one T-state. Depending upon the processor, the number of T-states necessary to complete one machine cycle must be known. For example, Intel 8085 processor needs four to six T-states to complete one machine cycle. The correlation of T-states, machine cycle and instruction cycle is shown in Figure 1.16 . Figure

Fig 1.16 *Example of instruction cycle, machine cycle and T-state correlation*

For the sake of example, execution of an instruction increment a memory location by one is illustrated through Figure 1.16. It is assumed that it is a 1-byte instruction, which is fetched by the first machine cycle. As the data, to be incremented by one, are available in external memory location, the next machine cycle reads this operand from memory

(brings the data byte within the processor). The data are then incremented by one by the processor and are stored back in the same memory location in the third machine cycle. Two questions may arise after this explanation of Figure 1.16, as follows

When the instruction was decoded?
When the data were incremented by one?

To answer the first question, it must be pointed out that the instruction must be decoded before the beginning of the second machine cycle, as the processor must know by that time what to do. As a rule, instruction decoding is carried out immediately after receiving the instruction byte within the processor. In other words, for every case, instruction decoding is done at the end of the first machine cycle. Does it not demand any extra time? Well it depends. If it is a hardware-based decoding, then it does not need any extra time. However, in case of micro-programming, it would consume one or two extra T-states. For example, Intel 8085 spends four T-states for fetching the first instruction byte during its first machine cycle, while for subsequent machine cycles it spends only three T-states. As a matter of fact, in its first machine cycle first three T-states are sufficient for fetching the first byte of instruction. Next T-state of the first machine cycle is devoted for instruction decoding. As the answer of the second question, we can say that the data would be incremented either at the end of second machine cycle or at the beginning beginning of the third machine cycle, depending upon the processor. Here also, the adopted technique plays an important role.

1.7 TIMINGS, CONTROL AND RESPONSE

Through the above discussions, it must be clear to the reader that timing and control play very important roles in smooth and efficient functioning of any processor. To further explain this concept, we may take up the example of interrupt. Although we shall have a detailed discussions on interrupt, it may be introduced here as an external asynchronous signal, which forces the processor to carry out something special for it by branching to a pre-defined address and, thus, executing a special program segment, known as interrupt service routine (ISR). As this is an asynchronous signal, it may be activated at any time during the execution of any instruction by the processor. However, the processor cannot leave an instruction's execution half-way to start doing something else for the sake of such an interrupting signal. To solve this problem, processors reserve a particular time-slot for checking the existence of any interrupt input signal during the execution of each and every instruction. For example, Intel 8085 processor had reserved the penultimate T-state of the last machine cycle of any instruction for this interrupt signal checking. If it is present, then the next instruction would not be executed immediately and the processor would start executing from the interrupt's ISR. However, the modified flowchart of the instruction cycle is presented in Figure 1.17, where the previously explained portion is shaded.
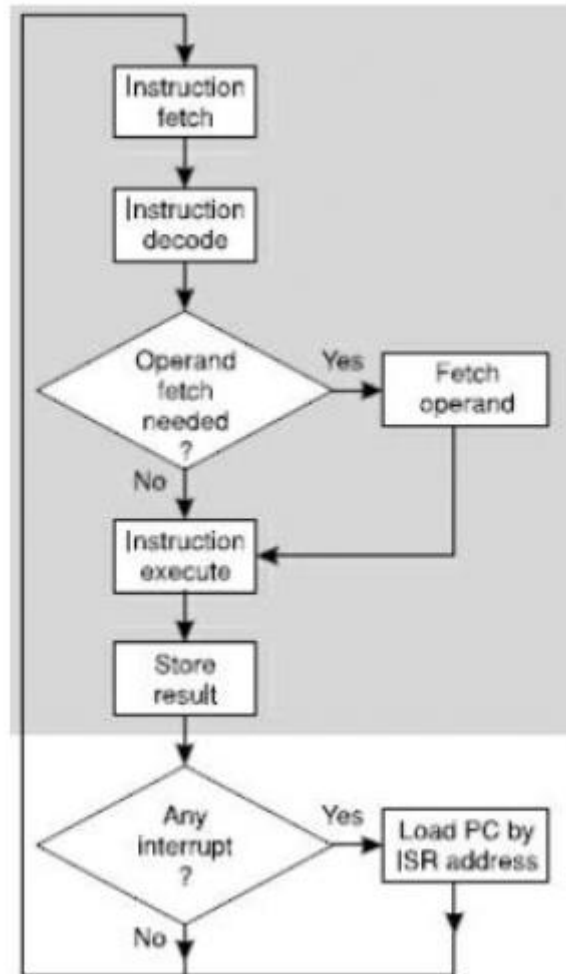
Fig 1.17: Modified Flow chart for simplified instruction cycle

1.8 REGISTER SET

To perform internal operations, all processors offer some internal registers, which can store temporary information or some operands. Similar to read/write memory, these registers are nothing but a combination of several flip-flops. Most of these registers are user (programmer) accessible and a few are not. The number of user accessible registers varies from processor to processor. Those processors that are memory oriented (e.g., Motorola 6800) offers lesser number of internal registers as it expects the data or operands would mainly be stored and manipulated within the read/write memory (RAM) of the system. On the other hand, some processors are register oriented (e.g., Zilog Z80), which offers a larger number of internal registers for the user. It may be noted that the program execution time for a processor would be less if the data are available within itself rather than looking outside for them. However, more internal registers means more complexity in instruction decoding as each register would demand a separate instruction to be provided by the instruction set of the processor. So far, we have been discussing about the general purpose registers. However, other types of registers are also available

within the processors. They are accumulator or result register, status register, stack pointer, program

counter, interrupt register and so on. Most of these registers, in most processors, would be user accessible. Apart from these, there are some registers that are purely for processor's own use, e.g., temporary registers. We shall now have a brief discussion about some of these special purpose registers.

STATUS REGISTER

Every processor performs some arithmetic or logical operations generating some results. Depending upon whether the result is zero or negative or produced a carry or odd/even parity, some additional actions might have to be taken by the programmer. Status register solves this problem by offering the result status of the last performed arithmetic or logical operation through its pre-assigned bits. Generally, each bit of this status register is assigned for one particular indication, e.g., carry, parity, zero, overflow and so on. These bits act as flags and their conditions (true or false) help the program to decide further course of actions and dictate the conditional program branching.

ACCUMULATOR

In earlier processors, result of all arithmetic or logical operations were made available only in the accumulator. In more recent register-to-register architecture, all relevant registers available within the processor may contain the result of similar operations.

PROGRAM COUNTER

This is one of the most important registers within any processor as it is responsible for holding the address of the memory location for next instruction byte/word to be fetched by the processor. After fetching every instruction byte, this is automatically incremented by one to point to the next byte. The only exception for this auto-increment of the program counter is in the case of program branching, when it is reloaded by a new value. This counter is always initialized during system reset so that the first executable instruction byte is fetched from a pre-defined location of the memory.

STACK POINTER

System stack is a RAM area, which is earmarked by the programmer to accommodate important information, e.g., return address or register values, in last-in-first-out (LIFO) sequence. Stack pointer always points to the top of the stack area.

GENERAL PURPOSE REGISTERS

These registers are available within the processor for temporary data storage and manipulation. For arithmetic or logical operations, one of the two operands must be within these registers (the other one should be in the accumulator). As already indicated, number of these registers vary, depending upon the processor.

STACK ORGANIZATION

Stack is an area within the system RAM earmarked for some special storage by the program or programmer. In other words, the stack consists of several bytes of read–write memory where some special data may be stored in and restored from, as per the program's requirements. Why this cannot be accomplished by using the available registers within the processor? This is because, the number of registers is very limited and they have their other specific purpose rather than storing return addresses. Stack is, generally, used to store some important address and data sets. The particular location within the stack, where the next such information to be stored, is known as the stack-top. Generally, the address of this stack-top is available in the register designated for this specific purpose and known as Stack Pointer. Figure 1.18(a) illustrates a sample stack area within the address space between FFF0H and FFFFH (16-bytes) and the stack pointer. It is assumed that some stack locations are already occupied (used for storage) and the stack pointer has the address of the next free location of stack, i.e., FFF9H.
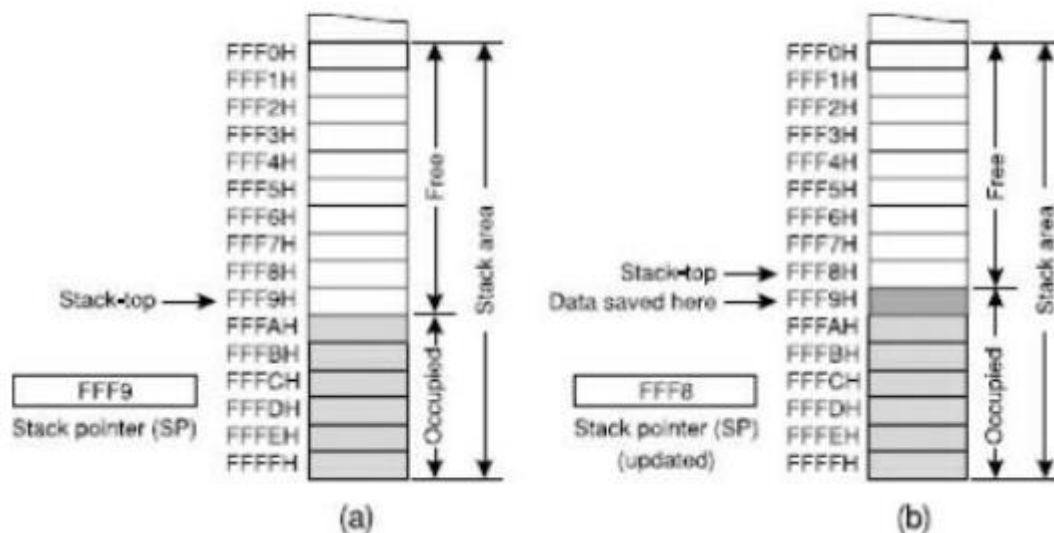
Fig 1.18 Stack and its operation

If any new data to be stored within the stack are included, it must be stored in the address pointed by the stack pointer, i.e., FFF9H, and in that case the stack pointer would show the next available free location for storage, i.e., FFF8H, as shown in Figure 1.18(b). Stack follows the last-in-first-out (LIFO) data movement technique. In other words, the data that is placed last on the stack-top must be retrieved first.

STACK AS STORAGE AREA

In general, every processor offers two instructions to handle the stack directly. These two instructions are PUSH and POP. PUSH instruction places the data on the stack-top and POP instruction takes it out from the stack-top. These data might be originally available within a general purpose register or some other data. It is already mentioned that the register within the processor, which holds the current stack-top address, is designated as

stack pointer (SP). Whenever any data are placed on the stack-top or taken out from it, SP is also automatically changed by the processor itself. Here, we use the term 'changed' as, for some processors, a PUSH operation increments the SP while for other processors the SP is decremented for a PUSH instruction.

SUBROUTINES AND STACK

Stacks are widely used for subroutine calls. In these cases, the return address from the subroutine is placed on the stack-top before branching to the subroutine. As subroutines are always terminated by a RETURN instruction, once the execution of
the subroutine is complete, this RETURN instruction forces the processor to load the program counter from the stack-top, producing an effect of returning to the original part of the program that was left to branch to the subroutine. As an example case of flow of program control during execution of subroutine call and return instructions, a portion of a program is shown in Figure 1.19, which includes a call to one of its subroutines (Get Average). When this Call Get Average instruction is executed, the processor stores the address of the next instruction (18F3H) on stack-top and loads the program counter by the address of the subroutine 224BH. The reader may ask how does the processor know about the address of the subroutine? Well, the address of the subroutine is included within the call instruction itself. The control is then transferred automatically to the subroutine, which terminates with a Return instruction. During the execution of this Return instruction, the processor always reloads the program counter from stack-top and, in this case, it loads the program counter by the value 18F3H, which it had saved over the stack. Therefore, the control is now automatically transferred to the next instruction after the call instruction, and the execution proceeds sequentially thereafter. Stack is also essential for service interrupts, which we are about to discuss now.
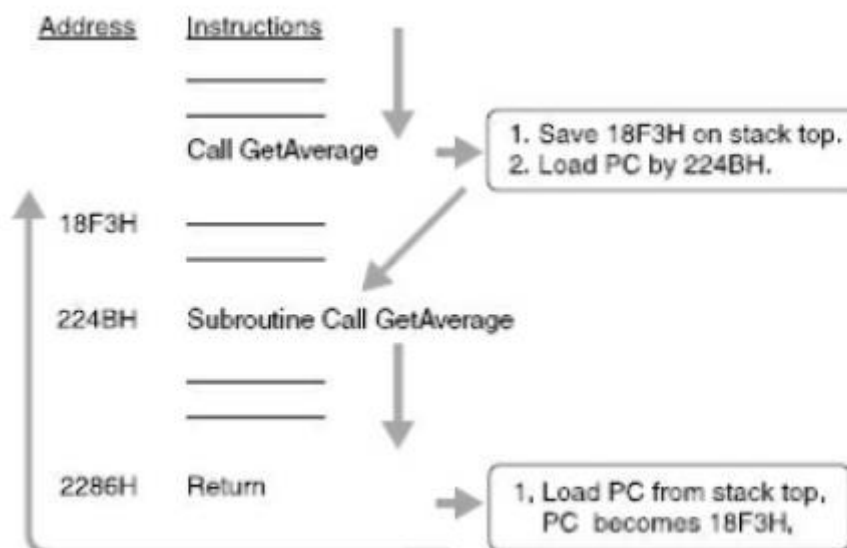

Fig 1.19: Functioning of stack during subroutine call and return

1.20 ALGORITHMS FOR BINARY MULTIPLICATION AND DIVISION

1.21 MULTIPLICATION ALGORITHMS

Apart from addition and subtraction, multiplication is another frequently used arithmetic operation. Several algorithms are available to implement it with binary numbers, both unsigned as well as signed. We shall show only a few of those in this section.

Paper and Pencil Method

The paper and pencil method that we adopt to perform multiplication of decimal numbers is also applicable for binary numbers, as illustrated in Figure 1.20 . Note its similarity with ANDing operation of Boolean algebra. To illustrate binary multiplication, we have selected two integers, 2 and 3 and shown their multiplication details by interchanging the multiplier and multiplicand to confirm that the order does not affect the result.

Fig 1.20　Example of multiplication method with binary numbers

In the first case 3 is multiplied by 2. As 3 in decimal is represented by 0011 and 2 by 0010 in binary, these binary values are written one below the other. Following the basic rule of multiplication as shown in Figure (c), four partial products are obtained. Note the way each partial product is placed in offset with the previous partial product, which we are familiar in our decimal multiplication. These partial products are finally added together to generate the result. The most important point may be noted here that the product of two 4-bit numbers may be as long as 8-bit. The same is applicable for 8-bit or 16-bit numbers, which may generate their products as 16-bit or 32-bit respectively.

Method of Repeated Additions

In computers, multiplication may be implemented in various methods. Another method is to perform repeated additions. With our example numbers, we may develop a program to add 2 three times to get the result 6. However, this method is time consuming in comparison to another method known as Booth's algorithm, which we are about to discuss now.
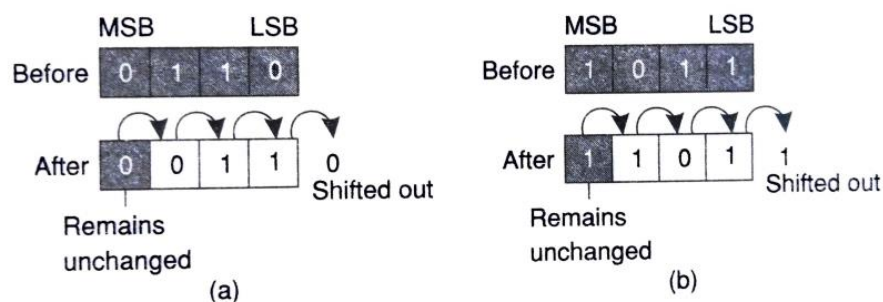
BOOTH'S ALGORITHM

Booth's algorithm (by Andrew D. Booth) for multiplication uses two's complement representation of binary numbers and is applicable for both positive and negative integers. Before discussing this algorithm, we should be familiar with one technique used as an important part in this algorithm, which is known as arithmetic right-shift.

Arithmetic right-shift

In normal right-shift operations of any microprocessor, all bits of the container, i.e., a register, are shifted uniformly one-bit towards right, making the present value of bit $(n-1)$, = the old value of bit, In this process, the least significant bit is thrown out of the register, generally pushed into the carry flag, and a new bit, the content of the carry flag is inserted in the place of most significant bit. This procedure is usually designated as rotate-right-through-carry. In another variation of this right-shift, the carry flag does not come into the picture and the least significant bit is shifted into the most significant bit. This technique is designated as rotate-right-circular.

In case of arithmetic right-shift, all bits are shifted one bit right and the least significant bit is thrown out, identical as what happens in the case of a normal right-shift operation. The difference is in the condition of the most significant bit, which remains unchanged even though it is shifted to its right, after an arithmetic right-shift. It looks like as if after a normal right-shift, the old content of the most significant bit is copied back to its original place, keeping it unchanged. Therefore, in arithmetic right-shift, the most significant bit or the original sign-bit of the number remains unchanged. Two examples of arithmetic right-shift, one with a positive and another with a negative number are illustrated in Figure 1.21 using 4-bit format. Note that the same principle is applicable for 8-bit, 16-bit and all other bit formats in the same manner.

Fig 1.21: Example of 4-bit arithmetic right-shift

Locations and Counters

Booth's algorithm performs this arithmetic right-shift as many times as the number of bits involved. In other words, for a 4-bit representation of data, four such arithmetic right-shifts are performed. For 8-bit data sets, eight right-shifts are necessary and for 16-bit data, 16 right-shift operations have to be implemented.

Generally, a location (register) is used as counter for this purpose, which is initially loaded with the number of bits represented (a known and constant value for a computer system) and is decremented by one after every shift. When this counter becomes zero, the multiplication operation is considered to be completed as per Booth's algorithm. The register usage for Booth's algorithm is presented in Figure 1.22, which would be referred for explanation purpose. Instead of registers, we shall designate these as locations, which is a general designation, and this would be more appropriate in the present context.
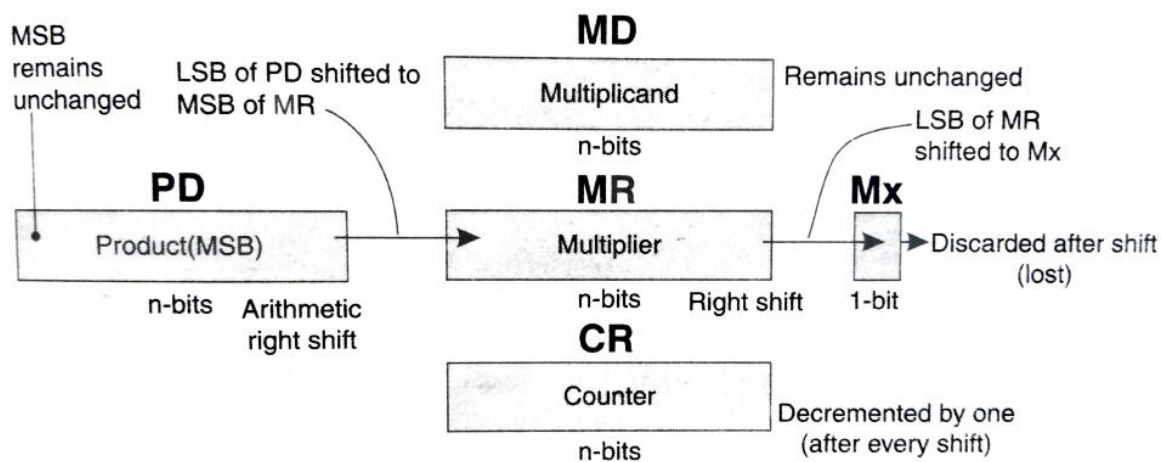

Fig 1.22 Locations involved for Booth's algorithm

- ❖ **MD** (n-bit) is for n-bit multiplicand
- ❖ **MR** (n-bit) is for n-bit multiplier (initially) and LS n-bit of product (finally)
- ❖ **CR** (n-bit) is for counter (from n to 0)
- ❖ **PD** (n-bit) is for MS n-bit of product (finally)
- ❖ **Mx** (1-bit) is for shift out from MR.

Details of Shift Operation

Now let us consider the shift operation which must be performed n times (till CR = 0). For the location PD ,it would be arithmetic right-shift. The LS bit of PD, coming out by this process, would be inserted within MS bit of MR and all original (old) bits MR are to be shifted one-bit right. The LS bit of MR, which would be coming out by this process, would be accommodated within Mx and the old (previous) content of Mx would be lost (discarded). Through Figure 1.23, all these operations are explained and may be correlated by the readers.
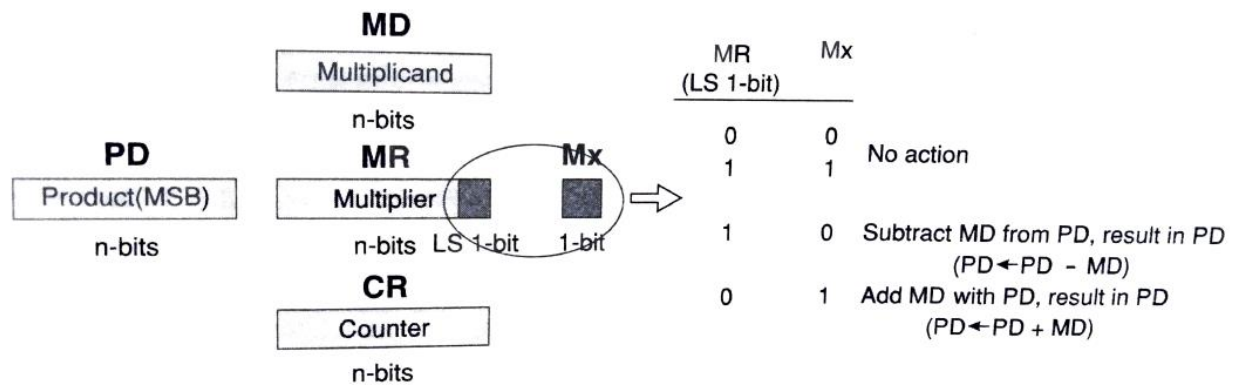
Fig 1.23 : Condition check and actions to be taken before every shift.

Finally, before implementing every shift operation, we are to ensure certain conditions and perform addition, subtraction or no such operation accordingly. The condition we are to check is the pattern generated by the two bits, the LS bit of location MR and 1-bit location Mx (both encircled in Figure 4.8). If these two bits are either 00 or 11, then we do not implement any addition or subtraction and directly proceed to the shift-right operation. However, if the pattern is 10 (LS bit of MR is 1 and Mx containing 0) then we are to subtract content of MD from the present content of PD and the result would be placed in PD. Note that the original content of PD would be lost. Any eventual borrowing during this subtraction would be neglected. It is needless to indicate that two's complement addition may be performed in place of subtraction. On the other hand, if the pattern is 01 (LS bit of MR being 0 and Mx having 1), then we are to add MD with the current content of PD and the result would be stored in PD overwriting PD's old content. Any carry generated by this subtraction or addition would be neglected. All four conditions and actions to be taken against each are shown at right side of Figure 1.23.

Algorithm and Flowchart

Having discussed all basic techniques related to Booth's algorithm, we are now in a position to discuss the steps involving it. The algorithm is presented through the following steps and also presented as flowchart through Figure 1.24.

- **Step I:** Load multiplicand in MD, multiplier in MR. For negative numbers, two's complement format to be used.
- **Step 2:** Initialize the down counter CR by the number of bits involved.
- **Step 3:** Clear locations PD (n-bits) and Mx (1-bit).
- **Step 4:** Check LS bit of MR and Mx jointly. If the pattern is 00 or 11 then go to Step 5. If 10, then PD = PD — MD. If 01, then PD = PD + MD.
- **Step 5:** Perform arithmetic right-shift with PD, MR and Mx. LS of PD goes to MS of MR and LS of MR goes to Mx. Old content of Mx is discarded.
- **Step 6:** Decrement CR by one. If CR is not zero then go to Step 4.
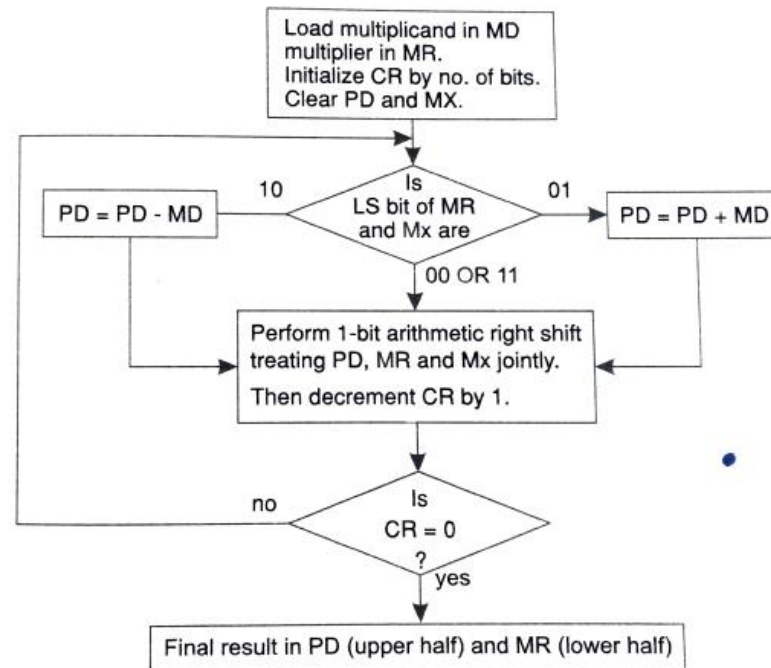- **Step 7**: Final result (or the product) is available in PD (higher part) and MR (lower part).

Fig 1.24 : Flow chart for Boot's Algorithm

Example 1 :   2 X 3

All locations are used except Mx, would be 4-bit locations as the maximum range of numbers covered is less than 7, in our case. We initialize MD by loading 2 (0010) and MR by 3 (0011). PD and Mx are cleared to 0000 and 0, respectively, and CR 1s loaded by 4, to count the number of cycles of iteration.

To start the first cycle, we check the pattern formed by LS bit of MR and Mx (indicated by an underline in the sketch). As they are 10, we subtract MD from PD (conditions indicated at right side of Figure 1.23), For subtraction, we calculate two's complement of 0010 in MD, which is 1110. This is added with PD (presently 0000) and the result 1110, is placed in PD. The next step is to perform one-bit arithmetic right-shift, with PD, MR and Mx together. After the shift, we get PD as 1111, MR as 0001 and Mx as 1. Note that LS bit from PD is shifted to MS bit of MR and LS bit of MR goes to Mx, as we have discussed before. The next step is to decrement the counter CR by 1, which becomes 3. This completes the first cycle of iteration and three more cycles remain. The reader may note that in Figure 1.25, those locations which do not contribute in an operational phase are lightly shaded.

As the counter CR is not 0, we start the second cycle by checking two bits, LS of MR and Mx. This is because, they appear as 11, there is no need for any addition or subtraction and we may perform only the arithmetic right-shift operation using PD, MR and Mx. After

shifting right, PD becomes 1111, MR . becomes 1000 and Mx becomes 1. By decrementing CR by one, we complete the second cycle, and then two more are remaining.

We then check two bits of MR and Mx (underlined) at the starting of the third cycle and as they are 01, we perform an addition of PD (presently 1111) with MD (having 0010). We place the result of this addition, 0001, in PD and then perform one-bit arithmetic nght-shift as before. This shifting makes PD as 0000, MR as 1100 and Mx as 0. Counting down the location CR, we get 1 and now we may start our last cycle.

As the two bits to be checked (LS of MR and Mx) are presently 00, we skip any addition or subtraction and simply perform the arithmetic right-shift of all the three locations (PD, MR and Mx) by one bit. This would leave 0000 in PD, 0110 in MR and 1 in Mx. Lastly, we decrement CR by 1 and as it is 0, we terminate the whole process. Note that the final product is now available as an 8-bit value within locations PD and MR (thick underlined), placed side by side. Location PD is to contain the most significant half and MR accommodates the least significant half of the product. In our example case, it contains 0000 0110, which is the correct answer (6 1n decimal).
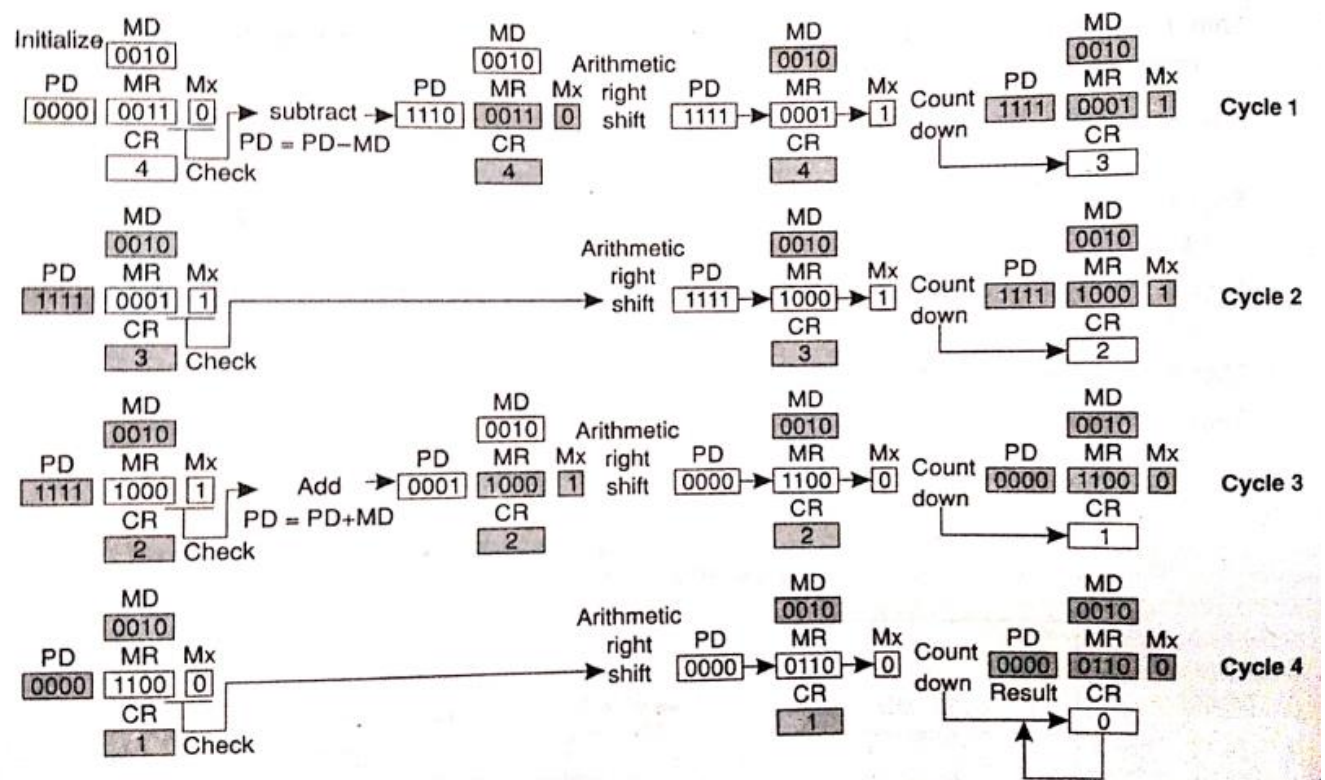

Fig 1.25: Illustration of Booth's algorithm for 2X3

Example 2 :   2 x (-3)
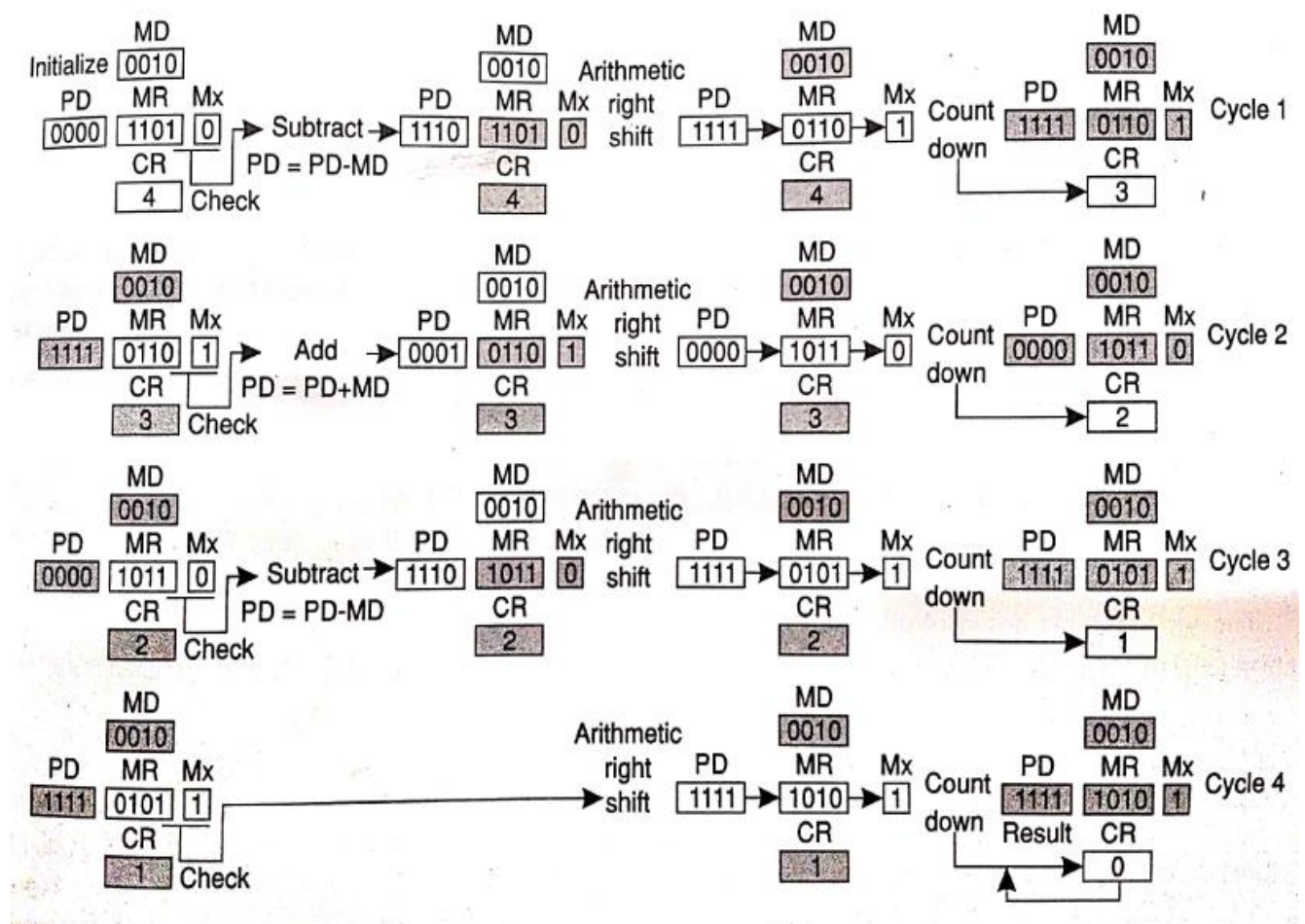

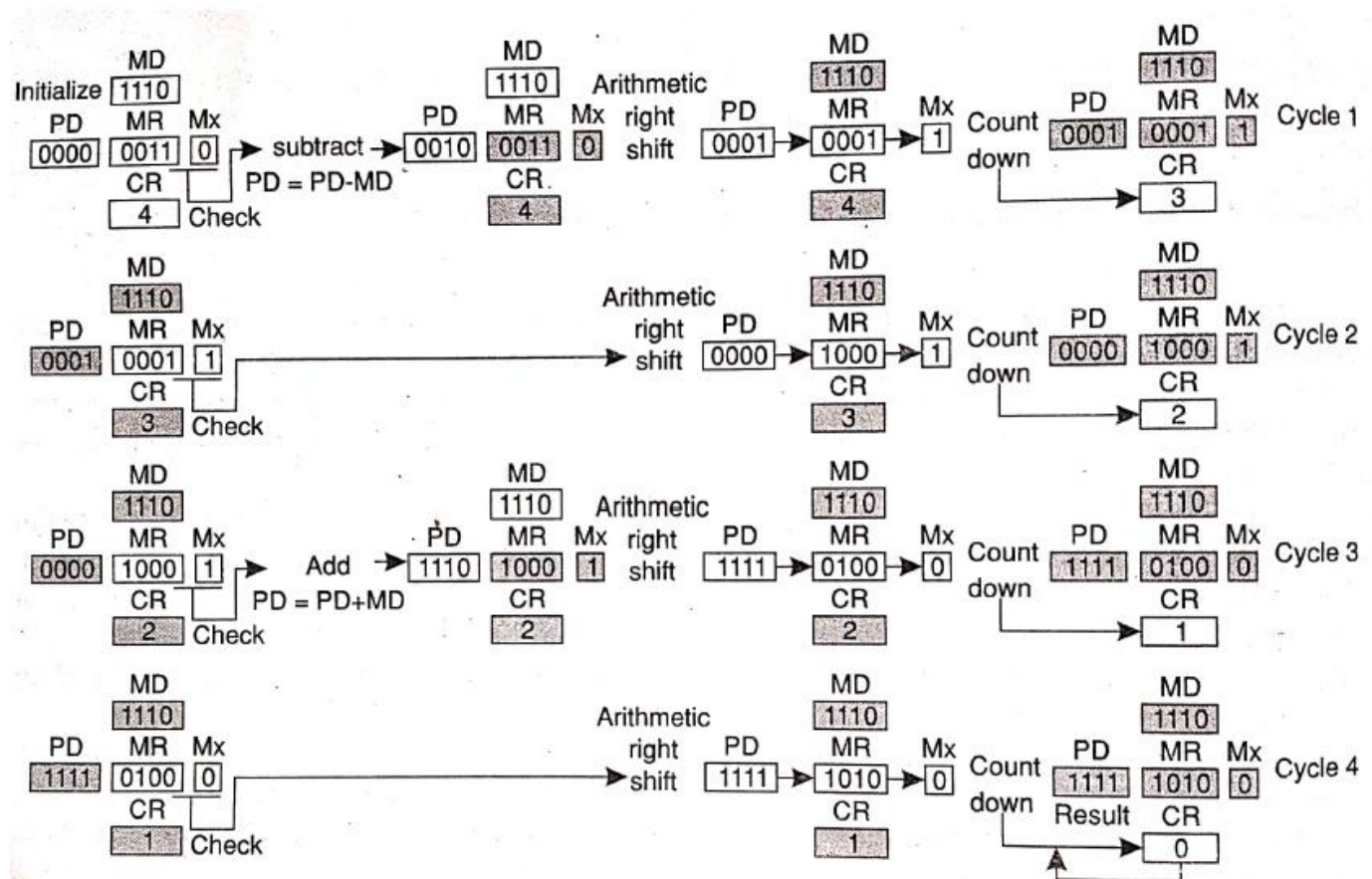*Fig 1.26: Illustration of Booth's algorithm for 2X-3*

Example 2 :   (- 2) x 3

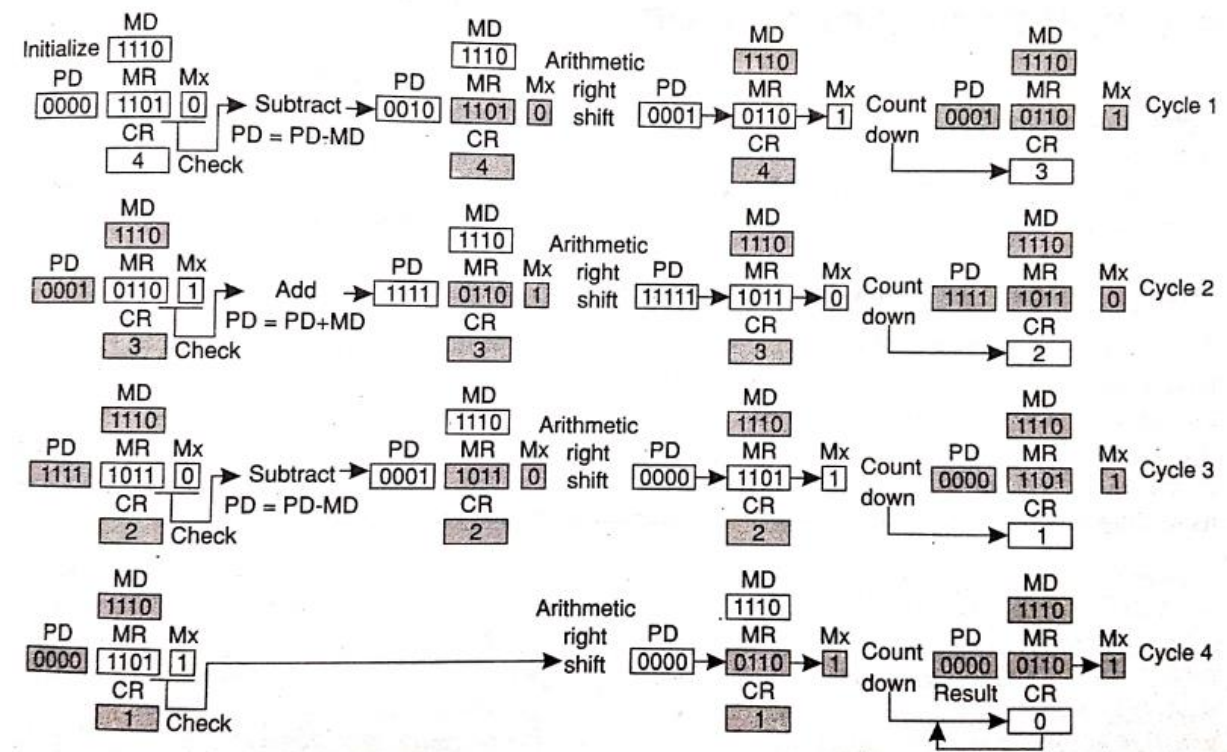*Fig 1.27: Illustration of Booth's algorithm for- 2X3*

Example 2 :   (- 2) x (- 3)

Fig 1.28: Illustration of Booth's algorithm for -2X-3

1.22 DIVISION ALGORITHMS

Just like multiplication may be carried out by repeated additions, division also may be completed by repeated subtraction till a negative reminder is encountered. However, there are other algorithms also for performing divisions with integers. We may also adopt the method used by us to perform division in longhand method (paper and pencil method). One such sample calculation is shown in Figure 1.29 . We spend some time on this as that would give us some insight related to the methods of unsigned binary division.

Paper and Pencil Method

To illustrate paper and pencil method of division, we take 5 as dividend and 2 as divisor. In 4-bit format, these numbers may be written as 0101 and 0010, respectively. This is shown in Figure 1.29(a). To initiate the process of division, the dividend has to be scanned from left to right, one digit at a time, and if not divisible, a zero to be placed in the place of quotient. Therefore, we start our scanning and first encounter with a zero [underlined in Figure 1.29 (b)]. As it is less than the divisor (10, we may neglect its leading zeros in this case), the first zero is inserted at the quotient [Figure 1.29 (b)].
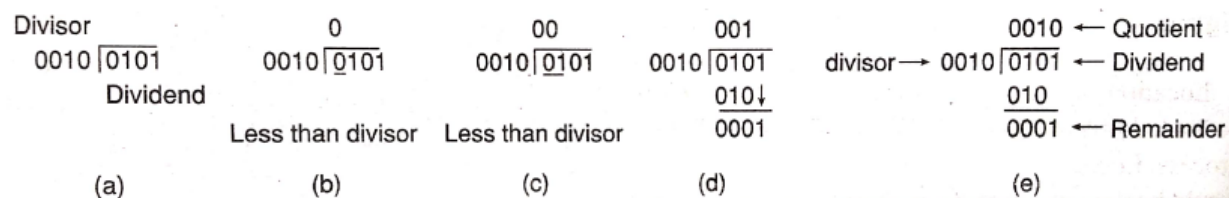
Fig 1.29 : Example of division by paper & pencil Method

We continue our scanning from left to right and next encounter with the left most two digits of the dividend, i.e., 01 (underlined). Again it is found to be less than the divisor, which forces us to add another zero in the quotient [Figure 1.29(c)]. As we continue our scanning, we next meet with 010 part of the dividend (its three leading digits) and find that it is equal (even greater would do) to the divisor. At this point, we may add a | in the place of quotient, making it 001, write the divisor below the dividend and perform a subtraction. The remainder in this case is 000. We then write the next considerable digit of the dividend, i.e., 1 at the right side of the remainder [shown by a vertical downward arrow in Figure 1.29(d)].

As the last step, we find that the present remainder 0001 is less than divisor. Therefore, we add another zero with the quotient making it 0010 and complete our process. Thus, finally, we get a quotient of 0010 (2 in decimal) and a remainder 0001 (1 in decimal) by our process, matching with our expectations.

An algorithm may be developed for the above method to divide unsigned integers. However, this method would not be applicable for signed integers, expressed in two's complement form. For that purpose, we have to adopt another algorithm (William Stallings, 2009), as described below.

Locations and Counters

To implement this algorithm for division of signed integers, we would need four locations; all should be n-bit wide, where 'n' is the number of bits being considered for divisor and dividend. In other words, for 4-bit numbers, we shall need 4-bit wide locations, for 8-bit numbers, we shall need 8-bit wide locations and so on. We designate these four locations as V, R, D and C, as indicated in Figure 1.30.
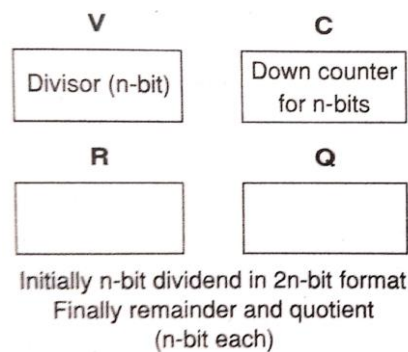

Fig 1.30: Locations involved for division algorithm.

Location V, is to contain n-bit divisor. If the divisor is negative, its two's complement form should be loaded in V. We shall only use its content, which would remain unchanged till the completion of the process. Location C is the n-bit down counter and initially to be loaded by n. At the end of every cycle, C would be decremented by 1. The operation of division would be complete when C becomes 0. The n-bit dividend must be expanded to 2n-bit form and be loaded in locations R and Q, where R would contain the most significant part. If negative, then the dividend should be changed to its two's complement form and expanded from n-bit to 2-bit format. In other words, if 2 is 0100, then it should be expanded as 00000100 and if 7 is 1101, then it should be expanded to 11111101. However, at the end of operation, R would contain n-bit remainder and Q would contain n-bit quotient
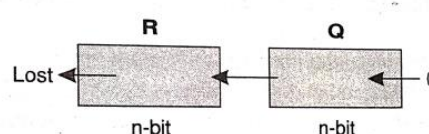
One bit Left-Shift

Fig 1.31: Detail of one-bit left shift operation for division algorithm

Let us consider the shifting technique to be implemented at the beginning of each iteration. In this case it would be a one bit left-shift considering R and Q, simultaneously. A zero has to be inserted at the LS bit of Q, and the MS bit of Q should be shifted to LS bit of R. The MS bit of R would be moved out and discarded.

Algorithm for Division of Signed Integers

The algorithm for the division of signed integers is explained through the following steps.

- **Step I:** Load V by n-bit divisor using two's complement form for negative numbers.
- **Step 2:** Load n-bit dividend within R and Q after expanding it to 27-bit format maintaining its sign as it is.
- **Step 3:** Load C by the number of bits being considered, i.e., n.
- **Step 4:** Perform 1-bit left-shift with R-Q, inserting a 0 at the LS bit of Q.
- **Step 5:** If the divisor (in V) and R have same sign, then replace R by R — V, otherwise replace R by R+V.
- **Step 6:** If the sign of R remains unchanged after Step 5, or R becomes 0, then set LS bit of Q as 1. Otherwise, if the sign of R changes after Step 5 and R becomes non-zero, then restore the value of R as it was after Step 4.
- **Step 7:** Decrement C by 1 and if it is not 0, then go to Step 4.
- **Step 8:** Find the remainder in R. If the divisor and dividend have same signs, then Q indicates the quotient. Otherwise, its two's complement would be the correct quotient.

Note that steps 4 to 7 are involved in the iteration process. Steps 1 to 3 are for initialization and the Step 8 is for getting the correct result. We next discuss two example cases for implementing this division algorithm.
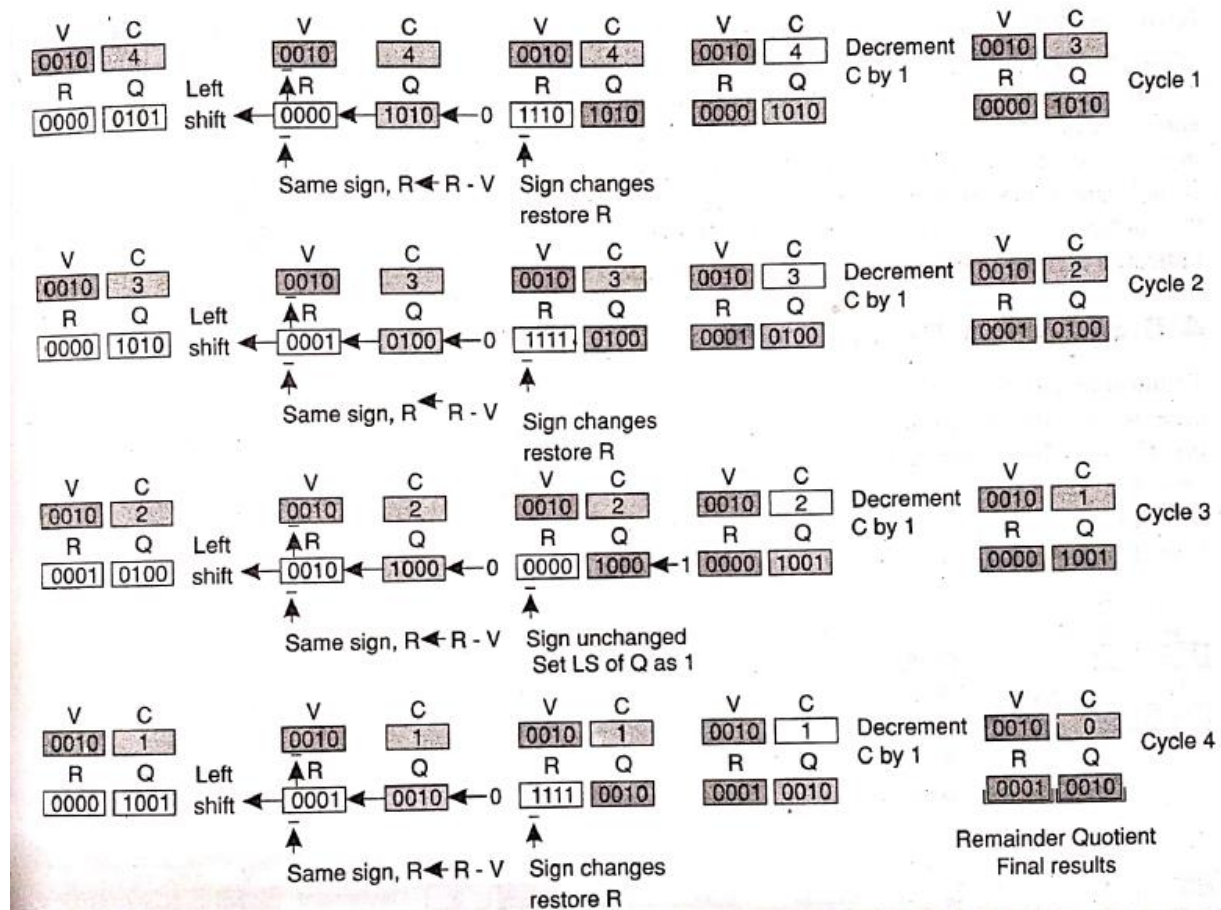
Example :   **5÷2**

Fig 1.32 : Illustration of division algorithm for 5 divided by 2

We start with initialization by loading the divisor (2 in decimal) in location V in its binary form of 4-bit, i.e., 0010. The counter C is initialized as 4 to represent 4-bit operation. The dividend (5 in decimal) is converted to its 8-bit representation, i.e., 00000101 and its lower half 0101 is loaded within location Q and the upper half 0000 1s loaded in location R.

The first cycle begins with a one-bit left-shift of R and Q and a 0 is inserted at the least significant position of Q. This changes Q to 1010 and R remains as 0000 after left-shift. As the MS bit of R and MS bit of V are both 0, a subtraction operation is to be performed to replace R by R — V. So, two's complement form of V, 1.e., 1110 is added with R, i.e., 0000 to get 1110, which becomes the value in R at this point. As there is a sign change between the present content of R (1110) and its previous content (0000), the previous value of R is restored and R becomes 0000 again. By decrementing C from 4 to 3, we complete the first cycle and three more cycles are pending.

The second cycle begins with a left-shift of R and Q together changing R to 0001 and Q to 0100. As the present signs of V and R are identical (both are 0), V is subtracted from R

and the result (1111) is placed in R. This is because of the sign change of R in this process, R is restored to its original value of 0001. The counter C is decremented by | to make it 2.

In the third cycle, locations R and Q are jointly shifted one-bit left, making R as 0010 and Q as 1000. As V and R are having same sign at this point, therefore, R is replaced by the result of R ~ V, which is 0000.Note that the sign of R remains unchanged forcing us to set the least significant bit of Q as 1.This changes Q to 1001 and R remains as 0000. Finally, C is decremented to 1, completing the third cycle.

The fourth cycle starts with the left-shift of R as well as Q, changing R to 0001 and Q to 0010. As R and V are of same sign, R is replaced by R — V, which is 1111. This change in sign of R with respect to is previous sign forces us to replace R by its old value of 0001. C is now decremented to 0 indicating the completion of the division operation. The result of the division is now available at Q and R. Q contains the quotient, i.e., 0010 or 2 in decimal and R contains the remainder 0001 or 1 in decimal. These results indicated that the operation is correctly performed.

Example : -5 ÷ -2

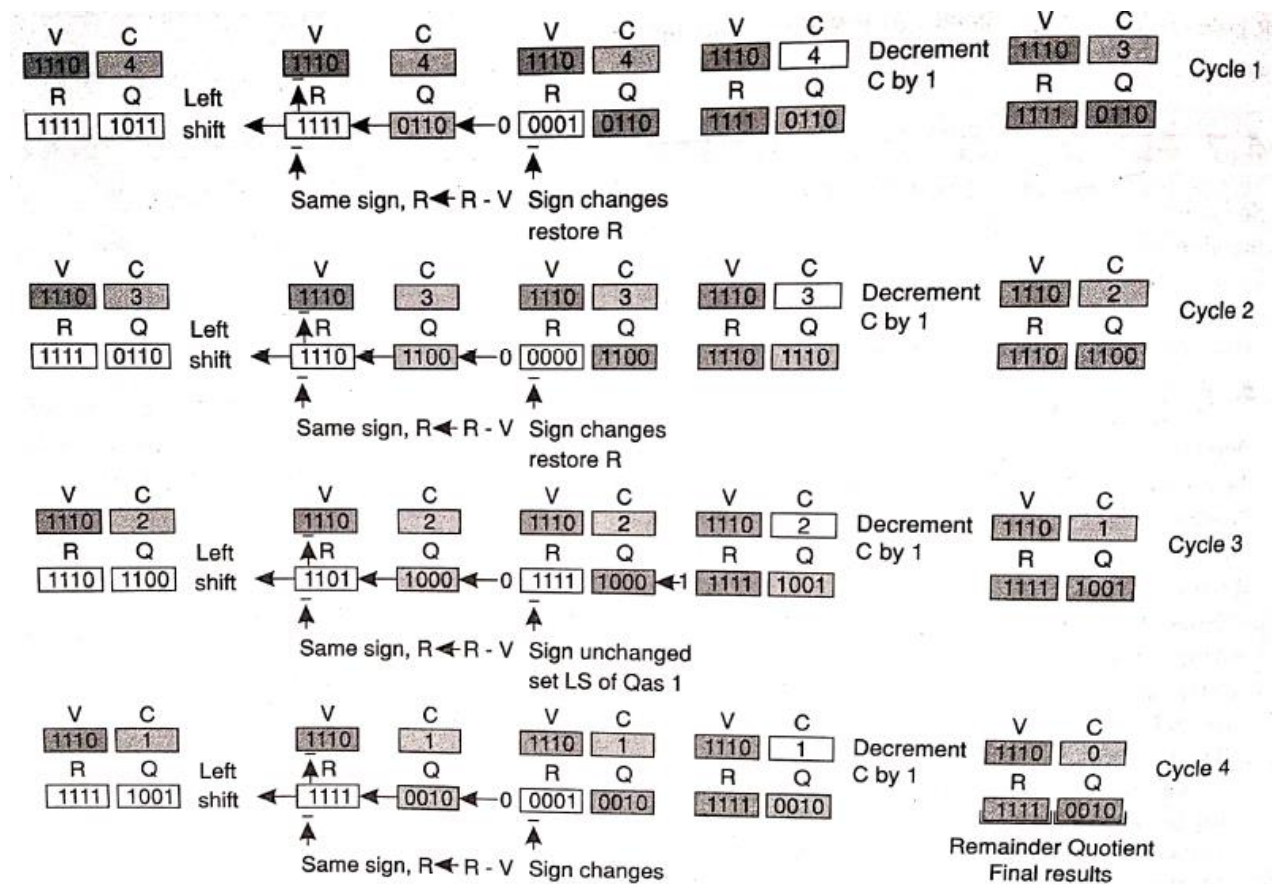Fig 1.33 : Illustration of division algorithm for -5 divided by -2

FIXED POINT AND FLOATING POINT NUMBER REPRESENTATIONS

Digital Computers use Binary number system to represent all types of information inside the computers. Alphanumeric characters are represented using binary bits (i.e., 0 and 1). Digital representations are easier to design, storage is easy, accuracy and precision are greater.

There are various types of number representation techniques for digital number representation, for example: Binary number system, octal number system, decimal number system, and hexadecimal number system etc. But Binary number system is most relevant and popular for representing numbers in digital computer system.

Storing Real Number

These are structures as following below –
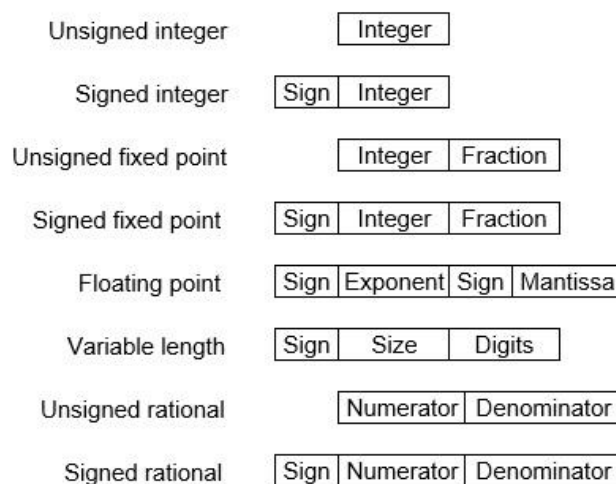
There are two major approaches to store real numbers (i.e., numbers with fractional component) in modern computing. These are (i) Fixed Point Notation and (ii) Floating Point Notation. In fixed point notation, there are a fixed number of digits after the decimal point, whereas floating point number allows for a varying number of digits after the decimal point.

 Fixed-Point Representation

This representation has fixed number of bits for integer part and for fractional part. For example, if given fixed-point representation is IIII.FFFF, then you can store minimum value is 0000.0001 and maximum value is 9999.9999. There are three parts of a fixed-point number representation: the sign field, integer field, and fractional field.

We can represent these numbers using:

 ❖ Signed representation: range from $-(2^{(k-1)}-1)$ to $(2^{(k-1)}-1)$, for k bits.

 ❖ 1's complement representation: range from $-(2^{(k-1)}-1)$ to $(2^{(k-1)}-1)$, for k bits.

 ❖ 2's complementation representation: range from $-(2^{(k-1)})$ to $(2^{(k-1)}-1)$, for k bits.

2's complementation representation is preferred in computer system because of unambiguous property and easier for arithmetic operations.

Example –Assume number is using 32-bit format which reserve 1 bit for the sign, 15 bits for the integer part and 16 bits for the fractional part.

Then, -43.625 is represented as following:

Where, 0 is used to represent + and 1 is used to represent −. 000000000101011 is 15 bit binary value for decimal 43 and 1010000000000000 is 16 bit binary value for fractional 0.625.

The advantage of using a fixed-point representation is performance and disadvantage is relatively limited range of values that they can represent. So, it is usually inadequate for numerical analysis as it does not allow enough numbers and accuracy. A number whose representation exceeds 32 bits would have to be stored inexactly.

These are above smallest positive number and largest positive number which can be store in 32-bit representation as given above format. Therefore, the smallest positive number is $2^{-16} \approx 0.000015$ approximate and the largest positive number is $(2^{15}-1)+(1-2^{-16})=2^{15}(1-2^{-16})$ =32768, and gap between these numbers is 2^{-16}.

We can move the radix point either left or right with the help of only integer field is 1.

Floating-Point Representation

This representation does not reserve a specific number of bits for the integer part or the fractional part. Instead it reserves a certain number of bits for the number (called the mantissa or significand) and a certain number of bits to say where within that number the decimal place sits (called the exponent).

The floating number representation of a number has two part: the first part represents a signed fixed point number called mantissa. The second part of designates the position of the decimal (or binary) point and is called the exponent. The fixed point mantissa may be fraction or an integer. Floating -point is always interpreted to represent a number in the following form: $M \times r^e$.

Only the mantissa m and the exponent e are physically represented in the register (including their sign). A floating-point binary number is represented in a similar manner

except that is uses base 2 for the exponent. A floating-point number is said to be normalized if the most significant digit of the mantissa is 1.
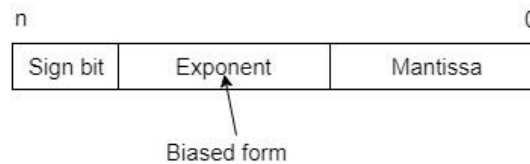
So, actual number is $(-1)^s(1+m) \times 2^{(e-Bias)}$, where *s* is the sign bit, *m* is the mantissa, *e* is the exponent value, and *Bias* is the bias number.

Note that signed integers and exponent are represented by either sign representation, or one's complement representation, or two's complement representation.

The floating point representation is more flexible. Any non-zero number can be represented in the normalized form of $\pm(1.b_1b_2b_3 \ldots)_2 \times 2^n$ This is normalized form of a number x.

Example –Suppose number is using 32-bit format: the 1 bit sign bit, 8 bits for signed exponent, and 23 bits for the fractional part. The leading bit 1 is not stored (as it is always 1 for a normalized number) and is referred to as a *"hidden bit"*.

Then −53.5 is normalized as $-53.5=(-110101.1)_2=(-1.101011) \times 2^5$, which is represented as following below,
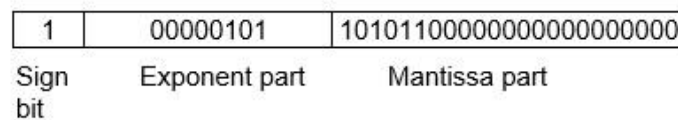
Where 00000101 is the 8-bit binary value of exponent value +5.

Note that 8-bit exponent field is used to store integer exponents $-126 \leq n \leq 127$.

The smallest normalized positive number that fits into 32 bits is $(1.00000000000000000000000)_2 \times 2^{-126}=2^{-126} \approx 1.18 \times 10^{-38}$, and largest normalized positive number that fits into 32 bits is $(1.11111111111111111111111)_2 \times 2^{127}=(2^{24}-1) \times 2^{104} \approx 3.40 \times 10^{38}$. These numbers are represented as following below,
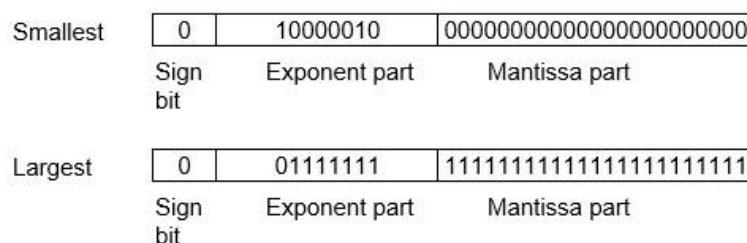
The precision of a floating-point format is the number of positions reserved for binary digits plus one (for the hidden bit). In the examples considered here the precision is 23+1=24.

The gap between 1 and the next normalized floating-point number is known as machine epsilon. the gap is $(1+2^{-23})-1=2^{-23}$for above example, but this is same as the smallest

positive floating-point number because of non-uniform spacing unlike in the fixed-point scenario.

Note that non-terminating binary numbers can be represented in floating point representation, e.g., $1/3 = (0.010101 \ldots)_2$ cannot be a floating-point number as its binary representation is non-terminating.

IEEE Floating point Number Representation –

IEEE (Institute of Electrical and Electronics Engineers) has standardized Floating-Point Representation as following diagram.

So, actual number is $(-1)^s(1+m)x2^{(e-Bias)}$, where *s* is the sign bit, *m* is the mantissa, *e* is the exponent value, and *Bias* is the bias number. The sign bit is 0 for positive number and 1 for negative number. Exponents are represented by or two's complement representation.

According to IEEE 754 standard, the floating-point number is represented in following ways:

❖ Half Precision (16 bit): 1 sign bit, 5 bit exponent, and 10 bit mantissa
❖ Single Precision (32 bit): 1 sign bit, 8 bit exponent, and 23 bit mantissa
❖ Double Precision (64 bit): 1 sign bit, 11 bit exponent, and 52 bit mantissa
❖ Quadruple Precision (128 bit): 1 sign bit, 15 bit exponent, and 112 bit mantissa

Special Value Representation –

There are some special values depended upon different values of the exponent and mantissa in the IEEE 754 standard.

❖ All the exponent bits 0 with all mantissa bits 0 represents 0. If sign bit is 0, then +0, else -0.
❖ All the exponent bits 1 with all mantissa bits 0 represents infinity. If sign bit is 0, then +∞, else -∞.
❖ All the exponent bits 0 and mantissa bits non-zero represents denormalized number.
❖ All the exponent bits 1 and mantissa bits non-zero represents error.

MODULE II
8051 ARCHITECTURE

2.1 INTRODUCTION

Salient features of 8051 microcontroller are given below.

- ❖ Eight bit CPU

- ❖ On chip clock oscillator

- ❖ 4Kbytes of internal program memory (code memory) *[ROM]*

- ❖ 128 bytes of internal data memory *[RAM]*

- ❖ 64 Kbytes of external program memory address space.

- ❖ 64 Kbytes of external data memory address space.

- ❖ 32 bi directional I/O lines (can be used as four 8 bit ports or 32 individually addressable I/Olines)

- ❖ Two 16 Bit Timer/Counter :T0, T1

- ❖ Full Duplex serial data receiver/transmitter

- ❖ Four Register banks with 8 registers in each bank.

- ❖ Sixteen bit Program counter (PC) and a data pointer (DPTR)

- ❖ 8 Bit Program Status Word (PSW)

- ❖ 8 Bit Stack Pointer

- ❖ Five vector interrupt structure (RESET not considered as an interrupt.)

- ❖ 8051 CPU consists of 8 bit ALU with associated registers like accumulator 'A' , B register, PSW, SP, 16 bit program counter, stack pointer.

- ❖ ALU can perform arithmetic and logic functions on 8 bit variables.

- ❖ 8051 has 128 bytes of internal RAM which is divided into

    - ♦ Working registers [00 – 1F]
    - ♦ Bit addressable memory area [20 – 2F]
    - ♦ General purpose memory area (Scratch pad memory) [30-7F]

2.2 THE 8051 ARCHITECTURE.
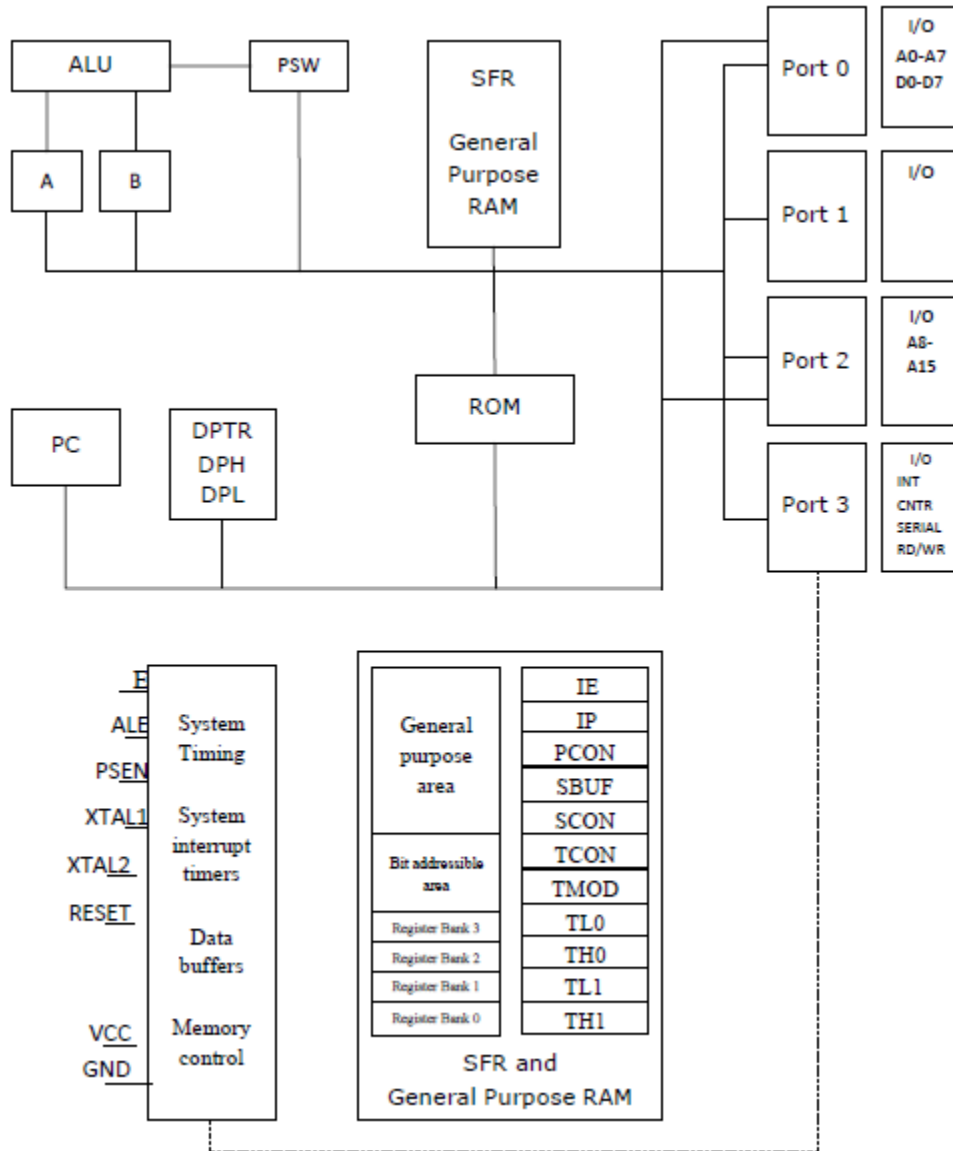
- ❖ 8051 has 4 K Bytes of internal ROM. The address space is from 0000 to 0FFFh. If the program size is more than 4 K Bytes 8051 will fetch the code automatically from external memory.
- ❖ Accumulator is an 8 bit register widely used for all arithmetic and logical operations. Accumulator is also used to transfer data between external memory. B register is used along with Accumulator for multiplication and division. A and B

registers together is also called MATH registers.

❖ PSW (Program Status Word). This is an 8 bit register which contains the arithmetic status ofALU and the bank select bits of register banks.

CY	AC	F0	RS1	RS0	OV	-	P

CY   -   carry flag

AC   -   auxiliary carry flag

F0   -   Available for user for general purpose

RS1,RS0 -   register bank select bits

OV   -   overflow

P   -   parity

❖ Stack Pointer (SP) – it contains the address of the data item on the top of the stack. Stack may reside anywhere on the internal RAM. On reset, SP is initialized to 07 so that the defaultstack will start from address 08 onwards.

❖ Data Pointer (DPTR) – DPH (Data pointer higher byte), DPL (Data pointer lower byte). This is a 16 bit register which is used to furnish address information for internal and external program memory and for external data memory.

❖ Program Counter (PC) – 16 bit PC contains the address of next instruction to be executed. On reset PC will set to 0000. After fetching every instruction PC will increment by one.

2.3 PIN DIAGRAM

PINOUT DESCRIPTION

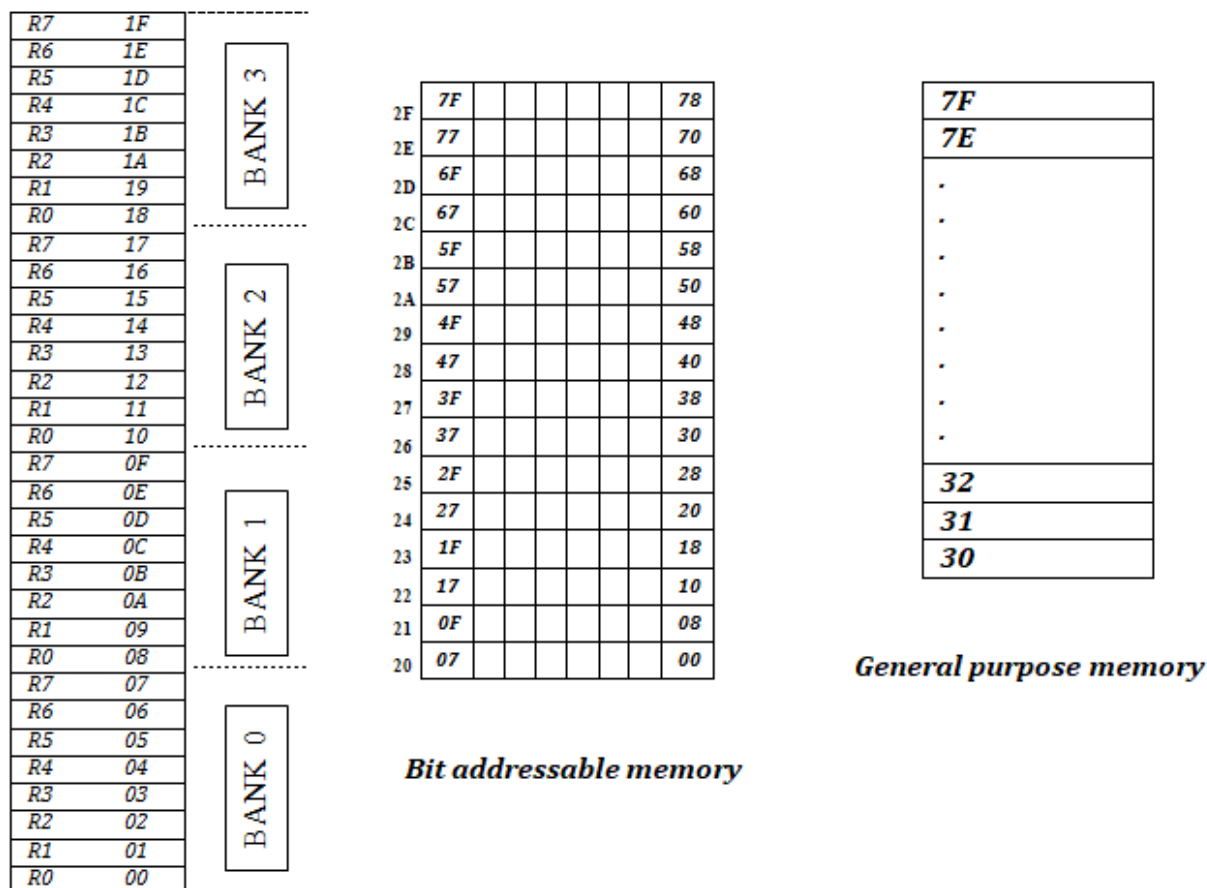|	
Pins 1-8	**PORT 1**. Each of these pins can be configured as an input or an output.
Pin 9	**RESET**. A logic one on this pin disables the microcontroller and clears the contents of most registers. In other words, the positive voltage on this pin resets the microcontroller. By applying logic zero to this pin, the program starts execution from the beginning.
Pins10-17	**PORT 3**. Similar to port 1, each of these pins can serve as general input or output. Besides, all of them have alternative functions
Pin 10	**RXD. Serial asynchronous communication input or Serial synchronous communication output.**
Pin 11	**TXD. Serial asynchronous communication output or Serial synchronous communication clock output.**
Pin 12	**INT0.External Interrupt 0 input**
Pin 13	**INT1. External Interrupt 1 input**
Pin 14	**T0. Counter 0 clock input**
Pin 15	**T1. Counter 1 clock input**
Pin 16	**WR. Write to external (additional) RAM**
Pin 17	**RD. Read from external RAM**
Pin 18, 19	**XTAL2, XTAL1. Internal oscillator input and output. A quartz crystal which specifies operating frequency is usually connected to these pins.**
Pin 20	**GND. Ground.**
Pin 21-28	**Port 2. If there is no intention to use external memory then these port pins are configured as general inputs/outputs. In case external memory is used, the higher address byte, i.e. addresses A8-A15 will appear on this port. Even though memory with capacity of 64Kb is not used, which means that not all eight port bits are used for its addressing, the rest of them are not available as inputs/outputs.**
Pin 29	**PSEN. If external ROM is used for storing program then a logic zero (0) appears on it every time the microcontroller reads a byte from memory.**
Pin 30	**ALE. Prior to reading from external memory, the microcontroller puts the lower address byte (A0-A7) on P0 and activates the ALE output. After receiving signal from the ALE pin, the external latch latches the state of P0 and uses it as a**

	memory chip address. Immediately after that, the ALE pin is returned its previous logic state and P0 is now used as a Data Bus.
Pin 31	EA. By applying logic zero to this pin, P2 and P3 are used for data and address transmission with no regard to whether there is internal memory or not. It means that even there is a program written to the microcontroller, it will not be executed. Instead, the program written to external ROM will be executed. By applying logic one to the EA pin, the microcontroller will use both memories, first internal then external (if exists).
Pin 32-39	PORT 0. Similar to P2, if external memory is not used, these pins can be used as general inputs/outputs. Otherwise, P0 is configured as address output (A0-A7) when the ALE pin is driven high (1) or as data output (Data Bus) when the ALE pin is driven low (0).
Pin 40	VCC. +5V power supply.

2.4 MEMORY ORGANIZATION

Internal RAM organization

Bit addressable memory

General purpose memory

Working Registers

Register Banks: 00h to 1Fh. The 8051 uses 8 general-purpose registers R0 through R7 (R0, R1, R2, R3, R4, R5, R6, and R7). There are four such register banks. Selection of register bank can be done through RS1,RS0 bits of PSW. On reset, the default Register Bank 0 will be selected.

Bit Addressable RAM: 20h to 2Fh . The 8051 supports a special feature which allows access to bit variables. This is where individual memory bits in Internal RAM can be set or cleared. In all there are 128 bits numbered 00h to 7Fh. Being bit variables any one variable can have a value 0 or 1. A bit variable can be set with a command such as SETB and cleared with a command such as CLR. Example instructions are:

SETB 25h ; sets the bit 25h (becomes 1)
CLR 25h ; clears bit 25h (becomes 0)

Note, bit 25h is actually bit 5 of Internal RAM location 24h.

The Bit Addressable area of the RAM is just 16 bytes of Internal RAM located between 20h and 2Fh.

General Purpose RAM: 30h to 7Fh. Even if 80 bytes of Internal RAM memory are available for general-purpose data storage, user should take care while using the memory location from 00 -2Fh since these locations are also the default register space, stack space, and bit addressable space. It is a good practice to use general purpose memory from 30 – 7Fh. The general purpose RAM can be accessed using direct or indirect addressing modes.

EXTERNAL MEMORY INTERFACING
Eg. Interfacing of 16 K Byte of RAM and 32 K Byte of EPROM to 8051

Number of address lines required for **16 Kbyte memory is 14 lines** and that **of 32Kbytes of memory is 15 lines.**

The connections of external memory is shown in figure. The lower order address and data bus are multiplexed. De-multiplexing is done by the latch. Initially the address will appear in the bus and this latched at the output of latch using ALE signal. The output of the latch is directly connected to the lower byte address lines of the memory. Later data will be available in this bus. Still the latch output is address it self. The higher byte of address bus is directly connected to the memory. The number of lines connected depends on the memory size.

The RD and WR (both active low) signals are connected to RAM for reading and writing the data. PSEN of microcontroller is connected to the output enable of the ROM to read the data from the memory.

EA (active low) pin is always grounded if we use only external memory. Otherwise, once the program size exceeds internal memory the microcontroller will automatically switch to external memory.
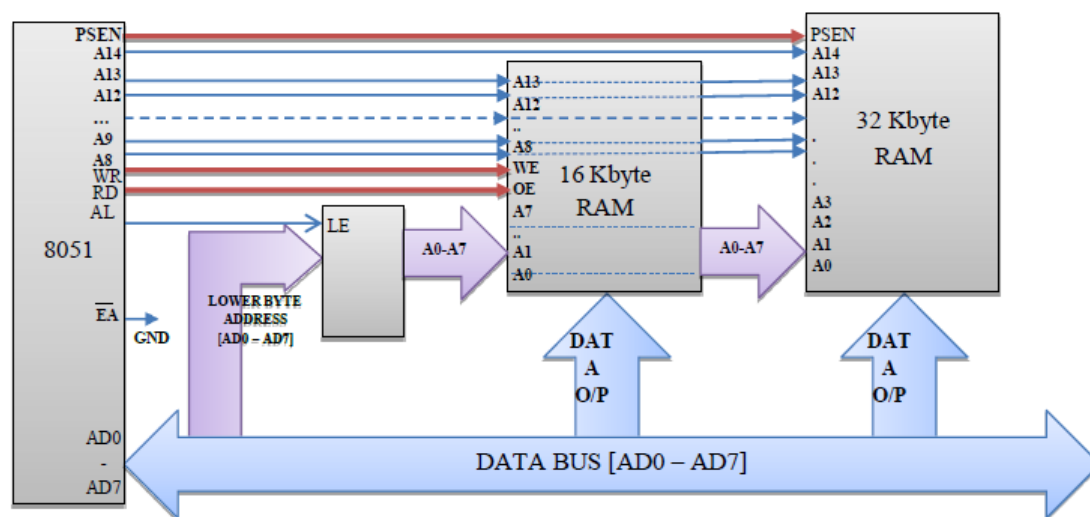
STACK

A stack is a last in first out memory. In 8051 internal RAM space can be used as stack. The address of the stack is contained in a register called stack pointer. Instructions PUSH and POP are used for stack operations. When a data is to be placed on the stack, the stack pointer increments before storing the data on the stack so that the stack grows up as data is stored (pre-increment). As the data is retrieved from the stack the byte is read from the stack, and then SP decrements to point the next available byte of stored data (post decrement). The stack pointer is set to 07 when the 8051 resets. So that default stack memory starts from address location 08 onwards (to avoid overwriting the default register bank ie., bank 0).

Eg; Show the stack and SP for the following.

```
                          [SP]=07        //CONTENT OF SP IS 07 (DEFAULT VALUE)
MOV R6, #25H              [R6]=25H       //CONTENT OF R6 IS 25H
MOV R1, #12H              [R1]=12H       //CONTENT OF R1 IS 12H
MOV R4, #0F3H            [R4]=F3H        //CONTENT OF R4 IS F3H


PUSH 6                   [SP]=08       [08]=[06]=25H        //CONTENT OF 08 IS 25H
PUSH 1                   [SP]=09       [09]=[01]=12H        //CONTENT OF 09 IS 12H
PUSH 4                   [SP]=0A       [0A]=[04]=F3H        //CONTENT OF 0A IS F3H

POP 6               [06]=[0A]=F3H          [SP]=09 //CONTENT OF 06 IS F3H
POP 1               [01]=[09]=12H          [SP]=08 //CONTENT OF 01 IS 12H
POP 4               [04]=[08]=25H          [SP]=07 //CONTENT OF 04 IS 25H
```

2.5 BASICS OF INTERRUPTS.

During program execution if peripheral devices needs service from microcontroller, device will generate interrupt and gets the service from microcontroller. When peripheral device activate the interrupt signal, the processor branches to a program called interrupt service routine. After executing the interrupt service routine the processor returns to the main program.

Steps taken by processor while processing an interrupt:

1. *It completes the execution of the current instruction.*
2. *PSW is pushed to stack.*
3. *PC content is pushed to stack.*
4. *Interrupt flag is reset.*
5. *PC is loaded with ISR address.*

ISR will always ends with RETI instruction. The execution of RETI instruction results in the following.

1. *POP the current stack top to the PC.*
2. *POP the current stack top to PSW.*

Classification of interrupts.

1. *External and internal interrupts.*
   External interrupts are those initiated by peripheral devices through the external pins of the microcontroller.
   Internal interrupts are those activated by the internal peripherals of the microcontroller like timers, serial controller etc.)
2. *Maskable and non-maskable interrupts.*
   The category of interrupts which can be disabled by the processor using program is called maskable interrupts.
   Non-maskable interrupts are those category by which the programmer cannot disable it using program.
3. *Vectored and non-vectored interrupt.*
   Starting address of the ISR is called interrupt vector. In vectored interrupts the starting address is predefined. In non-vectored interrputs, the starting address is provided by the peripheral as follows.
   - Microcontroller receives an interrupt request from external device.
   - Controller sends an acknowledgement (**INTA**) after completing the execution of current instruction.
   - The peripheral device sends the interrupt vector to the microcontroller.

2.6 8051 INTERRUPT STRUCTURE.

8051 has five interrupts. They are maskable and vectored interrupts. Out of these five, two are external interrupt and three are internal interrupts.

Interrupt source	Type	Vector address	Priority
External interrupt 0	External	0003	Highest
Timer 0 interrupt	Internal	000B	
External interrupt 1	External	0013	
Timer 1 interrupt	Internal	001B	
Serial interrupt	Internal	0023	Lowest

8051 makes use of two registers to deal with interrupts.

6. **IE Register**

   This is an 8 bit register used for enabling or disabling the interrupts. The structure of IE register is shown below.

   IE : Interrupt Enable Register (Bit Addressable)

   If the bit is 0, the corresponding interrupt is disabled. If the bit is 1, the corresponding interrupt is enabled.

EA	–	–	ES	ET1	EX1	ET0	EX0

EA	IE.7	Disables all interrupts. If EA = 0, no interrupt will be acknowledged. If EA = 1, interrupt source is individually enable or disabled by setting or clearing its enable bit.
-	IE.6	Not implemented, reserved for future use*.
-	IE.5	Not implemented, reserved for future use*.
ES	IE.4	Enable or disable the Serial port interrupt.
ET1	IE.3	Enable or disable the Timer 1 overflow interrupt.
EX1	IE.2	Enable or disable External interrupt 1.
ET0	IE.1	Enable or disable the Timer 0 overflow interrupt.
EX0	IE.0	Enable or disable External Interrupt 0.

7. **IP Register.**

   This is an 8 bit register used for setting the priority of the interrupts.

   IP : Interrupt Priority Register (Bit Addressable)

   If the bit is 0, the corresponding interrupt has a lower priority and if the bit is the corresponding interrupt has a higher priority.

–	–	–	PS	PT1	PX1	PT0	PX0

-	IP.7	Not implemented, reserved for future use*.
-	IP.6	Not implemented, reserved for future use*.
-	IP.5	Not implemented, reserved for future use*.
PS	IP.4	Defines the Serial Port interrupt priority level.
PT1	IP.3	Defines the Timer 1 Interrupt priority level.
PX1	IP.2	Defines External Interrupt priority level.
PT0	IP.1	Defines the Timer 0 interrupt priority level.
PX0	IP.0	Defines the External Interrupt 0 priority level.

2.7 TIMERS AND COUNTERS

Timers/Counters are used generally for

- Time reference
- Creating delay
- Wave form properties measurement
- Periodic interrupt generation
- Waveform generation

8051 has two timers, Timer 0 and Timer 1.

Timer in 8051 is used as timer, counter and baud rate generator. Timer always counts up irrespective of whether it is used as timer, counter, or baud rate generator: Timer is always incremented by the microcontroller. The time taken to count one digit up is based on master clock frequency.

If Master CLK=12 MHz,

Timer Clock frequency = Master CLK/12 = 1 MHz
Timer Clock Period = 1micro second

This indicates that one increment in count will take 1 micro second.

The two timers in 8051 share two SFRs (TMOD and TCON) which control the timers, and each timer also has two SFRs dedicated solely to itself (TH0/TL0 and TH1/TL1).

The following are timer related SFRs in 8051.

SFR Name	Description	SFR Address
TH0	*Timer 0 High Byte*	*8Ch*
TL0	*Timer 0 Low Byte*	*8Ah*
TH1	*Timer 1 High Byte*	*8Dh*
TL1	*Timer 1 Low Byte*	*8Bh*
TCON	*Timer Control*	*88h*
TMOD	*Timer Mode*	*89h*

TMOD Register

TMOD : Timer/Counter Mode Control Register (Not Bit Addressable)

GATE	C/T̄	M1	M0	GATE	C/T̄	M1	M0

| TIMER 1 | | | | TIMER 0 | | | |

GATE   When TRx (in TCON) is set and GATE = 1, TIMER/COUNTERx will run only while INTx pin is high (hardware control). When GATE = 0, TIMER/COUNTERx will run only while TRx = 1 (software control).

C/T̄   Timer or Counter selector. Cleared for Timer operation (input from internal system clock). Set for Counter operation (input from Tx input pin).

M1   Mode selector bit (NOTE 1).

M0   Mode selector bit (NOTE 1).

Note 1 :

M1	M0		OPERATING MODE
0	0	0	13–bit Timer
0	1	1	16–bit Timer/Counter
1	0	2	8–bit Auto–Reload Timer/Counter
1	1	3	(Timer 0) TL0 is an 8–bit Timer/Counter controlled by the standard Timer 0 control bits, TH0 is an 8–bit Timer and is controlled by Timer 1 control bits.
1	1	3	(Timer 1) Timer/Counter 1 stopped.

TCON Register

TCON : Timer/Counter Control Register (Bit Addressable)

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TF1   TCON.7   Timer 1 overflow flag. Set by hardware when the Timer/Counter 1 overflows. Cleared by hardware as processor vectors to the interrupt service routine.

TR1   TCON.6   Timer 1 run control bit. Set/cleared by software to turn Timer/Counter ON/OFF.

TF0   TCON.5   Timer 0 overflow flag. Set by hardware when the Timer/Counter 0 overflows. Cleared by hardware as 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 hardware when External interrupt edge is detected. Cleared by hardware when interrupt is processed.

IT1   TCON.2   Interrupt 1 type control bit. Set/cleared by software to specify falling edge/flow level triggered External Interrupt.
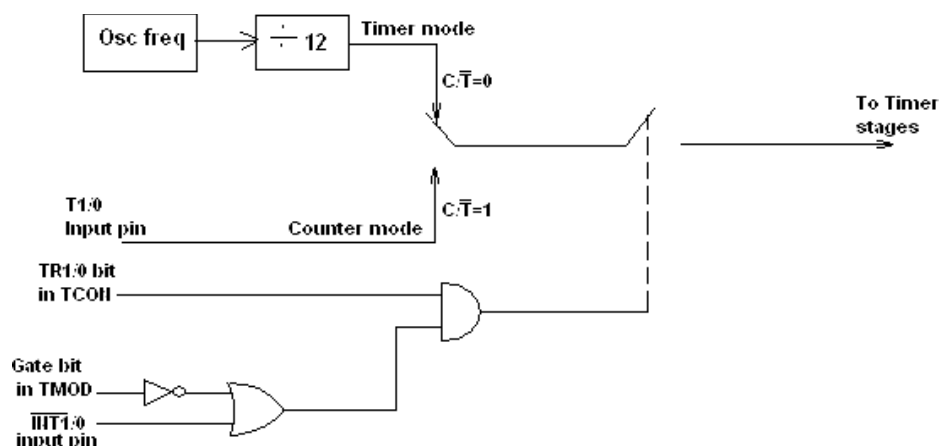
IE0   TCON.1   External Interrupt 0 edge flag. Set by hardware when External Interrupt edge detected. Cleared by hardware when interrupt is processed.

IT0   TCON.0   Interrupt 0 type control bit. Set/cleared by software to specify falling edge/low level triggered External Interrupt.
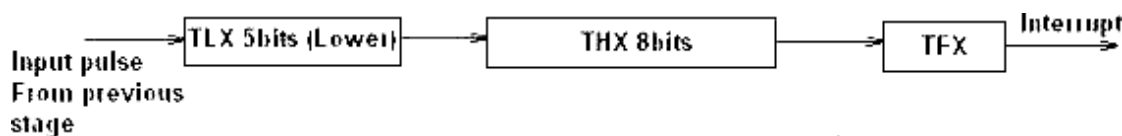
Timer/ Counter Control Logic.

TIMER MODES

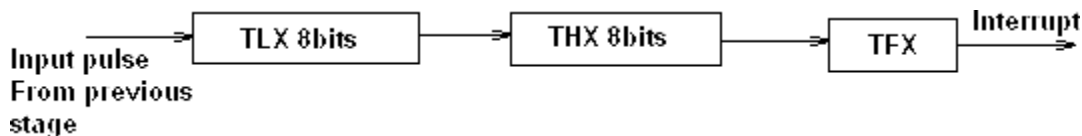Timers can operate in four different modes. They are as follows

Timer Mode-0: In this mode, the timer is used as a 13-bit UP counter as follows.

Fig. Operation of Timer on Mode-0

The lower 5 bits of TLX and 8 bits of THX are used for the 13 bit count.Upper 3 bits of TLX are ignored. When the counter rolls over from all 0's to all 1's, TFX flag is set and an interrupt is generated. The input pulse is obtained from the previous stage. If TR1/0 bit is 1 and Gate bit is 0, the counter continues counting up. If TR1/0 bit is 1 and Gate bit is 1, then the operation of the counter is controlled by input. This mode is useful to measure the width of a given pulse fed to input.

Timer Mode-1: This mode is similar to mode-0 except for the fact that the Timer operates in 16-bit mode.

Fig: Operation of Timer in Mode 1

Timer Mode-2: (Auto-Reload Mode): This is a 8 bit counter/timer operation. Counting is performed in TLX while THX stores a constant value. In this mode when the timer overflows i.e. TLX becomes FFH, it is fed with the value stored in THX. For example if we load THX with 50H then the

timer in mode 2 will count from 50H to FFH. After that 50H is again reloaded. This mode is useful in applications like fixed time sampling.

Fig: Operation of Timer in Mode 2

Timer Mode-3: Timer 1 in mode-3 simply holds its count. The effect is same as setting TR1=0. Timer0 in mode-3 establishes TL0 and TH0 as two separate counters.

Fig: Operation of Timer in Mode 3

Control bits TR1 and TF1 are used by Timer-0 (higher 8 bits) (TH0) in Mode-3 while TR0 and TF0 are available to Timer-0 lower 8 bits(TL0).

2.8 ASSEMBLY LANGUAGE PROGRAMMING

Programming in the sense of Microcontrollers (or any computer) means writing a sequence of instructions that are executed by the processor in a particular order to perform a predefined task. Programming also involves debugging and troubleshooting of instructions and instruction sequence to make sure that the desired task is performed.

Like any language, Programming Languages have certain words, grammar and rules. There are three types or levels of Programming Languages for 8051 Microcontroller. These levels are based

on how closely the statements in the language resemble the operations or tasks performed by the Microcontroller.

The three levels of Programming Languages are:

- ❖ Machine Language
- ❖ Assembly Language
- ❖ High-level Language
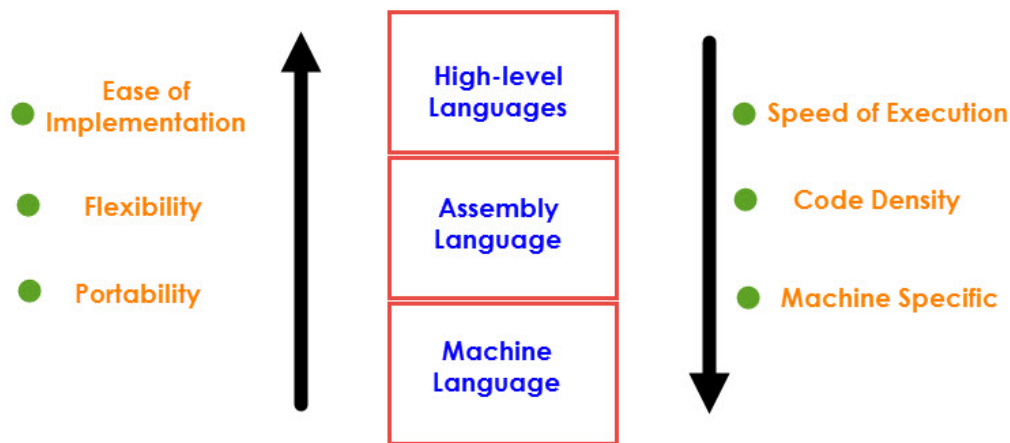
MACHINE LANGUAGE

In Machine language or Machine Code, the instructions are written in binary bit patterns i.e. combination of binary digits 1 and 0, which are stored as HIGH and LOW Voltage Levels. This is the lowest level of programming languages and is the language that a Microcontroller or Microprocessor actually understands.

ASSEMBLY LANGUAGE

The next level of Programming Language is the Assembly Language. Since Machine Language or Code involves all the instructions in 1's and 0's, it is very difficult for humans to program using it. Assembly Language is a pseudo-English representation of the Machine Language. The 8051 Microcontroller Assembly Language is a combination of English like words called Mnemonics and Hexadecimal codes.
It is also a low level language and requires extensive understanding of the architecture of the Microcontroller.

HIGH-LEVEL LANGUAGE

The name High-level language means that you need not worry about the architecture or other internal details of a microcontroller and they use words and statements that are easily understood by humans.

Few examples of High-level Languages are BASIC, C Pascal, C++ and Java. A program called Compiler will convert the Programs written in High-level languages to Machine Code.

Why Assembly Language?

Although High-level languages are easy to work with, the following reasons point out the advantage of Assembly Language

* ❖ The Programs written in Assembly gets executed faster and they occupy less memory.
* ❖ With the help of Assembly Language, you can directly exploit all the features of a Microcontroller.
* ❖ Using Assembly Language, you can have direct and accurate control of all the Microcontroller's resources like I/O Ports, RAM, SFRs, etc.
* ❖ Compared to High-level Languages, Assembly Language has less rules and restrictions.

STRUCTURE OF THE 8051 MICROCONTROLLER ASSEMBLY LANGUAGE

The Structure or Syntax of the 8051 Microcontroller Assembly Language is discussed here. Each line or statement of the assembly language program of 8051 Microcontroller consists of three fields: Label, Instruction and Comments.

The arrangement of these fields or the order in which they appear is shown below.

[Label:]   Instructions   [//Comments]

The brackets for Label and Comments mean that these fields are optional and may not be used in all statements in a program.

Before seeing about these three fields, let us first see an example of how a typical statement or line in an 8051 Microcontroller Assembly Language looks like.

TESTLABEL: MOV A, 24H   ; THIS IS A SAMPLE COMMENT

In the above statement, the "TESTLABEL" is the name of the Label, "MOV A, 24H" is the Instruction and the "THIS IS A SAMPLE COMMENT" is a Comment.

LABEL

The Label is programmer chosen name for a Memory Location or a statement in a program. The Label part of the statement is optional and if present, the Label must be terminated with a Colon (:). An important point to remember while selecting a name for the Label is that they should reduce the need for documentation.

INSTRUCTION

The Instruction is the main part of the 8051 Microcontroller Assembly Language Programming as it is responsible for the task performed by the Microcontroller. Any Instruction in the Assembly Language consists of two parts: Op-code and Operand(s).

The first part of the Instruction is the Op-code, which is short for Operation Code, specifies the operation to be performed by the Microcontroller. Op-codes in Assembly Language are called as Mnemonics. Op-codes are in binary format (used in Machine Language) while the Mnemonic (which are equivalent to Op-codes) are English like statements.
The second part of the instruction is called the Operand(s) and it represents the Data on which the operation is performed. There are two types of Operands: the Source Operand and the Destination Operand. The Source Operand is the Input of the operation and the Destination Operand is where the result is stored.

COMMENTS

The last part of the Structure of 8051 Assembly Language is the Comments. Comments are statements included by the developer for easier understanding of the code and is used for proper documentation of the Program.
Comments are optional and if used, they must begin with a semicolon (;) or double slash (//) depending on the Assembler.

2.9 ADDRESSING MODES

*Various methods of accessing the data are called addressing modes.*8051 addressing modes are

classified as follows.

1. Immediate addressing.
2. Register addressing.
3. Direct addressing.
4. Indirect addressing.
5. Relative addressing.
6. Absolute addressing.
7. Long addressing.
8. Indexed addressing.
9. Bit inherent addressing.
10. Bit direct addressing.

I.   IMMEDIATE ADDRESSING.

In this addressing mode the data is provided as a part of instruction itself. In other wordsdata immediately follows the instruction.

Eg.   MOV A,#30H

ADD A, #83   # Symbol indicates the data is immediate.

II.   REGISTER ADDRESSING.

In this addressing mode the register will hold the data. One of the eight general registers (R0 to R7) can be used and specified as the operand.

Eg.   MOV A,R0
ADD A,R6

R0 – R7 will be selected from the current selection of register bank. The default register bank will be bank 0.

III.   DIRECT ADDRESSING

There are two ways to access the internal memory. Using direct address and indirect address. Using direct addressing mode we can not only address the internal memory but SFRs also. In direct addressing, an 8 bit internal data memory address is specified as part of the instruction and hence, it can specify the address only in the range of 00H to FFH. In this addressing mode, data is obtained directly from the memory.

Eg.   MOV A,60h
ADD A,30h

IV.   INDIRECT ADDRESSING

The indirect addressing mode uses a register to hold the actual address that will be used in data movement. Registers R0 and R1 and DPTR are the only registers that can be used as data pointers. Indirect addressing cannot be used to refer to SFR registers. Both R0 and R1 can hold 8 bit address and DPTR can hold 16 bit address.

Eg.   MOV   A,@R0
        ADD   A,@R1
        MOVX A,@DPTR

V.   INDEXED ADDRESSING.

In indexed addressing, either the program counter (PC), or the data pointer (DTPR)—is used to hold the base address, and the A is used to hold the offset address. Adding the value of the base address to the value of the offset address forms the effective address. Indexed addressing is used with JMP or MOVC instructions. Look up tables are easily implemented with the help of index addressing.

Eg.   MOVC A, @A+DPTR // *copies the contents of memory location pointed by the sum of the accumulator A and the DPTR into accumulator A.*

MOVC A, @A+PC   // *copies the contents of memory location pointed by the sum of the accumulator A and the program counter into accumulator A.*

VI.   RELATIVE ADDRESSING.

Relative addressing is used only with conditional jump instructions. The relative address, (offset), is an 8 bit signed number, which is automatically added to the PC to make the address of the next instruction. The 8 bit signed offset value gives an address range of +127 to —128 locations. The jump destination is usually specified using a label and the assembler calculates the jump offset accordingly. The advantage of relative addressing is that the program code is easy to relocate and the address is relative to position in the memory.

Eg.   SJMP LOOP1
        JC BACK

VII.   ABSOLUTE ADDRESSING

Absolute addressing is used only by the AJMP (Absolute Jump) and ACALL (Absolute Call) instructions. These are 2 bytes instructions. The absolute addressing mode specifies the lowest 11 bit of the memory address as part of the instruction. The upper 5 bit of the destination address are the upper 5 bit of the current program counter. Hence, absolute addressing allows branching onlywithin the current 2 Kbyte page of the program memory.

Eg.   AJMP LOOP1
        ACALL LOOP2

VIII.   LONG ADDRESSING

The long addressing mode is used with the instructions LJMP and LCALL. These are 3 byte instructions. The address specifies a full 16 bit destination address so that a jump or a call can be made to a location within a 64 Kbyte code memory space.

Eg.   LJMP FINISH
        LCALL DELAY

IX.   BIT INHERENT ADDRESSING

In this addressing, the address of the flag which contains the operand, is implied in the opcode of the instruction.

Eg.   CLR C   ;   *Clears the carry flag to 0*

X.   BIT DIRECT ADDRESSING

In this addressing mode the direct address of the bit is specified in the instruction. The RAM space 20H to 2FH and most of the special function registers are bit addressable. Bit address values are between 00H to 7FH.

Eg.   CLR 07h   ;   Clears the bit 7 of 20h RAM space

        SETB 07H   ;   Sets the bit 7 of 20H RAM space.

2.10 INSTRUCTION SET

1. INSTRUCTION TIMINGS

The 8051 internal operations and external read/write operations are controlled by the oscillator clock.

T-state, Machine cycle and Instruction cycle are terms used in instruction timings.

T-state is defined as one subdivision of the operation performed in one clock period. The terms 'T-state' and 'clock period' are often used synonymously.

Machine cycle is defined as 12 oscillator periods. A machine cycle consists of six states and each state lasts for two oscillator periods. An instruction takes one to four machine cycles to execute an instruction. *Instruction cycle* is defined as the time required for completing the execution of an instruction. The 8051 instruction cycle consists of one to four machine cycles.

Eg. If 8051 microcontroller is operated with 12 MHz oscillator, find the execution time for the following four instructions.

1. ADD A, 45H
2. SUBB A, #55H
3. MOV DPTR, #2000H
4. MUL AB

*Since the oscillator frequency is 12 MHz, the clock period is, Clock period = 1/12 MHz = **0.08333 µS**.*
*Time for 1 machine cycle = 0.08333 µS x 12 =**1 µS**.*

Instruction	No. of machine cycles	Execution time
1. ADD A, 45H	1	1 µs
2. SUBB A, #55H	2	2 µs
3. MOV DPTR, #2000H	2	2 µs
4. MUL AB	4	4 µs

2. 8051 INSTRUCTIONS

The instructions of 8051 can be broadly classified under the following headings.
  ❖ Data transfer instructions
  ❖ Arithmetic instructions
  ❖ Logical instructions
  ❖ Branch instructions
  ❖ Subroutine instructions
  ❖ Bit manipulation instructions

Data transfer instructions.

In this group, the instructions perform data transfer operations of the following types.
  a. Move the contents of a register Rn to A
      i. MOV A,R2
      ii. MOV A,R7
  b. Move the contents of a register A to Rn
      i. MOV R4,A
      ii. MOV R1,A
  c. Move an immediate 8 bit data to register A or to Rn or to a memory location(direct orindirect)
      i. MOV A, #45H

    ii. MOV R6, #51H

    iii. MOV 30H, #44H

d. Move the contents of a memory location to A or A to a memory location using direct andindirect addressing

    i. MOV A, 65H

    ii. MOV A, @R0

    iii. MOV 45H, A

    iv. MOV @R1, A

e. Move the contents of a memory location to Rn or Rn to a memory location using directaddressing
   i. MOV R3, 65H
   ii. MOV 45H, R2
f. Move the contents of memory location to another memory location using direct andindirect addressing
   i. MOV 47H, 65H
   ii. MOV 45H, @R0
g. Move the contents of an external memory to A or A to an external memory
   i. MOVX A,@R1
   ii. MOVX @R0,A
   iii. MOVX A,@DPTR
   iv. MOVX@DPTR,A
h. Move the contents of program memory to A
   i. MOVC A, @A+PC
   ii. MOVC A, @A+DPTR

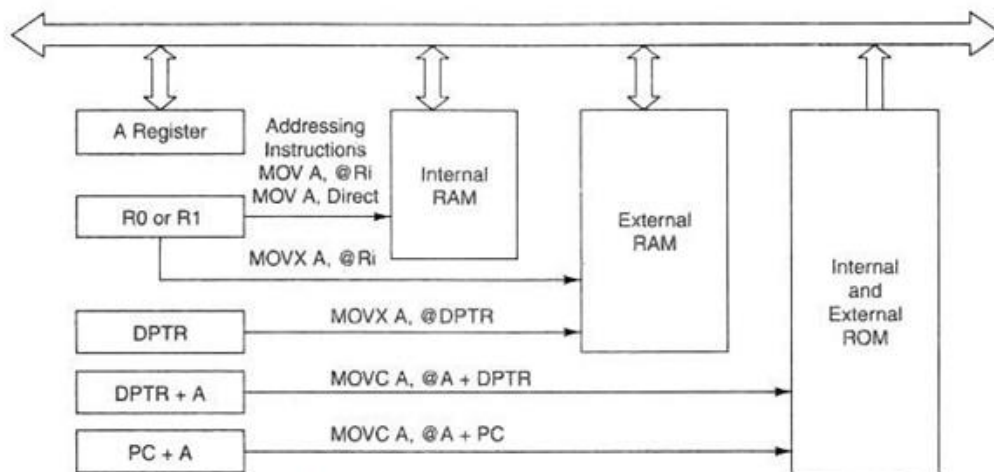
FIG. Addressing Using MOV, MOVX and MOVC

i. Push and Pop instructions

   [SP]=07   //CONTENT OF SP IS 07 (DEFAULT
   VALUE)MOV R6, #25H   [R6]=25H //CONTENT OF R6 IS
   25H

MOV R1, #12H [R1]=12H   //CONTENT OF R1 IS
12HMOV R4, #0F3H   [R4]=F3H
                    //CONTENT OF R4 IS F3H

PUSH 6   [SP]=08   [08]=[06]=25H //CONTENT OF 08
IS 25HPUSH 1   [SP]=09   [09]=[01]=12H //CONTENT OF 09
IS 12HPUSH 4   [SP]=0A   [0A]=[04]=F3H //CONTENT OF 0A
IS F3H

POP 6   [06]=[0A]=F3H [SP]=09   //CONTENT OF 06 IS
F3HPOP 1   [01]=[09]=12H [SP]=08   //CONTENT OF 01 IS
12HPOP 4   [04]=[08]=25H [SP]=07   //CONTENT OF 04 IS
25H

j. Exchange instructions
The content of source ie., register, direct memory or indirect memory will be exchangedwith the contents of destination ie., accumulator.
   i. XCH A,R3
   ii. XCH A,@R1
   iii. XCH A,54h
k. Exchange digit. Exchange the lower order nibble of Accumulator (A0-A3) with lower order nibble of the internal RAM location which is indirectly addressed by the register.
   i. XCHD A,@R1
   ii. XCHD A,@R0

Arithmetic instructions.

The 8051 can perform addition, subtraction. Multiplication and division operations on 8 bit numbers.

Addition
In this group, we have instructions to
   i. Add the contents of A with immediate data with or without carry.
      i. ADD A, #45H
      ii. ADDC A, #0B4H
   ii. Add the contents of A with register Rn with or without carry.
      i. ADD A, R5
      ii. ADDC A, R2

   iii. Add the contents of A with contents of memory with or without carry using
direct andindirect addressing
     i. ADD A, 51H
    ii. ADDC A, 75H
   iii. ADD A, @R1
   iv. ADDC A, @R0

CY AC and OV flags will be affected by this operation

Subtraction

In this group, we have instructions to
   i. Subtract the contents of A with immediate data with or without carry.
     i. SUBB A, #45H
    ii. SUBB A, #OB4H
  ii. Subtract the contents of A with register Rn with or without borrow.
     i. SUBB A, R5
    ii. SUBB A, R2
  iii. Subtract the contents of A with contents of memory with or without carry using
direct andindirect addressing
     i. SUBB A, 51H
    ii. SUBB A, 75H
   iii. SUBB A, @R1
   iv. SUBB A, @R0

CY AC and OV flags will be affected by this operation.

Multiplication

MUL AB. This instruction multiplies two 8 bit unsigned numbers which are stored in A and B register. After multiplication the lower byte of the result will be stored in accumulator and higher byte of result will be stored in B register.

```
MOV A,#45H      ;[A]=45H
MOV B,#0F5H     ;[B]=F5H
MUL AB          ;[A] x [B] = 45 x F5 = 4209
                ;[A]=09H, [B]=42H
```

Division

DIV AB. This instruction divides the 8 bit unsigned number which is stored in A by the 8 bit unsigned number which is stored in B register. After division the result will be stored in accumulator and remainder will be stored in B register.

```
Eg.   MOV A,#45H     ;[A]=0E8H
      MOV B,#0F5H    ;[B]=1BH
      DIV AB          ;[A] / [B] = E8 /1B = 08 H with remainder 10H
                      ;[A] = 08H, [B]=10H
```

DA A (Decimal Adjust After Addition).

When two BCD numbers are added, the answer is a non-BCD number. To get the result in BCD, weuse DA A instruction after the addition. DA A works as follows.

❖   If lower nibble is greater than 9 or auxiliary carry is 1, 6 is added to lower nibble.
❖   If upper nibble is greater than 9 or carry is 1, 6 is added to upper nibble.

```
Eg 1:   MOV A,#23H
        MOV R1,#55H
        ADD A,R1    // [A]=78
        DA A        // [A]=78   no changes in the accumulator after DA A

Eg 2:  MOV A,#53H
        MOV R1,#58H
        ADD A,R1    // [A]=ABh
        DA A         // [A]=11, C=1 . ANSWER IS 111. Accumulator data is changed
        after DA A
```

Increment: *increments the operand by one.*

INC A   INC Rn   INC DIRECT   INC @RiINC DPTR

INC increments the value of source by 1. If the initial value of register is FFh, incrementing the value will cause it to reset to 0. The Carry Flag is not set when the value "rolls over" from 255 to 0.

In the case of "INC DPTR", the value two-byte unsigned integer value of DPTR is incremented. If the initial value of DPTR is FFFFh, incrementing the value will cause it to reset to 0.

Decrement: decrements the operand by one.

DEC A   DEC Rn DEC DIRECT   DEC @Ri

DEC decrements the value of *source* by 1. If the initial value of is 0, decrementing the value will cause it to reset to FFh. The Carry Flag is not set when the value "rolls over" from 0 to FFh.

Logical Instructions

Logical AND

ANL destination, source: ANL does a bitwise "AND" operation between *source* and *destination*, leaving the resulting value in *destination*. The value in source is not affected. "AND" instruction logically AND the bits of source and destination.

> **ANL A,#DATA**
> **ANL A, Rn ANL**
> **A,DIRECT ANL**
> **A,@Ri**
>
> **ANL DIRECT,A ANL DIRECT, #DATA**

Logical OR

ORL destination, source:ORL does a bitwise "OR" operation between *source* and *destination*,

leaving the resulting value in *destination*. The value in source is not affected. " OR " instruction logically OR the bits of source and destination.

> **ORL A,#DATA**
> **ORL A, Rn ORL**
> **A,DIRECT ORL**
> **A,@Ri**
>
> **ORL DIRECT,A ORL DIRECT, #DATA**

Logical Ex-OR

XRL destination, source: XRL does a bitwise "EX-OR" operation between *source* and *destination*, leaving the resulting value in *destination*. The value in source is not affected. " XRL "instruction logically EX-OR the bits of source and destination.

> **XRL A,#DATA XRL**
> **A,Rn XRL A,DIRECT**
> **XRL A,@Ri**
>
> **XRL DIRECT,A XRL DIRECT, #DATA**

Logical NOT

CPL complements *operand*, leaving the result in *operand*. If *operand* is a single bit then the state ofthe bit will be reversed. If *operand* is the Accumulator then all the bits in the Accumulator will be reversed.

CPL A,   CPL C,   CPL bit address

SWAP A – Swap the upper nibble and lower nibble of A.

Rotate Instructions

RR A

This instruction is rotate right the accumulator. Its operation is illustrated below. Each bit is shifted one location to the right, with bit 0 going to bit 7.
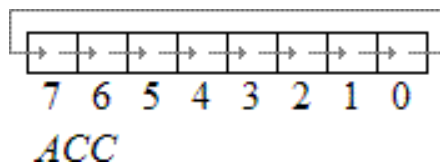
RL A

Rotate left the accumulator. Each bit is shifted one location to the left, with bit 7 going to bit 0
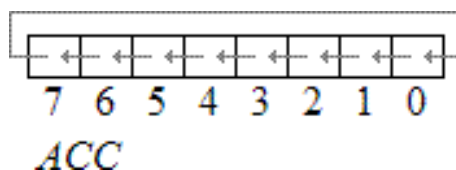
RRC A

Rotate right through the carry. Each bit is shifted one location to the right, with bit going into the carry bit in the PSW, while the carry was at goes into bit 7
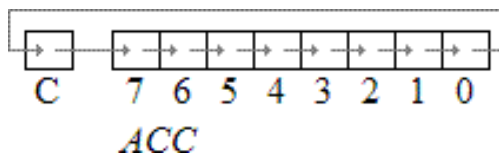
RLC A

Rotate left through the carry. Each bit is shifted one location to the left, with bit 7 going into the carry bit in the PSW, while the carry goes into bit 0.
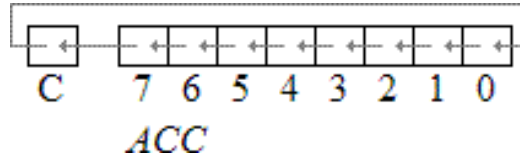
Branch (JUMP) Instructions

Jump and Call Program Range

There are 3 types of jump instructions. They are:-
1. Relative Jump
2. Short Absolute Jump
3. Long Absolute Jump

Relative Jump

Jump that replaces the PC (program counter) content with a new address that is greater than (the address following the jump instruction by 127 or less) or less than (the address following the jump by 128 or less) is called a relative jump. Schematically, the relative jump can be shown as follows: -
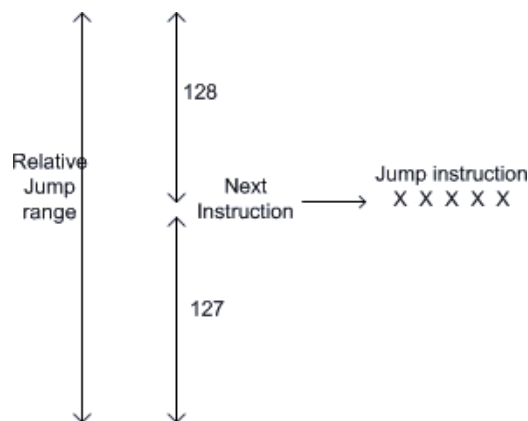
The advantages of the relative jump are as follows:-

1. Only 1 byte of jump address needs to be specified in the 2's complement form, ie. Forjumping ahead, the range is 0 to 127 and for jumping back, the range is -1 to -128.
2. Specifying only one byte reduces the size of the instruction and speeds up programexecution.

3. The program with relative jumps can be relocated without reassembling to generate absolute jump addresses.

    Disadvantages of the absolute jump: -
1. Short jump range (-128 to 127 from the instruction following the jump instruction)

    Instructions that use Relative Jump

        SJMP <relative address>; *this is unconditional jump*

        The remaining relative jumps are conditional jumps

        JC <relative address> JNC <relative address>
        JB bit, <relative address>
        JNB bit, <relative address>
        JBC bit, <relative address>
        CJNE <destination byte>, <source byte>,
        <relative address>
        DJNZ <byte>, <relative address>
        JZ <relative address> JNZ <relative address>

Short Absolute Jump

In this case only 11bits of the absolute jump address are needed. The absolute jump address is calculated in the following manner.

In 8051, 64 kbyte of program memory space is divided into 32 pages of 2 kbyte each. The hexadecimal addresses of the pages are given as follows:-

        Page (Hex)   *Address (Hex)*

        00   *0000 - 07FF*
        01   *0800 - 0FFF*
        02   *1000 - 17FF*
        03   *1800 - 1FFF*
        .
        .
        1E   *F000 - F7FF*
        1F   *F800 - FFFF*

It can be seen that the upper 5bits of the program counter (PC) hold the page number and the lower 11bits of the PC hold the address within that page. Thus, an absolute address is formed by taking page numbers of the instruction (from the program counter) following the jump and attaching the specified 11bits to it to form the 16-bit address.

Advantage: The instruction length becomes 2 bytes.

Example of short absolute jump: -

ACALL <address 11>

AJMP <address 11>

Long Absolute Jump/Call

Applications that need to access the entire program memory from 0000H to FFFFH use long absolute jump. Since the absolute address has to be specified in the op-code, the instruction length is 3 bytes (except for JMP @ A+DPTR). This jump is not re-locatable.

Example: -

LCALL <address 16>
LJMP <address 16>
JMP @A+DPTR

Another classification of jump instructions is

1. Unconditional Jump
2. Conditional Jump

1. **The unconditional jump** is a jump in which control is transferred unconditionally to the target location.
    a. **LJMP** (long jump). This is a 3-byte instruction. First byte is the op-code and second and third bytes represent the 16-bit target address which is any memory location from 0000 to FFFFH
    eg: LJMP 3000H

    b. **AJMP:** this causes unconditional branch to the indicated address, by loading the 11 bit address to 0 -10 bits of the program counter. The destination must be therefore within the same 2K blocks.
    c. **SJMP** (short jump). This is a 2-byte instruction. First byte is the op-code and second byte is the relative target address, 00 to FFH (forward +127 and backward -128 bytes from the current PC value). To calculate the target address of a short jump, the second byte is added to the PC value which is address of the instruction immediately below the jump.

2. **Conditional Jump instructions.**

JBC	Jump if bit $=1$ and clear bit
JNB	Jump if bit $=0$
JB	Jump if bit $=1$

JNC	Jump if CY $=0$
JC	Jump if CY $=1$
CJNE reg,#data	Jump if byte $\neq$ #data
CJNE A,byte	Jump if A $\neq$ byte
DJNZ	Decrement and Jump if A $\neq$ 0
JNZ	Jump if A $\neq$ 0
JZ	Jump if A $=0$

All conditional jumps are short jumps.

Bit level jump instructions:

Bit level JUMP instructions will check the conditions of the bit and if condition is true, it jumps to the address specified in the instruction. All the bit jumps are relative jumps.

JB bit, rel ; jump if the direct bit is set to the relative address specified.
JNB bit, rel ; jump if the direct bit is clear to the relative address specified.
JBC bit, rel ; jump if the direct bit is set to the relative address specified and then clear the bit.

Subroutine CALL And RETURN Instructions

Subroutines are handled by CALL and RET instructions
There are two types of CALL instructions

1. **LCALL address(16 bit)**
   This is long call instruction which unconditionally calls the subroutine located at the indicated 16 bit address. This is a 3 byte instruction. The LCALL instruction works as follows.

   a. During execution of LCALL, [PC] = [PC]+3; (if address where LCALL resides is say, 0x3254; during execution of this instruction [PC] = 3254h + 3h = 3257h
   b. [SP]=[SP]+1; (if SP contains default value 07, then SP increments and [SP]=08
   c. [[SP]] = [PC7-0]; (lower byte of PC content ie., 57 will be stored in memory location 08.
   d. [SP]=[SP]+1; (SP increments again and [SP]=09)
   e. [[SP]] = [PC15-8]; (higher byte of PC content ie., 32 will be stored in memory location 09. With these the address (0x3254) which was in PC is stored in stack.
   f. [PC]= address (16 bit); the new address of subroutine is loaded to PC. No flags are affected.

2. **ACALL address(11 bit)**

This is absolute call instruction which unconditionally calls the subroutine located at the indicated 11 bit address. This is a 2 byte instruction. The SCALL instruction works as follows.

a. During execution of SCALL, [PC] = [PC]+2; (if address where LCALL resides is say, 0x8549; during execution of this instruction [PC] = 8549h + 2h = 854Bh
b. [SP]=[SP]+1; (if SP contains default value 07, then SP increments and [SP]=08
c. [[SP]] = [PC7-0]; (lower byte of PC content ie., 4B will be stored in memory location 08.
d. [SP]=[SP]+1; (SP increments again and [SP]=09)
e. [[SP]] = [PC15-8]; (higher byte of PC content ie., 85 will be stored in memory location 09.

With these the address (0x854B) which was in PC is stored in stack.

f. [PC10-0]= address (11 bit);   the new address of subroutine is loaded to PC. No flags are affected.

RET instruction

RET instruction pops top two contents from the stack and load it to PC.

g. [PC15-8] = [[SP]] ;content of current top of the stack will be moved to higher byte of PC.
h. [SP]=[SP]-1; (SP decrements)
i. [PC7-0] = [[SP]] ;content of bottom of the stack will be moved to lower byte of PC.
j. [SP]=[SP]-1; (SP decrements again)

Bit manipulation instructions.

8051 has 128 bit addressable memory. Bit addressable SFRs and bit addressable PORT pins. It is possible to perform following bit wise operations for these bit addressable locations.

1. LOGICAL AND
   a. ANL C,BIT(BIT ADDRESS)   ; 'Logically and' carry and content of bit address, store result in carry
   b. ANL C, /BIT;   ; 'Logically and' carry and complement of content of bit address, store result in carry

2. LOGICAL OR
   a. ORL C,BIT(BIT ADDRESS)   ; 'Logically or' carry and content of bit address, store result in carry
   b. ORL C, /BIT;   ; 'Logically or' carry and complement of content of bit address, store result in carry
3. CLR bit
   a. CLR bit   ; Content of bit address specified will be cleared.
   b. CLR C   ; Content of carry will be cleared.
4. CPL bit
   a. CPL bit   ; Content of bit address specified will be complemented.
   b. CPL C   ; Content of carry will be complemented

MODULE III
PROGRAMMING AND INTERFACING OF 8051

3.1 SIMPLE PROGRAMMING EXAMPLES IN ASSEMBLY LANGUAGE

ASSEMBLER DIRECTIVES.

Assembler directives tell the assembler to do something other than creating the machine code for an instruction. In assembly language programming, the assembler directives instruct the assembler to
1. Process subsequent assembly language instructions
2. Define program constants
3. Reserve space for variables

The following are the widely used 8051 assembler directives.

ORG (origin)

The ORG directive is used to indicate the starting address. It can be used only when the program counter needs to be changed. The number that comes after ORG can be either in hex or in decimal.

> **Eg: ORG 0000H   ; Set PC to 0000.**

EQU and SET

EQU and SET directives assign numerical value or register name to the specified symbol name.

EQU is used to define a constant without storing information in the memory. The symbol defined with EQU should not be redefined.

SET directive allows redefinition of symbols at a later stage.

DB (DEFINE BYTE)

The DB directive is used to define an 8 bit data. DB directive initializes memory with 8 bit values. The numbers can be in decimal, binary, hex or in ASCII formats. For decimal, the 'D' after the decimal number is optional, but for binary and hexadecimal, 'B' and 'H' are required. For ASCII, the number is written in quotation marks ('LIKE This).

> DATA1: DB 40H   ; hex
> DATA2: DB 01011100B   ; b i n a r y
> DATA3: DB 48   ; decimal
> DATA4: DB 'HELLO W'   ; ASCII

END

The END directive signals the end of the assembly module. It indicates the end of the program to the assembler. Any text in the assembly file that appears after the END directive is ignored. If the END statement is missing, the assembler will generate an error message

3.2 ASSEMBLY LANGUAGE PROGRAMS.

1. **Write a program to add the values of locations 50H and 51H and store the result in locationsin 52h and 53H.**

ORG 0000H	; Set program counter 0000H
MOV A,50H	; Load the contents of Memory location 50H into A ADD ADD A,51H
	; Add the contents of memory 51H with CONTENTS A
MOV 52H,A	; Save the LS byte of the result in 52H
MOV A, #00	; Load 00H into A
ADDC A, #00	; Add the immediate data and carry to A
MOV 53H,A	; Save the MS byte of the result in location 53h
END	

2. **Write a program to store data FFH into RAM memory locations 50H to 58H using directaddressing mode**

ORG 0000H	; Set program counter 0000H
MOV A, #0FFH	; Load FFH into A
MOV 50H, A	; Store contents of A in location 50H
MOV 51H, A	; Store contents of A in location 5IH
MOV 52H, A	; Store contents of A in location 52
HMOV 53H, A	; Store contents of A in location 53H
MOV 54H, A	; Store contents of A in location 54H
MOV 55H, A	; Store contents of A in location 55H
MOV 56H, A	; Store contents of A in location 56H
MOV 57H, A	; Store contents of A in location 57H
MOV 58H, A	; Store contents of A in location 58H
END	

3. **Write a program to subtract a 16 bit number stored at locations 51H-52H from 55H-56H and store the result in locations 40H and 41H. Assume that the least significant byte of data or theresult is stored in low address. If the result is positive, then store 00H, else store 01H in 42H.**

ORG 0000H	; Set program counter 0000H
MOV A, 55H	; Load the contents of memory location 55 into A
CLR C	; Clear the borrow flag
SUBB A,51H	; Sub the contents of memory 51H from contents of A
MOV 40H,A	; Save the LS Byte of the result in location 40H
MOV A, 56H	; Load the contents of memory location 56H into A
SUBB A, 52H	; Subtract the content of memory 52H from the content A
MOV 41H,	; Save the MS byte of the result in location 41H.
MOV A, #00	; Load 005 into A
ADDC A, #00	; Add the immediate data and the carry flag to A
MOV 42H, A	; If result is positive, store00H, else store 0lH in 42H
END	

4. **Write a program to add two 16 bit numbers stored at locations 51H-52H and 55H-56H and store the result in locations 40H, 41H and 42H. Assume that the least significant byte of data and the result is stored in low address and the most significant byte of data or the resultis stored in high address.**

```
ORG 0000H           ; Set program counter 0000H
MOV A,51H           ; Load the contents of memory location 51H into A
ADD A,55H           ; Add the contents of 55H with contents of A
MOV 40H,A           ; Save the LS byte of the result in location 40H
MOV A,52H           ; Load the contents of 52H into A
ADDC A,56H          ; Add the contents of 56H and CY flag with A
MOV 41H,A           ; Save the second byte of the result in 41H
MOV A,#00           ; Load 00H into A
ADDC A,#00          ; Add the immediate data 00H and CY to A
MOV 42H,A           ; Save the MS byte of the result in location 42H
END
```

5. **Write a program to store data FFH into RAM memory locations 50H to 58H using indirect addressing mode.**

```
ORG 0000H           ; Set program counter 0000H
MOV A, #0FFH        ; Load FFH into A
MOV R0, #50H        ; Load pointer, R0-50H
MOV R5, #08H        ; Load counter, R5-08H
Start:MOV @R0, A    ; Copy contents of A to RAM pointed by R0
INC R0              ; Increment pointer
DJNZ R5, start      ; Repeat until R5 is zero
END
```

6. **Write a program to add two Binary Coded Decimal (BCD) numbers stored at locations 60H and 61H and store the result in BCD at memory locations 52H and 53H. Assume that the least significant byte of the result is stored in low address.**

```
ORG 0000H       ; Set program counter 00004
MOV A,60H       ; Load the contents of memory location 60H into A
ADD A,61H       ; Add the contents of memory location 61H with contents of A
DA A            ; Decimal adjustment of the sum in A
MOV 52H, A      ; Save the least significant byte of the result in location 52H
MOV A,#00       ; Load 00H into .A
ADDC A,#00H     ; Add the immediate data and the contents of carry flag to A
MOV 53H,A       ; Save the most significant byte of the result in location 53H
END
```

7. **Write a program to clear 10 RAM locations starting at RAM address 1000H.**

```
ORG 0000H           ;Set program counter 0000H
MOV DPTR, #1000H    ;Copy address 1000H to DPTR
CLR A               ;Clear A
MOV R6, #0AH        ;Load 0AH to R6
again: MOVX @DPTR,A ;Clear RAM location pointed by DPTR

INC DPTR            ;Increment DPTR
DJNZ R6, again      ;Loop until counter R6=0
END
```

8.   **Write a program to compute 1 + 2 + 3 + N (say N=15) and save the sum at70H**

```
ORG 0000H    ; Set program counter 0000H
N EQU 15
MOV R0,#00          ; Clear R0
CLR A              ; Clear A
again: INC R0              ; Increment R0
ADD A, R0          ; Add the contents of R0 with A
CJNE R0,#N,again    ; Loop until counter, R0, N
MOV 70H,A           ; Save the result in location 70H
END
```

9. **Write a program to multiply two 8 bit numbers stored at locations 70H and 71H and store the result at memory locations 52H and 53H. Assume that the least significant byte of the result is stored in low address.**

```
ORG 0000H ; Set program counter 00 OH

MOV A, 70H ; Load the contents of memory location 70h into A

MOV B, 71H ; Load the contents of memory location 71H into B

MUL AB        ; Perform multiplication
MOV 52H,A ; Save the least significant byte of the result in location 52H
MOV 53H,B ; Save the mostsignificant byte of the result in location 53
END
```

10. **Ten 8 bit numbers are stored in internal data memory from location 5oH. Write a program to increment the data.**
Assume that ten 8 bit numbers are stored in internal data memory from location 50H, henceR0 or R1 must be used as a pointer.

The program is as follows.

```
OPT 0000H
MOV R0,#50H
MOV R3,#0AH
Loopl: INC @R0
INC RO
DJNZ R3, loop
END
```

11. **Write a program to find the average of five 8 bit numbers. Store the result in H. (Assume that after adding five 8 bit numbers, the result is 8 bit only).**

```
ORG 0000H
MOV 40H,#05H
MOV 41H,#55H
MOV 42H,#06H
MOV 43H,#1AH
MOV 44H,#09H
MOV R0,#40H
MOV R5,#05H
MOV B,R5CLR A
Loop: ADD A,@RO
INC RO
DJNZ R5,Loop
DIV AB
MOV 55H,A
END
```

12. **Write a program to find the cube of an 8 bit number program is as follows**

```
ORG 0000H
MOV R1,#N
MOV A,R1
MOV B,R1
MUL AB              //SQUARE IS COMPUTEDMOV R2, B
MOV B, R1
MUL AB
MOV 50,A
MOV 51,B
MOV A,R2
MOV B, R1
MUL AB
ADD A, 51H
MOV 51H, A
MOV 52H, B
MOV A,#00H
ADDC A, 52H
MOV 52H, A          //CUBE IS STORED IN 52H,51H,50H
END
```

13. **Write a program to exchange the lower nibble of data present in external memory 6000H and6001H**

```
ORG 0000H              ; Set program counter 00h
MOV DPTR, #6000H ; Copy address 6000H to DPTR
MOVX A, @DPTR     ; Copy contents of 6000H to A
MOV R0, #45H         ; Load pointer, R0=45H
MOV @RO, A           ; Copy cont of A to RAM pointed by R0
INC DPL              ; Increment pointer
MOVX A, @DPTR     ; Copy contents of 6001H to A
XCHD A, @R0         ; Exchange lower nibble of A with RAM pointe
d by RO
MOVX @DPTR, A     ; Copy contents of A to 6001H
DEC DPL              ; Decrement pointer
MOV A, @R0           ; Copy cont of RAM pointed by R0 to A
MOVX @DPTR, A     ; Copy cont of A to RAM pointed by DPTR
END
```

14. **Write a program to count the number of and o's of 8 bit data stored in location 6000H.**

```
ORG 00008                  ; Set program counter 00008
MOV DPTR, #6000h           ; Copy address 6000H to DPTR
MOVX A, @DPTR           ; Copy number to A
MOV R0,#08              ; Copy 08 in RO
MOV R2,#00              ; Copy 00 in R2
MOV R3,#00              ; Copy 00 in R3
CLR C                      ; Clear carry flag
BACK: RLC A                ; Rotate A through carry flag
JC NEXT                 ; If CF = 1, branch to next
INC R2                  ; If CF = 0, increment R2
AJMP NEXT2
NEXT:    INC R3          ; If CF = 1, increment R3
NEXT2:  DJNZ RO,BACK     ; Repeat until RO is zero
END
```

15. **Write a program to shift a 24 bit number stored at 57H-55H to the left logically four places.Assume that the least significant byte of data is stored in lower address.**

```
            ORG 0000H        ; Set program counter 0000h
            MOV R1,#04       ; Set up loop count to 4
  again:    MOV A,55H        ; Place the least significant byte of data in A
            CLR C            ; Clear tne carry flag
            RLC A            ; Rotate contents of A (55h) left through carry
            MOV 55H,A
            MOV A,56H
            RLC A            ; Rotate contents of A (56H) left through carry
            MOV 56H,A
            MOV A,57H
            RLC A            ; Rotate contents of A      (57H) left through carry
            MOV 57H,A
            DJNZ R1,again    ; Repeat until R1 is zero
            END
```

3.3 INTERFACING WITH 8051 USING ASSEMBLY LANGUAGE PROGRAMMING:

LED INTERFACING TO 8051

Blinking 1 LED using 8051

This is the first project regarding 8051 and of course one of the simplest, blinking LED using 8051. The microcontroller used here is AT89S51 In the circuit, push button switch S1, capacitor C3 and resistor R3 forms the reset circuitry. When S1 is pressed, voltage at the reset pin (pin9) goes high and this resets the chip. C1, C2 and X1 are related to the on chip oscillator which produces the required clock frequency. P1.0 (pin1) is selected as the output pin. When P1.o goes high the transistor Q1 is forward biased and LED goes ON. When P1.0 goes low the transistor goes to cut off and the LED extinguishes. The transistor driver circuit for the LED can be avoided and the LED can be connected directly to the P1.0 pin with a series current limiting resistor(~1K). The time for which P1.o goes high and low (time period of the LED) is determined by the program. The circuit diagram for blinking 1 LED is shown below.
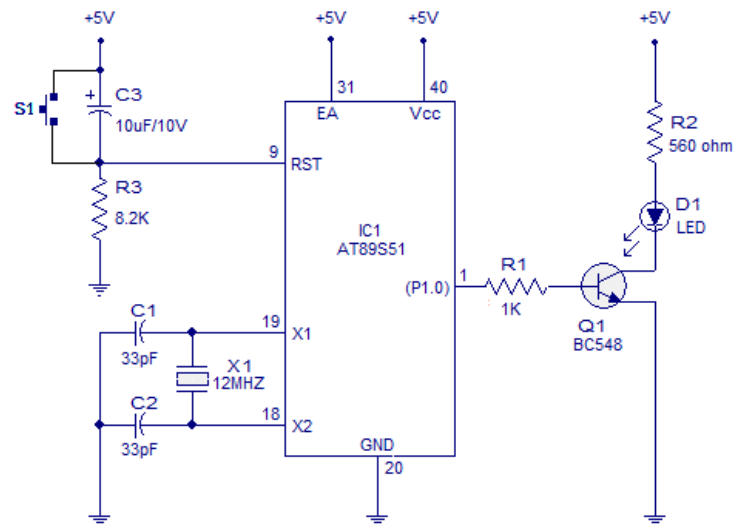
Blinking LED using 8051
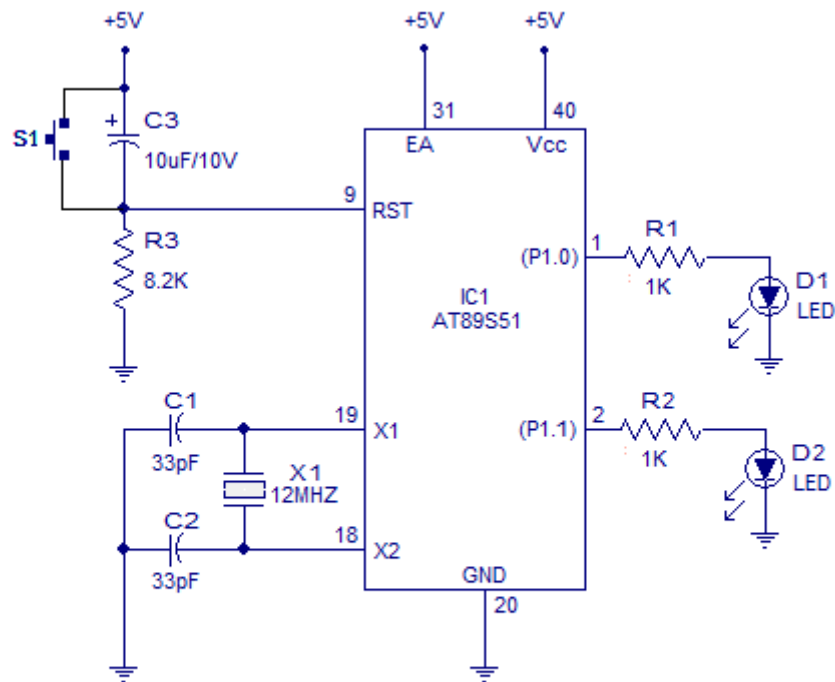
Program

```
START: CPL P1.0
        ACALL WAIT
        SJMP START
WAIT:  MOV R4,#05H
WAIT1: MOV R3,#00H
WAIT2: MOV R2,#00H
WAIT3: DJNZ R2,WAIT3
        DJNZ R3,WAIT2
        DJNZ R4,WAIT1
        RET
```

Blinking 2 LED alternatively using 8051.

This circuit can blink two LEDs alternatively. P1.0 and P1.1 are assigned as the outputs. When P1.0 goes high P1.0 goes low and vice versa and the LEDs follow the state of the corresponding port to which it is connected. Here there is no driver stage for the LEDs and they are connected directly to the corresponding ports through series current limiting resistors (R1 & R2). Circuit diagram is shown below.

Blinking 2 LED alternatively using 8051

Program

```
START: CPL P1.0
        ACALL WAIT
        CPL P1.0
        CPL P1.1
        ACALL WAIT
        CPL P1.1
        SJMP START
WAIT:  MOV R4,#05H
WAIT1: MOV R3,#00H
WAIT2: MOV R2,#00H
WAIT3: DJNZ R2,WAIT3
        DJNZ R3,WAIT2
        DJNZ R4,WAIT1
        RET
```

INTERFACING 7 SEGMENT DISPLAY TO 8051.

A NOTE ABOUT 7 SEGMENT LED DISPLAY.

This article is about how to interface a seven segment LED display to an 8051 microcontroller. 7 segment LED display is very popular and it can display digits from 0 to 9 and quite a few characters like A, b, C, ., H, E, e, F, n, o,t,u,y, etc. Knowledge about how to interface a seven segment display to a micro controller is very essential in designing embedded systems. A seven segment display consists of seven LEDs arranged in the form of a squarish **'8'** slightly inclined to the right and a single LED as the dot character. Different characters can be displayed by selectively glowing the required LED segments. Seven segment displays are of two types, *common cathode and common anode.* In common cathode type , the cathode of all LEDs are tied together to a single terminal which is usually labeled as 'com' and the anode of all LEDs are left alone as individual pins labeled as a, b, c, d, e, f, g & h (or dot) . In common anode type, the anode of all LEDs are tied together as a single terminal and cathodes are left alone as individual pins. The pin out scheme and picture of a typical 7 segment LED display is shown in the image below.
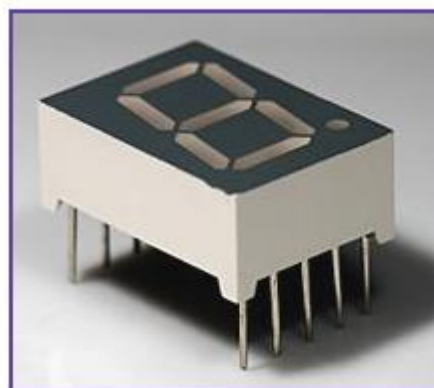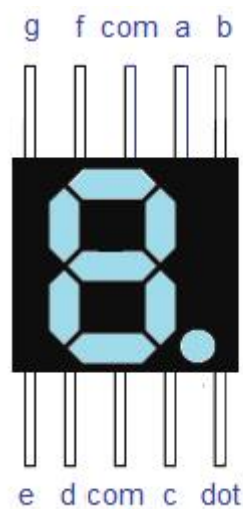
Digit drive pattern.

Digit drive pattern of a seven segment LED display is simply the different logic combinations of its terminals **'a' to 'h'** in order to display different digits and characters. The common digit drive patterns (0 to 9) of a seven segment display are shown in the table below.

Digit	a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1

The circuit diagram shown above is of an AT89S51 microcontroller based 0 to 9 counter which has a 7 segment LED display interfaced to it in order to display the count. This simple circuit illustrates two things. How to setup simple 0 to 9 up counter using 8051 and more importantly how to interface a seven segment LED display to 8051 in order to display a particular result. The common cathode seven segment display D1 is connected to the Port 1 of the microcontroller (AT89S51) as shown in the circuit diagram. R3 to R10 are current

limiting resistors. S3 is the reset switch and R2,C3 forms a debouncing circuitry. C1, C2 and X1 are related to the clock circuit. The software part of the project has to do the following tasks.

- Form a 0 to 9 counter with a predetermined delay (around 1/2 second here).

- Convert the current count into digit drive pattern.

- Put the current digit drive pattern into a port for displaying.

All the above said tasks are accomplished by the program given below.

Interfacing 7 segment display to 8051
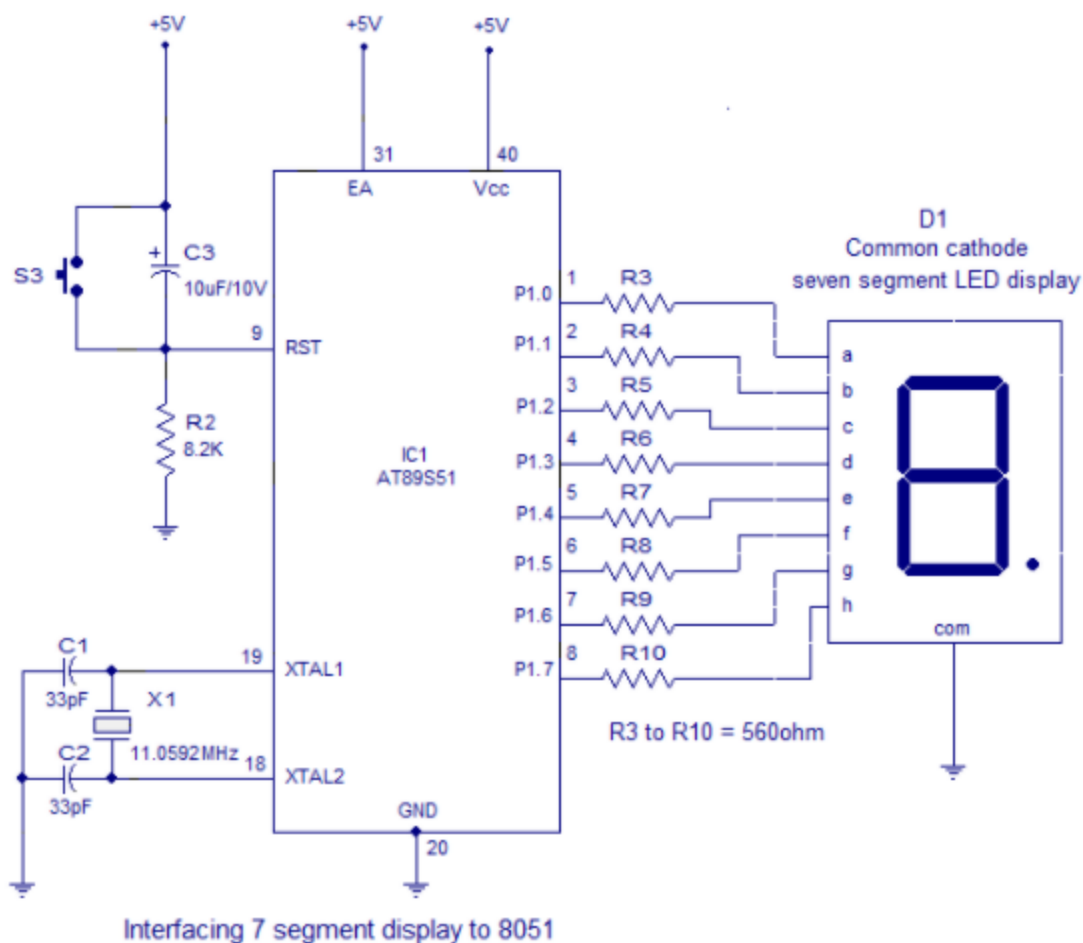
PROGRAM.

```
ORG 000H                        //initial starting address
START: MOV A,#00001001B         // initial value of accumulator
MOV B,A
MOV R0,#0AH                      //Register R0 initialized as counter which counts from 10 to
```

```
0
LABEL: MOV A,B
INC A
MOV B,A
MOVC A,@A+PC                 // adds the byte in A to the program counters address
MOV P1,A
ACALL DELAY                  // calls the delay of the timer
DEC R0                       //Counter R0 decremented by 1
MOV A,R0                     // R0 moved to accumulator to check if it is zero in next
instruction.
JZ START                     //Checks accumulator for zero and jumps to START.
                              Done to check if counting has been finished.
SJMP LABEL
DB 3FH                       // digit drive pattern for 0
DB 06H                       // digit drive pattern for 1
DB 5BH                       // digit drive pattern for 2
DB 4FH                       // digit drive pattern for 3
DB 66H                       // digit drive pattern for 4
DB 6DH                       // digit drive pattern for 5
DB 7DH                       // digit drive pattern for 6
DB 07H                       // digit drive pattern for 7
DB 7FH                       // digit drive pattern for 8
DB 6FH                       // digit drive pattern for 9
DELAY: MOV R4,#05H           // subroutine for delay
WAIT1: MOV R3,#00H
WAIT2: MOV R2,#00H
WAIT3: DJNZ R2,WAIT3
DJNZ R3,WAIT2
DJNZ R4,WAIT1
RET
END
```

ABOUT THE PROGRAM.

Instruction MOVC A,@A+PC is the instruction that produces the required digit drive pattern for the display. Execution of this instruction will add the value in the accumulator A with the content of the program counter(address of the next instruction) and will move the data present in the resultant address to A. After this the program resumes from the line after MOVC A,@A+PC.

In the program, initial value in A is 00001001B. Execution of MOVC A,@A+PC will add oooo1001B to the content in PC (address of next instruction). The result   will be the address of label DB 3FH (line15) and the data present in this address ie 3FH (digit drive pattern for 0) gets moved into the accumulator. Moving this pattern in the accumulator to Port 1 will display 0 which is the first count.

At the next count, value in A will advance to 00001010 and after the execution of MOVC A,@+PC ,the value in A will be 06H which is the digit drive pattern for 1 and this will display 1 which is the next count and this cycle gets repeated for subsequent counts.

The reason why accumulator is loaded with 00001001B (9 in decimal) initially is that the instructions from line 9 to line 15 consumes 9 bytes in total.

3.4 PROGRAMMING IN C

Embedded C

For programming the embedded hardware devices, we need to use Embedded C language instead of our conventional C language.

The key differences between conventional C and Embedded C are

❖ Embedded C has certain predefined variables for registers, ports etc. which are in 8051 e.g. ACC, P1, P2, TMOD etc.
❖ We can run super loop (infinite loop) in embedded C language.

We know that the programming in C language is solely done by dealing with different variables.

In case of Embedded C, these variables are nothing else but the memory locations of different memories of the microcontroller like code memory(ROM), data memory (RAM), external memory etc. To use these memory locations as variables, we need to use data types.

Data types

There are 7 different data types in embedded C for 8051…

1) unsigned char
   This data type is used to define an unsigned 8-bit variable. All 8-bits of this variable are used to specify data. Hence the range of this data type is $(0)_{10} \, to \, (255)_{10}$.
        e.g. unsigned char count;
2) signed char
   This data type is used to define a signed 8-bit variable. Here MSB of variable is used to show sign (+/-) while rest 7 bits are used to specify the magnitude of the variable. Hence the range of this data type is $(-128)_{10} \, to \, (127)_{10}$.
        e.g. signed char temp;
3) unsigned int
   This data type is used to define a 16-bit variable. Hence from this we can comment that this data types combines any 2 memory locations of the data memory as one variable. Here all 16 bits are used to specify data. So the range of this data type is $(0)_{10} \, to \, (65535)_{10}$.

4) signed int
   This data type is used to define a signed variable like *signed char* but of 16-bit size. Hence its range is $(-32768)_{10} \, to \, (32767)_{10}$

5) sfr
   This is an 8-bit data type used for defining names of Special Function Registers (SFR's) that are located in RAM memory locations 80 H to FF H only.
   e.g. sfr P0 = 0x80;

6) bit
   This data type is used to access single bits from the bit-addressable area of RAM.
   e.g. bit MYBIT = 0x32;

7) sbit
   The sbit data type is the one which is used to define or rather access single bits of the bit addressable SFR's of 8051 microcontroller.
   e.g. sbit En = P2^0;

Decision control structures

The decision control structures are used to decide whether to execute a particular block of code depending on the condition specified.
™ Following are some decision control structures:
   ❖ *if* statement
   ❖ *if…else* statement

if statement

if(condition)

{

statement-1; statement-2;

……..

}

if...else statement if(condition)

{

statement-1; statement-2;

………

}

else

{

statement n;

}

Loop statements

The loop statements are the one which are used when we want to execute a certain block of code for more than one times either depending on situation or by a predefined number of times.

™ Embedded C is basically having two loop statements:

❖ *for* loop

❖ *while* loop

1) <u>for loop</u>

for loops are used to repeat any particular piece of code a predefined number of times.

™for(initializations ; conditions ; updates)

        {
            statement-1;
            statement-2;
            ………
        }

2) <u>while loop</u>

while loop also has the provision to repeat a certain block of code but here the block is repeated depending on the condition specified. The loop keeps on repeating until the condition becomes false.

™ Format of while loop is:

         while(condition)

        {

        statement-1;

        statement- 2;

        ………

        }

Break & Continue Statements

1) break

The break statement, whenever is encountered in the loop, it forces the control to terminate the loop in which it is written.

2) ™ **continue**

Whenever this statement is encountered in any loop, the statements in the loop after it won't be executed i.e. will be skipped and again control will be transferred to check the condition of the loop.

Format of any C Program

#include <reg51.h>   **Header File**

sbit *<name>=<bit address>;*   **sfr bit definitions**

sfr *<name>=<sfr address>;*   **sfr definition**

Data-type udf1(*data-type var_name);*   **User defined function**

Data-type udf2(*data-type var_name);*

void main(void)   ***main function***

{

statement-1;

statement-2;

…………………;

Functions

Sometimes, there comes a situation in which in a program a group of statements is used frequently. Writing these statements again & again makes our program clumsy to write as well as it consumes more memory space. To overcome this problem there is a facility in C language to define a function. In function we can write the particular group of statements which is getting repeated continuously. Now anytime when we want to use that code group, we just have to call the function and it's done.

Types of functions

- No arguments, no return values

- With no arguments and a return value

- With arguments but no return value

- With arguments and return value

There are 3 ways to deal with a function:

- ❖ Define first, then use
- ❖ Do prototyping (i.e. Define first, use after main( ))
- ❖ Do prototyping in header file
    - a) **Define first, then use**

In this case, before writing the main function, we define the user-defined function and then use it in main( ) function whenever required.

b) **Do prototyping and define after main function**

In this case the function name, data type and argument data type are specified before writing main function to declare that we'll later implement this function.

c) **Do prototyping in header file**

In this case, define the function in a (user defined) header file and then just include that header file in your program.

Data Types in Embedded C

Data Type	Size in Bits	Data Range/Usage
unsigned char	8-bit	0 to 255
(signed) char	8-bit	-128 to +127
unsigned int	16-bit	0 to 65535
(signed) int	16-bit	-32768 to +32767
sbit	1-bit	SFR bit-addressable only
bit	1-bit	RAM bit-addressable only
sfr	8-bit	RAM addresses 80 – FFH only

Delay generation in 8051

The delay length in 8051 microcontroller depends on three factors:

- ❖ The crystal frequency
- ❖ the number of clock per machine
- ❖ the C compiler.

The original 8051 used 1/12 of the crystal oscillator frequency as one machine cycle. In other words, each machine cycle is equal to 12 clocks period of the crystal frequency connected to X1-X2 pins of 8051. To speed up the 8051, many recent versions of the 8051 have reduced the number of clocks per machine cycle from 12 to four, or even one. The frequency for the timer is always 1/12th the frequency of the crystal attached to the 8051, regardless of the 8051 version. In other words, AT89C51, DS5000, and DS89C4x0 the duration of the time to execute an instruction varies, but they all use 1/12th of the crystal's oscillator frequency for the clock source.

8051 has two different ways to generate time delay using C programming, regardless of 8051 version.

The **first method** is simply using *Loop* program function in which *Delay( )* function is made or by providing *for();* delay loop in Embedded C programming. You can define your own value of delay and how long you want to display. For example- *for(i=0;i<"any decimal value";i++);* this is the delay for loop used in embedded C.

Code to generate 250 ms delay on Port P1 of 8051:

```
#include "REG52.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++)     // this is For( ); loop delay used to define delay value in
                               Embedded C
{
   for (j=0;j<1275;j++);
}
}
```

The **second method** is using Timer registers TH, TL and TMOD that are accessible in embedded C by defining header file *reg52.h* Both timers 0 and 1 use the same register, called TMOD (timer mode), to set the various timer operation modes in 8051 C programming. There are four operating modes of timer 0 and 1.

To generate Time delay using timer registers:

- Load the TMOD value register indicating which timer (timer 0 or timer 1) is to be used and which timer mode (0 or is selected
- Load registers TL and TH with initial count value
- Start the timer
- Keep monitoring the timer flag (TF) until it rolls over from FFFFH to 0000.
- After the timer reaches its limit and rolls over, in order to repeat the process - TH and TL must be reloaded with the original value, and TR is turned off by setting value to 0 and TF must be reloaded to 0.
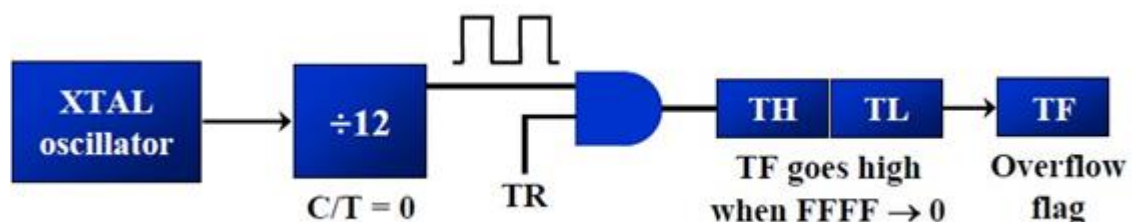
Code generating delay using timer register:

```
#include <REG52.h>
void T0Delay(void);
void main(void){

while (1)
{
P1=0x55;
T0Delay( );
P1=0xAA;
T0Delay ( );
}
}

void T0Delay( )
{
TMOD=0x01;    // timer 0, mode 1
TL0=0x00;     // load TL0
TH0=0x35;     // load TH0
TR0=1;        // turn on Timer0
while (TF0==0);        // wait for TF0 to roll over
TR0=0;     // turn off timer
TF0=0;     // clear TF0
}
```

Port programming

1. Write an 8051 C program to send values 00 – FF to port P1.

```
#include <reg51.h>
 void main(void)
{
unsigned char i;  for (i=0;i<=255;i++)
P1=i;
}
```

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

```
#include <reg51.h>
 void main(void)
{
unsigned char mynum( )="012345ABCD";
unsigned char i;
for (i=0;i<=10;i++)
 P1=mynum(i);
}
```

3. Write an 8051 C program to toggle all the bits of P1 continuously.

```
#include <reg51.h>  void main(void)
{
While (1)
{
p1=0x55;
p1=0xAA;
}
}
```

4. Write an 8051 C program to send values of −4 to +4 to port P1.

```
//Singed numbers
 #include <reg51.h>
void main(void)
{
char mynum[ ]={+1,-1,+2,-2,+3,-3,+4,-4};
unsigned char i;  for (i=0;i<=8;i++)
P1=mynum[i];
}
```

5. Write an 8051 C program to send values of −4 to +4 to port P1

```
//Singed numbers
 #include <reg51.h>
 void main(void)
{
signed char i;
for (i=-4;i<=4;i++)
P1=mynum[i];
}
```

6. Write an 8051 C program to toggle bit D0 of the port P1 (P1.0) 50,000 times.

```
#include <reg51.h>
sbit MYBIT=P1^0;
void main(void)
{
unsigned int z;
for (z=0;z<=50000;z++)
{
MYBIT=0;
MYBIT=1;
}
}
```

Note: sbit keyword allows access to the single bits of the SFR registers

7. LEDs are connected to bits P1 and P2. Write an 8051 C program that shows the count from 0 to FFH (0000 0000 to 1111 1111 in binary) on the LEDs.

```
#include <reg51.h>
#define LED P2;
void main(void)
{
P1=00; //clear P1
 LED=0; //clear P2
while(1)
{
P1++; //increment P1
 LED++; //increment P2
}
}
```

Note: Ports P0 – P3 are byte-accessable and we can use the P0 – P3 labels as defined in the 8051 header file <reg51.h>

8. Write an 8051 C program to get a byte of data form P1, wait 1/2 second, and then send it to P2.

```
#include <reg51.h>
void MSDelay(unsigned int);
void main(void)
{
unsigned char mybyte;
P1=0xFF; //make P1 input port
while (1)
{
mybyte=P1; //get a byte from P1
MSDelay(500);
P2=mybyte; //send it to P2
}
}
```

```
void MSDelay(unsigned int itime)
    {
    unsigned int i,j;
    for (i=0;i<itime;i++)
    for (j=0;j<1275;j++);
    }
```

9. Write an 8051 C program to get a byte of data form P0. If it is less than 100, send it to P1; otherwise, send it to P2.

```
#include <reg51.h>
void main(void)
{
unsigned char mybyte;
P0=0xFF; //make P0 input port
while (1)
{
mybyte=P0; //get a byte from P0
if (mybyte<100)
P1=mybyte; //send it to P1
```

```
void MSDelay(unsigned int itime)
    {
    unsigned int i,j;
    for (i=0;i<itime;i++)  for
    (j=0;j<1275;j++);
    }
```

```
else
P2=mybyte; //send it to P2
}
}
```

10. Write an 8051 C program to toggle only bit P2.4 continuously without disturbing the rest of the bits of P2

```
//Toggling an individual bit
#include <reg51.h>
sbit mybit=P2^4;
void main(void)
{
while (1)
{
mybit=1; //turn on P2.4
mybit=0; //turn off P2.4
}
}
```

Note:

- ❖ Ports P0 – P3 are bit-addressable and we use sbit data type to access a single bit of P0 - P3
- ❖ Use the Px^y format, where x is the port 0, 1, 2, or 3 and y is the bit 0 – 7 of that port

11. Write an 8051 C program to monitor bit P1.5. If it is high, send 55H to P0; otherwise, send AAH to P2

```
#include <reg51.h>
sbit mybit=P1^5;
 void main(void)
{
mybit=1; //make mybit an input
while (1)
{
        if (mybit==1)
        P0=0x55;
else
        P2=0xAA;
}
}
```

12. A door sensor is connected to the P1.1 pin, and a buzzer is connected to P1.7. Write an 8051 C program to monitor the door sensor, and when it opens, sound the buzzer. You can sound the buzzer by sending a square wave of a few hundred Hz.

```
#include <reg51.h>
void MSDelay(unsigned int);
sbit Dsensor=P1^1;
sbit Buzzer=P1^7;
void main(void)
{
Dsensor=1; //make P1.1 an input
while (1)
{
while (Dsensor==1)//while it opens
{
Buzzer=0;  MSDelay(200);
 Buzzer=1;  MSDelay(200);
}
}
}
```

```
void MSDelay(unsigned int itime)

{

unsigned int i,j;

for (i=0;i<itime;i++)  for
(j=0;j<1275;j++);

}
```

13. Write an 8051 C program to toggle all the bits of P0, P1, and P2 continuously with a 250 ms delay. Use the sfr keyword to declare the port addresses

```
sfr P0=0x80;
sfr P1=0x90;
sfr P2=0xA0;
void MSDelay(unsigned int);
void main(void)
{
        while (1)
        {
                P0=0x55;
                P1=0x55;
                P2=0x55;
                MSDelay(250);
                P0=0xAA;
                P1=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++);
        }
```

14. The data pins of an LCD are connected to P1. The information is latched into the LCD whenever its Enable pin goes from high to low. Write an 8051 C program to send "ECED-SVNIT" to this LCD

```
#include <reg51.h>
#define LCDData P1 //LCDData declaration
sbit En=P2^0; //the enable pin  void main(void)
{
        unsigned char message[ ] ="ECED-SVNIT";
        unsigned char z;
        for (z=0;z<10;z++)    //send 10 characters
        {
        LCDData=message[z];
        En=1; //a high-
        En=0; //-to-low pulse to latch data
        }
}
```

15. Write an 8051 C program to turn bit P1.5 on and off 50,000 times.

```
#include <reg51.h>
sbit MYBIT=0x95;
void main(void)
{
        unsigned int z;
        for (z=0;z<50000;z++)
        { MYBIT=1;  MYBIT=0;
        }
}
        Note
```
❖ We can access a single bit of any SFR if we specify the bit address

Code Conversion Programs

1. **Write an 8051 C program to convert packed BCD 0x29 to ASCII and display the bytes on P1 and P2.**

```
#include <reg51.h>
void main(void)
{
unsigned char x,y,z;
unsigned char mybyte=0x29;
x=mybyte&0x0F;
P1=x|0x30;
```

```
y=mybyte&0xF0;
y=y>>4;
P2=y|0x30;
}
```

2. Write an 8051 C program to convert ASCII digits of '4' and '7' to packed BCD and display them on P1.

```
#include <reg51.h>
void main(void)
{
unsigned char bcdbyte;
 unsigned char w='4';
 unsigned char z='7';
w=w&0x0F;
w=w<<4;
z=z&0x0F;
bcdbyte=w|z;
P1=bcdbyte;
}
```

3. Write an 8051 C program to calculate the checksum byte for the data 25H, 62H, 3FH, and 52H.

```
#include <reg51.h>
void main(void)
{
unsigned char mydata[ ]={0x25,0x62,0x3F,0x52};
 unsigned char sum=0;
unsigned char x;
unsigned char chksumbyte;  for (x=0;x<4;x++)
{
P2=mydata[x];
sum=sum+mydata[x];
}
chksumbyte=~sum+1;
P2=chksumbyte;
}
```

4. Write an 8051 C program to perform the checksum operation to ensure data integrity. If data is good, send ASCII character 'G' to P0. Otherwise send 'B' to P0.

```
#include <reg51.h>
void main(void)
{
unsigned char mydata[ ]={0x25,0x62,0x3F,0x52,0xE8};
```

```
unsigned char chksum=0;
unsigned char x;
for (x=0;x<5;x++)  chksum=chksum + mydata[x];
if (chksum==0)
P0='G';
else
P0='B';
}
```

5. Write an 8051 C program to convert 11111101 (FD hex) to decimal and display the digits on P0, P1 and P2.

```
#include <reg51.h>
void main(void)
{
unsigned char x,binbyte,d1,d2,d3;
binbyte=0xFD;
 x=binbyte/10;
d1=binbyte%10;
d2=x%10;
d3=x/10;
P0=d1;
P1=d2;
P2=d3;
}
```

INTERFACING THE KEYBOARD TO 8051 MICROCONTROLLER

The key board here we are interfacing is a matrix keyboard. This key board is designed with a particular rows and columns. These rows and columns are connected to the microcontroller through its ports of the micro controller 8051. We normally use 8*8 matrix key board. So only two ports of 8051 can be easily connected to the rows and columns of the key board.

When ever a key is pressed, a row and a column gets shorted through that pressed key and all the other keys are left open. When a key is pressed only a bit in the port goes high. Which indicates microcontroller that the key is pressed. By this high on the bit key in the corresponding column is identified.

Once we are sure that one of key in the key board is pressed next our aim is to identify that key. To do this we firstly check for particular row and then we check the corresponding column the key board.

To check the row of the pressed key in the keyboard, one of the row is made high by making one of bit in the output port of 8051 high . This is done until the row is found out. Once we get the row next out job is to find out the column of the pressed key. The column is detected by contents in the input ports with the help of a counter. The content of the input port is rotated with carry until the carry bit is set.

The contents of the counter is then compared and displayed in the display. This display is designed using a seven segment display and a BCD to seven segment decoder IC 7447.

The BCD equivalent number of counter is sent through output part of 8051 displays the number of pressed key.

Circuit diagram of INTERFACING KEY BOARD TO 8051.

The programming algorithm, program and the circuit diagram is as follows. Here program is explained with comments.
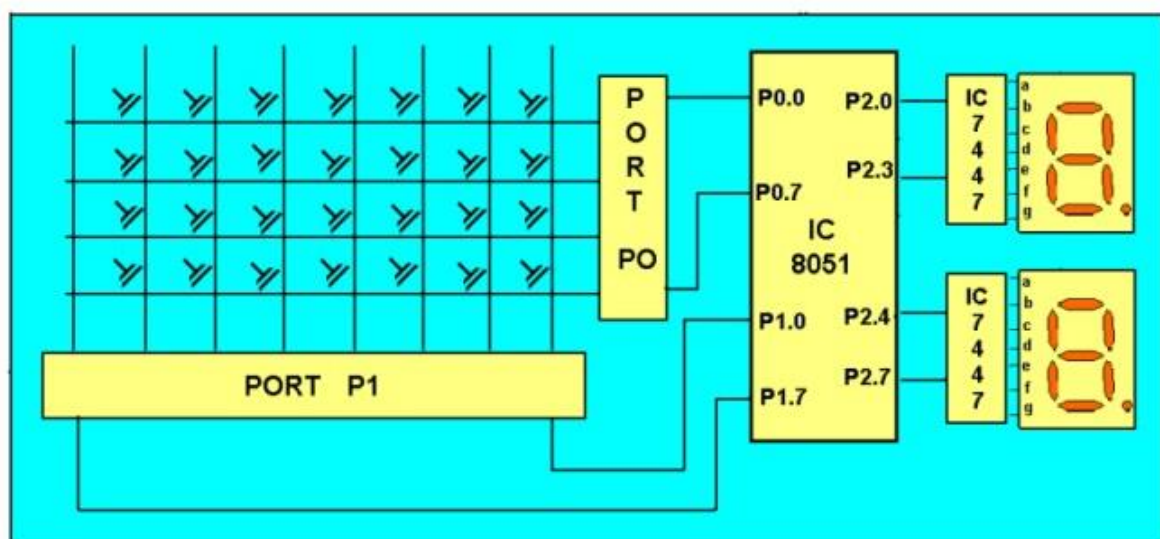
- ❖ The 8051 has 4 I/O ports P0 to P3 each with 8 I/O pins, P0.0 to P0.7,P1.0 to P1.7, P2.0 to P2.7, P3.0 to P3.7. The one of the port P1 (it understood that P1 means P1.0 to P1.7) as an I/P port for microcontroller 8051, port P0 as an O/P port of microcontroller 8051 and port P2 is used for displaying the number of pressed key.
- ❖ Make all rows of port P0 high so that it gives high signal when key is pressed.
- ❖ See if any key is pressed by scanning the port P1 by checking all columns for non zero condition.
- ❖ If any key is pressed, to identify which key is pressed make one row high at a time.
- ❖ Initiate a counter to hold the count so that each key is counted.
- ❖ Check port P1 for nonzero condition. If any nonzero number is there in [accumulator], start column scanning by following step 9.
- ❖ Otherwise make next row high in port P1.
- ❖ Add a count of 08h to the counter to move to the next row by repeating steps from step 6.
- ❖ If any key pressed is found, the [accumulator] content is rotated right through the carry until carry bit sets, while doing this increment the count in the counter till carry is found.
- ❖ Move the content in the counter to display in data field or to memory location
- ❖ To repeat the procedures go to step 2.

Start of main program:

to check that whether any key is pressed

```
start:  mov a,#00h
        mov p1,a        ;making all rows of port p1 zero
        mov a,#0fh
        mov p1,a        ;making all rows of port p1 high
press:  mov a,p2
        jz press        ;check until any key is pressed
```

after making sure that any key is pressed

```
        mov a,#01h      ;make one row high at a time
        mov r4,a
        mov r3,#00h     ;initiating counter
next:   mov a,r4
        mov p1,a        ;making one row high at a time
        mov a,p2        ;taking input from port A
        jnz colscan     ;after getting the row jump to check
                         column
        mov a,r4
        rl a            ;rotate left to check next row
        mov r4,a
        mov a,r3
        add a,#08h      ;increment counter by 08 count
        mov r3,a
        sjmp next       ;jump to check next row
```

after identifying the row to check the column following steps are followed

```
colscan:   mov r5,#00h
     in:   rrc a          ;rotate right with carry until get the carry
           jc out         ;jump on getting carry
           inc r3         ;increment one count
           jmp in
    out:   mov a,r3
           da a           ;decimal adjust the contents of counter
                            before display
           mov p2,a
           jmp start      ;repeat for check next key.
```

INTERFACING DAC TO 8051

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}

$$I_{out} = I_{ref}\left(\frac{D7}{2} + \frac{D6}{4} + \frac{D5}{8} + \frac{D4}{16} + \frac{D3}{32} + \frac{D2}{64} + \frac{D1}{128} + \frac{D0}{256}\right)$$

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 ×sinθ)

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
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

Circuit Diagram –
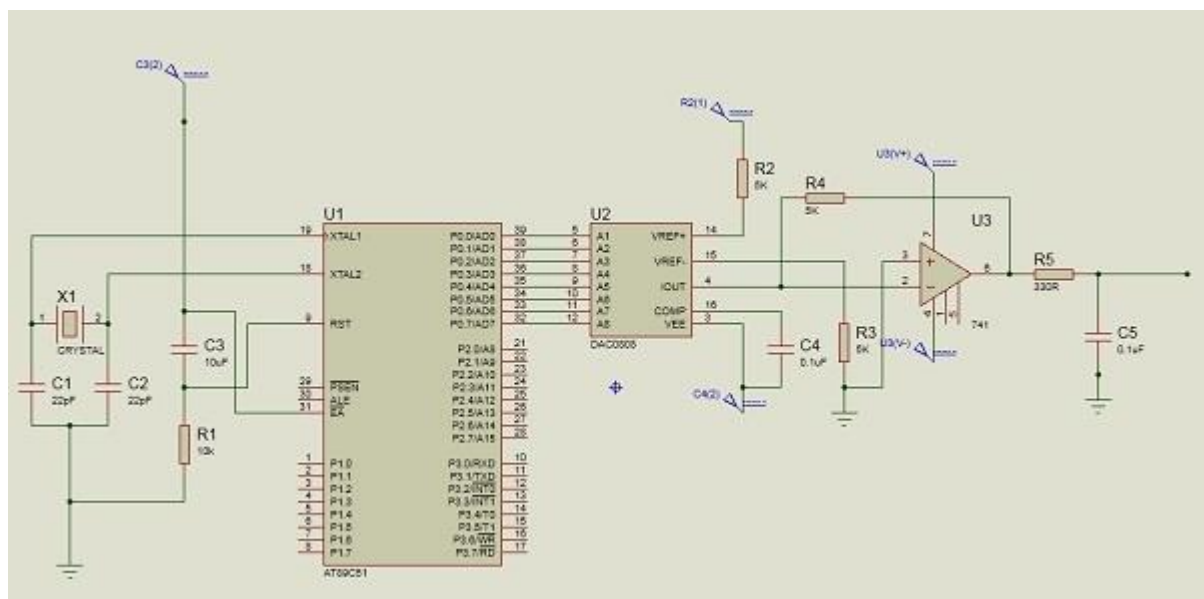
Program

```
#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];
    }
  }
}
```
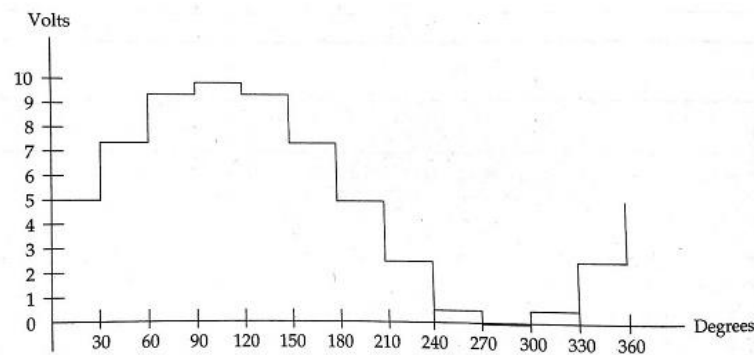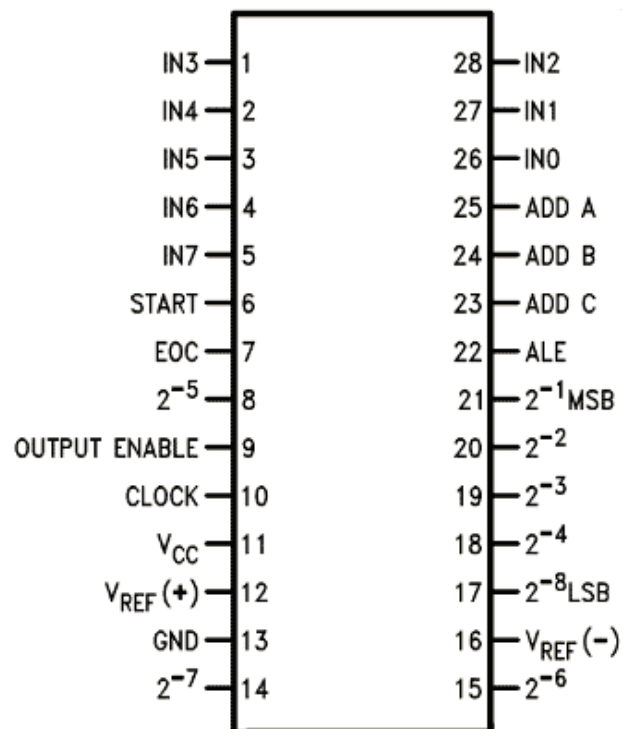

INTERFACING ADC TO 8051

ADC is the Analog to Digital converter, which converts analog data into digital format; usually it is used to convert **analog voltage** into digital format. Analog signal has infinite no of values like a sine wave or our speech, ADC converts them into particular levels or states, which can be measured in numbers as a physical quantity. Instead of continuous conversion, ADC converts data periodically, which is usually known as sampling rate. **Telephone modem** is one of the examples of ADC, which is used for internet, it converts analog data into digital data, so that computer can understand, because computer can only understand Digital data. The major advantage, of using ADC is that, we noise can be efficiently eliminated from the original signal and digital signal can travel more efficiently than analog one. That's the reason that digital audio is very clear, while listening.

In present time there are lots of microcontrollers in market which has inbuilt ADC with one or more channels. And by using their ADC register we can interface. When we select **8051 microcontroller** family for making any project, in which we need of an ADC conversion, then we use **external ADC**. Some external ADC chips are 0803,0804,0808,0809 and there are many more. Today we are going to interface 8-channel ADC with AT89s52 Microcontroller namely ADC0808/0809.

ADC0808/0809 is a monolithic CMOS device and microprocessor compatible control logic and has 28 pin which gives 8-bit value in output and 8- channel ADC input pins (IN0-IN7). Its

resolution is 8 so it can encode the analog data into one of the **256 levels** (2^8). This device has three channel address line namely: ADDA, ADDB and ADDC for selecting channel. Below is the Pin Diagram for ADC0808:

```
        IN3 ──┤1        28├── IN2
        IN4 ──┤2        27├── IN1
        IN5 ──┤3        26├── IN0
        IN6 ──┤4        25├── ADD A
        IN7 ──┤5        24├── ADD B
      START ──┤6        23├── ADD C
        EOC ──┤7        22├── ALE
       2⁻⁵ ──┤8        21├── 2⁻¹MSB
OUTPUT ENABLE ──┤9      20├── 2⁻²
      CLOCK ──┤10       19├── 2⁻³
        Vcc ──┤11       18├── 2⁻⁴
    VREF(+) ──┤12       17├── 2⁻⁸LSB
        GND ──┤13       16├── VREF(−)
       2⁻⁷ ──┤14       15├── 2⁻⁶
```

ADC0808/0809 **requires a clock pulse** for conversion. We can provide it by using oscillator or by using microcontroller. In this project we have applied frequency by using microcontroller.

We can select the any input channel by using the Address lines, like we can select the input line IN0 by keeping all three address lines (ADDA, ADDB and ADDC) Low. If we want to select input channel IN2 then we need to keep ADDA, ADDB low and ADDC high. For selecting all the other input channels, have a look on the given table:

ADC Channel Name	ADDC PIN	ADDB PIN	ADDA PIN
IN0	LOW	LOW	LOW
IN1	LOW	LOW	HIGH
IN2	LOW	HIGH	LOW
IN3	LOW	HIGH	HIGH
IN4	HIGH	LOW	LOW

| IN5	HIGH	LOW	HIGH
IN6	HIGH	HIGH	LOW
IN7	HIGH	HIGH	HIGH

Circuit Description:

Circuit of **"Interfacing ADC0808 with 8051"** is little complex which contains more connecting wire for connecting device to each other. In this circuit we have mainly used AT89s52 as 8051 microcontroller, ADC0808, Potentiometer and LCD.

A 16x2 LCD is connected with 89s52 microcontroller in 4-bit mode. Control pin RS, RW and En are directly connected to pin P2.0, GND and P2.2. And data pin D4-D7 is connected to pins P2.4, P2.5, P2.6 and P2.7 of 89s52. ADC0808 output pin are directly connected to port P1 of AT89s52. Address line pins ADDA, ADDB, AADC are connected at P3.0, P3.1, and P3.2.

ALE (Address latch enable), SC (Start conversion), EOC (End of conversion), OE (Output enable) and clock pins are connected at P3.3, P3.4, P3.5, P3.6 and P3.7.

And here we have used three potentiometers connected at pin 26, 27, and 28 of ADC0808.

A 9 volt battery and a 5 volt voltage regulator namely 7805 are used for powering the circuit.
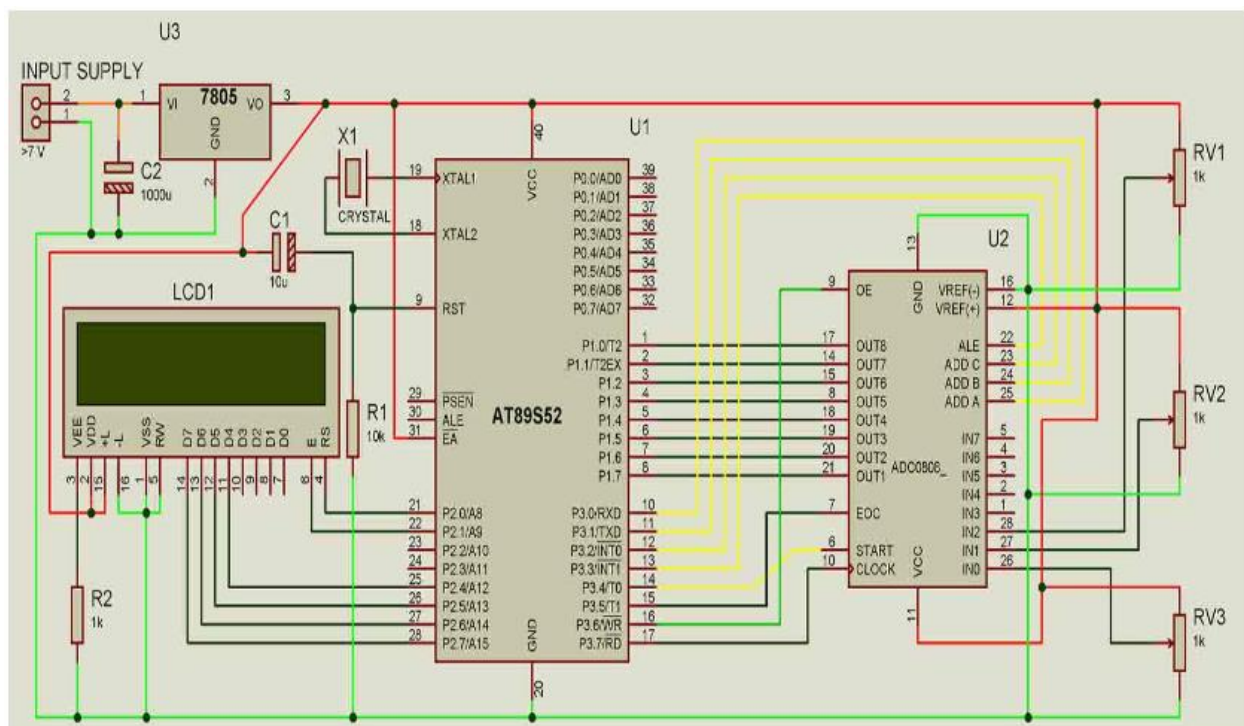
Working:

In this project we have interfaced three channels of ADC0808. And for demonstration we have used three variable resistors. When we power the circuit then microcontroller initialize the LCD by using appropriate command, gives clock to ADC chip, selects ADC channel by using address line and send start conversion signal to ADC. After this ADC first reads selected ADC channel input and gives its converted output to microcontroller. Then microcontroller shows its value at Ch1 position in LCD. And then microcontroller changes ADC channel by using address line. And then ADC reads selected channel and send output to microcontroller. And show on LCD as name Ch2. And like wise for other channels.
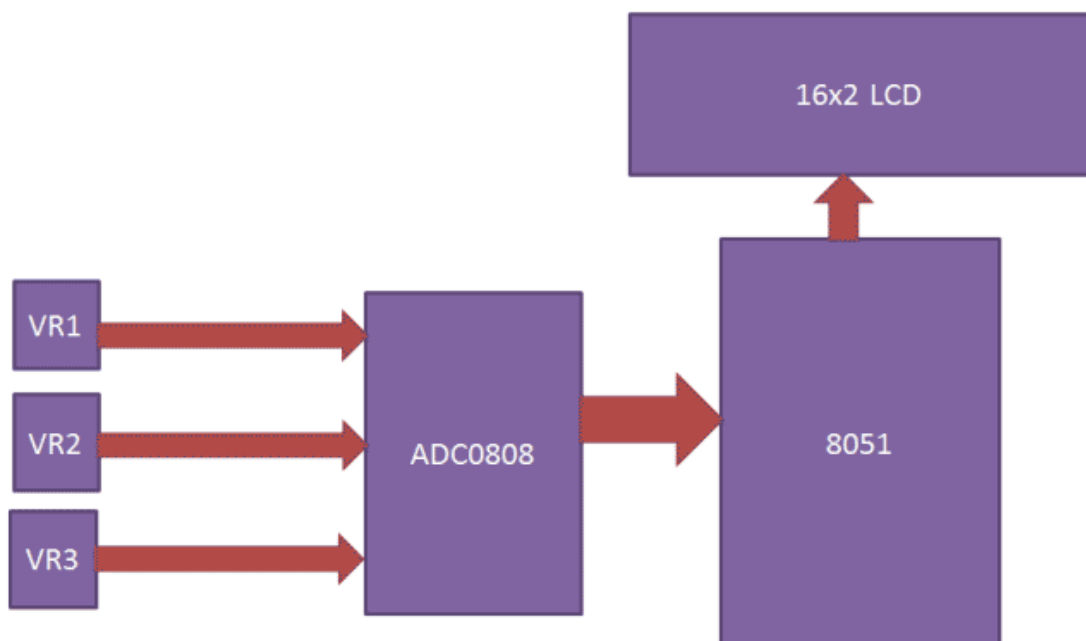
Working of ADC0808 is much similar to working of ADC0804. In this, first microcontroller provides a 500 KHz clock signal to ADC0808, using the Timer 0 interrupt, as ADC requires clock signal to operate. Now microcontroller sends a LOW to HIGH level signal to ALE pin (its active-high pin) of ADC0808 to enable the latch in the address. Then by applying HIGH to LOW Level signal to SC (Start Conversion), ADC starts analog to digital conversion. And then wait for the EOC (End of Conversion) pin to go LOW. When EOC goes LOW, it means analog to digital conversion has been completed and data is ready to use. After this, microcontroller enables the output line by applying a HIGH to LOW signal to OE pin of ADC0808.

ADC0808 gives ratio metric conversion output at its output pins. And the formula for radiometric conversion is given by:

$V_{in}/(V_{fs}-V_z)= D_x/(D_{max}-D_{min})$

Where

V_{in} is input voltage for conversion
V_{fs} is full scale Voltage
V_z is zero voltage
D_x is data point being measure

D_{max} is Maximum data limit
D_{min} is Minimum data limit

Program

In the program, first of all we include header file sand defines variable and input & output pins for ADC and LCD.
include<reg51.h>
#include<stdio.h>
sbit ale=P3^3;
sbit oe=P3^6;
sbit sc=P3^4;
sbit eoc=P3^5;
sbit clk=P3^7;
sbit ADDA=P3^0; //Address pins for selecting input channels.
sbit ADDB=P3^1;
sbit ADDC=P3^2;
#define lcdport P2 //lcd
sbit rs=P2^0;
sbit rw=P2^2;
sbit en=P2^1;
#define input_port P1 //ADC
int result[3],number;

Function for creating the delay has been created (void delay), along with some LCD functions like for LCD initialization, printing the string, for LCD commands etc. You can easily find them in Code. Check this article for LCD interfacing with 8051 and its functions. After this in main program we have initialize LCD and set the EOC, ALE, EO, SC pins accordingly.

```
void main()
{
 int i=0;
 eoc=1;
 ale=0;
 oe=0;
 sc=0;
 TMOD=0x02;
 TH0=0xFD;
lcd_ini( );
lcdprint("ADC 0808/0809 ");
```

And then program reads the ADC and stores ADC output in a variable and then sends it to LCD after decimal to ASCII conversion, using void read_adc( ) and void adc(int i) functions:
void read_adc( )

```
{
  number=0;
  ale=1;
  sc=1;
  delay(1);
  ale=0;
  sc=0;
  while(eoc==1);
  while(eoc==0);
  oe=1;
  number=input_port;
  delay(1);
  oe=0;
}

void adc(int i)
{
switch(i)
 {
  case 0:
   ADDC=0;
   ADDB=0;
   ADDA=0;
   lcdcmd(0xc0);
   read_adc( );
```

Full Program

```
        # include<reg51.h>
        #include<stdio.h>
        sbit ale=P3^3;
        sbit oe=P3^6;
        sbit sc=P3^4;
        sbit eoc=P3^5;
        sbit clk=P3^7;
        sbit ADDA=P3^0;  //Address pins for selecting input channels.
        sbit ADDB=P3^1;
        sbit ADDC=P3^2;
        #define lcdport P2  //lcd
        sbit rs=P2^0;
        sbit rw=P2^2;
        sbit en=P2^1;
        #define input_port P1  //ADC
        int result[3],number;
```

```c
void timer0() interrupt 1  // Function to generate clock of frequency 500KHZ
using Timer 0 interrupt.
{
clk=~clk;
}

void delay(unsigned int count)
{
int i,j;
for(i=0;i<count;i++)
 for(j=0;j<100;j++);
}

void daten()
{
  rs=1;
  rw=0;
  en=1;
  delay(1);
  en=0;
}

void lcd_data(unsigned char ch)
{
  lcdport=ch & 0xF0;
  daten();
  lcdport=ch<<4 & 0xF0;
  daten();
}

void cmden(void)
{
  rs=0;
  en=1;
  delay(1);
  en=0;
}

void lcdcmd(unsigned char ch)
{

  lcdport=ch & 0xf0;
  cmden();
  lcdport=ch<<4 & 0xF0;
```

```
        cmden();
}

lcdprint(unsigned char *str)  //Function to send string data to LCD.
{
  while(*str)
  {
    lcd_data(*str);
    str++;
  }
}

void lcd_ini()  //Function to inisialize the LCD
{
  lcdcmd(0x02);
  lcdcmd(0x28);
  lcdcmd(0x0e);
  lcdcmd(0x01);
}

void show()
{
  sprintf(result,"%d",number);
  lcdprint(result);
  lcdprint("  ");
}

void read_adc()
{
  number=0;
  ale=1;
  sc=1;
  delay(1);
  ale=0;
  sc=0;
  while(eoc==1);
  while(eoc==0);
  oe=1;
  number=input_port;
  delay(1);
  oe=0;
}

void adc(int i)  //Function to drive ADC
{
```

```
switch(i)
 {
  case 0:
   ADDC=0;  // Selecting input channel IN0 using address lines
   ADDB=0;
   ADDA=0;
   lcdcmd(0xc0);
   read_adc();
   show();
   break;

  case 1:
   ADDC=0;  // Selecting input channel IN1 using address lines
   ADDB=0;
   ADDA=1;
   lcdcmd(0xc6);
   read_adc();
   show();
  break;

  case 2:
   ADDC=0;  // Selecting input channel IN2 using address lines
   ADDB=1;
   ADDA=0;
   lcdcmd(0xcc);
   read_adc();
   show();
   break;
 }
}

void main()
{
 int i=0;
 eoc=1;
 ale=0;
 oe=0;
 sc=0;
 TMOD=0x02;
 TH0=0xFD;
lcd_ini();
lcdprint(" ADC 0808/0809 ");
lcdcmd(192);
lcdprint("  Interfacing  ");
```

```
            delay(500);
            lcdcmd(1);
            lcdprint("Circuit Digest ");
            lcdcmd(192);
            lcdprint("System Ready...   ");
            delay(500);
            lcdcmd(1);
            lcdprint("Ch1   Ch2   Ch3 ");
             IE=0x82;
             TR0=1;
            while(1)
            {
              for(i=0;i<3;i++)
              {
                adc(i);
                number=0;
              }
            }
            }
```

STEPPER MOTOR INTERFACING WITH 8051

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.
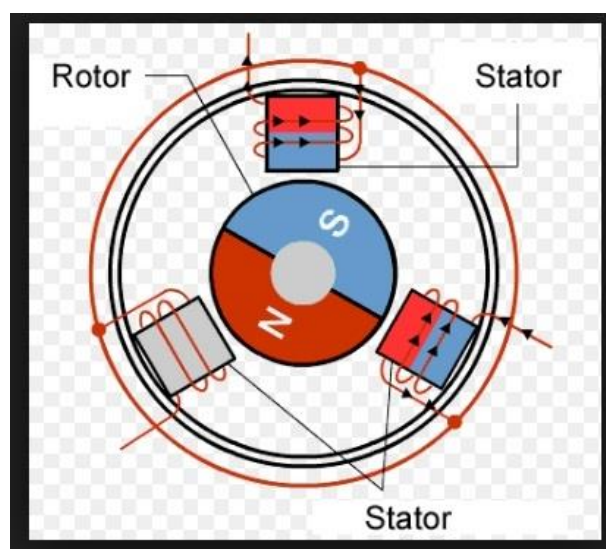
Normal motor shafts can move freely but the stepper motor shafts move in fixed repeatable increments.

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.

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

Steps	Winding A	Winding B	Winding C	Winding D
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

Program

```c
#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);
    }
  }
}
```

The circuit diagram is shown below: It uses the full drive mode.
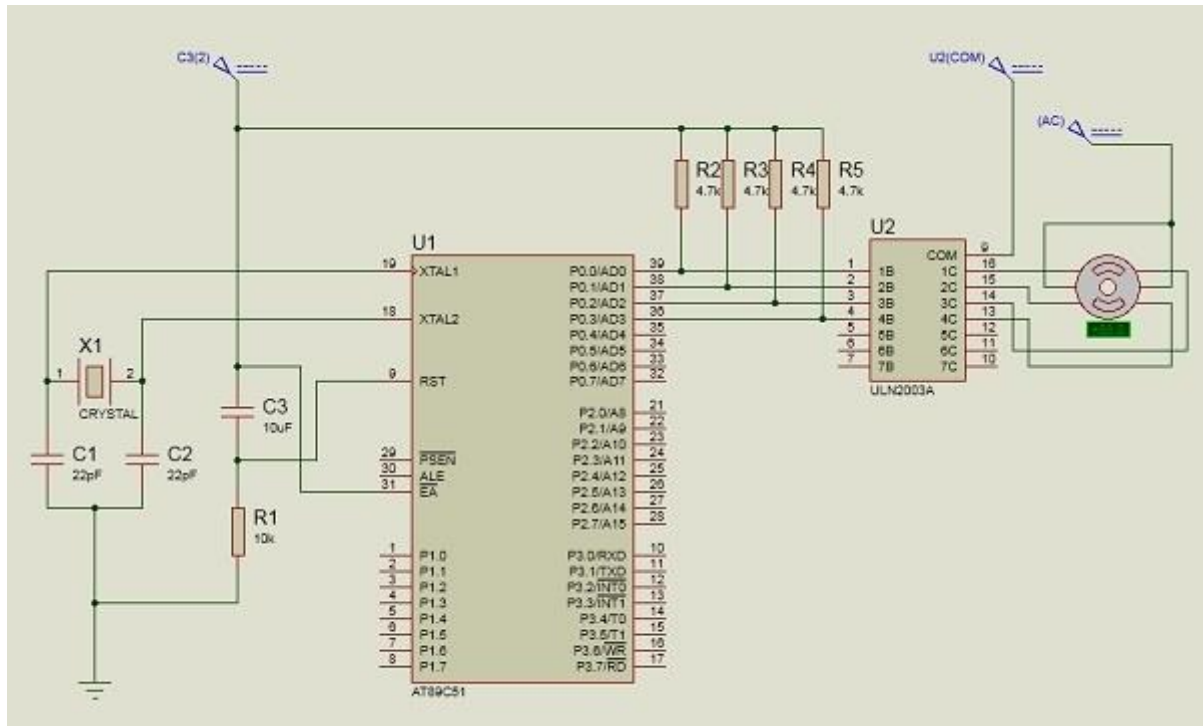
LCD INTERFACING WITH 8051 MICROCONTROLLER

Display units are the most important output devices in embedded projects and electronics products. 16x2 LCD is one of the most used display unit. 16x2 LCD means that there are two rows in which 16 characters can be displayed per line, and each character takes 5X7 matrix space on LCD. In this tutorial we are going to connect 16X2 LCD module to the 8051 microcontroller (AT89S52). **Interfacing LCD with 8051 microcontroller** might look quite complex to newbies, but after understanding the concept it would look very simple and easy. Although it may be time taking because you need to understand and connect 16 pins of LCD to the microcontroller. So first let's understand the 16 pins of LCD module.

We can divide it in five categories, Power Pins, contrast pin, Control Pins, Data pins and Backlight pins.

Category	Pin NO.	Pin Name	Function
Power Pins	1	VSS	Ground Pin, connected to Ground
	2	VDD or Vcc	Voltage Pin +5V
Contrast Pin	3	V0 or VEE	Contrast Setting, connected to Vcc thorough a variable resistor.
Control Pins	4	RS	Register Select Pin, RS=0 Command mode, RS=1 Data mode
	5	RW	Read/ Write pin, RW=0 Write mode, RW=1 Read mode
	6	E	Enable, a high to low pulse need to enable the LCD
Data Pins	7-14	D0-D7	Data Pins, Stores the Data to be displayed on LCD or the command instructions
Backlight Pins	15	LED+ or A	To power the Backlight +5V
	16	LED- or K	Backlight Ground

All the pins are clearly understandable by their name and functions, except the control pins, so they are explained below:

RS: RS is the register select pin. We need to set it to 1, if we are sending some data to be displayed on LCD. And we will set it to 0 if we are sending some command instruction like clear the screen (hex code 01).

RW: This is Read/write pin, we will set it to 0, if we are going to write some data on LCD. And set it to 1, if we are reading from LCD module. Generally this is set to 0, because we do not have need to read data from LCD. Only one instruction "Get LCD status", need to be read some times.

E: This pin is used to enable the module when a high to low pulse is given to it. A pulse of 450 ns should be given. That transition from HIGH to LOW makes the module ENABLE.

There are some preset command instructions in LCD, we have used them in our program below to prepare the LCD (in lcd_init() function). Some important command instructions are given below:

Hex Code	Command to LCD Instruction Register
0F	LCD ON, cursor ON
01	Clear display screen
02	Return home
04	Decrement cursor (shift cursor to left)
06	Increment cursor (shift cursor to right)
05	Shift display right
07	Shift display left
0E	Display ON, cursor blinking
80	Force cursor to beginning of first line
C0	Force cursor to beginning of second line
38	2 lines and 5×7 matrix
83	Cursor line 1 position 3
3C	Activate second line
08	Display OFF, cursor OFF
C1	Jump to second line, position 1
0C	Display ON, cursor OFF

C1	Jump to second line, position 1
C2	Jump to second line, position 2
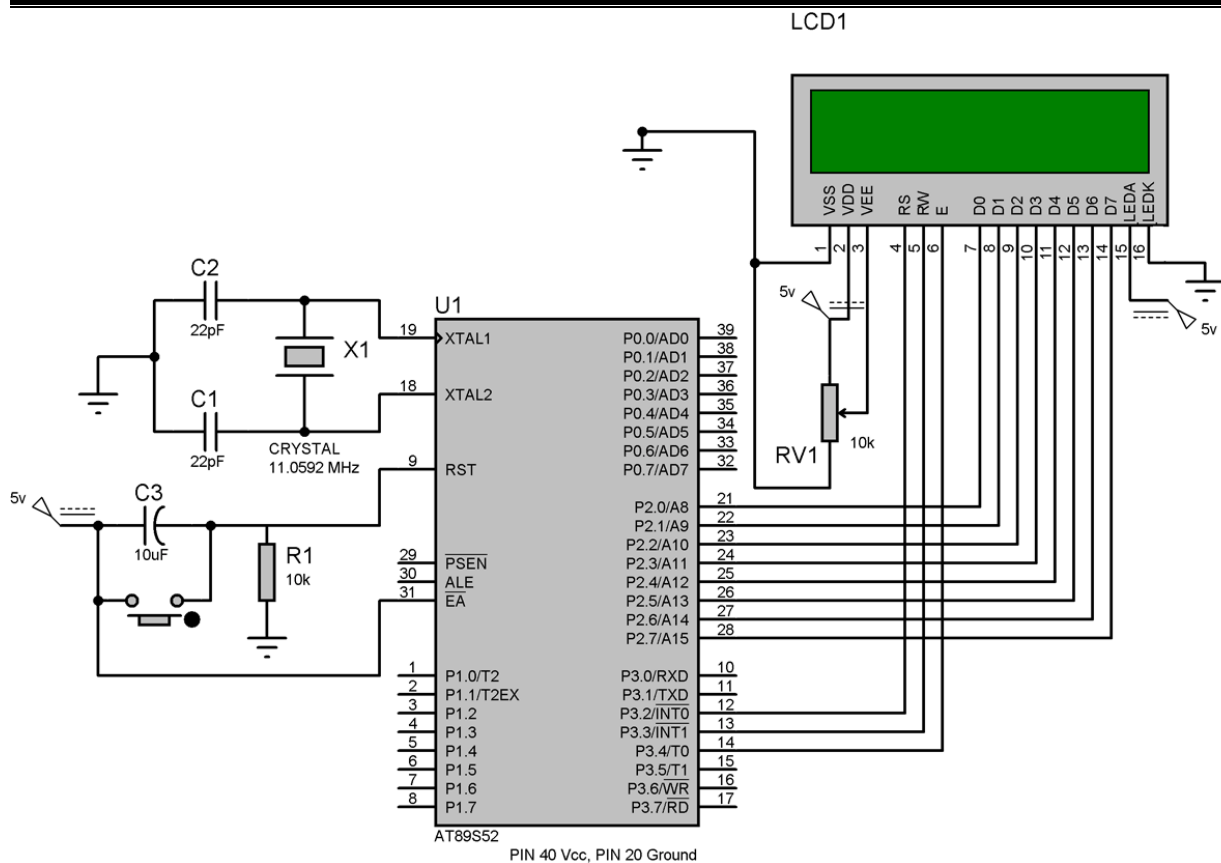
Circuit diagram for **LCD interfacing with 8051 microcontroller** is shown in the above figure. If you have basic understanding of 8051 then you must know about EA(PIN 31), XTAL1 & XTAL2, RST pin(PIN 9), Vcc and Ground Pin of 8051 microcontroller. I have used these Pins in above circuit.

So besides these above pins we have connected the data pins (D0-D7) of LCD to the Port 2 (P2_0 – P2_7) microcontroller. And control pins RS, RW and E to the pin 12,13,14 (pin 2,3,4 of port 3) of microcontroller respectively.

PIN 2(VDD) and PIN 15(Backlight supply) of LCD are connected to voltage (5v), and PIN 1 (VSS) and PIN 16(Backlight ground) are connected to ground.

Pin 3(V0) is connected to voltage (Vcc) through a variable resistor of 10k to adjust the contrast of LCD. Middle leg of the variable resistor is connected to PIN 3 and other two legs are connected to voltage supply and Ground.

Program

// Program for LCD Interfacing with 8051 Microcontroller (AT89S52)

```c
#include<reg51.h>
#define display_port P2     //Data pins connected to port 2 on microcontroller
sbit rs = P3^2;  //RS pin connected to pin 2 of port 3
sbit rw = P3^3;  // RW pin connected to pin 3 of port 3
sbit e =  P3^4;  //E pin connected to pin 4 of port 3

void msdelay(unsigned int time)  // Function for creating delay in milliseconds.
{
   unsigned i,j ;
   for(i=0;i<time;i++)
   for(j=0;j<1275;j++);
}
void lcd_cmd(unsigned char command)  //Function to send command instruction to LCD
{
   display_port = command;
   rs= 0;
   rw=0;
   e=1;
   msdelay(1);
   e=0;
}

void lcd_data(unsigned char disp_data)  //Function to send display data to LCD
{
   display_port = disp_data;
   rs= 1;
   rw=0;
   e=1;
   msdelay(1);
   e=0;
}

 void lcd_init()   //Function to prepare the LCD  and get it ready
{
  lcd_cmd(0x38);  // for using 2 lines and 5X7 matrix of LCD
  msdelay(10);
  lcd_cmd(0x0F);  // turn display ON, cursor blinking
  msdelay(10);
  lcd_cmd(0x01);  //clear screen
  msdelay(10);
```

```
  lcd_cmd(0x81);  // bring cursor to position 1 of line 1
  msdelay(10);
}
void main()
{
  unsigned char a[15]="CIRCUIT DIGEST";   //string of 14 characters with a null terminator.
  int l=0;
  lcd_init();
  while(a[l] != '\0') // searching the null terminator in the sentence
  {
    lcd_data(a[l]);
    l++;
    msdelay(50);
  }
}
```
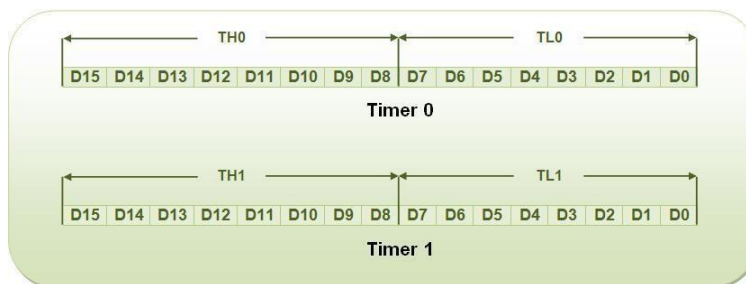
MODULE IV
ADVANCED CONCEPTS

TIMERS AND COUNTERS

Timers/Counters are used generally for

- Time reference
- Creating delay
- Wave form properties measurement
- Periodic interrupt generation
- Waveform generation

8051 has two timers, Timer 0 and Timer 1.

Timer in 8051 is used as timer, counter and baud rate generator. Timer always counts up irrespective of whether it is used as timer, counter, or baud rate generator: Timer is always incremented by the microcontroller. The time taken to count one digit up is based on master clock frequency.

If Master CLK=12 MHz,

Timer Clock frequency = Master CLK/12 = 1 MHz
Timer Clock Period = 1micro second

This indicates that one increment in count will take 1 micro second.

The two timers in 8051 share two SFRs (TMOD and TCON) which control the timers, and each timer also has two SFRs dedicated solely to itself (TH0/TL0 and TH1/TL1).

The following are timer related SFRs in 8051.

SFR Name	Description	SFR Address
TH0	*Timer 0 High Byte*	*8Ch*
TL0	*Timer 0 Low Byte*	*8Ah*
TH1	*Timer 1 High Byte*	*8Dh*
TL1	*Timer 1 Low Byte*	*8Bh*
TCON	*Timer Control*	*88h*
TMOD	*Timer Mode*	*89h*

TMOD Register

TMOD : Timer/Counter Mode Control Register (Not Bit Addressable)

GATE	C/T̄	M1	M0	GATE	C/T̄	M1	M0

TIMER 1	TIMER 0

GATE   When TRx (in TCON) is set and GATE = 1, TIMER/COUNTERx will run only while INTx pin is high (hardware control). When GATE = 0, TIMER/COUNTERx will run only while TRx = 1 (software control).

C/T̄   Timer or Counter selector. Cleared for Timer operation (input from internal system clock). Set for Counter operation (input from Tx input pin).

M1   Mode selector bit (NOTE 1).

M0   Mode selector bit (NOTE 1).

Note 1 :

M1	M0		OPERATING MODE
0	0	0	13–bit Timer
0	1	1	16–bit Timer/Counter
1	0	2	8–bit Auto–Reload Timer/Counter
1	1	3	(Timer 0) TL0 is an 8–bit Timer/Counter controlled by the standard Timer 0 control bits, TH0 is an 8–bit Timer and is controlled by Timer 1 control bits.
1	1	3	(Timer 1) Timer/Counter 1 stopped.

8051 timers have both software and hardware controls. The start and stop of a timer is controlled by software using the instruction **SETB TR1** and **CLR TR1** for timer 1, and **SETB TR0** and **CLR TR0** for timer 0.

The SETB instruction is used to start it and it is stopped by the CLR instruction. These instructions start and stop the timers as long as GATE = 0 in the TMOD register. Timers can be started and stopped by an external source by making GATE = 1 in the TMOD register.
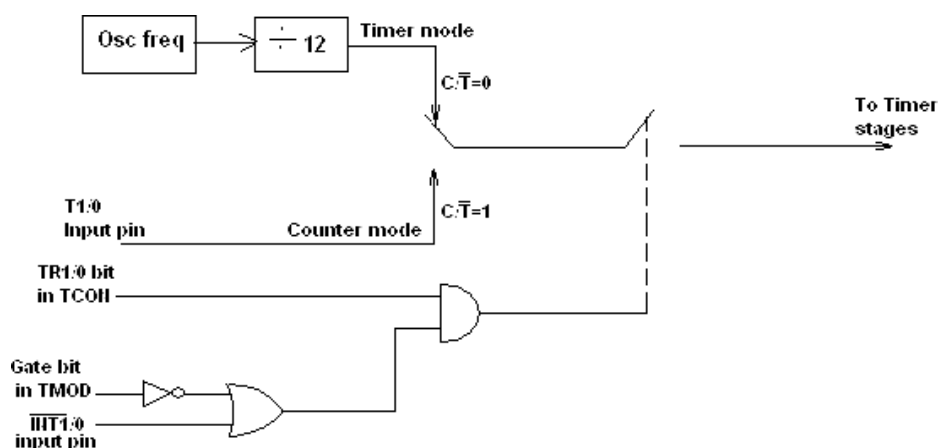
TCON Register

TCON : Timer/Counter Control Register (Bit Addressable)

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TF1	TCON.7	Timer 1 overflow flag. Set by hardware when the Timer/Counter 1 overflows. Cleared by hardware as processor vectors to the interrupt service routine.
TR1	TCON.6	Timer 1 run control bit. Set/cleared by software to turn Timer/Counter ON/OFF.
TF0	TCON.5	Timer 0 overflow flag. Set by hardware when the Timer/Counter 0 overflows. Cleared by hardware as 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 hardware when External interrupt edge is detected. Cleared by hardware when interrupt is processed.
IT1	TCON.2	Interrupt 1 type control bit. Set/cleared by software to specify falling edge/flow level triggered External Interrupt.
IE0	TCON.1	External Interrupt 0 edge flag. Set by hardware when External Interrupt edge detected. Cleared by hardware when interrupt is processed.
IT0	TCON.0	Interrupt 0 type control bit. Set/cleared by software to specify falling edge/low level triggered External Interrupt.

Timer/ Counter Control Logic.

TIMER MODES

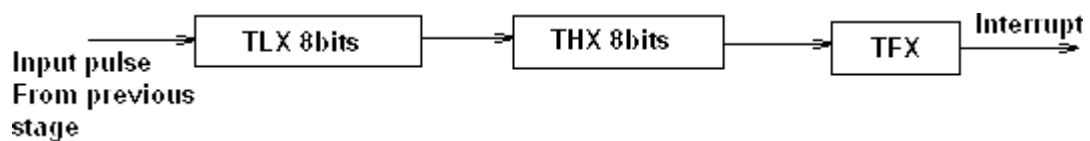Timers can operate in four different modes. They are as follows

Timer Mode-0: In this mode, the timer is used as a 13-bit UP counter as follows.
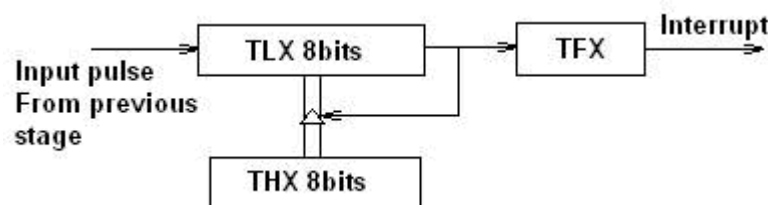

Fig. Operation of Timer on Mode-0

The lower 5 bits of TLX and 8 bits of THX are used for the 13 bit count.Upper 3 bits of TLX are ignored. When the counter rolls over from all 0's to all 1's, TFX flag is set and an interrupt is generated. The input pulse is obtained from the previous stage. If TR1/0 bit is 1 and Gate bit is 0,the counter continues counting up. If TR1/0 bit is 1 and Gate bit is 1, then the operation of the counter is controlled by input. This mode is useful to measure the width of a given pulse fed to input.

Timer Mode-1: This mode is similar to mode-0 except for the fact that the Timer operates in 16-bit mode.

Fig: Operation of Timer in Mode 1

Timer Mode-2: (Auto-Reload Mode): This is a 8 bit counter/timer operation. Counting is performed in TLX while THX stores a constant value. In this mode when the timer overflows i.e. TLX becomes FFH, it is fed with the value stored in THX. For example if we load THX with 50H then the timer in mode 2 will count from 50H to FFH. After that 50H is again reloaded. This mode is useful inapplications like fixed time sampling.

Fig: Operation of Timer in Mode 2

Timer Mode-3: Timer 1 in mode-3 simply holds its count. The effect is same as setting TR1=0.Timer0 in mode-3 establishes TL0 and TH0 as two separate counters.
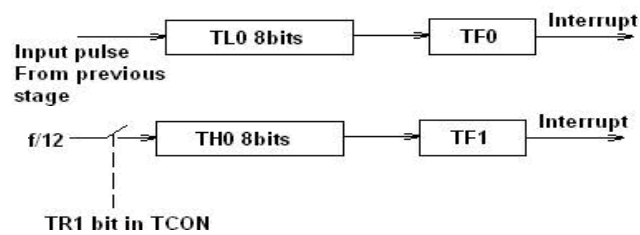
Fig: Operation of Timer in Mode 3

Control bits TR1 and TF1 are used by Timer-0 (higher 8 bits) (TH0) in Mode-3 while TR0 and TF0are available to Timer-0 lower 8 bits(TL0).
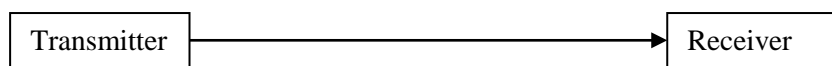
SERIAL COMMUNICATION.

DATA COMMUNICATION

The 8051 microcontroller is parallel device that transfers eight bits of data simultaneously over eight data lines to parallel I/O devices. Parallel data transfer over a long is very expensive. Hence, a serial communication is widely used in long distance communication. In serial data communication, 8-bit data is converted to serial bits using a parallel in serial out shift register and then it is transmitted over a single data line. The data byte is always transmitted with least significant bit first.

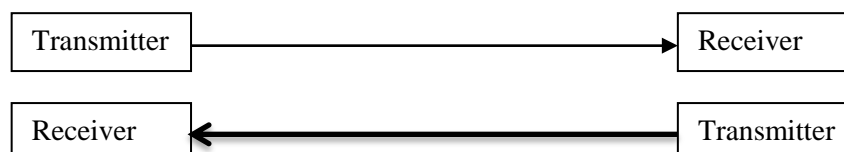## BASICS OF SERIAL DATA COMMUNICATION,
Communication Links

> *1. Simplex communication link:* In simplex transmission, the line is dedicated for transmission.The transmitter sends and the receiver receives the data.

2. Half duplex communication link: In half duplex, the communication link can be used for eithertransmission or reception. Data is transmitted in only one direction at a time.


> *3. Full duplex communication link:* If the data is transmitted in both ways at the same time, it is a full duplex i.e. transmission and reception can proceed simultaneously. This communication link requires two wires for data, one for transmission and one for reception.

Types of Serial communication:

Serial data communication uses two types of communication.

1. Synchronous serial data communication: In this transmitter and receiver are synchronized. It uses a common clock to synchronize the receiver and the transmitter. First the synch character is sent and then the data is transmitted. This format is generally used for high speed transmission..
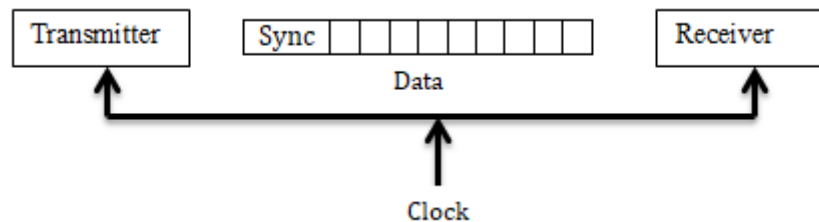
In Synchronous serial data communication a block of data is transmitted at a time

2. Asynchronous Serial data transmission: In this, different clock sources are used for transmitter and receiver. In this mode, data is transmitted with start and stop bits. A transmission begins with start bit, followed by data and then stop bit. For error checking purpose parity bit is included just prior to stop bit. In Asynchronous serial data communication a single byte is transmitted at a time.
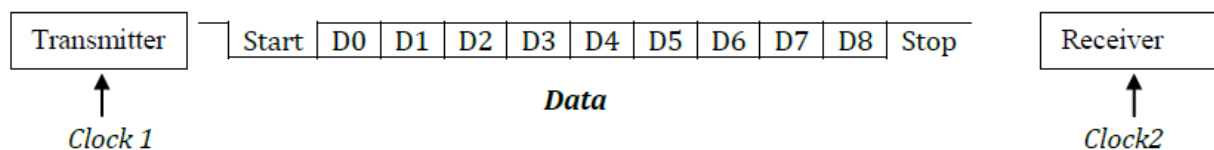
Baud rate:

The rate at which the data is transmitted is called baud or transfer rate. The baud rate is the reciprocal of the time to send one bit. In asynchronous transmission, baud rate is not equal to number of bits per second. This is because; each byte is preceded by a start bit and followed by parity and stop bit. For example, in synchronous transmission, if data is transmitted with 9600 baud, it means that 9600 bits are transmitted in one second. For bit transmission time = 1 second/ 9600 = 0.104 ms.

6.1.1. 8051 SERIAL COMMUNICATION

The 8051 supports a full duplex serial port.
Three special function registers support serial communication.

1. **SBUF Register**: Serial Buffer (SBUF) register is an 8-bit register. It has separate SBUF registers for data transmission and for data reception. For a byte of data to be transferred via the TXD line, it must be placed in SBUF register. Similarly, SBUF holds the 8-bit data received by the RXD pin and read to accept the received data.
2. **SCON register**: The contents of the Serial Control (SCON) register are shown below. This register contains mode selection bits, serial port interrupt bit (TI and RI) and also the ninth data bit for transmission and reception (TB8 and RB8).

Serial Port Control (SCON) Register

D7	D6	D5	D4	D3	D2	D1	D0
SM0	SM1	SM2	REN	TB8	RB8	TI	RI

- o SM0 (SCON.7) : Serial communication mode selection bit
- o SM1 (SCON.6) : Serial communication mode selection bit

SM0	SM1	Mode	Description	Baud rate
0	0	Mode 0	8-bit shift register mode	Fosc / 12
0	1	Mode 1	8-bit UART	Variable (set by timer 1)
1	0	Mode 2	9-bit UART	Fosc/ 32 or Fosc/64
1	1	Mode 3	9-bit UART	Variable (set by timer 1)

- o SM2 (SCON.5) : Multiprocessor communication bit. In modes 2 and 3, if set this will enable multiprocessor communication.

- o REN (SCON.4) : Enable serial reception

- o TB8 (SCON.3) : This is 9th bit that is transmitted in mode 2 & 3.

- o RB8 (SCON.2) : 9th data bit is received in modes 2 & 3.

- o TI (SCON.1)   : Transmit interrupt flag, set by hardware must be cleared by software.

- o RI (SCON.0)   : Receive interrupt flag, set by hardware must be cleared by software.

3. **PCON register**: The SMOD bit (bit 7) of PCON register controls the baud rate in asynchronous mode transmission.

Power mode Control (PCON) Register

D7	D6	D5	D4	D3	D2	D1	D0
SMOD	--	--	--	GF1	GF0	PD	IDL

- o SMD (PCON.7): Serial rate modify bit. Set to 1 by program to double baud rate using timer 1 for modes 1, 2, and 3. cleared by program to use timer 1 baud rate.

- o GF1 (PCON.3) : General Purpose user flag bit.
- o GF0 (PCON.2) : General Purpose user flag bit.
- o PD (PCON.1) : Power down bit. Set to 1 by program to enter power down configuration for CHMOS processors.
- o IDL (PCON.0) : Idle mode bit. Set to 1 by program to enter idle mode configuration for CHMOS processors.

SERIAL COMMUNICATION MODES

1. *Mode 0*

    In this mode serial port runs in synchronous mode. The data is transmitted and received through RXD pin and TXD is used for clock output. In this mode the baud rate is 1/12 of clock frequency.

2. *Mode 1*

In this mode SBUF becomes a 10 bit full duplex transceiver. The ten bits are 1 start bit, 8 data bit and 1 stop bit. The interrupt flag TI/RI will be set once transmission or reception isover. In this mode the baud rate is variable and is determined by the timer 1 overflow rate.

Baud rate   = $[2^{smod}/32]$ x Timer 1 overflow Rate

              = $[2^{smod}/32]$ x [Oscillator Clock Frequency] / [12 x [256 – [TH1]]]

3. *Mode 2*

This is similar to mode 1 except 11 bits are transmitted or received. The 11 bits are, 1 startbit, 8 data bit, a programmable 9th data bit, 1 stop bit.

Baud rate   = $[2^{smod}/64]$ x Oscillator Clock Frequency

4. *Mode 3*

This is similar to mode 2 except baud rate is calculated as in mode 1


CONNECTIONS TO RS-232

RS-232 standards:

To allow compatibility among data communication equipment made by various manufactures, an interfacing standard called RS232 was set by the Electronics Industries Association (EIA) in 1960. Since the standard was set long before the advent of logic family, its input and output voltage levels are not TTL compatible.

In RS232, a logic one (1) is represented by -3 to -25V and referred as MARK while logic zero

(0) is represented by +3 to +25V and referred as SPACE. For this reason to connect any RS232 to a microcontroller system we must use voltage converters such as MAX232 to convert the TTL logic level to RS232 voltage levels and vice-versa. MAX232 IC chips are commonly referred as linedrivers.

In RS232 standard we use two types of connectors. DB9 connector or DB25 connector.
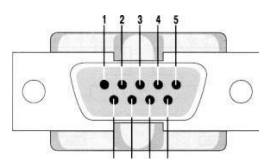

DB9 Male Connector   *DB25 Male*
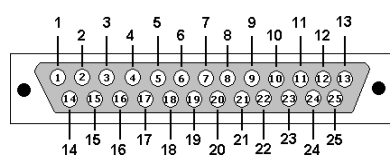
ConnectorThe pin description of DB9 and DB25 Connectors are as follows

DB-25 Pin No.	DB-9 Pin No.	Abbreviation	Full Name
Pin 2	Pin 3	TD	Transmit Data
Pin 3	Pin 2	RD	Receive Data
Pin 4	Pin 7	RTS	Request To Send
Pin 5	Pin 8	CTS	Clear To Send
Pin 6	Pin 6	DSR	Data Set Ready
Pin 7	Pin 5	SG	Signal Ground
Pin 8	Pin 1	CD	Carrier Detect
Pin 20	Pin 4	DTR	Data Terminal Ready
Pin 22	Pin 9	RI	Ring Indicator

The 8051 connection to MAX232 is as follows.

The 8051 has two pins that are used specifically for transferring and receiving data serially. These two pins are called TXD, RXD. Pin 11 of the 8051 (P3.1) assigned to TXD and pin 10 (P3.0) is designated as RXD. These pins TTL compatible; therefore they require line driver (MAX 232) to make them RS232 compatible. MAX 232 converts RS232 voltage levels to TTL voltage levels and vice versa. One advantage of the MAX232 is that it uses a +5V power source which is the same as the source voltage for the 8051. The typical connection diagram between MAX 232 and 8051 is shown below.
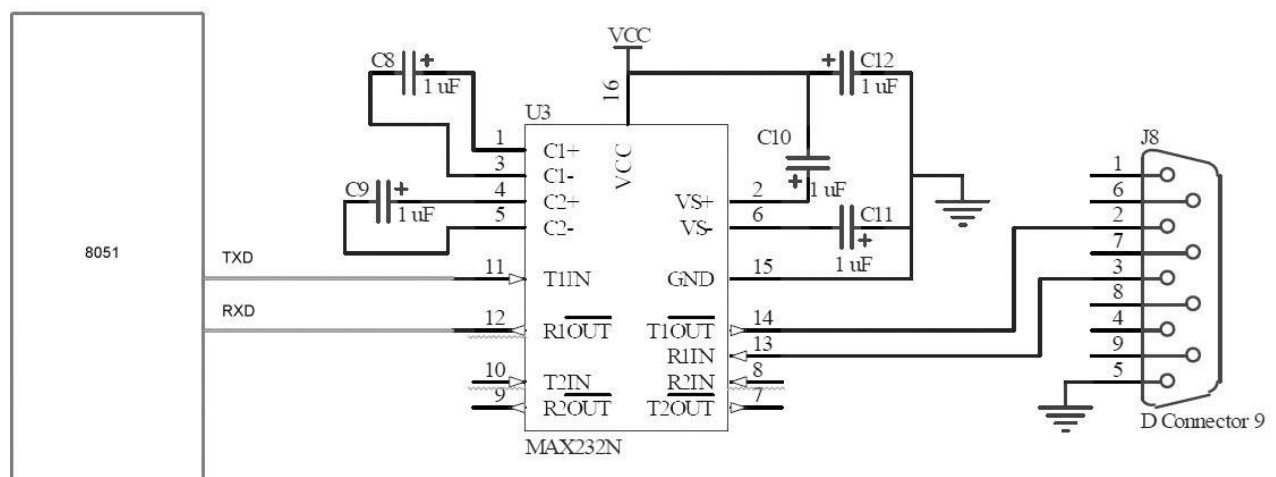
SERIAL COMMUNICATION PROGRAMMING IN ASSEMBLY AND C.

Steps to programming the 8051 to transfer data serially

1. The TMOD register is loaded with the value 20H, indicating the use of the Timer 1 inmode 2 (8-bit auto reload) to set the baud rate.
2. The TH1 is loaded with one of the values in table 5.1 to set the baud rate for serialdata transfer.
3. The SCON register is loaded with the value 50H, indicating serial mode 1, where an8-bit data is framed with start and stop bits.
4. TR1 is set to 1 start timer 1.
5. TI is cleared by the "CLR TI" instruction.
6. The character byte to be transferred serially is written into the SBUF register.
7. The TI flag bit is monitored with the use of the instruction JNB TI, target to see if thecharacter has been transferred completely.
8. To transfer the next character, go to step 5.

Example 1. Write a program for the 8051 to transfer letter 'A' serially at 4800- baud rate, 8 bit data,1 stop bit continuously.

```
ORG 0000H
LJMP START
ORG 0030H
```

```
START: MOV TMOD, #20H     ; select timer 1 mode 2
       MOV TH1, #0FAH      ; load count to get baud rate of 4800
       MOV SCON, #50H      ; initialize UART in mode 2
                           ; 8 bit data and 1 stop bit
       SETB TR1            ; start timer
AGAIN: MOV SBUF, #'A'      ; load char 'A' in SBUF
BACK:  JNB TI, BACK        ; Check for transmit interrupt flag
       CLR TI              ; Clear transmit interrupt flag
       SJMP AGAIN
       END
```

Example 2. Write a program for the 8051 to transfer the message 'EARTH' serially at 9600 baud, 8 bit data, 1 stop bit continuously.

```
        ORG 0000H
        LJMP START

        ORG 0030H
        START: MOV TMOD, #20H          ; select timer 1 mode 2
        MOV TH1, #0FDH                 ; load count to get reqd. baud rate of 9600
        MOV SCON, #50H                 ; initialise uart in mode 2
                                       ; 8 bit data and 1 stop bit
        SETB TR1                       ; start timer
        LOOP: MOV A, #'E'      ; load 1st letter 'E' in
        AACALL LOAD                    ; call load subroutine
        MOV A, #'A'           ; load 2nd letter 'A' in A
        ACALL LOAD                     ; call load subroutine
        MOV A, #'R'           ; load 3rd letter 'R' in A
        ACALL LOAD                     ; call load subroutine
        MOV A, #'T'           ; load 4th letter 'T' in A
        ACALL LOAD                     ; call load subroutine
        MOV A, #'H'           ; load 4th letter 'H' in A
        ACALL LOAD                     ; call load subroutine
        SJMP LOOP                      ; repeat steps
        LOAD: MOV SBUF, A
        HERE: JNB TI, HERE             ; Check for transmit interrupt flag
        CLR TI                         ; Clear transmit interrupt flag
        RET

        END
```

ARM 7 MICROCONTROLLER

INTRODUCTION:

The ARM was originally developed at Acorn Computers Limited of Cambridge , England, between 1983 and 1985. It was the first RISC microprocessor developed for commercial use and has some significant differences from subsequent RISC architectures. In 1990 ARM Limited was established as a separate company specifically to widen the exploitation of ARM technology and it is established as a market-leader for low-power and cost-sensitive embedded applications. The ARM is supported by a toolkit which includes an instruction set emulator for hardware modelling and software testing and benchmarking, an assembler, C and C++ compilers, a linker and a symbolic debugger.

The 16-bit CISC microprocessors that were available in 1983 were slower than standard memory parts. They also had instructions that took many clock cycles to complete (in some cases, many hundreds of clock cycles), giving them very long interrupt latencies. As a result of these frustrations with the commercial microprocessor offerings, the design of a proprietary microprocessor was considered and ARM chip was designed.

ARM 7TDMI-S Processor :

The ARM7TDMI-S processor is a member of the ARM family of general-purpose 32-bit microprocessors. The ARM family offers high performance for very low-power consumption and gate count. The ARM7TDMI-S processor has a Von Neumann architecture, with a single 32-bit data bus carrying both instructions and data. Only load, store, and swap instructions can access data from memory. The ARM7TDMI-S processor uses a three stage pipeline to increase the speed of the flow of instructions to the processor. This enables several operations to take place simultaneously, and the processing, and memory systems to operate continuously. In the three-stage pipeline the instructions are executed in three stages
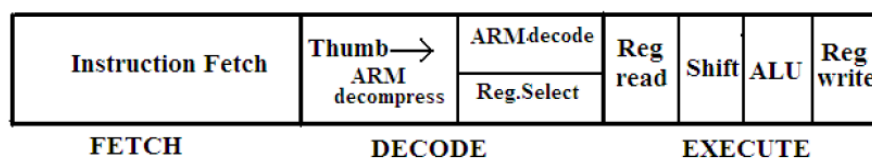
The three stage pipelined architecture of the ARM7 processor is shown in the above figure.

ARM7TDMIS stands for
      T: THUMB ;
      D for on-chip Debug support, enabling the processor to halt in response to a debug request,
      M: enhanced Multiplier, yield a full 64-bit result, high performance

I: Embedded ICE hardware (In Circuit emulator)
S : Synthesizable

FEATURES OF ARM PROCESSORS

The ARM processors are based on RISC architectures and this architecture has provided small implementations, and very low power consumption. Implementation size, performance, and very low power consumption remain the key features in the development of the ARM devices.

The typical RISC architectural features of ARM are :

❖ A large uniform register file
❖ A load/store architecture, where data-processing operations only operate on register
❖ contents, not directly on memory contents
❖ Simple addressing modes, with all load/store addresses being determined from register
❖ contents and instruction fields only uniform and fixed-length instruction fields, to
❖ simplify instruction decode.
❖ Control over both the Arithmetic Logic Unit (ALU) and shifter in most data-processing
❖ instructions to maximize the use of an ALU and a shifter
❖ Auto-increment and auto-decrement addressing modes to optimize program loops
❖ Load and Store Multiple instructions to maximize data throughput
❖ Conditional execution of almost all instructions to maximize execution throughput.

ARM Processor Families

Architecture version	Processor Families	Processor	Features	Microcontroller
ARM v4T	ARM7TDMI (1995)	ARM720T ARM740T	Von Neumann, 3-stage pipeline	LPC2100 series
	ARM9TDMI	ARM920T ARM922T ARM942T	MMU, Harvard, 5-stage pipeline	SAM9G, LPC29xx, LPC3xxx, STR9
ARM v5TE, ARM v5TEJ	ARM9E (1997)	ARM926EJ-S,	MMU, DSP, Jazelle,	SAM9XE
		ARM946E-S,	MPU, DSP	
		ARM966HS	MPU (optional), DSP	
	ARM10E	ARM1020E	MMU, DSP	

	(1999)	ARM1026EJ-S	MMU/MPU, DSP, Jazelle	
ARM v6	ARM11 (2003)	ARM1136J(F)-S	MMU, TrustZone, DSP, Jazelle	MSM7000, i.MX3x
		ARM1156T2(F)-S	MPU, DSP	
		ARM1176JZ(F)-S,	MMU, TrustZone, DSP, Jazelle	BCM2835
		ARM11 MP core N	MMU, Multiprocessor cache support DSP, Jazelle	
ARM v4T	ARM7TDMI (1995)	ARM720T ARM740T	Von Neumann, 3-stage pipeline	LPC2100 series
	ARM9TDMI	ARM920T ARM922T ARM942T	MMU, Harvard, 5-stage pipeline	SAM9G, LPC29xx, LPC3xxx, STR9
ARM v5TE, ARM v5TEJ	ARM9E (1997)	ARM926EJ-S,	MMU, DSP, Jazelle,	SAM9XE
		ARM946E-S,	MPU, DSP	
		ARM966HS	MPU (optional), DSP	
	ARM10E (1999)	ARM1020E	MMU, DSP	
		ARM1026EJ-S	MMU/MPU, DSP, Jazelle	
ARM v6	ARM11 (2003)	ARM1136J(F)-S	MMU, TrustZone, DSP, Jazelle	MSM7000, i.MX3x
		ARM1156T2(F)-S	MPU, DSP	
		ARM1176JZ(F)-S,	MMU, TrustZone, DSP, Jazelle	BCM2835
		ARM11 MP core N.	MMU/MPU, DSP, Jazelle	
ARM v6-M	Cortex	Cortex-M0	NVIC	LPC1200, 1100 series STM32F0x0, x1, x2

		Cortex-M1	FPGA TCM Interface, NVIC	STM32F1, F2, L1, W
ARM v7-M	Cortex	Cortex-M3	MPU (optional), NVIC	ST32F512-M, LPC1300, 1700, 1800
ARM v7-R	Cortex	Cortex-R4	MPU, DSP	STA1095, SAM4L, SAM4N, SAM4S
		Cortex-R4F	MPU, DSP, Floating Point	SAM4C, SAM4E, LPC40xx, 43xx, STM32 F3, F4
ARM v7-A	Cortex	Cortex-A8	MMU, Trust Zone, DSP, Jazelle, Neon, Floating Point	Freescale i.MX5X
		Cortex-A9	**MMU, Trust Zone, Multiprocessor, DSP, Jazelle, Neon Floating Point**	Freescale i.MX6QP

There are three basic instruction sets for ARM.

- A 32- bit ARM instruction set
- A 16 –bit Thumb instruction set and
- The 8-bit Java Byte code used in Jazelle state

The Thumb instruction set is a subset of the most commonly used32-bit ARM instructions. Thumb instructions operate with the standard ARM register configurations ,enabling excellent interoperability between ARM and Thumb states. This Thumb state is nearly 65% of the ARM code and can provide 160%of the performance of ARM code when working ona 16-bit memory system. This Thumb mode is used in embedded systems where memory resources are limited. The Jazelle mode is used in ARM9 processor to work with 8-bit Java code.

ARCHITECTURE OF ARM PROCESSORS:

The ARM 7 processor is based on Von Neman model with a single bus for both data and instructions..(The ARM9 uses Harvard model).Though this will decrease the performance of ARM, it is overcome by the pipe line concept. ARM uses the Advanced Microcontroller Bus Architecture (AMBA) bus architecture. This AMBA include two system buses: the AMBA High-Speed Bus (AHB) or the Advanced System Bus (ASB), and the Advanced Peripheral Bus (APB).

The ARM processor consists of

- ❖ Arithmetic Logic Unit (32-bit)
- ❖ One Booth multiplier(32-bit)
- ❖ One Barrel shifter

❖ One Control unit
❖ Register file of 37 registers each of 32 bits.

In addition to this the ARM also consists of a **Program status register** of 32 bits, Some special registers like the **instruction register**, memory data read and write register and memory address register ,one **Priority encoder** which is used in the multiple load and store instruction to indicate which register in the register file to be loaded or stored and Multiplexers etc.
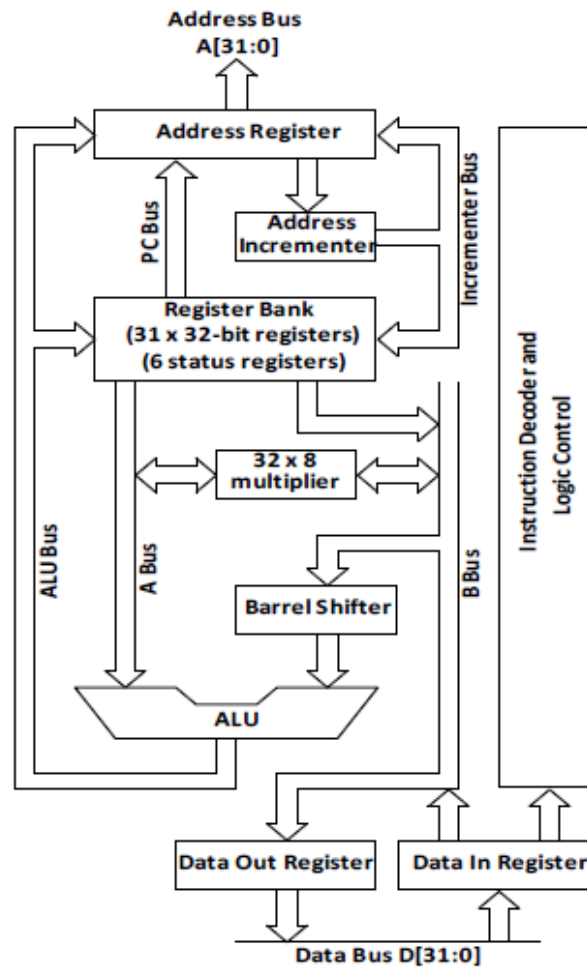
ARM Registers :
ARM has a total of 37 registers .In which - 31 are general-purpose registers of 32-bits, and six status registers .But all these registers are not seen at once. The processor state and operating mode decide which registers are available to the programmer. At any time, among the 31 general purpose registers only 16 registers are available to the user. The remaining 15 registers are used to speed up exception processing. there are two program status registers: CPSR and SPSR (the current and saved program status registers, respectively .In ARM state the registers r0 to r13 are orthogonal—any instruction that you can apply to r0 you can equally well apply to any of the other registers.

The main bank of 16 registers is used by all unprivileged code. These are the User mode registers. User mode is different from all other modes as it is unprivileged. In addition to this register bank ,there is also one 32-bit Current Program status Register(CPSR)
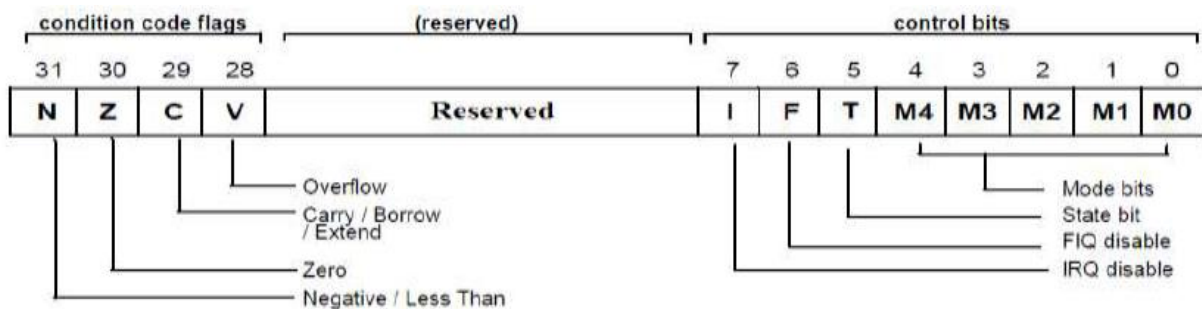
fiq :Fast interrupt request
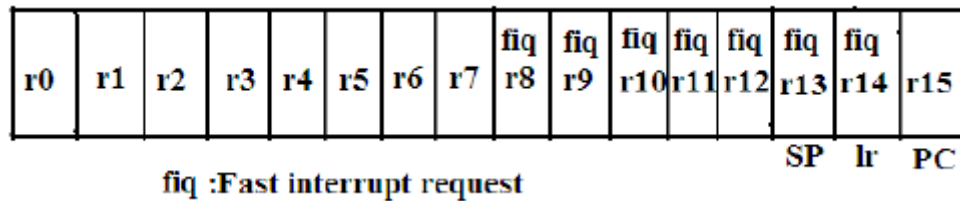
In the 15 registers ,the r13 acts as a stack pointer register and r14 acts as a link register and r15 acts as a program counter register. Register r13 is the sp register ,and it is used to store the address of the stack top. R13 is used by the PUSH and POP instructions in T variants, and by the SRS and RFE instructions from ARMv6.

Register 14 is the Link Register (LR). This register holds the address of the next instruction after a Branch and Link (BL or BLX) instruction, which is the instruction used to make a subroutine call. It is also used for return address information on entry to exception modes. At all other times, R14 can be used as a general-purpose register.

Register 15 is the Program Counter (PC). It can be used in most instructions as a pointer to the instruction which is two instructions after the instruction being executed.

The remaining 13 registers have no special hardware purpose.

CPSR : The ARM core uses the CPSR register to monitor and control internal operations. The CPSR is a dedicated 32-bit register and resides in the register file. The CPSR is divided into four fields, each of 8 bits wide : flags, status, extension, and control. The extension and status fields are reserved for future use. The control field contains the processor mode, state, and interrupt mask bits. The flags field contains the condition flags. The 32-bit CPSR register is shown below.

Processor Modes: There are seven processor modes .Six privileged modes abort, fast interrupt request, interrupt request, supervisor, system, and undefined and one non-privileged mode called user mode.

The processor enters abort mode when there is a failed attempt to access memory. Fast interrupt request and interrupt request modes correspond to the two interrupt levels available on the ARM processor. Supervisor mode is the mode that the processor is in after reset and is generally the mode that an operating system kernel operates in. System mode is a special version of user mode that allows full read-write access to the CPSR. Undefined mode is used when the processor encounters an instruction that is undefined or not supported by the implementation. User mode is used for programs and applications.

Banked Registers : Out of the 32 registers , 20 registers are hidden from a program at different times. These registers are called banked registers and are identified by the shading in the diagram. They are available only when the processor is in a particular mode; for example, abort mode has banked registers r13_abt , r14_abt and spsr _abt. Banked registers of a particular mode are denoted by an underline character post-fixed to the mode mnemonic or _mode.

When the T bit is 1, then the processor is in Thumb state. To change states the core executes a specialized branch instruction and when T= 0 the processor is in ARM state and executes ARM instructions. There are two interrupt request levels available on the ARM processor core— interrupt request (IRQ) and fast interrupt request (FIQ).

V, C , Z , N are the Condition flags .

V (oVerflow) : Set if the result causes a signed overflow

C (Carry) : Is set when the result causes an unsigned carry

Z (Zero) : This bit is set when the result after an arithmetic operation is zero, frequently used to indicate equality

N (Negative) : This bit is set when the bit 31 of the result is a binary 1.

THE ARM PROGRAMMER'S MODEL

A processor's instruction set defines the operations that the programmer can use to change the state of the system incorporating the processor. This state usually comprises the values of the data items in the processor's visible registers and the system's memory. Each instruction can be viewed as performing a defined transformation from the state before the instruction is executed to the state after it has completed. Note that although a processor will typically have many invisible registers involved in executing an instruction, the values of these registers before and after the instruction is executed are not significant; only the values in the visible registers have any significance. The visible registers in an ARM processor are shown in Figure

When writing user-level programs, only the 15 general-purpose 32-bit registers (r0 to r14), the program counter (r15) and the current program status register (CPSR) need be considered. The remaining registers are used only for system-level programming and for handling exceptions (for example, interrupts).
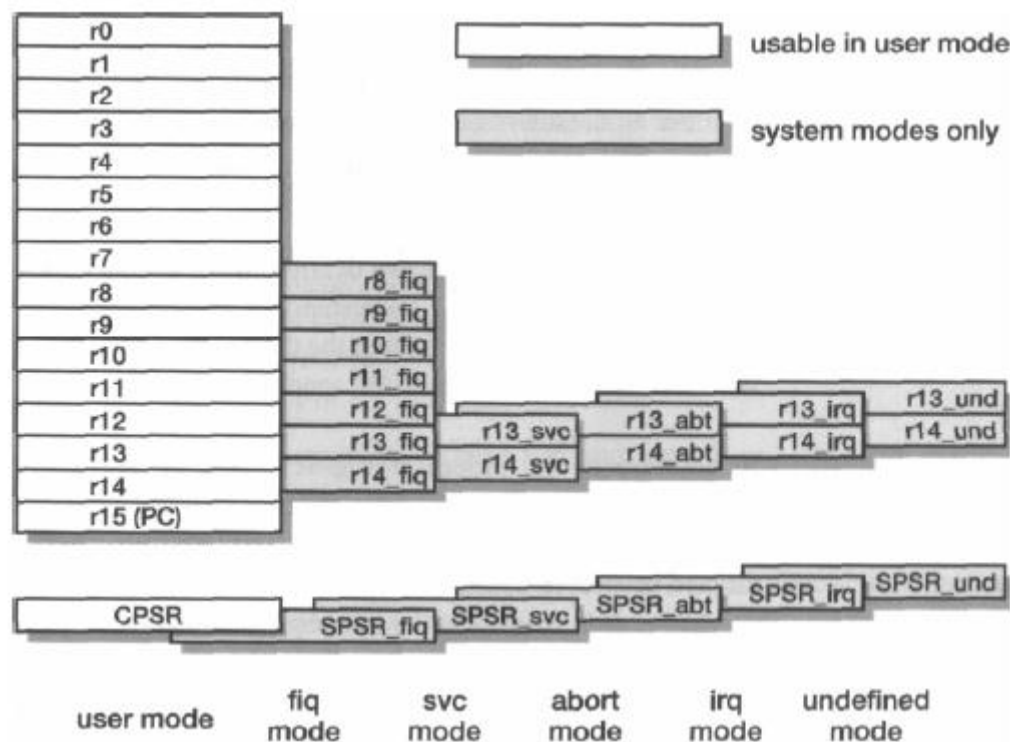
Figure : ARM's visible registers.

The Current Program Status Register (CPSR)

The CPSR is used in user-level programs to store the condition code bits. These bits are used, for example, to record the result of a comparison operation and to control whether or not a conditional branch is taken. The user-level programmer need not usually be concerned with how this register is configured, but for completeness the register is illustrated in Figure .
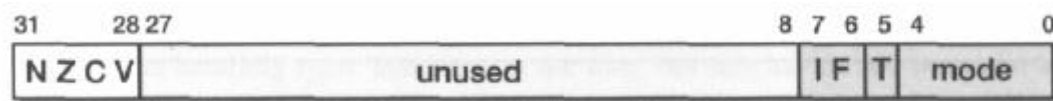

Figure: ARM CPSR format

The bits at the bottom of the register control the processor mode ,instruction set and interrupt enables ('I' and 'F') are protected from change by the user-level program. The condition code flags are in the top four bits of the register and have the following meanings:

- ❖ N: Negative; the last ALU operation which changed the flags produced a negative result (the top bit of the 32-bit result was a one).
- ❖ Z: Zero; the last ALU operation which changed the flags produced a zero result (every bit of the 32-bit result was zero).

❖ C: Carry; the last ALU operation which changed the flags generated a carry-out, either as a result of an arithmetic operation in the ALU or from the shifter.

❖ V: oVerflow; the last arithmetic ALU operation which changed the flags generated an overflow into the sign bit.

The memory system

In addition to the processor register state, an ARM system has memory state. Memory may be viewed as a linear array of bytes numbered from zero up to 2^{32}-l. Data items may be 8-bit bytes, 16-bit half-words or 32-bit words. Words are always aligned on 4-byte boundaries (that is, the two least significant address bits are zero) and half-words are aligned on even byte boundaries.

The memory organization is illustrated in Figure . This shows a small area of memory where each byte location has a unique number. A byte may occupy any of these locations, and a few examples are shown in the figure. A word-sized data item must occupy a group of four byte locations starting at a byte address which is a multiple of four, and again the figure contains a couple of examples. Half-words occupy two byte locations starting at an even byte address.
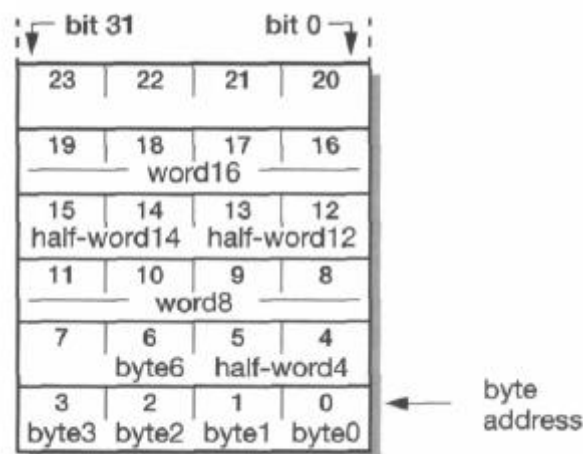
Figure: ARM memory organization

Load-store architecture

In common with most RISC processors, ARM employs a load-store architecture. This means that the instruction set will only process (add, subtract, and so on) values which are in registers (or specified directly within the instruction itself), and will always place the results of such processing into a register. The only operations which apply to memory state are

ones which copy memory values into registers(load instructions) or copy register values into memory (store instructions).

CISC processors typically allow a value from memory to be added to a value in a register, and sometimes allow a value in a register to be added to a value in memory. ARM does not support such 'memory-to-memory' operations. Therefore all ARM instructions fall into one of the following three categories:

1. Data processing instructions. These use and change only register values. For example, an instruction can add two registers and place the result in a register.

2. Data transfer instructions. These copy memory values into registers (load instructions) or copy register values into memory (store instructions). An addi tional form, useful only in systems code, exchanges a memory value with a register value.

3. Control flow instructions. Normal instruction execution uses instructions stored at consecutive memory addresses. Control flow instructions cause execution to switch to a different address, either permanently (branch instructions) or saving a return address to resume the original sequence (branch and link instructions) or trapping into system code (supervisor calls).

Supervisor mode

The ARM processor supports a protected supervisor mode. The protection mechanism ensures that user code cannot gain supervisor privileges without appropriate checks being carried out to ensure that the code is not attempting illegal operations.

The upshot of this for the user-level programmer is that system-level functions can only be accessed through specified supervisor calls. These functions generally include any accesses to hardware peripheral registers, and to widely used operations such as character input and output. User-level programmers are principally concerned with devising algorithms to operate on the data 'owned' by their programs, and rely on the operating system to handle all transactions with the world outside their programs.

The ARM instruction set

All ARM instructions are 32 bits wide (except the compressed 16-bit Thumb instructions) and are aligned on 4-byte boundaries in memory. The most notable features of the ARM instruction set are:

❖ The load-store architecture;

❖ 3-address data processing instructions (that is, the two source operand registers and the result register are all independently specified);

❖ Conditional execution of every instruction;

❖ The inclusion of very powerful load and store multiple register instructions;

❖ The ability to perform a general shift operation and a general ALU operation in a single instruction that executes in a single clock cycle;

❖ Open instruction set extension through the coprocessor instruction set, including adding new registers and data types to the programmer's model;

❖ A very dense 16-bit compressed representation of the instruction set in the Thumb architecture.

The I/O system

The ARM handles I/O (input/output) peripherals (such as disk controllers, network interfaces, and so on) as memory-mapped devices with interrupt support. The internal registers in these devices appear as addressable locations within the ARM's memory map and may be read and written using the same (load-store) instructions as any other memory locations.

Peripherals may attract the processor's attention by making an interrupt request using either the normal interrupt *(IRQ)* or the fast interrupt *(FIQ)* input. Both interrupt inputs are level-sensitive and maskable. Normally most interrupt sources share the IRQ input, with just one or two time-critical sources connected to the higher-priority FIQ input. Some systems may include direct memory access (DMA) hardware external to the processor to handle high-bandwidth I/O traffic. Interrupts are a form *of exception* and are handled as outlined below.

ARM exceptions

The ARM architecture supports a range of interrupts, traps and supervisor calls, all grouped under the general heading of exceptions. The general way these are handled is the same in all cases:

+ The current state is saved by copying the PC into *rl4_exc* and the CPSR into SPSR_exc (where *exc* stands for the exception type).

+ The processor operating mode is changed to the appropriate exception mode.

🔸 The PC is forced to a value between 0016 and 1C16, the particular value depending on the type of exception.

The instruction at the location the PC is forced to (the *vector address)* will usually contain a branch to the exception handler. The exception handler will use rl3_exc, which will normally have been initialized to point to a dedicated stack in memory, to save some user registers for use as work registers.

The return to the user program is achieved by restoring the user registers and then using an instruction to restore the PC and the CPSR atomically. This may involve some adjustment of the PC value saved in *rl4_exc* to compensate for the state of the pipeline when the exception arose.

SYSTEM SOFTWARE

❖ The ARM C compiler

The ARM C compiler is compliant with the ANSI (American National Standards Institute) standard for C and is supported by the appropriate library of standard functions. It uses the ARM Procedure Call Standard for all externally available functions. It can be told to produce assembly source output instead of ARM object format, so the code can be inspected, or even hand optimized, and then assembled subsequently. The compiler can also produce Thumb code.
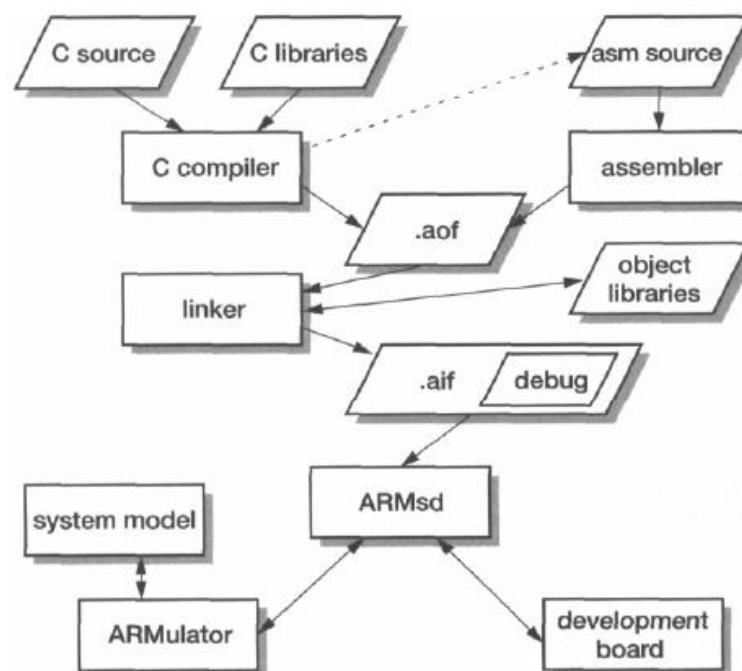

Figure : The structure of the ARM cross-development toolkit

❖ The ARM assembler

The ARM assembler is a full macro assembler which produces ARM object format output that can be linked with output from the C compiler. Assembly source language is near machine-level, with most assembly instructions translating into single ARM (or Thumb) instructions.

❖ The linker

The linker takes one or more object files and combines them into an executable program.

It resolves symbolic references between the object files and extracts object modules from libraries as needed by the program. It can assemble the various components of the program in a number of different ways, depending on whether the code is to run in RAM (Random Access Memory, which can be read and written) or ROM (Read Only Memory), whether overlays are required, and so on.

Normally the linker includes debug tables in the output file. If the object files were compiled with full debug information, this will include full symbolic debug tables (so the program can be debugged using the variable names in the source program). The linker can also produce object library modules that are not executable but are ready for efficient linking with object files in the future.

❖ ARMsd

The ARM symbolic debugger is a front-end interface to assist in debugging programs running either under emulation (on the ARMulator) or remotely on a target system such as the ARM development board. The remote system must support the appropriate remote debug protocols either via a serial line or through a JTAG test interface. Debugging a system where the processor core is embedded within an application-specific system chip is a complex issue.

At its most basic, ARMsd allows an executable program to be loaded into the ARMulator or a development board and run. It allows the setting of breakpoints, which are addresses in the code that, if executed, cause execution to halt so that the processor state can be examined. In the ARMulator, or when running on hardware with appropriate support, it also allows the setting of watchpoints. These are memory addresses that, if accessed as data addresses, cause execution to halt in a similar way.

At a more sophisticated level ARMsd supports full source level debugging, allowing the C programmer to debug a program using the source file to specify breakpoints and using variable names from the original program.

❖ **ARMulator**

The ARMulator *(ARM emulator)* is a suite of programs that models the behaviour of various ARM processor cores in software on a host system. It can operate at various levels of accuracy:

- *Instruction-accurate* modelling gives the exact behaviour of the system state without regard to the precise timing characteristics of the processor.
- *Cycle-accurate* modelling gives the exact behaviour of the processor on a cycleby-cycle basis, allowing the exact number of clock cycles that a program requires to be established.
- *Timing-accurate* modelling presents signals at the correct time within a cycle, allowing logic delays to be accounted for.

All these approaches run considerably slower than the real hardware, but the first incurs the smallest speed penalty and is best suited to software development.

At its simplest, the ARMulator allows an ARM program developed using the C compiler or assembler to be tested and debugged on a host machine with no ARM processor connected. It allows the number of clock cycles the program takes to execute to be measured exactly, so the performance of the target system can be evaluated. At its most complex, the ARMulator can be used as the centre of a complete, timing-accurate, C model of the target system, with full details of the cache and memory management functions added, running an operating system. In between these two extremes the ARMulator comes with a set of model prototyping modules including a rapid prototype memory model and coprocessor interfacing support. The ARMulator can also be used as the core of a timing-accurate ARM behavioural model in a hardware simulation environment based around a language such as VHDL. (VHDL is a standard, widely supported hardware description language.) A VHDL 'wrapper' must be generated to interface the ARMulator C code to the VHDL environment.
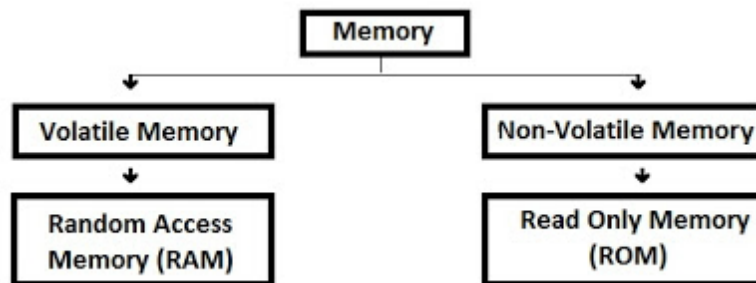
MODULE V
THE MEMORY SYSTEM

Memory is an essential element of a computer. Without its memory, a computer is of hardly any use. Memory plays an important role in saving and retrieving data. The performance of the computer system depends upon the size of the memory. Memory is of following types:

1. **Primary Memory / Volatile Memory.**
2. **Secondary Memory / Non Volatile Memory.**

1. Primary Memory / Volatile Memory: Primary Memory is internal memory of the computer. RAM AND ROM both form part of primary memory. The primary memory provides main working space to the computer.The following terms comes under primary memory of a computer are discussed below:

❖ **Random Access Memory (RAM):** The primary storage is referred to as random access memory (RAM) because it is possible to randomly select and use any location of the memory directly store and retrieve data. It takes same time to any address of the memory as the first address. It is also called read/write memory. The storage of data and instructions inside the primary storage is temporary. It disappears from RAM as soon as the power to the computer is switched off. The memories, which lose their content on failure of power supply, are known as volatile memories .So now we can say that RAM is volatile memory.

              RAM is of two types

                    1. Static RAM (SRAM)
                    2. Dynamic RAM (DRAM)

Static RAM (SRAM)

The word **static** indicates that the memory retains its contents as long as power remains applied. However, data is lost when the power gets down due to volatile nature. SRAM chips use a matrix of 6-transistors and no capacitors. Transistors do not require power to prevent leakage, so SRAM need not have to be refreshed on a regular basis. Because of the extra space in the matrix, SRAM uses more chips than DRAM for the same amount of storage space, thus making the manufacturing costs higher. Static RAM is used as cache memory needs to be very fast and small.

Dynamic RAM (DRAM)

DRAM, unlike SRAM, must be continually **refreshed** in order for it to maintain the data. This is done by placing the memory on a refresh circuit that rewrites the data several hundred times per second. DRAM is used for most system memory because it is cheap and small. All DRAMs are made up of memory cells. These cells are composed of one capacitor and one transistor.

❖ **Read Only Memory (ROM):** There is another memory in computer, which is called Read Only Memory (ROM). Again it is the ICs inside the PC that form the ROM. The storage of program and data in the ROM is permanent. The ROM stores some standard processing programs supplied by the manufacturers to operate the personal computer. The ROM can only be read by the CPU but it cannot be changed. The basic input/output program is stored in the ROM that examines and initializes various equipment attached to the PC when the power switch is ON. The memories, which do not lose their content on failure of power supply, are known as non-volatile memories. ROM is non-volatile memory.

➕ **PROM:** There is another type of primary memory in computer, which is called Programmable Read Only Memory (PROM). You know that it is not possible to modify or erase programs stored in ROM, but it is possible for you to store your program in PROM chip. Once the programmers' are written it cannot be changed and remain intact even if power is switched off. Therefore programs or instructions written in PROM or ROM cannot be erased or changed.

➕ **EPROM:** This stands for Erasable Programmable Read Only Memory, which overcome the problem of PROM & ROM. EPROM chip can be programmed time and again by erasing the information stored earlier in it. Information stored in EPROM exposing the chip for some

time ultraviolet light and it erases chip is reprogrammed using a special programming facility. When the EPROM is in use information can only be read.

➕ **Cache Memory:** The speed of CPU is extremely high compared to the access time of main memory. Therefore the performance of CPU decreases due to the slow speed of main memory. To decrease the mismatch in operating speed, a small memory chip is attached between CPU and Main memory whose access time is very close to the processing speed of CPU. It is called CACHE memory. CACHE memories are accessed much faster than conventional RAM. It is used to store programs or data currently being executed or temporary data frequently used by the CPU. So each memory makes main memory to be faster and larger than it really is. It is also very expensive to have bigger size of cache memory and its size is normally kept small.

➕ **Registers:** The CPU processes data and instructions with high speed; there is also movement of data between various units of computer. It is necessary to transfer the processed data with high speed. So the computer uses a number of special memory units called registers. They are not part of the main memory but they store data or information temporarily and pass it on as directed by the control unit.

2. Secondary Memory / Non-Volatile Memory:

Secondary memory is external and permanent in nature. The secondary memory is concerned with magnetic memory. Secondary memory can be stored on storage media like floppy disks, magnetic disks, magnetic tapes, This memory can also be stored optically on Optical disks - CD-ROM. The following terms comes under secondary memory of a computer are discussed below:

➕ **Magnetic Tape:** Magnetic tapes are used for large computers like mainframe computers where large volume of data is stored for a longer time. In PC also you can use tapes in the form of cassettes. The cost of storing data in tapes is inexpensive. Tapes consist of magnetic materials that store data permanently. It can be 12.5 mm to 25 mm wide plastic film-type and 500 meter to 1200 meter long which is coated with magnetic material. The deck is connected to the central processor and information is fed into or read from the tape through the processor. It's similar to cassette tape recorder.

- **Magnetic Disk:** You might have seen the gramophone record, which is circular like a disk and coated with magnetic material. Magnetic disks used in computer are made on the same principle. It rotates with very high speed inside the computer drive. Data is stored on both the surface of the disk. Magnetic disks are most popular for direct access storage device. Each disk consists of a number of invisible concentric circles called tracks. Information is recorded on tracks of a disk surface in the form of tiny magnetic spots. The presence of a magnetic spot represents one bit and its absence represents zero bit. The information stored in a disk can be read many times without affecting the stored data. So the reading operation is non-destructive. But if you want to write a new data, then the existing data is erased from the disk and new data is recorded. For Example-Floppy Disk.

- **Optical Disk:** With every new application and software there is greater demand for memory capacity. It is the necessity to store large volume of data that has led to the development of optical disk storage medium. Optical disks can be divided into the following categories:

  a) **Compact Disk/ Read Only Memory (CD-ROM**
  b) **Write Once, Read Many (WORM)**
 c) **Erasable Optical Disk**

Comparison Table RAM & ROM

Basis for Comparison	RAM	ROM
Stands for	Random Access Memory	Read Only Memory
Memory type	Volatile	Non-volatile
Memory capacity	1 to 256 GB per chip	4 to 8 MB per chip
Operation type	Read and Write both.	Only Read.
Speed	Fast	Comparatively slow.
Storage type	Temporary	Permanent
Also referred as	Primary memory	Secondary memory

Basis for Comparison	RAM	ROM
Presence of data according to power source	The stored data in RAM lost in case of power failure.	Data retained in ROM even if the power is turned off.
Accessibility to processor	Processor can directly access the data in RAM.	Processor cannot directly access the data in ROM.
Cost	High	Comparatively low
Types	SRAM and DRAM	PROM, EPROM and EEPROM

Comparison Table SRAM & DRAM

BASIS FOR COMPARISON	SRAM	DRAM
Speed	Faster	Slower
Size	Small	Large
Cost	Expensive	Cheap
Used in	Cache memory	Main memory
Density	Less dense	Highly dense
Construction	Complex and uses transistors and latches.	Simple and uses capacitors and very few transistors.
Single block of memory requires	6 transistors	Only one transistor.
Charge leakage property	Not present	Present hence require power refresh circuitry
Power consumption	Low	High

MEMORY HIERARCHY
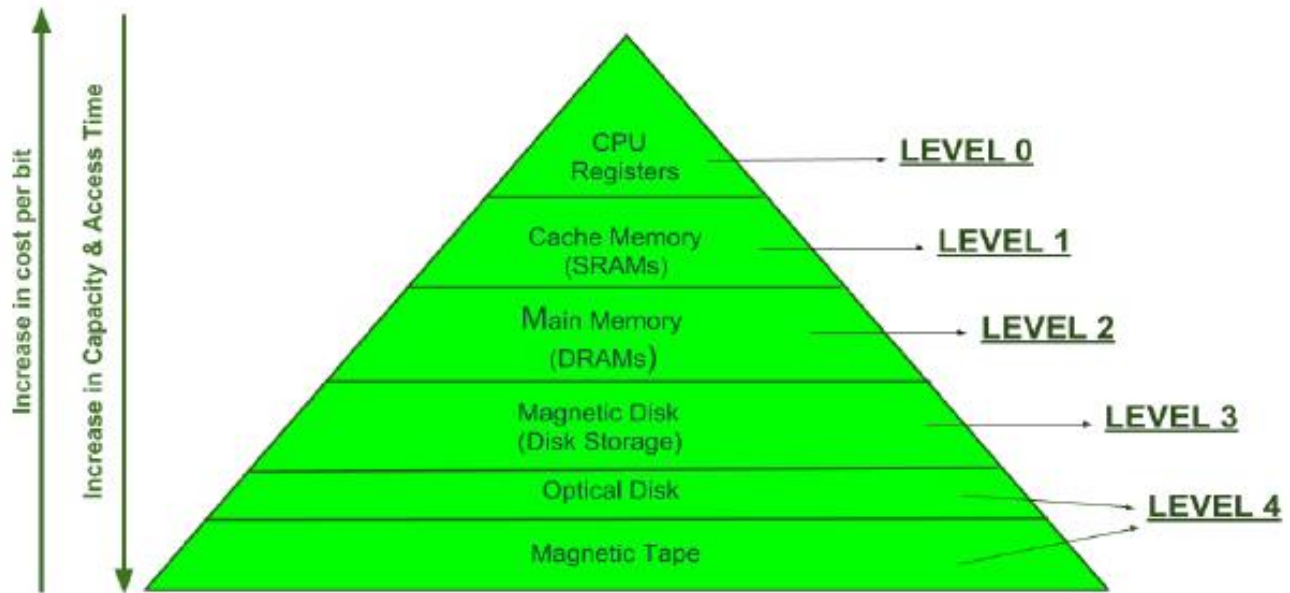
Characteristics of Memory Hierarchy are following when we go from top to bottom.

- Capacity in terms of storage increases.
- Cost per bit of storage decreases.
- Frequency of access of the memory by the CPU decreases.
- Access time by the CPU increases

We can infer the following characteristics of Memory Hierarchy Design from above figure:

1. **Capacity:** It is the global volume of information the memory can store. As we move from top to bottom in the Hierarchy, the capacity increases.

2. **Access Time:**

   It is the time interval between the read/write request and the availability of the data. As we move from top to bottom in the Hierarchy, the access time increases.

3. **Performance:**

   Earlier when the computer system was designed without Memory Hierarchy design, the speed gap increases between the CPU registers and Main Memory due to large difference in access time. This results in lower performance of the system and thus, enhancement was required. This enhancement was made in the form of Memory Hierarchy Design because of which the performance of the system increases. One of the most significant ways to increase system performance is minimizing how far down the memory hierarchy one has to go to manipulate data.

4. **Cost per bit:**

   As we move from bottom to top in the Hierarchy, the cost per bit increases i.e. Internal Memory is costlier than External Memory.

CACHE MEMORY

Cache Memory is a special very high-speed memory. It is used to speed up and synchronizing with high-speed CPU. Cache memory is costlier than main memory or disk memory but economical than CPU registers. Cache memory is an extremely fast memory type that acts as a buffer between RAM and the CPU. It holds frequently requested data and instructions so that they are immediately available to the CPU when needed.

Cache memory is used to reduce the average time to access data from the Main memory. The cache is a smaller and faster memory which stores copies of the data from frequently used main memory locations. There are various different independent caches in a CPU, which store instructions and data.
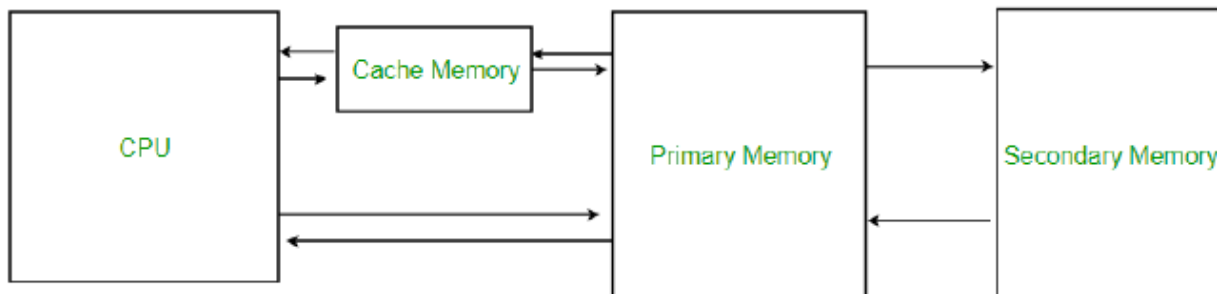
Levels of memory:

- **Level 1 or Register –** It is a type of memory in which data is stored and accepted that are immediately stored in CPU. Most commonly used register is accumulator, Program counter, address register etc.

- **Level 2 or Cache memory –** It is the fastest memory which has faster access time where data is temporarily stored for faster access.

- **Level 3 or Main Memory –** It is memory on which computer works currently. It is small in size and once power is off data no longer stays in this memory.

- **Level 4 or Secondary Memory –** It is external memory which is not as fast as main memory but data stays permanently in this memory.

Cache Performance:

When the processor needs to read or write a location in main memory, it first checks for a corresponding entry in the cache.

- If the processor finds that the memory location is in the cache, a **cache hit** has occurred and data is read from cache

- If the processor **does not** find the memory location in the cache, a **cache miss** has occurred. For a cache miss, the cache allocates a new entry and copies in data from main memory, then the request is fulfilled from the contents of the cache.

Cache Performance Improvement

The performance of cache memory is frequently measured in terms of a quantity called **Hit ratio.**

Hit ratio = hit / (hit + miss) = no. of hits/total accesses

We can improve Cache performance using higher cache block size, higher associativity, reduce miss rate, reduce miss penalty, and reduce the time to hit in the cache.

Cache Mapping:

There are three different types of mapping used for the purpose of cache memory which are as follows: Direct mapping, Associative mapping, and Set-Associative mapping. These are explained below.

1. **Direct Mapping –** The simplest technique, known as direct mapping, maps each block of main memory into only one possible cache line. or In Direct mapping, assigne each

---

memory block to a specific line in the cache. If a line is previously taken up by a memory block when a new block needs to be loaded, the old block is trashed. An address space is split into two parts index field and a tag field. The cache is used to store the tag field whereas the rest is stored in the main memory. Direct mapping`s performance is directly proportional to the Hit ratio.

i = j modulo m
where
i=cache line number

 j= main memory block number

 m=number of lines in the cache

For purposes of cache access, each main memory address can be viewed as consisting of three fields. The least significant w bits identify a unique word or byte within a block of main memory. In most contemporary machines, the address is at the byte level. The remaining s bits specify one of the 2^S blocks of main memory. The cache logic interprets these s bits as a tag of s-r bits (most significant portion) and a line field of r bits. This latter field identifies one of the $m=2^r$ lines of the cache.
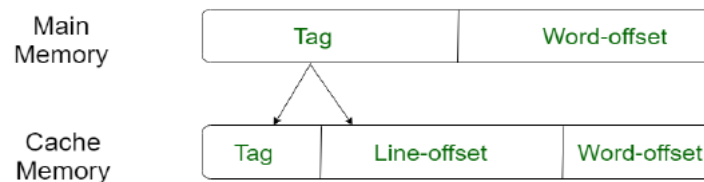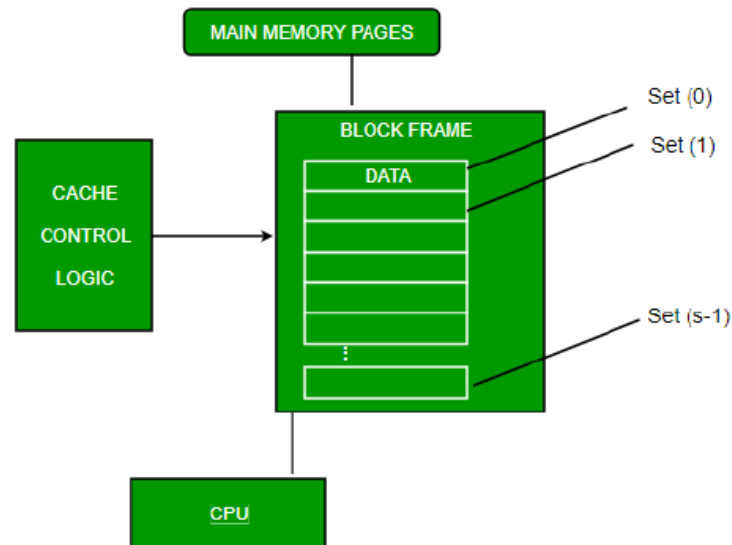
2. **Associative Mapping –** In this type of mapping, the associative memory is used to store content and addresses of the memory word. Any block can go into any line of the cache. This means that the word id bits are used to identify which word in the block is needed, but the tag becomes all of the remaining bits. This enables the placement of any word at any place in the cache memory. It is considered to be the fastest and the most flexible mapping form.
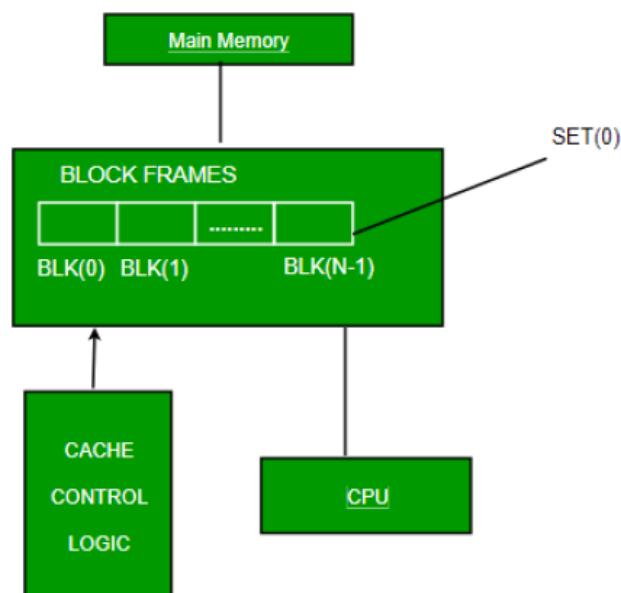
3. **Set-associative Mapping –** This form of mapping is an enhanced form of direct mapping where the drawbacks of direct mapping are removed. Set associative addresses the problem of possible thrashing in the direct mapping method. It does this by saying that instead of having exactly one line that a block can map to in the cache, we will group a few lines together creating a *set*. Then a block in memory can map to any one of the lines of a specific set..Set-associative mapping allows that each word that is present in the cache can have two or more words in the main memory for the same index address. Set associative cache mapping combines the best of direct and associative cache mapping techniques.

In this case, the cache consists of a number of sets, each of which consists of a number of lines. The relationships are
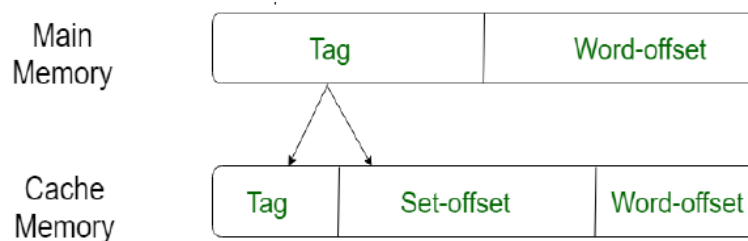
m = v * k

i= j mod v

where

i=cache set number
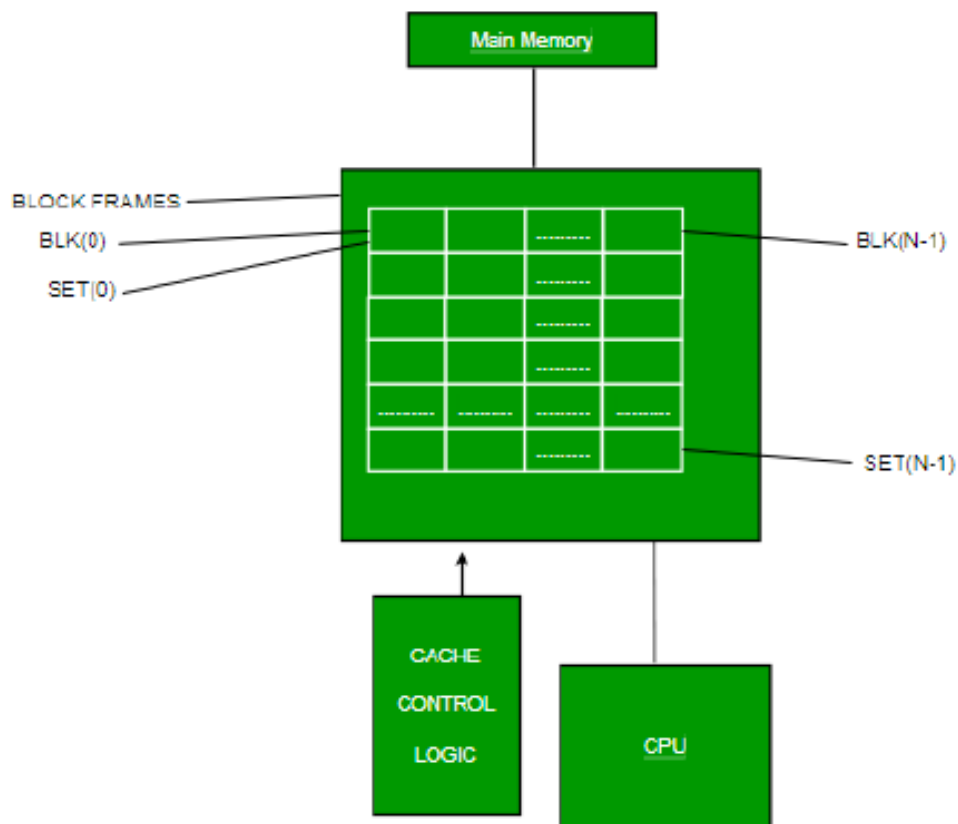
j=main memory block number

v=number of sets

m=number of lines in the cache number of sets

k=number of lines in each set

Application of Cache Memory

- ❖ Usually, the cache memory can store a reasonable number of blocks at any given time, but this number is small compared to the total number of blocks in the main memory.
- ❖ The correspondence between the main memory blocks and those in the cache is specified by a mapping function.

Types of Cache

- ❖ **Primary Cache –** A primary cache is always located on the processor chip. This cache is small and its access time is comparable to that of processor registers.

❖ **Secondary Cache –** Secondary cache is placed between the primary cache and the rest of the memory. It is referred to as the level 2 (L2) cache. Often, the Level 2 cache is also housed on the processor chip.

LOCALITY OF REFERENCE

Since size of cache memory is less as compared to main memory. So to check which part of main memory should be given priority and loaded in cache is decided based on locality of reference.

Types of Locality of reference

❖ **Spatial Locality of reference** This says that there is a chance that element will be present in the close proximity to the reference point and next time if again searched then more close proximity to the point of reference.

❖ **Temporal Locality of reference** In this Least recently used algorithm will be used. Whenever there is page fault occurs within a word will not only load word in main memory but complete page fault will be loaded because spatial locality of reference rule says that if you are referring any word next word will be referred in its register that's why we load complete page table so the complete block will be loaded.

VIRTUAL MEMORY

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard that's set up to emulate the computer's RAM.

The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory serves two purposes. First, it allows us to extend the use of physical memory by using disk. Second, it allows us to have memory protection, because each virtual address is translated to a physical address.

Following are the situations, when entire program is not required to be loaded fully in main memory.

❖ User written error handling routines are used only when an error occured in the data or computation.

- ❖ Certain options and features of a program may be used rarely.

- ❖ Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.

- ❖ The ability to execute a program that is only partially in memory would counter many benefits.

- ❖ Less number of I/O would be needed to load or swap each user program into memory.

- ❖ A program would no longer be constrained by the amount of physical memory that is available.

- ❖ Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.

Modern microprocessors intended for general-purpose use, a memory management unit, or MMU, is built into the hardware. The MMU's job is to translate virtual addresses into physical addresses. A basic example is given below:

Secondary Memory
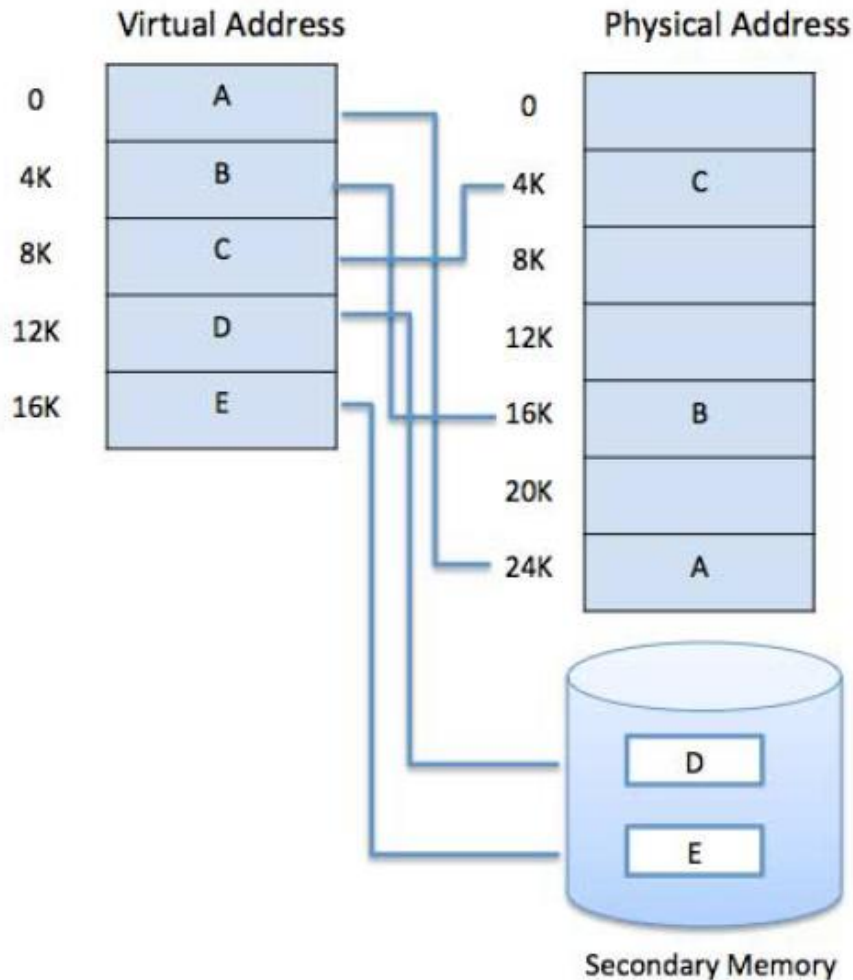
Paging

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard that's set up to emulate the computer's RAM. Paging technique plays an important role in implementing virtual memory.

Paging is a memory management technique in which process address space is broken into blocks of the same size called **pages** (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages.

Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called **frames** and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.
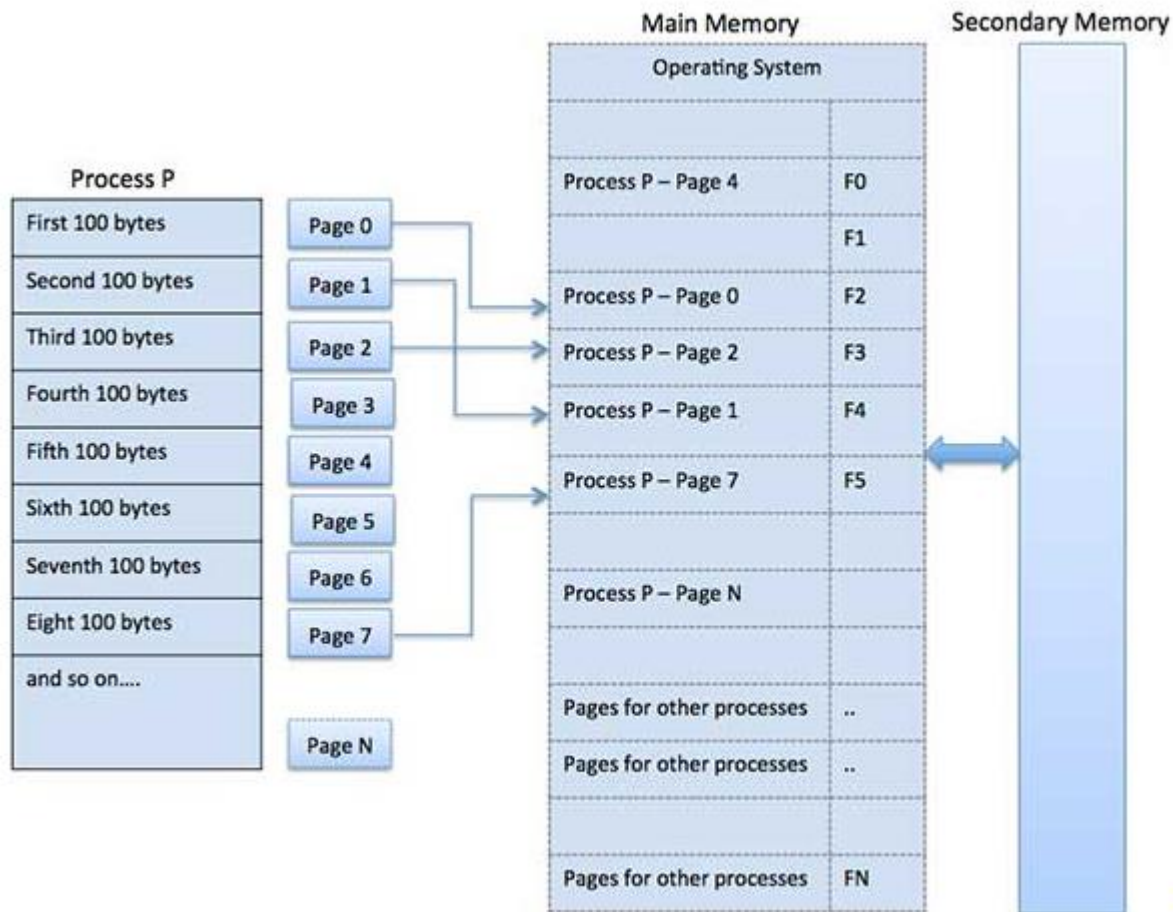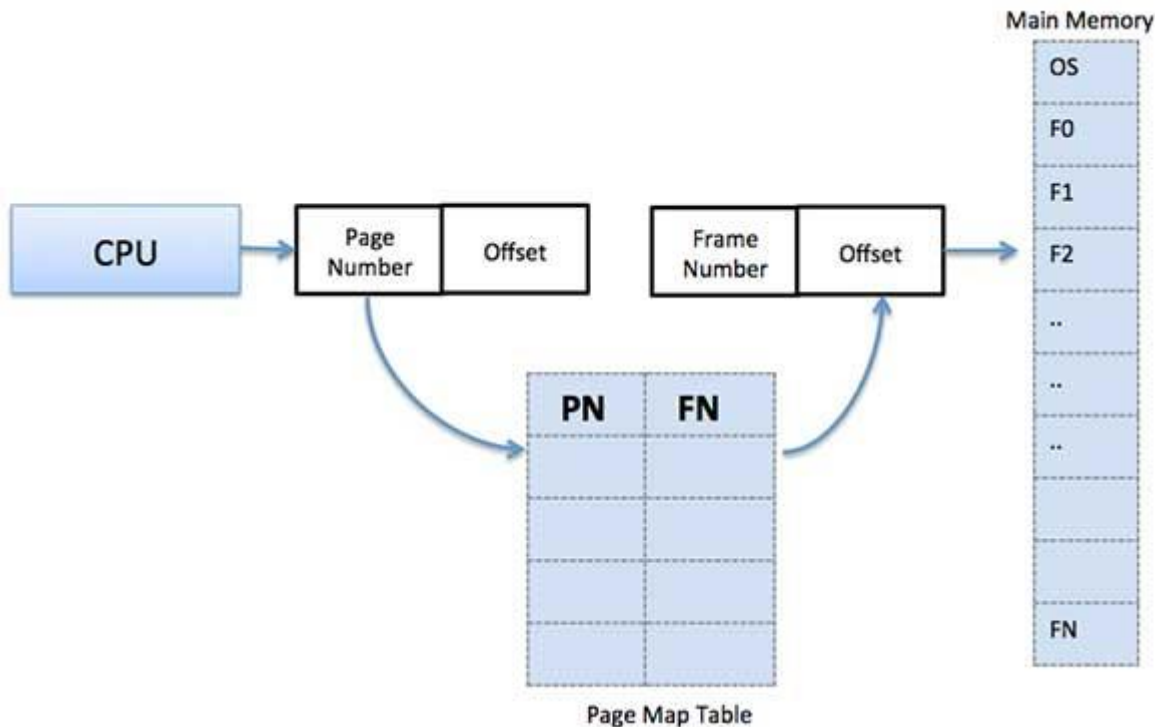

ADDRESS TRANSLATION

Page address is called **logical address** and represented by **page number** and the **offset**.

Logical Address = Page number + page offset

Frame address is called **physical address** and represented by a **frame number** and the **offset**.

Physical Address = Frame number + page offset

A data structure called **page map table** is used to keep track of the relation between a page of a process to a frame in physical memory.

Main Memory

Page Map Table

When the system allocates a frame to any page, it translates this logical address into a physical address and create entry into the page table to be used throughout execution of the program.

When a process is to be executed, its corresponding pages are loaded into any available memory frames. Suppose you have a program of 8Kb but your memory can accommodate only 5Kb at a given point in time, then the paging concept will come into picture. When a computer runs out of RAM, the operating system (OS) will move idle or unwanted pages of memory to secondary memory to free up RAM for other processes and brings them back when needed by the program.

This process continues during the whole execution of the program where the OS keeps removing idle pages from the main memory and write them onto the secondary memory and bring them back when required by the program.

Advantages and Disadvantages of Paging

Here is a list of advantages and disadvantages of paging

❖ Paging reduces external fragmentation, but still suffer from internal fragmentation.

❖ Paging is simple to implement and assumed as an efficient memory management technique.

❖ Due to equal size of the pages and frames, swapping becomes very easy.

❖ Page table requires extra memory space, so may not be good for a system having small RAM.

Segmentation

Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions. Each segment is actually a different logical address space of the program.

When a process is to be executed, its corresponding segmentation are loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory.

Segmentation memory management works very similar to paging but here segments are of variable-length where as in paging pages are of fixed size.

A program segment contains the program's main function, utility functions, data structures, and so on. The operating system maintains a **segment map table** for every process and a list of free memory blocks along with segment numbers, their size and corresponding memory locations in main memory. For each segment, the table stores the starting address of the segment and the length of the segment. A reference to a memory location includes a value that identifies a segment and an offset.

SN	Size	Memory Address
1	400	100
2	200	500
3	100	800
N	x	NM
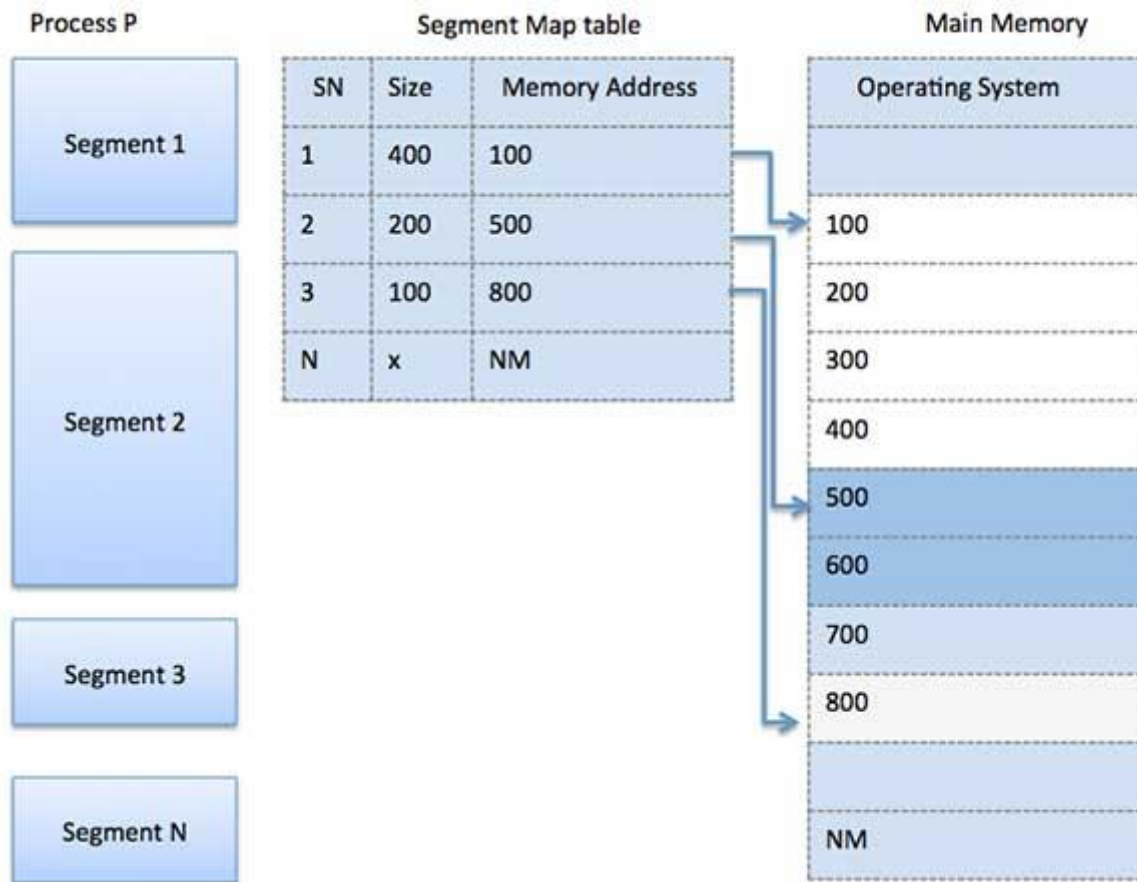
INPUT/OUTPUT ORGANIZATION

The processor bus is the bus defied by the signals on the processor chip itself. Devices that require a very high-speed connection to the processor, such as the main memory, may be connected directly to this bus. For electrical reasons, only a few devices can be connected in this manner. The motherboard usually provides another bus that can support more devices. The two buses are interconnected by a circuit, which we will call a bridge, that translates the signals and protocols of one bus into those of the other. Devices connected to the expansion bus appear to the processor as if they were connected directly to the processor's own bus. The only difference is that the bridge circuit introduces a small delay in data transfers between the processor and those devices.

It is not possible to define a uniform standard for the processor bus. The structure of this bus is closely tied to the architecture of the processor. It is also dependent on the electrical characteristics of the processor chip, such as its clock speed. The expansion bus is not subject to these limitations, and therefore it can use a standardized signaling scheme. A number of standards have been developed. Some have evolved by default, when a particular design became commercially successful. For example, IBM developed a bus they called ISA (Industry Standard Architecture) for their personal computer known at the time as PC AT.

Some standards have been developed through industrial cooperative efforts, even among competing companies driven by their common self-interest in having compatible products. In some cases, organizations such as the IEEE (Institute of Electrical and Electronics Engineers), ANSI (American National Standards Institute), or international bodies such as ISO (International Standards Organization) have blessed these standards and given them an official status.

A given computer may use more than one bus standards. A typical Pentium computer has both a PCI bus and an ISA bus, thus providing the user with a wide range of devices to choose from.
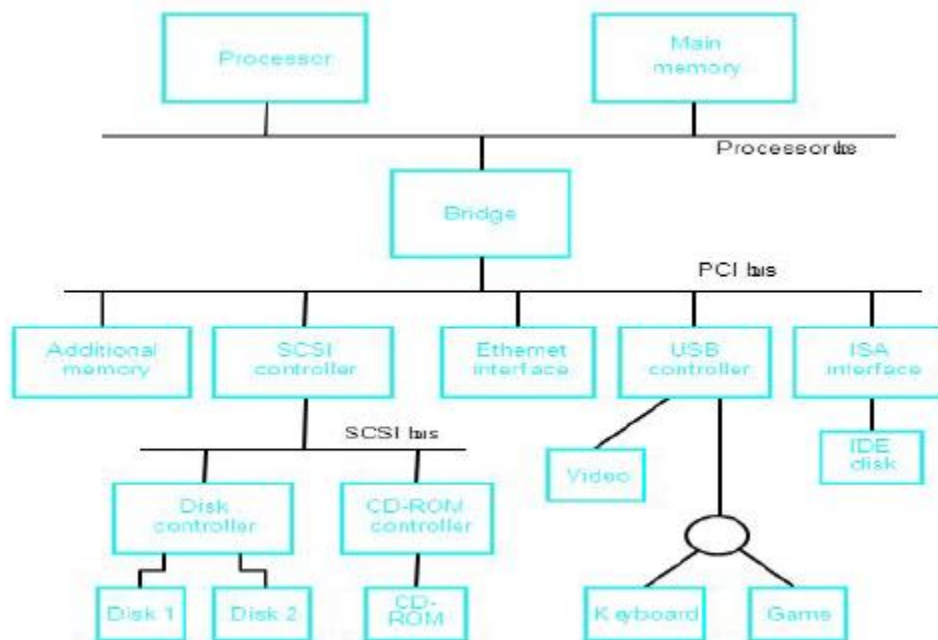
Peripheral Component Interconnect (PCI) Bus:-

The PCI bus is a good example of a system bus that grew out of the need for standardization. It supports the functions found on a processor bus bit in a standardized format that is independent of any particular processor. Devices connected to the PCI bus appear to the processor as if they were connected directly to the processor bus. They are assigned addresses in the memory address space of the processor.

The PCI follows a sequence of bus standards that were used primarily in IBM PCs. Early PCs used the 8-bit XT bus, whose signals closely mimicked those of Intel's 80x86 processors. Later, the 16-bit bus used on the PC At computers became known as the ISA bus. Its extended 32-bit version is known as the EISA bus. Other buses developed in the eighties with similar capabilities are the Microchannel used in IBM PCs and the NuBus used in Macintosh computers.

The PCI was developed as a low-cost bus that is truly processor independent. Its design anticipated a rapidly growing demand for bus bandwidth to support high-speed disks and graphic and video devices, as well as the specialized needs of multiprocessor systems. As a result, the PCI is still popular as an industry standard almost a decade after it was first introduced in 1992.

An important feature that the PCI pioneered is a plug-and-play capability for connecting I/O devices. To connect a new device, the user simply connects the device interface board to the bus. The software takes care of the rest.

DATA TRANSFER

Programmed I/O

Programmed I/O (PIO) refers to data transfers initiated by a CPU under driver software

control to access registers or memory on a device. The CPU issues a command then waits for I/O operations to be complete. As the CPU is faster than the I/O module, the problem with programmed I/O is that the CPU has to wait a long time for the I/O module of concern to be ready

for either transmission or reception of data. The CPU, while waiting, must repeatedly check the status of the I/O module, and this process is known as Polling. As a result, the level of the performance of the entire system is severely degraded.

Programmed I/O basically works in these ways:

> ❖ CPU requests I/O operation
>
> ❖ I/O module performs operation
>
> ❖ I/O module sets status bits
>
> ❖ CPU checks status bits periodically
>
> ❖ I/O module does not inform CPU directly
>
> ❖ I/O module does not interrupt CPU
>
> ❖ CPU may wait or come back later

Interrupt Driven I/O

The CPU issues commands to the I/O module then proceeds with its normal work until interrupted by I/O device on completion of its work. For input, the device interrupts the CPU when new data has arrived and is ready to be retrieved by the system processor. The actual actions to perform depend on whether the device uses I/O ports, memory mapping.

For output, the device delivers an interrupt either when it is ready to accept new data or to acknowledge a successful data transfer. Memory-mapped and DMA-capable devices usually generate interrupts to tell the system they are done with the buffer. Although Interrupt relieves the CPU of having to wait for the devices, but it is still inefficient in data transfer of large amount because the CPU has to transfer the data word by word between I/O module and memory.

Below are the basic operations of Interrupt:

> ❖ CPU issues read command
>
> ❖ I/O module gets data from peripheral whilst CPU does other work
>
> ❖ I/O module interrupts CPU
>
> ❖ CPU requests data
>
> ❖ I/O module transfers data

ASYNCHRONOUS TRANSMISSION

In contrast, asynchronous transmission works in spurts and must insert a start bit before each data character and a stop bit at its termination to inform the receiver where it begins and ends. The term *asynchronous* is used to describe the process where transmitted data is encoded with start and stop bits, specifying the beginning and end of each character.

An example of asynchronous transmission is shown in the following figure.

Start-bit									Stop-bit
0	1	2	3	4	5	6	7	8	1

These additional bits provide the timing or synchronization for the connection by indicating when a complete character has been sent or received; thus, timing for each character begins with the start bit and ends with the stop bit.

When gaps appear between character transmissions, the asynchronous line is said to be in a mark state. A mark is a binary 1 (or negative voltage) that is sent during periods of inactivity on the line as shown in the following figure
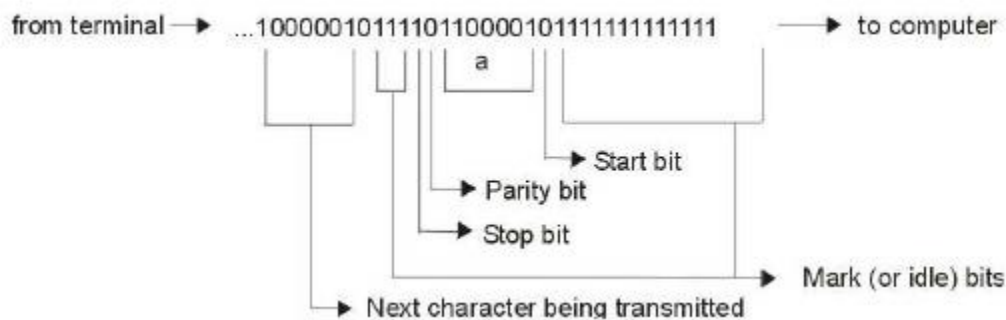
When the mark state is interrupted by a positive voltage (a binary 0), the receiving system knows that data characters are going to follow. It is for this reason that the start bit, which precedes the data character bit, which signals the end of a character, is always a mark bit (binary 1).

The following is a list of characteristics specific to asynchronous communication:

   ❖ Each character is preceded by a start bit a

   ❖ Gaps or spaces between characters may exist.

With asynchronous transmission, a large text document is organized into long strings of letters (or characters) that make up the words within the sentences and paragraphs. These characters are sent over the communication link one at a time and reassembled at the remote location.

In asynchronous transmission, ASCII character would actually be transmitted using 10 bits. For example, "0100 0001" would become "**1** 0100 0001 **0**". The extra one (or zero, depending on parity bit) at the start and end of the transmission tells the receiver first that a character is coming and secondly that the character has ended. This method of transmission is used when data are sent intermittently as opposed to in a solid stream. In the previous example the start and stop bits are in bold.

The start and stop bits must be of opposite polarity. This allows the receiver to recognize when the second packet of information is being sent.

Asynchronous transmission is used commonly for communications over telephone lines.

SYNCHRONOUS TRANSMISSION

The term *synchronous* is used to describe a continuous and timed bound /clock based transfer of data blocks.

- ❖ It is a data transfer method in which a continuous stream of data signals is accompanied by timing signals (generated by an electronic clock) to ensure that the transmitter and the receiver are in step (synchronized) with one another.
- ❖ The data is sent in blocks (called frames or packets) spaced by fixed time intervals
- ❖ Synchronous transmission modes are used when large amounts of data must be transferred very quickly from one location to the other.
- ❖ Synchronous transmission synchronizes transmission speeds at both the receiving and sending end of the transmission by using clock signals.
- ❖ A continual stream of data is then sent between the two nodes.

The data blocks are grouped and spaced in regular intervals and are preceded by special characters called synchronous idle characters. See the following illustration

After the syn characters are received by the remote device, they are decoded and used to synchronize the connection. After the connection is correctly synchronized, data

transmission may begin. An analogy of synchronous transmission would be the transmission of a large text document. Before the document is transferred across the synchronous line, it is first broken into blocks of sentences or paragraphs. The blocks are then sent over the communication link to the remote site.

The timing needed for synchronous connections is obtained from the devices located on the communication link. All devices on the synchronous link must be set to the same clocking.

The following is a list of characteristics specific to synchronous communication:

- ❖ There are no gaps between characters being transmitted.
- ❖ Timing is supplied by modems or other devices at each end of the connection.
- ❖ Special syn characters precede the data being transmitted.
- ❖ The syn characters are used between blocks of data for timing purposes

Due to there being no start and stop bits the data transfer rate is quicker although more errors will occur, as the clocks will eventually get out of sync, and the receiving device would have the wrong time that had been agreed in protocol for sending /receiving data, so some bytes could become corrupted (by losing bits).

Ways to get around this problem include re-check digits to ensure the bytes is correctly interpreted a protocols (such as Ethernet, SONET, and Token Ring) use synchronous transmission.

Direct Memory Access:

The data transfer between a fast storage media such as magnetic disk and memory unit is limited by the speed of the CPU. Thus we can allow the peripherals directly communicate with each other using the memory buses, removing the intervention of the CPU. This type of data transfer technique is known as DMA or direct memory access. During DMA the CPU is idle and it has no control over the memory buses. The DMA controller takes over the buses to manage the transfer directly between the I/O devices and the memory unit.
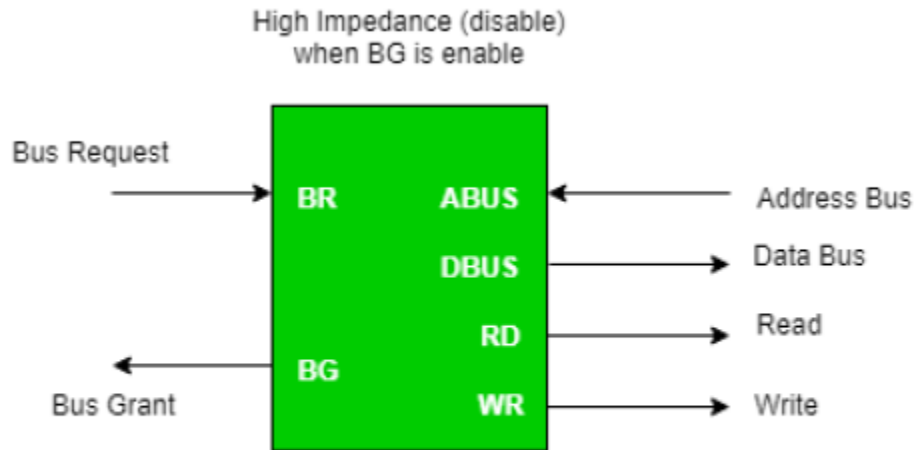
Figure - CPU Bus Signals for DMA Transfer

Bus Request : It is used by the DMA controller to request the CPU to relinquish the control of the buses.

 Bus Grant : It is activated by the CPU to Inform the external DMA controller that the buses are in high impedance state and the requesting DMA can take control of the buses. Once the DMA has taken the control of the buses it transfers the data. This transfer can take place in many ways.

Types of DMA transfer using DMA controller:

Burst Transfer : DMA returns the bus after complete data transfer. A register is used as a byte count, being decremented for each byte transfer, and upon the byte count reaching zero, the DMAC will release the bus. When the DMAC operates in burst mode, the CPU is halted for the duration of the data transfer. Steps involved are:

❖ Bus grant request time.

❖ Transfer the entire block of data at transfer rate of device because the device is usually slow than the speed at which the data can be transferred to CPU.

❖ Release the control of the bus back to CPU So, total time taken to transfer the N bytes = Bus grant request time + (N) * (memory transfer rate) + Bus release control time.

Where,

X μsec =data transfer time or preparation time (words/block)

Y μsec =memory cycle time or cycle time or transfer time (words/block)

% CPU idle (Blocked)=(Y/X+Y)*100

   % CPU Busy=(X/X+Y)*100

   Cyclic Stealing : An alternative method in which DMA controller transfers one word at a time after which it must return the control of the buses to the CPU. The CPU delays its operation only for one memory cycle to allow the direct memory I/O transfer to "steal" one memory cycle.

   Steps Involved are:

   1. Buffer the byte into the buffer

   2. Inform the CPU that the device has 1 byte to transfer (i.e. bus grant request)

   3. Transfer the byte (at system bus speed)

   4. Release the control of the bus back to CPU.

Before moving on transfer next byte of data, device performs step 1 again so that bus isn't tied up and the transfer won't depend upon the transfer rate of device. So, for 1 byte of transfer of data, time taken by using cycle stealing mode (T). = time required for bus grant + 1 bus cycle to transfer data + time required to release the bus, it will be N x T

In cycle stealing mode we always follow pipelining concept that when one byte is getting transferred then Device is parallel preparing the next byte. "The fraction of CPU time to the data transfer time" if asked then cycle stealing mode is used.

Where,

X μsec =data transfer time or preparation time

(words/block)

Y μsec =memory cycle time or cycle time or transfer

time (words/block)

% CPU idle (Blocked) =(Y/X)*100

   % CPU busy=(X/Y)*100

Interleaved mode: In this technique , the DMA controller takes over the system bus when the microprocessor is not using it. An alternate half cycle i.e. half cycle DMA + half cycle processor.
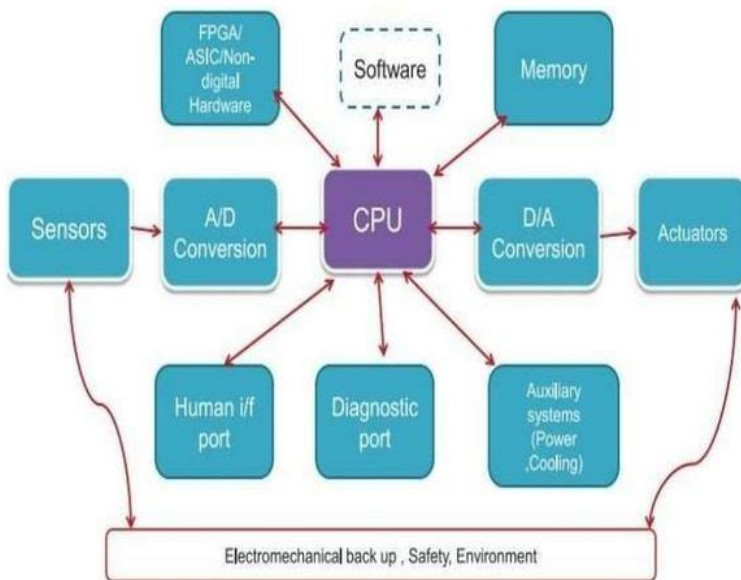
EMBEDDED SYSTEMS

MODULE 1

Syllabus:

Introduction to Embedded Systems– Components of embedded system hardware–
Software embedded into the system – Embedded Processors - CPU architecture of
ARM processor (ARM9) – CPU Bus Organization and Protocol. Design and
Development life cycle model - Embedded system design process – Challenges in
Embedded system design

EMBEDDED SYSTEMS

Embedded system- definitions

1. "An embedded system is a system that has software embedded into computer-hardware, which makes a system dedicated for an application (s) or specific part of an application or product or part of a larger system."

2. "An embedded system is one that has a dedicated purpose software embedded in a computer hardware."

3. "It is a dedicated computer based system for an application(s) or product. It may be an independent system or a part of large system. Its software usually embeds into a ROM (Read Only Memory) or flash."

4. "It is any device that includes a programmable computer but is not itself intended to be a general purpose computer."

5. "Embedded Systems are the electronic systems that contain a microprocessor or a microcontroller, but we do not think of them as computers – the computer is hidden or embedded in the system."

BLOCK DIAGRAM OF TYPICAL EMBEDDED SYSTEMS

Embedded systems are basically designed to regulate a physical variable or the state of some devices by sending some control signals to the Actuators/Devices connected to the output ports of the system in response to the input signals provided by the Sensors/Devices connected tothe input ports.

A typical embedded system contains a single-chip controller which acts as the master brain of the system. The controller can be a Micro-processor or a Micro-controller or a Field Programmable Gate Array device (FPGA) or a Digital Signal Processor (DSP) or an Application Specific Integrated Circuit (ASIC)/ Application specific system processor (ASSP). Key boards, push button switches etc. are examples for user interface input devices. LEDS, Liquid crystal displays (LCDS), Buzzers etc. are examples for common user interface output devices.

It's not necessary that all embedded systems should incorporate these I /O user interfaces.It depends only on the type of the application for which the embedded system is designed. Some embedded systems do not require any manual intervention for its operation. They automatically sense the variations in the input parameters through the sensors connected to the input port of thesystem.

The information from sensors is passed to the processor after signal conditioning & digitalization. Upon receiving the sensor data, the processor of the embedded system performs some pre-defined operations with the help of the firmware program embedded in the system & sends some signals to the actuators connected to the output port of the embedded system.

For an embedded system, it is the responsibility of designer to impart intelligence to the system. So, an embedded system without control algorithm in memory cannot be capable of making any decision depending on the situations as well as changes in the real world.

 The Memory of the embedded when is responsible for storing the control algorithm as software program code. In most of embedded systems, this memory is a kind of Read Only Memory (ROM) and it is not available for the user for modifications which means that the memory protected from unwanted user interaction. The common types of memories used in embedded systems FROM, UV-EPROM, EE-PROM & FLASH memory. Depending on the application, the memory may vary from a few bytes to megabytes.

Sometimes the system requires temporary memory while performing arithmetic operations and this type of memory known as "Working Memory". Random Access Memory (RAM) used in most the systems as the working memory. Various types of RAM like SRAM, DRAM and NVRAM are used for this purpose. The size of the RAM also varies from few bytes to kilobytes or megabytes depending on the application.

COMPONENTS OF EMBEDDED SYSTEM HARDWARE

An embedded system has three components:
1. **Hardware**.

2. **Application software**. This may perform concurrently the series of tasks or multiple tasks.

3. **Real Time Operating system (RTOS)** that supervises the application software and

providemechanism 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 ofapplication program. A small scale embedded system may not have RTOS.

Hardware.
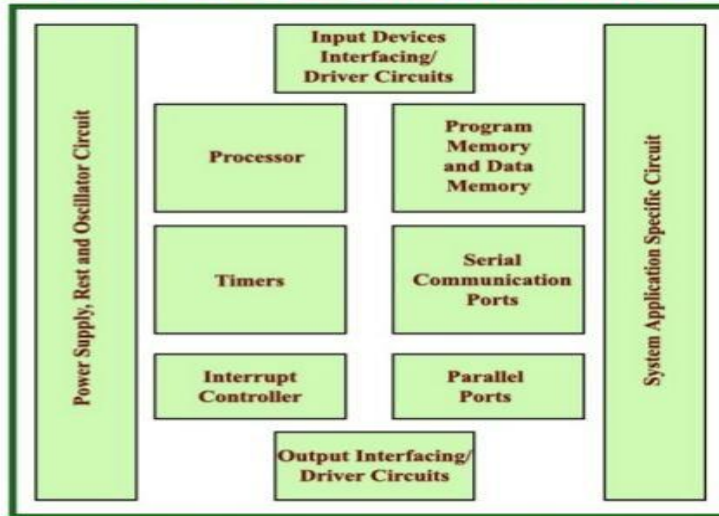
Processor - A Processor is the heart of the Embedded System. The main criteria for the processor are the processing power needed to perform the tasks within the system. Processors can be of the following categories:

➢ General Purpose Processor (GPP)

➢ Microprocessor

➢ Microcontroller

➢ Embedded Processor

➢ Digital Signal Processor / Media Processor

➢ Application Specific System Processor (ASSP)

➢ Application Specific Instruction Processors (ASIPs)

Power Source

Most embedded systems have a **power supply of their own**. The supply has specific operation range of voltages in one of the following 4 power ranges: **5.0 V + 0.25 V; 3.3 V + 0.3 V; 2.0 V + 0.2 V and 1.5 V + 0.2 V.** Certain systems do not have a power source of their own, so they are connect to external power supply.

For e.g. **A Graphic accelerator** do not have its own power supply.

Clock Oscillator Circuit *(Clocking Units)*

The clock is an another basic unit of a system. A processor needs a clock oscillator circuit as the clock controls the time for executing an instruction. The clock controls the various clocking requirements of the CPU, system clocks and the CPU machine cycles. For processing units, a highly stable oscillator is required as the clock signal provides the synchronizing of all other system units.

System Timers & Real-Time Clocks *(RTC)*

To schedule the various system tasks and for real-time programming, a system clock or an RTC is needed. These clocks drives the timers for various timing & counting needs in a system. System clock & RTC are also used to obtain delays and time-outs. A timer circuit is usually configured as the system-clock.

Another timer circuit is suitably configured as the real-time clock *(RTC)* for periodic saving of time & date in the system. Microcontrollers has built-in internal timer circuits for counting & timing devices

Timer - Embedded systems often require mechanisms for counting the occurrence of events and for

performing tasks at regular intervals.

➢ Embedded processors are often equipped with hardware support for this functionality.

➢ Timer is a device, which counts the input at regular interval using clock pulses at its input.

➢ The count increment on each pulse and store in a register, called count register.

➢ Timer is used for generating delay and for generating waveforms with specific delay.

Serial Port - A serial port is a serial communication interface through which information transfers in

or out one bit at a time.

➢ Serial data transmission is much more common in new communication protocols due to a reduction in the I/O pin count, hence a reduction in cost.

➢ Common serial protocols include UART, SPI, SCI and I2C etc.

➢ In most of the embedded systems at least two serial ports are provided.

Parallel Port - A parallel port is a type of interface found on computers or embedded systems for connecting peripherals.

The name refers to the way the data is sent; parallel ports send multiple bits of data at once.

Parallel ports require multiple data lines in their cables and port connectors, and tend to be larger than contemporary serial ports.

Interrupt Controller - An interrupt is a signal to the processor emitted by hardware or softwareindicating an event that needs immediate attention.

Interrupts allow an embedded system to respond to multiple real world events in rapid time.

By managing the interaction with external systems through effective use of interrupts can dramatically improve system efficiency and the use of processing resources.

In an embedded system there are usually multiple interrupt sources.

These interrupt sources share a single pin. The sharing is controlled by a piece of hardwarecalled an interrupt controller that allows individual interrupts to be either enabled or disabled.

System Application Specific Circuit - These are the dedicated circuits for the implementationof the application of particular system. This may varies from one system to other.

Reset Circuit, Power-up Reset & Watchdog-Timer Reset

Reset can be activated by an external reset circuit that activates on power-up *(switching- on)* the system. The reset circuit is a simple circuit *(such as an RC circuit)* whose output connects to the reset pin of the processor. To reset a processor, the reset circuit should activate for a fixed period of a few clock cycles & then deactivate thereby making the processor's reset pin active and then deactivate. Reset can also be activated by any one of the following:

(i) Software instruction *(e.g. RST instruction)*

(ii) Reset after a time-out by a programmed timer known as a watchdog timer

The watchdog timer is a timing device that resets the system after a predefined time-out of a few clock cycles. A watchdog timer reset is very essential in embedded systems because it helps in rescuing the system if the system program gets stuck due to a fault. On restart, the system can function normally. Most microcontrollers have on-chip watchdog timers.

Reset means that the processor begins the processing of instructions from a starting address. That address is one that is set by default in a processor on a power-up. From that memory address *(start-up addresses),* program-instructions are fetched following the reset of the processor. In certain processors, there are two start-up addresses. One is for the power-up reset and the processor fetches the program bytes from this address upon power-up. The other one is for after the execution of a "reset" instruction or after a timeout such as a watchdog timer based reset. Here, processor fetches the bytes program bytes from this second address on executing the reset instruction or on the watchdog timer based reset.

Memory

An embedded system uses different types of memory modules for a wide range of

taskssuch as storage of software code and instructions for hardware
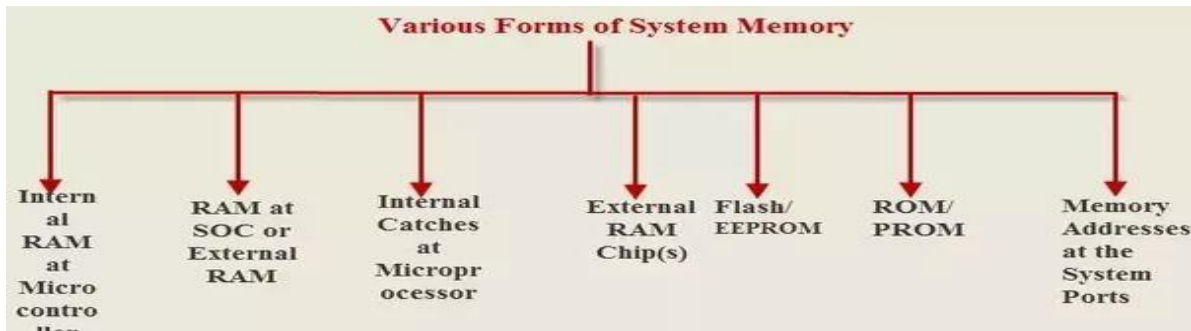
a. ROM or EPROM or Flash

1. Stores 'Application' program from where the processor fetches the instruction codes

2. Stores codes for system booting, initializing, Initial input data and Strings.

3. Stores Codes for RTOS.

4. Stores Pointers (addresses) of various service routines.

b. Internal, External and Buffer RAM - The function of these memory are

1. Storing the variables during program run,

2. Storing the stacks,

3. Storing input or output buffers for example, for speech or image .

c. EEPROM or Flash - Storing non-volatile results of processing.

d. Caches memory – Functions Assigned to the Caches

1. Storing copies of the instructions, data and branch-transfer instructions in

advance fromexternal memories and

2. Storing temporarily the results in write back caches during fast processing.

EMBEDDED PROCESSORS

*1.*General Purpose Processor *(GPP)*

A GPP is a general-purpose processor with instruction set designed not specific to the applications.   General purpose processors (GPP) are designed for general purpose computers such as PCs or workstations. The computation speed of a GPP is the main concern and the cost of the GPP is usually much higher than DSPs and microcontrollers. All techniques that can increase CPU speed have been applied to GPPs. For example, GPPs usually include on-chip cache and on-chip DMAs. Commonly used math operations are also supported by the on-chip

hardware. GPPs are not designed for fast real-time applications. Scalar structure is common in GPPs but rarely seen in DSPs and microcontrollers.

 E.g. Microprocessor; Embedded Processor.

2. Application-Specific Integrated Circuit (ASIC)

An application-specific integrated circuit is an integrated circuit (IC) customized for a particular use, rather than intended for general-purpose use. For example, a chip designed to run in a digital voice recorder is an ASIC. Modern ASICs often include entire microprocessors, memory blocks including ROM, RAM, EEPROM, flash memory and other large building blocks. Such an ASIC is often termed a SoC (system-on-chip). Designers of digital ASICs often use a hardware description language (HDL), such as Verilog or VHDL, to describe the functionality of ASICs.

3. Application Specific Instruction-Set Processor *(ASIP)*

An application-specific instruction set processor (ASIP) is a component used in system-on-a-chip design. The instruction set of an ASIP is tailored to benefit a specific application. This specialization of the core provides a tradeoff between the flexibility of a general purpose CPU and the performance of an ASIC. An ASIP is a processor with an instruction set designed for specific applications.

E.g. Microcontroller; Embedded microcontroller; Digital Signal Processor (DSP) and media processor; Network processor; IO processor etc.

4. Multi-core processors or multi-processor:

A multi-core processor is an integrated circuit (IC) to which two or more processors have been attached for enhanced performance, reduced power consumption, and more efficient simultaneous processing of multiple tasks (*see* parallel processing). A dual core set-up is somewhat comparable to having multiple, separate processors installed in the same computer, but because the two processors are actually plugged into the same socket, the connection between them is faster. In a dual core processor in practice, performance gains are said to be about fifty percent: a dual core processor is likely to be about one-and-a-half times as powerful as a single core processor

 The following are important considerations when selecting a processor:

(1) Instruction set

(2) Maximum bits in an operand *(8 or 16 or 32)*

(3) Clock frequency in MHz

(4) Processing speed in MIPS *(Million Instructions Per Second)*

(5) Processor ability to solve complex algorithms

5. MICROPROCESSOR

A microprocessor is a single VLSI chip that has a CPU. It may also have some other units *(e.g., caches, floating point processing arithmetic unit, pipelining units etc.)* that are additionally present for the faster processing of instructions. The CPU is a unit that fetches & processes a set of instructions. The CPU instruction set includes instructions for data transfer operations, ALU operations, stack operations, IO operations, program control & sequencing operations. The important microprocessors used in the embedded systems are ARM, 68HCxxx, 80x86 and SPARC family of microprocessors. *A microprocessor is used as general-purpose processor when large embedded software has to be located in the external memory chips.*

6. MICROCONTROLLER *(MICRO-COMPUTER)*

A microcontroller is a single-chip VLSI unit having limited computational capabilities, possesses enhanced IO and a number of on-chip functional units. A microcontroller has processor, memory & several other hardware units *(peripherals)* in it; these form the microcomputer part of the embedded system. Choosing a microcontroller as a processing unit depends upon the application-specific features in it. Microcontrollers are particularly suited for use in embedded systems for real-time control applications with on-chip memory & peripherals/devices. A microcontroller is used when a small embedded software has to be located in the internal memory and when on-chip functional units such as various types of peripherals such as ports, timer, ADC, PWM etc. are required.

7. SINGLE PURPOSE PROCESSORS

(1) **Coprocessor** *(for e.g., math coprocessor, floating point processor etc.)*

(2) **Graphics processor:** - An image consists of a no. of pixels.

- A separate graphics processor is required for applications such as gaming, display fromgraphics memory and rotate an image or its segments.

(3) **Pixel coprocessor:** - A pixel coprocessor is required in digital cameras for operation onimages such as rotate right, rotate-left, rotate- up, rotate-down, shift to next, shift to previous.

(4) **Encryption engine:** - A suitable algorithm runs in this processor to encrypt data for securetransmission.

(5) **Decryption engine:** - A suitable algorithm runs in this processor to decrypt the encrypteddata at receiver's end.

(6) **A discrete cosine transformation *(DCT)* and inverse transformation *(DCIT)* processor** isrequired in speech and video processing.

(7) **Protocol stack processor:** - A protocol stack required before an application data is sent to anetwork.

- At the receiver's end, the protocol stack is received and application data is accepted accordingly. **(E.g. for Wi-Fi protocol)**

- MP3 decompression is done before retrieving or playing files.

(8) Network processor:

- A network processor's functions are to establish a connection, finish, send & receive acknowledgements, send and receive retransmission requests and check and correct received dataframe errors.

- The network processor's functions include all protocol related processing functions.

(9)Accelerator

- For e.g., **Java codes accelerator** is a coprocessor that accelerates computations by takingactions that are in Java programs.

(10) Controller

- E.g., DMA Controller *(Direct Memory Access),* bus control, peripheral management

(11) **CODEC** *(Coder & Decoder)*

- A CODEC is a processor circuit encodes input bits or decodes the encoded information into a complete set of bits or original signals. The CODEC functions as a compression & decompression unit signals such voice, speech, image or video signals

JPEG CODEC:- This is a processor for jpg compression and decompression.

MPEG CODEC:- MPEG3 CODEC is a processor for mp3 compression of audio/video data streams for storing or transmitting

EMBEDDED SOFTWARE IN A SYSTEM

The software program code (instruction codes) is the brain of an embedded system. An embedded system processor executes software that is specific to a given application. The program codes are placed in the ROM (or flash memory or PROM) for the execution of tasks when the system runs. This final "machine implementable software" is called the "ROM image" that is being embedded into the ROM similar to an "image" in an "image frame". Each program code is in bytes format & these bytes are saved at each address of the system memory (ROM). The code bytes are required at each ROM address to execute the tasks.

So, a machine implementable software file (ROM image) is similar to a table having many rows and only two columns; 1st column for memory address & 2nd column for corresponding code byte in a memory address. By changing ROM image, the same hardware platform will work differently.

1) MACHINE-CODE BASED CODING

In machine-code based coding, the programmer defines the   machine code bytescorresponding each memory addresses for a program.

- Machine-code based coding is done only in specific situations because it is time consuming and the programmer must have to understand the processor instructions set and their corresponding machine codes

2) CODING IN ASSEMBLY LANGUAGE

Small programs can be coded in assembly language after understanding the processor & its instruction set. "Assembler" is the software used for converting the codes written in assemblylanguage (similar to a compiler for high level language like C, Java etc) . Assembly language coding is extremely useful for configuring devices like ports, ADC, DAC etc. But, Assembly language based programming is also very time consuming while making larger programs/ codes. Full coding in assembly may be done only for a few simple, small-scale embedded systems.
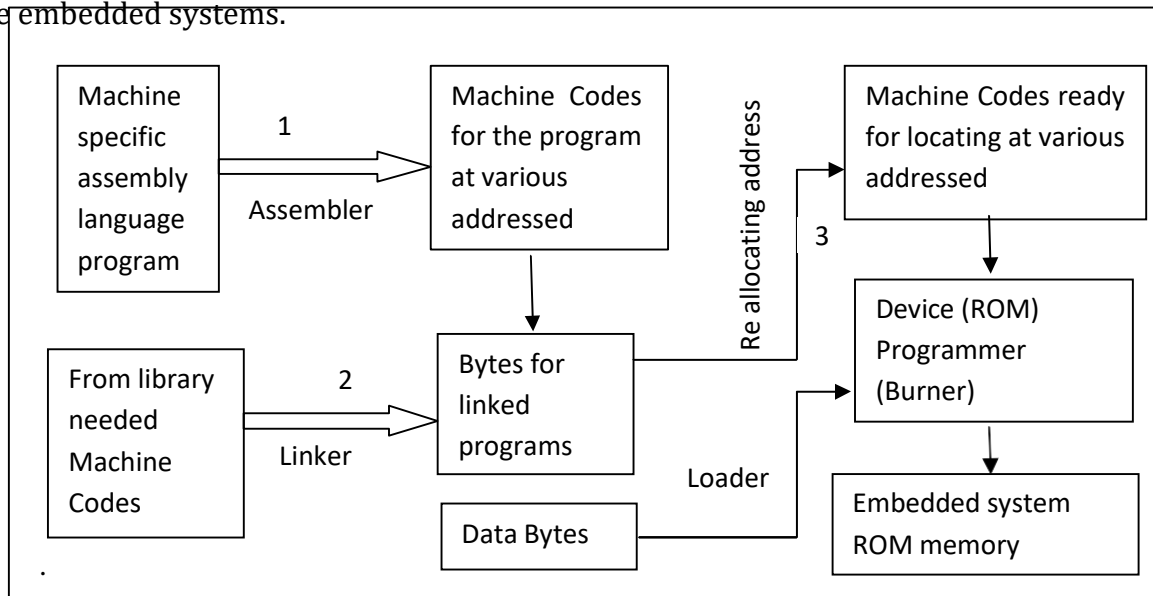
Figure: shows the process of converting an assembly language program into machine implementable software file and then finally obtaining a ROM image file.

Assembler, Linker, Locator & Loader are the software required for the whole process.

(a) First step is called "Assembling" in which assembler software translates the assemblysoftware into the machine codes.

(b) Next step is called linking; a linker links these codes (if necessary) with the other codes takenfrom the library.

(c) For a final program, a no. of other codes are to be linked together.

For eg., there are the standard codes for delay function (eg. delay( ) in Arduino). If 'delay( )' is included in the program, the program codes for the delay( ) must link with the final assembled code. The linked file in binary is known as executable file (a file with '.EXE'

extension).

(**d**) In embedded systems, the next step after linking is the use of a "Locator" software which locates the already fixed ROM addresses. For eg., in a memory-mapped IO scheme; IO port addresses, IO devices addresses etc. are permanently assigned to some memory locations.

- The locator software "re-allocates" the memory addresses in a linked file & creates a file with permanent memory allocation for each of the code bytes in a standard format

- Eg for such a standard file format is "Intel .HEX file format".

(**e**) In the next step, the "Loader" software performs the task of placing/ loading the code bytes as an "image to be placed in ROM" by finding the exact available ROM memory addresses

- For many processors, the available memory addresses may not start from "0000H".

- The "loader" finds the appropriate "start address" for the final program codes.

(**f**) Lastly "Programmer Device/ Equipment" takes as input the ROM image file.

For eg. (in .HEX format) & "writes" the image as byte by byte into the memory. So the process of placing the codes into ROM or flash memory is also called "Burning" into the ROM/flash.

3) CODING IN HIGH LEVEL LANGUAGE

For large software programs development, high-level language like C, C++, visual C++, Java etc. are used. 'C" is usually the preferred language. The programmer needs to understand only the hardware organization of the whole embedded system when coding in high level language
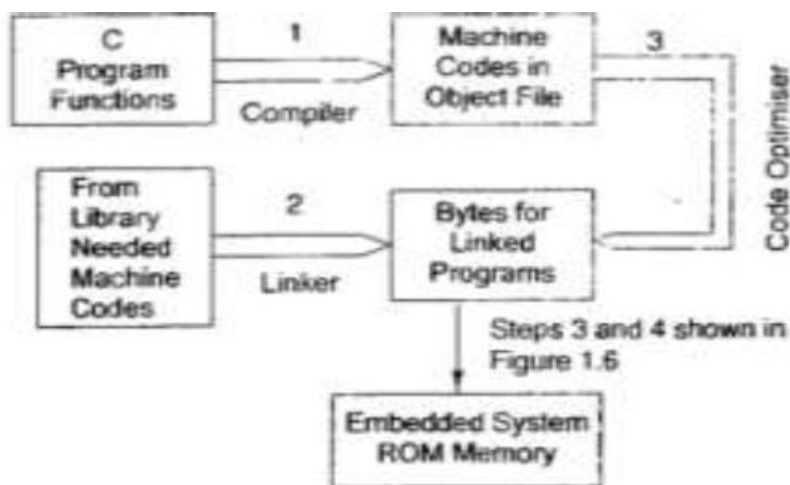
Figure shows the process of converting a C program into the ROM image file.

-First, the compiler software generates the object codes (file with .OBJ extension). The

compilerassembles the codes according to the processor instruction set

- Before linking, the compiler may use a "Code-Optimizer" that optimizes the codes by removing the "redundant/ unnecessary variables or steps" written in the program

- The linker links the object codes with other standard program codes in the program (similar to the linking step in assembling process). For eg, the linker includes the program codes for the pre-defined functions like printf(), delay() etc

- Codes for some standard devices & the device control management also link at this

stage;For eg., a printer device management & its driver codes

- After linking, the other steps for creating a file for ROM image are same as that discussed inprevious section of assembling process (Locating, Loading & Burning).

4) DEVICE DRIVERS

- A device driver is a software written in a high level language that controls functions/actions ofa device such as opening the device, "connecting the device to other devices", "reading the device", "writing to the device" & "closing the device"

- Each of the device driver software can access a parallel port, serial port, keyboard, mice, disk, network, display, file etc. at specific addresses.

- A device driver software controls 3 functions,

(i) Initializing a device by placing appropriate bits at the control register of the device.

(ii) Calling an Interrupt Service Routine upon setting up a status flag in the status register of thedevice for running the ISR.

(iii) Re-setting the status flag after an interrupt service is completed Device Manager Software

- A device manager software provide programs for

• Detecting the presence of devices connected to the system (both virtual & physical)

• Listing out the available devices connected to the system

• Initializing the devices testing the devices

• Allocates "Port Addresses" for the various physical & virtual devices (Eg. COM2, COM3 etcfor virtual serial ports in Windows)

ARM PROCESSOR

Advanced RISC machine (ARM) is the first reduced instruction set computer (RISC) processor for commercial use, which is currently being developed by ARM Holdings.

RISC:- A Reduced Instruction Set Computer (RISC) is a microprocessor that has been designedto perform a small set of instructions, with the aim of increasing the overall speed of the processor. The RISC concept first originated in the early 1970's when an IBM

research team proved that 20% of instruction did 80% of the work. The RISC architecture follows the philosophy that one instruction should be performed every clock cycle.

Comparison between CISC and RISC

CISC	RISC
Larger set of instructions. Easy to program	Smaller set of Instructions. Difficult to program.
Simpler design of compiler, considering larger set of instructions.	Complex design of compiler.
Many addressing modes causing complex instruction formats.	Few addressing modes, fix instruction format.
Instruction length is variable.	Instruction length varies.
Higher clock cycles per second.	Low clock cycle per second.
Emphasis is on hardware.	Emphasis is on software.
Control unit implements large instruction set using micro-program unit.	Each instruction is to be executed by hardware.
Slower execution, as instructions are to be read from memory and decoded by the decoder unit.	Faster execution, as each instruction is to be executed by hardware
Pipelining is not possible.	Pipelining of instructions is possible, considering single clock cycle.
Mainly used in normal pc's, workstations & servers.	Mainly used for real time applications.

ARM9

- The ARM9TDMI is a member of the ARM family of general-purpose microprocessors.
- The ARM9TDMI is targeted at embedded control applications where high performance, low die size and low power are all important.
- The ARM9TDMI supports both the 32-bit ARM and 16-bit Thumb instruction sets, allowing the user to trade off between high performance and high code density.
- ARM and Thumb are two different instruction sets supported by ARM cores with a "T" in their name. Thumb mode allows for code to be smaller, and can potentially be faster if the target has slow memory
- The ARM9TDMI supports the ARM debug architecture and includes logic to assist in both hardware and software debug.
- The ARM9TDMI supports both bidirectional and unidirectional connection to external memory systems.
- The ARM9TDMI also includes support for coprocessors.

- The ARM9TDMI processor core is implemented using a five-stage pipeline consisting of fetch, decode, execute, memory and write stages.
- The device has a Harvard architecture, and the simple bus interface eases connection to either a cached or SRAM-based memory system.

Advanced RISC Machine (ARM) Processor Fundamentals

Like all RISC processors, the ARM processor also uses an architecture known as the "Load-Store Architecture" which means ARM has only two instructions for transferring data in & outof the processor & its memory.

The two instructions are

(i) 'LOAD' Instruction:- Copy data from Memory to CPU registers in ARM core

(ii) 'STORE' Instruction :- Copy data from CPU registers of ARM core to Memory.
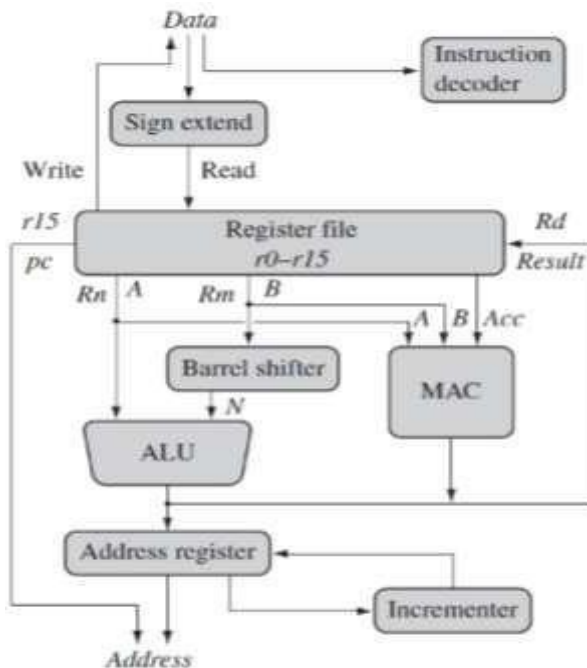
- In ARM, there are no other instructions that directly manipulate data in the memory.
- Fig. shows a Von-Neumann implementation of ARM core, in which the instructions & the data share the same bus (A Harvard implementation of ARM core has two separate buses of instructions & data)
- So, Data/ Instructions enters the processor core through the 'Data Bus' . The 'Instruction Decoder' translates instructions before they are executed.
- Data items are placed in the 'Register File', which is a CPU register bank made up of 16 general purpose registers (r0 to r15) in which each register has a size of 32-bit.
- The ARM core is a 32-bit processor in which the general purpose registers (r0 - r15) of 'Register_File' can hold only 32-bit data values. The 'Sign Extend Hardware'

converts the signed 8 or 16-bit data inputs to 32-bit data values while they read from memory

- ARM instructions have two Source Registers : 'Rn' & 'Rm'.One Destination Register :'Rd'(also called as Result Register).
- The source operands are read from the register file using internal buses 'A_bus' & 'B_bus' respectively.
- The ALU or MAC(Multiply accumulator) unit of ARM takes the source data in the registers 'Rn' & 'Rm' through A bus & B bus for computing the result.
- After passing through the functional units (ALU or MAC), the result in 'Rd' register is written back to the register file through the 'Result Bus'.
- Here, the data in one of the source register 'Rm' can also be shifted in a 'Barrel_Shifter' unit before it enters the ALU . Using the combined operation of the Barrel Shifter & ALU, the ARM can generate a wide range of expressions & addresses.
- In ARM, the ALU also generates the memory addresses that is to be placed in the addressregister for broadcasting the address through the 'Address Bus'.
- The 'Incrementer' is used to update the address register for a data transfer operation from sequential memory locations.

ARM 9 ARCHITECTURE
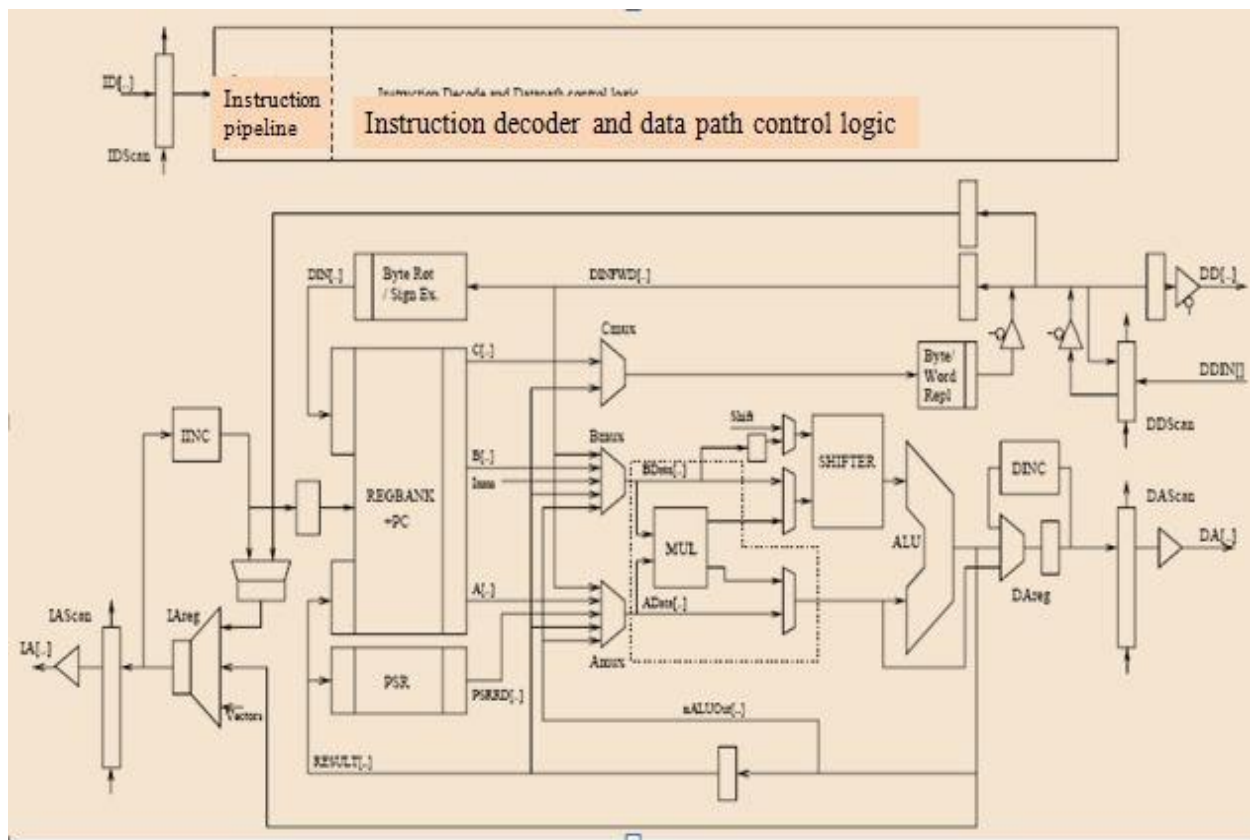
The main parts of the ARM processor are:

1. Register file: The processor has a total of 37 registers made up of 31 general 32 bit registersand 6 status registers

2. Booth Multiplier

3. Barrel shifter

4. Arithmetic Logic Unit (ALU)

5. Control Unit.

Registers

The processor has a total of 37 registers made up of 31 general 32 bit registers and 6 status registers. At any one time 16 general registers (R0 to R15) and one or two status registers are visible to the programmer. The visible registers depend on the processor mode. In all modes 16 registers, R0 to R15, are directly accessible. All registers except R15 are general purpose andmay be used to hold data or address values. Register R15 holds the Program Counter (PC). A seventeenth register (the CPSR - Current Program Status Register) is also accessible. It contains condition code flags and the current mode bits and may be thought of as an extension to the PC. R14 is used as the subroutine link register and receives a copy of R15 when a Branch and Link instruction is executed.

Program status register:- It contains the processor flags (Z, S, V and C). The modes bits alsoexist in the program status register in addition to the interrupt and fast interrupt disable bits

Some special registers: Some registers are used like the instruction register, memory data readand write register and memory address register

Multiplexers: Many multiplexers are used to control the operation of the processor buses

Arithmetic Logic Unit (ALU) :- The ALU has two 32-bits inputs. The first comes from the register file while the other comes from the shifter. ALU outputs modify the status register flags.

Booth multiplier:- The multiplier has three 32-bit inputs. All the inputs come from the register file. The multiplier output is only the 32 least significant bits of the product.

Barrel shifter:- A **barrel shifter** is a digital circuit that can **shift** a data word by a specified number of bits without the use of any sequential logic, only pure combinational logic. The barrel shifter has a 32-bit input to be shifted. This input is coming from the register file or it could be immediate data. The shifter has other control inputs coming from instruction register. Shift field in the instruction controls the operation of the barrel shifter. This field indicates the type of shift to be performed (logical left or right, arithmetic right or rotate right). The amount by which the register should be shifted is contained in an immediate field in the instruction or it could be the lower 6 bits of a register in the register file.

Control Unit :- For any microprocessor, control unit is the heart of the system. It is responsiblefor the system operation and so the control unit design is the most important

part in the hole design. Control unit is usually a pure combinational circuit. The processor timing is also included in the control unit.

Pipeline implementation

The ARM9TDMI implementation uses a five-stage pipeline design. These five stages are:

1. Instruction fetch (F):- The instruction is fetched from memory and placed in the instruction pipeline
2. Instruction decode (D):- The instruction is decoded and register operands read from the register files. There are 3 operand read ports in the register file so most ARM instructions can source all their operands in one cycle
3. Execute (E):- An operand is shifted and the ALU result generated. If the instruction is a loador store, the memory address is computed in the ALU
4. Data memory access (M):- Data memory is accessed if required. Otherwise the ALU resultis simply buffered for one cycle
5. Register write (W):- The result generated by the instruction are written back to the register file, including any data loaded from memory
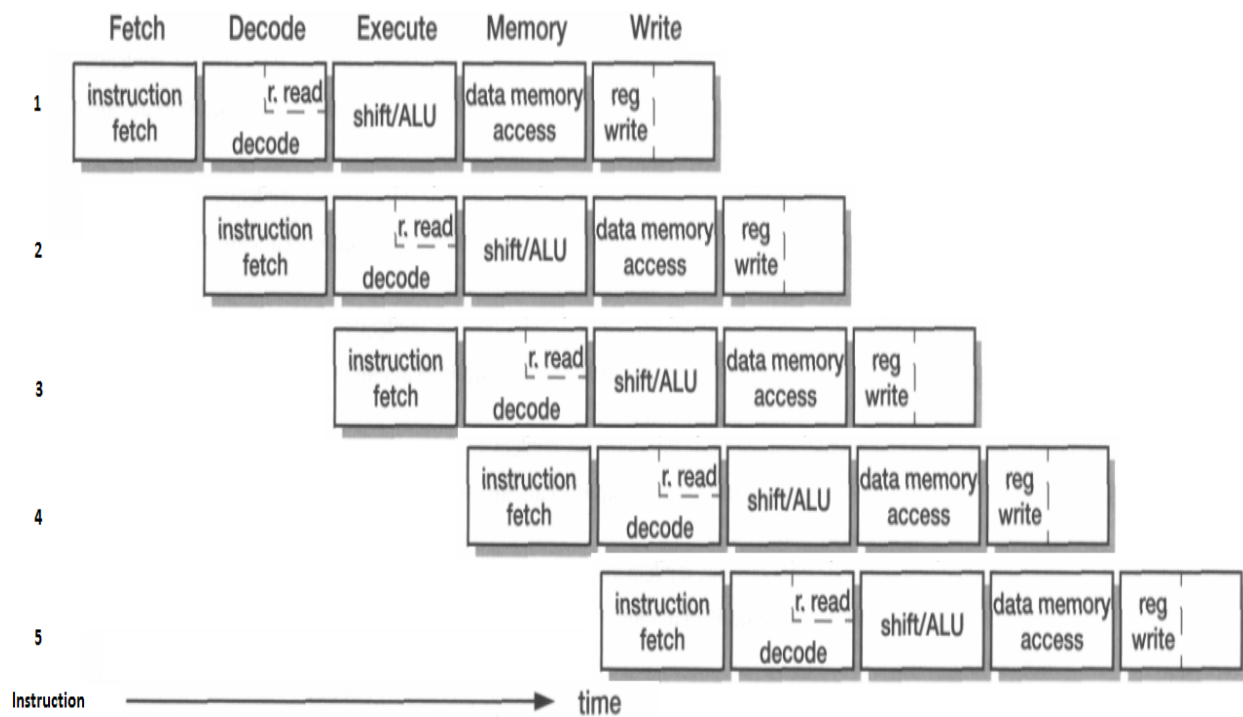
ARM Bus Organization -AMBA
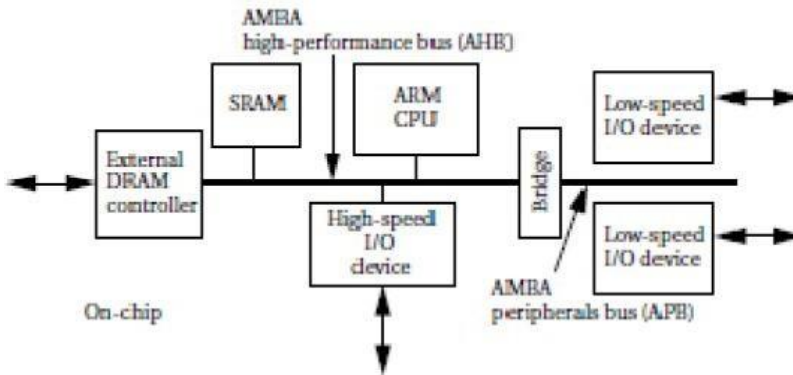
Figure: ARM Bus structure

ARM has created a separate bus specification for single-chip systems. **The AMBA[Advanced Microcontroller Bus Architecture] bus** [ARM99A] supports CPUs, memories, and peripherals integrated in a system-on-silicon.

The AMBA specification includes two buses.

The AMBA high-performance bus (AHB) is optimized for high-speed transfers and is directly connected to the CPU. It supports several high performance features: pipelining, burst transfers.

Burst mode is a temporary high-speed data transmission mode used to facilitate sequential data transfer at maximum throughput. Burst mode data transfer rate (DTR) speeds canbe approximately two to five times faster than normal transmission protocols.,

Split transactions:- the Split and Retry responses are used by slaves which require a large number of cycles to complete a transfer. These responses allow a data phase transfer to appear completed to avoid sharing the bus, but at the same time indicate that the transfer should be re- attempted when the master is next granted the bus.

The difference between them is that a SPLIT response tells the Arbiter to give priority to all other masters until the SPLIT transfer can be completed (effectively ignoring all further requests from this master until the SPLIT slave indicates it can complete the SPLIT transfer), whereas the RETRY response only tells the Arbiter to give priority to higher priority masters.

A SPLIT response is more complicated to implement than a RETRY, but has the advantage that it allows the maximum efficiency to be made of the bus bandwidth.

Multiple bus masters.

The AMBA bus specification supports multiple bus masters on the high performance ASB. Asimple two wire request and grant mechanism is implemented between the arbiter and each bus master. The arbiter ensures that only one bus master is active on the bus and also ensures that when no masters are requesting the bus, a default master is granted.

The specification also supports a shared lock signal. This allows bus masters to indicate

that the current transfer is indivisible from the following transfer and prevents other bus masters fromgaining access to the bus until the locked transfers have completed. A bridge can be used to connect the AHB[AMBA High-performance Bus] that is a single clock-edge protocol, to an AMBA peripherals bus(APB). This bus is designed to be simple and easy to implement; it also consumes relatively little power. The AHB assumes that all peripherals act as slaves, simplifying the logic required in both the peripherals and the bus controller. It also does not perform pipelined operations, which simplifies the bus logic.

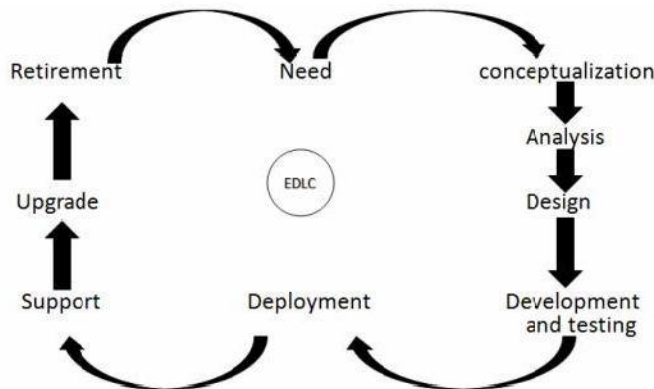**Embedded Product Development Life Cycle (EDLC)**

Figure : Phases of EDLC

EDLC is Embedded Product Development Life Cycle:- It is an Analysis, Design, Implementation based problem solving approach for embedded systems development.

Different Phases of EDLC: The following figure depicts the different phases in EDLC:

Need :- The need may come from an individual or from the public or from a company.

Conceptualization :- Defines the scope of concept, performs cost benefit analysis and feasibility study and prepare project management and risk management plans.

Analysis:- The product is defined in detail with respect to the inputs, processes, outputs, and interfaces at a functional level.

Design :- The design phase identifies application environment and creates an overall architecturefor the product.

Development and Testing :- Development phase transforms the design into a realizable product.

Deployment :- Deployment is the process of launching the first fully functional model of the product in the market.

Support :- The support phase deals with the operational and maintenance of the product in the production environment.

Upgrades :- Deals with the development of upgrades (new versions) for the product which is already present in the market.

Retirement/Disposal :- The retirement/disposal of the product is a gradual process. This phase is the final phase in a product development life cycle where the product is declared as discontinued from the market.

Waterfall Model

· Linear or waterfall model is one of the EDLC models which adopted in most of the olden systems.
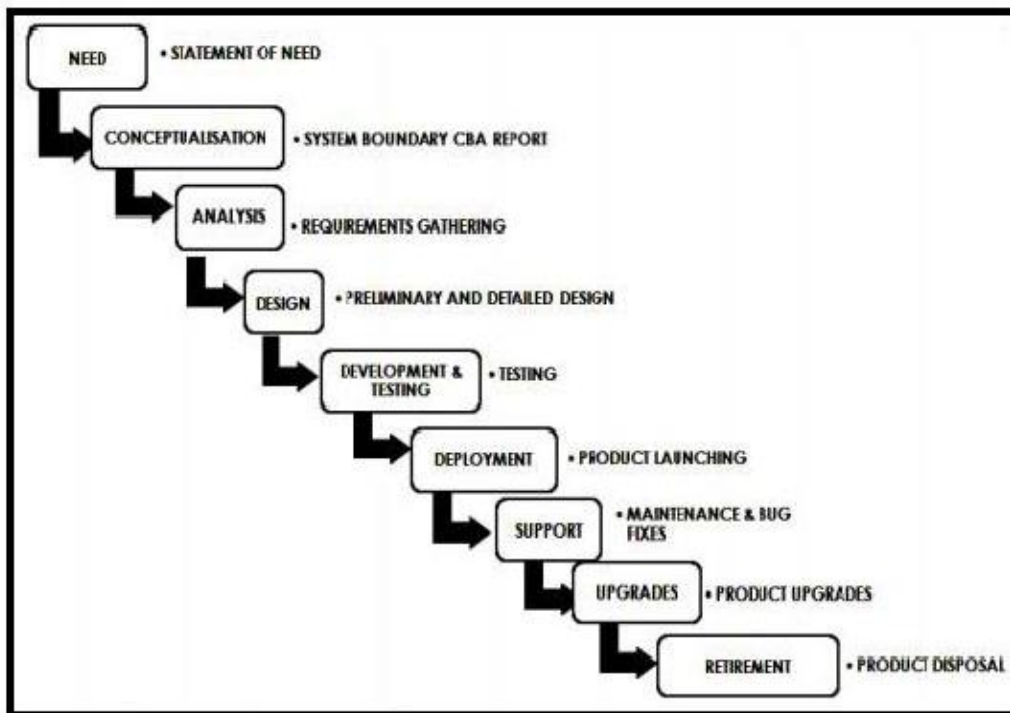
Figure: Waterfall Model

In this approach each phase of EDLC (Embedded Development Product Lifecycle) is **executed in sequence.** It establishes analysis and design with **highly structured development phases**. The execution flow is **unidirectional.** The **output of one phase serves as the input of the next phase.** All activities involved in each phase are well planned so that what should be done in the next phase and how it can be done. The feedback of each phase is available only after they are executed. It implements extensive review systems to ensure the process flow is going in the right direction. One significant feature of this model is that even **if you identify bugs in the current design the development process proceeds with the design.** The fixes for the bug are postponed till the support phase.

Advantages

Product development is rich in terms of: · Documentation, Easy project management, Goodcontrol over cost & Schedule

Drawbacks

· It assumes all the analysis can be done without doing any design or implementation

· The risk analysis is performed only once.

· The working product is available only at the end of the development phase

· Bug fixes and correction are performed only at the maintenance / support phase of the lifecycle.

Challenges in Embedded Systems

1. Amount and type of hardware needed. Optimizing various hardware elements for a particulardesign.

2. Taking into account the design metrics

Design metrics examples –power dissipation, physical size, number of gates and engineering, prototype development and manufacturing costs.

3. Optimizing the Power Dissipation:- Clock Rate Reduction and Operating Voltage Reduction

4. Disable use of certain structural units of the processor to reduce power dissipation the processor to reduce power dissipation. Control of power requirement, for example, by screen auto-brightness control

5. Process Deadlines:- Meeting the deadline of all processes in the system while keeping the memory, power dissipation, processor clock rate and cost at minimum is a challenge.

6. Flexibility and Upgradeability:- Ability to offer the different versions of a product for marketing and offering the product in advanced versions later on.

7. Reliability:- Designing reliable product by appropriate design and thorough testing, verification and validation is a challenge.

8. Testing, Verification and Validation:- Testing – to find errors and to validate that the implemented software is as per the specifications and requirements to get reliable product.