A

Text Book

Of

Embedded C Programming

(For Fourth Semester Diploma in Electronics & Communication Engg.)

As per new curriculum w.e.f. 2021-22

(As per C - 20 syllabus)

Umamaheshwar V.
Sl.Gr. Lecturer, Govt. Polytechnic for Women
Shiralakoppa

Ishwarkumar Nirale
Lecturer in E&C Dept.
Govt. Polytechnic, Mudhol

eastern book promoters belgaum

Publishers

Vijay I. Revanaki
Praveenkumar I. Bevinamarad

First Edition - 2022

© Publishers

No part of this publication may be reproduced stored in a retrieved system or transmitted in any form, or by any means, electronc, mechanical, photocopying, recording or otherwise, without the prior written permission of the Author and Publishers.

No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the author and publishers assume no responsibility for errors and omissions.

Price : ₹ 185-00
(A xerox Avoidable Price)

Published by :

eastern book promoters belgaum
J.T.Math Road, Gadag - 582 101
Mob: 9448390180, 9448390181
08372-232155,200014
email: cbpb_gdg@yahoo.co.in

Type setting at : Juktimath Computers

Preface...

This Book is written according to the ***C-20*** *- A Skill base Curriculum, the revised syllabus of the course* ***"Embedded C Programming"*** *of Fourth Semester - Electronics and Communication Programme as prescribed by the Department of Collegiate and Technical Education.*

The course content is of 13 Weeks which includes Basics of C programming, 8051 Microcontroller its Architecture and techniques of programming (Assembly and Embedded C). Also emphasizes on usage of I/O ports, Interrupts, Timers/Counters, Serial I/O and Interfacing the 8051.

The presentation of the Book is deliberately made simple in order to help students from rural areas to undergo Technical Education. Summary and review questions have been included at the end of each Week. Laboratory Practicing Exercises/Programs have been included. We hope this Volume will cater the needs of Fourth Semester E&C Diploma Students.

Any Errors, Omissions and suggestions for the improvement of this book is greatly acknowledged.

25/April/2022

Umamaheshwar.V
Ishwarkumar Nirale

Acknowledgements.....

We take this opportunity to express our grateful acknowledgements to the following individuals:

- *Praveenkumar Ishwar Bevinamarad and Vijay Ishwar Revanaki of eastern book promoters belgaum, Gadag.*
- *Sri Ravishankar D.S.Principal of Government Polytechnic for Women, Shiralakoppa.*
- *Sri Sanjeevkumar Chavan, Principal of Government Polytechnic, Mudhol,*
- *Principals, HOD's, Senior faculties, Lecturers Working in various polytechnics of Karnataka,*
- *Our beloved students who demanded and supported to write this book.*

We cordially express our deep gratitude to all our friends, colleagues, Well-wishers and everyone who directly or indirectly motivated us to prepare this study material.

25/April/2022

Umamaheshwar.V
Ishwarkumar Nirale

Government of Karnataka

DEPARTMENT OF COLLEGIATE AND TECHNICAL EDUCATION

Programme	Electronics and Communication	Semester	IV
Course Code	20EC43P	Type of Course	Programme Core
Course Name	Embedded C Programming	Contact Hours	8 hours/week 104 hours/ semester
Teaching Scheme CIE Marks	L:T:P :: 3:1:4 60	Credits SEE Marks	6 40

1. Rationale

C is a general purpose programming language which is robust and highly portable used for scripting system applications which form a major part of all operating systems. C language is available on a very wide range of platforms, from embedded microcontrollers to supercomputers. Microcontroller is a compressed microcomputer manufactured to control the functions of embedded systems in various fields such as automobile, aeronautics, robotics, mobile communication, electronic appliances, industrial processing, defense, space, medical applications etc. The future of the micro controller depends on machine learning in embedded systems.

2. Course Outcomes: On successful completion of the course, the students will be able to:

CO1	Write the code using C constructs for a given requirement, execute the program, debug and to demonstrate that the program produces the required result/output.
CO2	List the various components and the characteristics of each component in a 8051 Microcontroller.

CO3	Write an embedded program for a given requirement, test and troubleshoot to obtain the desired output.
CO4	Identify the right microcontroller/peripheral device using data sheets / specification sheets for a given application.

3. Course Content

Week	C O	PO	Lecture (Knowledge Criteria)	Tutorial (Activity Criteria)	Practice (Performance Criteria)
			3 hours/week	1 hour/week	4hours/week (2 hours/ batch twice in a week)
1	1	1,2,3	1.Introduction to C - features, compilation process 2.C tokens, variables and identifiers, constants. 3. Data types - classification, memory requirement, range of values, usage.	Refer Table 1	1. Familiarisation of TURBO C. 2. Programs to illustrate the use of different data types and verify their memory size.
2	1	1,2,3	1.Operators and Operands- Arithmetic, logical, relational operators. 2.Unary, conditional, assignment and special operators, precedence and associativity. 3.Basic input and output functions, format specifiers, preprocessor directive & library functions	Refer Table 1	1a. Compute simple interest given the principal, interest rate and duration. b. Compute compound interest given P,t,r,n. 2a. Compute the area of a circle, square, rectangle and triangle. b. Swap contents of two variables without using intermediate variables.

3	1	1,2,3	1.Flowchart and Algorithm. Structure of a C program, simple C programs. 2.Branching- conditional -if, if-else, example programs. 3.Nested if-else, switch, example programs.	Refer Table 1	1a. Compute the largest of three numbers using if-else and ternary operators. b. Compute the result of a student using nested if 2. Given the resistance and tolerance, generate the color bands of the resistor using a switch statement.
4	1	1,2, 3	1.Looping- for, while, do-while loops. 2.Example programs on looping. 3.Arrays- definition, declaration, initializing single dimensional arrays. Examples.	Refer Table 1	1a.Compute factorial of a single digit number. b. Compute the sum of digits of a given 3 digit number reducing it to a single digit. 2. Sort an array of numbers in ascending order and descending order.
5	1	1,2, 3	1.Strings- declaration, initialization with an example. Two dimensional arrays- declaration, initialization with an example. 2.Functions- elements of user defined functions, example. 3.Pointers- introduction with example. Structures- introduction with example.	Refer Table 1	1a. Compute the length of a string and reverse the string using string functions. 1b. Compute the sum of two matrices. 2a. Compute cube of a number using a

					function. 2b. Store the details of an employee using a structure and print the details
6	2	1	1.Introduction to the concepts of embedded systems, microprocessors, microcontrollers. 2.Selection of 8 bit, 16 bit, 32 bit, 64 bit microcontrollers. Introduction to 8051 microcontroller. 3. Architecture of 8051 microcontroller, PSW and special function registers.	Refer Table 1	1. Identification of program development tools. 2. Familiarization of program development using Keil.
7	2	1,2, 3	1.Memory organization, general purpose RAM, bit addressable RAM. 2.Register banks, Pin details of 8051.	Refer Table 1	1. Familiarize with the structure of the 8051 assembly program and executing it. 3. Interfacing external data and code memory. 2. Write and execute simple ALP to understand different addressing modes.
8	3	1,2, 3	1.8051 Addressing modes. 2.Instruction set- classification, syntax and function of data transfer instructions, 3.Arithmetic instructions, Logical instructions.	Refer Table 1	1.Write and execute an ALP to (a) Move a block of data within internal RAM

					(b) Exchange a block of data between internal RAM and external memory. 2. Write an ALP to (a) evaluate simple arithmetic expression such as y= (((5*2)-(4+1))/3) %2. (b) Perform addition of three 8-bit BCD numbers to result in BCD form.
9	3	1,2, 3	1.Bit level instructions, jump instructions. 2.Introduction to Embedded C and its applicability to 8051. 3.General structure of embedded C program, data types.	Refer Table 1 Refer Table 1	1.Write an ALP to (a) Rotate or shift 16-bit data. (b) Evaluate simple logical expression such as Y= a&&b||c^!d where a, b, c and d are 8-bit data. 2.Write and execute an assembly and embedded C program to convert (a)Packed BCD to unpacked BCD (b) Unpacked BCD to packed BCD.

10	3	1,2,3	1. Memory types and models, pointers,pointer's memory type. 2. Time-delay generation using loops, example program. 3 Arithmetic and logical operators, example programs.		1. Write and execute a program to search a given 8-bit number in an array of N numbers using embedded C. 2 Write and execute a program to toggle a particular bit in the internal RAM with the use of delay subroutine.
11	3,4	7	1.Features of I/O ports. Interface I/O devices such as LED, buzzer with programs. 2.Polling & interrupt methods, executing an interrupt, IE and IP registers. 3.Enabling, disabling and priority setting, example programs.		1. Write and execute an embedded C program to toggle the LED/buzzer with tone using push-button switch. 2. Write ALPs to enable, disable and priority setting of interrupts and verify it in IE and IP registers.
123,4	7	1.	Bit structure and function of TMOD and TCON registers, mode 1 operation of timers.		1. Write and execute an embedded C program to generate a

			2 Time delay generation & example programs. 3 Bit structure of SCON register, SBUF register, TI and RI flags, working of serial port for data transmission and reception.		square wave on P1.2 using timer 0 in mode 1 to generate delay. 2. Observe the square wave of the above program on CRO by downloading the program to the micro-controller kit. Interfacing experiments
13	3,4	5,7	1 Interfacing 8051 to Multiplexed seven-segment display with assembly/C program. 2. Interfacing 8051 to ADC 0804, waveform generation using DAC 0808 with assembly/C program. 3. Interfacing 8051 to DC motor, Stepper motor with assembly /C program.		1. Program to control direction and speed of a stepper motor/ dc motor. Study the data sheets of stepper motor/dc motor. 2. Program to control traffic lights OR Program to generate sine/ rectangular / triangular waveforms.
Total in hours			39	13	52

CONTENTS

Sl.No.	Chapters	Page Nos.
Week 1.	**Introduction to C Programming**	**1**
1.1	Introduction to C	
1.1.1	Features of C Language	
1.1.2	Application of C	
1.1.3	Basic Structure of C Program	
1.1.4	Compilation of a C Program	
1.1.5	C Character Set	
1.2	C Tokens	
1.2.1	Keywords	
1.2.2	Identifiers	
1.2.3	Constants/Literals	
1.2.4	Variables	
1.3	Data Types	
1.3.1	Primitive or Built-in Data Types	
1.3.2	int : Integer Type	
1.3.3	float : Floating Point Type	
1.3.4	char : Character Type	
1.3.5	double : Double Precision Floating Point Type	
1.3.6	void : Non-Specific Type	
1.3.7	Data Type Modifiers	
*	Lab Programs	
*	Review Questions	

Week 2.	**C Operators & I/O Statements**	**39**
2.1	C Operators	
2.1.1	Unary Operators	
2.1.2	Arithmetic Operators	
2.1.3	Relational Operators	
2.1.4	Logical Operators	
2.1.5	Bitwise Operators	
2.1.6	Ternary (Conditional) Operator(?:)	
2.2	Miscellaneous Operators	
2.2.1	Assignment Operators	
2.2.2	Special Operators	
2.2.3	Operator Precedence and Associativity in C	
2.3	Input and Output Functions (I/O Functions)	
2.3.1	Formatted Input-The Scanf( ) Function	
2.3.2	Reading and Integer Input	
2.3.3	Reading a Floating-Point (Real Number) Input	
2.3.4	Reading a Character Input	
2.3.5	Formatted Output-The Printf( ) Function	
2.3.6	Integer Output	
2.3.7	Floating Point Output	
2.3.8	Character Output	
2.3.9	Unformatted Input Functions	
2.3.10	The getchar( ) Function	
2.3.11	The gets( ) Function	
2.3.12	Unformatted Output Functions	
2.3.13	The putchar( ) Function	
2.3.14	The puts( ) Function	
2.3.15	Preprocessor Directives	
2.3.16	C Library Functions	
*	Lab Programs	
*	Review Questions	

Week 3. Algorithms, Flowcharts and Control Structures 85

3.1 Algorithm

3.1.1 Characteristics of Algorithm

3.1.2 Algorithm Notations

3.1.3 Examples of Algorithms

3.1.4 Flowchart

3.1.5 Examples of Flowcharts

3.2 Control Statements

3.2.1 Conditional Control Statements

3.2.2 if statement

3.2.3 if else statement

3.3 Nested if statement

3.3.1 Switch Statement

* Lab Programs

* Review Questions

Week 4. Looping and Arrays 113

4.1 Looping

4.1.1 while Statement

4.1.2 do while Statement

4.1.3 for Loop

4.2 Nested for Loops

4.2.1 Unconditional Control Statement

4.2.2 The "break" Statement

4.2.3 The "continue" Statement

4.2.4 The "goto" Statements

4.3 Arrays

4.3.1 Classification of an Arrays

4.3.2 One dimensional Arrays

4.3.3 Declaration of One-dimensional Array

4.3.4 Initialization of One-dimensional Array

* Lab Programs

* Review Questions

Week 5. Strings, Functions & Structures 145

5.1 Strings

5.1.1 Declaring and Initializing String Variables

5.1.2 Reading and Writing Strings

5.1.3 Two-dimensional Array

5.1.4 Syntax to Declare Two-Dimensional Array

5.1.5 Initialization of 2D Array

5.1.6 C String Functions

5.2 Functions in C

5.2.1 How to call C Functions in a Program

5.2.2 Recursion

5.3 Pointers in C

5.3.1 Accessing the Address of Variables

5.3.2 Declaring Pointers (Creating Pointers)

5.3.3 Assigning Address to Pointer

5.3.4 Accessing Variable Value using Pointer

5.3.5 Structure in C

5.3.6 How to Creae Structure ?
5.3.7 Creating and using Structure Variables
5.3.8 Memory Allocation of Structure
* Lab Programs
* Review Questions

Week 6. 8051 Architecture **183**
6.1 Introduction to Concepts of Embedded Systems
6.1.1 Characteristics of an Embedded System
6.1.2 Applications
6.1.3 Microprocessor
6.1.4 Microcontroller
6.1.5 Comparison between Microprocessor and Microcontroller
6.2 Selection of Microcontroller
6.2.1 Features of 8051 Microcontroller
6.3 Architecture/Block Diagram Description
6.3.1 Program Status Word (PSW) Register (Flag Register)
6.3.2 Special Function Registers(SFRs)
* Lab Exercises
* Review Questions

Week 7. 8051 Architecture Continued **225**
7.1 Basic Registers in 8051
7.1.1 Pin Description of 8051
7.2 Memory Organization of 8051

7.2.1 Internal RAM Organization
7.2.2 Internal ROM Organization
7.2.3 Stack & Stack Pointer
7.3 Interfacing External Data & Code Memory
7.3.1 Interfacing External Program (Code) Memory
7.3.2 Interfacing External Data (RAM) Memory
* Lab Programs
* Review Questions

Week 8. 8051 - Instruction Set **247**
8.1 Addressing Modes of 8051
8.2 8051 Instruction Set
8.2.1 Data Transfer Instructions
8.3 Arithmetic Instructions
8.3.1 Logical Instructions
* Lab Exercises
* Review Questions

Week 9. Instruction Set Continued & Introduction to Embedded C **295**
9.1 Bit Level Logical Instruction
9.1.1 Program Branching and Machine Control Instructions
9.1.2 Unconditional Branching
9.1.3 Conditional Branching
9.1.4 CALL
9.2 Introduction to Embedded C
9.2.1 Pros & Cons of Embedded C

9.3 General Structure of Embedded C Program
9.3.1 Data Types in 8051 C Programming
* Lab Programs
* Review Questions

Week 10. Embedded C Continued... 333

10.1 Memory Types and Memory Models
10.1.1 Pointers
10.1.2 Types
10.2 Time Delays
10.3 Arithmetic & Logical Operators in 8051 C
* Simple Programs
* Lab Programs
* Review Questions

Week 11. I/O Ports & Interrupt Programming 353

11.1 Features of I/O Ports
11.1.1 Interfacing LED
11.2 Interrupts of 8051
11.2.1 Comparison of Interrupt Method and Polling
11.2.2 Steps in Executing an Interrupt
11.2.3 Bit Structure of IE
11.2.4 Bit Structure of IP (Interrupt Priority) Register
11.3 Examples Showing Enabling, Disabling and Priority Setting of Interrupts
* Lab Programs
* Review Questions

Week 12.Timers/Counters & Serial Communication 375

12.1 Timer/Counters
12.1.1 Timer 0 Register
12.1.2 Timer 1 Register
12.1.3 TMOD Register
12.1.4 TCON Register
12.1.5 Model Operation of Timers
12.1.6 Mode 2 (8-bit autoreload) Timer Programming
12.1.7 Programming Mode 2 - Timer 1
12.2 Time Delay Generation Calculation
12.3 8051 - Serial Communication
12.3.1 SBUF Register (Serial Buffer Register)
12.3.2 SCON Register (Serial Control Register)
12.3.3 Working of Serial Port
* Lab Programs
* Review Questions

Week 13. Interfacing the 8051 405

13.1 Seven Segment Display
13.2 Interfacing ADC 0804 to 8051
13.2.1 Interfacing DAC 0808 to 8051
13.3 Interfacing Stepper Motor to 8051
13.3.1 Relay and DC Motor
* Lab Programs
* Review Questions

Week 1

Introduction to C Programming

Introduction to C Programming

1.1 Introduction to C

1.1.1 Features of C Language

1.1.2 Application of C

1.1.3 Basic Structure of C Program

1.1.4 Compilation of a C Program

1.1.5 C Character Set

1.2 C Tokens

1.2.1 Keywords

1.2.2 Identifiers

1.2.3 Constants/Literals

1.2.4 Variables

1.3 Data Types

1.3.1 Primitive or Built-in Data Types

1.3.2 int : Integer Type

1.3.3 float : Floating Point Type

1.3.4 char : Character Type

1.3.5 double : Double Precision Floating Point Type

1.3.6 void : Non-Specific Type

1.3.7 Data Type Modifiers

* Lab Programs

* Review Questions

What is an instruction & program ?

- An Instruction is a command given to the computer to perform a specified operation (function) on given data.
- Programs are simply set of instructions or computer code. Programs are given to computer- by-computer programmer in specific programming language.

What is Computer Language?

- Generally, we use languages like English, Hindi, etc., to make communication between two persons. That means when we want to make communication between two persons, we need a language through which persons can express their feelings. Similarly, when we want to make communication between user and computer or between two or more computers, we need a language through which user can give information to the computer and vice versa. When a user wants to give any instruction to the computer the user needs a specific language and that language is known as a computer language.
- Thus, Computer languages are the languages through which the user can communicate with the computer by writing program instructions.

1.1 Introduction to C:

- C is a programming language developed at AT & T's Bell Laboratories of USA in 1972 by **Dennis Ritchie.(Known as Father of C language)**
- C is a successor of 'Basic Combined Programming Language' (BCPL) called B language.
- C is structured, portable, reusable, general purpose programming language

- C was invented basically to write UNIX operating system.
- Any programming Language can be divided in to two categories.

1. Problem oriented (High level language)
2. Machine oriented (Low level language) But C is considered as a Middle level Language.

- Many later languages have borrowed syntax/features directly or indirectly from the C language. Like syntax of Java, PHP, JavaScript, and many other languages are mainly based on the C language. C++ is nearly a superset of C language (Few programs may compile in C, but not in C++).

1.1.1 Features of C Language:

1. C is a general-purpose, structured programming language
2. C is a middle level language. (Although, C is **intended to do low-level programming.** It is used to develop system applications such as kernel, driver, etc. It **also supports the features of a high-level language**. That is why it is known as mid-level language.)
3. It helps in development of system & application softwares.
4. It produces efficient and fast running programs.
5. No rigid format. Any number of statements can be typed in a single line.
6. **Portability:** any c program can be run on different machines with little or no modifications.

7. C is a robust language supports rich set of built-in functions and operators.
8. Limited number of key words or reserve words (32),
9. C is extensible because it **can easily adopt new features.**
10. C supports rich set of data types and has got ability to extend itself by adding functions to its library.

1.1.2 Applications of C:

Because of its portability and efficiency, c is used to develop the system as well as application software as listed below ;

System software	Applications software
1. Operating systems	1. Data Base Management Systems (DBMS)
2. Interpreter	2. Graphics packages
3. Compilers	3. Spread sheets
4. Assemblers	4. CAD/CAM applications
5. Editors	5. Word processors
6. Loaders	6. Office automation tools
7. Linkers	7. Scientific & engineering applications

1.1.3 Basic structure of c program:

The basic structure of a c program is as shown in the figure below ;

```
preprocessor statements
global declarations main()
{
declarations;
statements;
Comments
}
user defined functions
```

Preprocessor statements:

- These statements begins with # symbol , and are also called as preprocessor directives.
- These statements direct the C preprocessor to include header files (files with extension ".**h**") and also symbolic constants in a c program.
- Some of the preprocessor statements are given below;

#include<stdio.h>	;	for standard input/output functions
#include"Test.h"	;	for the inclusion of header file Test
#define PI 3.142	;	for defining symbolic constant, PI=3.142

Global declarations:

- Variables or functions whose existence is known in the main() function and other user defined functions are called as global variables (of functions) and their declaration is called as global declarations.
- These declarations should be made before main()

The main() function:

- As the name itself indicates this is the main function of every C program. The execution of a C program starts with main() function.
- The function main() should be written in lowercase letters and should not be terminated by semicolon. It calls other library functions and user defined functions.
- Imp Note: there must be one and only main() function in every C program.

Braces:

- Every C program uses a pair of curly braces ({,}). The left brace ({) indicates the beginning of main() function. On the other hand right brace (}) indicates the end of the main() function.
- The braces also be used to indicate the beginning and end of the other user defined functions and the compound statements.

Declarations:

The declaration is a part of c program where all variables, arrays, functions etc. used in the program are declared and

may initialized (assigned initial values) with their basic data types.

Statements:

- These are instructions to the computer to perform some specific operations.
- They may be input/output statements, arithmetic statements, control statements and other statements. They might also include comments.

Comments:

- We can add comments in our program to describe what we are doing in the program. These comments are ignored by the compiler and are not executed.
- To add a single line comment, start it by adding two forward slashes // followed by the comment.
- To add multiline comment, enclose it between /*.....* / , just like in the program above.

User defined functions:

- These are sub programs. The user defined functions contain set of statements to perform a specific task.
- These are written by the user, hence the name user defined functions.
- They may be written before or after main() function.

Sample C program:

```
#include<stdio.h>
void main()
{
printf("Welcome to Electronics\n");
}
```

Output:

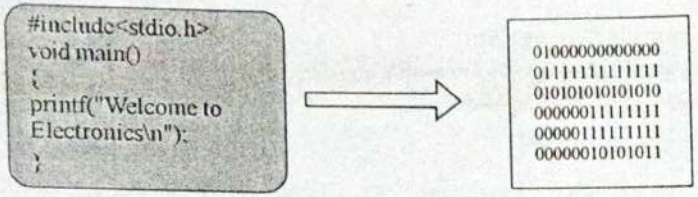
Welcome to Electronics

- The first line tells the compiler to include the standard input/output header file to perform the reading and printing of data.
- Second line is the **main()** ,the main() function of a c program. This is the starting point of the execution.
- The body of the function contains only one statement i.e. printf("Welcome to Electronics\n");.

When this statement is taken for execution **main()** calls the **printf()** function ,which is included in <stdio.h>. Then printf() prints **Welcome to Electronics** on the computer screen

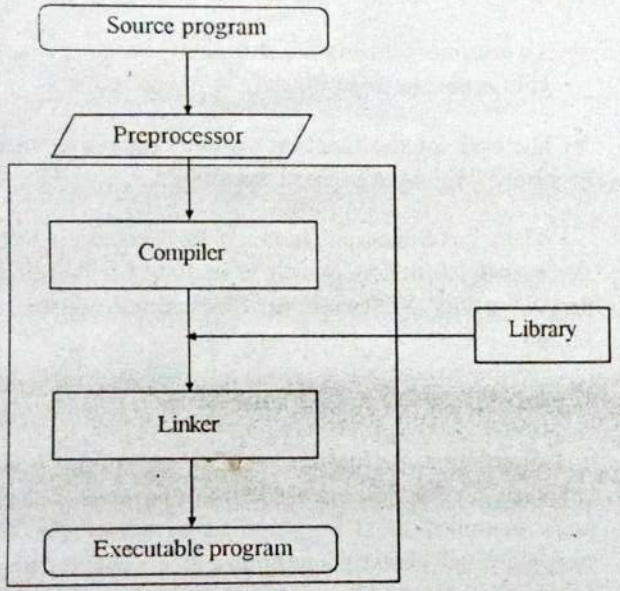
1.1.4 Compilation of a C Program:

Compiling a C program means translating it into machine language. C compilers are used for this purpose. The C program to be compiled must be typed using editor. An editor is a program which allows the programmer to write the program and edit it.



The C compilation process converts the source code taken as input into the object code or machine code. The compilation process can be divided into four steps, i.e.,

1. Pre-processing,
2. Compiling,
3. Assembling,
4. Linking.



Preprocessing:

Preprocessing is the first step. The preprocessor obeys commands that begin with # (known as directives) by:

- removing comments
- expanding included files
- Replacing the symbolic constants with values

➢ If you included a header file such as #include <stdio.h>, it will look for the stdio.h file and copy the header file into the source code file.

➢ The preprocessor also replaces symbolic constants defined using #define with their values.

Compiling:

Compiling is the second step. It takes the output of the pre-processor and generates assembly language, an intermediate human readable language, specific to the target processor.

Assembling:

- Assembly is the third step of compilation. The assembler will convert the assembly code into pure binary code or machine code (zeros and ones). This code is also known as object code.
- The object file extension in DOS is 'obj'. If the source file name is sample.c, the object file name would be 'sample.obj'.

Linking:

- Linking is the final step of compilation. The linker merges all the object code from multiple modules into a single one.

If we are using a function from libraries, linker will link our code with that library function code.

- The linker's output is the executable script. The executable file name is similar to the source file, which varies only in its extensions. In DOS, the executable file extension is '.exe',

1.1.5 C Character Set

➢ As every language contains a set of characters used to construct words, statements, etc., C language also has a set of characters which include alphabets, digits, and special symbols. C language supports a total of 256 characters.

➢ Every C program contains statements. These statements are constructed using words and these words are constructed using characters from C character set. C language character set contains the following set of characters...

❖ Alphabets

❖ Digits

❖ Special Symbols

Alphabets

C language supports all the alphabets from the English language. Lower- and upper-case letters together support 52 alphabets.

lower case letters - a to z
UPPER CASE LETTERS - A to Z

Digits

C language supports 10 digits which are used to construct numerical values in C language.

Digits - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Special Symbols

C language supports a rich set of special symbols that include symbols to perform mathematical operations, to check conditions, white spaces, backspaces, and other special symbols.

Special Symbols - ~ @ # $ % ^ & * ( ) _ - + = { } [ ] ; ' " / ? , > , < \ | tab newline space NULL bell backspace vertical tab etc.,

Imp Note : Every character in C language has its equivalent ASCII (American Standard Code for Information Interchange) value.

1.2 C Tokens:

Tokens are the smallest individual unit of a program. They are the building blocks of a program. For example, lets us consider an English paragraph:

"I love programming in C."

The above sentence is made with various basic elements such as - Alphabets (a-z, A-Z), punctuation marks and blank spaces. These are tokens of the given paragraph. Similarly, all basic elements of a program collectively known as Tokens.

There are five types of tokens.

1. Keywords
2. Literals
3. Identifiers
4. Operators
5. Separators

Consider the below C statement -

int num = 5 + 4;

Here,

- int is keyword.
- num is identifier.
- = and + are operators.
- 5 and 4 are literals/constants
- ; and white spaces are separators.

1.2.1 Keywords:

- Keywords are the reserved words whose meaning is predefined by the programming language specification.
- They convey some special meaning in programming and we must not use them for other purposes.
- For example - int is a C keyword that defines integer data type. You must not use int for other purposes.
- There is total 32 keywords in C as per ANSI standards.

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

Note: Apart from the 32 standard C keyword, different compilers can have a greater number of keywords.

1.2.2 Identifiers:

- In C language identifiers are the names given to variables, constants, functions and user- define data.
- In real life, you may think name given to a person or object as an identifier. In programming, we use identifiers to identify a programming element uniquely.
- These identifiers are defined against a set of rules.

1. Identifier must not be a keyword. For example - you cannot use int, return etc. as an identifier.
2. Identifiers must begin with an alphabet a-z A-Z or underscore _ symbol.
3. Identifiers can contain digits 0-9.
4. Identifiers must not contain special character other than alphabets, digits and underscore symbol.
5. C is a case sensitive language hence you must be careful while naming an identifier. For example - num, NUM, Num all are different.

Examples of valid & invalid identifiers:

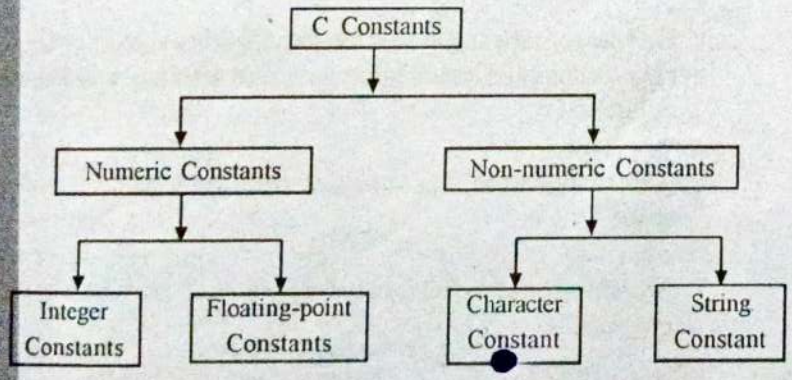
Valid Identifiers
num
_num
num1
_num1
_1num1
NUM
NUM_

Invalid	*Reasons for Invalid*
int	int is a keyword and must not be used as an identifier
1num	Identifier should begin with an alphabet or underscore symbol
num$	It must not contain any special symbol inthis case
first num	It must not contain spaces
date-of-birth	It must not contain any special symbol - in thiscase

Note : *We must give a meaningful name to an identifie. However, it is not a naming convention but considered as a good programming practice. A meaningful identifier increases program readability and clarity.*

1.2.3 Constants/Literals:

- Constant is a fixed value that does not change during the execution of the program.
- In C, constants are categorized based on type of the data. Following figure shows the different types of constants in C.



- **An integer constant** is a whole number. It is a sequence of digits without a decimal point. It is prefixed with optional plus or a minus sign. The general form of integer constant is

Sign	Digits

Where,

Sign → an optional plus sign for positive numbers and minus sign for negative numbers

Digits → a sequence of digits

Example: 500,2,-45,-164,+164 etc..

- **A floating point constant** is a number with a decimal point. It is defined as a sequence of digits preceded and followed by a decimal point. They also have prefixed plus or minus sign. The general form of floating point or real constant is

Sign	Integer part	Decimal point	Fractional part

Where,

Sign → an optional plus or minus sign

Integer part → sequence of digits before decimal point

Decimal point → period symbol (.)

Fractional part → sequence of digits after decimal point

Example: 12.56, -164.5 ,+0.123 etc..

❖ **Character constant:**

A character constant is a single character enclosed within a pair of apostrophes

Example: '1','a', '?' ,' ' (blank space)

❖ **String constant:**

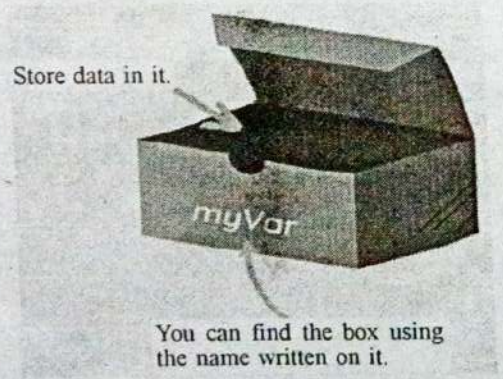
A string constant is a sequence of characters enclosed with in a pair of double quotes.

Example: "HI", "GPT", "DTE","2021", " "(blank space)

1.2.4 Variables:

➢ The quantity that changes during the execution of a program is called a variable.

➢ The variables are the names given to identify the specific program elements. Therefore, variables are also called as **identifiers.**



➢ The variables represent a particular memory location where data can be stored. or Variable is like a container (storage space) with a name in which you can store data.

➢ The C compiler allocates a storage space, depending upon the datatype of the variable.

➢ We can change the value stored in a variable as many times as we want, but we should make sure we store the correct data type value.

Rules for forming a variable name:

1. A variable can have alphabets, digits, and underscore.
2. A variable name can start with the alphabet, and underscore only. It can't start with a digit.
3. No special characters except underscore (_) is allowed in forming a variable name.
4. A variable name must not be any reserved word or keyword, e.g., int, float, etc.
5. Both uppercase and lowercase variables are significant in C (The C compiler treats uppercase and lowercase variables differently.).
6. You cannot have two variables with same name in same scope.
7. Variable must be declared before it is used anywhere in the program.

Examples of valid and invalid identifiers:

Valid variable names	Invalid variable names
Marks	8ab
total_marks	total marks
gross_salary	gross-salary

Declaration of a variable:

- All variables must be declared before they are used in a C program.
- The purpose of declaring a variable is to reserve the amount of memory required for these variables.
- **General syntax:**

Data type	Varlist	Semicolon

Where,

Data type → is a basic data type as int,floar,char,double

Varlist→ One or more variables of same data type, separated by commas

Semicolon → a delimiter(separator) of this declaration

Examples:

1. int sum;
2. float a,b,c;
3. char name;
4. double percentage [20];

Assigning values to variables:

- We know that the variables represent some memory location, where the data is stored & initially the allocated memory contains garbage value.
- To avoid this, values are assigned to variables. The process of giving values to variables is called assignment of values.
- The assignment operator "=" is used to assign a value to a variable.
- General syntax is

Variable name	=	value	;

Where,

Variable name → represents memory location where value is to be stored

= → Assignment operator

Value → is a constant or variable

There are 2 methods of assigning values to variables.

1. In the first method initial values can be assigned to variables within declaration section. This is called Initialization.

   Example: *int x=1;*
   float sum=0.0;

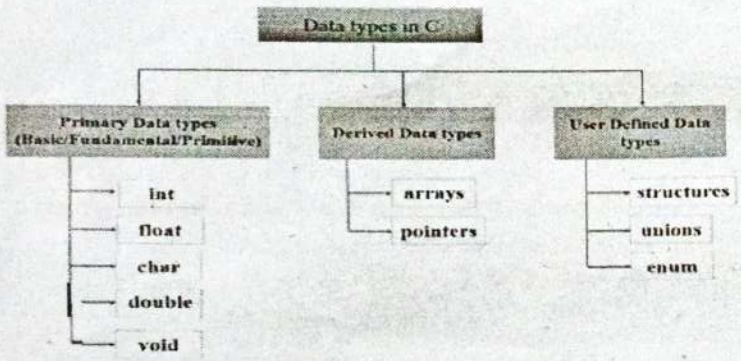
2. In the second method initial values can be assigned to variables in the executable part of the program. This method doesn't require data type to be specified.

Example: *x=1;*
sum=0.0

1.3 Data Types :

Data types indicates the type of the data that a variable or constant can hold. The data may be numeric or non-numeric in nature. In C, data types are classified into

1. Primitive or built-in data types
2. Derived data types
3. User defined data types



1.3.1 Primitive or Built-in Data Types:

They are basic or primitive data types and designate a single value. There are four fundamentals built in data types in C which are given below.

1. Integer
2. Real or floating-point number
3. Double precision real number
4. Characters

In C, there is one and only non-specific data type called void. It does not specify anything. The table shows built in data types, keywords and their size in bytes.

Types	Keyword	Size (in bytes)
Integer	int	2
Real (Floating point)	float	4
Double precision	double	8
Character	char	1
Non specific	void	0

1.3.2 *int: Integer Type*

- **int** is a key word used to indicate an **integer** number. Any integer number is a sequence of digits without a decimal point.
- The range of int depends on word length of a computer. Here word length of a computer means the number of bits that can be accessed at a time by the processor.
- In general, if a computer is n bit computer, then range of int is given as,

  -2^{n-1} **to** $+2^{n-1} -1$

  So for 8 bit computer the range of int is , -128 to +127

  And for 16 bit computer the range of int is , -32768 to +32767.
- Some of the valid and invalid integers are listed below

Valid	Invalid
1) -258	3,333 (comma not allowed)
2) 14504	-34.0 (decimal point)
3) 0	9999999(out of range)

1.3.3 *float: Floating Point Type* or real

- Float is a keyword used to indicate floating point number. The floating-point numbers are the same as real numbers.
- They are called as floating-point numbers because the decimal point is shifted either to the left or to the right of some digits during manipulations.
- These numbers may be expressed either in decimal (fixed) form or scientific notation. The scientific notation is also called as **mantissa exponent** notation.
- Some of the valid and invalid floating-point numbers are listed below-

Valid	Invalid
1) -258.238	0,0 (comma not allowed)
2) 2.58238 E+02	-2.2.2 (No two decimal points)
3) 0.0258238 E +04	9999999.9999(out of range)

In the above list of valid examples (1) is in decimal notation and examples (2) and (3) are in scientific notation. And the character E in the examples stands for Exponent.

- We can use mantissa exponent notation, when the given data is either too large or too small. For example, if the given number is **0.000000000012** then this number cannot be treated as ordinary floating-point number. This will be written as $\mathbf{1.2 \times 10^{-11}}$, which would be written in mantissa exponent form as $\mathbf{1.2E^{-11}}$

1.3.4 char: Character Type:

- char is a keyword used to indicate character type data. The data may be character constant or string constant.

- A character constant may be defined as any single character enclosed within a pair of apostrophes.

  Examples: 'A', 's', '1', '#', ' '(blank space) etc..

- A character type data takes 8-bits for storage. The character constant may be signed or unsigned. The signed characters have values in the range -128 to +127. On the other hand, the unsigned characters have the values between 0 to 255.
- Each character constant has an integer value. This value is given by the ASCII (American Standard Code for Information Interchange) character set. Some of character constants and their ASCII values are given in the table below

Character constant	ASCII value
'A'	65
'B'	66
'Z'	99
'a'	97
'z'	122
'0'	48
'9'	57
'&'	38
'='	61
'{'	123

- A string constant is defined as a sequence of printable ASCII characters place between a pair of double quotes.

Examples: "GPT"," INDIA"," Mudhol"," shiralkoppa","164" etc..

1.3.5 *double: Double Precision Floating Point Type*

- double is a keyword used to indicate double precision floating point number. The precision is associated with the accuracy of the data.
- It is used whenever more accuracy is required in representing the floating point numbers.
- A double value is normal float value only, but the number of significant digits that are stored after the decimal point is double than that of float.
- The float usually stores a maximum of 6 digits after the decimal point, where as double stores 16 significant digits after the decimal point. **Example:** 3.142857142857143

1.3.6 *void: Non-specific Type:*

- The void is an empty data type that has no value. This can be used in functions and pointers.
- It is used in functions to specify the return value or the arguments.

1.3.7 Data Type Modifiers:

In c language Data Type Modifiers are keywords used to change the current properties of data type.

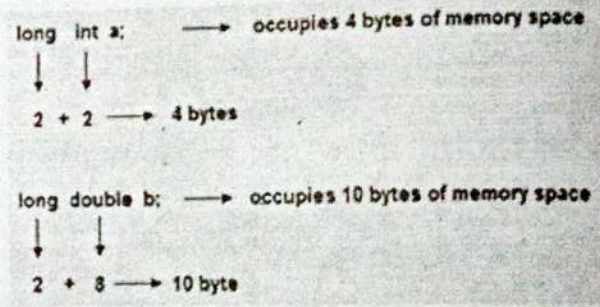
Modifiers are prefixed with basic data types to modify (either **increase** or **decrease**) the amount of **storage space** and change the **storage type** (can store only **positive** or both **positive & negative values**) allocated to a variable.

Data type modifiers are classified into following types.

- long
- short
- unsigned
- signed

long:

This can be used to increased size of the current data type to 2 more bytes, which can be applied on int or double data types. For example, int occupy 2 bytes of memory if we use long with integer variable then it occupies 4 bytes of memory.



short ;

In general, int data type occupies different memory spaces for a different operating system; to allocate fixed memory space **short** keyword can be used.

Syntax

short int a; --> occupies 2 bytes of memory space in every operating system.

unsigned

The unsigned prefix restricts only positive numbers to be contained in it.

Syntax

unsigned int a =100; // right

unsigned int a=-100; // wrong

Signed

This keyword accepts both negative or positive value and this is default properties or data type modifiers for every data type.

Example

int a=10; // right

int a=-10;// wrong

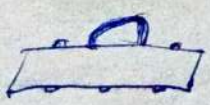
signed int a=10; // right

signed inta=-10; // right

From the above examples, one can say that it's a default property of every data type. So signed keyword is optional.

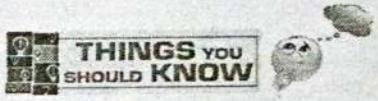
Note: The following table shows the applicability of qualifiers to basic types.

Sr.No.	Data Type	Qualifier
1.	char	signed,unsigned.
2.	int	short,long,signed,unsigned.
3.	float	No qualifier.
4.	double	long.
5.	void	No qualifier.



Summary:

Type	Size (bits)	Size (bytes)	Range
char	8	1	-128 to 127
unsigned char	8	1	0 to 255
int	16	2	-2^{15} to $2^{15}-1$
unsigned int	16	2	0 to $2^{16}-1$
short int	8	1	-128 to 127
unsigned short int	8	1	0 to 255
long int	32	4	-2^{31} to $2^{31}-1$
unsigned long int	32	4	0 to $2^{32}-1$
float	32	4	3.4E-38 to 3.4E+38
double	64	8	1.7E-308 to 1.7E+308
long double	80	10	3.4E-4932 to 1.1E+4932



Delimiters:

- The delimiters limit the boundary between the basic elements of a program. Hence the name delimiters. These are also called as separators.
- They separate constants, variables and statements. The comma, semicolon, apostrophes (single quotes), double quotes and the blank spaces are the most commonly used delimiters.

Backslash character constants in C:

- There are some characters which have special meaning in C language.
- They should be preceded by backslash symbol to make use of special function of them.

➢ Given below is the list of special characters and their purpose.

Backslash_character	Meaning
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\"	Double quote
\'	Single quote
\\	Backslash
\v	Vertical tab
\a	Alert or bell
\?	Question mark
\N	Octal constant (N is an octal constant)
\XN	Hexadecimal constant (N-hex.dcml cnst)

Lab Programs

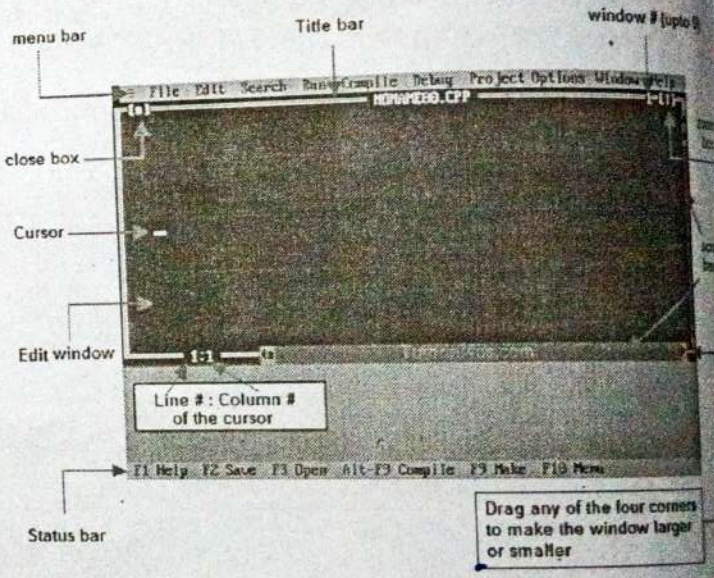
Experiment 1 a:

Familiarisation of TURBO2 C.

Turbo C:

The Borland Turbo C / TCLite IDE is a software package which provides an integrated development environment to assist programmers in developing C or C++ programs. In this, IDE integrates editor, compiler, linker, debugger, and other necessary development tools into a single environment.

Typical Turbo C IDE screen look like in the below figure:



When IDE screen appears, the menu bar is activated contains various menus such as;

- **File :** This menu contains group of commands used for save, edit, print program, exit from Turbo C editor etc.
- **Edit:** This menu contains group of commands used for editing C program source code. Example Copy, Cut, Paste, Undo etc.
- **Search:** This menu contains group of commands used for searching specific word as well as replacing it with another one.
- **Run:** This menu contains group of commands used for running C program.
- **Compile:** This menu contains group of commands used for compiling C program.
- **Debug:** This menu contains group of commands used for debugging C program.
- **Project:** This menu contains group of commands used for opening, closing and creating projects.
- **Options:** This menu contains group of commands used for configuring IDE of Turbo C and setting up directories etc.
- **Windows:** This menu contains group of commands used for opening, closing various windows of IDE.
- **Help**: This menu is used to get help about specific topic of C language. Similarly, to get help about a specific keyword or identifier of C.

Shortcut keys Related to TC Editor

Alt + x : Close TC Editor.

Ctrl + f9 : Run C Program.

Alt + f9 : Compile C Code.

Alt + f5 : To view the output on the screen

Alt + Enter : Get Full Screen or Half Screen TC Editor

Ctrl + y : Delete complete above line of cursor

Shift + > : Select Line of Code.

Ctrl + Insert : Copy.

Shift + Insert : Paste.

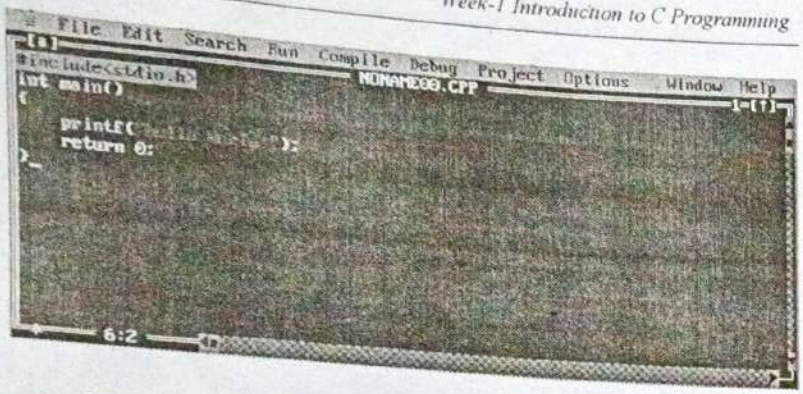
Shift + Delete : Delete.

❖ **Step by step procedure to edit and run a simple program:**

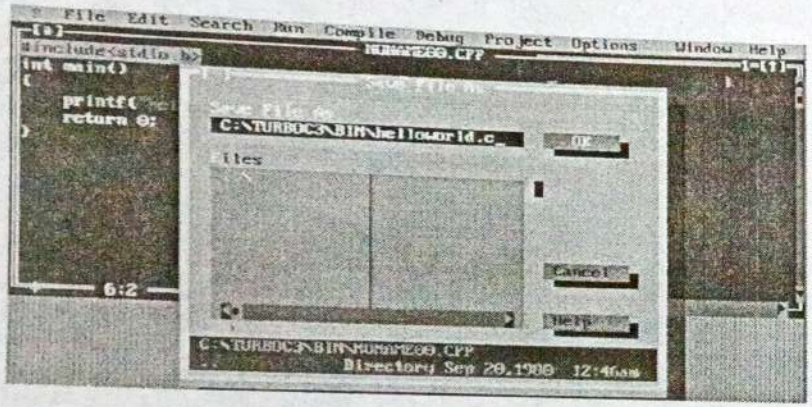
Step 1: Locate the TC.exe file and open it. You will find it at location C:\TC\BIN\.

Step 2: File > New (as shown in above picture) and then write your C program

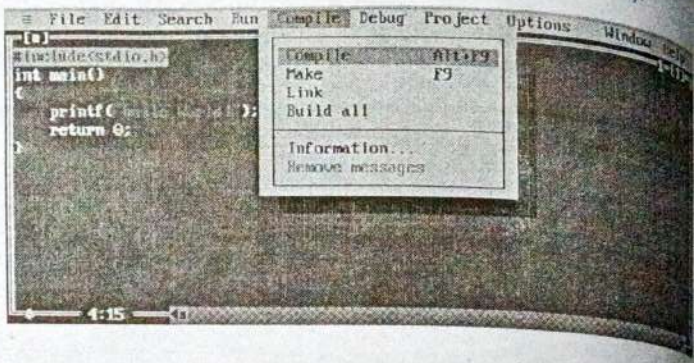
```
#include<stdio.h>
int main()
{
    printf("hello World!");
    return 0;
}
```



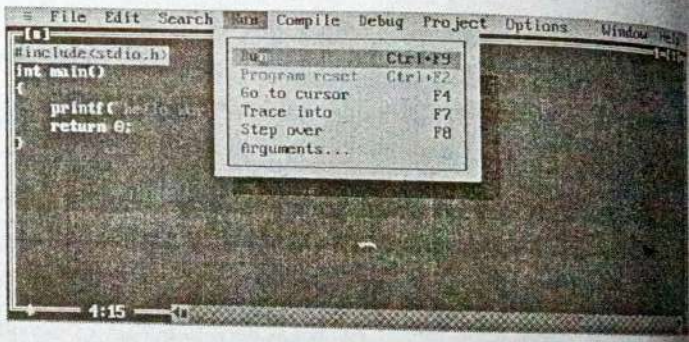
Step 3: Save the program using **F2** (OR file > Save), remember the extension should be **".c"**. In the below screenshot I have given the name as **helloworld.c**.



Step 4: Compile the program using Alt + F9 **OR** Compile > Compile (as shown in the below screenshot).



Step 5: Press Ctrl + F9 to Run (or select Run > Run in menu bar) the C program.



Step 6: Alt+F5 to view the output of the program at the output screen.

Experiment 1b:

Programs to illustrate the use of different data types and verify their memory size.

The storage representation and machine instructions differ from machine to machine. "**sizeof** "operator can use to get the exact size of a type or a variable in bytes on a particular platform.

Program:

```
#include<stdio.h>
#include<conio.h> // headerfile that consist, the definitions for getch() & clrscr() functions
void main()
{
clrscr(); //command that clears previous output history
printf("storage size of int is: %d\n",sizeof(int));
printf("storage size of float is: %d\n",sizeof(float));
printf("storage size of char is: %d\n",sizeof(char));
printf("storage size of double is: %d\n",sizeof(double));
printf("storage size of long int is: %d\n",sizeof(long int));
printf("storage size of short int is: %d\n",sizeof(short int));
printf("storage size of long double is: %d\n",sizeof(long double));
getch(); //command that holds the output on the screen until a character press from keyboard
}
```


Output:

```
storage size of int is: 2
storage size of float is: 4
storage size of char is: 1
storage size of double is: 8
storage size of long int is: 4
storage size of short int is: 2
storage size of long double is: 10
```

Review Questions :

1. What is a token ? Give example for each type of token.
2. What is a constant? Explain different types of C constants.
3. What is variable? Write the rules and guidelines for naming variable ?
4. Define the term Identifier. Give an example.
5. List the basic data types with byte specification.
6. List the important features of C language.
7. Explain the basic data types of C language.
8. Describe the basic structure of C program.
9. Distinguish between keywords and identifiers.
10. How do you make comments in C programming?
11. Why and when do we use the #define directive?
12. Why and when do we use #include directive?
13. Differentiate between constants and variables?
14. Find errors if any in the following program.

```
#include(stdio.h)
main
{
printf("Hello C\n");
}
```

15. What is the need of backslash constants ? Explain ?
16. Discuss the basic steps involved in compilation of a C program.

Week 2

C Operators & I/O Statements

C Operators & I/O Statements

2.1 C Operators
- 2.1.1 Unary Operators
- 2.1.2 Arithmetic Operators
- 2.1.3 Relational Operators
- 2.1.4 Logical Operators
- 2.1.5 Bitwise Operators
- 2.1.6 Ternary (Conditional) Operator(?:)

2.2 Miscellaneous Operators
- 2.2.1 Assignment Operators
- 2.2.2 Special Operators
- 2.2.3 Operator Precedence and Associativity in C

2.3 Input and Output Functions (I/O Functions)
- 2.3.1 Formatted Input-The Scanf( ) Function
- 2.3.2 Reading and Integer Input
- 2.3.3 Reading a Floating-Point (Real Number) Input
- 2.3.4 Reading a Character Input
- 2.3.5 Formatted Output-The Printf( ) Function
- 2.3.6 Integer Output
- 2.3.7 Floating Point Output
- 2.3.8 Character Output
- 2.3.9 Unformatted Input Functions
- 2.3.10 The getchar( ) Function
- 2.3.11 The gets( ) Function
- 2.3.12 Unformatted Output Functions
- 2.3.13 The putchar( ) Function
- 2.3.14 The puts( ) Function
- 2.3.15 Preprocessor Directives
- 2.3.16 C Library Functions

* Lab Programs
* Review Questions

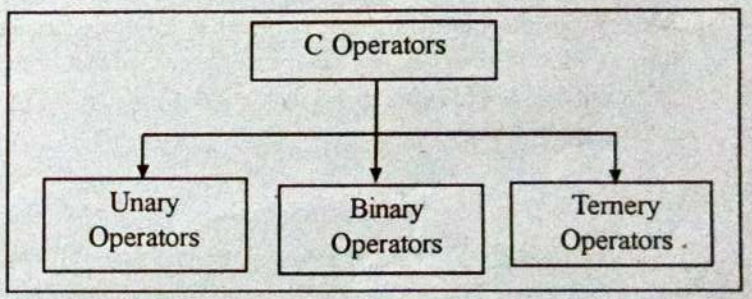
2.1 C Operators

Operators in C are tools or symbols, which are used to perform a specific operation on data. Operations are performed on operands.

What is an Operand?

An operand is a value on which any operator works. For example, when we say 4+5, here, numbers 4 and 5 are operands whereas + is an operator.

Operators can be classified into three broad categories, according to the number of operands used. Which are as follows:



2.1.1 Unary Operators:

An operator that acts upon only one operand is known as a unary operator.The C unary operators are

1. Unary minus
2. Logical not operator
3. Bitwise complementation
4. Increment & Decrement Operator
5. Size of() operator

Unary minus:

Any positive operand (value) associated with unary minus operator get its value changed to negative.

Example: Let a=3 and b=4

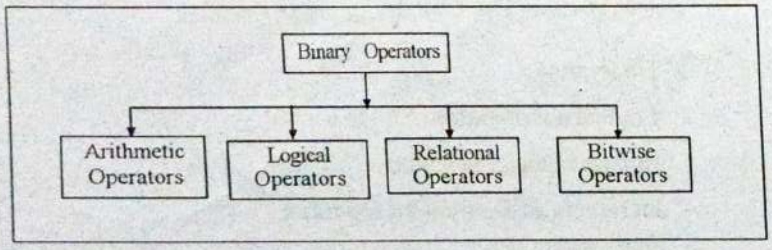
$$C = a+(-b)$$
$$= 3+(-4)$$
$$= -1$$

Here, c = -1, because b was initially a positive integer variable, when operated by unary minus get its value changed to negative.

The **Logical Not operator is discussed in logical operators, **bitwise complementation** is discussed in bitwise operators, **increment & decrement operators** are discussed in arithmetic operators & size of () operator is discussed in special operators in the later session.

Binary Operators:

These operators act upon two operands. Hence the name binary operators. The binary operators are further classified into four categories as shown in the figure.



2.1.2 Arithmetic Operators:

- The arithmetic operators are the symbols that are used to perform basic mathematical operations like addition, subtraction, multiplication, division and percentage modulo.
- The following table provides information about arithmetic operators.

Operator	Meaning	Example
+	Addition	10 + 5 = 15
-	Subtraction	10 - 5 = 5
*	Multiplication	10 * 5 = 50
/	Division	10 / 5 = 2
%(modulus)	Remainder of the Division	5 % 2 = 1
++	Increment	Let a=5; a++; ⇒ a = 6
--	Decrement	Let a=5; a--; ⇒ a = 4

Important Note:

- When both operands are of same type data then result is of the same type. But when operands are of different data types, then the result is promoted to the highest data type.
- Promotion of data type is as follows:

  char < short int < int < long int < float < double.
- The remainder operator (%) requires that both operands be integers and the second operand be nonzero. Similarly, the division operator (/) requires that the second operand be nonzero.

- Division of One integer quantity by another is referred to as integer division. This operation always results in a truncated quotient (i.e., the decimal portion of the quotient will be dropped).

- On the other hand, if a division operation is carried out with two floating-point numbers, or with one floating-point number and one integer, the result will be a floating-point quotient.

Increment & Decrement Operators:

- C programming has two operators increment ++ and decrement -- to change the value of an operand (constant or variable) by 1.

- These two operators are unary operators, meaning they only operate on a single operand.

- The increment operators **add one (1)** to the existing value of the operand and the decrement operator **subtracts one (1)** from the existing value of the operand.

- The increment and decrement operators are used in front of the operand (++a) or after the operand (a++). If it is used in front of the operand, we call it as pre-increment or pre-decrement and if it is used after the operand, we call it as post-increment or post- decrement.

Note:

1. Pre-Increment or Pre-Decrement

- In the case of pre-increment, the value of the variable is increased by one before the expression evaluation. In the case of pre-decrement, the value of the variable is decreased by one before the expression evaluation.

- That means, when we use pre-increment or pre-decrement, first the value of the variable is incremented or decremented by one, then the modified value is used in the expression evaluation.

2. Post-Increment or Post-Decrement

- In the case of post-increment, the value of the variable is increased by one after the expression evaluation. In the case of post-decrement, the value of the variable is decreased by one after the expression evaluation.

- That means, when we use post-increment or post-decrement, first the expression is evaluated with existing value, then the value of the variable is incremented or decremented by one.

2.1.3 *Relational Operators:*

- The relational operators are the symbols that are used to compare two values. That means the relational operators are used to check the relationship between two values. Every relational operator has two results TRUE or FALSE.

- In simple words, the relational operators are used to define conditions in a program. The following table provides information about relational operators.

Operator	Meaning	Example
<	Returns TRUE if the first value is smaller than second value otherwise returns FALSE	10 < 5 is FALSE
>	Returns TRUE if the first value is larger than second value otherwise returns FALSE	10 > 5 is TRUE
<=	Returns TRUE if the first value is smaller than or equal to second value otherwise returns FALSE	10 <= 5 is FALSE

>=	Returns TRUE if the first value is larger than or equal to second value otherwise returns FALSE	10 >= 5 is TRUE
==	Returns TRUE if both values are equal otherwise returns FALSE	10 == 5 is FALSE
!=	Returns TRUE if both values are not equal otherwise returns FALSE	10 != 5 is TRUE

2.1.4 *Logical Operators:*

➢ The logical operators are the symbols that are used to combine multiple conditions into one condition. The following table provides information about logical operators.

Operator	Meaning	Example
&&	**Logical AND** - Returns TRUE if all conditions are TRUE otherwise returns FALSE	10 < 5 && 12 > 10 is FALSE
||	**Logical OR** - Returns FALSE if all conditions are FALSE otherwise returns TRUE	10 < 5 || 12 > 10 is TRUE
!	**Logical NOT** - Returns TRUE if condition is FLASE and returns FALSE if it is TRUE	!(10 < 5 && 12 > 10) is TRUE

➢ **Logical AND** - Returns TRUE only if all conditions are TRUE, if any of the conditions is FALSE then complete condition becomes FALSE.

➢ **Logical OR** - Returns FALSE only if all conditions are FALSE, if any of the conditions is TRUE then complete condition becomes TRUE.

2.1.5 *Bitwise Operators:*

➢ The bitwise operators are used to perform bit-level operations in the c programming language. When we use the bitwise operators, the operations are performed based on the binary values.

➢ The following table describes all the bitwise operators in the C programming language.

Suppose a = 10, b = 5

a = 0000 1010 (8-bit binary representation of 10)

b = 0000 0101 (8-bit binary representation of 5)

Operator	Meaning	Example
&	**Bitwise AND** performs anding operation on two binary bits value. If both are 1 then will result is 1 otherwise 0.	0000 1010 & 0000 0101 0000 0000
|	**Bitwise OR** returns 1 if any of the two binary bits are 1 otherwise 0.	0000 1010 | 0000 0101 0000 1111
^	**Bitwise XOR** returns 1 if both the binary bits are different else returns 0.	0000 1010 ^ 0000 0101 0000 1111
~	**Bitwise COMPLEMENT** is a unary operator. It returns the complement of the binary value i.e., if the binary bit is 0 returns 1 else returns 0.	~ 0000 1010 1111 0101
<<	**Bitwise LEFT SHIFT** operator is unary operator. It shifts the binary bits to the left. It inserts a 0-bit value to the extreme right of the binary value.	0000 1010 << 2 = 0010 1000

>>	**Bitwise RIGHT SHIFT** operator is unary operator. It shifts the binary bits to the right. It inserts a 0-bit value to the extreme left of the binary value.	0000 1010 << 2 = 0000 0010

2.1.6 Ternary (Conditional) Operator (?:)

- The conditional operator is also called a ternary operator because it requires three operands. This operator is used for decision making.
- In this operator, first we verify a condition, then we perform one operation out of the two operations based on the condition result.
- If the condition is TRUE the first option is performed, if the condition is FALSE the second option is performed.
- The conditional operator is used with the following syntax.

<conditional-expression>? <true-expression> : <false-expression>

- ✶ It accepts three operands, conditional-expression, true-expression and false-expression.
- ✶ The conditional-expression is followed by ? symbol, followed by true-expression. true- expression is followed by : symbol and false-expression.
- ✶ If conditional-expression is true then true-expression gets evaluated otherwise false- expression. In no case both the expressions are evaluated.

Example

A = (10<15)?100:200;

Result ⇒ A value is 100

2.2 *Miscellaneous Operators:*

2.2.1 Assignment Operators:

- The assignment operators are used to assign right-hand side value (Rvalue) to the left-hand side variable (Lvalue).
- For example, if the value "10" is to be assigned for the variable "sum", it can be assigned as "sum = 10;"
- There are 2 categories of assignment operators in C language. They are,
  1. Simple assignment operator (Example: =)
  2. Compound assignment operators (Example: +=, -=, *=, /=, %=, &=, ^=)- The assignment operator used along with arithmetic/logical operators
- The following table describes all the assignment operators in the C programming language.

Operator	Meaning	Example
=	Assign the right-hand side value to left-hand side variable	A = 15
+=	Add both left and right-hand side values and store the result into left-hand side variable	A += 10 ⇒ A = A+10
-=	Subtract right-hand side value from left-hand side variable value and store the result into left-hand side variable	A -= B ⇒ A = A-B
*=	Multiply right-hand side value with left-hand side variable value and store the result into left-hand side variable	A *= B ⇒ A = A*B

/=	Divide left-hand side variable value with right-hand side variable value and store the result into the left-hand side variable	A /= B ⇒ A = A/B
%=	Divide left-hand side variable value with right-hand side variable value and store the remainder into the left-hand side variable	A %= B ⇒ A = A%B
&=	Bitwise ANDing left-hand side variable value with right hand side variable value & store the result into the left-hand side variable	A &= B ⇒A=A&B
^=	Bitwise XORing left-hand side variable value with right hand side variable value & store the result into the left-hand side variable	A ^= B ⇒A=A^B
|=	Bitwise ORing left-hand side variable value with right hand side variable value & store the result into the left-hand side variable	A |= B ⇒A=A|B
<<=	Bitwise Left shifting the left side variable value with indicated number of bits in the right-hand side & store the result into the left-hand side variable.	A <<= 2 ⇒A = A << 2
>>=	Bitwise Right shifting the left side variable value with indicated number of bits in the right-hand side & store the result into the left-hand side variable.	A >>= 2 ⇒A = A >> 2

2.2.2 Special Operators:

The following are the special operators in c programming language.

Operator	Description	Example
sizeof()	returns the **size(length in bytes) of entity,** for eg. a variable or an array, etc.	sizeof(x) will return size of the variable x
&	returns the memory address of the variable	&x will return address of the variable x
*	represents pointer to an object. The * operator returns the value stored at a memory address.	m = &x (memory address of variable x) * m will return the value stored at memory address m
. (dot) operator	used to access individual elements of a **C structure or C union.**	If **emp** is a structure with an element int age in it, then **emp.age** will return the value of age
-> (arrow) operator	used to access structure or union elements using a pointer to structure or union.	If **p** is a pointer to the emp structure, then we can access age element using **p->age**
[] operator	used to access array elements using indexing	if arr is an array, then we can access its values using arr[index], where index represents the array index starting from zero

2.2.3 Operator Precedence and Associativity in C :

- Order of evaluation of operations can alter the final result. For example, consider evaluating following expression without any mathematics rule.

```
5 + 3 * 4 / 2
```

- If you do not follow any specific rule to evaluate such expressions, then you probably lead to ambiguous result. Such as one might think it will evaluate either 32 or 11 or 18.5 etc.

- To overcome such ambiguity, in mathematics we follow BODMAS rule. Similarly, in computer programming we follow operator precedence and associativity rule.

Precedence of operators

- *Precedence* of an operator specifies its priority compared to another operator. If an expression contains different types of operators, then precedence of operators specifies the order of evaluation each operator.

- For example, consider the given expression **int**

```
x = 5 + 4 * 2;
```

The result of above expression is 13 instead of 18. Since * operator has higher precedence than + operator. Hence, + operator is evaluated after * operator.

- Now consider the above expression again with slight change, int

```
x = (5 + 4) * 2;
```

This time result of above expression is 18. Because precedence

of ( ) operator is more than * operator. Thus ( ) operator evaluates first to 5 + 4 = 9. Then * operator evaluates to 9 * 2 = 18.

- Precedence of an operator affects result of an expression. But, what if two operators have same precedence? This ambiguity (confusion) is resolved by associativity of operators.

Associativity of operators

- If an expression contains more than one operator with same precedence. Then operator precedence along with its associativity defines the order of evaluation of expression.

- Operator associativity can either be left-to-right or right-to-left. Means if an expression contains two or more operators of same precedence. Then they are evaluated in either left to right or right to left order.

- Consider the below expression

```
int x = 5 * 4 / 4 % 3;
```

In the above expression precedence of all three operators are same. Hence, the result of expression is ambiguous (if we follow only precedence). The result can either be 2 or 20 etc. However, order of evaluation ambiguity is resolved if we follow precedence along with associativity rule. All the three operators are left to right associated, hence the expression is guaranteed to evaluate 2.

- Consider another operator that associates from right-to-left.

```
x = y = z;
```

In the above expression the right-most operator is evaluated first, proceeding towards left. First the value z is assigned to y then the value of y is assigned to x.

Important note:

- Operators with same precedence have same associativity.
- In C programming compilers follow operator precedence and associativity rule to resolve order of evaluation of operators.

Operator precedence and associativity table

- Here operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type) * & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	? :	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

2.3 *Input and Output Functions (I/O functions):*

- Reading data, processing it and writing the processed data or result of a program are the essential functions of a program. The C is a functional language. It provides a number of macros and functions to enable the programmer to effectively carry out these input and output operations.
- I/O functions are grouped into 2 categories :
  - **Unformatted I/O functions:** These I/O functions do not allow the programmer to specify the type of data & the way in which it should be read in or written out.
  - **Formatted I/O functions:** These I/O functions allow the programmer to specify the type of data & the way in which it should be read in or written out.
- The following is the list of formatted and unformatted I/O functions :

Function	Formatted	Unformatted
Input	scanf()	getchar(), gets()
Ouput	printf()	putchar(),puts()

2.3.1 *Formatted Input- The scanf() Function*

- C provides the scanf() function to read data values of different data types from standard input device (keyboard).
- The scanf() function is built-in function defined in a header file called "stdio.h". When we want to use scanf() function in our program, we need to include the respective header file (stdio.h) using #include statement.
- The scanf() function has the following syntax...

```
scanf("control string", address_list);
```

Function name — Function arguments

Where,

Control string → It is a sequence of one or more-character groups. Each character group is a combination of the % symbol and one of the conversion characters. Basically, the control string specifies type of values which are to be supplied to variables.

address_list → It specifies address of memory locations where the values of input variables to be stored.

- The rules that governs the syntax of scanf() function are given below,

- The control string must be enclosed with in a pair of double quotes.
- For every input variable there must be a character group. And each character group must begin with a % and followed by a conversion character (given in table below).
- Multiple character groups are allowed in the control string & in such case they may be contiguous or separated by commas or by blank spaces.
- The address_list contains addresses (memory locations) of input variables.
- All address_list variables are separated by commas. Do not use blank spaces.
- address_list must not be enclosed with in double quotes.
- The values in address_list variables should match in number(count), type and order of the variables.
- There must be a comma to separate address_list and control string.

Examples:

1. scanf("%d %d %d",&a,&b,&c);
2. scanf("%d %f %c",&p,&q,&r);
3. scanf("%s",name);

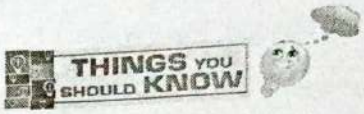
The different character groups and meaning associated with scanf() & printf() are listed as follows.

Character group	Meaning
%c	Read a single character
%d	Read a decimal integer
%f	Read a floating-point number
%s	Read a string
%e	Read a floating-point number (scientific notation)
%g	Read a floating-point number (scientific notation)
%h	Read short int number
%o	Read an octal number
%u	Read an Unsigned integer
%x	Read a hexadecimal number
%p	Read a pointer
%ld	Read long integers
%hd	Read short integers
%lf	Read long double floating point number

2.3.2 *Reading an Integer Input:*

- As shown in the table %d is the character group to input decimal integer data through the standard input device.
- For example: scanf("%d",&number);

  Let the input for the above statement be 50, then 50 is stored in the memory location represented by number.This is very simple input statement & doesn't specify how many digits are there in an input number.



Some variations in scanf() statement to read integer input:

- Specific way of accepting data through field specification.

  Field specification means the number of columns reserved (fields) for the number to be input.

  Syntax: scanf("%wd",&var_name);

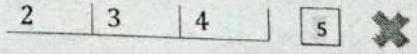
- Here the % sign denotes the conversion specification; w signifies the integer number that defines the field width of the number to be read. d defines the number to be read in integer format.
- Suppose you are reading a 3-digit number for a variable, say num. Let num=164,which is an integer then field width should be specified as 3 in the control string as shown

  scanf("%3d",&num);

Note:

- The number of characters in the data item should not exceed the field width specified. If the input number has more digits than the specified field width then all extra digits are not stored.

  Suppose for the same example above if the input data to be assigned is **2345**, then only **234** is stored in the memory location number and 5 is not stored.



* And if the input has less digits, then the specified field width then extra spaces (unused spaces) are filled with blank spaces.

Suppose if field width is 7 and input data to be assigned is **2345** then, the 3 extra spaces are filled with blank spaces(b).

b	b	b	2	3	4	5

➢ In some situation, it is required to skip the input number without assigning it to its respective variable. This is done by placing an asterisk symbol * between the % symbol and the field specifier.

For example: scanf("%d %*d %d",&a,&b,&c);

And assume following data is entered, 23 45 10

Then **23** is assigned to a, **45** is skipped due to * in second character group and **10** is assigned to variable c.

2.3.3 *Reading a floating-Point (real number) Input:*

➢ Floating point numbers can be input by %f specification. The field width is generally not specified for a real number, whether it is decimal or scientific notation. The following statement accepts real value for the variable z.

scanf("%f",&z);

suppose the number to be input for z is 50, then 50.0 is stored in the memory location designated by z.

➢ In case of inputting double type numbers, the character group should be %lf rather than simply **%f**. For example,

scanf("%lf",&z);

2.3.4 *Reading a Character Input:*

➢ Character constant can be input by %c specification and the %s character group is used for string constants. Consider the example,

scanf("%c",&symbol);

Here symbol is a character variable. If the character is A then the value 65 is stored in the memory. Because ASCII value of A=65.

➢ If the character input is a name, then sequence of characters that constitute name should be typed. In that case, how to read sequence of characters:

* **Method I:** The total number of characters present in the name must be specified in the control string. The scanf() function to read a name with %c character group is,

scanf("%10c",name);

If the input name is Ishwarkumar then only first 10 characters (Ishwarkuma)are stored in the memory & note that there is no ampersand (&) before variable name.

* **Method II:** Another simple way of accepting a sequence of characters is carried out by the %s character group. For example,

scanf("%s",name); Here also there is no ampersand **(&)** before variable name

Note: In the both methods the variable name(string) must be declared as one- dimensional array of character.

Example: char name[10];

- Any number of characters can be included in a string (not exceeding the maximum limit specified in the instruction).
- Always remember that each string ends with a null character (or a white space).
- However, accepting a string with blank spaces is possible. It is carried out by using the

symbol "^" **and new line** characters as in the following statement.

scanf("%[^\n]", message);

This means that go on typing any ASCII printable characters till the new line (\n) character is pressed. And if you input the message.

"I would like to learn electronics engg"

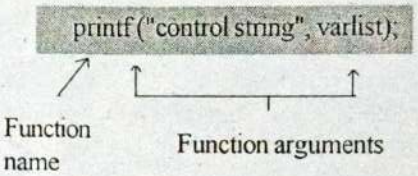
Then all characters (32 alphabets + 6 blanks) will be stored in the consecutive memory locations. And it should be noted that the string variable message should be declared as array of characters more or equal to number of characters in the message you type. Thus one can have the following declaration.

char message [40];

2.3.5 *Formatted Output- The printf() Function:*

- C provides the printf() function to display the data on the standard output device(monitor). The printf() function is also included in the header file **stdio.h.**

- The general form of printf() function is



- **Where,**

Control string → specifies the type & format of the values to be displayed

Varlist → is a list of variables to be displayed on the monitor

- Note: Here the control string may contain string or message we want to print on screen. Optionally it may contain format specifiers (%d,%f etc..) in case we need print variable value as well.
- printf () replaces all occurrence of format specifiers with the variable value. Optionally it typecasts a variable value to specifier type if needed.
- Examples on variation in the printf () function

1. **printf ("Welcome to GPT Mudhol");**

   This statement displays all the sequence of characters enclosed with in pair of double quotes on the screen

2. **printf ("%d", number);**

   This statement displays only the decimal integer value held by the variable number

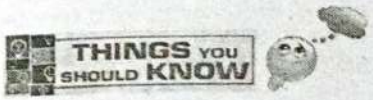
3. **printf ("The college code =%d", number);**

This statement prints the message and in front of the message it prints the content of variable number.

2.3.6 Integer Output:

➢ Integer number can be displayed on the monitor by using %d character group. Suppose the variable number is to be displayed. And also assume that its value is 164. Then the statement.

```
printf("%d",number); //displays 164 on the screen
```



➢ In case of integers the place holder (character group) can accept modifiers to specify the minimum field width and left justification (left most value to be stored in left most place). But remember by default all output is right justified. We can force the information to be left justified by putting a minus sign after the %. For example

Print statement	Output (assume x=1234)						
printf("%d",x);	1	2	3	4			
Printf("%6d",x);			1	2	3	4	Right justified
Printf("%-6d",x);	1	2	3	4			Left justified

➢ As shown in the above examples, if the number width in digits is less than the minimum field width the empty places are filled with blank spaces. However, if the number width is greater than the minimum field width, then the number is printed in full, overriding the minimum field width specification.

➢ It is also possible to pad (append) the empty places with zeros. If we want to pad with zeros, we have to place zero before minimum width specifier.

Print statement	Output (assume x=1234)						
printf("%3d",x);	1	2	3	4			Overriding minimum field width
Printf("%6d",x);	0	0	1	2	3	4	Padding with 0's

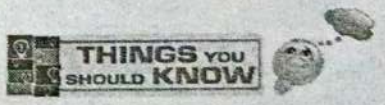
2.3.7 *Floating Point Output:*

➢ A floating-point number can be displayed either in a decimal form or scientific notation.

The %d and %e character groups are used respectively. For example

printf("%f",x);

If x contains the value 18, then it will appear on the screen as 24.000000



➢ In case of floating-point numbers placeholder (character group) can accept modifiers that specify minimum field width, precision and left justification. To add a modifier, we have to place a decimal point followed by the precision after the field width specifier.

➢ For e and f formats the precision modifier determines the number of decimal places to be displayed. For example

Print statement	Output (assume x=3456.12065)
printf("%f",x);	3 4 5 6 . 1 2 0 6 5 0 6 decimal places (Default precision is 6)
printf("%7.2f",x);	3 4 5 6 . 1 2 7 characters wide 2 decimal places
printf("%9.2f",x); (Right justified)	_ _ _ 3 4 5 6 . 1 2 9 characters wide 2 decimal places
printf("%-9.2f",x); (Left justified)	3 4 5 6 . 1 2 _ _ 9 characters wide 2 decimal places
printf("%9.3e",x);	3 . 4 5 6 e + 0 3 9 characters wide 3 decimal places
printf("%12.6e",x);	12 characters wide 3 . 4 5 6 1 2 0 e + 0 3 6 decimal places

2.3.8 *Character Output:*

- A single character or string (sequence of characters) can be displayed onto the standard output device by using printf() function. The %c and %s character groups are used for this purpose respectively.

- For example: **printf("%c",ch);**

  Where ch is a variable of type char. When this statement is executed, you will find on the screen A, if you have element A for ch.

- A string is displayed with following statement:

  printf("%s",town);

  Here the variable town is a string of characters. And before using the variable town, it must be declared as

  char town[15];

  If the string to be displayed is "Jamakhandi", then it would be pictorially shown as,

J	a	m	a	k	h	a	n	d	i				

2.3.9 *Unformatted Input Functions:*

These functions are primarily concerned with reading the character type data from the keyboard. The getchar() and gets() functions are used for this purpose.

2.3.10 *The getchar() Function:*

- This function reads a single character from the standard input device (keyboard). There is no parameter with in the parenthesis.

- **General Syntax:** ch_var=getchar();

  Where,

  ch_var? is a character type variable to which an accepted character is assigned.

2.3.11 *The gets() Function:*

- The gets () function reads in everything you enter from the keyboard until the **ENTER** key or **RETURN** is pressed. Here everything means a string which is a sequence of all printable ASCII characters.

- The **RETURN** key that you pressed will not be stored at the end of the string. It overcomes the limitation of scanf() statement with %s specification.

- **General syntax:** gets(string);

Where,

String→is a sequence of characters and it is of type char.

2.3.12 *Unformatted Output Functions:*

These functions are primarily concerned with displaying or printing the character type data on the monitor. The putchar() and puts() functions are used for this purpose.

2.3.13 *The Putchar() Function:*

- This function prints a single character on the screen. The character to be displayed is of type char.

- **General syntax:** putchar(ch_var);

Where,

ch_var→is a character variable which is enclosed within the paranthesis.

- **Example program**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    clrscr();
    char ch;
    printf("Enter any character\n");
    ch = getchar();
    putchar(ch);
getch();
}
```

Output:
Enter any character
A
A

2.3.14 *The Puts() Function:*

- This function prints a string of characters on the screen. The newline character that signals the end of the string will not be displayed.
- General syntax: puts(string);

  Where,

  String→is a sequence of characters
- **Example program:**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    clrscr();
    char name[30];
    printf("Enter your favourite color\n");
```

Output:
Enter your favourite color
Blue
Blue

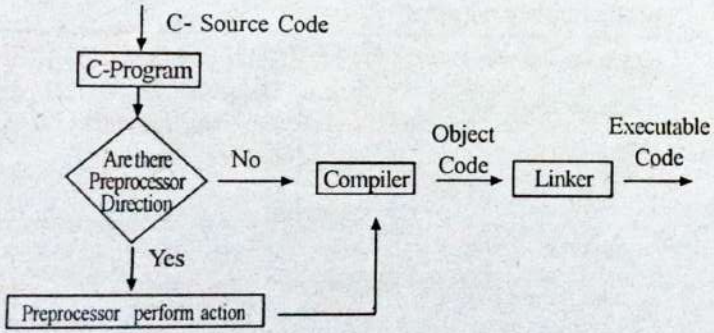
```
    gets(name);
    puts(name);
    getch();
}
```

2.3.15 *Preprocessor Directives:*

➢ **Preprocessor** in C is not a part of the compilation process but done just before compilation. Let's take an example. Suppose you want to go somewhere on your bike. So, going to a random place is part of your processing, whereas picking up the keys of the bike, turning on the ignition, and sitting on the bike is a part of **preprocessing**. So, this is just an example of preprocessing.

➢ **Definition:**

The term "preprocessor" is self-explanatory. The word **"pre"** means "before" and the word **"processor"** refers to "making something". Before the source code is compiled, it gets automatically processed due to the presence of preprocessor directives.



➢ The C preprocessor directives are useful in performing the following actions.

- Substitution of values
- Including files
- Conditional compilation

➢ The C preprocessor directives are usually written in the beginning of the program. However, its not mandatory, but just a good programming practice.

➢ The preprocessor directives begins with # symbol and followed by either the keyword.

include or define or some other keywords.

➢ Following table shows list of preprocessor directives and their usage.

Preprocessor directive	Use
#include	To include the contents of a file
#define	To create symbolic constants or macro
#if	To test whether compile time condition is true
#undef	To destroy a macro that was already created using #define
#else	To specify the alternative action if a test fails
#endif	To end the processor condition

2.3.16 *C Library Functions:*

➢ Library functions are built-in functions that are grouped together and placed in a common location called library.

- Each function here performs a specific operation. We can use this library functions to get the pre-defined output.
- All C standard library functions are declared by using many header files. These library functions are created at the time of designing the compilers. Normally the standard header files are having .h as extension which designates a header file.
- We include the header files in our C program by using **#include<filename.h>**. Whenever the program is run and executed, the related files are included in the C program.
- Some of the important & commonly used header files are given in the table below :

Header file	Used for
stdio.h	Input/output functions
conio.h	Console input/output functions
math.h	Mathematical functions
string.h	String manipulation functions
malloc.h	Memory allocation/deallocation functions
stdlib.h	Some standard library functions
ctype.h	Character manipulation functions
time.h	Time computing functions
graphics.h	Graphical functions
dos.h	Function linking to DOS routines

- **Advantages of C Standard Library Functions**

- It reduces application development time as you don't have to write code for functionalities provided by standard library functions.
- Reliable and time-tested code. Standard library functions are used everywhere from long time.

- Standard library functions are optimized for faster execution and for using *less memory space.*
- Some of the most commonly used functions in the C program in the header file definitions are listed as,

stdio.h library functions	
printf()	shows the output in the user's format.
scanf()	used to take the user's input which can be a character, numeric value, string,etc.
getchar()	takes character input from the user.
gets()	reads a line.
putchar()	Prints character on the output screen
puts()	Prints a line

math.h library functions	
pow()	finds the power of the given number.
sin()	finds the sine of the given number.
exp()	finds the exponent of the given number.
cos()	finds the cosine of the given number.
sqrt()	finds the square root of the number.
log()	finds the logarithmic value of the number.

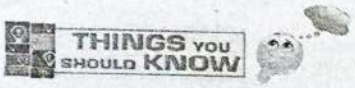
conio.h library functions	
clrscr()	Clears previous output history from command window
getch()	Holds output on screen until a key pressed from keyboard

Note: The sample library functions (stdio.h,math.h,conio.h) include these common functions (there are many other functions too).

Lab Programs

Experiment 2.a

Compute simple interest given the principal, interest rate and duration.



Simple interest formula is given by.

$$SI = \frac{P \times T \times R}{100}$$

Where,

P is the principle amount

T is the time(duration) and R is the rate of interest

C program:

```
#include <stdio.h>
#include <conio.h>
void main()
{
  clrscr();//clears previous output history from console
  float principle, time, rate, SI;

  //Input principle, rate and time
  printf("Enter principle (amount): ");
  scanf("%f", &principle);
```



```
  printf("Enter time(duration): ");
  scanf("%f", &time);

  printf("Enter rate of interest: ");
  scanf("%f", &rate);

  // Calculate simple interest
  SI = (principle * time * rate) / 100;

  // Print the resultant value of SI
  printf("Simple Interest = %f", SI);

  getch();// holds output on screen until a character press from
  keyboard
}
```

Output:

Enter principle (amount): 1200

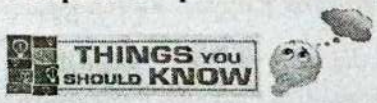
Enter time(duration): 2

Enter rate of interest: 5.4

Simple Interest = 129.600006

Experiment 2.b

Compute compound interest given P,t,r,n.



Compound interest formula is given by.

$$CI = P\left(1 + \frac{R}{100}\right)^{T}$$

Where,

P is principle amount

R is the rate of interest and T is the time span

C program:

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
void main()
{
 clrscr();
 float principle, rate, time, CI;

 // Input principle, time and rate
 printf("Enter principle (amount):\n ");
 scanf("%f", &principle);

 printf("Enter time:\n ");
 scanf("%f", &time);

 printf("Enter rate of interest:\n ");
 scanf("%f", &rate);

 //Calculate compound interest
 CI = principle* (pow((1 + rate / 100), time));

 // Print the resultant CI
 printf("Compound Interest = %f", CI);

 getch();
}
```

Output:

Enter principle (amount): 1200

Enter time: 2

Enter rate of interest: 5.4

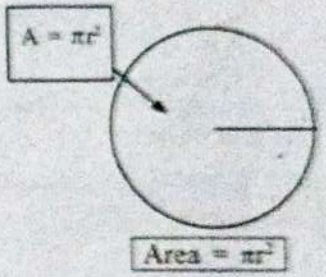
Compound Interest = 1333.099243

Experiment 2.c

- **Compute the area of a circle, square, rectangle and triangle.**

1. Area of circle is given by,

$A = \pi r^2$



Where,

Π=3.142 and

r= radius of the circle

C program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
  clrscr();
  float radius, area;
  //Input radius value of circle
  printf("Enter the radius of Circle in cm :\n ");
  scanf("%d", &radius);
  //calculate area of circle
  area = 3.14 * radius * radius;
  //print area of circle
  printf("Area of Circle : %f\n", area);
  getch();
}
```

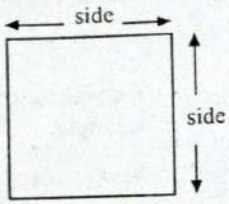
Output:

Enter the radius of Circle in cm : 2

Area of Circle : 12.560000

2. Area of square is given by,

A=side^2 or (side X side)



Area = side^2

C program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
  clrscr();
  int side, area;
  //read length of side
  printf("Enter the Length of Side :\n ");
  scanf("%d", &side);
  //compute area of square and print it
  area = side * side;
  printf("Area of Square : %d\n", area); getch();
}
```

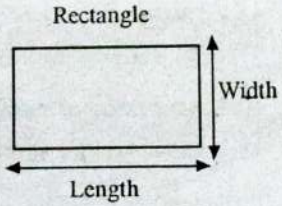
Output:

Enter the Length of Side : 5

Area of Square : 25

3. Area of Rectangle is given by the formula,

Area=length X width(breadth)



Area of rectangle = Length × Width

C program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
  clrscr();
  int length, width, area;
  //read length and width of rectangle
  printf("Enter the Length of Rectangle :\n");
  scanf("%d", &length);
  printf("Enter the Width of Rectangle :\n");
  scanf("%d", &width);
  //compute area and print it
  area = length * width;
  printf("Area of Rectangle : %d", area);
  getch();
}
```

Output:

Enter the Length of Rectangle : 5

Enter the width of Rectangle : 4

Area of Rectangle : 20

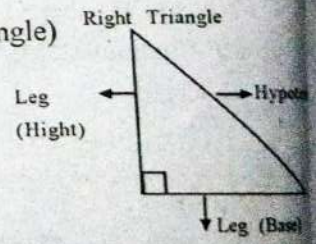
4. Area of triangle (right angle triangle) is given by,

Area =1/2× b × h

Where,

b =base of triangle

h=height of triangle



Area of Right Triangle = 1/2 × Base × Heigh

C program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
  clrscr();
  int base, height;
  float area;

  //Input the base and height of right-angle triangle
  printf("Enter the base of Right Angle Triangle :\n");
  scanf("%d", &base);

  printf("Enter the height of Right Angle Triangle :\n");
  scanf("%d", &height);

  //Compute the area and print
  area = 0.5 * base * height;
  printf("Area of Right Angle Triangle : %f", area);

  getch();
}
```

Output:

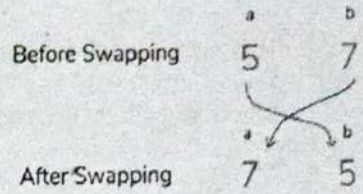
Enter the base of Right Angle Triangle : 4
Enter the height of Right Angle Triangle : 8
Area of Right Angle Triangle : 16.0

Experiment 2.d

Swap contents of two variables without using intermediate variables Swapping means?

Swapping two number in C programming language means **exchanging the values of two variables.**

Suppose you have two variable a & b. Value of a is 5 & value of b is 7. So, after swapping the value of a will become 7 & value of b will become 5.



C program:

```
#include <stdio.h>
#include <conio.h>
void main()
{
  clrscr();
  int a = 5, b = 7;
  printf("Before Swapping: a = %d, b = %d\n", a, b);

  // Code to swap 'a' and 'b'
  a = a + b; // a now becomes 12 b
  = a - b; // a becomes 5
  a = a - b; // a becomes 7

  a = a * b; // a now becomes 35
  a = a / b; // y becomes 5
  a = a / b; // x becomes 7

  printf("After Swapping: a = %d, b = %d\n", a, b);

  getch();
}
```

attemptive approach

Output:

Before Swapping: a = 5, b = 7
After Swapping: a = 7, b = 5

Review Questions :

1. Give a general form of conditional operator, Explain with an example.
2. List the formatted and unformatted I/O functions.
3. Describe scanf() statement with its general form. Explain it with an example.
4. With general syntax explain formatted input-output statements. Give example for each.
5. Define the following with an example:

   i. Increment Operator. ii. Decrement Operator
6. Explain different logical operators with examples.
7. List the different relational operators, give examples for each.
8. Write a C program to convert Celsius temperature into Fahrenheit temperature.Read the temperature value in Celsius from keyboard.(Hint: C = 5/9 x (F-32)).
9. What is the use of modulus operator, illustrate it with an example.
10. Define the terms

    i. precedence ii. Associativity
11. What are header files ? Why do you need them ?
12. What are pre-processor directives ? Write a note on them.

13. What is printed by the following program ?

```
void main()
{
int x=5,y=10,z=10;
x=y==z;
printf("%d",x);
getch();
}
```

14. What is the output of the following program?

```
void main()
{
int x=100, y=200;
printf("%d, (x>y)? x : y");
getch();
}
```

15. Write a C program to find perimeter of a circle? (Hint: Perimeter=2πr)

Week 3

Algorithms, Flowcharts and Control Structures

Algorithms, Flowcharts and Control Structures

3.1 Algorithm
- 3.1.1 Characteristics of Algorithm
- 3.1.2 Algorithm Notations
- 3.1.3 Examples of Algorithms
- 3.1.4 Flowchart
- 3.1.5 Examples of Flowcharts

3.2 Control Statements
- 3.2.1 Conditional Control Statements
- 3.2.2 if statement
- 3.2.3 if else statement

3.3 Nested if statement
- 3.3.1 Switch Statement

* Lab Programs
* Review Questions

Problem solving by a computer:

- Problem solving by a computer is creative and challenging process. It requires a careful study of the problem. In problem solving by computer, one should understand the given problem clearly in terms of the requirement such as input and output. Then decide which operations to be performed.
- Algorithm, flowchart and pseudocode are the problem-solving techniques.

3.1 *Algorithm :*

This is a problem-solving technique. An algorithm can be defined as "a step-by-step procedure to solve a particular problem".

Qualities of good algorithm:

- Input and output should be defined precisely.
- Each step-in algorithm should be clear and unambiguous.
- Algorithm should be most effective among many different ways to solve a problem.
- An algorithm shouldn't have computer code. Instead, the algorithm should be written in

  such a way that, it can be used in similar programming languages.

3.1.1 Characteristics of Algorithm:

Each and every algorithm is characterised by the following five important characteristics:

1. **Input:** it may accept a zero or more inputs
2. **Output:** It should produce at least one result
3. **Definiteness:** Each instruction must be clear, well defined and precise. There should not be any ambiguity
4. **Finiteness:** it should be a sequence of finite instructions. That is, it should end after a fixed time. It should not enter into infinite loop.
5. **Effectiveness:** This means operations must be simple and carried out in a finite time at one or more levels of complexity.

3.1.2 Algorithm Notations:

While writing algorithms the following notations are considered;

1. **Name of the algorithm:** It specifies the problem to be solved
2. **Step number:** Identification tag of an instruction and it is an unsigned positive number
3. **Explanatory comment:** It follows the step number and describes the operation. It should be written with in pair of square brackets
4. **Termination:** it specifies end of the algorithm. It is generally a STOP statement and last instruction of the algorithm.

3.1.3 Examples of Algorithms:

1. **Write an algorithm to add two numbers entered by user**

Step 1 Start

Step 2 Declare variables num1, num2 and sum

Step 3 Read values num1 and num2

Step 4 Add num1 and num2 and assign the result to sum. sum ← num1 + num2

Step 5 Print the value of variable sum

Step 6 Stop.

2. **Write an algorithm to compute the area of a circle**

Step 1 Start

Step 2 Declare the variables radius and area

Step 3 Read the value of radius

Step 4 Compute the area. **Area→ 3.142*radius*radius**

Step 5 Print the area

Step 6 Stop.

3. **Write an algorithm to compute average of 3 numbers**

Step 1 Start

Step 2 Declare the variables a,b,c and average

Step 3 Read the values of a,b and c

Step 4 Compute the average. **Average → (a+b+c)/3**

Step 5 Print the average

Step 6 Stop.

4. **Write an algorithm to find simple interest.(Given P,R,T)**

Step 1 Start

Step 2 Declare the variables P,R,T and SI

Step 3 Read the values of P,R and T

Step 4 Compute the SI. **SI→(P*T*R)/100**

Step 5 Print the SI

Step 6 Stop.

5. **Write an algorithm to find the largest among three different numbers entered by user.**

Step 1 Start

Step 2 Declare variables a,b and c

Step 3 Read variables a,b and c.

Step 4 If a>b and If a>c Display a is the largest number. Else Display c is the largest number. Else If b>c Display b is the largest number. Else Display c is the greatest number

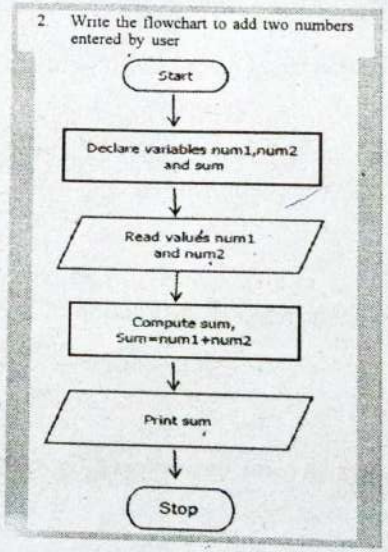
Step 5 Stop.

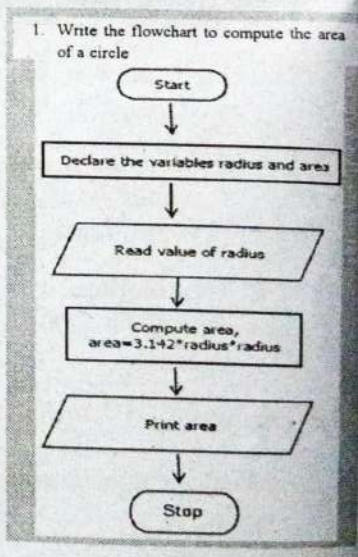
3.1.4 *Flowchart:*

- This is chart showing a flow of logic involved in solving a problem. This is defined for an algorithm.
- The flowchart can be defined as **"a diagrammatic representation of an algorithm."**. It is also referred as the blue print of an algorithm. It is also defined as a visual or graphical representation of an algorithm.
- The flowchart is an easy way to understand and analyse the problem. It is machine independent. It can be used for any type of problems.
- Flowcharts makes use of geometrical figures to specify a particular operation. Following table shows list of geometrical figures used in flowchart and their functions.

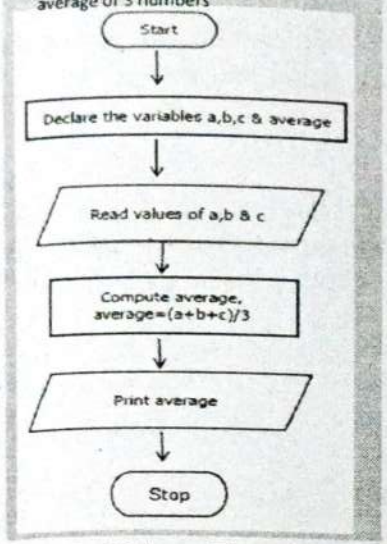
Geometrical Figure	Name	Function
	Oval	Start and Stop
	Parallelogram	Input and output
	Rectangle	Processing
	Diamond	Decision making
	Arrows	Connections
	Small circle	Continuation
	Hexagon	Repetition / looping

3.1.5 Examples of Flowcharts:

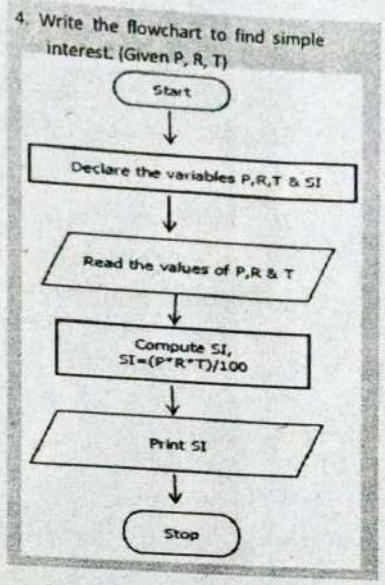




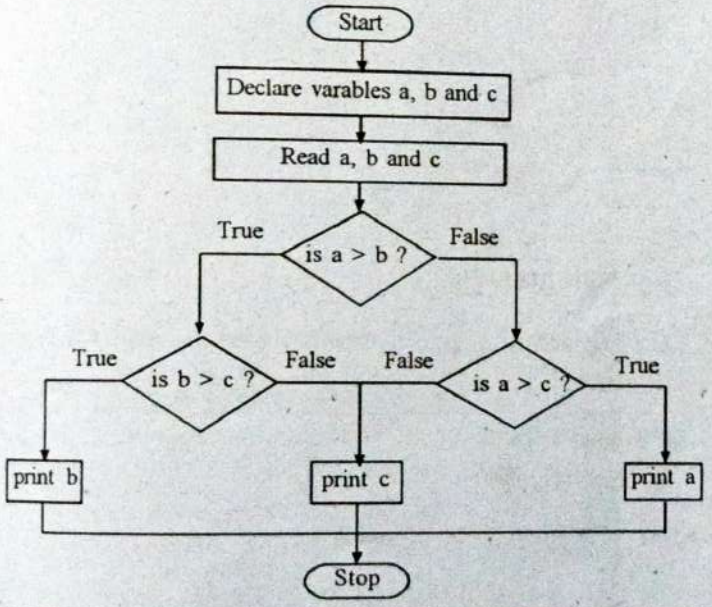
3. Write the flowchart to compute average of 3 numbers



4. Write the flowchart to find simple interest. (Given P, R, T)



5. Write an Flowchart to find the largest among three different numbers entered by user



3.2 *Control Statements:*

(Decision making and branching):

➢ Normally the statements in a program are executed in the order in which they appear in the program. This type of execution is called as a sequential execution.

➢ But in most of the programs, it is necessary to,

* Select a set of statements from several alternatives(options).
* Skip certain statements depending on some conditions in the program and continue the execution from some other part.
* Repeat a set of statements for a known number of times or until a specified condition is fulfilled.

Under such situations, we make use of control statements, they are generally called as control structures. They define the order of execution of statements in a program. Branching is so called because the program chooses to follow one branch or another.

C provides a wide range of control statements to control the flow of statements that are executed in a C program.

1. **Conditional control statements:** control statements that need any condition to control the program execution flow
2. **Unconditional control statements:** control statements that do not need any condition to control the program execution flow.

3.2.1 Conditional Control Statements:

➢ Here in these statements, it is necessary to check the condition

to make a decision. This involves performing a logical test. This test results in either true or false. Depending upon the truthness or falsity of the condition, the statements to be executed are determined.

- After that the control transfers to that statement in the program and starts executing the statements from that point. This is known as conditional execution. The condition execution involves both decision making and branching.

The various conditional control statements defined in C are

1. if statement
2. if-else statement
3. Nested if statement
4. Switch statement

3.2.2 *if Statement:*

- This is used to execute a statement or set of statements conditionally. It is also called as one way branching.
- Here logical condition is evaluated which results in either a **true** or **false** value. If the result of the logical test is **true** (non zero value) then the statement that immediately follows if statement is executed. If condition is **false** (zero value) then control transfers to the next executable statement, outside the body of **if**.

Syntax:

```
if (condition)
{
   Statement;
}
```

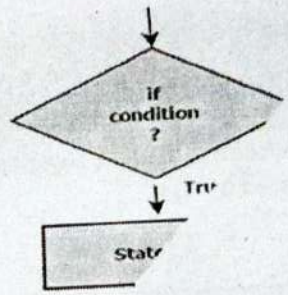

Where,

Condition → is a logical expression that results is true or false
Statement → a simple or a compound statement

Note:

A simple statement is a single statement. On the other hand, compound statement is a collection of two or more statements placed between a pair of braces. But, for a simple statement no braces are required.

Flowchart:



Example program:

C program to accept a number and print it if it is an even number

```
#include<stdio.h>
#include<conio.h>
void main()
{
```

```
clrscr();
int number;
printf("Enter a number:");
scanf("%d",&number);
if(number%2==0)
{
printf("%d is even number",number);
}
getch();
}
```

Output:
Enter a number: 4 4 is even number

3.2.3 *if else Statement:*

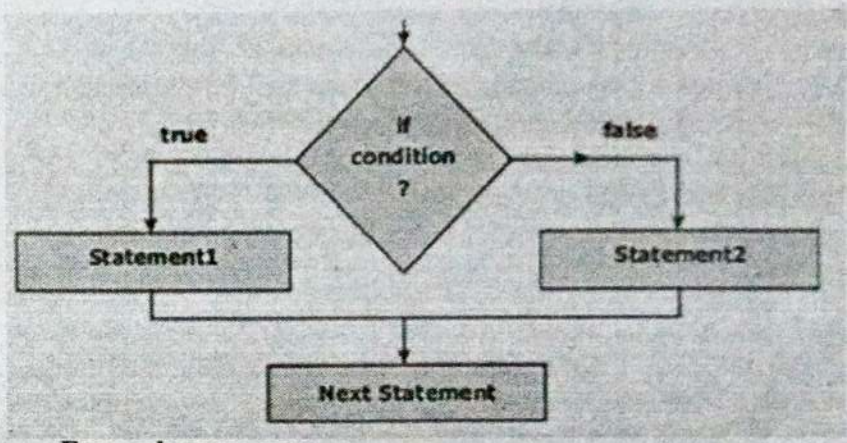
The if statement is used to execute only one action. If there are two statements to be executed alternatively, then if else statement is used. The if else statement is a two-way branching.

Syntax:

```
if (condition)
{
Statement1;
}
else
{
Statement2;
}
```

The condition is tested and if the result of logical test is true, then statement 1 is executed. Otherwise, statement 2 is executed. After executing one of these two statements, the control transfers to the statement immediately following if-else structure. Here statement1 and statement2 can be simple or compound statements.

Flowchart:



Example program:

C program to accept a number and print it whether it is an even or odd number

```
#include<stdio.h> #include<conio.h> void main()
{
clrscr();
int number;
printf("Enter a number:");
scanf("%d",&number);
if(number%2==0)
{
printf("%d is even number",number);
}
Else
{
printf("%d is odd number",number);
}
getch();
}
```

Output:
Run1
Enter a number: 4
4 is even number
Run2
Enter a number: 5
5 is odd number

3.3 *Nested if Statement:*

If there are more than two alternatives to select, then the nested-if statements are used. Enclosing an if statement within another if statement is called nested if statement.

Syntax:

```
if(condition1)
{
    if(condition2)
    Statement1;
    else
    Statement2;
}
    else
{
    Statement3;
}
```

The compiler first checks whether the **if** condition is *true* or not. If it is true the control flows to the next **if** inside the first **if** and checks whether the condition is true or not. In case it is true the if block of statements is executed otherwise the **else** block of statements are executed. Now if the first if condition turns out to be false, the **else** block of statement is executed.

Example Program:

/*Write a C program to accept three integers and print the largest amongst them. */


```
#include <stdio.h>
#include <conio.h>
void main()
{
  clrscr();
  int a,b,c;
  printf("Enter the values of a,b and c\n");
  scanf("%d %d %d",&a,&b,&c);
  if (a > b)
{
  if (a > c)
{
  printf("The greater number is : %d", a);
}
  else
{
  printf("The greater number is : %d", c);
}}
  else
{
  if (b > c)
  printf("The greater number is : %d", b);
  else
  printf("The greater number is : %d", c);
}
getch();
}
```

Output:

```
Enter the values of a,b and c
2 3 5
The greater number is :5
```

3.3.1 Switch Statement:

- Nested if allows to select one of many alternatives but its time consuming, since it tests all conditions and based on the result a particular branch is chosen for execution. To overcome this switch statement is used.
- The switch statement provides multiple way of branching. That is, it allows user to select any one of several alternatives, depending on the value of expression. Depending on the value of expression, the control transfers to a particular case label(branch) and all the statements followed by that case label are executed.

Syntax:

```
switch (variable or expression)
    {
    case label1:  Block1;
                  (or)
                  Statement-1;
                  break;

    case label2:  Block2;
                  (or)
                  Statement-2;
                  break;

      - - - - - - - -
      -     - - - - -

    case labeln:  Blockn;
                  (or)
                  Statement-n;
                  break;
    default       default – block;
    }
```

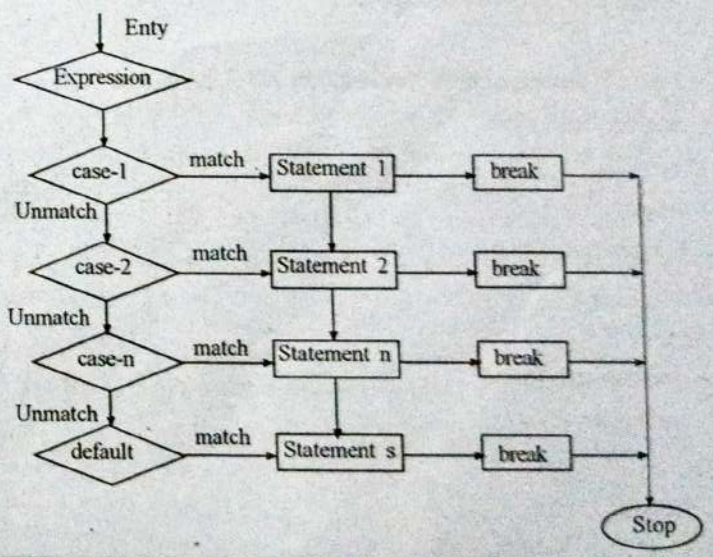

Here,

The expression or variable is of type of **int** or **char.** Depending on the value of an expression, the execution branches to a particular case label and then all the statements belonging to that case label are executed. The **break** statement indicates the end of a particular case label and there by statement is terminated. The case **default** is executed, when the value of an expression is not matched with any of the case labels.

Important Note:

1. There is no semicolon (;) placed at the end of **switch(expression).**
2. In switch each case block should end with **break** statement.
3. All case labels should be unique i.e., they can't be identical.
4. Case labels should end with colon (:)
5. The default label is optional, and there can be at most one default label.

Flowchart:



Example program:

/*C program to design calculator with basic operations using switch.*/

```
#include <stdio.h>
#include <conio.h>
void main()
{
 clrscr();
 int num1,num2;
 float result;
 char ch; //to store operator choice

 printf("Enter first number: ");
 scanf("%d",&num1);
 printf("Enter second number: ");
 scanf("%d",&num2);

 printf("Choose operation to perform (+,-,*,/): ");
 scanf(" %c",&ch);

 result=0;
 switch(ch)
{
 case '+':
 result=num1+num2;
 break;

 case '-':
```

Output:

```
Enter first number: 10
Enter second number: 20
Choose operation to perform (+,-,*,/,%): +
Result: 10 + 20 = 30.000000
```



```
 result=num1-num2;
 break;

 case '*':
 result=num1*num2;
 break;

 case '/': result=(float)num1/(float)num2;
 break;

 default:
 printf("Invalid operation.\n");
}
 printf("Result: %d %c %d = %f\n",num1,ch,num2,result);
 getch();
}
```

Lab Programs

Experiment 3 a

Compute the largest of three numbers using if-else and ternary operators.

Using if -else statement:

```
#include <stdio.h>
#include <conio.h>
int main()
{
  clrscr();
int a, b, c;

printf("Enter three numbers: ");
scanf("%d %d %d", &a, &b, &c);

if (a >= b && a >= c)
printf("%d is the largest number.", a);

else if (b >= a && b >= c)
printf("%d is the largest number.", b);

else
printf("%d is the largest number.", c);

getch();
}
```

Output:
Enter three numbers:
5
6
7
7 is the largest number

Using ternary operators:

```
#include <stdio.h>
#include <conio.h>
void main()
{
  clrscr();
  int a, b, c, largest;

  printf("Enter three numbers: ");
  scanf("%d %d %d", &a, &b, &c);

  largest = a > b ? (a > c ? a : c) : (b > c ? b : c);
  printf("%d is the largest number.", largest); getch();
}
```

Output:
Enter three numbers:
5
6
7
7 is the largest number.

Experiment 3 b

Compute the result of a student using nested if.

Here let us assume that student is a candidate appeared for 6 subjects in SSLC exam, of maximum marks (600). The result of the student is calculated as follows

1. First student will input the marks obtained in all subjects one by one
2. Percentage will be calculated using formula

$$\text{percentage} = \frac{\text{Total marks obtained in all subjects}}{\text{Maximum marks (600)}} \times 100$$

3. Then the result will be computed as per below rules

Percentage	Result
>=85	Distinction
>=60 && <85	First class
>=50 && <60	Second class
>=35 && <50	Pass Class
* If in any subject <35 marks	Fail class

C program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
  clrscr();
  float s1,s2,s3,s4,s5,s6; float per,total;
  printf("Enter the marks of all subjects one by one ranging from
  (1 to 100)\n");
  scanf("%f%f%f%f%f%f",&s1,&s2,&s3,&s4,&s5,&s6);
  total=s1+s2+s3+s4+s5+s6;
  printf("Total marks=%.1f\n",total);
  per=(total/600)*100;
  printf("You have secured percentage=%.2f\n",per);
  if(s1<35 ||s2<35 ||s3<35 ||s4<35 ||s5<35 ||s6<35)
  printf("Result:Fail\n");
  else
```



```
{
  if(per>=85) printf("Result:Distinction");
  else if(per>=60)
  printf("Result:First class");
  else if(per>=50)
  printf("Result:Second classs");
  else
  printf("Result:Pass");
}
  getch();
}
```

Result:

Run1:

Enter the marks of all subjects one by one ranging from (1 to 100)

90 90 90 90 90 90

Total marks=540.0

You have secured percentage=90.00 Result:Distinction

Run2:

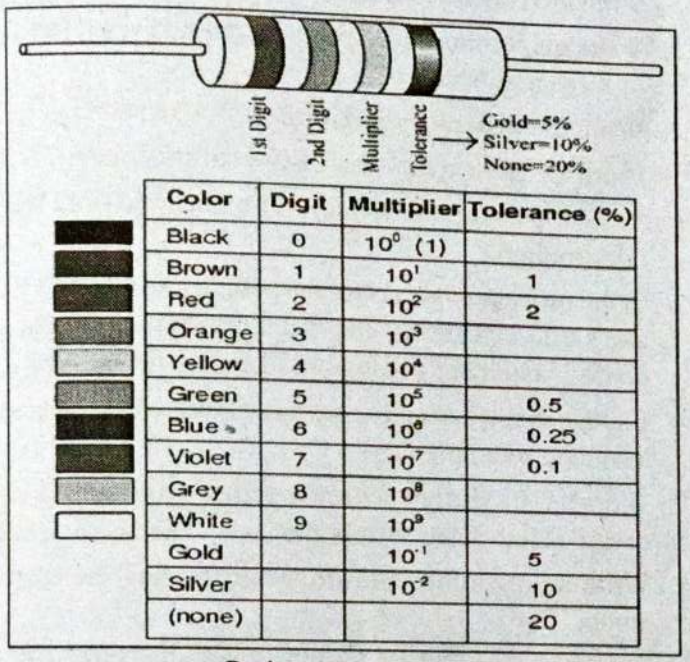
Enter the marks of all subjects one by one ranging from (1 to 100)
30 90 90 90 90 90

Total marks=480.0

You have secured percentage=80.00 Result:Fail

Experiment 3 b

Given the resistance and tolerance, generate the color bands of the resistor using a switch statement



Color	Digit	Multiplier	Tolerance (%)
Black	0	10^0 (1)	
Brown	1	10^1	1
Red	2	10^2	2
Orange	3	10^3	
Yellow	4	10^4	
Green	5	10^5	0.5
Blue	6	10^6	0.25
Violet	7	10^7	0.1
Grey	8	10^8	
White	9	10^9	
Gold		10^{-1}	5
Silver		10^{-2}	10
(none)			20

Resistor color code chart

C program:

```
#include<stdio.h>
#include<conio.h>
void main ()
{
   int digit[3],tol,i;
   clrscr();
   printf("Enter first digit within [0-9]:\n");
   scanf("%d",&digit[0]);
   printf("Enter second digit within [0-9]:\n");
```



```
   scanf("%d",&digit[1]);
   printf("Enter the multiplier within [0-9]:\n");
   scanf("%d",&digit[2]);
   printf("Enter the tolerance in percentage [5 or 10 or 20]\n");
   scanf("%d",&tol);
   printf("The resistance value entered is %d%dx10^%d
   ohm\n",digit[0],digit[1],digit[2]);
   printf("The color code for given resistance value is\n");
   for(i=0;i<3;i++)
   {
   switch (digit[i])
   {
      case 0:printf("black  ");
          break;
      case 1:printf("brown  ");
          break;
      case 2:printf("red  ");
          break;
      case 3:printf("orange  ");
          break;
      case 4:printf("yellow  ");
          break;
      case 5:printf("green  ");
          break;
      case 6:printf("blue  ");
          break;
      case 7:printf("violet  ");
          break;
      case 8:printf("grey  ");
          break;
      case 9:printf("white  ");
          break;
```

```
    default: printf("unknown value  ");
}       }

    switch (tol)
    {
                    /*colour band for tolerance*/
    case 5: printf("gold");
        break;
    case 10:printf("silver");
        break;
    case 20:printf("no colour");
        break;
    default:printf("unknown value");
        break;
    }
    getch();
}
```

Output:

Enter first digit within [0-9]:
1
Enter second digit within [0-9]:
2
Enter the multiplier within [0-9]:
3
Enter the tolerance in percentage [5 or 10 or 20]
5
The resistance value entered is 12x10^3 ohm
The color code for given resistance value is
brown red orange gold

Review Questions :

1. Define algorithm? Explain its characteristics.
2. Define flow chart? List the symbols used in flow-chart an write their significances.
3. Write a algorithm and C program to find the sum and averag of three numbers.
4. Write the general syntax of if & if-else statement.
5. Illustrate and explain nested if -else statement.
6. Write a C program to accept a number and print it if it is a odd number.
7. In the process of grouping the people into CHILD,YOUNG and OLD categories, depending on the age group, which control statement you would like choose and why ?
8. Explain the syntax and semantics of the switch statement wi a suitable example.
9. Write a C program to check whether given year is leap yea or not. (Hint: Leap year is the year which is divisible by 4
10. Write an algorithm and flowchart to find the largest of numbers.
11. Assuming that x=2, y=1 and z=0 initially, what will be the values after executing the following code segment switch(

```
{
case 2: x=1;
        y=x+1;
case 1: x=0;
        break;
default: x=1;
}
```

Week 4

Looping and Arrays

Looping And Arrays

4.1 Looping

4.1.1 while Statement

4.1.2 do while Statement

4.1.3 for Loop

4.2 Nested for Loops

4.2.1 Unconditional Control Statement

4.2.2 The "break" Statement

4.2.3 The "continue" Statement

4.2.4 The "goto" Statements

4.3 Arrays

4.3.1 Classification of an Arrays

4.3.2 One dimensional Arrays

4.3.3 Declaration of One-dimensional Array

4.3.4 Initialization of One-dimensional Array

* Lab Programs

* Review Questions

4.1 Looping

- Looping is a powerful programming technique through which a group of statements is executed repeatedly, until specified condition is satisfied. Looping is also called a repetitive or an iterative control mechanism.

- A loop is a program essentially contains two parts, one is called **body** of the loop and other is known as the control statements. The **control** statement performs a logical test where result is either **true** or **false.** If the result of this logical test is true, then the statements contained in the body of the loop are executed. Otherwise, the loop is terminated.

- The control statement can be placed either before or after the body of the loop. If the control statement is placed before the body of the loop, it is called **entry-controlled loop**. If the control statement is placed after the body of the loop, it is called as **exit controlled loop**. C supports both types of loop statements.

- There must be a proper logical test condition in the control statement, so that statements are executed repeatedly and loop is terminated gracefully. If the logical test condition is carelessly designed, then there may be possibility of formation of an infinite loop which keeps executing the statements over and over again. Therefore, for efficient looping mechanism, the following steps are included.

  - **Initialization**: To set initial value to a loop counter
  - **Decision** : An appropriate test condition to determine whether loop to be executed or not
  - **Updation** : Incrementing or decrementing loop counter value

Types of C Loops:

C language provides three looping statements...

1. while statement
2. do-while statement
3. for statement

4.1.1 *while Statement*

- The while is an entry-controlled loop statement. The test condition is evaluated and if the condition is true, then the body of the loop is executed.
- The execution process is repeated until the test condition becomes false and the control is transferred out of the loop.
- On exit, the program continues with the statement immediately after the body of the loop.
- The body of the loop may have one or more statements. The braces are needed only if the body contains two or more statements. It's a good practice to use braces even if the body has only one statement.
- **Syntax:**

```
while (Test Condition)
{
  Body of the loop
  Updation Expression;
}
```

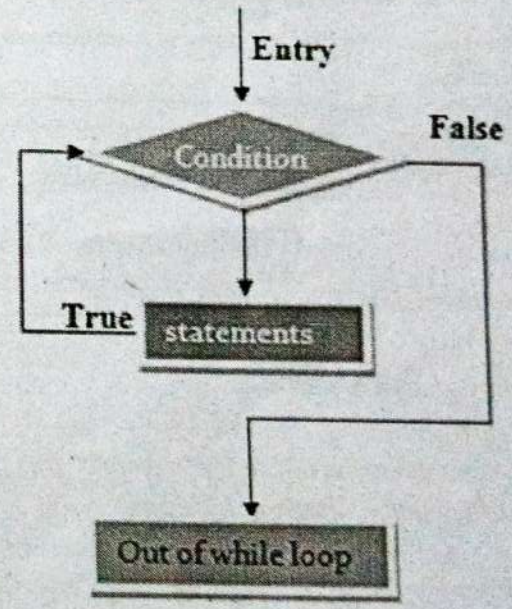

Where,

While → is a keyword

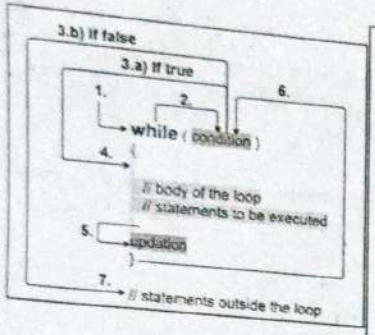
Test condition →is a logical expression that results in either true or false Body of loop? may be simple or a compound statement.

Updation expression→Increment or decrement expression of loop counter

- **Flow chart**



➢ ***Easy way to memorize:***



1. Control falls into the while loop.
2. The flow jumps to Condition
3. Condition is tested.
   a) If Condition yields true, the flow goes into the Body.
   b) If Condition yields false, the flow goes outside the loop
4. The statements inside the body of the loop get executed.
5. Updation takes place.
6. Control flows back to Step 2 & repeats everything again

Example Program:

/* C program to print "n" natural numbers using while loop statement */

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,i=1;//initialization
clrscr();
printf("Enter the value of n:\n");
scanf("%d",&n);
printf("The natural numbers from 1 to %d are\n",n);
while(i<=n) //condition checking (decision)
{
printf("%d\t",i);
i++; //updation
}
getch();
}
```

Output:

```
Enter the value of n:
5
The natural numbers from 1 to 5 are
1    2    3    4    5
```


4.1.2 do while Statement

➢ The do-while is an exit-controlled loop statement. Because in do-while, the condition is checked at the end of the loop. The code given between the do and while block will be executed until condition is true.

➢ The do-while loop will execute at least one time even if condition is false initially.

➢ On exit, the program continues with the statement immediately after the body of the loop.

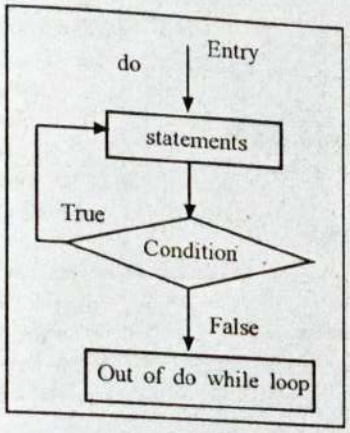
➢ The body of the loop may have one or more statements. The braces are needed only if the body contains two or more statements. It's a good practice to use braces even if the body has only one statement.

➢ **Syntax:**

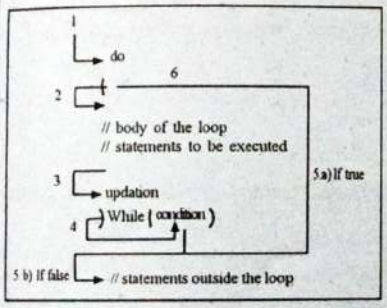
Where,

do and **while**	→	are keywords
Test condition	→	is a logical expression that results either true or false
Body of loop	→	may be simple or a compound statement
Updation expression	→	Increment or decrement expression of loop counter

➢ **Flowchart:**



➢ ***Easy way to memorize:***



1. Control falls into the do-while loop.
2. The statements inside the body of the loop get executed.
3. Updation takes place.
4. The flow jumps to Condition.
5. Condition is tested.
   a. If Condition yields true, goto Step 6.
   b. If Condition yields false, the flow goes outside the loop
6. Flow goes back to Step

Example Program:

/* C program to print "n" natural numbers using do while loop statement */

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,i=1;//initialization
clrscr();
printf("Enter the value of n:\n");
scanf("%d",&n);
printf("The natural numbers from 1 to %d are\n",n); do
{
printf("%d\t",i);
i++; //updation
}while(i<=n); //condition checking (decision)
getch();
}
```

Output:
Enter the value of n:
5
The natural numbers from 1 to 5 are
1 2 3 4 5

4.1.3 *for Statement*

➢ The for loop is used in the case where we need to execute some part of the code until the given condition is satisfied.

➢ The for loop is also called as a per-tested loop. It is better to use for loop if the number of iterations is known in advance.

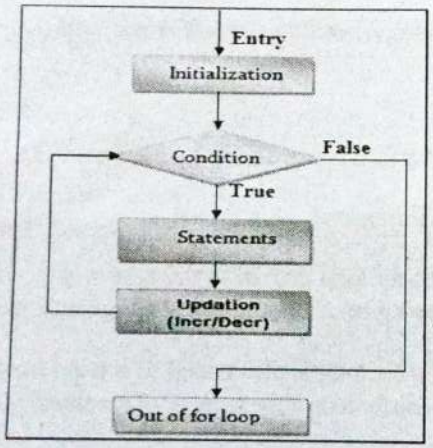
➢ **Syntax:**

```
for (initialize expression; test condition; updation)
{
    Statement-1;
    Statement-2;
}
```

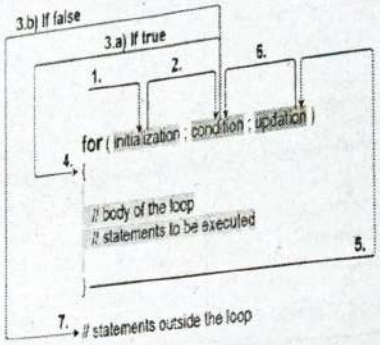
Where,

Initialize expression	→	Initialize loop index before the loop begins
Test condition	→	is conditional expression that tests whether loop index reached fixed value i.e., determines whether to continue or not
Updation	→	modifies the loop index(increment/ decrement) after every iteration

➢ **Flowchart:**



➢ ***Easy way to memorize:***



1. Control falls into the for loop. Initialization is done
2. The flow jumps to Condition
3. Condition is tested.
   a. If Condition yields true, the flow goes into the Body
   b. If Condition yields false, the flow goes outside the loop
4. The statements inside the body of the loop get executed.
5. The flow goes to the Updation
6. Updation takes place and the flow goes to Step 3 again
7. The for loop has ended and the flow has gone outside.

THINGS YOU SHOULD KNOW

➢ for is a keyword so it must be used only in lower case letters

➢ Every for statement must be provided with initialization, condition, and modification (They can be empty but must be separated with ";")

Ex: for (; ;) or for (; condition ; modification) or for (; condition ;)

Example Program:

/* C program to print "n" natural numbers using for loop statement */

```
#include<stdio.h>
#include<conio.h>
void main()
```

```
{
int n,i;
clrscr();
printf("Enter the value of n:\n");
scanf("%d",&n);
printf("The natural numbers from 1 to %d are\n",n);
for(i=1,i<=n;i++)
{
printf("%d\t",i);
}
getch();
}
```

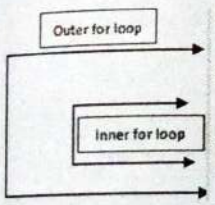
Output:

```
Enter the value of n:
5
The natural numbers from 1 to 5 are
1    2    3    4    5
```

4.2 Nested for Loops:

- If there are many data items to be processed against a set of elements repeatedly, there a single for statement is not enough. Then, we must use the nested for loop. If one for statement is completely placed within the other, it is known as the nested for loop statement.

- **Syntax:**




```
for(initialization; test_expression;updation) //outer loop
{
for(initialization; test_expression;updation) //inner loop
{
//statement(s) inside the body
} //End of inner for loop
} //End of outer for loop
```

- Here statement(s)-inside the body of inner for loop is executed for number of times specified by maximum limit of the second for loop, for each index value of the inner for loop.

Rules for nested for loop:

1. One loop must completely be placed inside the other i.e., no overlapping.
2. Each loop must have different index variables.

Example Program:

/* C program to tables up to given number n */

```
#include <stdio.h>
#include <conio.h>
void main()
{
    clrscr();
    int n;// variable declaration
    printf("Enter the value of n :");
    scanf("%d",&n);
    // Displaying the n tables.
    for(int i=1;i<=n;i++) // outer loop
    {
        for(int j=1;j<=10;j++) // inner loop
        {
            printf("%d\t",(i*j)); // printing the value.
        }
        printf("\n");
    }
    getch();
}
```

Result:

Enter the value of n : 3

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30

4.2.1 Unconditional Control Statements:

In c, there are control statements that do not need any condition to control the program execution flow. These control statements are called as unconditional control statements. C programming language provides the following unconditional control statements...

1. break
2. continue
3. goto

The above three statements do not need any condition to control the program execution flow.

4.2.2 The "break" Statement:

In C, the break statement is used to perform the following two things...

1. break statement is used to terminate the switch case statement
2. break statement is also used to terminate looping statements like while, do-while and for.
3. When break is used in nested loops then, only the innermost loop is terminated.

When a break statement is encountered inside the switch case statement, the execution control moves out of the switch statement directly.

Some working examples:

1.How break statement works inside a switch case.

```
switch(expression)
{
    case 1:
        // statements;
    case 2:
        // statements;
    case 3:
        // statements;
        break;
    case 4:
        // statements;
        break;
    default:
        // statements;
}
// statements below switch
```

2. How break statement works inside for loop

```
for(initialization; condition; update)
{
    // body of loop
    if(condition)
        break;
}
// statements below for loop
```

3. How break statement works insidea while loop.

```
while(condition)
{
    // body of while loop
    if(condition)
        break;
}
// statements below while loop
```

4. How break statement works insidea do...while loop

```
do
{
    // body of do while loop
    if(condition)
        break;
} while(condition);
// statements below do while loop
```

5. How break statement works within loop

```
for(initialization; condition; update)
{
    // body of for loop
    while(condition)
    {
        // body of inner while loop
        if(condition)
            break;
    }
    // statements below nested while loop
}
// statements below for loop
```

4.2.3 The "continue" Statement:

- The continue statement is used to move the program execution control to the beginning of the looping statement. When the continue statement is encountered in a looping statement, the execution control skips the rest of the statements in the looping block and directly jumps to the beginning of the loop.
- The continue statement can be used with looping statements like while, do-while and for.

Some working examples:

1. How continue statement works with for loop.

```
for(initialization; condition; update)
{
    // body of loop

    if(condition)
        continue;
}
// statements below for loop
```

2. How continue statement works with while loop.

```
while(condition)
{
    // body of while loop

    if(condition)
        continue;
}
// statements below while loop
```

3. How continue statement works with do...while loop.

```
do
{
    // body of do while loop

    if(condition)
        continue;
} while(condition);
// statements below do while loop
```


4.2.4 The "goto" Statement:

- C supports the "goto" statement to branch unconditionally from one point to another in the program.
- Although it may not be essential to use the "goto" statement in a highly structured language like "C", there may be occasions when the use of goto is necessary.
- The goto requires a label in order to identify the place where the branch is to be made.
- A label is any valid variable name and must be followed by a colon (:).
- The label is placed immediately before the statement where the control is to be transferred. The label can be anywhere in the program either before or after the goto label statement.

Syntax :

```
goto label;
.............
label:
statement;
```

In this syntax, label is an identifier. When, the control of program reaches to goto statement, the control of the program will jump to the label: and executes the code below it.

```
goto label;
.............
.............
.............
label:
statement;
```

Forward Jump

```
label:
Statement;
.............
.............
.............
goto label;
```

Backward Jump

- **"goto"** breaks the normal sequential execution of the program.

- If the "label:" is before the statement "goto label;" a loop will be formed and some statements will be executed repeatedly. Such a jump is known as a **"backward jump".**

- If the "label:" is placed after the "goto label;" some statements will be skipped and the jump is known as a **"forward jump".**

Example program to illustrate goto statement:

```
#include<stdio.h>
#include<conio.h>
void main( )
{
   clrscr();
   printf("Hello\t");
   goto label;
   printf("GPT");
   label:
   printf("Mudhol");
   getch();
}
```

Output:
Hello Mudhol

4.3 *Arrays:*

Why arrays?

- The fundamental data types, namely char int, float, and double are used to store only one value at any given time. Hence these fundamental data types can handle limited amounts of data.

- In some cases, we need to handle large volume of data in terms of reading, processing and printing. To process such large amounts of data, we need a powerful data type that would facilitate efficient storing, accessing and manipulation

of data items. For example: If the user wants to store marks of 50 students. This can be done by creating 50 variables individually but, this process is rather tedious and impracticable. This type of problem can be handled in C programming using arrays.

Definition:

"Array is a data structure that hold finite sequential collection of homogeneous data." To make it simple let's break the words.

- Array is a **collection** - Array is a container that can hold a collection of data.
- Array is **finite** - The collection of data in array is always finite, which is determined prior to its use.
- Array is **sequential** - Array stores collection of data sequentially in memory.
- Array contains **homogeneous data** - The collection of data in array must share a [illegible] ne data type.

4.3.1 Classification of an Arrays:

Arrays are classified into 2 types;

1. One dimensional array
2. Multidimensional arrays
   - Two-dimensional array
   - Three-dimensional array
   - four-dimensional array etc…

The dimensionality of an array is determined by the number of subscripts present in the given array. If there is only one

subscript, then it is called a one-dimensional array. If there are two subscripts, it is called a two-dimensional array and so on.

4.3.2 One dimensional Array:

This is a linear list of a fixed number of data items of the same type. All these items are accessed using the same name and a single subscript. It is similar to a row or a column matrix. It is also called a single dimensional array or a one subscripted variable.

4.3.3 Declaration of One-Dimensional Array:

Like any other variable, arrays must be declared before they are used. The syntax of declaring a one-dimensional array is as follows,

data_type arrayname[size];

where,

arrayname → is the name of an array

size → number of elements of type data_type. And the size must be an integer constant specified within a pair of square brackets

data_type → any basic data type or a user defined data type.

Examples:

int marks[5];

Here, we declared an array, mark, of integer type. And its size is 5. Meaning, it can hold 5 integer values.

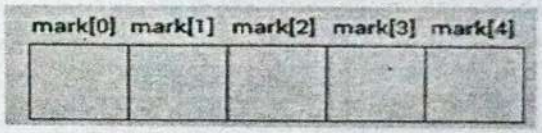
It's important to note that the size and type of an array cannot be changed once it is declared.

How to access Array Elements?

- The individual elements of an array are identified using the combination of 'arrayName' and 'indexValue'. We use the following general syntax to access individual elements of an array...

  arrayName [indexValue] ;

- Suppose you declared an array mark as above. The first element is mark[0], the second element is mark[1] and so on.



Few keynotes:

- Arrays have 0 as the first index, not 1. In this example, mark[0] is the first element.
- If the size of an array is n, to access the last element, the n-1 index is used. In this example, mark[4].

4.3.4 *Initialization of One-Dimensional Array:*

After an array is declared it must be initialized. Otherwise, it will contain garbage value(any random value). There are two ways to initialize an array.

1. **Static array initialization -** Initializes all elements of array during its declaration.
2. **Dynamic array initialization** - The declared array is initialized sometime later during execution of program.

1. Static initialization of array

We define value of all array elements within a pair of curly braces { and } during its declaration. Values are separated using comma, and must be of same type.

Example of static array initialization

int marks[5] = {90, 86, 89, 76, 91};

Note: Size of array is optional when declaring and initializing array at once. The C compiler automatically determines array size using number of array elements. Hence, you can write above array initialization as.

int marks[] = {90, 86, 89, 76, 91};

2. **Dynamic initialization of array**

You can assign values to an array element dynamically during execution of program. First declare array with a fixed size. Then use the following syntax to assign values to an element dynamically.

array_name[index] = some_value;

Example to initialize an array dynamically

```
marks[0] = 90; // Assigns 90 to first element of marks array
marks[1] = 86; // Assigns 86 to second element of marks array
```

* Instead of hard-coding marks values, you can ask user to input values to array using scanf() function.

```
scanf("%d", &marks[0]); // Input an integer from user and
assigns to first element of marks

scanf("%d", &marks[4]); // Input an integer from user and
assigns to fifth element of marks
```

* The array index is an integer value, so instead of hard-coding you can wrap array input code inside a loop.

```
int index;
// Run a loop from 0 to 4
for(index = 0; index < 5; index++)
{
```



```
// Replace the hard-coded index with 'index' variable
scanf("%d", &marks[index]);
}
```

The above code will run 5 times from 0 to 4. In each iteration it asks user to input an integer and stores it in successive elements of marks array.

Example program:

/* Program to take 5 values from the user and store them in an array Print the elements stored in the array */

```
#include<stdio.h>
#include<conio.h> void
main()
{
   clrscr();
   int values[5];
   printf("Enter 5 integers:\n ");
// taking input and storing it in an array
for(int i = 0; i < 5; ++i) {
scanf("%d", &values[i]);
}
printf("Displaying integers: \n");
// printing elements of an array
for(int i = 0; i < 5; ++i)
{
printf("%d\n", values[i]);
}
getch();
}
```

Output:

```
Enter 5 integers:
10
-30
34
0
20
Displaying
integers:
10
-30
34
0
20
```

Lab Programs

Experiment 4 a

Compute factorial of a single digit number.

What is factorial?

Factorial is the product of an integer with it's all below integer till 1. In mathematic representation factorial represents by ! sign.

For Example:

Factorial 5 is:

5! = 120 [That is equivalent to 5*4*3*2*1 =120]

C program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
clrscr(); int n,i;
long int fact=1;
printf("Please enter a number whose factorial is to be found\n");
scanf("%d",&n);
if(n<0)
printf("Error!Factorial of a negative number can't be found\n");
else
{
for(i=n;i>0;i--)
```



```
{
fact=fact*i;
}
printf("Factorial of %d= %ld",n,fact);
}
getch();
}
```

Output:

Please enter a number whose factorial is to be found 5
Factorial of 5= 120

Experiment 4 b

Compute the sum of digits of a given 3-digit number reducing it to a single digit.

For example:

If number is 999 the sum will be 9+9+9 = 27 = 2+7 = 9

```
#include<stdio.h>
#include<conio.h>
void main()
{
clrscr();
long int num;
int sum = 0, rem;
printf("Enter a number: \n");
scanf("%ld", &num);
while(num / 10 != 0)
```

Output:
Enter a number:
999
The sum is
9

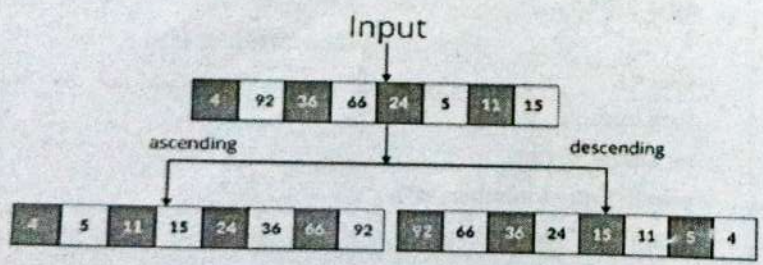
```
{
sum = 0;
while(num != 0)
{
rem = num % 10; // get the last digit of num
sum += rem;     // add rem to sum
num = num / 10; // remove the last digit from num
}
num = sum;
}
printf("The sum is %d\n", sum); getch();
}
```

Experiment 4 c

Sort an array of numbers in ascending order and descending

Sorting an array means to arrange the elements in the array in a certain order.

Following figure depicts the result after sorting an array in ascending and descending order.



C program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
  clrscr();
  int i,j,n,a[10],temp;
  printf("Please enter the size of array : \n") ;
                                    //Fetch size of array

  scanf("%d",&n) ;
  printf("Enter the %d elements one by one : \n",n) ;
                                    //Fetch elements of array

  for(i=0;i<n;i++)
  {
    scanf("%d",&a[i]) ;
  }
  for(i=0;i<n;i++) // loop for sorting array in ascending order
  {
    for(j=i+1;j<n;j++)
    {
    if(a[i]>a[j])
      {
          temp=a[i];
          a[i]=a[j];
          a[j]=temp;
```

```
        }
    }
}
printf("Ascending order of an array : \n");
                                    //print ascending order
for(i=0;i<n;i++)
{
    printf("%d\t",a[i]) ;
}
for(i=0;i<n;i++) // loop for sorting array in descending
order
    {
        for(j=i+1;j<n;j++)
        {
            if(a[i]<a[j])
            {
                temp=a[i] ;
                ;a[i]=a[j];
                a[j]=temp;
            }
        }
    }
printf("\nDescending order of an array : \n"); // print
descending order
for(i=0;i<n;i++)
{
    printf("%d\t",a[i]) ;
}
getch();
}
```


Result:

Please enter the size of array : 5

Enter the 5 elements one by one : 3

4 5 1 2

Ascending order of an array : 1

2 3 4 5

Descending order of an array: 5

4 3 2 1

Review Questions :

1. Give the general syntax and an example of do.....while loop statement. Why it is called as "exit-controlled" loop?
2. Differentiate between break and continue statements.
3. Explain for statement with an example.
4. Explain nested for loop with the general syntax.
5. Write a program to generate Fibonacci series up ton terms. (Hint:The series looks like 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...)
6. Distinguish between the while and do-while statements ?
7. What do you mean by forward and reverse jump? Give suitable examples.
8. What is an array? How do you classify arrays ?
9. How do you declare and initialization of arrays ?
10. Write a program to find the sum of n natural numbers.
11. How to access elements of an array? Illustrate it with an example
12. Write a program to count even and odd from n natural numbers.
13. Write a program to search an element in an array.

Week 5

Strings, Functions & Structures

Strings, Functions & Structures

5.1 Strings

5.1.1 Declaring and Initializing String Variables

5.1.2 Reading and Writing Strings

5.1.3 Two-diemensional Array

5.1.4 Syntax to Declare Two-Dimensional Array

5.1.5 Initialization of 2D Array

5.1.6 C String Functions

5.2 Functions in C

5.2.1 How to call C Functions in a Program

5.2.2 Recursion

5.3 Pointers in C

5.3.1 Accessing the Address of Variables

5.3.2 Declaring Pointers (Creating Pointers)

5.3.3 Assigning Address to Pointer

5.3.4 Accessing Variable Value using Pointer

5.3.5 Structure in C

5.3.6 How to Creae Structure ?

5.3.7 Creating and using Structure Variables

5.3.8 Memory Allocation of Structure

* Lab Programs

* Review Questions

5.1 Strings

- String in C is defined as an "array of characters that are terminated with a special character (Null character) '\0'."
- It can have alphabets in both cases, digits and special characters. It is like having a group of characters that forms words or sentences.
- This null character ('\0') indicates the end of the string.

For example, the string "COMPUTER" is stored as string of 8 characters & is stored in main memory as

C	O	M	P	U	T	E	R	\0
ch[0]	ch[1]	ch[2]	ch[3]	ch[4]	ch[5]	ch[6]	ch[7]	ch[8]

Where ch → is an array of characters

The very first character occupies 0th subscript place, the second character is stored at the 1st subscript place and so on, the last character occupies $(n-1)^{th}$ subscript in an n-character array. This rule is applicable to both numerical and character arrays. However, an n- character array contains (n+1) array elements. Because, each one-dimensional character array(string) ends with a null character (\0).

5.1.1 *Declaring and Initializing String Variables :*

Like numeric array, character array must be declared before they appear in a C program. There is no change in the syntax of declaring strings except that the change in the data type (char).

There are two ways to declare a string in c language.

1. By char array
2. By string literal

1) Let's see the example of declaring string by char array in C language.

char ch[10]={'e', 'l', 'e', 'c', 't', 'r', 'o', 'n', 'i', 'c', '\0'};

As we know, array index starts from 0, so it will be represented as in the figure given below.

e	l	e	c	t	r	o	n	i	c	\0
ch[0]	ch[1]	ch[2]	ch[3]	ch[4]	ch[5]	ch[6]	ch[7]	ch[8]	ch[9]	ch[10]

While declaring string, size is not mandatory. So we can write the above code as given below :

char ch[]={'e', 'l', 'e', 'c', 't', 'r', 'o', 'n', 'i', 'c', '\0'};

2) We can also define the string by the string literal in C language. For example :

char ch[]="electronic";

In such case, '\0' will be appended at the end of the string by the compiler.

5.1.2 Reading and Writing Strings :

A string can be read in and written out using input and output statements.

Reading Strings :

A string or a group of strings can be read from the input device (keyboard) using the following functions.

- scanf() with the %s format specification
- gets() function
- getchar() function

Writing Strings :

A string or a group of strings can be written to the monitor using the following output functions

- printf() with %s format specification
- puts() function
- puthchar() function

Sample Programs :

1. C program that illustrates string declaration and its initialization

```
#include<stdio.h>
#include<conio.h>
void main()
{
  // declare and initialize string
  char str[] = "Ishwar";

  // print string
  printf("%s",str);

  getch();
}
```

Output:
Ishwar

2. C program to read a string from the user and print the same on the console

```
#include<stdio.h>
#include<conio.h>
void main()
```

```
{
    clrscr();
    // declaring string
    char str[50];

    // reading string
    printf("Please enter your name\n");
    scanf("%s",str);

    // print string
    printf("Your name is %s",str);
    getch();
}
```

Output:
Please enter your name
Ishwar
Your name is Ishwar

3. C program to accept a string and count the number of characters in it and print the same

```
#include<stdio.h>
#include<conio.h>
void main()
{
    clrscr();
    char name[10];
    int count=0, i;
    printf("Enter your name\n");
    //get string input. Note that We are not using & name here.
    scanf("%s",name);
    //Logic to traverse through each element in the string
    for(i=0;name[i]!='\0';i++)
    count++;
    //print the name with character count
    printf("%s contains %d characters\n",name,count);
getch();
}
```

Output:
Enter your name
Ishwar
Ishwar contains 6 characters

5.1.3 *Two-Dimensional Array:*

Multi-dimensional array is an array of array or more precisely collection of arrays. Unlike one- dimensional array, multi-dimensional array stores collection of arrays.

Let us revise the concept of dimension.

- One-dimensional array : Collection of data/values.
- Two-dimensional array : Collection of one-dimensional array.
- Three-dimensional array : Collection of two-dimensional array.
- N-dimensional array : Collection of N-1 dimensional array.

We can declare array with any dimension. However, two-dimensional array is most popular and widely used to solve many mathematical problems.

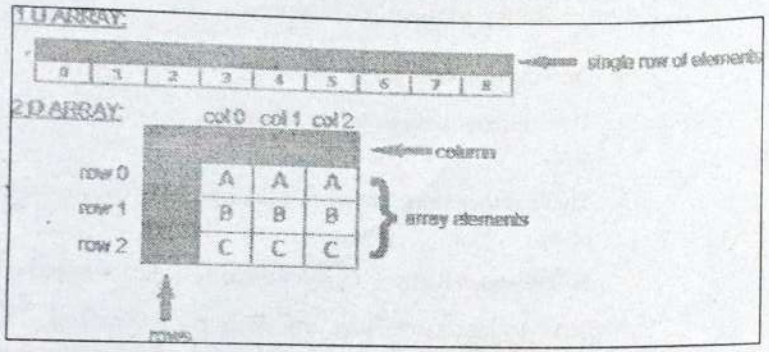
Note: Whether it is one, two or N-dimensional array. All array elements are stored sequentially in memory.

What is a two-dimensional array in C?

A 2D array is like a matrix and has a row and a column of elements (Although in memory these are stored in contiguous memory locations).

- A 1-D array, as we saw in the previous tutorial, is a linear list of data and needed only one index to access the element like a[2]. To access a two-dimensional array we need a pair of indices one for the row and another for the column like a[1][2].
- Similar to a one-dimensional array, we have the same name for all the elements present in the matrix.

- The difference that we have here is that a two-dimensional array is not linear in nature
- The following figure illustrates the difference between a one-dimensional array and a two-dimensional array:



- In the above figure, we can clearly see that the 2D array has two dimensions just like any two-dimensional figure such as a square or a rectangle.

5.1.4 *Syntax to Declare Two-Dimensional Array*

type array_name[row-size][col-size];

where,

type → is a valid C data type.

array_name→ is a valid C identifier that denotes name of the array.

row-size → is a constant that specifies matrix row size.

col-size → is also a constant that specifies column size.

Example:

int matrix[3][4];

The above statement declares a two-dimensional integer array of size 3x4 i.e. 3 rows and 4 columns (in terms of matrix).

5.1.5 *Initialization of 2D Array:*

2D array initialization can be done while declaring the array as well. In simple words, we can store certain elements in the array while writing the program.

You can initialize a two-dimensional array in any of the given form.

```
1) int matrix[4][3] = {
        {10, 20, 30},  // Initializes matrix[0]
        {40, 50, 60},  // Initializes matrix[1]
        {70, 80, 90},  // Initializes matrix[2]
        {100, 110, 120} // Initializes matrix[3]
    };
```

2) If you have mentioned row and column size specifically then curly braces for each row inside array initialization is optional. Hence, you can write the above initialization as.

```
int matrix[4][3] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110,
120};
```

3) Array column size is optional if specifying individual rows within pair of curly braces.

```
int matrix[4][] = {
        {10, 20, 30},  // Initializes matrix[0]
        {40, 50, 60},  // Initializes matrix[1]
        {70, 80, 90},  // Initializes matrix[2]
        {100, 110, 120} // Initializes matrix[3]
    };
```

There are several ways to initialize a two-dimensional array. It depends on programmers how they initialize. However, the first approach is considered as standard to initialize a two- dimensional array.

How to access two-dimensional array

Two-dimensional array uses two index values to access a particular element. Where first index specifies row and second specifies column to access.

Example to access two-dimensional array

matrix[0][0] = 10; // Assign 10 to first element of first row
matrix[0][1] = 20; // Assign 20 to second element of first row
matrix[1][2] = 60; // Assign 60 to third element of second row
matrix[3][0] = 100; // Assign 100 to first element of fourth row

Since array indexes are integer value, hence you can also wire a loop to access two-dimensional array. Two-dimensional array needs two index values to access any array element. Therefore, you need two loops. One outer loop to access for each row of the matrix, second loop for each column of the matrix.

```
// Runs 4 times iterating through each row
for(i=0; i<4; i++)
{
  // Runs 3 times for each row.
  for(j=0; j<3; j++)
  {
  scanf("%d", &matrix[i][j]);
  }
}
```


Sample program:

C program to read elements of a matrix and print the same in the matrix format:

```
#include<stdio.h>
#include<conio.h>
void main()
{
  clrscr();
  int i, j, m, n;
  int matrix[10][10];

  printf("Enter number of rows : ");
  scanf("%d",&m);
  printf("Enter number of columns : ");
  scanf("%d",&n);

  /* Input data in matrix */
  for(i=0;i<m;i++)
  {
    for (j=0;j<n;j++)
    {
      printf("Enter data in [%d][%d]: ", i, j);
      scanf("%d",&matrix[i][j]);
    }
  }
```

Output:

```
Enter number of rows : 2
Enter number of columns : 2
Enter data in [0][0]: 1
Enter data in [0][1]: 2
Enter data in [1][0]: 3
Enter data in [1][1]: 4
The matrix display is as follows
1     2
3     4
```

```
    /* Display the matrix */
    printf("The matrix display is as follows\n");
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            printf("%d\t",matrix[i][j]);
        }
        printf("\n");
    }
    getch();
}
```

5.1.6 C String Functions

There are many important string functions defined in "string.h" library.

No.	Function	Description
1)	strlen(string_name)	returns the length of string name.
2)	strcpy(destination, source)	copies the contents of source string to destination string.
3)	strcat(first_string, second_string)	concats or joins first string with second string. The result of the string is stored in first string.
4)	strcmp(first_string,second_string)	compares the first string with second string. If both strings are same, it returns 0.

5)	strrev(string)	returns reverse string.
6)	strlwr(string)	returns string characters lowercase.
7)	strupr(string)	returns string characters uppercase.

1. strlen() function:

The strlen() function returns the length of the given string. It doesn't count null character '\0'.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char ch[20]={'i','s','h','w','a','r','\0'};
printf("Length of string is: %d",strlen(ch));
getch();
}
```

Output:
Length of string is:6

2. strcpy() function:

The strcpy(destination, source) function copies the source string in destination.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
```

Output:
Value of second string is: ishwar

```
char ch[20]={'i','s','h','w','a','r','\0'}; char ch2[20];
strcpy(ch2,ch);
printf("Value of second string is: %s",ch2); getch();
}
```

3. strcat() function:

The strcat(first_string, second_string) function concatenates two strings and result is returned to first_string.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char ch[10]={'h', 'e', 'l', 'l', 'o', '\0'};
char ch2[10]={'c', '\0'};
strcat(ch,ch2);
printf("Value of first string is: %s",ch);
getch();
}
```

Output:
Value of first string is: helloc

4. strcmp() function:

The strcmp(first_string, second_string) function compares two string and returns 0 if both strings are equal.

Here, we are using gets() function which reads string from the console.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
```

Output:
Enter 1st string: gpt
Enter 2nd string: gpt
Strings are equal


```
{
char str1[20],str2[20];
printf("Enter 1st string: ");
gets(str1); //reads string from console
printf("Enter 2nd string: ");
gets(str2);
if(strcmp(str1,str2)==0)
printf("Strings are equal");
else
printf("Strings are not equal");
getch();
}
```

5. strrev() function:

The strrev(string) function returns reverse of the given string. Let's see a simple example of strrev() function.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char str[20];
printf("Enter string: ");
gets(str); //reads string from console
printf("String is: %s",str);
printf("\nReverse String is: %s",strrev(str));
getch();
}
```

Output:
Enter string:
Gpt
String is: Gpt
Reverse String is: tpG

6. strlwr() & strupr() function:

The strlwr(string) function returns string characters in lowercase & The strupr(string) function returns string characters in uppercase. Let's see a simple example of strupr() and strlwr() functions.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char str[20];
printf("Enter string: ");
gets(str); //reads string from console
printf("String is: %s",str);
printf("\nLower String is: %s",strlwr(str));
printf("\nUpper String is: %s",strupr(str));
getch();
}
```

Output:
Enter string: IshwARNirale
String is: IshwARNirale
Lower String is: ishwarnirale
Upper String is: ISHWARNIRALE

5.2 *Functions in C:*

- A function is a block of code that performs a particular task.
- There are many situations where we might need to write same line of code for more than once in a program. This may lead to unnecessary repetition of code, bugs and even becomes boring for the programmer. So, C language provides an approach in which you can declare and define a group of statements once in the form of a function and it can be called and used whenever required. These functions defined by the user are also known as User-defined Functions.

- C functions can be classified into two categories,
  1. Library functions/Built in functions
  2. User-defined functions
- **Library functions** are those functions which are already defined in C library, example printf(), scanf(), strcat() etc. You just need to include appropriate header files to use these functions. These are already declared and defined in C libraries.
- A User-defined functions on the other hand, are those functions which are defined by the user at the time of writing program. These functions are made for code reusability and for saving time and space

Advantages of Using Functions

1. It provides modularity to your program's structure.
2. It makes your code reusable. You just have to call the function by its name to use it, wherever required.
3. In case of large programs with thousands of code lines, debugging and editing becomes easier if you use functions.
4. It makes the program more readable and easier to understand.

* There are 3 aspects in each C function. They are,

✓ **Function declaration or prototype** - This informs compiler about the function name, function parameters and return value's data type.

✓ **Function call** - This calls the actual function

✓ **Function definition** - This contains all the statements to be executed.

1. *Function Declaration*

Like variable declarations, functions are also declared. Function declaration tells the compiler that there exists a function with some name that may be used later in the program. Function declaration is also known as function prototype or function signature.

Syntax of function declaration

return_type function_name(parameter_list);

➢ **return_type** - Return type defines the data type of value returned by the function.

A function does some calculation and may return a resultant value. So that the result of one function can be used by another function. For example - sqrt() function returns square root of given number, which is later used by calling function.

You must mention return type as **void** if your function does not return any value.

➢ **function name** - Function name is a valid C identifier that uniquely identifies the function. You must follow identifier naming rules while naming a function.

➢ **parameter list** - A function may accept input. Parameter list contains input type and variable name given to the function. Multiple inputs are separated using comma,

➢ A function declaration must be terminated using semicolon; You can declare a function anywhere in the program. However, it is best practice to declare functions below pre-processor directives.

Important note: Function declaration is optional. You can directly define and call a function skipping its declaration. However, you must define the function prior to its use.

2. Function definition

Function definition contains block of code to do task that the function is intended to do. Block of code must contain a return statement if return type of the function is not void.

Syntax of function definition

```
return_type function_name(parameter list)
{
   // Function body
}
```

Function declaration and definition syntax must be same. You are free to put function definition anywhere in your program, below its declaration. However, its good programming practice to put all declarations below main () function.

3. *Function calling*

The most important part of function is its calling. Calling of a function in general is execution of the function. It transfers program control from driver (current) function to called function. We can optionally pass input to the calling function.

Syntax of function call

function_name(parameter_list);

Function name - Name of the function to execute.

Parameter list - Comma separated input given to the function. Parameter list must match type and order mentioned in function declaration. Leave parameter list as blank if function does not accept any input.

Sample program:

C program to input two numbers from user. Find sum of the given two number using function.

C Program:

```
#include<stdio.h>
#include<conio.h>

// Addition function declaration
int add(int num1, int num2);
// Main function definition
void main()
{
    clrscr();
    // Variable declaration
    int n1, n2, sum;

    // Input two numbers from user
    printf("Enter two numbers:\n ");
    scanf("%d%d", &n1, &n2);
    /* Addition function call.n1 and n2 are
                        parameters passed to add function.
    Value returned by add() is stored in sum.*/
```

Output:
Enter two numbers:
2 3
Sum=5


```
    sum = add(n1, n2);

    // Print value of sum
    printf("Sum = %d", sum);
    getch();
}
    /* Addition function definition.
    * Return type of the function is int. num1 -
                First parameter of the function of int type
    * num2 - Second parameter of the function of int type*/
int add(int num1, int num2)
{
    int s = num1 + num2;
    // Return value of sum to the main function
    return s;
}
```

5.2.1 *How to call C Functions in a Program ?*

There are two ways that a C function can be called from a program. They are,

1. Call by value
2. Call by reference

1. Call by value:

- In call by value method, the value of the variable is passed to the function as parameter.
- The value of the actual parameter cannot be modified by formal parameter.
- Different Memory is allocated for both actual and formal

parameters. Because, value of actual parameter is copied to formal parameter.

Note:

Actual parameter - This is the argument which is used in function call. **Formal**

parameter - This is the argument which is used in function definition **C program:** (swapping)

```
#include<stdio.h>
#include<conio.h>
// function prototype/ function declaration
void swap(int a, int b);
void main()
{
clrscr();
int m=20,n=40;
// calling swap function by value
printf("values before swap m = %d and n = %d\n", m, n);
swap(m,n);
getch();
}
//function definition
void swap(int a, int b)
{
int temp;
temp=a;
a=b;
b=temp;
printf("values after swap m = %d and n = %d\n", a, b);
}
```

Output:
values before swap m = 20 and n = 40
values after swap m = 40 and n = 20

- In this program, the values of the variables "m" and "n" are passed to the function "swap".
- These values are copied to formal parameters "a" and "b" in swap function and used.

2. Call by reference:

- In call by reference method, the address of the variable is passed to the function as parameter.
- The value of the actual parameter can be modified by formal parameter.
- Same memory is used for both actual and formal parameters since only address is used by both parameters.

C program:

```
#include<stdio.h>
#include<conio.h>
// function prototype/function declaration
void swap(int *a, int *b);
void main()
{
clrscr();
int m = 20, n = 40;
// calling swap function by reference
printf("values before swap m = %d and n = %d\n",m,n);
swap(&m, &n);
getch();
}
//function definition
void swap(int *a, int *b)
{
```

Output:
values before swap m = 20 and n = 40
values after swap m = 40 and n = 20

```
int temp;
temp = *a;
*a = *b;
*b = temp;
printf("values after swap a = %d and b = %d", *a, *b);
}
```

- In this program, the address of the variables "m" and "n" are passed to the function "swap".
- These values are not copied to formal parameters "a" and "b" in swap function.
- Because, they are just holding the address of those variables.
- This address is used to access and change the values of the variables.

5.2.2 *Recursion:*

So far, we have seen the programs in which one function calls another function. But if you ask the question "Can a function, call itself?" then the answer is yes. This is possible because of the concept called the recursion.

Recursion is a technique that defines a function in terms of itself. That is a function which performs a particular task is repeatedly calling itself. There must be an exclusive stopping statement (terminating condition) in a recursive function. Otherwise, the function will not be terminated and enters into infinite loop.

* The general form of a recursive function definition is given below,

```
type_of_return_value rec_function_name (parameter definition)
{
declaration; terminating condition;
value=rec_function_name(expression); return(value);
}
```


Where,

rec_function_name → is the name of the recursive function

value → is the value to be returned to the calling function

terminating condition → is a stopping statement for the recursive function

Example program:

C program to accept a number from the keyboard and compute its factorial by recursive function.

```
#include<stdio.h>
#include<conio.h>
void main()
{
clrscr();
int num,fact;
int rec_func(int n);
printf("Enter a number\n");
scanf("%d",&num);
fact=rec_func(num);
printf("Factorial of %d =%d\n",num,fact);
getch();
}
//Recursive function for factorial
int rec_func(int n)
{
int fct;
if(n==0) //stopping condition
return(1); else
fct=n*rec_func(n-1); //recursive call
return(fct);
}
```

Output:
Enter a number
5
Factorial of 5 =120

5.3 Pointers in C:

- In the c programming language, we use normal variables to store user data values. When we declare a variable, the compiler allocates required memory with the specified name. In the c programming language, every variable has a name, datatype, value, storage class, and address. We use a special type of variable called a pointer to store the address of another variable with the same datatype. A pointer is defined as follows...

- "Pointer is a special type of variable used to store the memory location address of a variable."

- In the c programming language, we can create pointer variables of any datatype. Every pointer stores the address the variable with same datatype only. That means integer pointer is used store the address of integer variable only.

5.3.1 Accessing the Address of Variables

In c programming language, we use the reference operator "&" to access the address of variable. For example, to access the address of a variable "marks" we use "&marks". We use the following printf statement to display memory location address of variable "marks"...

Example Code

```
printf("Address : %u", &marks) ;
```

In the above example statement %u is used to display address of marks variable. Address of any memory location is unsigned integer value.

5.3.2 Declaring Pointers (Creating Pointers)

In c programming language, declaration of pointer variable is similar to the creation of normal variable but the name is prefixed with * symbol. We use the following syntax to declare a pointer variable...

```
datatype *pointerName ;
```

A variable declaration prefixed with * symbol becomes a pointer variable.

Example:

```
int *ptr ;
```

In the above example declaration, the variable "ptr" is a pointer variable that can be used to store any integer variable address.

5.3.3 Assigning Address to Pointer

To assign address to a pointer variable we use assignment operator with the following syntax...

```
pointerVariableName = & variableName ;
```

For example, consider the following variables declaration...

```
int a, *ptr ; ptr = &a ;
```

In the above declaration, variable "a" is a normal integer variable and variable "ptr" is an integer pointer variable. The address of variable "a" is assigned to pointer variable "prt". Here we say that pointer variable ptr is pointing to variable a.

5.3.4 Accessing Variable Value Using Pointer

Pointer variables are used to store the address of other variables. We can use this address to access the value of the variable through its pointer. We use the symbol "*" infront of pointer variable name to access the value of variable to which the pointer is pointing. We use the following general syntax...

*pointerVariableName

Example code:

```
#include<stdio.h>
#include<conio.h>
void main()
{
   clrscr();
   int a = 10, *ptr ; clrscr();
   ptr = &a ;
   printf("Address of variable a = %u\n", ptr) ;
   printf("Value of variable a = %d\n", *ptr) ;
   printf("Address of variable ptr = %u\n", &ptr) ;
   getch();
}
```

Output:
Address of variable a=65524
Value of variable a=10
Address of variable ptr=65522

5.3.5 *Structures in C*

- In C programming language, a structure is a collection of elements of the different data type. The structure is used to create user-defined data type in the C programming language. As the structure used to create a user-defined data type, the structure is also said to be "user-defined data type in C".
- In other words, a structure is a collection of non-homogeneous elements. Using structure, we can define new data types

called user-defined data types that holds multiple values of the different data type.

- The formal definition of structure is as follows...
- **"Structure is a collection of different type of elements under a single name that acts as user defined data type in C".**
- Generally, structures are used to define a record in the c programming language. Structures allow us to combine elements of a different data type into a group. The elements that are defined in a structure are called members of structure.

5.3.6 How to Create Structure?

To create structure in c, we use the keyword called "struct". We use the following syntax to create structures in c programming language.

```
struct <structure_name>
{
   data_type member1;
   data_type member2, member3;
   .
   .
   .
};
```

Example:

```
struct Student
{
   char stud_name[30];
   int roll_number;
   float percentage;
};
```

Note:

- Every structure must terminate with semicolon symbol (;).
- "struct" is a keyword, it must be used in lowercase letters only.

5.3.7 Creating and Using Structure Variables

- In a c programming language, there are two ways to create structure variables. We can create structure variable while defining the structure and we can also create after terminating structure using struct keyword.
- To access members of a structure using structure variable, we use dot (.) operator.

Example:

```
#include<stdio.h>
#include<conio.h>
struct Student          // structure declaration
{
 char stud_name[30];
 int roll_number;
 float percentage;
} stud_1 ;

void main()
{
 struct Student stud_2; // using struct keyword
 printf("Enter details of stud_1 : \n");
```



```
 printf("Name : ");
 scanf("%s", stud_1.stud_name);
 printf("Roll Number : ");
 scanf("%d", &stud_1.roll_number);
 printf("Percentage : ");
 scanf("%f", &stud_1.percentage);

 printf("***** Student 1 Details *****\n);
 printf("Name of the Student : %s\n", stud_1.stud_name);
 printf("Roll Number of the Student : %i\n", stud_1.roll_number);
 printf("Percentage of the Student : %f\n", stud_1.percentage);
 getch();
}
```

In the above example program, the structure variable **"stud_1"** is created while defining the structure and the variable **"stud_2"** is created using struct keyword. Whenever we access the members of a structure, we use the dot (.) operator.

5.3.8 Memory Allocation of Structure

When the structures are used in the c programming language, the memory does not allocate on defining a structure. The memory is allocated when we create the variable of a particular structure. As long as the variable of a structure is created no memory is allocated. The size of memory allocated is equal to the sum of memory required by individual members of that structure. In the above example program, the variables stud_1 and stud_2 is allocated with 36 bytes of memory each.

```
struct Student{
   char stud_name[30];———— 30 bytes
   int roll_number;———————— 02 bytes
   float percentage;————— 04 bytes
};
                          sum = 36 bytes
```

Here the variable of Student structure is allocated with 36 bytes of memory.

Important note:

- All the members of a structure can be used simultaneously.
- Until variable of a structure is created no memory is allocated.
- The memory required by a structure variable is sum of the memory required by individual members of that structure.

Lab Programs

Experiment 5 a

Compute the length of a string and reverse the string using string functions.

Note: All string library functions are defined in a library called string.h. To make use of those standard functions the library file has to be included in the program using the statement #include<string.h>.

strlen() function is used to find length of string and strrev() is the function used to reverse a string

C program:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
clrscr();
char name[20];
printf("Please enter a string\n");
gets(name);
printf("The length of the given string is=%d\n",strlen(name));
printf("The reverse of given string is %s\n",strrev(name));
getch();
}
```

Output:

```
Please enter a string
Ishwar
The length of the given string is=6
The reverse of given string is rawhsI
```

Experiment 5 b

Compute the sum of two matrices

Note : Addition of two matrices is possible only when both the matrices contain same number of rows and columns.

C Program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int row,col,mat1[10][10],mat2[10][10],sum[10][10],i,j;
printf("Enter the number of rows and columns :\n");
scanf("%d %d", &row, &col);
printf("Input Matrix 1 elements :\n");
for(i=0;i<row;i++)
for(j=0;j<col;j++)
{
scanf("%d",&mat1[i][j]);
}
printf("Matrix 1\n");
for(i=0;i<row;i++)
{
for(j=0;j<col;j++)
{
printf("%d\t",mat1[i][j]);
}
printf("\n");
}
```



```
printf("Input Matrix 2 elements :\n");
for(i=0;i<row;i++)
for(j=0;j<col;j++)
{
scanf("%d", &mat2[i][j]);
}
printf("Matrix 2\n");
for(i=0;i<row;i++)
{
for(j=0;j<col;j++)
{
printf("%d\t",mat1[i][j]);
}
printf("\n");
}
// Adding Two matrices
printf("Added Matrix\n");
for(i=0;i<row;i++)
for(j=0;j<col;j++)
{
sum[i][j]=mat1[i][j]+mat2[i][j];
}
//print the result
for (i = 0; i < row;  i++)
{
for (j = 0; j < col; j++)
{
```

```
printf("%d\t",sum[i][j]);
}
printf("\n");
}
getch( );
}
```

Output:

Enter the number of rows and columns :
2
2
Input Matrix 1 elements :
1 2 3 4
Matrix 1
1 2
3 4
Input Matrix 2 elements :
1 2 3 4
Matrix 2
1 2
3 4
Added Matrix
2 4
6 8

Experiment 5 c

Compute cube of a number using a function.

Suppose a user has entered 5. Cube of 5 is : 5 * 5 * 5 (125). Similarly, We can calculate cube of any number by multiplying the same number three times.

Here in the program, we have used the data type long int instead just int as to hold long range value of result.

C program:

```
#include<stdio.h>
#include<conio.h>
long int findcube(long int);//function prototype
void main()
{
clrscr();
long int number,cube;
printf("Please enter a number\n");
scanf("%ld",&number);
cube=findcube(number);//function call
printf("The cube of the number %ld is %ld",number,cube);
getch();
}
long int findcube(long int n)//function definition
{
return(n*n*n);
}
```

Ouptut:

Please enter a number 5

The cube of the number 5 is 125

Experiment 5 d

Store the details of an employee using a structure and print the details.

```
#include<stdio.h>
#include<conio.h>
void main()
{
clrscr();
/*structure declaration*/
struct employee{
   char name[30];
   int empId;
   float salary;
};
/*declare structure variable*/
struct employee emp;

/*read employee details*/
printf("\nEnter employee details :\n");
printf("Please enter name of the employee:\n");
gets(emp.name);
printf("Please enter the employee Id:\n");
scanf("%d",&emp.empId);
printf("Please enter the salary:\n");
scanf("%f",&emp.salary);

/*print employee details*/
printf("Entered employee details are:\n");
printf("-------\n");
printf("Name: %s\n",emp.name);
printf("Id: %d\n",emp.empId);
printf("Salary: %f\n",emp.salary);
getch();
}
```


Output:

```
Enter employee details :
Please enter name of the employee:
Ishwar Nirale
Please enter the employee Id:
145437
Please enter the salary:
25000
Entered employee details are:
-------
Name: Ishwar Nirale
Id:145437
Salary:25000
```

Review Questions :

1. With an example, explain declaration & initialization of two-dimensional array.
2. Define string. How to Declare & Initialize string variables with an example.
3. Explain how to read the string from the terminal and write a string to the terminal with an example.
4. Write a C program to check whether the given string is a palindrome ornot using string handling functions.
5. Explain any 5 string handling functions with an example.
6. Define a structure. Explain with an example, the general syntax of a structure.
7. How to call C Functions in a Program?
8. How does function definition differs from function declaration?
9. How do you declare a pointer variable? Explain with suitable examples
10. Write a program to concatenate two strings using in-built string function.
11. Write a C program for transpose of a matrix.

Week 6

8051 Architecture

8051 Architecture

6.1 Introduction to Concepts of Embedded Systems

6.1.1 Characteristics of an Embedded System

6.1.2 Applications

6.1.3 Microprocessor

6.1.4 Microcontroller

6.1.5 Comparison between Microprocessor and Microcontroller

6.2 Selection of Microcontroller

6.2.1 Features of 8051 Microcontroller

6.3 Architecture/Block Diagram Description

6.3.1 Program Status Word (PSW) Register (Flag Register)

6.3.2 Special Function Registers(SFRs)

* Lab Exercises

* Review Questions

6.1 Introduction to Concepts of Embedded Systems

➢ As its name suggests, Embedded means something that is attached to another thing. An embedded system can be thought of as a computer hardware system having software embedded in it. An embedded system can be an independent system or it can be a part of a large system. An embedded system is a microcontroller or microprocessor-based system which is designed to perform a specific task.

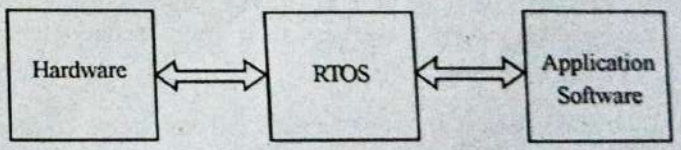
➢ **For example,** a fire alarm is an embedded system; it will sense only smoke.

An embedded system has three components ?

✓ It has hardware.

✓ It has application software.

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



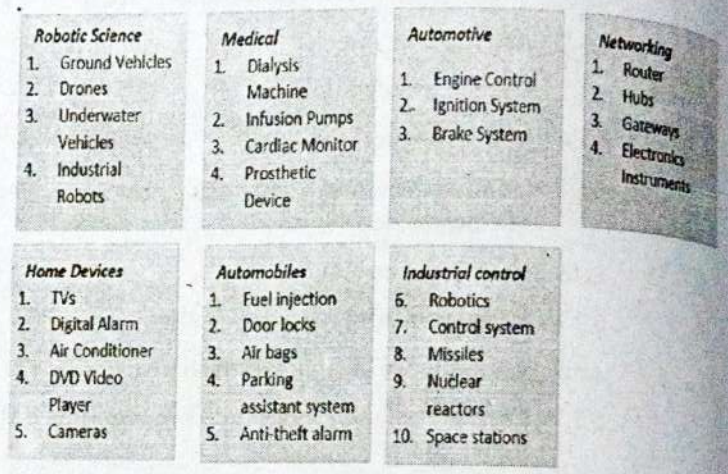
So, we can define an embedded system as a "Microcontroller based, software driven, reliable, real-time control system."

6.1.1 Characteristics of an Embedded System

- ✓ **Single-functioned** - An embedded system usually performs a specialized operation and does the same repeatedly. For example: A pager always functions as a pager.
- ✓ **Tightly constrained**- All computing systems have constraints on design metrics, but those on an embedded system can be especially tight. Design metrics is a measure of an implementation's features such as its cost, size, power, and performance. It must be of a size to fit on a single chip, must perform fast enough to process data in real time and consume minimum power to extend battery life.
- ✓ **Reactive and Real time** - Many embedded systems must continually react to changes in the system's environment and must compute certain results in real time without any delay. Consider an example of a car cruise controller; it continually monitors and reacts to speed and brake sensors. It must compute acceleration or de-accelerations repeatedly within a limited time; a delayed computation can result in failure to control of the car.
- ✓ **Microprocessors based** - It must be microprocessor or microcontroller based.
- ✓ **Memory** - It must have a memory, as its software usually embeds in ROM. It does not need any secondary memories in the computer.
- ✓ **Connected** - It must have connected peripherals to connect input and output devices.

- ✓ **HW-SW systems** - Software is used for more features and flexibility. Hardware is used for performance and security.

6.1.2 Applications:



Advantages

1. Easily Customizable
2. Low power consumption
3. Low cost
4. Enhanced performance

Disadvantages

1. High development effort
2. Larger time to market

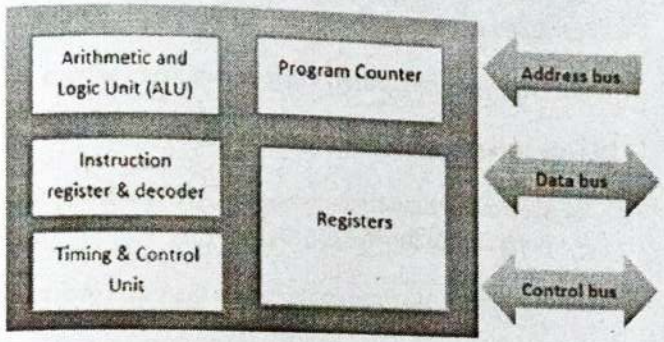
6.1.3 Microprocessor:

Microprocessor is an integrated chip (IC) which has only the CPU inside them. These microprocessors don't have RAM,

ROM & other peripherals inside the chip. The system designer (user) has to add them externally to make them functional.

A microprocessor is general purpose programming device whose tasks are unspecific (not specific). It provides flexibility to the programmer to execute multiple tasks at same time.

The basic block diagram of microprocessor is as shown below,



It mainly includes,

1. Arithmetic & Logic Unit
2. Registers
3. Timing & control unit
4. Instruction register & decoder
5. Program counter
6. System bus

Arithmetic & Logic Unit (ALU)

It is the computational (calculation) unit of microprocessor. It performs arithmetic & logical operations on various data. The operations are such as add, subtract, AND, OR, Ex- OR,

complement & so on. Whenever there is a need to perform an operation on a data then data is sent to ALU to perform necessary function.

Registers:

Registers are the small internal storage devices. Input, output data & various other binary data is stored in this unit for further processing.

Timing & Control Unit:

Control unit, as the name indicates it controls the flow of data & signals in the microprocessor. It generates the necessary control signals such as read, write, memory selection & I/O selection etc... for various data which are fed to the microprocessor. The timing unit sends the timing signals for the entire system.

Instruction register & decoder:

All instructions which are fetched from memory are loaded into instruction register. It is then sent to instruction decoder. The instruction decoder decodes the instruction & converts to the language (machine language) in which microprocessor can understand.

Program Counter (PC)

Program Counter stores the address of next instruction to be executed. It also keeps count on the number of instructions already executed. It is usually denoted by letter PC.

System bus:

A bus is the set of hardware lines or circuit lines which

help the communication between processor or other input/output units.

The system bus is basically classified into 3 groups:

a. Control bus

b. Data bus

c. Address bus

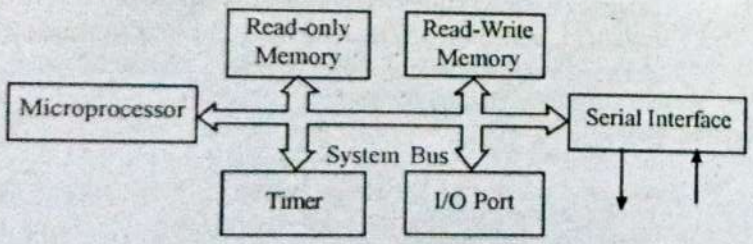
Control bus: It is a bidirectional (2 way) bus, which is used to transfer the control signals (read, write etc...) & timing signals between the microprocessor & other peripheral components.

Data bus: It is also a bidirectional bus, that is used to transfer data bidirectionally between microprocessor & other peripheral devices.

Address bus: It is a unidirectional (1 way) bus which is used to carry the address information between microprocessor & peripherals.

After connecting all external devices such as RAM, ROM, I/O devices etc. the microprocessor-based system looks like below.

Schematic Arrangement of a Microprocessor Based System



Advantages of a microprocessor:

- General purpose programming device
- Very flexible from design point of view
- High speed
- High accuracy & reliability
- Used to perform multitask operations
- Only require limited current and low voltages

Disadvantages of a microprocessor:

- Do not have internal memory (RAM & ROM), input/output ports inside the microprocessor
- No timers, no interrupts inside the microprocessor
- Make the system expensive
- Need proper interfacing components for functioning

Applications:

- Widely used in modern computer systems
- Used in robotics
- Used in space systems
- Used in telecommunication fields

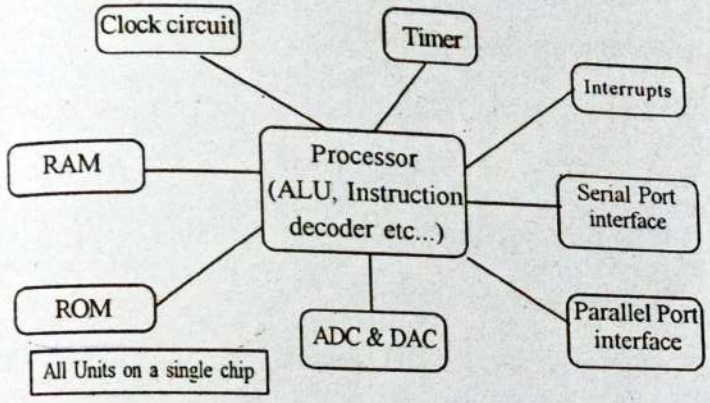
6.1.4 Microcontroller

Microcontroller has a CPU, in addition with a fixed amount

of RAM, ROM, Counters,clock circuits and other peripherals all embedded on a single chip. It is also referred as **"On chip peripheral programming device".**

Microcontrollers are designed to perform specific tasks. For example, washing machine, remote, mobiles, watches etc... These all need small amount of RAM, ROM, I/O ports etc... hence can be embedded on a single chip. This reduces size & cost of the system.

The basic block diagram of microcontroller is as shown in the figure.



It mainly includes,

a. Processor

b. Memory (RAM, ROM)

c. Parallel and serial I/O ports

d. ADC & DAC

e. Interrupt control etc...

Processor:

It is the brain of the system. It mainly consists of ALU, control unit (CU), instruction register & decoder. It controls the overall operation. CPU is responsible for mainly-

a. Fetching the instruction

b. Decoding the instruction

c. Executing the instruction

d. Memory/ I/O selection

Memory (RAM, ROM)

- The memory in a microcontroller is a storage unit, which stores data and program.
- Microcontroller usually has certain amount of RAM for storing the data & certain amount of ROM for storing the program.

Parallel & serial I/O ports

- Parallel and serial I/O ports are mainly used to drive/interface various devices, such as LCDs', LEDs', Printers, memories etc... to microcontroller.
- Parallel port is used to exchange more than one bit at a time with numerous (many) parallel wires. Each wire can carry one bit so the port can transmit many bits at the same time.
- Serial port is used to transmit the data one bit at a time, that is serially one after another.

Timers/Counters

- A microcontroller may have more than one timer/counter. The

timers & counters provide all timing & counting functions inside microcontroller.

- The major operations include clock functions, pulse generations, frequency measurements etc... This also can be used for counting external pulses.

ADC & DAC

- ADC converters are used for converting the analog signal to digital form. The digital output can be used for various digital applications. Example measurement devices.
- DAC perform reversal operation of ADC conversion. DAC converts the digital into analog format. It is usually used for controlling analog devices like DC motor etc...

Interrupt control

- The interrupt control is used for providing interrupt for a working program.
- The interrupt may be external (activated by using interrupt pin) or internal (by using interrupt instructions during programming).

Disadvantages of microcontrollers

1. Microcontrollers have got more complex architecture than that of microprocessor.
2. Can be used only for small specific task.
3. It is not possible to interface **"high powered devices"** directly with microcontrollers.
4. If any one unit inside the chip isn't working then entire microcontroller has to be replaced.

Advantages of microcontrollers

1. Low time required for performing operation.
2. It is easy to use, troubleshooting and system maintenance is straightforward.
3. Cost and size of the system is less.
4. Easy to interface additional RAM, ROM and I/O ports.
5. Microcontroller can act as a microcomputer without any digital parts.

Applications of microcontroller:

1. Used in robotics field.
2. Used in automobile applications.
3. Used in biomedical instruments.
4. Used in defence system.
5. Used in communication systems.
6. Used in light sensing & control devices.
7. Used in fire detection & safety devices.
8. Used in temperature sensing & control devices.
9. Used in various embedded devices.
10. Used in washing machine, microwave oven, refrigerator etc...

6.1.5 Comparison between Microprocessor and Microcontroller

Microprocessor	Microcontroller
Microprocessors are multitasking in nature, can perform multiple tasks at a time.	It is task oriented (task specific).
It is a general-purpose progr amming device.	It is specific purpose progr amming device.
RAM, ROM, I/O ports & timers can be added externally & can vary in numbers.	These are built in can't be added externally and are fixed in numbers.
It has one- or two-bit handling instructions.	It has many bit handling instructions.
Microprocessor based system requires more hardware & hence more space is needed.	Microcontroller based system requires less hardware & hence reduces the space needed.
Microprocessor based system is more flexible from design point of view.	Microcontroller based system is less flexible from design point of view.
Microprocessor based system is heavier and costlier.	It is light in weight and cheap.
As more hardware required, so need of more power for operation.	Requires less hardware, so power consumption is less.
Access time for memory or I/O devices is more.	Less access time needed, hence are fast in operation.
Less number of pins are multifunctional.	More number of pins are multifunctional.
If any unit isn't working then only that unit is replaceable.	If any unit isn't working then entire IC is to be replaced'.
Ex: Personal computer	Ex: Washing machine, remote controls etc..

6.2 Selection of Microcontroller:

A microcontroller is selected for a task by considering all following aspects related to it.

1. **Speed**

   The highest speed, the microcontroller can support as per application to be built.

2. **Packaging**

   Is it 40 pin DIP (Dual inline Package) or QFP (Quad Flat Package)? This is important in terms of space.

3. **Power consumption**

   This is an important criterion for battery powered products.

4. **Memory (RAM/ROM)**

   Amount of on chip RAM for storing data & on chip ROM for storing code should be sufficient.

5. **Cost per unit**

   This is important in terms of final cost of the products.

6. **Availability**

   Should be available in the needed quantities both now & in future in the market.

7. **Software environment**

   Availability of software development tools such as assembler, compiler & debugger etc...

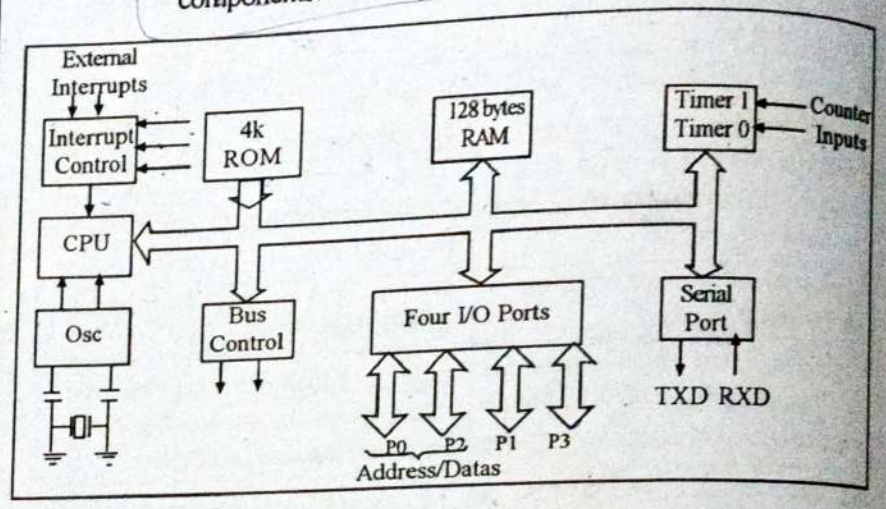
8. Finally, it should satisfy the computing needs of the task efficiently.

6.2.1 Features of 8051 Microcontroller

1. **4KB** bytes on-chip program memory (ROM)
2. **128** bytes on-chip data memory (RAM)
3. 8-bit bidirectional data bus
4. 16-bit unidirectional address bus
5. 32 general purpose registers each of 8-bit organized into 4 banks (Bank 0,1,2, Bank 3)
6. 32 bidirectional I/O pins organized into four 8-bit ports (P0, P1, P2 & P3)
7. 16-bit Timers (usually 2, but may have more or less)
8. 16-bit program counter and data pointer
9. Only one serial port for serial communication
10. Six interrupt sources
11. On chip oscillator, which should be activated by external clock
12. 2 level prioritized interrupts
13. Capable of addressing 64KB of external program memory & 64KB of data memory
14. Direct bit and byte addressability
15. Binary & decimal arithmetic

6.3 Architecture/Block Diagram Description:

The functional block diagram of the 8051 microcontroller is as shown in Figure. It mainly consists of the following components:



- On chip RAM 128 Bytes.
- On chip ROM 4K Bytes.
- Two Timers.
- Six Interrupts.
- One Serial Port.
- Four I/O Ports.

CPU:

It is Central processing unit. It is brain of the system. It has the following functional units.

1) ALU:

- It is arithmetic and logical unit.

- It works on 8 bits of data.
- It is used to perform the arithmetic operations like addition, subtraction, multiplication and division on bit of data.
- It is used to perform the logical operations like AND, OR XOR, NOT, ROTATE SWAP and SHIFT operations.

2) Working Registers

- The A and B registers are called as working registers.
- They are 8-bit registers.
- They are used to hold the data during data processing.

3) Program Status word (PSW) register

- It is an 8-bit register.
- It holds the status of arithmetic and logical operations.

4) Program Counter (PC)

- It is 16-bit register, it is used to store address of next instruction to be executed.
- PC gets automatically incremented as soon as any instruction is fetched.

5) DPTR (Data pointer)

- It is also 16-bit register.
- Used to store 16-bit data & also stores 16-bit address of external memory.

Memory:

The 8051 has the following memory units

1) RAM

- It is Random Access Memory. RAM is used to store data hence is also referred as data memory.
- 8051 has 128 bytes of RAM which is divided into:
  - a) 32 bytes of four register banks of 8 registers each.
  - b) 16 bytes of bit addressable location.
  - c) 80 bytes of general RAM memory.

2) ROM

- It is Read Only Memory. It is 4KB in size.
- It is used to store program codes and instructions.

3) Stack memory

- It is part of RAM memory & is used to hold the data.
- It is pointed by SP (stack pointer).

I/O ports

- These IO ports are used to communicate between 8051 MC and external world.
- The 8051 has four ports named as P0,P1,P2 and P3. Each port is having 8 pins. Totally 32 IO pins are available.

Timers/Counters:

- The 8051 has two 16 Bit-timers and counters -T0 and T1.
- They are used to generate the time delay and count an event.

Serial Port:

- The 8051 has only one serial port.
- It is used to transfer the data between two systems.

Interrupt Controller:

- It is used to control the interrupt occurrence to the controller.
- The 8051 has 2 external and 3 internal interrupts.

Oscillator circuit

- It is connected between XTAL1 and XTAL2 pins.
- It generates clock pulse for controller.

System bus:

A bus is the set of hardware lines or circuit lines which help the communication between processor or other input/output units.

The system bus is basically classified into 3 groups

d. Control bus

e. Data bus

f. Address bus

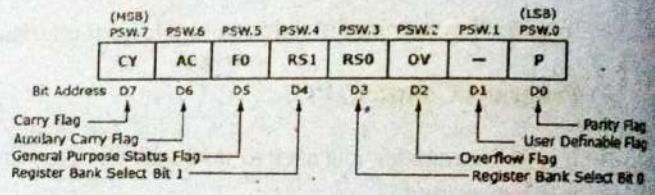
Control bus: It is a bidirectional (2 way) bus, which is used to transfer the control signals (read, write etc...) & Timing signals between the microprocessor & other peripheral components.

Data bus: It is also a bidirectional bus, that is used to transfer data bidirectionally between microprocessor & other peripheral devices.

Address bus: It is a unidirectional (1 way) bus which is used to carry the address information between microprocessor & peripherals.

6.3.1 Program Status Word (PSW) Register (Flag Register)

- The program status word (PSW) register is an 8-bit register that contain the flag(status) bits that reflect the current status of CPU. It is also known as flag register.
- They are generally changed by the ALU after every arithmetic or logic operation. However, flags can be also be changed by the programmer.
- PSW is a bit addressable register, that is each bit can be individually set or reset by the programmer.
- Out of 8 bits only 6 bits are used. The 2 unused bits are user defined flags.
- Following figure shows the program status word (PSW) register.



- **Four** of the flags are called **conditional flags**

a) PSW.7- CY(Carry)

b) PSW.6- AC (Auxiliary carry)

c) PSW.2- OV (overflow)

d) PSW.0- P(Parity)

- The remaining 2 of the flags are called as status flags, which are used to select the register banks.

a) PSW.3- RS0 (Register bank select bit 0)

b) PSW.4 -RS1 (Register bank select bit 1)

Brief description of the above mentioned PSW bits are as follows.

CY -Carry flag (PSW.7)

- The carry flag is set (1) whenever there is a carry out from the MSB(D7) bit of the result obtained.
- Its value is affected after an 8-bit addition or subtraction operation.
- Hence, If CY=1 (set)-means carry occurred at MSB (D7) bit
  If CY=0(reset)- means no carry at MSB (D7) bit.

AC -Auxiliary Carry flag (PSW.6)

- If there is a carry from D3 to D4 bit during an ADD or SUB operation, the AC bit is set; otherwise, it is cleared. It is used for the instruction to perform binary coded decimal arithmetic.
- Hence, if AC=1 - Auxiliary Carry Occurred at bit 3
  if AC=0 - Auxiliary Carry Not Occurred at bit 3

OV-Overflow flag (PSW.2)

- This flag is set whenever the result of signed number operation is too large causing the high order bit to overflow in to the sign bit.
- It is used only to detect errors in signed arithmetic operations
- An 8-bit signed number has a range **-80H.....0.....+7FH.** Any result out of this range will cause an overflow.
- Hence, if OV=1 - Overflow occurred
  If OV=0 - Overflow not occurred

P-Parity flag (PSW.0)

- The parity flag represents the number of 1's in the accumulator (A) register only.
- If the 'A' register contains odd number of 1's then P is set & for even number of 1's, P is reset.
- Hence, if P=1 - odd number of 1's in A register
  If P=0 - Even number of 1's in A register

F0-User defined flag (PSW.5)

- Flag 0 available to user for general purpose
- It can be used by us to store any user defined information

PSW.1: Undefined flag.

RS0 and RS1:

We can select corresponding register bank bit using RS0 and RS1. By default, RS1 and RS0 are 0 always and bank 0 is selected. Other banks are selected as per below table;

RS1	RS0	Register bank	Address
0	0	Bank 0	00-07H
0	1	Bank 1	08-0FH
1	0	Bank 2	10-17H
1	1	Bank 3	18-1FH

Functions or Uses of PSW

1. There are instructions in 8051, that tests the condition of flags in the PSW registers & make decision based on status of the flags. Thus, the programmer uses these flags to change the

program control (using conditional branching). Ex-JC- Jump if carry flag is set JNC-Jump if carry flag is not set etc...

2. Programmer can select register bank by setting corresponding bits in PSW.

Examples:-

1) Show the status of the CY, AC and P flags after the addition of 38H and 2FH in the following instructions.

MOV A, #38H

ADD A, #2FH ; after the addition A=67H, CY=0

Solution:

```
 38    00111000
+2F    00101111
 67    01100111
```

CY=0 since there is no carry beyond the D7 bit AC=1 since there is a carry from the D3 to the D4 bit
P=1 since the accumulator has an odd number of 1s (it has five 1s)

2) Show the status of the CY, AC and P flags after the addition of 9CH and 64H in the following instructions.

MOV A, #9CH

ADD A, #64H ;after the addition A=00H, CY=1

Solution:

```
 9C    10011100
+64    01100100
100    00000000
```


CY=1 since there is a carry beyond the D7 bit
AC=1 since there is a carry from the D3 to the D4 bit
P=0 since the accumulator has an even number of 1s (it has zero 1s).

3) Examples using sample data in a pictorial representation

6.3.2 Special Function Registers (SFRs)

- There are 21 special function registers at the top of the internal RAM.
- The address from 00 to 7F (lower 128 bytes) refers to internal RAM and the upper 128 bytes from 80 to FF are used for SFR's.
- The special function registers are used to control specific functionality of the 8051 microcontrollers.

The two important features of SFR are:

1. The address from 80 to FF (upper 128 bytes of RAM) are used for SFRs. SFRs can be accessed by their name or by their addresses.
2. Not all the address (80 to FF) are used by SFRs. Some of the addresses are reserved and must not be used by the programmer

Register Name	Name	Internal RAM address
A	Accumulator	0E0h
B	B register	0F0h
PSW	Program Status Word	0D0h
SP	Stack Pointer	81h
DPL	Data Pointer Low	82h

DPH	Data Pointer High	83h
P0	Port0	80h
P1	Port1	90h
P2	Port2	0A0h
P3	Port3	0B0h
IP	Interrupt Priority	0B8h
IE	Interrupt Enable	0A8h
TMOD	Timer Mode Control	89h
TCON	Timer/Counter control	88h
TH0	Timer0 high byte	8Ch
TH1	Timer1 high byte	8Dh
TL0	Timer0 low byte	8Ah
TL1	Timer1 low byte	8Bh
SBUF	Serial Buffer	99h
SCON	Serial Control	98h
PCON	Power Control	87H

Lab Exercises:

Expt 1. Identification of program development tools.

Program development is a process of program design, development, implementation, translation, testing, debugging and maintenance. The availability of right tools makes the program development process easier.

The program development tools (software) include the following.

(1) Text Editors

(2) Translators

(3) Linkers

(4) Loaders

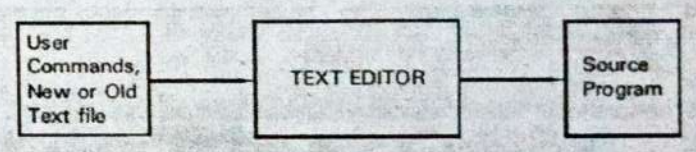
(5) Debuggers

(6) EXE to COM converters

(7) Library builders

Text Editor:

- Text editor allows the user to enter and modify the source program. It provides a set of commands for insertion, deletion, modification of a single character, statement or block of statements, etc. Such data may be a program written in a high-level language, a report in natural language, or a numerical input.



- It is a software used to review and modify text materials and other programs interactively. It can be a text editor, screen editor, etc.
- Some examples of editors are Norton Editor.
- (NE), DOS editor (EDIT), Non Document Mode of WordStar (WS), Visual Editor (VI), etc.

Translators:

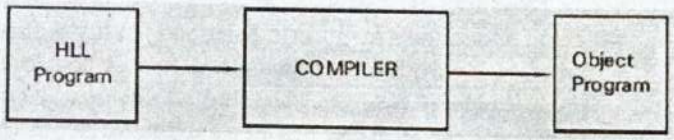
Each machine executes code in its own machine language. High-level languages are oriented toward solving a problem

9 R 6

rather than detailing the work of the machine. Programs have been developed to translate from these user-oriented languages to machine-oriented languages. These programs are called translators.

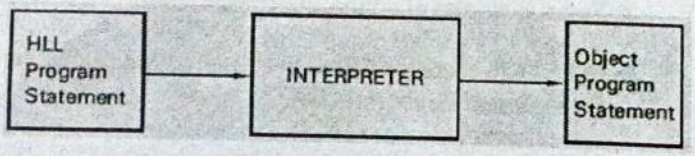
Translators can be classified into the following four categories:

1. Compilers:



- A program that translates a high-level language into a machine language program is called a complier. The input to the compiler is a source file. The compiler produces the output as a relocatable object code.
- A complier checks all kinds of limits, ranges, errors etc. It reports program errors during translation. Compiler occupies large memory space and takes longer time to produce results.
- Examples: Borland or Microsoft compiler for C, Turbo Pascal, etc.

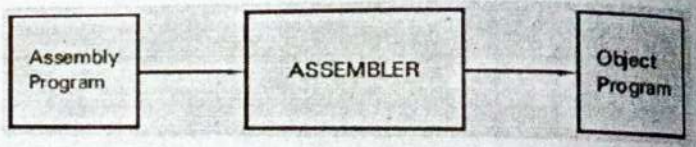
2. Interpreter:



- Interpreter translates high-level language program into object codes, statement by statement.

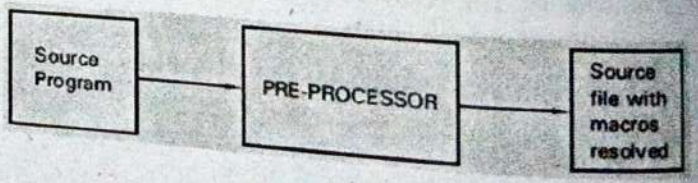
- It does not translate the entire program at a time. It takes up one statement of high-level language program at a time, translates it and executes it. Thereafter it takes up the next statement of the high-level language program and repeats the process.
- An interpreter occupies less memory space; it is cheaper and suitable for a smaller system.
- Example: BASIC interpreter

3. Assembler:



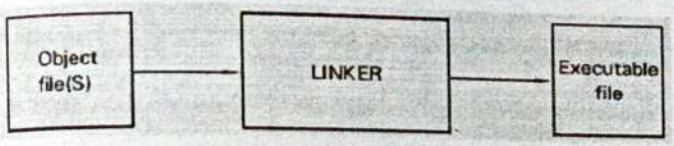
- Assembler is a software that translates assembly language program to a machine language program.
- The input to the assembler is known as a source file. The assembler produces the output as relocatable object code. It reports program errors during translation.
- Some examples of assemblers are Borland's Turbo assembler (TASM), Microsoft's Macro assembler (MASM), etc.

4. Preprocessor:



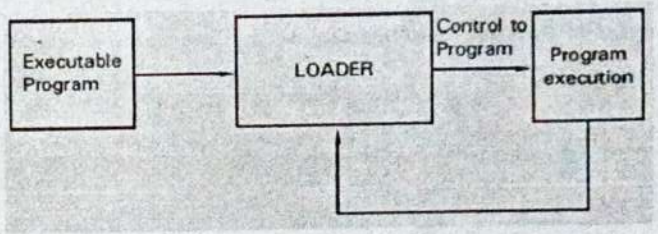
- Pre-processor translates source program with macros and header files to a program with the macros resolved and header files included.
- The pre-processor's output is transferred to the next phase of program translation. Pre-processors are built into assemblers or compilers.
- Examples of translators using pre-processors are: Borland's Turbo assembler (TASM), Microsoft's Macro assembler (MASM), High Level Language Compilers such as Borland C, Microsoft C, etc.

Linkers:



- When writing large programs, it is usually much more efficient to divide them into smaller modules. Each module can be individually written, tested and debugged. When all the modules work, they can be linked together to form a large functioning program.
- Linker is a software that joins together several object files and library functions into one large executable file. It resolves the external symbols (variables) referenced in one module and defined in other modules.
- It reports linking errors if external symbols cannot be resolved.
- Examples: Borland's Turbo linker (TLINK), Microsoft's linker (LINK), etc.

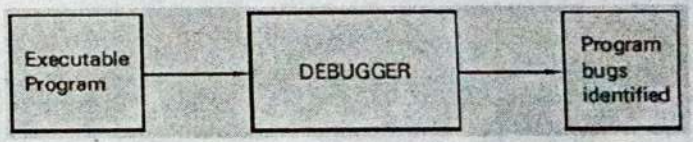
Loaders:



- A translator can generate relocatable object code and leave it in a file on a secondary storage device. Linker combines relocatable object program modules and library functions into an executable File.
- The loader is a program that moves the executable file produced by a linker from the secondary storage device to memory for execution. After loading the program in main memory, the loader transfers control to the program for execution.
- The operating system shell functions as a loader. Example: The MS-DOS shell COMMAND.COM.

Debugger:

- Debugger, is a program that allows the execution of a program in single step mode under control of the user. The process of locating and correcting an error using a debugger is known as debugging.



- A debugger executes the object program generated by an assembler or compiler with debug options. The debuggers that facilitate program debugging at source level are known as source level debuggers.

➢ Debuggers provide the following facilities:

(1) Single stepping

(2) Setting breakpoints in the program

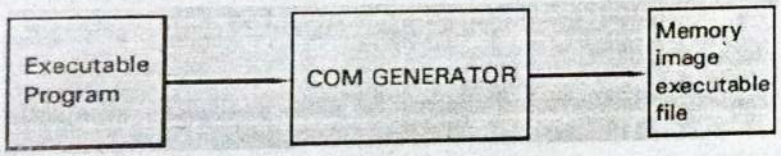
(3) Initiating a debug conversation when control reaches a breakpoint

(4) Program tracing

(5) Displaying values of variables

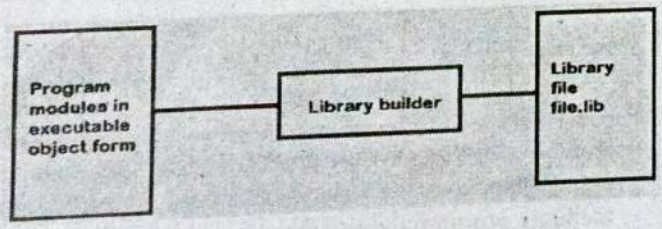
(6) It allows the user to inspect the registers and the memory locations after a program has executed.

COM CONVERTER



EXE to COM converter is a software that converts executable file (EXE) to a command file (COM). Some examples of EXE to COM converter are TLINK/T, EXE2BIN etc.

Library builder:



➢ Library builder is a program that combines all the program modules into a single file known as library. The library can be shared by multiple users for program development.

➢ It includes very frequently used functions. The compiler developers supply the built-in functions in the form of a library.

➢ Examples: Microsoft's LIB, Borland's TLIB utilities, etc.

Expt 2. Familiarization of program development using Keil.

The µVision3/4 IDE is a Windows-based software development platform that combines a robust editor, project manager, and makes facility. µVision3/4 integrates all tools including the C compiler, macro assembler, linker/locator, and HEX file generator.

The installation steps for keil software are given below:

1. Double click on Keil µvision exe file.
2. Then click on Next.
3. Tick the check box towards to license agreements and click Next.
4. Select Destination folder and click Next.
5. To fill the names and e-mail ID in the text boxes then click Next.
6. Finally click on Finish

The keil software flow

First open the icon keil µvision3/4 and the following steps are given below. The menu bar provides you with menus for editor operations, project maintenance, development tool option settings, program debugging, external tool control, window selection and

manipulation, and on-line help. The toolbar buttons allow you to rapidly execute µVision3/4 commands. A Status Bar provides editor and debugger information. The various toolbars and the status bar can be enabled or disabled from the View Menu commands. To create and execute an 8051 ASM or C code, following steps are followed.

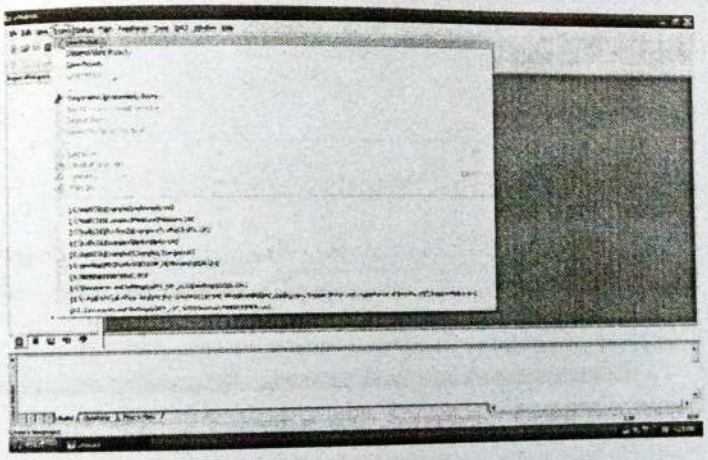
- Creating a Project
- Simulation in IDE
- HEX file creation
- Uploading and execution through Flash Magic Software Flow

Creating a Project

This will explain the steps required to setup a simple application.

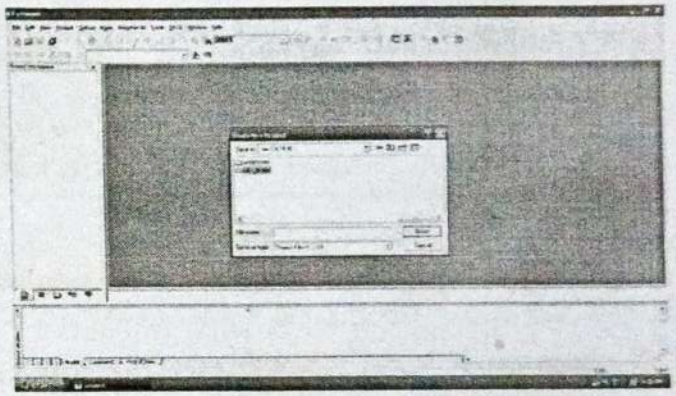
STEP 1: Go to "Project" and close the current project "Close Project"(If any previous project is opened).

STEP 2: Go to the "Project" and click on "New Project"



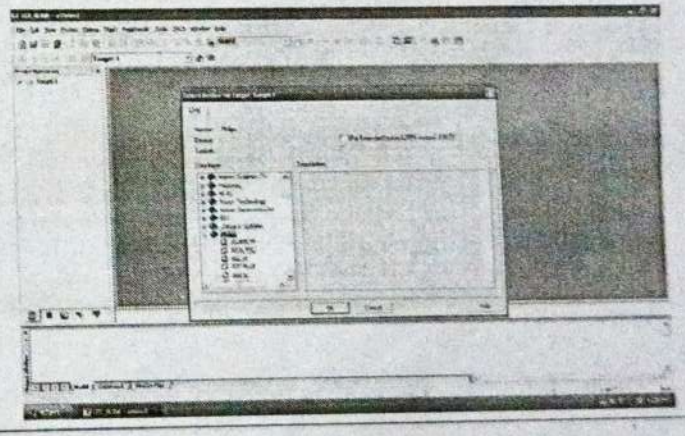
STEP 3: Here we can see a small window as "Create New Project" and here we can select our destination to wherever we want.

STEP 4: Open any drive and create a "New Folder" and to give any relevant name related to the Project.

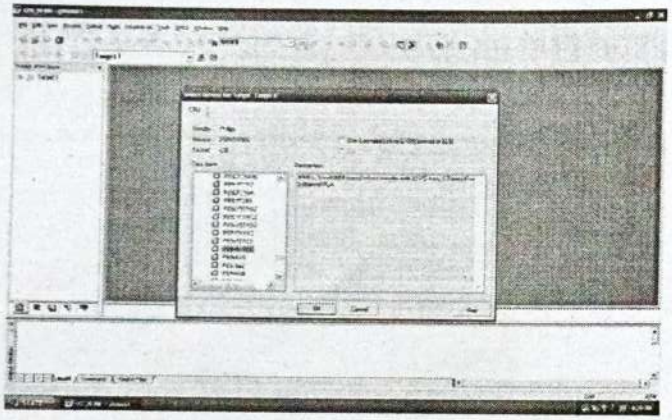


STEP 5: Select and open the created folder and give a file name.

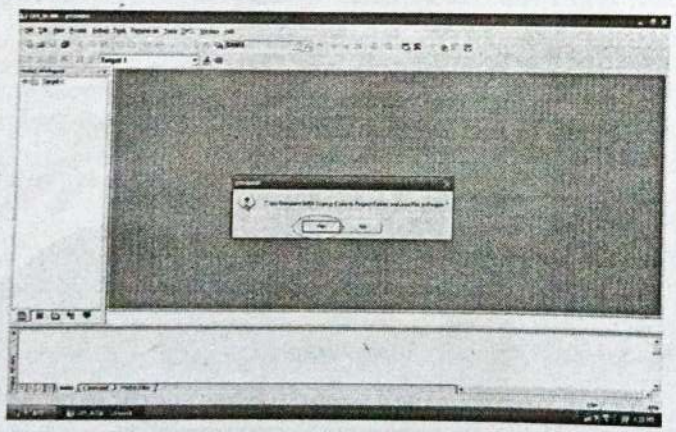
STEP 6: Again, we can see a small window as "Select Device for Target 'Target 1'", here select Philips.



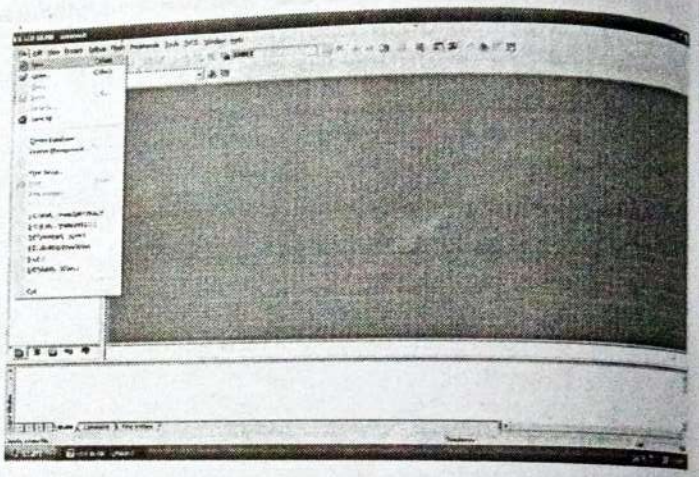
STEP 7: Within the **Philips** select P89V51RD2 or within atmel select AT89C51RD2.



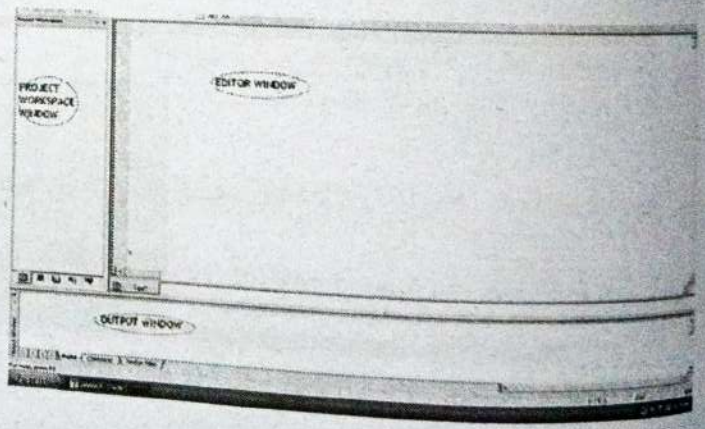
STEP 8: If we want to add 8051 Start-up file we can click "Yes".



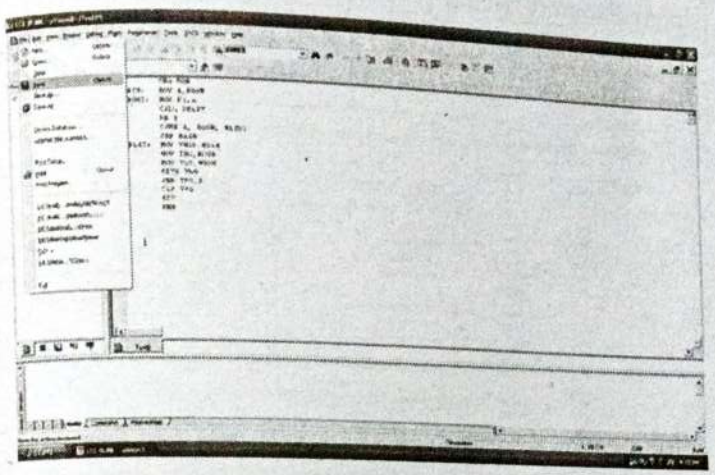
STEP 9: Next go to "File" and click on "New".



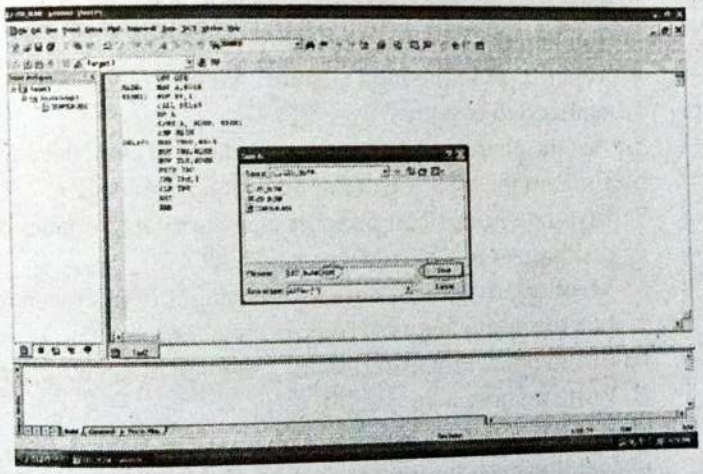
STEP 10: These are the main three windows in the keil IDE (Integrated Development Environment). One is Project Workspace second is Editor Window and third is Output Window.



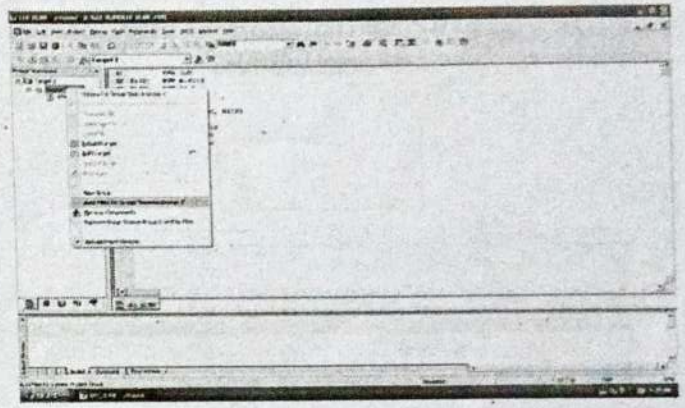
STEP 11: In editor window we can start to write our program.



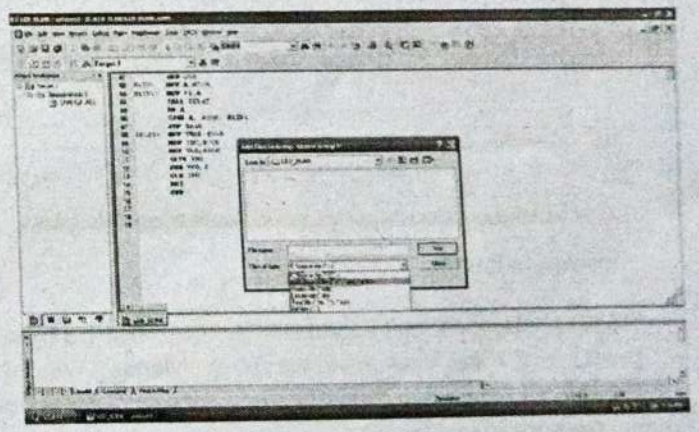
STEP 12: Save the file, if the program is in "C" save as "filename.C" or else if it is in ASM save as "filename.ASM".



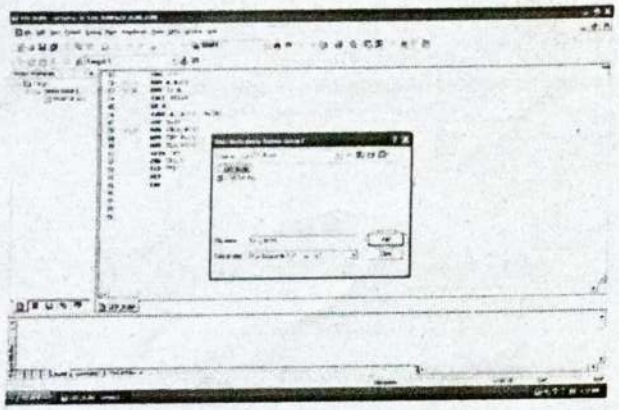
STEP 13: Next add this source file to Group1, for that go to "Project Workspace" drag the "Target 1" in that right click on "Source Group1" and click on "Add Files to Group "Source Group1"".



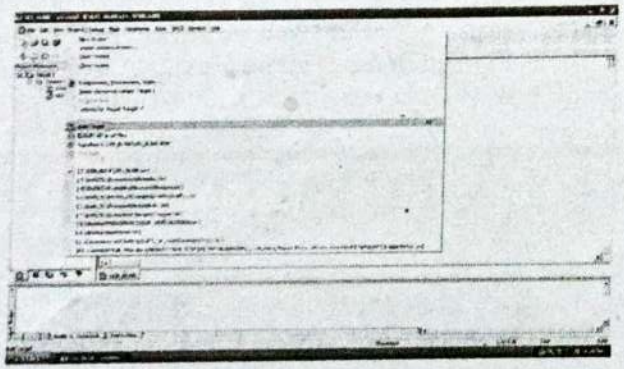
STEP 14: Here one small window will open as "Add Files to Group "Source Group1", default the Files of type will be in C source Files (*.C). If our program is in C we need to select C source Files (*.C) or select ASM Source file (*.s,*.src,*.a*).



STEP 15: Select the source file and add the file then close the small window.



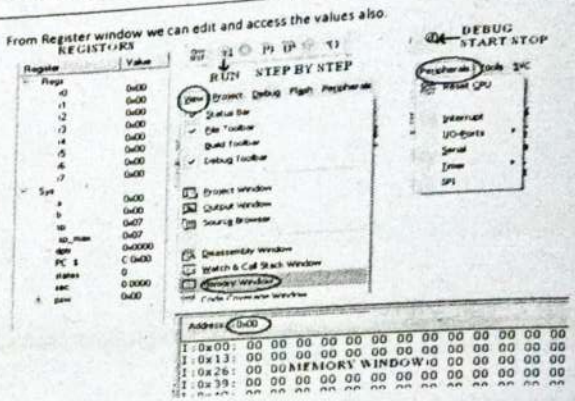
STEP 16: Then go to "Project" click on "Build Target" or F7. There we can see errors and warning in Output Window.



Simulation Part

STEP 17: Go to "Project" click on "Rebuild all target Files" and start Debug. From View menu we can get Memory Window and from Peripherals we can get I/O ports, Serial etc. For accessing the memory space,

Memory type	Text to type
Internal memory(data)	d:0x_memory location E.g., d:50h
External data memory (64K)	x:0x_memory location E.g., x:1234h
External code memory (64K)	c:0x_memory location E.g., c:1234h



Review Questions :

1. List the features of 8051 microcontroller.
2. Differentiate between a microcontrollers and microprocessors.s
3. Explain the significance of PSW of 8051.
4. What is an embedded system? Mention the components of embedded systems?
5. List the characteristics and applications of embedded systems.
6. Explain the architecture of 8051 microcontroller.
7. List the various components and the characteristics of each component in a8051 Microcontroller.
8. Mention advantages and disadvantages of microcontrollers.
9. List the applications of microcontroller.
10. List the different criterion for selection of a microcontroller.
11. Write a short note on SFRs.
12. Draw the architecture/block diagram of 8051.
13. List the special function registers of 8051.
14. List the functions of conditional and user flags of 8051.

Week

7

8051 Architecture Continued....

8051 Architecture Continued

7.1 Basic Registers in 8051

7.1.1 Pin Description of 8051

7.2 Memory Organization of 8051

7.2.1 Internal RAM Organization

7.2.2 Internal ROM Organization

7.2.3 Stack & Stack Pointer

7.3 Interfacing External Data & Code Memory

7.3.1 Interfacing External Program (Code) Memory

7.3.2 Interfacing External Data (RAM) Memory

* Lab Programs

* Review Questions

7.1 Basic Registers in 8051

The most widely used registers of 8051 are,

1. Accumulator (A register)
2. B register
3. Program Counter (PC)
4. Data Pointer (DPTR)

Accumulator (A register)

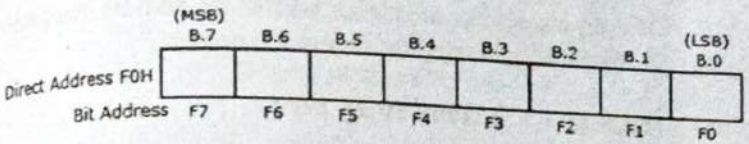
- It is an 8-bit register. It is usually accessed in both byte and bit.
- Its physical address is E0H. It holds source operand & receives the result of arithmetic instructions & logical operations (addition, subtraction, multiplication, division, AND, OR etc...)
- A register with address is shown below. To access bit '0' or to access all 8 bits (as a single byte) the same physical address E0H is used. To access the second bit (i.e. Bit1) E1H is used & for 3rd bit E2H and so on

	(MSB) ACC.7	ACC.6	ACC.5	ACC.4	ACC.3	ACC.2	ACC.1	(LSB) ACC.0
Direct Address E0H								
Bit Address	E7	E6	E5	E4	E3	E2	E1	E0

Register B:

- In addition to accumulator an 8-bit B register is available as a general-purpose register. The major purpose of this register is in executing multiplication and division

- Register B is also byte addressable & bit addressable. To access bit 0 or to access all 8 bits (as a single byte). Physical address F0H is used. To access bit 1 you may use F1H & so on. 'B' register with address is shown below

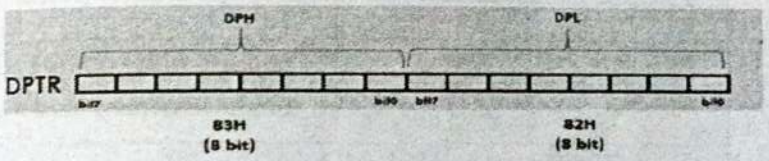


DPTR (Data Pointer) Register:

- The data pointer (DPTR) consists of a high byte (DPH) & a low byte (DPL). Its function is to hold 16-bit address. It may be manipulated as 16-bit data register or as 2 independents 8-bit registers

- Beside this, DPTR register is primarily used for external memory addressing & is also used for storing data & intermediate values.

- The DPTR doesn't have a single internal address, DPH (83H) & DPL (82H) have separate internal addresses



Program Counter (PC):

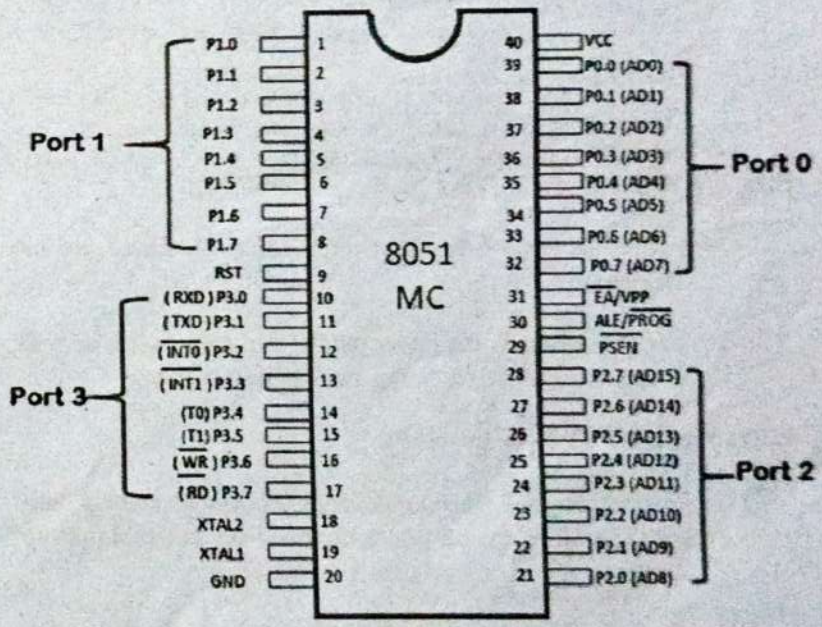
- It is a 16-bit register, which is used to hold the address of memory location from which next instruction is to be fetched.

- PC starts at 0000h when the 8051 initializes & is incremented every time after an instruction is executed.

- PC is not always incremented by 1. "Branch, jump & interrupt operations" load the PC with an address other than the next sequential location.

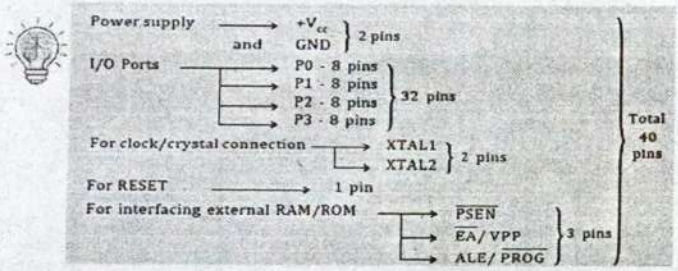
- Activating power on reset will cause all values in the register to be lost. It means the value of PC is 0000h upon reset, forcing the CPU to fetch the first opcode from ROM location **0000h**. It means we must place the first byte of opcode in ROM location **0000h**.

7.1.1 Pin Description of 8051:



In 8051, I/O operations are done using four ports (32 pins). It is a 40 pin DIP (Dual in Line package). It reserves 32 pins for I/O operations, which are arranged in 4 ports, each of 8 pins (4*8=32). The other 8 pins are designated as Vcc, Gnd, XTAL1, XTAL2, RST, EA, ALE & PSEN

The 8051 has 32 I/O pins configured as four eight-bit parallel ports (P0, P1, P2 & P3). All four ports are bidirectional that is each pin will be configured as input or output. All port pins are multiplexed except the pins of port1



PORT 0 -P0.0 to P0.7 (Pins 32-39)

- PORT0 is an 8-bit bidirectional I/O port. These are bit addressable.
- These pins act as the lower part of the address bus as well as data bus while accessing the external memory

PORT1 (Pins 1 to 8) (P1.0 to P1.7)

- Port1 (P1) is an 8-bit bidirectional I/O port with internal pull-ups. Therefore, the port does not need any external pull-up resistors.
- It functions simply as I/O port & doesn't have any alternative function

PORT2 (P2.0 to P2.7) (Pins 21 to 28) :

- PORT2 is a general purpose 8-bit input/output port when no external memory is present.
- These pins acts as the higher part of the address bus (A8-A15) while accessing the external memory.

PORT3 (P3.0 to P3.7) (Pins 10 to 17) :

- All port pins of port 3 are multifunctional. Therefore, each pin of port 3 can be programmed to use as I/O pin or as one of the alternate functions.
- They have special functions as shown below. It includes 2 external interrupts,2 counter inputs,2 special data lines for serial communication and 2 timing control pins

Pin Number	Function	Description
P3.0-10	Rx	Serial Input
P3.1-11	Tx	Serial Ouput
P3.2-12	$\overline{INT0}$	External Interrupt 0
P3.3-13	INT1	External Interrupt 1
P3.4-14	T0	Timer 0
P3.5-16	T1	Timer 1
P3.6-16	$\overline{WR}$	External Memory Write
P3.7-17	RD	External Memory Read

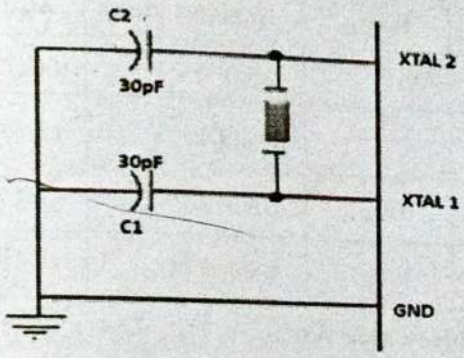
Power Supply Pins Vcc (pin 40) & Gnd (pin 20)

- 8051 operates on DC power supply of 5V with respect to ground.

- The 5V is to be connected to pin Vcc.
- Pin 20 provides ground for the reference

XTAL1, XTAL2 oscillator pins (Pin 19 & 18)

- 8051 has on chip oscillator but requires external clock to run it. A quartz crystal is connected between XTAL1 (pin 19) & XTAL2 (pin 18) on the chip.
- The crystal also needs 2 capacitors of 30pF for generating signal of desired frequency. One side of each capacitor is connected to crystal & other sides are connected to ground.



ALE or Address Latch Enable (pin 30)

- AD0 to AD7 lines are multiplexed. To demultiplex these lines for lower half of an address an external latch & ALE signal of 8051 is used.
- When ALE=0 then P0 pins work as Data bus & when ALE=1 then the P0 pins acts as address bus. ALE is active high pin.

RST (Reset Pin 9)

- This pin is used to reset 8051. For proper reset operation, reset signal must be held high at least for 2 machine cycles.
- Upon reset microcontroller will terminate all its activities, the value in the register will be lost and it will set the program counter (PC) to 0000h

$\overline{\text{PSE}}$(Program Store Enable, Pin 29)

- It is the active low output control signal used to activate enable signal ($\overline{\text{OE}}$) of the external "ROM/EPROM ". This signal acts as the read strobe(pin) to external program memory.

(External access, Pin 31)

- It is an active low input pin & is activated upon low pulse
- When $\overline{\text{EA}}$ =1, program fetches address 0000H to 0FFFH(**4KB**) are directed to internal ROM & program fetches to addresses 1000H to FFFFH (**60KB**) are directed to external ROM/EPROM
- When $\overline{\text{EA}}$ =0, all addresses (0000H to FFFFH) (**64KB**) fetched by program are directed to external ROM/EPROM.

7.2 Memory Organization of 8051 :

Depending upon the storage of program/data either inside or outside the 8051 chips, memories are classified in to 2 types;

a. On chip memory (internal memory)

b. External memory

The above 2 memories are again classified into 2 sub types based on the type of content that is stored in it

a. **Program memory (ROM)**- It is used to store the program being executed. It is un editable (Can't make any changes) once burnt

b. **Data memory (RAM)**- It is used as temporary storage to store data & intermediate results during operation of microcontroller

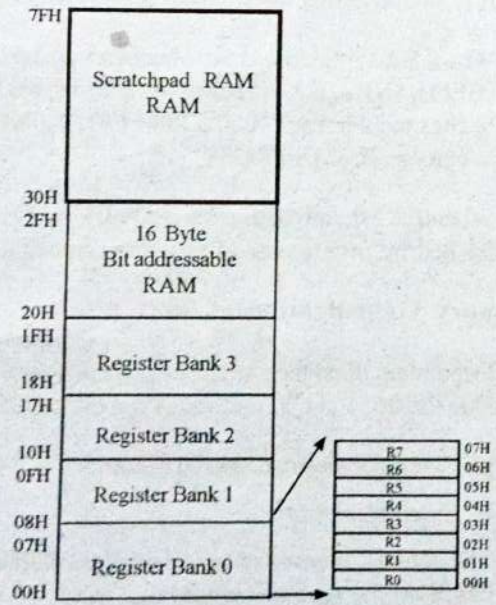
7.2.1 Internal RAM Organization :

The 8051 has 128 bytes internal RAM (data memory). It is accessed using RAM address register. As shown in the below figure the internal RAM of 8051 is organized in to 3 distinct areas

a. Working register area (00H-1FH)-32 bytes

b. Bit addressable area (20H-2FH)-16 bytes

c. General purpose area (30H-7FH)-80 bytes



Working Registers (Register Banks)

- The first 32 bytes from 00H to 1FH of internal RAM constitutes 32 working registers. They are organized into 4 banks of eight registers each.
- The 4 register banks are numbered 0 to 3 and they consist of eight registers named R0 to R7 Each register can be addressed by name or by its RAM address
- Only one register bank is in use at a time. Bits RS0 & RS1 in the PSW determine which bank of registers is currently in use

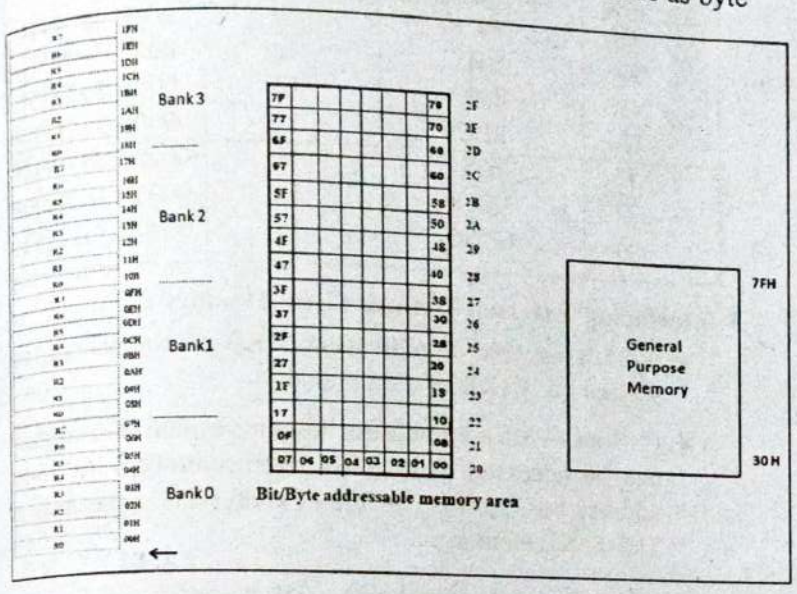
RS1	RS0	Register bank	Address
0	0	Bank 0	00-07H
0	1	Bank 1	08-0FH
1	0	Bank 2	10-17H
1	1	Bank 3	18-1FH

Bit/Byte Addressable Area

- The 8051 provides 16 bytes of bit addressable area. It occupies RAM byte address from 20H to 2FH, forming a total of 128 (16X8) addressable bits
- An addressable bit may be specified by its bit address of 00H to 7FH or 8 bits may form any byte address from 20H to 2FH
- For example, bit address 4EH refers to bit 6 of byte address 29H

General Purpose Area

➢ The RAM area above bit addressable area from 30H to 7FH is called general purpose RAM. It is addressable as byte

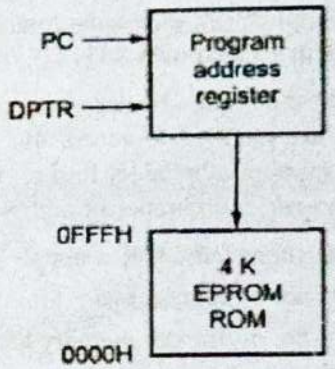


7.2.2 Internal ROM Organization

➢ The 8051 has 4KB of internal ROM with address space from 0000H to FFFFH.

➢ It is programmed by manufacturer when the chip is built. This part cannot be erased or altered after fabrication(soldering)

➢ This is used to store final version of the program and is accessed using program address register



7.2.3 Stack & Stack Pointer

➢ Stack is a part of RAM used by the CPU for temporary storage of data or address. An 8-bit register called stack pointer is used to access the stack. On power up or upon reset the stack pointer holds address 07h. This means that 08 is the first location used for the stack.

➢ The stack is of last in first out type. The stack pointer is always pointing to the top of the stack and incremented by one whenever the data is pushed on to the stack. The stack pointer will be decremented by one whenever data is retrieved from the stack.

➢ The stack is useful when the program execution sequence is to be altered during subroutine call OR jumping to interrupt service routines

Need of External Memory Interfacing and memory addressing

➢ If 4K on chip ROM and 128 bytes of RAM is insufficient for

user for the application, externally total of 64K ROM and RAM can be interfaced with 8051.

➢ Eight-bit address is required to access internal RAM (00 - 7F). Twelve bits are required to access 4K internal ROM. In general, the number of memory locations in a chip is equal to 2 to the power of a number of address pins

i.e., 2K locations OR 2**K × nbits

where K= no. of address bits

n = no. of bits per memory location

For example, the no. of address lines for 8K and 16K memory can be calculated as follows

For 8k byte memory

2^3 1K byte= 3 + 10 (10 lines for 1K byte)

= 12 address lines are required to address 8Kb memory

The address ranges from

0	0000	0000	0000		(13 lines)
To :	: :	: :	: :	: :	min
	: :	: :	: :	: :	To
1	1111	1111	1111		(13 lines max)

For 16K byte memory

= 16 1K

= 2^4 1K = 4 + 10 address lines

= 14 address lines are required to address 16KB memory

The address range from	00	0000	0000	0000
To	11	1111	1111	1111

i.e., from 0000 to 3FFFH

Similarly for 32K = 2^5 1K = 5 + 10 = 15 address lines

address ranges from	000	0000	0000	0000
To	111	1111	1111	1111

i.e., from 0000H to 7FFFH

The no. of address and the address ranges for different memory sizes are as shown in table 1.3.

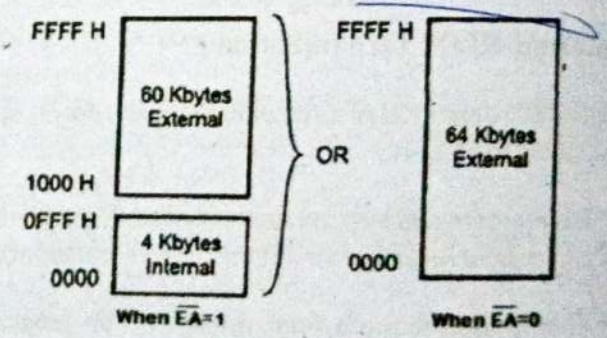
Memory size	Capacity in decimal(Bytes)	No.of address lines	Address range in Hex.
1K	1024	10	000H To 3FFH
2K	2048	11	000H To 7FFH
4K	4096	12	000H To FFFH
8K	8192	13	0000H To 1FFFH
16	16384	14	0000H To 3FFFH
32	32768	15	0000H To 7FFFH
64	65536	16	0000H To FFFFH

7.3 Interfacing External Data & Code Memory :

➢ 8051 is capable of addressing 64KB of external program memory & 64KB of data memory

➢ (* Note - With n bit address bus a maximum of 2n memory can be accessed & as 8051 microcontrollers has 16-bit address bus, so it is capable of addressing a maximum of 216=64KB memory)

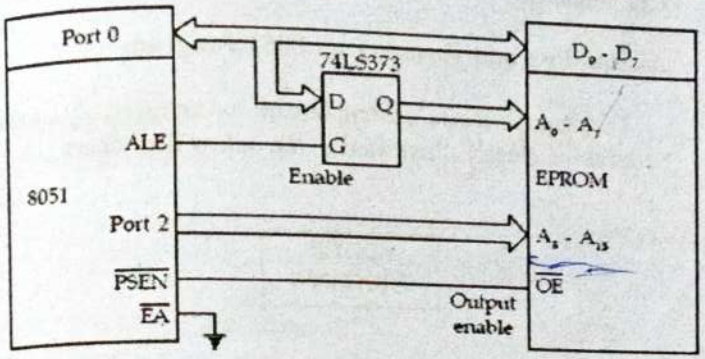
7.3.1 Interfacing External Program (Code) Memory



➢ In 8051 when $\overline{EA}$ pin is connected to Vcc, Program fetches to address from 0000H to 0FFFH are directed to the internal ROM and program fetches to address from 1000H to FFFFH are directed to the external ROM/EPROM

- On the other hand, when $\overline{EA}$ pin is grounded, all addresses (0000H to FFFFH) fetched by program are directed to external ROM/EPROM

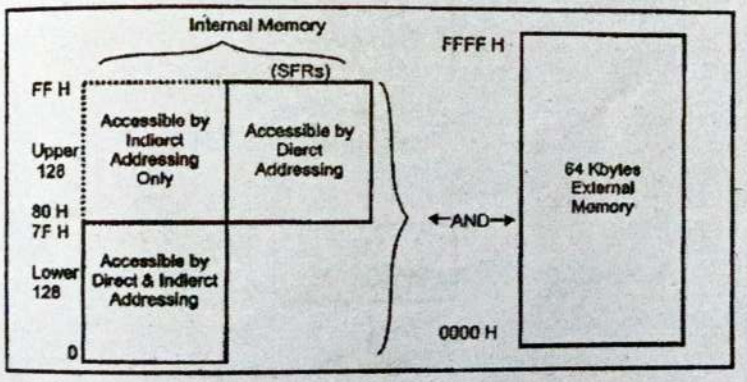
- The signal $\overline{PSEN}$ is used to activate output enable ($\overline{OE}$) signal of the external ROM/EPROM as shown in figure below



As shown in figure PORT0 provides lower order 8-bit address & PORT2 provides higher order 8-bit address

Ex: MOVC A, @A+DPTR; Copy the contents of external ROM address formed by adding A & DPTR to A

7.3.2 Interfacing External Data (RAM) Memory

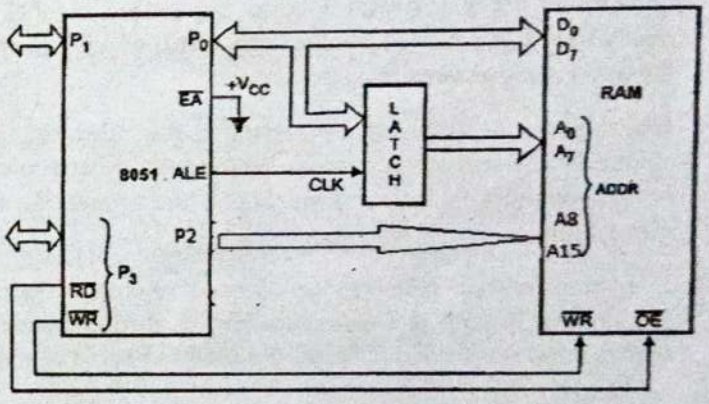


- The 8051 can address up to 64 KB of external memory. The "MOVX" instruction is used to access external data memory

- Below figure shows circuit diagram for connecting external data memory.

- The multiplexed address/data bus provided by PORT0 is demultiplexed by external latch & ALE signal. PORT2 gives higher order address bus.

- The & signals from 8051 selects memory read & memory write operations respectively.



Example: MOVX A, @DPTR; Copy the content of external address in DPTR to A

Lab Program :

1) Familiarize with the structure of the 8051- assembly program and executing it.

Structure of Assembly Language

The assembler instruction format of 8051 is as shown below

Label : Mnemonic Operands; Comments

Label : It is an identifier used to assign the address of the first byte of an instruction. It is a user defined word used in branch instructions to branch the flow. It is optional

Mnemonic : The colon (:) operator separates the label and mnemonic. The mnemonic is the code word, which represents the operational code of an instruction

Operands : A single space separates the mnemonic and operands. The instruction may have zero operands, one operand or two operands which depends upon the operation of an instruction.

If an instruction is having two operands, a comma operator separates each operand. The first operand is called as destination operand and the second operand is called as source operand.

Comments : The semicolon (;) operator separates the operands and comments. This field is an optional, which represents the operation of an instruction flow.

Example:

Up: ADD A, R0 ; Adds contents of R0 to ACC and stores the result in ACC

Where,

Up → label

ADD → Mnemonic

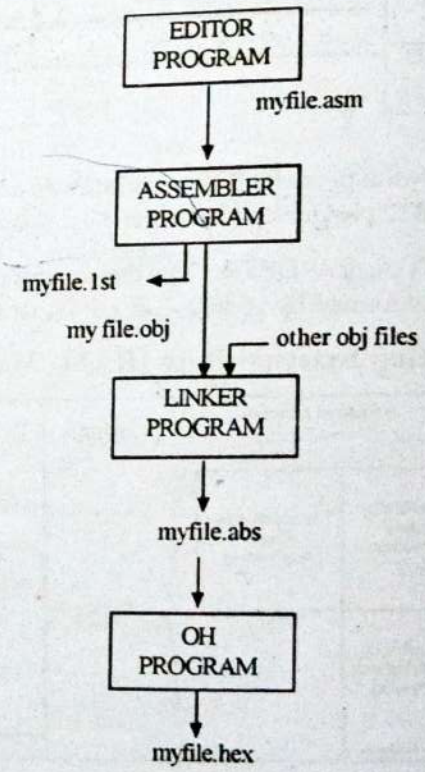
A → Destination operand

R0 → Source operand

Adds contents of R0 to ACC and stores the result in ACC→ Comments lin

Assembling and Running an 8051 Program

The steps to create, assemble, and run an assembly language program is clearly illustrated in the below flow chart



- First, we use an editor to type in a program. Editors like MS-DOS EDIT program that comes with all Microsoft operating systems can be used to create or edit a program. The Editor must be able to produce an ASCII file. The ".asm" extension for the source file is used by an assembler in the next step.

- The ".asm" source file is fed to an 8051 assembler. The assembler then converts the assembly language instructions into machine code instructions and produces an .obj file (object file) and a .lst file (list file). It is also called as a source file and may have ".src" extensions also.

- The ".lst" file is optional and is useful to lists all the opcodes, addresses as well as errors that the assemblers detected.

- Assemblers require a third step called linking. The link program takes one or more object files and produces an absolute object file with the extension ".abs".

- Next, the ".abs" file is fed to a program called "OH" (object to hex converter), which creates a file with the extension ".hex" that is ready to burn in to the ROM.

2. Write and execute simple ALP to understand different addressing modes.

Label	Mnemonic	Description
	ORG 0000H	; Initialize starting address of Program.
	MOV R0,#51H	; *Immediate addressing mode* *Copies value 51h into R0 register*
	MOV R2,#02H	; *Immediate addressing mode* Copies value 02h into R2 register
	MOV A,50H	; *Direct addressing mode*

		Copies content of memory location 50h into accumulator
	ADD A,R2	; *Register addressing mode* Performs A←(A) + (R2)
	MOV @R0,A	; *Register indirect addressing mode* Copies accumulator content to memory location pointed by R0 register
LOOP	:SJMP LOOP	; Stop
	END	; End

Result :

Input :

Memory	50h
Data	03H

Output :

Memory	51h
Data	05h

Review Questions :

1. Explain the function of program counter.
2. Draw the pin diagram of 8051 and explain each pin.
3. Explain the control signals provided by port 3.
4. Explain memory organization of 8051 microcontroller.
5. Explain the internal RAM organization of 8051.
6. Name the pins of 8051 used for external memory interfacing and list their functions.
7. Explain the functions of regA, regB, PC,SP and DPTR.
8. Describe the method of interfacing 2K data RAM to 8051 microcontroller.
9. Explain the register banks used in 8051 with neat sketch.
10. With a neat diagram explain how external RAM can be interfaced with 8051.
11. Explain the method of interfacing external Code memory with 8051.

Week 8

8051-Instruction Set

8051 - Instruction Set

8.1 Addressing Modes of 8051

8.2 8051 Instruction Set

8.2.1 Data Transfer Instructions

8.3 Arithmetic Instructions

8.3.1 Logical Instructions

* Lab Exercises

* Review Questions

8.1 Addressing Modes of 8051

The various ways of accessing the data are called as addressing modes. Or The method of specifying source of operand and output of result in an instruction is known as addressing mode.

The 5 major 8051 addressing modes are as follows :

1. Immediate Addressing
2. Register Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Indexed Addressing

1. Immediate Addressing

➢ In the Immediate Addressing mode, the source operand is an 8- or 16-bit constant value. This constant is preceded by '#' symbol in the instruction itself (rather than in a register or a memory location).

➢ The destination register should hold the same data size which is specified by the source

Examples:

```
ADD A, #30H          ; Here immediate data is 30H, of 8-bit
MOV DPTR, #0FE00H    ; Here immediate data is FE00H, of
                       16-bit
```

2. Register Addressing

- In the Register Addressing mode, the instruction involves transfer of information between registers that is both source & destination operands are registers only.
- Hence the user must provide the name of register from where the operand/data need to be fetched
- **Note :** Both operands can't be bank registers (R0-R7) simultaneously

Examples :

ADD A, R0	;	Content of R0 register is copied in to accumulator
MOV R4, A	;	Content of accumulator is copied in to R4 register

3. Direct Addressing

- In this mode the direct address of memory location is provided in instruction to fetch the operand.
- Only internal RAM and SFR's address can be used in this type of instruction. But at most only one operand is used as address and other operand must be a register

Examples :

MOV 46H, A	;	Copies content of accumulator in to address 46H
MOV A, 51H	;	Copies content of address 51H in to accumulator

4. Register Indirect Addressing

- In Register Indirect Addressing Mode, as its name suggests the data is read or stored in register indirectly. The '@' sign indicates that the register holds the address of memory location
- The registers which hold the address are called pointer/index registers. They are used to point the operand
  - ✓ Only registers R0, R1 and DPTR can be used as pointer registers.
  - ✓ R0 and R1 registers can hold an 8-bit address whereas DPTR can hold a 16-bit address.
  - ✓ R0 and R1 are useful in accessing operands which are in the internal memory whereas DPTR is useful in accessing operands which are in the external memory.

Examples :

MOV @R0,A	;	Load contents pointed by R0 register in to accumulator
MOVX A,@DPTR	;	Copy the contents of external data memory pointed by DPTR in to accumulator

5. Indexed Addressing

Only program memory (ROM) can be accessed in index addressing. Either DPTR or PC can be used as an index register. Its operation is similar to indirect addressing mode

Examples :

MOVC A,@A+DPTR;

Here C means code memory. Here content of A register is added with content of DPTR & resultant is the address of memory location from where data to be copied in to 'A' register

8.2 8051 Instruction Set

The instructions of 8051 are divided into five functional groups as below :

- Arithmetic instructions
- Logical instructions
- Data transfer instructions
- Boolean variable instructions
- Program branching and machine control instructions

Arithmetic instructions are useful to perform Arithmetic operations such as addition, subtraction, multiplication, division, increment, decrement etc. Arithmetic operations effect the flags, such as Carry Flag (CY), Overflow Flag (OV) etc, in the PSW register.

Logical instructions perform standard Boolean operations such as AND, OR, XOR, NOT (compliment) etc. Other logical operations are clear accumulator, rotate accumulator left and right, and swap nibbles in accumulator etc.

Data transfer instructions are used to transfer data between an internal RAM location and SFR location without going through the accumulator. Data can also be transferred between the internal and external RAM by using indirect addressing.

The Boolean variable instructions of processor can perform single bit operations. The Boolean Variable operations include set, clear, as well as and, or and complement instructions. Also included are bit-level moves or conditional jump instructions. All bit accesses use direct addressing

Program branching and machine control instructions are used to control the flow of actions in a program. Some instructions provide decision making capabilities and transfer control to other parts of the program e.g., conditional and unconditional branches

8.2.1 Data Transfer Instructions

The data transfer instructions of 8051 are MOV, MOVC, MOVX, XCH, XCHD, PUSH and POP are discussed below in detail with examples

MOV : Move byte Variable

The byte variable indicated by the source operand is copied into the location specified by the destination operand. The source byte is not affected. No other registers or flag is affected. This is by far the most flexible operation. Fifteen combinations of source and destination addressing modes are allowed.

Syntax :

MOV dest_operand, src_operand

Operation :

dest_operand ? src_operand

Src_operand is copied into destination operand. Source Operand can be Register, memory location or immediate value and destination operand can be Register or memory location as illustrated below

MOV A,Rn	; A←Rn, Where Rn = R0 to R7
	MOV A,direct
	; A ← [direct memory content]
MOV A,@Ri	; A ← [memory content pointed to by Ri], Where Ri = R0 or R1
MOV A,#data	; A ← immediate data
MOV Rn,A	; Rn ←A, Where Rn = R0 to R7
	MOV Rn,direct
	; Rn ← [direct memory content]
	MOV Rn, #data
	; Rn ← immediate data
MOV direct,A	; [direct memory content] ←A
MOV direct,Rn	; [direct memory content] ← Rn
MOV direct,@Ri	; [direct memory content] ← [memory content pointed to by Ri],
MOV direct,#data	; [direct memory content] ←Immediate data
MOV @Ri,A	; [memory content pointed to by Ri] ← A
MOV @Ri,direct	; [memory content pointed to by Ri]← [direct memory content]
MOV @Ri,#data	; [memory content pointed to by Ri] ← Immediate data

Examples :

MOV A, R4	**; A ← R4**
Before execution let,	A = 25H, R4 = 33H,
After execution,	A = 33H, R4 = 33H
MOV A, 40H	**; A ← [40H]**
Before execution let,	A = 25H, [40H] = 33H,
After execution,	A = 33H, [40H] = 33H
MOV A, @R1	**; A ← @R1**
Before execution let,	R1 = 50H, A = 25H, [50H] = 42H,
After execution,	A = 42H, R1 = 50H, [50H] = 42H
MOV 60H, #24H	**[60H] ← #24H**
Before execution let,	[60H] = 25H,
After execution,	[60H] = 24H

MOV DPTR, #data16: Load Data Pointer with a 16-bit constant

This is the only instruction which moves 16 bits of data at once. The 16-bit constant is loaded into the DPTR register. DPH is loaded with high-order byte, while DPL loads the lower-order byte. No flags are affected.

Syntax :

MOV DPTR, #data16

Operation:

DPTR ← data16

DPH ← higher byte

DPL ← Lower byte

16-bit data is copied into DPTR register

Example :

MOV DPTR, # 1234H

Before execution let, DPTR = 0000H, i.e., DPH = 00H and DPL = 00H

After execution, DPTR = 1234H, i.e., DPH = 12H and DPL = 34H

Instructions to access external ROM/Program memory

MOVC A,@A+ <base-reg>: Move Code byte

The MOVC instructions load the Accumulator with a code byte or constant from program memory. The address of the byte fetched is the sum of the original unsigned 8- bit Accumulator contents and the contents of a 16-bit base register, which may be either the Data Pointer or the PC.

Key points to remember in accessing external ROM/ Program memory

- All external data moves with external ROM involve A register
- MOVC is used with internal or external ROM & can address 4K of internal code & 64K of external code memory

- When PC is used to access external ROM, it is incremented by 1 before it is added to A to form physical address of external ROM

Syntax :

MOVC A,@A+ <base-reg>

Operation :

A ← [A + base_reg]

Accumulator A is added with base register. The result is acting as memory location. The content of this location is moved in to the accumulator A. Base register may be data pointer (DPTR) or Program Counter (PC) as illustrated below

MOVC A,@A+DPTR ; A ← [A + DPTR]
MOVC A,@A+PC ; PC ← PC + 1, A ← [A + PC]

Examples :

MOVC A,@A+DPTR

Before execution let, DPTR = 1234H,A = 12H
A+DPTR =12 + 1234 = [1246] = 58H
After execution, DPTR = 1234H, A = 58H

MOVC A,@A+PC

Before execution let, PC = 1234H,A = 18H
PC + 1 = 1234 + 1 = 1235H
A + PC = 12 + 1235 = [1247] = 42H
After execution, PC= 1235H, A = 42H

Instructions to access external data memory

MOVX : Move External

The MOVX instructions transfer data between the Accumulator

and a byte of external data memory, which is why "X" is appended to MOV. There are two types of instructions, differing in whether they provide an 8-bit or 16-bit indirect address to the external data RAM.

Key points to remember in accessing external data memory

- All external data moves with external RAM involve 'A' register
- While accessing external RAM, Ri can address 256 bytes & DPTR can address 64KB
- MOVX instruction is used to access external RAM or I/O address

Syntax :

MOVX dest_operand, src_operand

Operation :

dest_operand ← src_operand

Src_operand is copied into destination operand. Source and destination operand can be Register A or indirect address in R0, R1 or DPTR as illustrated below

MOVX A,@Ri ; A ← [Ri], Where Ri = R0 or R1

MOVX A, @DPTR ; A ← [DPTR]

MOVX @Ri, A ; [Ri] ← A, Where Ri = R0 or R1
MOVX @DPTR, A ; [DPTR] ← A

Examples:

MOVX A, @R1 ; A ← [R1]

Before execution let, R1 = 50H, A = 25H, [50H] = 42H,
After execution, A = 42H, R1 = 50H, [50H] = 42H

MOVX A, @DPTR

Before execution let, DPTR = 1234H, A = 12H, [1234]= 58H
After execution, DPTR = 1234H, A=58H, [1234]= 58H

MOVX @R1, A; [R1] ← A

Before execution let, R1 = 50H, A = 25H, [50H] = 42H,
After execution, A = 25H, R1 = 50H, [50H] = 25H

MOVX @DPTR, A

Before execution let, DPTR = 1234H, A=12H, [1234]= 58H
After execution, DPTR = 1234H,A=12H, [1234]=12H

XCH: Exchange Accumulator with source operand

XCH exchanges the Accumulator A with that of the Source operand. No flags are affected.

Syntax :

XCH A, Src_operand

Operation :

A > Src_operand

A and source operand contents are exchanged, the source operand can use register, direct, or register-indirect addressing as illustrated below

XCH A,Rn ; A > Rn, Where Rn is R0 to R7
XCH A,direct ; A > [direct address]
XCH A,@Ri ; A > [Ri], Where Ri = R0 or R1

Example:

XCH A, R4 ; **A > R4**

Before execution let, A = 25H, R4 = 33H,

After execution, A = 33H, R4 = 25H

XCH A, 40H ; **A > [40H]**

Before execution let, A = 68H, [40H] = 49H,

After execution, A = 49H, [40H] = 68H

XCH A, @R1 ; **A > [R1]**

Before execution let, R1 = 50H, A = 25H, [50H] = 42H,

After execution, A = 42H, R1 = 50H, [50H] = 25H

XCHD : Exchange Digit

XCHD exchanges the low-order nibble of the Accumulator (bits 3 through 0), generally representing a hexadecimal or BCD digit, with that of the internal RAM location indirectly addressed by the specified register. The high-order nibbles (bits 7-4) of each register are not affected. No flags are affected.

Syntax :

XCHD A,@Ri , Where Ri = R0 or R1

Operation :

A[3:0] > [Ri[3:0]]

A[7:4] and [Ri[7:4]] are unaffected

Lower nibble of A and [Ri] contents are exchanged where ar higher nibble of A and [Ri] are unaffected

Example:

XCHD A, @R1 ; A> [R1]

Before execution let, R1 = 50H, A = 25H, [50H] = 49H,

After execution, A = 29H, R1 = 50H, [50H] = 45H

PUSH Direct : Push onto stack

The Stack Pointer is incremented by one. The contents of the indicated variable is then copied into the internal RAM location addressed by the Stack Pointer. Otherwise, no flags are affected.

Syntax :

PUSH direct

Operation :

SP ← SP + 1

[SP] ←[direct]

This instruction supports only direct addressing mode, the instructions such as "PUSH A", "PUSH Rn" are illegal instructions.

POP Direct : Pop from stack.

The contents of the internal RAM location addressed by the Stack Pointer is read, and the Stack Pointer is decremented by one. The value read is then transferred to the directly addressed byte indicated. No flags are affected.

Syntax :

POP direct

Operation :

[direct] ← [SP] SP ←SP - 1

This instruction supports only direct addressing mode, the instructions such as "POP A", "POP Rn" are illegal instructions.

8.3 Arithmetic Instructions

The arithmetic instructions of 8051 are ADD, ADDC, SUBB, INC, DEC, MUL, DIV and DA A are discussed below in detail with examples

ADD : Addition

ADD adds the data byte specified by the source operand to the accumulator, leaving the result in the accumulator. It affects the carry flag (CY), auxiliary carry flag (AC) and the overflow flag (OV)

Syntax :

ADD A, src_operand

Operation :

A ← A + Src_operand

Src_operand adds with accumulator and stores the result in accumulator A. Source Operand can be Register, memory location or immediate value as below

ADD A, Rn	; A ← A + Rn, Where Rn = R0 to R7
ADD A, direct	; A ← A + [direct memory content]
ADD A, @Ri	; A ←A + [memory content pointed to by Ri], Where Ri = R0 or R1
ADD A, #data	; A ← A + immediate data

Examples:

ADD A, R0	**; A←A + R0**
Before execution let,	A = 25H, R0 = 32H
After execution,	A = 25H + 32H = 57H, R0 = 32H
ADD A, 40H	**; A← A + [40H]**
Before execution let,	A = 25H, [40H] = 32H
After execution,	A = 25H + 32H = 57H, [40H] = 32H
ADD A, @R1	**; A← A + @R1**
Before execution let,	A = 25H, R1 = 50H, [50H] = 32H
After execution,	A = 25H + 32H = 57H, R1 = 50H, [50H] = 32H
ADD A, #24H	**; A← A + 24H**
Before execution let,	A = 27H, Immediate data = 24H
After execution,	A = 27H + 24H = 4BH

ADDC : Addition with carry bit

ADDC adds the data byte specified by the source operand to the accumulator along with the carry flag (CY), leaving the result in the accumulator. It affects the carry flag (CY), auxiliary carry flag (AC) and the overflow flag (OV)

Syntax :

ADDC A, src_operand

Operation :

A ← A + Src_operand + CY

Src_operand adds with accumulator and carry flag and then stores the result in accumulator A. Source Operand can be Register, memory location or immediate value as below

ADDC A, Rn	; A ← A + Rn + CY, Where Rn = R0 to R7
ADDC A, direct	; A ← A + [direct memory content] + CY
ADDC A, @Ri	; A ← A + [memory content pointed to by Ri] + CY, Where Ri = R0 or R1
ADDC A, #data	; A ← A + immediate data + CY

Examples :

ADDC A, R0 **; A ← A + R0 + CY**
Before execution let, A = 25H, R0 = 32H, CY = 1
After execution, A = 25H + 32H + 1 = 58H, R0 = 32H

ADDC A, 40H **; A← A + [40H] + CY**
Before execution let, A = 25H, [40H] = 32H, CY = 0
After execution, A = 25H + 32H + 0 = 57H, [40H] = 32H

ADDC A, @R1 **; A← A + @R1 + CY**
Before execution let, A = 25H, R1 = 50H, [50H] = 32H, CY = 1
After execution, A = 25H + 32H + 1 = 58H, R1 = 50H, [50H] = 32H

ADDC A, #24H **; A← A + 24H + CY**
Before execution let, A = 27H, data = 24H, CY = 1
After execution, A = 27H + 24H + 1 = 4CH

SUBB: Subtraction with borrow (CY bit)

SUBB subtracts the specified data byte of source operand and the carry flag together from the accumulator, leaving the result in the accumulator. It affects the carry flag (CY), auxiliary carry flag (AC) and the overflow flag (OV)

Syntax:

SUBB A, src_operand

Operation:

A ← A - Src_operand - CY

Src_operand and carry bit subtracts from accumulator and then stores the result in accumulator A. Source Operand can be Register, memory location or immediate value as below

SUBB A, Rn	; A ← A -Rn -CY, Where Rn = R0 to R7
SUBB A, direct	; A ← A - [direct memory content] - CY
SUBB A, @Ri	; A ← A - [memory content pointed to by Ri] + CY, Where Ri = R0 or R1
SUBB A, #data	; A ← A - immediate data - CY

Examples:

SUBB A, R0 **; A←A - R0 - CY**
Before execution let, A = 45H, R0 = 32H, CY = 1
After execution, A = 45H - 32H - 1 = 12H, R0 = 32H

SUBB A, 40H **; A← A - [40H] - CY**
Before execution let, A = 45H, [40H] = 32H, CY = 0
After execution, A = 45H - 32H - 0 = 13H, [40H] = 32H

SUBB A, @R1 ; **A← A - @R1 - CY**

Before execution let, A = 45H, R1=50H, [50H]=32H, CY=1

After execution, A = 45H - 32H - 1 = 12H, R1 = 50H, [50H] = 32H

SUBB A, #24H ; **A← A - 24H - CY**

Before execution let, A = 27H, data = 24H, CY = 1

After execution, A = 27H - 24H - 1 = 02H

INC: Increment

Increments the content of operand and stores the incremented value in the same operand itself. It has only one operand. Operand value is 8-bit but, for DPTR operand value is 16-bit

Syntax :

INC Operand

Operation :

Operand ← Operand + 1

Operand adds with 1 and stores the result in same operand. Source Operand can be Register, memory location or data pointer register as below

INC A	; A ← A + 1
INC Rn	; Rn ← Rn + 1, Where Rn = R0 to R7
INC direct	; [direct memory] ← [direct memory content] + 1
INC @Ri	; [memory content pointed to by Ri] ← [memory content pointed to by Ri] + 1, Where Ri = R0 or R1
INC DPTR	; DPTR ← DPTR + 1

Examples:

INC A ; **A← A + 1**

Before execution let, A = 25H

After execution, A = 25H + 1 = 26H

INC R3 ; **R3← R3 + 1**

Before execution let, R3 = 44H

After execution, R3 = 44H + 1 = 45H,

INC 45H ; **[45H] ← [45H] + 1**

Before execution let, [45H] = 32H

After execution, [45H] = 32H + 1 = 33H

INC @R1 ; **@R1 ? @R1 + 1**

Before execution let, R1 = 50H, [50H] = 42H

After execution, [50H] = 42H + 1 = 43H, R1 = 50H

INC DPTR ; **DPTR? DPTR + 1**

Before execution let, DPTR = 2768H

After execution, DPTR = 2768H + 1 = 2769H

DEC: Decrement

Decrements the content of operand and stores the decremented value in the same operand itself. It has only one operand. Operand value is always 8-bit.

- **Note:** Decrement operator can't be used on DPTR (16-bit content)

Syntax :

DEC Operand

Operation:

Operand ← Operand - 1

Operand value is subtracted by 1 and stores the result in same operand. Source Operand can be Register or memory location as below

DEC A	; A ← A - 1
DEC Rn	; Rn ← Rn - 1, Where Rn = R0 to R7
DEC direct	; [direct memory] ? [direct memory content] - 1
DEC @Ri	; [memory content pointed to by Ri] ← [memory content pointed to by Ri] - 1, Where Ri = R0 or R1

Examples:

DEC A **; A← A - 1**
Before execution let, A = 25H
After execution, A = 25H - 1 = 24H

DEC R3 **; R3← R3 - 1**
Before execution let, R3 = 44H
After execution, R3 = 44H - 1 = 43H

DEC 45H **; [45H] ← [45H] - 1**
Before execution let, [45H] = 32H
After execution, [45H] = 32H - 1 = 31H

DEC @R1 **; @R1 ← @R1 - 1**
Before execution let, R1 = 50H, [50H] = 42H
After execution, [50H] = 42H - 1 = 41H, R1 = 50H

MUL AB: Multiply A with B

MUL AB multiplies the unsigned 8-bit integers in the Accumulator A and register B. The low-order byte of the 16-bit product is left in the Accumulator A, and the high- order byte in register B. If the product is greater than 255 (0FFH), the overflow flag is set; otherwise, it is cleared. The carry flag is always cleared

Syntax:

MUL AB

Operation:

B[15-8]: A[7-0] ← A X B

Multiply A with B and stores the result in B: A format

Example:

MUL AB ; B:A← A X B
Before execution let, A = 55H ,B = 17H
After execution, B = 07H ; Higher byte result
A = A3H ; Lower byte result

DIV AB: Divide A by B

DIV AB divides the unsigned eight-bit integer in the Accumulator by the unsigned eight-bit integer in register B. The Accumulator receives the integer part of the quotient; register B receives the integer remainder. The carry (CY) and overflow (OV) flags are cleared.

Exception: if B had originally contained 00H, the values returned in the Accumulator and B-register are undefined and the overflow flag are set. The carry flag is cleared in any case.

Syntax:

DIV AB

Operation:

A[Quotient]:B[Remainder] ← A / B

Divide A by B, stores the quotient in A and remainder in B

Example:

DIV AB

Before execution let, A = 09H, B = 02H
After execution, A = 04H ; Quotient
B = 01H ; Remainder

DAA: Decimal-adjust Accumulator for Addition

DAA adjusts the 8-bit value in ACC resulting from operations like ADD or ADDC and produces two 4-bit digits in packed Binary Coded Decimal (BCD) format. Effectively, this instruction performs the decimal conversion by adding 00H, 06H, 60H or 66H to the accumulator, depending on the initial value of ACC and PSW

- If ACC bits A[3-0] are greater than 9 (xxxx1010-xxxx1111), or if AC=1, then a value 6 is added to the lower nibble accumulator to produce a correct BCD digit in the lower order nibble

- If ACC bits A[7-4] are greater than 9 (1010xxxx-1111xxxx), or if CY=1, then these high order bits will be increased by 6 to produce a correct proper BCD in the high order nibble but not clear the carry

Syntax:

DA A

Operation:

If A [3-0] > 9 or AC = 1 then
A [3-0] ←A [3-0] + 6
And
If A [7-4] > 9 or CY = 1
then A [7-4] ← A [7-4] + 6

Example 1:

MOV A, H
#47H ADD A, H
#38H DA A

Since lower nibble is greater than 9, adds 6 to it as clearly illustrated below 47H

+ 38H
7FH Invalid BCD
+ 6 adds 6 after DA A
85H Final result in accumulator A

Example 2 :

MOV A, #97H ADD A, #38H DA A

Since lower nibble and higher nibble is greater than 9, adds 6 to both as clearly illustrated below

97H
+ 38H
CFH Invalid BCD
+ 66 adds 6 after DA A
135H Final result in accumulator A = 35H and CY = 1

8.3.1 Logical Instructions

The logical instructions of 8051 are ANL, ORL, XRL, CLR, CPL, RL, RLC, RR, RRC and SWAP are discussed below in detail with examples

Note: In ANL, ORL, XRL when destination is accumulator the source can use register, direct address, immediate or register indirect addressing. When destination is a direct address, the source can be accumulator or immediate data only

ANL : Logical AND for byte variables

ANL performs the bitwise logical-AND operation between the variables indicated and stores the results in the destination variable. No flags are affected. The two operands allow six addressing mode combinations.

Syntax :

ANL dst_operand, src_operand

Operation :

dst_operand ← dst_operand & Src_operand

Src_operand ANDs with dst_operand and stores the result in dst_operand. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing. When the destination is a direct address, the source can be the

Accumulator or immediate data as illustrated below

ANL A, Rn	; A ← A & Rn, Where Rn = R0 to R7
ANL A, direct	; A ← A & [direct memory content]
ANL A, @Ri	; A ← A & [memory content pointed to by Ri], Where Ri = R0 or R1
ANL A, #data	; A ← A & immediate data
ANL direct, A	; [direct memory content] ← [direct memory content] & A
ANL direct, #data	; [direct memory content] ← [direct memory content] & immediate data

Examples :

ANL A, R4 ; A← A & R4

Before execution let,	A = 25H,	00100101
	R4 = 33H,	00110011
	A & R4 =	00100001
After execution,	A = 21H R4 = 33H	

ANL A, 40H ; A ← A & [40H]

Before execution let,	A = 25H,	00100101
	[40H] = 33H,	00110011
	A & [40H] =	00100001
After execution,	A = 21H [40H] = 33H	

ANL A, @R1 ; A← A & @R1

Before execution let,	R1 = 50H, A = 25H,	00100101
	[50H] = 33H,	00110011
	A & [50H] =	00100001

After execution, A = 21H
R1 = 50H
[50H] = 33H

ANL A, #24H ; **A ← A & #24H**

Before execution let, A = 25H, 0010 0101
Immediate data = 0010 0100
A & 24 = 0010 0100

After execution, A = 24H

ANL 60H, #24H ; **[60H]←[60H] & #24H**

Before execution let, [60H] = 25H, 0010 0101
Immediate data = 0010 0100
A & 24 = 0010 0100

After execution, [60H] = 24H

ANL 60H, A ; **[60H]← [60H] & A**

Before execution let, A = 25H, 0010 0101
[60H] = 33H, 0011 0011
A & [60H] = 0010 0001

After execution, A = 25H [60H] = 21H

ORL: Logical OR for byte variables

ORL performs the bitwise logical-OR operation between the variables indicated and stores the results in the destination variable. No flags are affected. The two operands allow six addressing mode combinations.

Syntax:

ORL dst_operand, src_operand

Operation :

dst_operand ← dst_operand V Src_operand

Src_operand ORs with dst_operand and stores the result in dst_operand. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing. When the destination is a direct address, the source can be the Accumulator or immediate data as illustrated below

ORL A, Rn	;	A ← A V Rn, Where Rn = R0 to R7
ORL A, direct	;	A ← A V [direct memory content]
ORL A, @Ri	;	A ← A V [memory content pointed to by Ri], Where Ri = R0 or R1
ORL A, #data	;	A ← A V immediate data
ORL direct, A	;	[direct memory content] ← [direct memory content] V A
ORL direct, #data	;	[direct memory content] ← [direct memory content] V immediate data

Examples :

ORL A, R4 ; **A← A V R4**

Before execution let, A = 25H, 0010 0101
R4 = 33H, 0011 0011
A V R4 = 0011 0111

After execution, A = 37H R4 = 33H

ORL A, 40H ; **A← A V [40H]**

Before execution let, A = 25H, 0010 0101
[40H] = 33H, 0011 0011
A V [40H] = 0011 0111

After execution, A = 37H [40H] = 33H

ORL A, @R1 ; **A← A V @R1**

Before execution let, R1=50H, A=25H, 0010 0101
[50H] = 33H, 0011 0011
A V [50H] = 0011 0111

After execution, A = 37H R1 = 50H
[50H] = 33H

ORL A, #24H ; **A← A V #28H**

Before execution let, A = 25H, 0010 0101
Immediate data = 0010 1000
A V 24H = 0010 1101

After execution, A = 2DH

ORL 60H, #24H ; **[60H]← [60H] V #28H**

Before execution let, [60H] = 25H, 0010 0101
Immediate data = 0010 1000
A V 24 = 0010 1101

After execution, [60H] = 2DH

ORL 60H, A ; **[60H]← [60H] V A**

Before execution let, A = 25H, 0010 0101
[60H] = 33H, 0011 0011
A V [60H] = 0011 0111

After execution, A = 25H [60H] = 37H

XRL : Logical Ex-OR for byte variables

XRL performs the bitwise logical-Ex-OR operation between the variables indicated and stores the results in the destination variable. No flags are affected. The two operands allow six addressing mode combinations.

Syntax :

XRL dst_operand, src_operand

Operation :

dst_operand ← dst_operand ¥ Src_operand

Src_operand Ex-ORs with dst_operand and stores the result in dst_operand. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing. When the destination is a direct address, the source can be the Accumulator or immediate data as illustrated below

XRL A, Rn ; A ←A ¥ Rn, Where Rn = R0 to R7
XRL A, direct

; A ← A ¥ [direct memory content]

XRL A, @Ri ; A ← A ¥ [memory content pointed to by Ri], Where Ri = R0 or R1

XRL A, #data ; A ← A ¥ immediate data

XRL direct, A ; [direct memory content] ← [direct memory content] ¥ A XRL direct, #data

; [direct memory content] ← [direct memory content] ¥ immediate data

Examples:

XRL A, R4 ; **A← A ¥ R4**

Before execution let, A = 25H, 0010 0101
R4 = 33H, 0011 0011
A ¥ R4 = 0001 0110

After execution, A = 16H, R4 = 33H

XRL A, 40H ; **A← A ¥ [40H]**

Before execution let, A = 25H, 0010 0101
[40H] = 33H, 0011 0011
A ¥ [40H] = 0001 0110

After execution, A = 16H.[40H] = 33H

XRL A, @R1 ; **A← A ¥ @R1**

Before execution let, R1 = 50H, A=25H, 0010 0101
[50H] = 33H, 0011 0011
A ¥ [50H] = 0001 0110

After execution, A = 16H, R1 = 50H, [50H] = 33H

XRL A, #24H ; **A←A ¥ #28H**

Before execution let, A = 25H, 0010 0101
Immediate data = 0010 1000
A ¥ 24H = 0000 1101

After execution, A = 0DH

XRL 60H, #28H ; **[60H] ← [60H] ¥ #28H**

Before execution let, [60H] = 25H, 0010 0101
Immediate data = 0010 1000
A ¥ 28 = 0000 1101

After execution, [60H] = 0DH

XRL 60H, A ; **[60H]← [60H] ¥ A**

Before execution let, A = 25H, 0010 0101
[60H] = 33H, 0011 0011
A ¥ [60H] = 0001 0110

After execution, A = 25H [60H] = 16H

CLR A : Clear Accumulator

CLR A clears the Accumulator (all bits set to 0). No flags are affected

Syntax :

CLR A

Operation :

A ← 00H

Accumulator register content is cleared and loaded with 00

Example :

CLR A ; A← 00

Before execution let, A = 25H,

After execution A = 00

CPLA : Complement Accumulator

CPLA logically complements each bit of the Accumulator (one's complement). Bits 1s are changed to a 0s and 0s are changed to a 1s. No flags are affected.

Syntax :

CPL A

Operation :

A ← A'

Accumulator contents are ones complemented and stored back again in accumulator A

Example :

CPL A	; A← A'	
Before execution let,	A = 25H,	0010 0101
After execution	A = DAH,	1101 1010

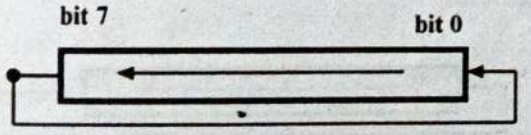
RL A: Rotate Accumulator Left

The eight bits in the Accumulator are rotated one bit towards the left. Bit 7 is moved into the bit 0 position. No flags are affected.

Syntax :

RL A

Operation :

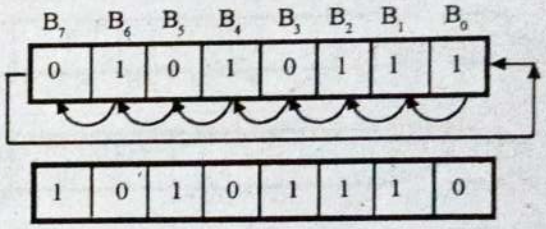


Accumulator contents are rotated towards left by 1 bit position. The MSB (bit 7) is moved into LSB (bit 0)

Example :

RL A

Before execution let, A = 57H, 0101 0111



After execution A = 0AEH, 1010 1110

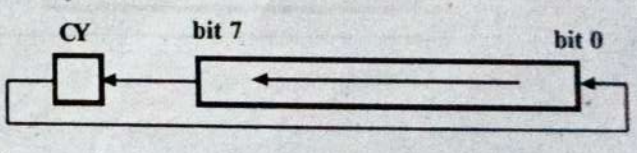
RLC A : Rotate Accumulator Left through the Carry flag

The eight bits in the Accumulator and the carry flag are together rotated one bit towards the left. Bit 7 moves into the carry flag; the original state of the carry flag moves into the bit 0 position. Hence, only carry flag is affected and No other flags are affected

Syntax :

RLC A

Operation :

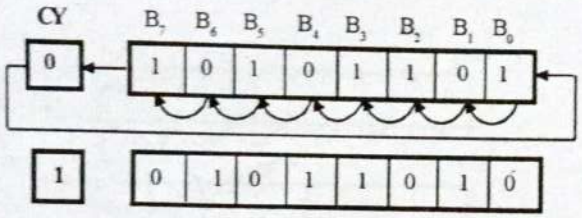


Accumulator contents are rotated through carry towards left by 1 bit position. The MSB (bit 7) is moved into CY flag, CY bit is moved into LSB (bit 0)

Example:

RLC A

Before execution let, CY = 0, A = 0ADH, 1010 1101



After execution CY = 1, A = 5AH, 0101 1010

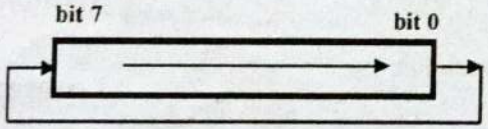
RR A: Rotate Accumulator Right

The eight bits in the Accumulator are rotated one bit towards right. Bit 0 is moved into the bit 7 position. No flags are affected.

Syntax:

RR A

Operation:

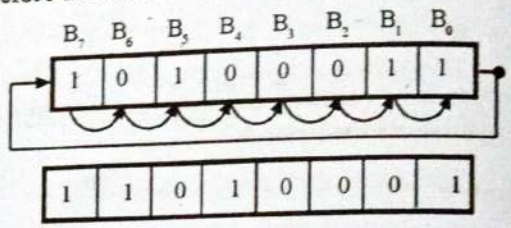


Accumulator contents are rotated towards left by 1 bit position. The LSB (bit 0) is moved into MSB (bit 7)

Example:

RR A

Before execution let, A = 0A3H, 1010 0011



After execution A = 0D1H, 1101 0001

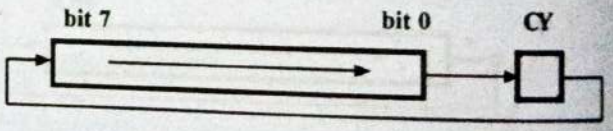
RRC A: Rotate Accumulator Right through the Carry flag

The eight bits in the Accumulator and the carry flag are together rotated one bit towards the right. Bit 0 moves into the carry flag; the original state of the carry flag moves into the bit 7 position. Hence, only carry flag is affected and No other flags are affected

Syntax:

RRC A

Operation:

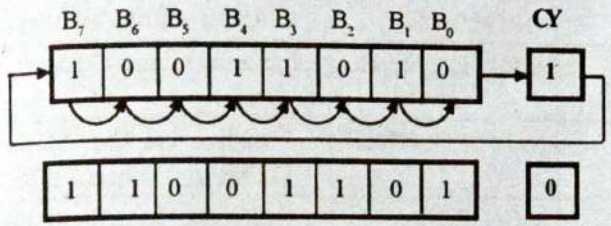


Accumulator contents are rotated through carry towards right by 1 bit position. The LSB (bit 0) is moved into CY flag, CY bit is moved into MSB (bit 7)

Example:

RRC A

Before execution let, CY = 1, A = 9AH, 1001 1010



After execution CY = 0, A = 0CDH, 1100 1101

SWAP A: Swap nibbles within the Accumulator

SWAP A interchanges the low- and high-order nibbles (four-bit fields) of the Accumulator (bits 3 through 0 exchanges with bits 7 through 4). The operation can also be thought of as a 4-bit rotate instruction. No flags are affected

Syntax:

SWAP A

Operation :

A(3-0) ← A(7-4)

Accumulator lower nibble is exchanged with higher nibble

Example:

SWAP A

Before execution let, A = 97H,

After execution A = 79H

Lab Exercises:

1. Write and execute an ALP to

(a) Move a block of data within internal RAM

Label	Mnemonic	Description
	ORG 0000H	; Initialize starting address of Program.
	MOV R0, #50H	; Initialize Internal Memory 1 (Source) Pointer R0 to 50 h.
	MOV R1, #60H	; Initialize Internal Memory 2 (Destin.) Pointer R0 to 60 h.
	MOV A, @ R0	; Copy the data from Internal Memory (Source)to A. (Length of the Block)
	MOV R3, A	; Copy the Contents of A into R3 as a Counter.
	INC R0	; Increment the Internal Memory Pointer 1.
BACK:	MOV A, @ R0	; Copy the data from Internal Memory 1 to A. (Get the First Data byte from Source Internal Memory)
	MOV @R1, A	; Transfer the first data from One Internal memory1 to another Internal Memory 2.
	INC R0	; Increment Internal Memory Pointer 1.
	INC R1	; Increment another Internal Memory Pointer 2.
	DJNZ R3, BACK	; Decrement the counter & repeat Data Transfer,if it is not zero.
LOOP:	SJMP LOOP	; Stop
	END	; END

Example :

Input :

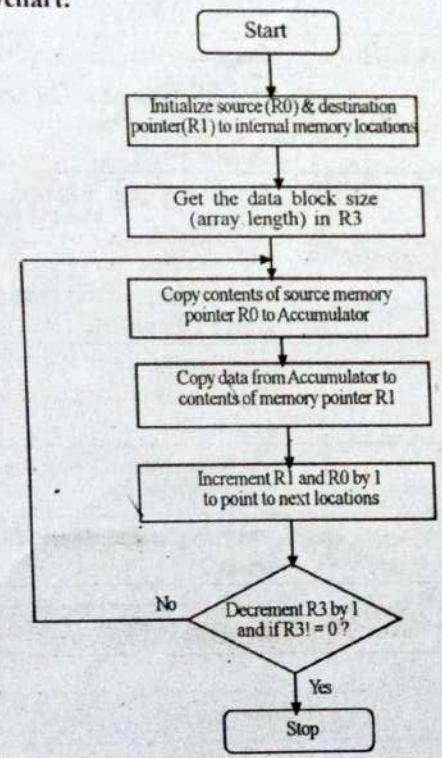
Internal Memory	50h	51h	52h	53h	54h	55h
Data	05	22	33	55	77	89

Length of the Block = 05

Output :

Internal Memory	60h	61h	62h	63h	64h
Data	22	33	55	77	89

Flowchart:



(b) Exchange a block of data between internal RAM and external memory.

Label	Mnemonic	Description
	ORG 0000H	; Initialize starting address of Program
	MOV R1, #60H	; Initialize Internal Memory Pointer R0 to 50 h.
	MOV DPTR,#9000H	; Initialize the Ext. Memory Pointer DPTR to location 200H.
	MOV R2, #05H	; Store the Length of the Block in Register R3 (Counter).
BACK	:MOVXA,@DPTR	Copy memory content pointed by DPTR register to acc
	MOV @R1, A	Store the 8 bit number in external RAM whose address is held by R1
	INC R1	Increment R1 by 1 to point to next location of internal RAM
	INC DPTR	Increment DPTR by 1 to point to next location of external RAM
	DJNZ R2, BACK	Decrement R2 and jump to BACK if R2 is not equal to zero (R2?R2-1 and jump to BACK if R2!=0)
LOOP	SJMP LOOP	; Stop
	END	; END

Example:

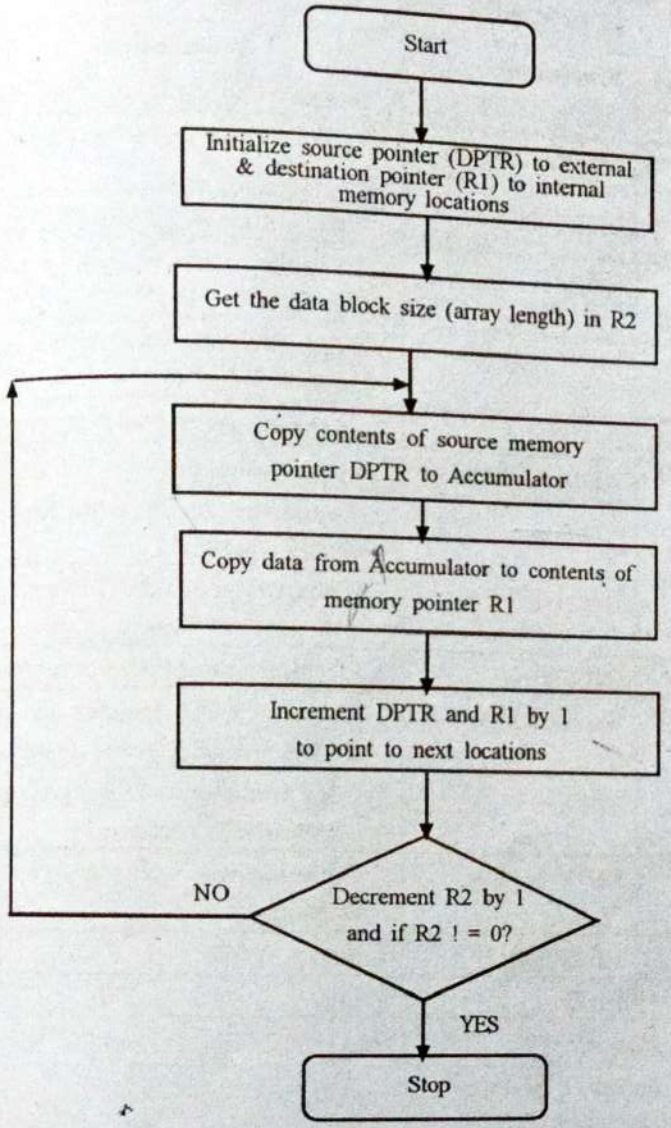
Input:

External memory	9000H	9001H	9002H	9003h	9004H
Data	01	02	03	04	05

Output:

Internal memory	60H	61H	62H	63h	64H
Data	01	02	03	04	05

Flowchart:



2. Write an ALP to

(a) evaluate simple arithmetic expression such as y= (((5*2)-(4+1))/3) %2.

Label	Mnemonic	Description
	ORG 0000H	; Initialize starting address of Program.
	MOV A, #4	; Copy immediate data 4 in to accumulator
	MOV R1,#1	; Copy immediate data 1 in to R1 register
	ADD A, R1	; Perform 4+1
	MOV R1, A	; Store 4+1 In R1
	MOV B, #05H	; Copy immediate data 05h in to B register
	MOV A, #02H	; Copy immediate data 02H in to accumulator
	MUL AB	; Perfom 5*2
	SUBB A,R1	; Perform (5*2)-(4+1)
	MOV B,#03H	; Copy immediate data 03H in to B register
	DIV AB	; Perform (((5*2)-(4+1))/3)
	MOV B,#02H	; Copy immediate data 02H in to B register
	DIV AB	; Perform (((5*2)-(4+1))/3) %2
	MOV 30H, B	; Store Result In 30h Location
LOOP	SJMP LOOP	; Stop
	END	; End

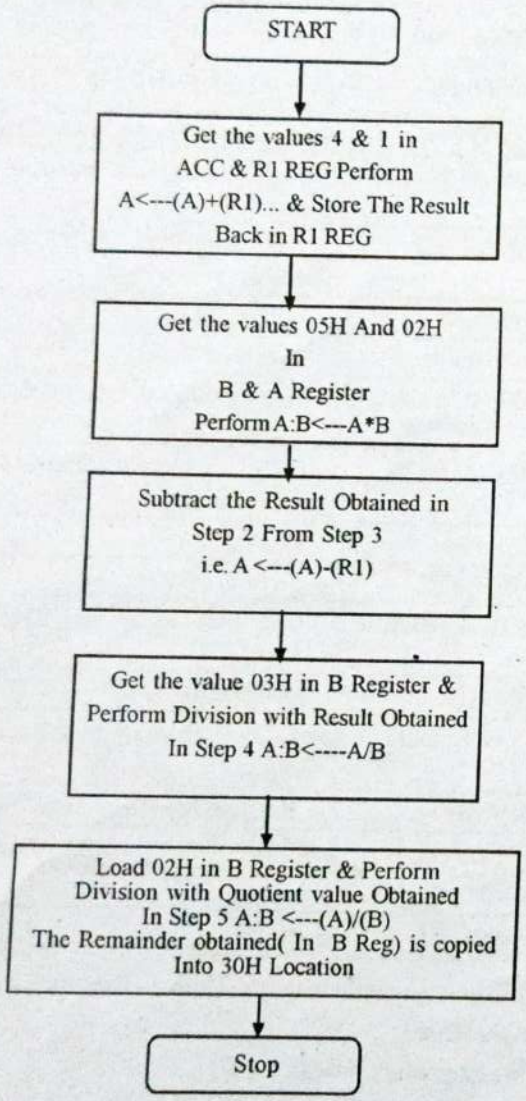
Result: See the result in internal memory location d:0x30

Sample Run:

Before execution- d:0x30 = 00H

After Execution: d:0x30 = 01H

Flowchart:



(b) Perform addition of three 8-bit BCD numbers to result in BCD form.

Label	Mnemonic	Description
	ORG 0000H	; Initialize starting address of Program
	MOV R0,#30H	; initialize internal memory pointer R0 to 30h
	MOV R2,#02H	; Load the (count-1) value in R2 register
	MOV R3,#00H	; Initialize register R3 to 00h to count the carry
	MOV A,@R0	; Copy the first BCD number from internal RAM to acc
BACK:	INC R0	; Increment the internal memory pointer
	ADD A,@R0	; performs the operation A? (A)+((R0))
	DA A	; Adjust the result in acc to a valid decimal number
	JNC NEXT	; If no carry go to NEXT else continue with next instruction
	INC R3	; Increment the content of register R3
NEXT:	:DJNZ R2,BACK	; Decrement R2 register and go to BACK if it isn't zero else continue
	MOV 40H,R3	; Copy the content of R3 register (carry count) at 40h location
	MOV 41H,A	; Copy the sum value in Acc at 41h location
LOOP	SJMP LOOP	; Stop
	END	; End

Result:

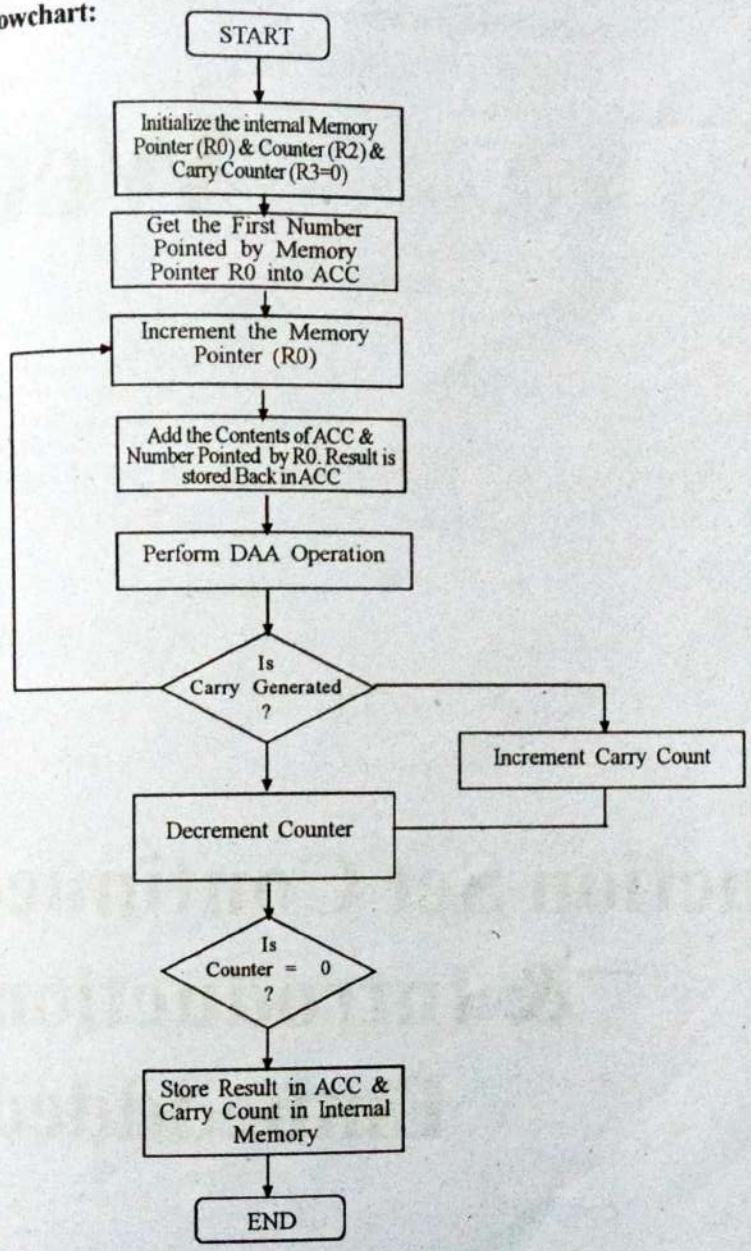
Input

Memory	30h	31h	32h
Data	02	03	04

Output:

Memory	40h	41h
Data	00	09

Flowchart:



Review Questions :

1. Classify the instruction set of 8051. Give one example for each.
2. List the different types of addressing modes and explain with example.
3. Explain the PUSH and POP instructions.
4. Explain the following instructions with suitable examples.

   i. MOVX ii. XCHD

5. Explain DAA instruction with suitable examples.
6. List the arithmetic instructions of 8051 microcontroller.
7. Write an ALP to move a block of data from external memory to internal memory.
8. Point out the mistake in the following instructions (i) PUSH R1 (ii) MOV R1,#398 (iii) RET label(iv) MOV R1,R2 (v) DIV A,B.
9. What are addressing modes? List them.
10. Indicate the uses of the following instructions.

    NOP,XRL,ORL,ANL,MOVX

Week 9

Instruction Set Continued... & Introduction to Embeddded C

Instruction Set Continued & Introduction to Embedded C

9.1 Bit Level Logical Instruction

9.1.1 Program Branching and Machine Control Instructions

9.1.2 Unconditional Branching

9.1.3 Conditional Branching

9.1.4 CALL

9.2 Introduction to Embedded C

9.2.1 Pros & Cons of Embedded C

9.3 General Structure of Embedded C Program

9.3.1 Data Types in 8051 C Programming

* Lab Programs

* Review Questions

9.1 Bit Level Logical Instructions

The Boolean variable instructions of 8051 are related to the bit operations such as CLR, SETB, CPL, ANL, ORL and MOV are discussed below in detail with examples.

CLR bit: Clear bit

CLR bit clears the indicated bit (reset to 0). No other flags are affected. CLR can operate on the carry flag or any directly addressable bit.

128 Bits

Syntax:

CLR bit

Operation:

Bit ← 0

Bit is loaded with logic 0.

Example:

CLR C ; C← 0

CLR ACC.4 ; D4 bit of register A is cleared, ACC.4 ←0

Before execution let,	A = 0FFH,	1111 1111B
After execution,	A = 0EFH,	1110 1111B

CLR P1.6 ; D6 bit of port 1 register is cleared, P1.6 ←0

Before execution let,	P1 = 58H,	0101 1000B
After execution,	P1 = 18H,	0001 1000B

SETB bit: set bit

SETB sets the indicated bit to one. SETB can operate on the carry flag or any directly addressable bit. No other flags are affected.

Syntax:

SETB bit

Operation:

Bit ← 1

Bit is loaded with logic 1.

Example:

SETB C ; C← 1

SETB ACC.4 ; D4 bit of register A is set, ACC.4← 1

Before execution let,	A = 00H,	0000 0000B
After execution,	A = 10H,	0001 0000B

SETB P1.6 ; D6 bit of port 1 register is set, P1.6 ←1

Before execution let,	P1 = 28H,	0010 1000B
After execution,	A = 68H,	0110 1000B

CPL bit: Complement bit

CPL bit complements the bit variable specified. A bit that had been a 1 is changed to 0 and 0 is changed to 1. No other flags are affected. CPL can operate on the carry or any directly addressable bit.

Syntax:

CPL bit

Operation:

Bit ← (not) bit

Bit is complemented and reloaded back into bit.

Example:

CPL C ; C ← not C

Before execution let,	C = 0	
After execution,	C = 1	

CPL ACC.4 ; D4 bit of register A is complemented,

ACC.4 ← not ACC.4

Before execution let,	A = 00H,	0000 0000B
After execution,	A = 10H,	0001 0000B

CPL P1.6 ;D6 bit of port 1 register is complemented,

P1.6 ← not P1.6

Before execution let,	P1 = 68H,	0110 1000B
After execution,	A = 28H,	0010 1000B

ANL C, src_bit: Logical ANDing of C bit with Source bit

ANL performs the logical-AND operation between the Carry flag bit with source bit and stores the results in the Carry flag. No other flags are affected. It operates only on the carry and any directly addressable bit.

Syntax:

ANL C, src_bit

Operation:

C ←C & src_bit

C bit and src bit are ANDed and stores the result in C flag bit.

Example:

ANL C, ACC.4 ; C ← C & ACC.4

Before execution let, ACC.4 = 0, C = 1
After execution, ACC.4 = 0, C = 0

ANL C, P1.2 ; C ← C & P1.2

Before execution let, P1.2 = 1, C = 1
After execution, P1.2 = 1, C = 1

ANL C, /src_bit: Logical ANDing of C bit with complemented Source bit

ANL performs the logical-AND operation between the Carry flag bit with inverted source bit and stores the results in the Carry flag. No other flags are affected. It operates only on the carry and any directly addressable bit.

Syntax:

ANL C, /src_bit

Operation:

C ← C & (NOT src_bit)

C bit and inverted src_bit are ANDed and stores the result in C flag bit.

Example:

ANL C, /ACC.4 ; C ← C & (NOT ACC.4)

Before execution let, ACC.4 = 0, C = 1

After execution, ACC.4 = 0, C = 1
(NOT ACC.4 =1)

ANL C, P2.2 ; C ← C & P2.2

Before execution let, P2.2 = 1, C = 1
(NOT P2.2 = 0)

After execution, P2.2 = 1, C = 0

ORL C, src_bit: Logical ORing of C bit with Source bit

ORL performs the logical-OR operation between the Carry flag bit with source bit and stores the results in the Carry flag. No other flags are affected. It operates only on the carry and any directly addressable bit.

Syntax:

ORL C, src_bit

Operation:

C ← C | src_bit

C bit and src bit are ORed and stores the result in C flag bit.

Example:

ORL C, ACC.4 ; C ← C | ACC.4

Before execution let, ACC.4 = 1, C = 0
After execution, ACC.4 = 1, C = 1

ORL C, P3.2 ; C ← C | P3.2

Before execution let, P3.2 = 1, C = 1
After execution, P3.2 = 1, C = 1

ORL C, /src_bit: Logical ORing of C bit with complemented Source bit

ORL performs the logical-OR operation between the Carry flag bit with inverted source bit and stores the results in the Carry flag. No other flags are affected. It operates only on the carry and any directly addressable bit.

Syntax:

ORL C, /src_bit

Operation:

C ← C | (NOT src_bit)

C bit and inverted src_bit are ORed and stores the result in C flag bit.

Example:

ORL C, /ACC.4 ; C ← C | (NOT ACC.4)

Before execution let, ACC.4 = 0, C = 0

After execution, ACC.4 = 0, C = 1 (NOT ACC.4 =1)

ORL C, P0.2 ; C ← C & P0.2

Before execution let, P0.2 = 1, C = 0 (NOT P0.2 = 0)

After execution, P0.2 = 1, C = 0

MOV dest_bit, src_bit: Move bit data

MOV dest_bit, src_bit copies the Boolean variable indicated by the second operand into the location specified by the first

operand. One of the operands must be the carry flag; the other may be any directly addressable bit. No other register or flag is affected.

Syntax:

MOV dest_bit, src_bit

Operation:

dest_bit ← src_bit

source bit is copied into destination bit. Source and destination bit is either carry flag or directly addressable bit as illustrated below

MOV C,bit ; C ← bit

MOV bit,C ; bit ← C

Example:

MOV C, ACC.3 ; C ← ACC.3

Before execution let, ACC.3 = 0, C = 1

After execution, ACC.3 = 0, C = 0

MOV P1.2, C ; P1.2 ← C

Before execution let, P1.2 = 1, C = 0

After execution, P1.2 = 0, C = 0

9.1.1 Program Branching and Machine Control Instructions

The Program branching and machine control instructions of 8051 are related to the unconditional and conditional jumping operations such as CALL, RET, JMP, JZ, JNZ, JC, JNC, JB, JNB, CJNE, DJNZ and NOP are discussed below in detail with examples:

- JUMP and CALL instructions change the flow of the program by changing the contents of program counter.
- A JUMP permanently changes the program flow whereas CALL temporarily changes the program flow to allow another part of the program to run.

JUMP

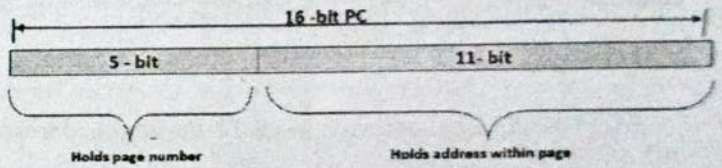
JUMP instruction transfers control to a subroutine and never returns back. There are two types of JUMP instructions :

- Conditional JUMP
- Unconditional JUMP

- The JUMP instruction which changes the program flow irrespective of the condition is called unconditional JUMP.
- The JUMP instruction which changes the program flow if certain condition exits is called conditional JUMP.
- Depending upon the condition i.e., true or false, program branches (jumps) to the specified address.
  - i.e., if condition is true: then jumps to specified label
  - if condition is false: No jump, executes from very next instruction.
- All conditional jumps are short jumps, meaning that the target address cannot be more than -128 bytes backward to +127 bytes forward of the PC of the instruction following the JUMP

Type of Jump or CALL	Ranges	No of bytes	Example
Relative/short range	-128b to +127b	2-byte instruction	JC, JNC, JB, JNB, JBC, JZ, JNZ, DJNZ, CJNE, SJMP
Absolute range	Within a page (2K bytes)	2-byte instruction	ACALL, AJMP
Long range	Anywhere in ROM(0000-FFFFH)	3-byte instruction	LCALL, LJMP

In absolute range, it JUMP/CALL can be within a single page. The upper 5-bits of PC holds the page number and lower 11-bits holds the address within that page as illustrated below :



9.1.2 Unconditional Branching

SJMP address (8 bits): Short Jump

It is a 2-byte short jump instruction uses relative address. The jump distance is limited to a range of -128 to +127 bytes relative to the instruction following jump.

With this instruction one can jump 127 bytes location forward and 128 bytes locations backward from the current position. The destination address is a signed 8-bit value (00 to FFH) called offset

Syntax

SJMP label;

PC [7:0]← label address; loads 8-bit target address in to PC

AJMP address (11 bits): Absolute Jump

It is absolute range jump instruction & is of 2 bytes. The destination address consists of 11 bits. It transfers the program execution to target address unconditionally.

Using AJMP instruction it is possible to jump with in 2KB range

(**Note:** The whole 64KB ROM is divided in to 32 pages(blocks) of 2KB each)

Syntax:

AJMP label

PC [10:0]← label address; loads 11-bit target address in to PC

LJMP address (16 bits): Long jump

It is a long-range jump instruction & is of 3-bytes. The destination address consists of 16- bits. It transfers the program execution to the target address unconditionally. The destination can be anywhere within 64KB program (ROM) memory space. The jump address ranges from 0000H to FFFFH.

Syntax:

LJMP label

PC [15:0]← label address; loads 16- bit target address in to PC

9.1.3 Conditional Branching :

JC rel/label: Jump to label/relative address, if carry is set

This instruction will be executed & branch or jump to relative address only if the carry is set (CY=1). If the carry is zero (Not set). The assembler simply ignores this instruction & execute the next instruction.

Syntax:

JC rel/label

Operation:

If C=1, then; check whether CY=1 or not

PC [7:0]? label address; if CY=1, jump to label

Else

Next; If CY=0, continue execution with next instruction

JNC rel/label: Jump to label/relative address, if carry is not set

This instruction will be executed & branch or jump to relative address only if carry is not set (CY=0). If the carry is 1(set), the assembler simply ignores this instruction & execute the next instruction

Syntax:

JNC rel/label

Operation:

If C=0, then; check whether CY=0 or not

PC [7:0]←label address; if CY=0, jump to label

Else

Next; If CY=1, continue execution with next instruction

JZ rel: Jump to label/relative address, if accumulator is zero

This instruction will be executed & branch or jump to relative address only if the accumulator content is zero. If the accumulator content is not zero, then the assembler simply ignores this instruction & execute the next instruction.

Syntax:

JZ rel/label

Operation:

If A=0, then; check whether A=0 or not

PC [7:0]← label address; if A=0, jump to label

Else

Next; If A≠0, continue execution with next instruction

JNZ rel: Jump to label/relative address, if accumulator is not zero

This instruction will be executed & branch or jump to relative address only if the accumulator content is not zero. If the accumulator content is zero, then the assembler simply ignores this instruction & execute the next instruction.

Syntax:

JNZ rel/label

Operation:

If A ≠ 0, then; check whether A ≠ 0 or not

PC [7:0]← label address; if A≠0, jump to label

Else

Next; If A=0, continue execution with next instruction

JB bit, rel/label: Jump to label/relative address, if bit is equal to1

This instruction will branch/jump to the relative/label if the indicated bit in the instruction is set. Otherwise, next instruction will be executed.

Syntax:

JB bit, rel/label

Operation:

If bit=1, then; check whether bit=1 or not

PC← Label address; if bit=1, jump to the label

Else

Next; If bit=0, continue execution with next instruction

JNB bit, rel/label: Jump to label/relative address, if bit is not equal to1

This instruction will branch/jump to the relative/label if the indicated bit in the instruction is not set. Otherwise, next instruction will be executed.

Syntax:

JNB bit, rel/label

Operation:

If bit ≠1, then; check whether bit≠1 or not

PC← Label address; if bit≠1, jump to the label

Else

Next; If bit=1, continue execution with next instruction

JBC bit, rel/label: Jump to relative address/label, if bit is equal to 1 & clear it

If the indicated bit in the instruction is set(1) then it will jump to the relative address or label & clear it, otherwise next instruction will be executed.

Syntax:

JBC bit, rel/label

Operation:

If bit=1, then; check whether bit=1 or not

bit←0 ;clear the bit

PC← Label address; if bit=1, jump to the label

Else

Next; If bit=0, continue execution with next instruction

CJNE: Compare and Jump if Not Equal

This instruction compares the magnitudes of source & destination. If they are not equal then it jumps to the relative address/label. It affects the carry flag only.

The carry flag is set if the unsigned integer value of <dest-byte> is less than the unsigned integer value of <src-byte>; otherwise, the carry is cleared.

Syntax:

CJNE <dest-byte>,<src-byte>, rel/label

If <dest-byte> ≠ <src-byte> then

PC← label address; Jump to label/rel address

CY (carry flag) is affected; either it will be set or reset

Else

Next; Continue execution with next instruction

The first two operands allow four addressing mode combinations: The Accumulator may be compared with any directly addressed byte or immediate data, and any indirect RAM location or working register can be compared with an immediate constant as illustrated below.

```
CJNE A, direct, rel   ; jump if A!=[direct], if A>[direct], C = 0 else C = 1
CJNE A, #data, rel    ; jump if A!=data, if A>data, C = 0 else C = 1
CJNE Rn, #data, rel   ; jump if Rn!=data, if Rn>data, C = 0 else C = 1
CJNE @Ri, data, rel   ; jump if [Ri]!=data, if [Ri]>data, C = 0 else C = 1
```

DJNZ: Decrement and Jump if Not Zero

The instruction decrements the byte content by 1 & branch/jumps to the relative address or label if it is not zero. If it is zero next instruction will be executed.

Syntax:

DJNZ <byte>,rel/label

Operation:

Byte← byte-1; Decrement byte by 1

If byte ≠ 0; Compare the byte with zero

Then

PC← label address; Jump to the label

Else

Next; If equal to zero, continue execution with next instruction

The byte operand allows two addressing mode combinations: directly addressed byte or working register as illustrated below.

DJNZ Rn,rel ; Rn← Rn-1, if Rn!=0 then Jump to label else next instruction

DJNZ direct,rel ; [direct]← [direct]-1, if [direct]!=0 then Jump to label else next instruction

9.1.4 CALL

CALL instruction transfers control to a subroutine and returns back with RET instruction. There are two types of CALL:

- ACALL - Absolute CALL
- LCALL - Long CALL

ACALL is a 2-byte instruction In ACALL, the target address is within 2K bytes of the current program counter (PC). LCALL is a 3-byte instruction in which one byte is opcode and the other two are the 16-bit address of the target subroutine. The operational steps of CALL are as follows :

- If a subroutine is called, the PC is pushed onto the stack and the stack pointer is incremented by 2
- Then the program counter is loaded with the new address and control is transferred to the subroutine
- At the end of the procedure, when RET instruction is executed, PC is popped back the stack, which returns control to the instruction after the CALL-in main program.

ACALL add/label: Absolute Call

This is a 2-byte instruction uses 11-bit target address. Since

address bits are of 11 bits, CALL can address within 2K bytes. Whenever a subroutine is called from main program, Program counter (PC) holds the address of next instruction to be executed is saved on to the stack and the program execution branches to subroutine.

RET instruction at the end of subroutine is used to get back to main program. When RET instruction is encountered, the program counter retrieves address stored on stack & main program execution continues.

Syntax: ACALL label/add

Operation:

SP←SP+1 ; SP increments by 1
[SP]← PC[7:0]; lower byte address is moved to stack
SP←SP+1; SP increments by 1
[SP]←PC[15:8]; Higher byte address is moved to stack
PC[10:0]←Subroutine address; Load 11 bit subroutine address in to PC

LCALL add/label: Long CALL

This is a 3-byte instruction uses 16-bit target address. Since address bits are of 16-bits, LCALL can address anywhere within 64KB. Whenever a subroutine is called from main program, program counter (PC) holds the address of the next instruction to be executed, is saved on to the stack. And the program execution branches to subroutine.

RET instruction at the end of subroutine is used to go back to main program. When RET instruction is encountered, the program counter retrieves address stored on stack & main program execution continues.

Syntax: LCALL label/add

Operation:

SP ← SP+1 ; SP increments by 1
[SP] ← PC[7:0]; lower byte address is moved to stack
SP ← SP+1; SP increments by 1
[SP] ← PC[15:8]; Higher byte address is moved to stack
PC ← Subroutine address; Load 16-bit subroutine address in to PC

RET: Return from subroutine

RET pops the high & low order bytes of PC successively from the stack, decrementing the stack pointer by two. Program execution continues at the resulting address

Generally, the instruction immediately following ACALL or LCALL. No flags are affected.

Syntax: RET

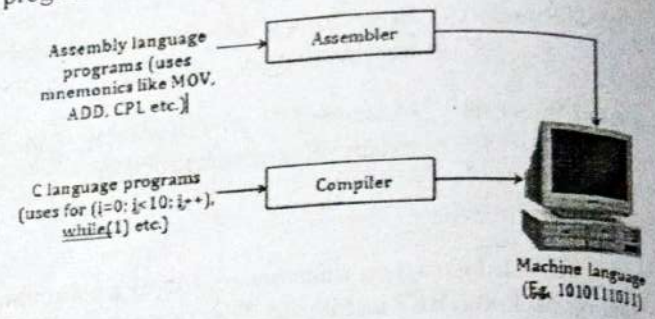
Operation:

PC[15:8]←[SP] ; Higher byte address is recovered from stack
SP←SP-1; SP decrements by 1
PC[7:0]←[SP]; Lower byte address is recovered from stack
SP←SP-1; SP decrements by 1

9.2 Introduction to Embedded C

Assembly language programming of 8051 has been discussed in the preceding chapters. We know that assembler (a kind of software) is required to convert programs written in assembly language into machine language. Similarly, 8051 can be programmed

using C language. Compiler (software) is required to convert the programs written in C into machine language



Programming 8051 using C language is preferred when the complexity of program increases. Programming 8051 with C is popular because of ease of programming compare to assembly language. It is easy to write programs in C and takes less time. The disadvantage of using C language for 8051 is that the machine code generated from C programs is approximately 75% greater than that of assembly programs. There are various assemblers available in the market. Eg. : Pinnacle, A51, SBC51 etc. Similarly, C compilers available are Keil's μ vision.

9.2.1 Pros & Cons of Embedded C

Pros (Advantages)

1. It is easy to write modify and up-date. Also takes less time to write.
2. Code available in function libraries such as squrt, sine can be used.
3. 'C' code is portable to other micro-controllers with little OR no modifications.
4. C programs are more readable.

Cons (Disadvantages)

1. Microcontrollers have limited on-chip ROM and the code space. The size of the hex file produced by the compiler is large. C language program produces a hex file that is much larger than assembly language.

2. The general-purpose registers of 8051 such as A, B & R0 to R7 are under the control of 8051and not accessed by C statements.

3. Misuse of data types increases memory size of hex file.

9.3 General Structure of Embedded C Program

The structure of Embedded C program is as below:

```
Pre-processor directives
Constant's definitions
Global variables declarations
void main ()
{
Local variables declarations;
Statements/expressions;                    /*Comments*/
.............
.............
}
```

Preprocessor directives:

These are statements that begin with '#' symbol and are used to include header files in the program as well as to define symbolic constants in the program.

- **#include**→ this directive is used to include header files in the program where all functions are defined

Example: #include<reg51.h>

The library file reg51.h contains the definitions of all the special function registers and their bits.

- **#define**→ this directive is used to define symbolic constants in the program.

Example:#define PI 3.142

Global variable declarations:

A global variable is a variable that is declared before the main function and can be accessed by any function in the program.

main() function:

The execution of a program starts with the main function. Every program use only one main() function.

Local variable declarations:

A local variable is a variable declared within a function and it is valid only to be used within that particular function.

Statements/Expressions:

A valid executable statement represents the body of the main function.

Comments:

Comments helps the reader to understand the code easily. It starts with /* and ends with */ for a multi-line comment and // for a single line comment.

9.3.1 Data Types in 8051 'C' Programming

Data types indicates the type of value that an entity(variable/ constants) can hold. Each variable must be declared with their data types before their use in the program.

The most widely used data types in 8051 'C' are listed below;

1. Unsigned char

- It is an 8-bit data type that takes the value in the range from 0 - 255 (00 to FFH). It can be used to store string of ASCII characters
- The character data type is the most natural choice as 8051 is an 8-bit microcontroller & unsigned char is also 8-bit data type
- **Example:** unsigned char x;

  Here 'x' is a variable consisting positive 8-bit number

2. Signed char

- It is also an 8-bit data type that takes the value ranges from -128 to +127.
- The MSB of the data byte is used to represent sign bit i.e., if MSB = 1 the number is negative and if MSB = 0 the number is considered as positive. In a data byte consists of 8 bits, one bit is used for sign representation and the remaining 7 bits are used to represent the magnitude of the signed number.
- It is important to note that by default the char data type is signed i.e., if we do not specify unsigned the compiler assumes signed value.

- **Example:** Singed char x;

  Here 'x' is variable consisting positive/negative 8-bit number

3. Unsigned int

- To store 16-bit data unsigned int is used. Unsigned int is a 16-bit data type that takes the value in the range of 0 to 65535 (0000 to FFFF H). By default, compiler will consider signed int, if the keyword unsigned is not used before int.
- Since 8051 is 8-bit microcontroller 2 bytes of RAM is used for 16-bit operation. This increases memory. Therefore, unless it is required to store 16-bit data, unsigned int must not be used; instead, unsigned char is used.
- The misuse of data type (unsigned int instead of unsigned char) unnecessarily increases the memory size of hex file
- **Example:** unsigned int x;

  Here 'x' is variable consisting positive 16-bit number

4. Signed int

- It is a 16-bit data type that takes the range of values from -32, 768 to +32767.
- The MSB of the 16-bit data is reserved for sign convention and the remaining 15 bits represents magnitude of the signed number. It has already been mentioned that, if MSB = 1 represents negative number and if MSB = 0 represents positive number.
- **Example**: signed int x;

Here 'x' is variable consisting positive/negative 16-bit number

5. **Bit**

➢ The bit data type is used to access single bit of bit addressable memory. Spacefrom 20H to 2FH (128 bits). Bit variables can store bit values of either 0 or 1

➢ **Syntax**: bit name=0Xaddress

Where, bit is a keyword, name- is a variable,

0Xaddress- 8-bit address between 0X20 to 0X2F

➢ **Example:** bit num=0

The variable "num" is of the type bit & is initialised to bit zero

6. **Sbit**

➢ Sbit is similar to bit data type except it is designed to access single bits of SFRregister. For example, all 8-bit ports of 8051(P0,P1,P2 & P3) present in SFR are bit addressable. Theindividual bits of ports can be accessed by using sbit data type

➢ **Syntax:** sbit bit_name=sfr^n;

Where, sbit→is a keyword, bit_name→ is user defined variable name Sfr^n→ is the location of the particular SFR register

➢ **Example:** sbit num=P0^4

Here 5th bit of PORT 0 can be accessed using identifier "num".

7. **Sfr**

➢ The "sfr" keyword (data type) is widely used 8051 C data type, which is designed specifically to access 8-bit (byte) of Special Function Registers

➢ **Syntax:**sfr sfr_name=0Xaddress;

Where, sfr→ is a keyword

Sfr_name→ is user defined valid identifier

0Xaddress→ is the 8 bit address of sfr register

➢ **Example:** Sfr num=0X90;

Here 0X90 is the address of port 1. Hence port1 can be accessed by using "num"

Some of the commonly used data type in 8051C are listed below:

Data type	Size	Range of values
1. Unsigned char	8 bit	0 to 255
2. Signed char	8 bit	-128 to +127
3. Unsigned int	16 bit	0 to 65535
4. Signed int	16 bit	-32,768 to +32,767
5. Sbit	1 bit	SFR bit addressable only
6. bit	1 bit	RAM bit addressable only
7. Sfr	8 bit	RAM address 80 to FF (SFR) only

Lab Programs:

1. Write an ALP to

(a) Rotate or shift 16-bit data.

Label	Mnemonic	Description
	ORG 0000H	; Initialize starting address of Program.
	CLR C	; clear carry bit
	MOV DPTR,#1234H	; copy the immediate 16 bit data into DPTR reg
	MOV A,DPL	; copy contents of DPL (lower 8 bits of DPTR) into acc
	RLC A	; Rotate contents of acc left once through the carry bit
	MOV DPL,A	; Copy the contents of acc back into DPL reg after rotation
	MOV A,DPH	; copy contents of DPH (upper 8 bits of DPTR) into acc
	RLC A	; Rotate contents of acc left once through the carry bit
	MOV DPH,A	; Copy the contents of acc back into DPH reg after rotation
LOOP:	SJMP LOOP	; Stop
	END	; End

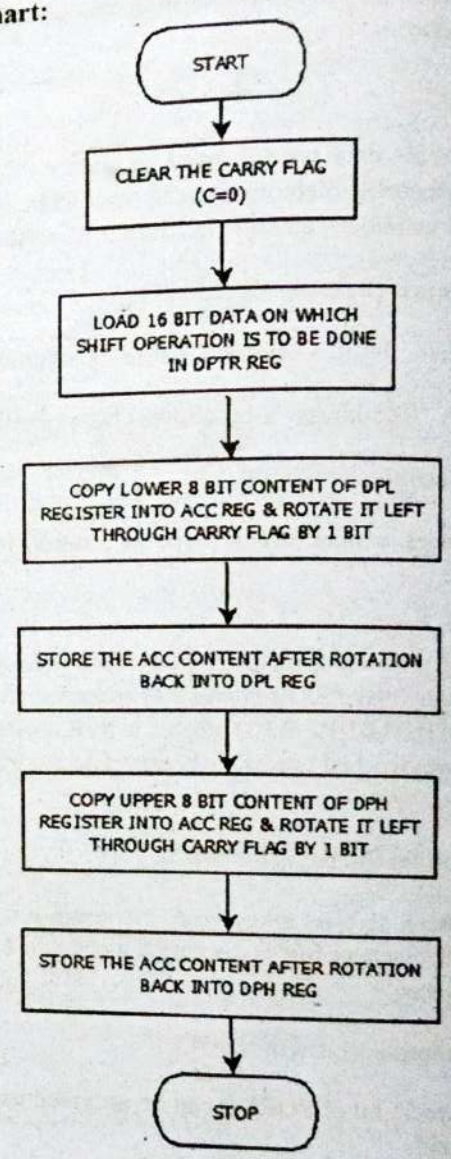
Result: Input:

DPTR: 1234H

Output:

DPTR: 2468H

Flowchart:



(b) Evaluate simple logical expression such as Y=a&&b||c^!d where a, b, c and d are 8-bit data.

Label	Mnemonic	Description
	ORG 0000H	; Initialize starting address of Program.
	MOV R0,#30H	; Initialize internal memory pointer R0 to 30h
	MOV A,@R0	; copy the data 'a' from internal RAM to acc
	INC R0	; Increment the internal memory pointer
	ANL A,@R0	; perform the operation A←(A) && (R0) ie. (A && B)
	MOV R2,A	; Copy the result of AND operation into register R2
	INC R0	; Increment the internal memory pointer
	MOV A,@R0	; Copy the data 'd' form internal RAM to acc
	CPL A	; Complement Acc to perform !d
	INC R0	; Increment the internal memory pointer
	XRL A,@R0	; perform the operation A←(A) ^ ((R0)) ie. (C XOR d!)
	ORL A,R2	; perform the operation A←(A) || (R2) ie(a&&b||c^!d)
	INC R0	; increment the internal memory pointer
	MOV @R0,A	; Store the final result in the internal memory pointed by R0
LOOP:	SJMP LOOP	; Stop
	END	; End

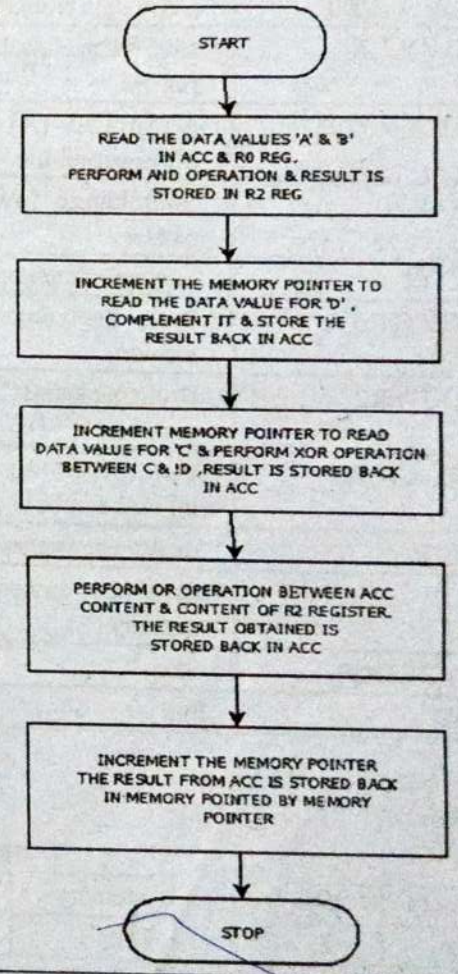
Result: Input

Memory	30h(A)	31h(B)	32h(D)	33h(C)
Data	12	02	03	25

Output

Memory	34h(A)
Data	DB

Flowchart



2. Write and execute an assembly and embedded C program to convert (a)Packed BCD to unpacked BCD

Label	Mnemonic	Description
	ORG 0000H	; Initialize starting address of Program.
	MOV R0,#50H	; Initialize memory pointer R0 to 50h
	MOV A,@R0	; Copy data from memory into Acc
	MOV R2,A	; save the accumulator content into reg R2
	ANL A,#0F0H	; perform A←(A) & #F0h.. mask off lower nibble
	SWAP A	; Interchange lower and upper nibbles
	INC R0	; increment memory pointer
	MOV @R0,A	; Store the content of Acc into memory
	MOV A,R2	; Bring packed BCD number again into Acc
	ANL A,#0FH	; Perform A←(A) & #0Fh.. mask off higher nibble
	INC R0	; increment memory pointer
	MOV @R0,A	; store the content of Acc into memory
LOOP	SJMP LOOP	; Stop
	END	; End

RESULT:

Input :

Memory	50h
Data	36

Output:

Memory	51h	52h
Data	03	06

C Program:

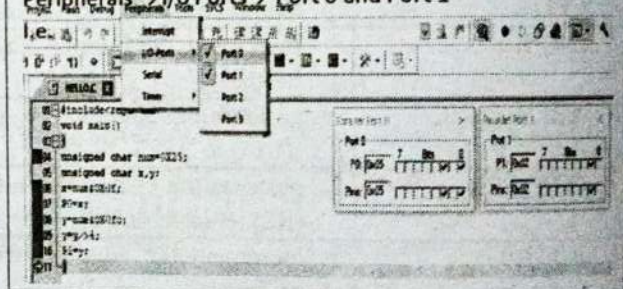
```
#include<reg51.h>
void main()
{
unsigned char num=0X36;
unsigned char x,y;
x=num&0X0f;
P0=x;
y=num&0X0f0;
y=y>>4;
P1=y;
}
```

Output:

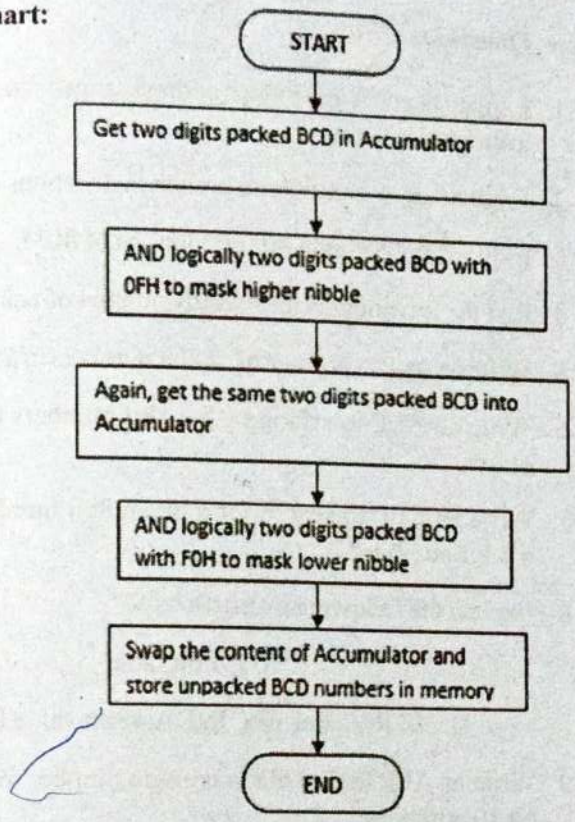
To see the result stored in P0 and P1 ports,In debugging mode,Click on,

Peripherals →I/O Ports→ Port 0 and Port 1

i.e.,



Flowchart:



(b) Unpacked BCD to packed BCD.

Label	Mnemonic	Description
	ORG 0000H	; Initialize starting address of Program.
	MOV R0,#50H	;Initialize memory pointer R0 to 50h
	MOV A,@R0	; Copy data from memory into Acc
	SWAP A	; Interchange lower and upper nibbles

	MOV R2,A	; save the accumulator content into reg R2
	INC R0	; increment internal memory pointer
	MOV A,@R0	; copy second byte data from memory into A
	ORL A,R2	; Perform bitwise OR operation between A & reg R2
	INC R0	; increment internal memory pointer
	MOV @R0,A	; Store content of Acc into memory
LOOP :	SJMP LOOP	; Stop
	END	; End

RESULT:

INPUT:

Internal Memory	50H	51H
Data	03	02

OUTPUT:

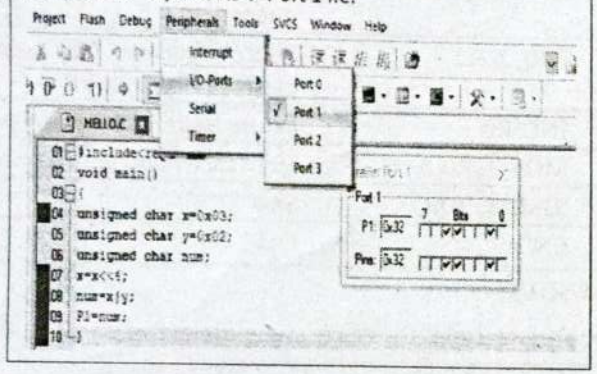
Internal Memory	52H
Data	32

C Program:

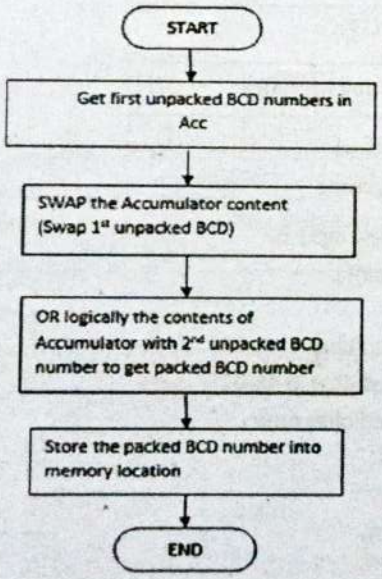
```
#include<reg51.h>
void main()
{
unsigned char x=0x03;
unsigned char y=0x02;
unsigned char num;
x=x<<4;
num=x|y;
P1=num;
}
```

Output:

To see the result stored in P1 port,
In debugging mode,Click on,
Peripherals →I/O Ports→ Port 1 i.e.



Flowchart:



Review Questions :

1 Explain the different branch address ranges supported by 8051 instructions

2 Explain any 5 conditional branch instructions of 8051

3 Explain any 5 single bit instructions of 8051

4 List the advantages and disadvantages of using 8051 C

5 Write an ALP to arrange a list 8bit numbers in ascending order

6 Write an ALP to arrange a list 8bit numbers in descending order

7 Write an ALP to search for a given 8bit number in a list of n 8 bit numbers

8 Explain the following instructions

i.RRC A ii. J B bit,label

iii.DJNZ Rn,label iv. CJNE A,#8bit value,label

9 Write an ALP to find the maximum number from a given 8-bit 10 numbers

10 Write an ALP to count the number of 1's and 0's in a given 8 bit number

11 Describe the general structure of embedded C program

12 Explain the data types in 8051 C.

13 Explain bit and SFR data types.

Week 10

Embedded C Continued....

Embedded C Continued...

10.1 Memory Types and Memory Models

10.1.1 Pointers

10.1.2 Types

10.2 Time Delays

10.3 Arithmetic & Logical Operators in 8051 C

* Simple Programs
* Lab Programs
* Review Questions

10.1 Memory Types and Memory Models

The 8051 has various types of memory space including internal & external code and data memory. Thus, when variables are declared we should know in which type of memory they reside.

For this purpose, several memory type specifiers such as code, data, idata, bdata, xdata&pdata are used.

Memory type	Description
code	Code memory (64 Kbytes)
data	Directly addressable internal data memory 128 bytes
bdata	Bit addressable internal data memory 16bytes
xdata	External data memory 64 Kbytes.
idata	Indirectly addressable internal data memory (256 bytes)
pdata	Paged external data memory (256 bytes)

Example;

1. Unsigned int **data** count; variable count resides in the internal data memory
2. Bit **bdata** num; variable num resides in bit addressable locations of internal data memory

Various memory models used in 8051 C

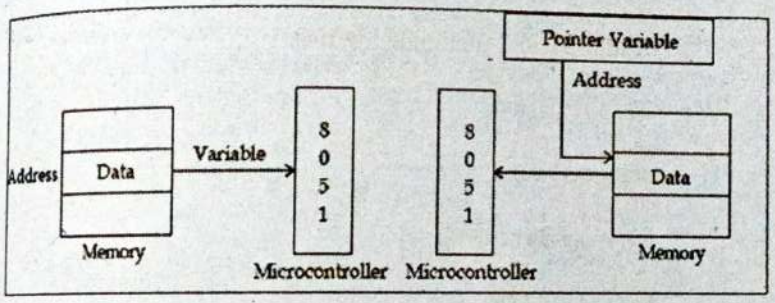
Memory model	Description
Small	Variables default to internal data memory (data)
Compact	Variables default to the first 256 bytes of external data memory(pdata)
Large	Variables default to external data memory(xdata)

Memory models are used to reside all declared variables in a default memory type. Once memory model is defined all variables are assigned to corresponding memory type & it is not necessary to explicitly specify memory type for all variables.

10.1.1 Pointers

In 8051 assembly language registers R0, R1 and DPTR are used as memory pointers. These registers are used to hold the address of the memory location where the data is stored. Indirect addressing is used to access the data.

In 8051 C compilers in direct addressing can be done through specially defined pointer variables. Pointer variable is a type of variable in which it holds the address of data stored in memory rather than the data itself. Normal variables hold thedata. The following figure illustrates the difference between a pointer variable and normal variable.



Declaration syntax:

➢ data type * pointer variable name;

Eg: int*p;

Here star (*) is used indicate a pointer variable

➢ intn;

int*p=&n;//pointerinitialization

The symbol ampersand '&' refers to 'address of'. The pointer variable 'P' is initialized to the address of normal variable n.

10.1.2 Types:

There are 2 types of pointers namely

a) Typed Pointers

b) Untyped Pointers

Typed Pointers:

Typed pointers are the one through which we can specify the memory area where pointer is stored & also specify the memory area where the data pointed by the pointer is stored.

Eg.:int data * xdata P=&n;

↑ ↑

Internal data memory External data memory

➢ The above statement declares a pointer variable 'P' to reside in external data memory (xdata) and this pointer points to data of type int that itself stored in the variable n in internal data memory (data).

➢ The memory type specifier data, before the * specifies the "data memory type" while.

➢ The memory type specifier 'xdata' after * specifies the pointer memory type.

➢ Thus, pointer declaration where data memory types are explicitly specified are called typed pointers.

Untyped Pointers:

Untyped pointers are another kind of pointers in which the data memory type is not mentioned explicitly when declaring pointers.

Eg: int * xdata P=&n;

↑ ↑

datatype External data memory

- Here it does not explicitly specify the memory type of data pointed by the pointers. The pointers can point to the data residing in any type of memory. Untyped pointers have the advantage that, they can be used to point to any data in dependent of the type of memory in which the data is stored. That's why untyped pointers are also called as "generic pointers".
- Untyped pointers are slower than typed pointers and are declared just like normal 'C".

10.2 Time Delays

- For various reasons it might be necessary to include some sort of time delays in the embedded system programs. Very accurate techniques using timers/counters in the processor exist to achieve this.
- But one simple approach (not involving timers) is to let the processor to count for a while(time) before it continues its execution. This can be achieved using a loop in the program where program does not do anything useful except incrementing the loop counter as shown below:

```
for(i=0;i<=n;i++)
{
.
}
```


- Here until the loop counter reaches the value 'n', program will not exit the loop & continue execution with next instruction following the loop statement.
- Thus, we can create delay in 8051 C by 2 ways

  a. Using simple for loop

  b. Using the 8051 timers

1 msec Delay:

- The approximate 1 msec delay is given by the count of 1275 times for an oscillating frequency of **11.0592MHz** for 8051.That is, for(i=0;i<1275;i++); gives **1msec** delay for **11.0592MHz** crystal frequency of 8051.
- For exact time duration we always use standard of "for loop" as below that is explained by ATMEL
- void msdelay(unsigned int itime)

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

- Note that with the above structure, the program counts to itimeX1275 times. If the value of itime=1 then it will produce 1ms delay.

Factors that can affect the accuracy of delay:

1. The design of 8051
2. The number of machine cycles
3. The number of clock periods per machine cycle
4. The crystal frequency connected to XTAL1-XTAL2 input pins
5. Compiler choice because different compilers produce different code

Important Note:

Indefinite Loop:

In embedded system it must be required to continuously execute a section of a program indefinitely (infinite number of times). To achieve this indefinite loop that is, loop without any exit condition can be used. The following two statements will execute their body (statement block) indefinite or infinite number of times.

for(;;)	while(1)
{	{
Statement block;	Statement block;
}	}

Sample programs:

1. **Write an 8051 C program to toggle bits of P1 continuously forever with some delay**


```
#include<reg51.h>
void main(void)
{
unsigned int x;
for(;;)  //repeat forever
{
P1=0X55;
for(x=0;x<40000;x++); //delay size unknown
P1=0XAA;
for(x=0;x<40000;x++);
}}
```

2. **Write 8051 C program to toggle bits of P0 & P2 ports continuously with a 250msec delay**

```
#include<reg51.h>
void MSdelay (unsigned int);
void main(void)
{
while(1) //repeat forever
{
P0=0X55;
P2=0X55;
MSdelay(250);
P0=0XAA;
P2=0XAA;
MSdelay(250);
}}
void MSdelay(unsigned int itime)
{
```

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

3. If the on period is 400 m sec and duty cycles is 75%. Write an 8051 C program to generate square wave on P2.1

The OFF period can be calculated as follows

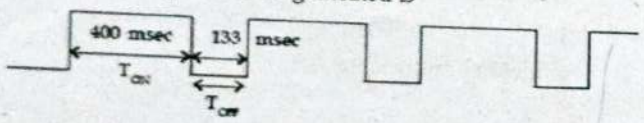
$$\text{Duty cycle} = \frac{T_{ON}}{T_{ON} + T_{OFF}}$$

$$D = 0.75, \quad T_{ON} = 400 \text{ msec}, \quad T_{OFF} = ?$$

$$0.75 = \frac{400 \text{ msec}}{400 \text{ msec} + T_{OFF}}$$

$$T_{OFF} = \frac{400 \text{ msec}}{0.75} - 400 \text{ msec} = 133 \text{ msec}$$

The square wave required to be generated is




```
#include<reg51.h>
void delay(unsigned int); // function declaration
void main(void)
{
while(1)
{
P2^1=1;
```



```
delay(400);
P2^1=0;
delay(133);
}}
void delay(unsigned int x);        // function defined
{
unsigned int i,j;   // variable declared
for(i=0; i<=x; i++)        // loop repeated for x times
for(j=0; j<=1275; j++);    // loop for 1 msec
}
```

10.3 Arithmetic & Logical Operators in 8051 C

All arithmetic and logical operators available in conventional C are also available in 8051 C. They are listed as follows:

a) Arithmetic operators

1. + Addition
2. -- Subtraction
3. * Multiplication
4. / Division (gives quotient after division)
5. % Modulus (gives remainder after division)

b) Relational operators (result is True or False)

1. == True if values are equal.
2. ! = True if values are not equal.

3. < True if the first value is less than second.
4. > True if the first value is greater than second.
5. <= True if the first value is less than or equal to second value.
6. >= True if the first value is greater than or equal to second value.
7. && True if both values are true.
8. || True if either value is true.

c) Bitwise logical operators

1. & logicalAND
2. | logicalOR
3. ^ logicalXOR
4. ~ logicalNOT (One's complement)

d) Bitwise shift operators

1. << shift left (equal to multiply by 2)
2. >> shift right (equal to divide by 2)

e) Assignment operator

1. = set equal to

Illustration of arithmetic, logical and shift operators.

a) P1 =0X55&0XFF

Bitwise AND operation

	55	=0101 0101
	FF	=1111 1111
		55H=01010101

P1 =0X55

b) P1 =0X55^0XFF

Bitwise OR operation

	55	=0101 0101
	FF	=1111 1111
		55H=11110111

P1 =0XFF

c) P2 =~0X86

It is a bitwise NOT operation

86	=10000110
	79=01111001

P2 =0X79

d) P0 =0X23>>3

It is a right shift operation 23 is shifted right by 3 places.

23	=00100011
Right shift 3 times	=00000100

P0=04

Sample programs:

1. Write a C program to read P1.3 and send it to P2.0 after inverting it.

Solution:

```
#include<reg51.h>
bit x;
void main(void)
{
```

```
P1^3 = 1;          // make P1.3 as input pin
while(1)
{
x=P1^3;  // read a bit from P1.3
P2^0=~x; // complement & send it to P2.0
}}
```

2. Write 'C' program to toggle all the pins of P1 using XOR operation. Initially send 55H to P1

Solution:

```
#include<reg51.h>
void main(void)
{
P1=0X55; //send 55 to P1
while(1) //Repeat forever
{
unsigned char i;
P1=P1^0XFF;//XOR operation to toggle P1
for(i=0;i<200;i++);//small delay loop
}}
```

Lab Programs

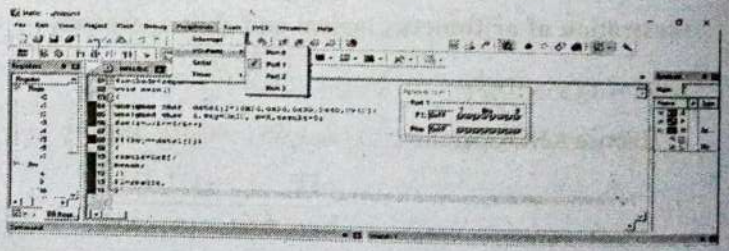
1. Write and execute a program to search a given 8-bit number in an array of N numbers using embedded C.

```
#include<reg51.h>
void main()
```



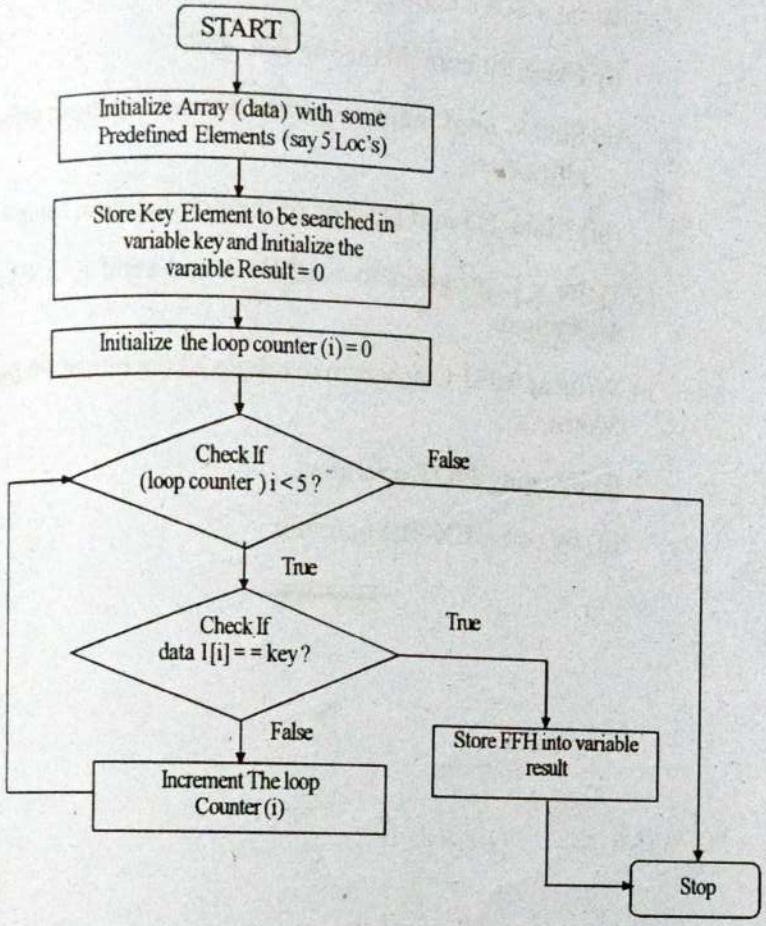
```
{
unsigned char data1[]={0x10,0x20,0x30,0x40,0x50};
unsigned char i,key=0x20, n=5,result=0;
for(i=0;i<5;i++)
{
if(key==data1[i])
{
result=0xff;
break;
}}
P1=result;
}
```

Output:
To see the result stored in P1 port,
In debugging mode,Click on,
Peripherals →I/O Ports→ Port 1
I.e.



If FFH is available in P1 port, means search is successful and if 00H is available in P1 means search is unsuccessful.

Flowchart:



2. Write and execute a program to toggle a particular bit in the internal RAM with the use of delay subroutine.

```
#include<reg51.h>
void delay(unsigned int t)
{
unsigned int i,j;
```

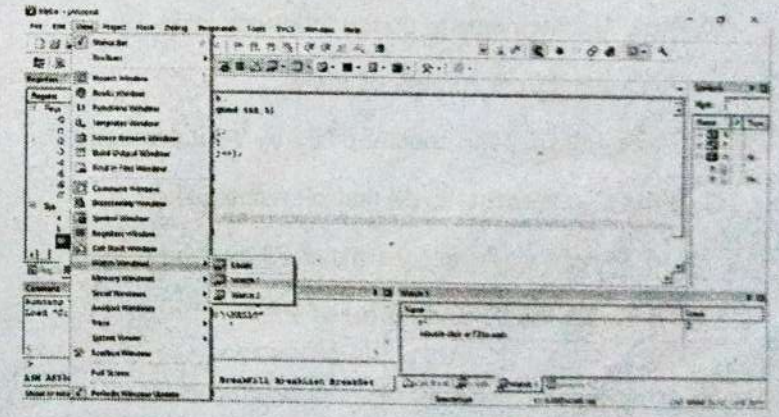


```
for(i=0;i<t;i++)
for(j=0;j<1275;j++);
}
void main()
{
bit b1;
b1=1;
delay(200);
b1=0;
while(1);
}
```

Output:

View ' Watch Windows ' Watch 1

Observe the output in debug mode by pointing the cursor to the statement b1=1;



Review Questions :

1. Explain the different memory models available in 8051C.
2. Explain the different memory type specifiers available in 8051C.
3. Explain the different logical operators available in 8051C.
4. Explain the different arithmetic operators available in 8051C.
5. Write a C program to read P1.2 and send it to P2.3 after inverting it.
6. Write a 8051 C program to do the following (i) Clear P0 contents using EX -OR (ii) Set D4 bit of memory location temp1 without affecting other bits (iii) Mask D5 and D3 bits of memorylocation temp2.
7. Write a 8051 C program to convert ASCII digits '4' and '7' into packed BCD and to display on port P1.
8. Write 8051 C program to turn bit P0.3 on and off 10,000 times.
9. Write 8051 C program to convert packed BCD 0X34 to ASCII and display the bytes on P1 and P2.
10. Write C statements to do the following tasks.

    (i) To toggle the content of P0.

    (ii) To left shift the content of P2 by 2-bit positions.

11. Write C statements to do the following tasks.

    (iii) To right shift the contents of P3 by 4-bit position.

    (iv) To set the d1 and d3 bit of P1.

12. Write a 8051 C program to do the following :

    (i) Clear P0 contents using EX -OR

    (ii) Set D4 bitof memory location temp1 without affecting other bits

    (iii) Mask D5 and D3 bits of memorylocation temp2

13. Write a C program to read P1.2 and send it to P2.3 after inverting it.
14. Write an 8051 C program to toggle all the bits of P0 for every 500ms.

    (i) by using NOT operator

    (ii) by using EX-OR operator.

Week 11

I/O Ports & Interrupt Programming

I/O Ports & Interrupt Programming

11.1 Features of I/O Ports

11.1.1 Interfacing LED

11.2. Interrupts of 8051

11.2.1 Comparison of Interrupt Method and Polling

11.2.2 Steps in Executing an Interrupt

11.2.3 Bit Structure of IE

11.2.4 Bit Structure of IP (Interrupt Priority) Register

11.3 Examples Showing Enabling, Disabling and Priority Setting of Interrupts

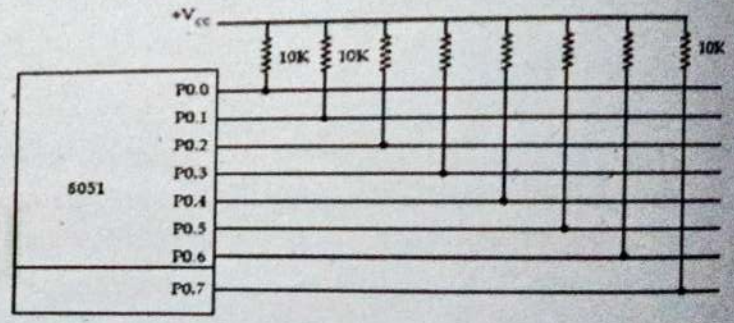
* Lab Programs

* Review Questions

11.1 Features of I/O Ports

- 8051 has four I/O ports namely P0, P1, P2, P3. All ports upon Reset are configured as output ports.
- To use any of these ports as input port, it must be programmed by writing 1's to all bits.
- To use the Port 0 pins of 8051 as both input and output ports, each pin must be connected externally to a 10 K resistor. This is due to the fact that P0 is an open drain or open collector. It is also used for sending address and data to an external memory.
- Port1, Port2, Port3 are all similar and requires no Pullup resistors.
- Port2 is used for dual purposes. It is used to carry the address bits when interfacing with external memory.
- PORT 3 acts as a normal Input/Output port, but it has additional functions such as, serial transmit and receive pins, two external interrupt pins, two external counter inputs, read and write pins for memory access.

PORT 0 (P0) (Pins 32 to 39)



PORT 0 (P0) can be used as a general purpose 8-bit input or output port. It is used for dual purpose i.e.,when no external memory is present, it is used as general - purpose input/output

port. But if external memory is connected to 8051 then P0 provides both address and data (AD0-AD7).

8051 multiplexes address and data through P0 to save pins. To use pins of Port 0 as input/output ports each pin must be connected externally to 10K ohm pull-up resistor as shown in fig. 4.1. This is because the P0 is an open drain or open collector terminals. Upon reset the P0 is configured as output port. To make P0 as input, the P0 must be programmed by writing 1 to all its bits.

PORT 1 (P1) (Pins 1 to 8)

The port 1 (P1) is a general-purpose input/output port with internal pull-ups. Therefore, the port does not need any external pull-up resistors. Upon reset the port 1 is configured as output port. To make P0 as input, the P0 must be programmed by writing 1 to all its bits.

PORT 2 (P2) (Pins 21 to 28)

PORT 2 (P2) can also be used as a general purpose 8 bit port when no external memory is present, but if external memory access is required then PORT 2 (P2) will act as an address bus in conjunction with PORT 0 (P0) to access external memory. PORT 2 (P2) acts as A8-A15. Upon reset the P2 is configured as output port. To make P2 as input, the P2 must be programmed by writing 1 to all its bits. Port 2 does not need any external pull-up resistors.

PORT 3 (P3) (Pins 10 to 17)

PORT 3 (P3) acts as a normal input/output, but port 3 (P3)

has additional functions such as, serial transmit and receive pins, two external interrupt pins, two external counter inputs, read and write pins for memory access. Port 3 does not need any external pull-up resistors. Some alternate pin functions of PORT 3 are listed below;

Pin Number	Function	Description
P3.0 – 10	Rx	Serial Input
P3.1 – 11	Tx	Serial Output
P3.2 – 12	$\overline{INT0}$	External Interrupt 0
P3.3 – 13	$\overline{INT1}$	External Interrupt 1
P3.4 – 14	T0	Timer 0
P3.5 – 15	T1	Timer 1
P3.6 – 16	$\overline{WR}$	External Memory Write
P3.7 – 17	$\overline{RD}$	External Memory Read

11.1.2 Interfacing LED

- LED is nothing but light emitting diode which emits light only during forward bias. It is used as indicator in almost all electric instruments and in all electronic instruments. The LED can be interfaced to 8051. The current supplied by 8051 may not be sufficient to drive the LED. Therefore, a driver transistor is used to increase the output current.

I/O Ports & Interrupt Programming

11.1 Features of I/O Ports

11.1.1 Interfacing LED

11.2. Interrupts of 8051

11.2.1 Comparison of Interrupt Method and Polling

11.2.2 Steps in Executing an Interrupt

11.2.3 Bit Structure of IE

11.2.4 Bit Structure of IP (Interrupt Priority) Register

11.3 Examples Showing Enabling, Disabling and Priority Setting of Interrupts

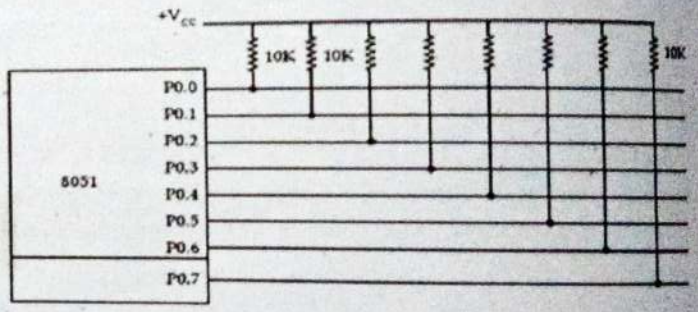
* Lab Programs

* Review Questions

11.1 Features of I/O Ports

- 8051 has four I/O ports namely P0, P1, P2, P3. All ports upon Reset are configured as output ports.
- To use any of these ports as input port, it must be programmed by writing 1's to all bits.
- To use the Port 0 pins of 8051 as both input and output ports, each pin must be connected externally to a 10 K resistor. This is due to the fact that P0 is an open drain or open collector. It is also used for sending address and data to an external memory.
- Port1, Port2, Port3 are all similar and requires no Pullup resistors.
- Port2 is used for dual purposes. It is used to carry the address bits when interfacing with external memory.
- PORT 3 acts as a normal Input/Output port, but it has additional functions such as, serial transmit and receive pins, two external interrupt pins, two external counter inputs, read and write pins for memory access.

PORT 0 (P0) (Pins 32 to 39)



PORT 0 (P0) can be used as a general purpose 8-bit input or output port. It is used for dual purpose i.e.,when no external memory is present, it is used as general - purpose input/output

port. But if external memory is connected to 8051 then P0 provides both address and data (AD0-AD7).

8051 multiplexes address and data through P0 to save pins. To use pins of Port 0 as input/output ports each pin must be connected externally to 10K ohm pull-up resistor as shown in fig. 4.1. This is because the P0 is an open drain or open collector terminals. Upon reset the P0 is configured as output port. To make P0 as input, the P0 must be programmed by writing 1 to all its bits.

PORT 1 (P1) (Pins 1 to 8)

The port 1 (P1) is a general-purpose input/output port with internal pull-ups. Therefore, the port does not need any external pull-up resistors. Upon reset the port 1 is configured as output port. To make P0 as input, the P0 must be programmed by writing 1 to all its bits.

PORT 2 (P2) (Pins 21 to 28)

PORT 2 (P2) can also be used as a general purpose 8 bit port when no external memory is present, but if external memory access is required then PORT 2 (P2) will act as an address bus in conjunction with PORT 0 (P0) to access external memory. PORT 2 (P2) acts as A8-A15. Upon reset the P2 is configured as output port. To make P2 as input, the P2 must be programmed by writing 1 to all its bits. Port 2 does not need any external pull-up resistors.

PORT 3 (P3) (Pins 10 to 17)

PORT 3 (P3) acts as a normal input/output, but port 3 (P3)

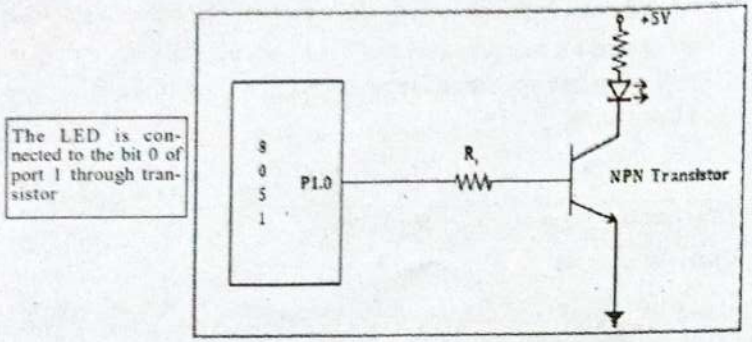
has additional functions such as, serial transmit and receive pins, two external interrupt pins, two external counter inputs, read and write pins for memory access. Port 3 does not need any external pull-up resistors. Some alternate pin functions of PORT 3 are listed below;

Pin Number	Function	Description
P3.0 – 10	Rx	Serial Input
P3.1 – 11	Tx	Serial Output
P3.2 – 12	$\overline{INT0}$	External Interrupt 0
P3.3 – 13	$\overline{INT1}$	External Interrupt 1
P3.4 – 14	T0	Timer 0
P3.5 – 15	T1	Timer 1
P3.6 – 16	$\overline{WR}$	External Memory Write
P3.7 – 17	$\overline{RD}$	External Memory Read

11.1.2 Interfacing LED

- LED is nothing but light emitting diode which emits light only during forward bias. It is used as indicator in almost all electric instruments and in all electronic instruments. The LED can be interfaced to 8051. The current supplied by 8051 may not be sufficient to drive the LED. Therefore, a driver transistor is used to increase the output current.

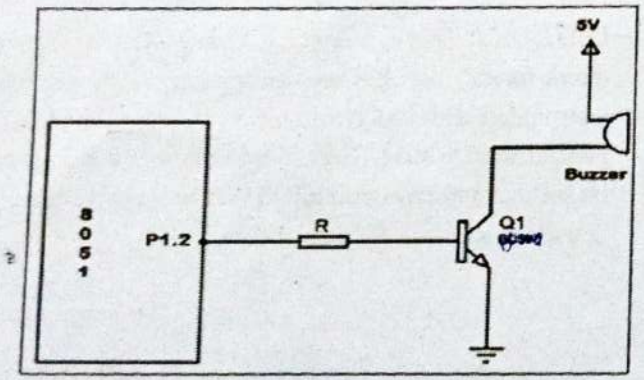
➢ The figure shows the LED interfacing circuit to 8051.



Interfacing buzzer:

➢ Buzzer is an electrical device, which is similar to a bell that makes a buzzing noise and is used for signalling. Typical uses of buzzers and beepers include alarm devices, timers and confirmation of user input such as a mouse click or keystroke.

➢ The buzzer can be interfaced to 8051. The current supplied by 8051 may not be sufficient to drive the Buzzer ON. Therefore, a driver transistor is used to increase the output current



Assembly language and Cprogram to turn ON LED/Buzzer for some timer interval then turn LED/Buzzer OFF, connected to port1.0 and port 1.2 respectively.

ALP:

```
        ORG0000H
BACK:   CLR P 1.0        ;LED=OFF
        CLR P 1.2        ;Buzzer = OFF
        ACALLDELAY       ;Calldelay
        SETBP1.0         ;SwitchONLED
        SETB P1.2        ;Switch ON Buzzer
        ACALLDELAY       ;Calldelay
        SJMPBACK         ;Repeat
DELAY:  MOVR3,#200
HERE:   DJNZR3,HERE
        RET
        END
```

C Program:

```
#include<reg51.h>
void delay (void); //function prototype
void main(void)
{
while(1)//repeated loop
{
P1^0 = 0;//LED is turned off
```

```
P1^2 = 0;// Buzzer is turned off
delay( );
P1^0 = 1;// LED is turned ON
P1^2 = 1;// Buzzer is turned ON
delay( );
}
}
void delay(void)//function definition
{
unsigned char i, j;
for(i=0; i<200; i++)
for(j=0; j<200; j++);
}
```

11.2 Interrupts of 8051

- Interrupt is an external or internal event that interrupts the normal program execution and requests the microcontroller for service.
- The event may take place at any time. Interrupt handling logic is incorporated inside the chip. In such a case, processor will suspend current execution of the program, execute a subroutine, and then resume normal program flow. The subroutine that is associated with the interrupt is called an interrupt handler or Interrupt service routine (ISR), is only executed when a certain event (interrupt) occurs.

- The situation can be seen very frequently, in our every daily life. Suppose a person is busy in doing some work, say writing a letter and all of a sudden telephone bell rings. Then the person will stop writing the book, answer the telephone, and resume the writing the book. Sometime there are 4,5 telephone lines are available. In that case he may have to decide about the priority, in answering the phones.Some time he himself is very busy in important meeting, and does not want to get disturbed by the phone calls. He simply ignores the calls.
- In a Microcontroller Interrupts are handled similar to the above example. When the Interrupt occurs, microcontroller suspends the execution of the main program temporarily and branches to the ISR.The ISR (sub-Program) executes till the appearance of RETI (Return from Interrupt) instruction. Then the controller continues to execute the main program from where it left off.

11.2.1 Comparison of Interrupt Method and Polling

A single Microcontroller can serve several devices.There are two ways to do that i.e., Interrupt and Polling. The two interrupt methods can be compared as mentioned below:

Interrupt	Polling
In this method when ever any device needs service the device notifies the Microcontroller by sending Interrupt signal.	In this method Microcontroller continuously monitors the status of the device. If the condition is met then it gives service.
It saves the Microcontroller's useful time.	The Microcontroller wastes its time in checking the status of each device.
It is efficient.	It is inefficient

In this method device can be served on Priority basis.	Polling method uses" round-robin" fashion of checking the device status. No Priority can be assigned.
Microcontroller can masks or ignores any request from any device.	Masking device request is not possible.

Different types of Interrupts:

1. Reset
2. External hardware interrupt 0 → INT0
3. Timer 0 → TF0
4. External hardware interrupt 1 → INT1
5. Timer 1 → TF1
6. Serial Com → RI/TI

Interrupt Vector Table (IVT):

For every interrupt there is a fixed location in memory that holds the address of ISR. Thus, the group of memory locations which are set aside to hold address of ISR is called Interrupt Vector Table (IVT).

Interrupt	Flag	AddressofISR
1.Reset	_	0000h
2.External0	INT0(P3.2)	0003h
3.Timer0	TF0	000Bh
4.External1	INT1(P3.3)	0013h
5.Timer1	TF1	001Bh
6.Serial	RI/TI	0023h

- Reset is one of the interrupts, when it is activated 8051 jumps to address 0000. Two interrupts are set aside for Timer 0and Timer 1, two interrupts are set a side for external hardware interrupts INT0 and INT1. One serial Interrupt-a single Interrupt that manages both receive and transmit.
- In vector table limited number of bytes is set aside for each interrupt. For example, a total of 8 bytes from location 0003 to 000A is set aside for external hardware interrupt 0.
- If the service routine for a given interrupt is short enough to fit in the memory space allocated to it, then it is placed in vector table otherwise an LJMP instruction is placed in the vector table to point to the address of ISR. In that case rest of the bytes allocated to that interrupt are unused.

11.2.2 Steps in Executing an Interrupt

On receiving an Interrupt, 8051 microcontroller executes the following steps:

1. It completes the execution of the current Instruction and saves the contents of the Program Counter (PCholds the address of the next instruction to be executed) on to the stack.
2. It also saves the current status of all the interrupts internally.
3. It jumps to the fixed memory location called Interrupt Vector table that contains the address of the ISR (Interrupt Service Routine).
4. Microcontroller starts executing the ISR until it encounters RETI (return from ISR) the last instruction of the subroutine.
5. Microcontroller returns to the main program from the ISR after executing RETI instruction.

6. The Program counter gets address from the stack, and then it starts executing from that address.

How to enable an interrupt:

- Upon reset all interrupts are disabled (masked), that is none will be responded by the microcontroller if they are activated. The interrupts must be enabled by software method in order for microcontroller to respond them
- There is a register called Interrupt Enable (IE) that is responsible for enabling & disabling interrupts. IE register is bit/byte addressable.

The IE register format:

1. The bit D7 of the IE register must be set to High.
2. If EA=1 Interrupts are enabled and will be respond to if their corresponding bits in IE are high. If EA=0 then Interrupts are disabled.

The structure of Interrupt Enable Register is as follows;

11.2.3 Bit Structure of IE

D7							D0
EA	—	ET2	ES	ET1	EX1	ET0	EX0
		TF2	Serial	TF1	INT1	TF0	INT0

Name	Bit	Bit Address	Description
EA	IE.7	0AFh	Global interrupt enable / disable. If EA=0 all interrupts are disabled. If EA=1 Each interrupt is individually enabled / disabled.

--	IE.6	0AEh	Undefined.
ET2	IE.5	0ADh	Enables or Disables the Timer 2 over flow interrupt (8052).
ES	IE.4	0ACh	Enables or Disables Serial port interrupt.
ET1	IE.3	0ABh	Enables or Disables Timer 1over flow interrupt.
EX1	IE.2	0AAh	Enables or Disables External interrupt 1.
ET0	IE.1	0A9h	Enables or Disables Timer 0 overflow interrupt.
EX0	IE.0	0A8h	IE.0 Enables or Disables External interrupt 0.

Interrupt Priority in 8051

Consider a situation when all the Interrupts enabled at the same time, usingIE register. If two or more interrupts occurs simultaneously, then 8051 does the following steps:

1. It latches all the interrupts occurred at the same time.
2. It services the interrupts according to their priority.
3. After servicing, it will service the priority interrupt that occurred instead of returning to the main program. This will continue till all the interrupts are over then return to the main program.

When the 8051 is powered up, the priorities assigned as shown in the following table ;

Highest to Lowest Priority		
1	ExternalInterrupt0	INT0
2	TimerInterrupt0	TF0
3	ExternalInterrupt1	INT1
4	TimerInterrupt1	TF1
5	SerialCommunication	RI+TI

Note:

- Here their order is as per their occurrence in Interrupt Vector Table (IVT)
- The default priority upon reset can be changed by programming the IP (Interrupt Priority) register.

11.2.4 Bit Structure of IP (Interrupt Priority) Register

Bit Structure of IP

D7							D0
—	—	PT2	PS	PT1	PX1	PT0	PX0
		TF2	Serial	TF1	INT1	TF0	INT0

Summary of Interrupt Priority register:

Name	Bit	Bit Address	Description
--	IP.7	--	Undefined
--	IP.6	--	Undefined
PT2	IP.5	0BDh	Priority for Timer 2 Interrupt (8052 only)

PS	IP.4	0BCh	Priority for Serial Port Interrupt -RI/TI
PT1	IP.3	0BBh	Priority for Timer Interrupt 1 -TF1
PX1	IP.2	0BAh	Priority for External Interrupt 1 - INT 1
PT0	IP.1	0B9h	Priority for Timer Interrupt 0 -TF0
PX0	IP.0	0B8h	Priority for External Interrupt 0 -INT 0

The priority bit of IP register is made high to assign higher priority and made 0f or lowest priority. Suppose the Interrupt Priority register is loaded with 15h using the instruction MOV IP, #15 h. If interrupts INT1, TF0, TI and TF1 occurs simultaneously, then 8051service the interrupts in the following manner:

- IP=15h=00010101 indicates that the priority bits PS,PX1 and PX0 are assigned higher priority. Hence high priority group consists of 3 interrupts takes the priority levels (according to the priority levels at reset) INT0,
- INT1 and Serial Interrupt (RI/TI). The remaining interrupts are of low priority inthe order TF0 and TF1.
- If interrupts INT1, TF0, TI and TF1 occurs simultaneously then 8051 services the interrupts in following order:

1) INT1(Higher priority group)
2) Serial Interrupt (RI/TI) (Higher priority group)
3) TF0 (Lower Priority group)
4) TF1(Lower Priority group)

11.3 Examples Showing Enabling, Disabling and Priority Setting of Interrupts

It has already explained in the preceding section that, Interrupts are individually enabled or disabled through a special function register IE (Interrupt Enable).The Register IE is bit addressable. When the microcontroller is reset, all the interrupts are disabled ie interrupts are said to be masked, under this condition microcontroller will not respond to any interrupt. The interrupts must be enabled through IE register. To Enable an Interrupt, the bit D7 of IE register must be set to1(HIGH).

1. Write assembly language instructions to enable Serial Interrupt, timer 0 and External Hardware interrupt 1.

Solution:

MOV IE,#10010110B ; Set EA=1 and enable the required Interrupts by

Making the corresponding Bits to High.

* This can also be done using single bit Instructions as follows:

SETB IE.7 ; Enableall.

SETB IE.4 ; EnableSerialInterrupt.

SETB IE.1 ; EnableTimer0Interrupt.

SETB IE.2 ; EnableExternalInterrupt1

Similarly, to disable an interrupt individually clear the used bit.

E.g.:CLR IE.3.

2. Write assembly language instruction to assign Highest priority to INT1.

Solution:

MOV IP#00000100B.

* The single bit instruction can also be used set the priority.

SETB IP.2 //Enable IP.2 to set the higher Priority.

3. Write assembly language instruction to assign Higher priority to Serial Interrupt and Timer interrupt1.

Solution:

MOV IP #00011000B.

* The single bit instructions can also be used set the priority.

SETB IP.3// Enable IP.3 to set the higher Priority.
SETB IP.5//Enable IP.5 to set the higher Priority.

In this case, when both the interrupts Serial and Timer1 (TF1) occurs simultaneously, then TF1 will be served first later Serial interrupt is attended this is because of Highest priority among higher priority interrupts will be served first.

4. Write assembly language instructions to disable all Interrupts.

Solution:

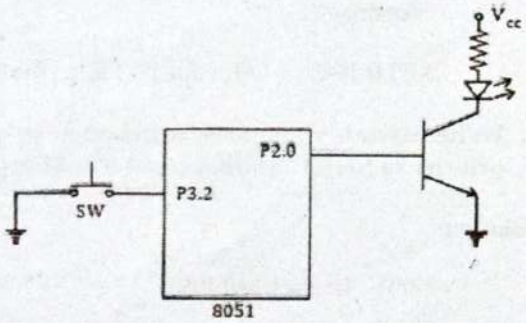
MOV IE,#00000000B ; Set EA=0, disables all Interrupts.

* This can also be done using single bit Instructions as follows

CLR IE.7 ;Disables all Interrupts.

Lab Programs:

1. Write and execute an embedded C program to toggle the LED/buzzer with tone using push-button switch.



**Here basically microcontroller to scan the switch status and indicate it using LED/buzzer connected toP3.2.

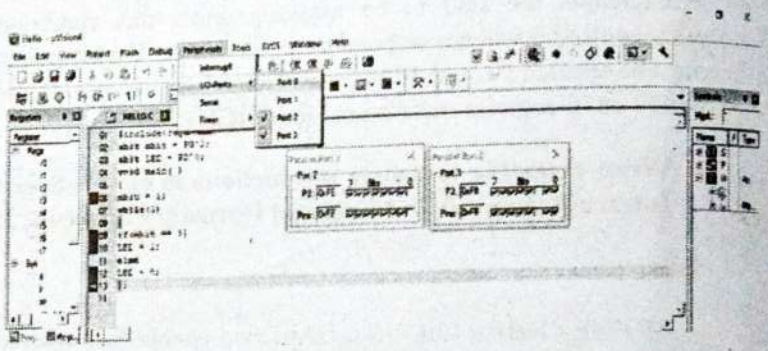
C program:

```
#include<reg51.h>
sbitswitch = P3^2;
sbit LED = P2^0;
void main( )
{
switch = 1;// make P3.2 as input
while(1)
{
if(mbit == 1)//Results true If switch is in pressed condition
LED = 1;// LED ON
```



```
else
LED = 0;//LED OFF
}}
```

Output:



Here, In debugging mode output is seen in, peripherals→ I/O ports

One can see change in the output (P2.0 (LED)) status just by operating Input (P3.2 (Switch)) in the I/O ports window.

2. Write ALPs to enable, disable and priority setting of interrupts and verify it in IE and IPregisters.

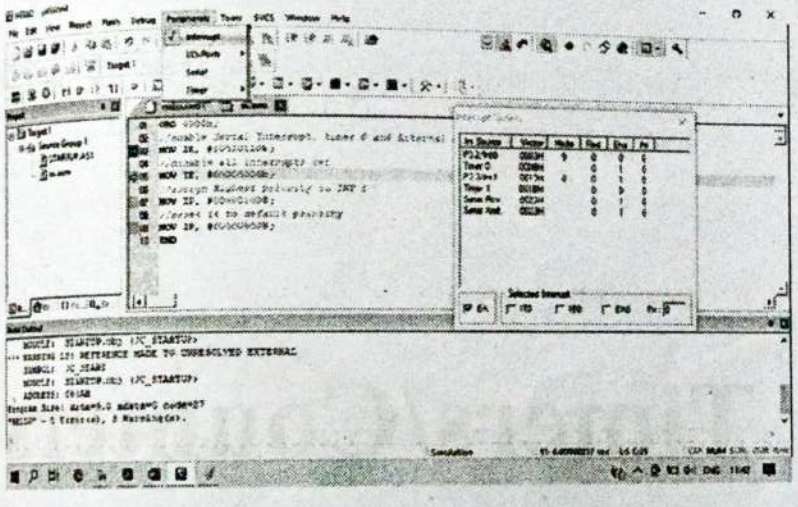
ALP:

```
ORG 0000h;
//enable Serial Interrupt, timer 0 and External Hardware interrupt 1.
MOV IE, #10010110b;
```

```
//disable all interrupts set
MOV IE, #00000000b;
//assign Highest priority to INT 1
MOV IP, #00000100B;
//reset it to default priority
MOV IP, #00000000B;
END
```

Result:

To see the result in IE & IP register, in debugging mode ' click on Peripherals'click on interrupt tab



Review Questions :

1. Compare the interrupt method and polling method of servicing devices.
2. Explain the bit structure of IP register.
3. Explain the bit structure of IE register.
4. List the steps involved in executing an interrupt.
5. Write an ALP and a C program to send values 0 to 4 to port P2.
6. Write an ALP and a C program to toggle the bits of port P3 continuously.
7. Write a program to generate a square wave of 50% duty cycle on bit 3 of port 1.
8. Write an 8051 C program to read a byte from P0. If it is less than 50, send it to P1, otherwise, send it to P2.
9. Discuss the interrupt priority order achieved by the execution of MOV IP,#11H instruction.
10. List the interrupts of 8051 and their vector locations.
11. Write the schematic, algorithm and a program for 8051 to sense the push button switch andaccordingly control the on/ off of LED.
12. Write a C program to continuously send 00h to 20h to port P0.

Week 12

Timers/Counters & Serial Communication

Timers/Counters & Serial Communication

12.1 Timer/Counters
12.1.1 Timer 0 Register
12.1.2 Timer 1 Register
12.1.3 TMOD Register
12.1.4 TCON Register
12.1.5 Model Operation of Timers
12.1.6 Mode 2 (8-bitautoreload) Timer Programming
12.1.7 Programming Mode 2 - Timer 1
12.2 Time Delay Generation Calculation
12.3 8051 - Serial Communication
12.3.1 SBUF Register (Serial Buffer Register)
12.3.2 SCON Register (Serial Control Register)
12.3.3 Working of Serial Port
* Lab Programs
* Review Questions

12.1 Timers/Counters

Microcontroller 8051 uses its on-chip timers and counters when interfaced with the external world for timing and control applications like interval timing, event counting, periodic interrupt scheduling (Real Time Operating System - RTOS), frequency/ speed measurement etc.

The 8051 microcontroller has two 16-bit timer/counters. These are denoted as timer 0 and timer 1. They can be used either as timers or counters. Timers are used to generate time delay. Counters are used to count the events (number of clock pulse).

Two external pins of 8051 are associated with timers/counters, namely.

Pin No.	Port Pin	Pin Name	Description
14	P3.4	T0	Timer/counter0externali/p
15	P3.5	T1	Timer/counter1externali/p

The 8051 has two timers, one mode and one control register:

- Timer 0 (T0)
- Timer 1 (T1)
- Timer mode Register (TMOD)
- Timer Control Register (TCON)

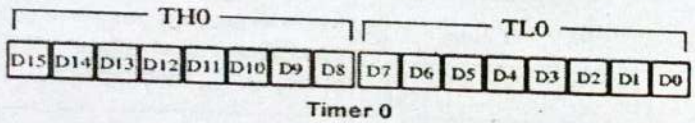
They can be used either as timers or as event counters. Both Timer 0 and Timer 1 are 16 bits wide. Whereas TMOD and TCON are 8-bit wide.

Since the 8051 has an 8-bit architecture, each 16-bit timer is

accessed as two separate registers of low byte and high byte. Each timer along with mode and control are discussed separately in the below section

12.1.1 Timer 0 Register

- The 16-bit register of Timer 0 is accessed as low byte and high byte. The low byte register is called TLO (Timer 0 low byte) and the high byte register is referred to as THO (Timer 0 high byte).
- These registers can be accessed and read like any other register, such as A, B, RO, Rl, R2, etc.



- For example, the assembly instructions as below:

MOV TL0, #4FH	;	moves the value 4FH into TLO, the low byte of Timer 0.
MOV TH0, A	;	moves content of register A into THO, the high byte of Timer 0
MOV R5, TH0	;	saves THO (high byte of Timer 0) in R5.

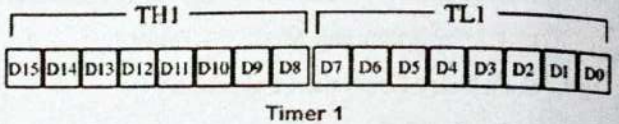
- For example, the 8051 C operational statements as below:

TL0 = 0x4F	;	moves the value 4FH into TLO, the low byte of Timer 0.
TH0 = Acc	;	moves content of register A into THO, the high byte of Timer0

num = TH0	;	saves THO (high byte of Timer 0) in variable num

12.1.2 Timer 1 Register

- Timer 1 register is same as that of timer 0. The 16-bit register of Timer 1 is accessed as low byte and high byte. The low byte register is called TL1 (Timer 1 low byte) and the high byte register is referred to as TH1 (Timer 1 high byte).
- These registers can be accessed and read like any other register, such as A, B, RO, Rl, R2, etc



- For example, the assembly instructions as below:

MOV TL1, #4FH	;	moves the value 4FH into TL1, the low byte of Timer 1.
MOV TH1, A	;	moves the content of register A into TH1, the high byte of Timer 1
MOV R5, TH1	;	saves THO (high byte of Timer 1) in R5.

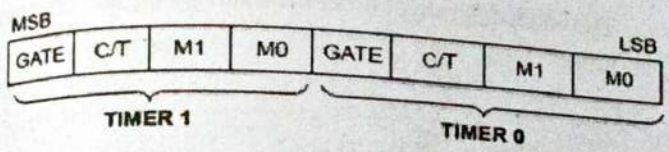
- For example, the 8051 C operational statements as below:

TL1 = 0x4F	;	moves the value 4FH into TL1, the low byte of Timer 1.
TH1 = Acc	;	moves the content of register A into TH1, the high byte of Timer 1

num = TH1 ; saves THO (high byte of Timer 1) in variable num.

12.1.3 TMOD Register

➢ This is an 8-bit register which is used by both timers 0 and 1 to set the various timer modes. In this TMOD register, lower 4 bits are set aside for timer0 and the upper 4 bits are set aside for timer1. In each case, the lower 2 bits are used to set the timer mode and upper 2 bits to specify the operation.



➢ **GATE bit:** Every timer has a means of starting and stopping. Some timers do this by software, some by hardware, and some have both software and hardware controls.

GATE=1: The hardware way of starting and stopping the timer by an external source i.e., by INT1/INT0.

GATE=0: Start and stop the timers is by software. No need of external hardware.

➢ **C/T bar bit:** It is used to decide whether a timer is used as a time delay generator or an event counter.

C/T bar = 0: Timer 0/1 is used as a timer

C/T bar = 0: Timer 0/1 is used as a counter

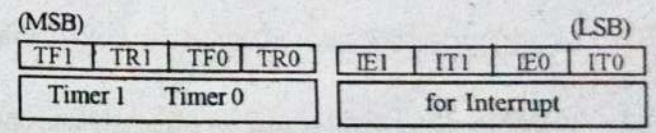
➢ **M1 & M0 bits:** These are 2 bits used in combine to decide in which mode timer is being operated in

M1	M0	MODE	DESCRIPTION
0	0	0	13-bit timer
0	1	1	16-bit timer
1	0	2	8-bit auto-reload
1	1	3	Split mode

➢ The same explanation holds good for lower nibble (timer 0) as well as higher nibble (timer 1).

12.1.4 TCON Register

This is an 8-bit register which is used to control the timer/counter such as starting the timer/counter, stopping timer/counter etc. Bits and functions of every bit of TCON register are as follows:



➢ **TF1: Timer1 over flow flag bit.**

TF1 = 1, when timer1 rolls from all 1s to 0 i.e., overflows

TF1 = 0, Overflow not occurred in timer 1

➢ **TR1: Timer 1 run control bit.**

TR1 = 1, enable/start timer1

TR1 = 0, halt/stop timer1

➢ **TF0: Timer0 over flow flag bit.**

TF0 = 1, when timer0 rolls from all 1s to 0 i.e., overflows

TF0 = 0, Overflow not occurred in timer0

- **TR0: Timer 0 run control bit.**

  TR0 = 1, enable/start timer0 count

  TR0 = 0, halt/stop timer0 count

- **IE1: External interrupt 1 Edge flag.** Not related to timer operations.

- **IT1: External interrupt1 signal type control bit.**

  IT1 = 1 Enable external interrupt 1, to be triggered by a falling edge signal.

  IT1 = 0 enable a low-level signal on external interrupt1 to generate an interrupt.

- **IE0: External interrupt 0 Edge flag.** Not related to timer operations.

- **IT0: External interrupt0 signal type control bit.**

  IT0 = 1 Enable external interrupt 0, to be triggered by a falling edge signal.

  IT0 = 0 enable a low-level signal on external interrupt0 to generate an interrupt.

12.1.5 Mode1 Operation of Timers

In the preceding section, the importance of TMOD register has been explained. It helps to initialize the timer mode in the beginning of a program. There are four modes of operation i.e., mode0, mode1, mode2 and mode3. The most commonly used mode1 and mode2 are explained as follows:

Mode1-Timer Programming

The characteristics of mode1 operation are as follows:

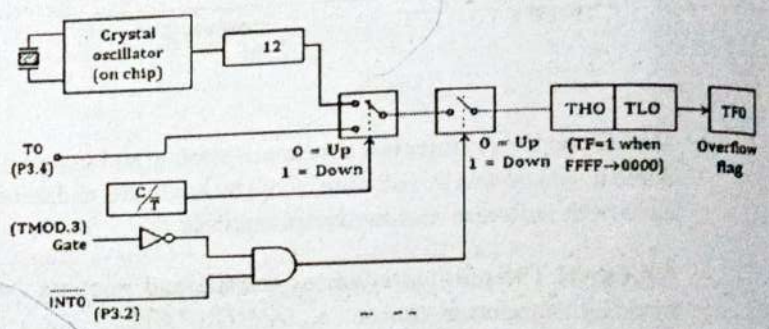
1. Since it is a 16-bit timer mode, 16-bit value from 0000H to FFFFH is loaded into the combined timer register TL and TH.

2. Timer must be started immediately after loading TL and TH registers with16-bit initial value. This can be done with the help of instruction "SETBTRx" (x=0or1).

3. Timer flag (TF) is set once the timer overflows from FFFF to 0000H. Timer may be stopped when TF =1 with the help of instruction CLRTRx(x=0or1). OR the timer must be reloaded to repeat the process.

Programming Mode1-Timer 0

The block diagram of timer 0-mode1 configuration is as shown in figure.



Steps to program Timer 0 in Mode1

1. Load 01 into TMOD register.
2. Load the registers TL0 and TH0 with initial count values.
3. Start the timer by setting TR0 = 1 (TCON.4).
4. Keep monitoring the timer flag (TF0), with the help of instruction JNBTF0, address.
5. Stop the timer when TF0=1.

6. Clear theTF0 flag for next round.

7. Repeat again.

Programming Mode1-Timer1

To program a timer in mode1, the same procedure as that of programming Timer 0 is adopted except that timer 1 uses registerTL1 and TH1, interrupt INT1, Timer overflow flag TF1 and T1 pin, gate (TMOD.7) pin.

12.1.6 Mode 2 (8-bitautoreload) Timer Programming

The characteristics of mode 2 operation are as follows:

1. It is an 8bit timer mode, it allows 8 bit values from 00 to FF to be loaded intoTHx register (x=0 or x=1).

2. After THx is loaded, 8051 copies it to TLx, then the timer must be started.This is done with the help of SETBTRx; (x=0 or 1).

3. The timer starts incrementing register TLx (x= 0 or1) from initial value to the FF. When it overflows (or rolls over) FF to 00, it sets TFx (x = 0 or 1) flag.

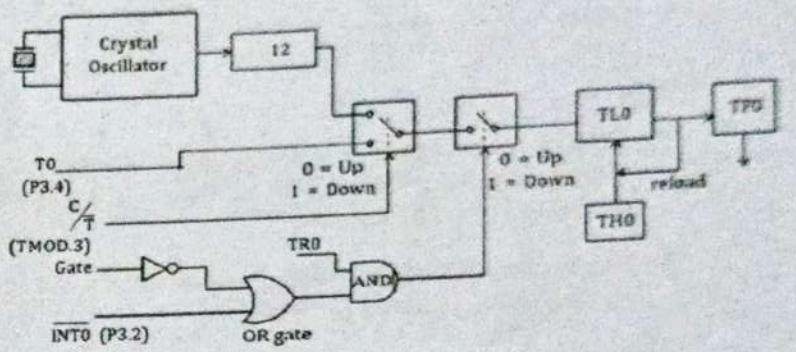
4. When TFx flag is set, the TLx register automatically loads the original value stored in THx.

5. To repeat the process use must simply clear theTFx flag. It is not required to reload the initial value toTHx register.

Note:*The mode is used to set the baudrate in serial communication.

Programming Mode 2-Timer 0

The block diagram of timer 0-mode 2 operation is given in figure



Steps to program Timer 0- Mode 2

1. Load 02 HintoTMOD register.

2. Load TH0 register with initial value.

3. Start timer (SETBTR 0).

4. Keep monitoring TF0 flag (JNB, TF0, label or target).

5. Clear TF0 flag (CLRTF0).

6. Repeat again from step4.

12.1.7 Programming Mode 2-Timer 1

To program a timer in mode 2, the same procedure as that of programming Timer 0 is adopted except that timer1 uses register TL1 and TH1, interrupt INT1, Timer overflow flag F1 and T1 pin, gate (TMOD.7) pin.

12.2 Time Delay Generation Calculation

The required time delay can be calculated using following formula

$$\text{Time delay}(T) = \frac{12}{\text{Crystal Frequency}} \times (\text{Final Value - Initial value} + 1)$$

Where,

- **Crystal frequency**→ It is frequency of crystal that is connected between XTAL1 & XTAL2 pins. It may vary from 10MHz to 40Mhz.
- **Final value**→This is the maximum possible value in a timer in a particular mode of operation.

The maximum value depends on the mode of the timer as shown in Table.

Mode	Timer Size	Maximum count value	
		Hexadecimal	Decimal
Mode 0	13-bit timer	1FFFH	8191
Mode 1	16-bit timer	FFFFH	65535
Mode 2	8-bit auto reload timer	FFH	255

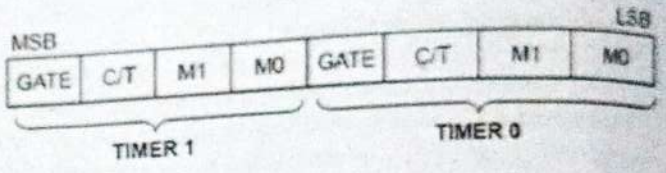
- **Initial Value**→It is the value that should be loaded into timer registers. From this initial value timer will count till final value.
- In mode 0 & 1- operation both TH & TL should be loaded where as in mode 2 operation only TH should be loaded with initial value.

Examples:

1. **Write an ALP and a8051 C program to generate a time delay of 50mS. Use timer1 in mode1 and a crystal frequency of 12MHz.**

Solution:

To program the timer 1 in mode 1, the TMOD bits should be configured as below :



TMOD = 00001 XXXXB = 10H

XTAL = 12 MHz,

Therefore Clock period T = (1/12)*12 = 1 us.

Time delay = 50ms

Look at the following steps.

(a) T = 50 ms, the period of delay.

(b) 50 ms / 1 us = 50000

(c) 65536 - 50000 = 15536 in decimal, and in hex it is 3CB0H

(d) TL1 = B0H and TH1 = 3CH (hex).

8051 ALP

```
MOV TMOD, #10H          ; Timer 1, mode 1
MOV TL1, #0B0H          ; TL1=B0, low byte of timer
MOV TH1, #3CH           ; TH1=3C, the high byte
SETB TR1                ; Start timer 1
BACK: JNB TF1, BACK     ; until timer rolls over
CLR TR1                 ; Stop the timer 1
CLR TF1
```

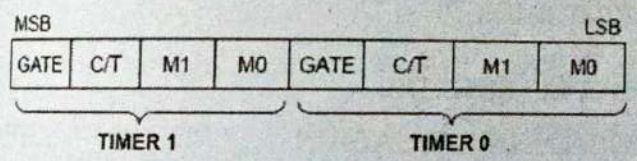
8051 C Code

```
#include<reg51.h>
void main()
{
  TMOD=0x10;
  TL1=0xB0;
  TH1=0x3C;
  TR1=1;
  while(TF1==0);
  TR1=0;
  TF1=0;
}
```

2. Write an ALP and a8051 C program to generate a square wave of 10KHz at P1.2. Use timer1 in mode 2 and a crystal frequency of 12MHz.

Solution:

To program the timer 1 in mode 2, the TMOD bits should be configured as below :



TMOD = 0010 XXXXB = 20H

XTAL = 12MHz,

Therefore Clock period T = (1/12)*12 = 1 us.

Delay frequency f = 10KHz

Time delay = 1/f = 1/10Khz = 100usec

For square ON time = OFF time = 50usec

Look at the following steps.

(a) ON time = OFF time, $T_{ON} = T_{OFF} = 50$ us

(b) 50 us / 1 us = 50us

(c) 256 - 50 = 206 in decimal, and in hex it is CEH.

(d) TH1 = CEH (hex).

8051 ALP

```
MOV TMOD,#20H        ;T1/8-bit/auto reload
MOV TH1,#0CEH        ;TH1 = CE
SETB TR1             ;start the timer 1
BACK:JNB TF1,BACK    ;till timer rolls over
CPL P1.2             ;P1.2 to high to low or low to high
CLR TF1              ;clear Timer 1 flag
SJMP BACK            ;mode 2 is auto-reload by repeating
END
```

8051 C Code

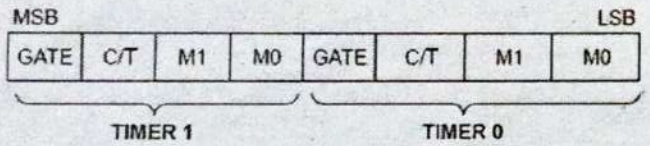
```
#include<reg51.h>
sbit x=P1^2;
void main(void)
{
    TMOD=0X20;
```

```
        TH0= 0XCE;
        TR1=1;
        while(TF1==0)
        {
        x=~x;
        TF1=0;
        }
}
```

3. Write an ALP and a8051 C program to generate a square wave of 25% duty cycle and an on period of 10mS at P1.4. Use timer0 in mode 1 and a crystal frequency of 12MHz.

Solution:

To program the timer 0 in mode 1, the TMOD bits should be configured as below :



TMOD = XXXX 0001B = 01H

XTAL = 12 MHz,

Therefore Clock period T = (1/12)*12 = 1 us.

Time delay = 10ms

For 25% duty cycle, ON Time = 2.5ms and OFF Time = 7.5ms. Note that OFF time is 3 times of ON time

Look at the following steps.

(a) T = 2.5 ms, the period of ON time delay.

(b) 2.5 ms / 1 us = 2500

(c) 65536 - 2500 = in decimal, and in hex it is F63CH

(d) TL0 = 3CH and TH0 = F6H (hex).

8051 ALP

```
ORG 0000H
MOV TMOD, #01H
BACK: MOV TL0, #3CH
MOV TH0, #0F6H
SETB P1.4
ACALL DELAY        ; ON time 2.5ms
CLR P1.4
ACALL DELAY        ; OFF time 2.5 * 3 = 7.5ms
ACALL DELAY
ACALL DELAY
SJMP BACK

DELAY: SETB TR0
AGAIN: JNB TF0, AGAIN
CLR TR0
CLR TF0
RET
END
```

8051 C Code

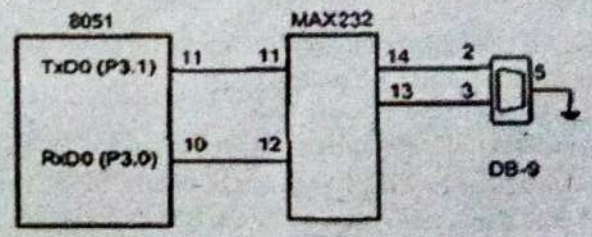
```
#include<reg51.h>
sbit x=P1^4;
void delay(void);// function prototype
void main(void)
{
TMOD=0x01;
while(1)
{
    TL0= 0x3C;
    TH0= 0xF6;
    x = 1;
    delay();    //25% on time
    x = 0;
    delay();
    delay();    //75% Off time
    delay();
}}
void delay()
{
TR0=1;
While (TF0==0);
    TR0=0;
    TF0=0;
}
```


12.3 8051- Serial Communication

- Whenever data is required to send across long distance, parallel interfacing becomes expensive and does not work. When the data is to be sent over a short distance, say from CPU to Memory, serial interfacing is preferred.

- One of the 8051's many powerful feature is its integrated UART, (Universal Asynchronous Receiver Transmitter) otherwise known as a serial port. The fact that the 8051 has an integrated serial port means that you may very easily read and write values to the serial port. Serial interfacing is of two types Synchronous and Asynchronous types. In synchronous type, data is always associated with clock. In asynchronous type only data is sent serially. The communication using COM port of computer is an example of asynchronous communication.

- The 8051 has two pins that are used specifically for transferring and receiving data serially. These two pins are called TXD and RXD and are part of the Port-3 group (Port-3.0 and Port-3.1). Pin 11 of the 8051 is assigned to TXD and pin 10 is designated as RXD. These pins are TTL compatible. But in this communication data is sent using RS-232 level. These voltages are converted into TTL levels and vice versa using a line driver IC-MAX232.



- One advantage of the MAX232 chip is that it uses a +5 V

power source which, is the same as the source voltage for the 8051. In other words, with a single +5 V power supply we can power both the 8051 and MAX232, with no need for the dual power supplies that are common in many older systems.

➢ In Asynchronous serial communication, the data is sent serially one bit at a time. The number of bits transferred per second is called as Baud rate. In 8051 different baud rates can be generated using Timer1 in mode 2(auto-reload).

There are SFR's (Special Function Registers)associated with serial data transfer, namely:

a) SCON - Serial Control Register whose address is 98h.

b) SBUF - Serial Buffer Register whose address is 99h.

c) TMOD - Timer Mode Control Register whose address is 89h.

d) TH1 & TL1 - Timer1 lower & higher byte.

e) TCON - Timer control Register

12.3.1 SBUF Register (Serial Buffer Register)

SBUF is an 8-bit register used solely for serial communication in the 8051.It is not bit addressable.

➢ For a byte of data to be transferred via the TxD line, it must be placed in the SBUF register. Similarly, SBUF holds the byte of data when it is received by the 8051's RxD line. SBUF can be accessed like any other register in the 8051.

➢ Look at the following examples of how this register is

accessed:

```
MOV SBUF,#'D'      ;load SBUF=44H, ASCII for 'D'
MOV SBUF,A         ;copy accumulator into SBUF
MOV A,SBUF         ;copy SBUF into accumulator
```

➢ The moment a byte is written into SBUF, it is framed with the start and stop bits and transferred serially via the TxD pin. Similarly, when the bits are received serially via RxD, the 8051 deframes it by eliminating the stop and start bits, making a byte out of the data received, and then placing it in the SBUF.

12.3.2 SCON Register (Serial Control Register)

➢ The SCON register is an 8-bit register used to program the start bit, stop bit, and data bits of data framing, among other things. The following describes various bits of the SCON register.

D7						D0
SM0	SM1	SM2	REN	TB8	RB8	TI RI

➢ **SM0, SM1 (serial port mode)**

Specify framing format, how to calculate baud as per the below table-

SM0	SM1	MODE	DESCRIPTION	BAUD RATE
0	0	0	8-bitShift Register	1/12 the quartz frequency
0	1	1	8-bit UART	Determined by the timer 1
1	0	2	9-bit UART	1/32 the quartz frequency (1/64 the quartz frequency)
1	1	3	9-bit UART	Determined by the timer 1

- **SM2:** Multi processor communication bit

  SM2 = 0: single processor selection

  SM2 = 1: multiprocessor communication

- **REN:** Receive Enable

  REN = 1: The 8051 will accept incoming data from serial port

  REN = 0: the receiver is disabled

- **TB8:** Transmitter bit 8 (D8). Since all registers are 8-bit wide, this bit solves the problem of transmitting the 9th bit in modes 2 and 3.

  TB8 = 1: Transmit a logic 1 in the 9th bit

  TB8 = 0: Transmit a logic 0 in the 9th bit

- **RB8:** Receive bit 8 (D8). This bit solves the problem of receiving the 9th bit in modes 2 and 3.

  RB8 = 1: If 9th bit received is a logic 1

  RB8 = 0: If 9th bit received is a logic 0

- **TI:** Transmit interrupt flag bit. When 8051 finishes the transfer of the 8-bit character, it set TI to 1 to indicate that it is ready to transfer the next character. The TI is always raised at the beginning of the stop bit

  TI = 1: Ready to transfer next character

  TI = 0: Busy in transferring the present character

- **RI:** Receive interrupt flag bit: When 8051 receives a character, the UART removes start bit and stop bit, then the UART puts the 8-bit character in SBUF

RI = 1: Indicate that a new byte is ready to be picked up in SBUF

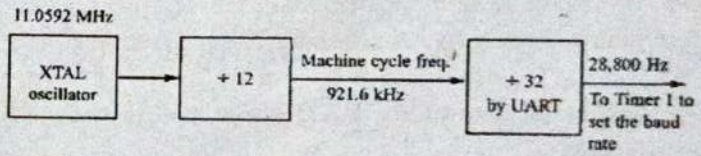
RI = 0: SBUF is not ready to pick up new byte

Baud Rate

Baud rate represents the speed of operation. Baud rate is used to measure the number of bits transmitted per second. Hence its unit is bps (bits per second). The baud rates supported by 8051 are :

- 9600
- 4800
- 2400
- 2000

Baud rate in 8051 can be set via timer 1 in operating in mode 2 (8-bit auto-reload). When used for serial port, the frequency of timer tick is determined by (XTAL/12)/32 and only 1 bit is transmitted for each timer period. The below figure gives the clear picture of setting the baud rate through timer 1.



The below table shows the values to be loaded in TH1 to set the timer for the respective baud rate

Baudrate	TH1 (Decimal)	TH1 (Hex)
9600	-3	FD
4800	-6	FA
2400	-12	F4
1200	-24	E8

Note: XTAL= 11.0592MHz

12.3.3 Working of Serial Port

The working of serial port as follows :

- Timer 1 in mode 2 (8-bit auto-reload) is used to set the baud rate in TMOD register to generate the clock for the serial port
- The serial mode, start, stop bits etc. are loaded in SCON register
- To transmit the data serially, the data is written in SBUF register
- SUBF is framed with start bit and stop bits and then the data is transmitted one bit at a time on the TXD pin at the initialized baud rate
- Monitor TI (Transmit Interrupt Flag). Once the transmission is complete, the TI bit is set indicating the user can write next data into SBUF
- To receive the data serially, enable REN (Receive Enable) and required baud rate is set using Timer 1, mode 2
- The serial data on the RXD pin is loaded into the SBUF register
- Monitor RI (Receive Interrupt flag). Once the reception is complete, the RI bit is set indicating to read the next value in SBUF

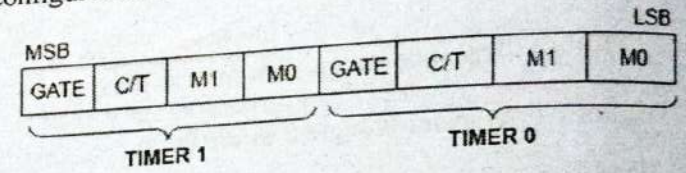
Lab Programs:

1. Write and execute an embedded C program to generate a square wave on P1.2 using timer 0 in mode 1 to generate delay.

Solution:

As frequency of square wave to be generated is not given, so we write the code for generating a square wave of 2Khz frequency.

To program the timer 0 in mode 1, the TMOD bits should be configured as below :



TMOD = XXXX 0001B = 01H

XTAL = 11.0592MHz, (crystal frequency in microcontroller kit)

Therefore Clock period T = (1/11.0592)*12 = 1.085 us.

Delay frequency f = 2KHz

Time delay = 1/f = 1/2Khz = 500usec

For square ON time = OFF time = 250usec

Look at the following steps.

(a) T = 250us, the period of ON time delay.

(b) 250us / 1.085 us = 230 (round off)

(c) 65536 - 230 = 65306 in decimal, and in hex it is FF1AH.

(d) TL0 = 1AH and TH0 = FFH (hex).

8051 C Code

```
#include<reg51.h>
sbit x=P1^2;
void delay(void);// function prototype
void main(void)
{
TMOD=0x01;
while(1)
{
    TL0= 0x1A;
    TH0= 0xFF;
    x = ~x;
    delay();
}}
void delay()
{
TR0=1;
While (TF0==0);
    TR0=0;
    TF0=0;
}
```

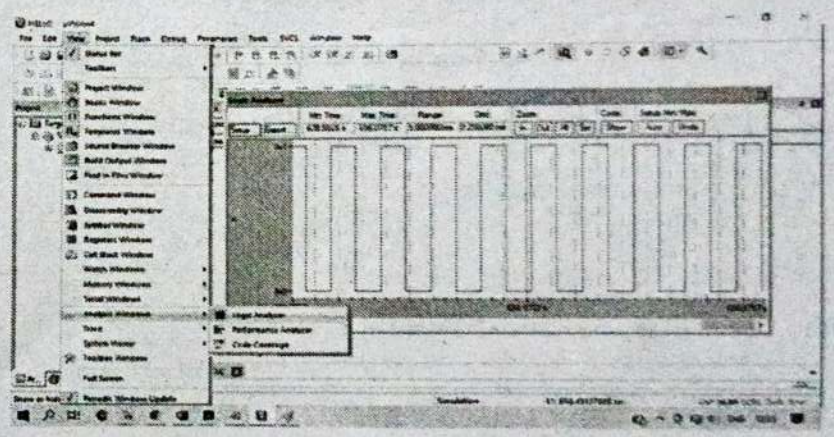

2. Observe the square wave of the above program on CRO by downloading the program to the microcontroller kit.

Note: The procedure for downloading the above program into the kit depends on the type of kit, manufacturer. So please ask the vendor to provide the same.

Meanwhile, you can observe the output in logic analyzer window which is built in to the keil IDE itself.

- To see the logic analyzer window. In debugging mode → click on View → Analysis window → Logic analyzer → Click on Set up

- A window appears where you are supposed to insert or set a variable name whose output waveform you want to see (in our case x). Then hit enter



Review Questions :

1. Explain the bit structure of TMOD register.
2. Explain the bit structure of SCON register.
3. Discuss the operation of timer 0 in mode 1.
4. Discuss the operation of timer 1 in mode 2.
5. Explain the uses of TCON register.
6. Write an ALP and a 8051 C program to generate a square wave of 10KHz at P1.2 . Use timer 1 in mode 2 and a crystal frequency of 10MHz.
7. Write an ALP and a 8051 C program to generate a square wave of 25% duty cycle and an on period of 10mS at P1.4 . Use timer 0 in mode 1 and a crystal frequency of 12MHz.
8. Summarize the scheme of interfacing RS232 to 8051.
9. Discuss the significance of SI and RI flags.
10. Discuss the significance of SBUF register.
11. Summarize the working of serial port.

Week 13

Interfacing the 8051

Interfacing the 8051

13.1 Seven Segment Display

13.2 Interfacing ADC 0804 to 8051

13.2.1 Interfacing DAC 0808 to 8051

13.3 Interfacing Stepper Motor to 8051

13.3.1 Relay and DC Motor

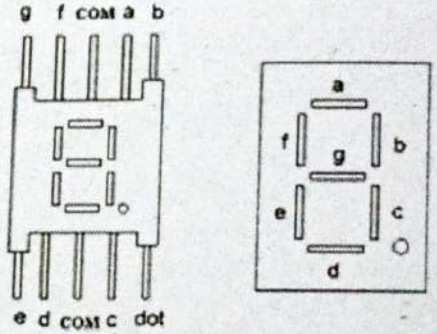
* Lab Programs

* Review Questions

13.1 Seven Segment Display

Seven segment displays are important display units in Electronics and widely used to display numbers from 0 to 9. It can also display some character alphabets like A,B,C,H,F,E etc.

Before interfacing, we should learn about 7 segment displays. It's the simplest unit to display numbers and characters. It just consists of 8 LEDs, each LED used to illuminate one segment of unit and the 8th LED used to illuminate DOT in 7 segment display.



Seven-Segment Display

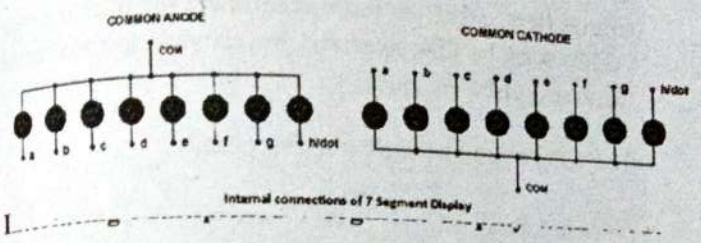
We can refer each segment as a LINE, as we can see there are 7 lines in the unit, which are used to display a number/ character. We can refer each line/segment "a,b,c,d,e,f,g" and for dot character we will use "h". There are 10 pins, in which 8 pins are used to refer a,b,c,d,e,f,g and h/dp, the two middle pins are common anode/cathode of all he LEDs. These common anode/ cathodes are internally shorted so we need to connect only one COM pin.

There are two types of 7 segment displays: Common Anode and Common Cathode:

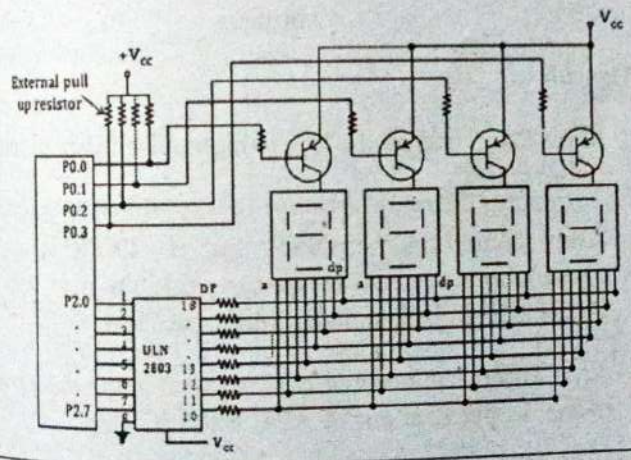
Common Anode: In this all the positive terminals (Anodes) of

all the 8 LEDs are connected together, named as COM. And all the negative thermals are left alone.

Common Cathode: In this all the Negative terminals (cathode) of all the 8 LEDs are connected together (see diagram below), named as COM. And all the positive terminals are left alone.



Multiplexed four numbers of 7 segment displays can be interfaced to the 8051. One port line (say for example port 2) are used for sending data and another 4bits of a port P0.0, P0.1, P0.2, P0.3) are used to select one out of four. The figure shows the interfacing circuit of multiplexed 7segment displays to 8051. The segment inputs are connected in parallel to all four displays through a driver **IC ULN 2803**. The select lines are connected to each display individually through driver transistor.



When the base of the transistor is low then VCC is connected to common anode terminal of display. The segment lines are made low to glow the segments.

- ✓ The following steps are performed to display 4 digits on 7 segments.
- ✓ Send the 1st seven segment code to port 2.
- ✓ Select display 1 by making P0.1 = 0.
- ✓ Call delay.
- ✓ Send the 2nd seven segment code to port 2.
- ✓ Select the second 7 segment display by setting P0.1 = 1.
- ✓ Call delay again.
- ✓ Repeat the above procedure to display 3rd and 4th digits.
- ✓ Refresh the multiplexed seven segment display by repeating the above steps.

The code for seven segment display are formed as indicated below :

	P2.7	P2.6	P2.5	P2.4	P2.3	P2.2	P2.1	P2.0	
Number	a	b	c	d	e	f	g	DP	code
0	1	1	1	1	1	1	0	0	FCh
1	0	1	1	0	0	0	0	0	60h
2	1	1	0	1	1	0	1	0	DAh
3	1	1	1	1	0	0	1	0	F2h
4	0	1	1	0	0	1	1	0	66h
5	1	0	1	1	0	1	1	0	B6h
6	1	0	1	1	1	1	1	0	BEh
7	1	1	1	0	0	0	0	0	E0h
8	1	1	1	1	1	1	1	0	FEh
9	1	1	1	0	0	1	1	0	E6h
DP	0	0	0	0	0	0	0	1	01h

Note that the IC ULN 2803 is a driver and inverts the incoming port bits.

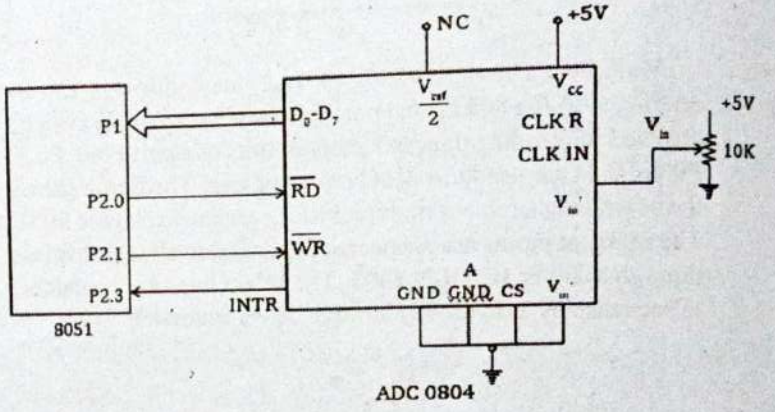
a) Assembly Language Program

	ORG 0400H	; create look up table for 7 segment codes at say 0400H
	DB60H,0DAH,0F2H,66H	; 7segment codes for digits 1,2,3,4 respectively
	ORG 0000H	; Initialize starting address of Program.
REFRESH:	MOV R0,#00H	;
	MOV DPTR,#0400H	; DPTR points the segment codes
	MOV R3,#0FEh	; Select Ist display
	MOV P0,R3	; Send to P0.
BACK:	MOV A,R0	; R0 holds the index to table of codes
	MOV A,@A+DPTR	; Copy Ist 7segmentcodeintoACC
	MOV P2,A	; Send it toP2
	MOV A,R3	; Select display
	MOV P0,A	; Send it P0
	SETB C	; Set the carry to rotate into ACC
	RLC A	; Left shift to select next display
	MOV R3,A	; Store in R3
	ACALL DELAY	; Call delay
	INC R0	; Select next code
	CJNE R0,#04,BACK	; Repeat 4 times
	SJMP REFRESH	; Repeat and display new 4 digit no
DELAY:	MOV R4, #30H	; Delay subroutine
LOOP:	DJNZ R4, LOOP	
	RET	
	END	

13.2 Interfacing ADC 0804 to 8051

Analog to digital converter is most common used in data acquisition system. To measure and control the physical parameters like temperature, pressure, light etc. ADC's are used to convert analog signal into digital. Then the digital signals applied to any digital device (PC or microcontroller) for further processing. ADC' scan be characterized by conversion time, resolution, accuracy etc.

The figure shows the interfacing circuit.



Algorithm:

1. Make CS=0 and send a low to high pulse to pin to start the conversion.
2. Keep monitoring the **INTR** pin. If **INTR** is low the conversion is finished & we can go to the next step & If **INTR** is high, keep polling until it goes low.
3. After **INTR** has become low, we send a low to high pulse to the to get data out of ADC 0804 IC chip.

ALP:

```
        ORG 0000H
        MOV P1,#0FFH; make port1 as input port
        SETB P2.3     ; make P2.3 as input
Repeat: CLR P2.1      ;
        SETB P2.1     ; send start of conversion
WAIT:   JB P2.3,WAIT  ; wait till the conversion is over
        CLR P2.0      ; RD=0
        MOV A,P1      ; Read
        SETB P2.0     ; RD=1
        SJMP Repeat   ; Repeat
```

C Program:

```
#include<reg51.h>
void main(void)
{
P1 = 0xFF              //make P1 as input
P2^3 = 1;              //make P2.3 as input
unsigned int x;
while(1)
{
P2^1 = 0;              //WR = low to high
```



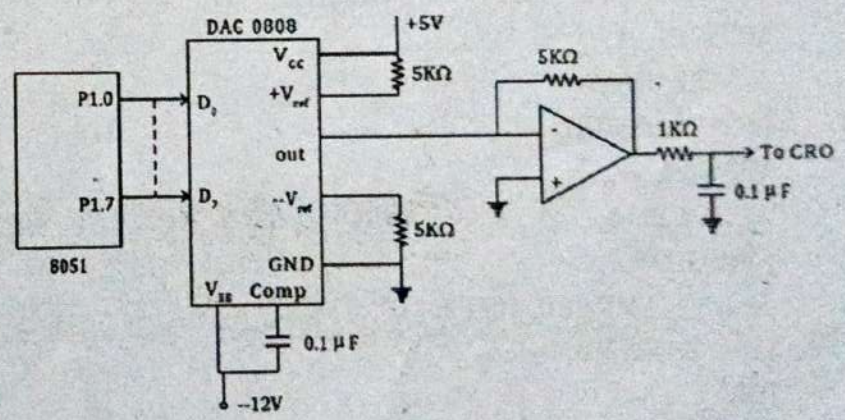
```
P2^1 = 1;
while (P2^3 == 1)      //wait till P2.3 becomes 0
P2^0 = 0;
x = P1;                //Read digital value
P2^0 = 1;              //Read next
}}
```

13.2.1 Interfacing DAC 0808 to 8051

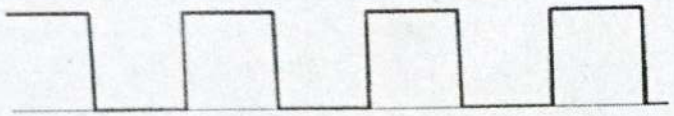
Digital to Analog converter IC 0808 is used widely to convert digital pulses into analog signals. The DAC0808 uses R-2R ladder network type. The DAC's are characterized with resolution accuracy. The resolution is defined as the smallest step size of the analog voltage generated by the DAC. The DAC0808 is a 8 bit DAC

i.e., 28= 256 levels of the output are available.

The figure shows the interfacing circuit



Example 1. Interface a DAC to 8051 and generate a 1 kHz square wave of 50% duty cycle.



- A square wave has only two amplitudes, a minimum (0V) and a maximum (10V) corresponding to P1 values of 00H and FFH respectively.
- 1 kHz frequency corresponds to a time period T = 1 ms

  A 50 % Duty cycle Square wave has

  T_{on} = 0.5 ms& T_{off}= 0.5 ms

Solution :

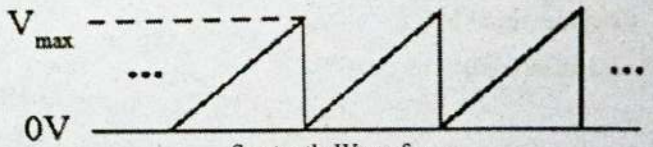
a)Assembly Language Program

```
ORG 0000H
TOP :MOV A, #00H
MOV P1, A
ACALL delay
MOV A, #0FFH
MOV P1, A
ACALL delay
SJMP TOP
delay :   MOV R0, #0FFH
here : DJNZ R0, here
RET
END
```


b) C Program

```
#include<reg51.h>
void main(void)
{
unsigned int i;
while(1)
{
P1 = 0x00;
for(i=0; i<637; i++)// 637=1275/2 (1ms/2=0.5ms)
P1 = 0xFF;
for(i=0; i<637; i++)
}}
```

Example 2. Interface a DAC 0808 to 8051 at port P1 and write an 8051 program to generate a sawtooth waveform



Sawtooth Wave form

It has an amplitude which is increases linearly with time to a maximum value, the maximum value in the above interfacing circuit is generated when D7 - D0 = FF H.

From an initial value 00H, the value at P1 is continuously incremented to FF and the next increment makes FF to 0 and hence a saw tooth wave form is generated.

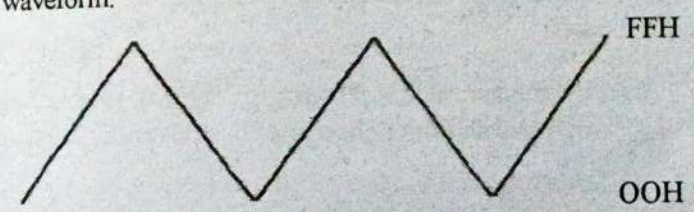
Assembly Language Program

```
ORG 0000H
MOV A,#00H          ;Initial value
AGAIN: MOV P1,A     ;Output on p1
  INC A             ;Keep Incrementing
  SJMP AGAIN
  END
```

C Program

```
#include <reg51.h>
void main()
{
unsigned char i;
while(1)            // repeat continuously
{
for (i=0; i<255; i++)
P1=i;
}}
```

Example 3. Interface a DAC to 8051 and generate a triangular waveform.



To generate a triangular waveform the input to the DAC should increment from 00H to FFH and then decrement from **FFH to 00H.**

a) Assembly Language Program

```
START: MOV A, #00H        ; initial value
UP:  MOV P1, A
  INC A                   ; increment till FFH
  CJNE A,#0FFH,UP
DOWN: MOV P1, A
  DEC A                   ; decrement till 00H
  CJNE A,#00H,DOWN
SJMP START
END
```

b) C Program

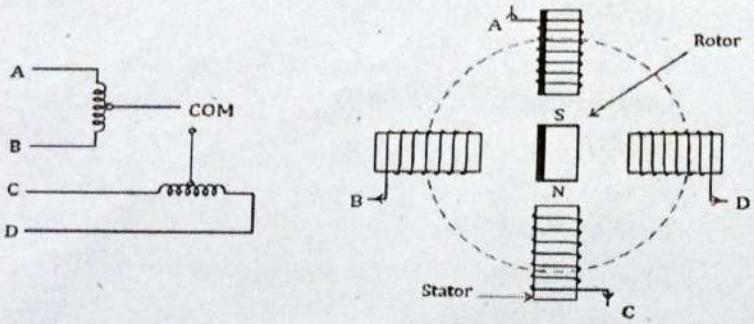
```
#include <reg51.h>
void main()
{
unsigned char i;
while(1)
{
for (i=0; i<0xFF; i++)
P1=i;                 // increasing ramp
for (i=0xFF; i>0; i--)
P1=i;                 // decreasing ramp
}}
```

13.3 Interfacing Stepper Motor to 8051

Stepper motor is a digital actuator whose input is programmed digital data and output is in the form of discrete angular rotation. Stepper motor translates electrical pulses into mechanical movement. A stepper motor shaft rotates in fixed increments, so that the shaft position can be controlled precisely. The stepper motors are used for position control applications such as in dot matrix printers, disk drives,robotics etc.

The moving part of a stepper motor is called rotor that is typically made from permanent magnet material. The rotor is surrounded by a stator which is an electro magnet made up of 2 windings. The 4 windings are paired with a centre tap common as shown in figure.

The two windings have a centre tap, com; which is connected to +5V. The centre tap allows us to change the direction of rotation of motor. When stator A is Energised, and hence it becomes electro magnet to attract the rotor. The stators or Energised in sequence causes rotor to rotate in steps.

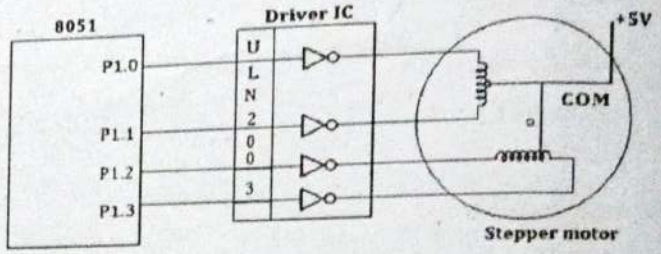


For interfacing stepper motor with 8051 four windings of stepper motor are connected to 4 bits of a port 1. A driver IC ULN 2003 is required to drive the motor.This IC is connected

in between the port bits and the stepper motor windings. The figure shows the interfacing circuit. The normal 4 step sequence is also given in table

	Windings				P1.0 to P1.3 value
	A	B	C	D	
Step 1	1	0	0	1	09
Step 2	1	1	0	0	0C
Step 3	0	1	1	0	06
Step 4	0	0	1	1	03

Clockwise ↓ ↑ Anticlockwise



Programming Steps

1. The port 1 is connected to the 4 windings of stepper motor, normal 4 step sequences is loaded into port 1. Any one of the sequence 99h, cch, 66h, 33h can be taken as initial value.
2. Rotating the initial value either left or right will lead to the next step.
3. If the initial value is shifted left, the motor rotates in anticlock wise and shifting right rotates in clockwise direction.

ALP:

```
ORG 0000H
MOV A,#99H          // Load initial value
NEXT: MOV P1,A      //send it to port 1
RR A                //Rotate clock wise
ACALL DELAY         //call delay
SJMP NEXT           //repeat
DELAY: MOV R3,#0FFH
BACK:DJNZ R3,BACK
RET
END
```

C program:

```
#include<reg51.h>
void delay(unsigned int);
void main()
{
unsigned char i,val;
while(1)
{
val=0X99;
for(i=0;i<4;i++)
{
P1=val;
val=val>>1;
```



```
delay(500);
}}}
void delay(unsigned int t)
{
unsigned int i, j;
for(i=0; i<t; i++)
for(j=0; j<1275; j++);
}
```

13.3.1 Relay and DC Motor

Introduction to DC motor

- ✓ DC motors are widely used at places where constant speed is required. They are very easy to interface and program.
- ✓ Besides they can easily switch between clockwise and anticlockwise motion. Their typical usage is in fans, blowers, conveyor belts, pumps, lifts etc.
- ✓ A DC motor simply requires two pins to control its movements.
- ✓ By sending a 01 or 10 on these pins we can move the motor clockwise or anticlockwise. Sending a 11 or a 00 on the two pins will stop the motor.

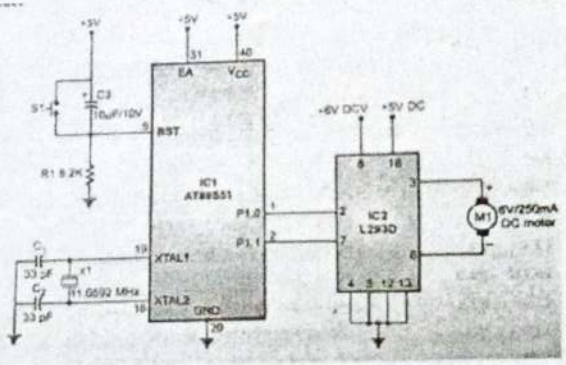
Need of relay -L293D:

- ✓ DC motor are controlled by our μc 8051
- ✓ However sources/sinks current around 15mA at voltage 5V
- ✓ A DC motor however requires much higher current and

voltages to drive real world appliances. For this purpose we use relays.

✓ A typical relay like L293D can provide a max current around 600mA at voltage 4.5V to 36V. We could also use a relay driver like ULN2003.

Interface diagram:



Working:

✓ Pins P1.0 and P1.1 are used to control the motor via the relay L293D.

✓ By sending values on these pins we can start/ stop the motor in clockwise and anticlockwise direction. The motor is connected at the output of L293D.

✓ The logic table for controlling the motor movement is given below ;

P1.0	P1.1	Movement	
1	0	Clockwise	In some books you will find two to be opposite. That is also correct
0	1	Anticlockwise	
0	0	Stop	
1	1	Stop	

Program to operate a DC motor with Relay:

```
ORG 0000H
BACK:SETB P1.0
CLR P1.1          ; clockwise
ACALL DELAY       ; call to a subroutine "delay"
CLR P1.0
CLR P1.1          ; stop
ACALL DELAY       ; 1 second delay
CLR P1.0
SETB P1.1         ; Anti clockwise
ACALL DELAY       ; 1 second delay
CLR P1.0
CLR P1.1          ; stop
ACALL DELAY
SJMP BACK         ; Repeat
DELAY:MOV R3,#0FFH
BACK1:DJNZ R3,BACK1
RET
HERE:SJMPHERE
END
```

Lab Programs :

1. Program to control direction and speed of a stepper motor

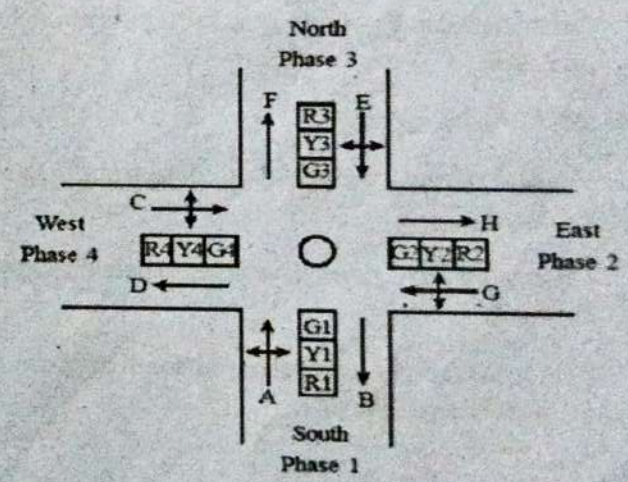
```
#include<reg51.h>
```

```
sbit direction=P1^6;        //Toggle switch to control direction of
rotation
sbit speed=P1^7; //push button switch to control speed of rotation
void delay();               // Function prototype
void main()
{
while(1)
{
if(direction==0)   //anticlockwise rotation
{
P2=0x06;
delay();
P2=0x0c;
delay();
P2=0x09;
delay();
P2=0x03;
delay();
}
else               //clock wise rotation
{
P2=0x03;
delay();
P2=0x09;
delay();
P2=0x0c;
delay();
P2=0x06;
delay();
```



```
}}
void delay()
{
unsigned int i;
if (speed==0)      // rotate with more speed
{
for(i=0;i<300;i++);
}
else               //rotate with less speed
{
for(i=0;i<30000;i++);
}}
```

2. Program to control traffic lights:



Four way traffic flow layout

The diagram shown above illustrates the four way traffic flow layout. The traffic flow can be classified into the following four phases

- **PHASE 1**

  Phase 1 permit the vehicles from A to F, D and H roads.

  Green signal in A permits vehicles to pass through north, east and west

- **PHASE 2**

  Phase 2 permits the vehicle to pass from G to D, B and F roads.

  Green signal in G permits vehicles to pass through west, north and south.

- **PHASE 3**

  Phase 3 permits the vehicle to pass from E to B, H and D roads.

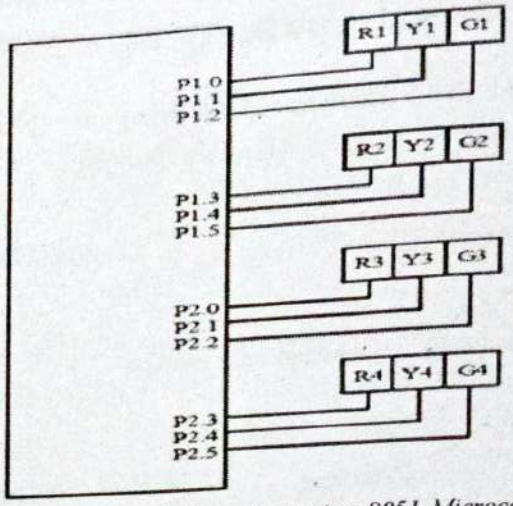
  Green signal in E permits vehicles to pass through south, east and west.

- **PHASE 4**

  Phase 4 permits the vehicle to pass from C to F, H and B roads.

  Green signal in C permits vehicles to pass through north, east and west.

  The cycle repeats again from phase 1 to phase 4 and thus the traffic is regulated



Four way traffic light system using 8051 Microcontoller

C Code:

```
#include<reg51.h>
sbit R1=P1^0;
sbit Y1=P1^1;
sbit G1=P1^2;
sbit R2=P1^3;
sbit Y2=P1^4;
sbit G2=P1^5;
sbit R3=P2^0;
sbit Y3=P2^1;
sbit G3=P2^2;
sbit R4=P2^3;
sbit Y4=P2^4;
sbit G4=P2^5;
```

```
void msdelay(unsigned int t)
{
int i,j;
int k=100*t;
for(i=0;i<k;i++)
for(j=0;j<1275;j++);
}
void clear()
{
R1=0,R2=0,R3=0,R4=0,Y1=0,Y2=0;
Y3=0,Y4=0,G1=0,G2=0,G3=0,G4=0;
}
void phase1()
{
R1=1,R2=1,G3=1,R4=1;
msdelay(25);
Y3=1;
Y1=1;
msdelay(5);
}
void phase2()
{
G4=1,R2=1,R3=1,R1=1;
msdelay(25);
Y2=1;
```



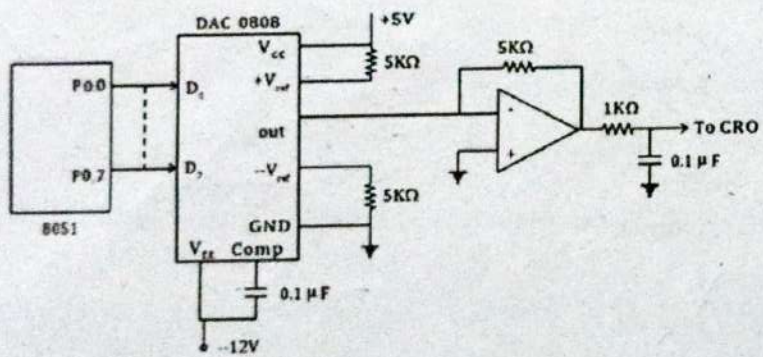
```
Y4=1;
msdelay(5);
}
void phase3()
{
G1=1,R2=1,R3=1,R4=1;
msdelay(25);
Y1=1;
Y3=1;
msdelay(5);
}
void phase4()
{
G2=1,R1=1,R3=1,R4=1;
msdelay(25);
Y2=1;
Y4=1;
msdelay(5);
}

void main()
{
P1=0x00;
P2=0x00;
while(1)
```

```
{
phase1();
clear();
phase2();
clear();
phase3();
clear();
phase4();
clear();
}}
```

3. Program to generate sine/ rectangular / triangular waveforms.

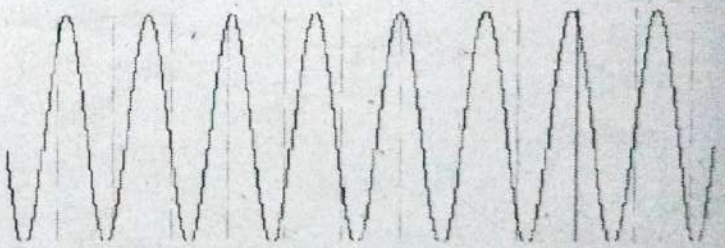
Interfacing diagram:



Sine waveform:

```
#include<reg51.h>
void main ()
{
unsigned char value[36]={128,150,172,192,210,226,239,248,
254,255, 254, 248,239,226,210, 192,
172,150,128, 106,84,64, 46, 30, 17, 8,2,0,2, 8,17,30,46,64,84,106};
unsigned char x;
while(1)
{
for(x=0;x<36;x++)
{
P0=value[x];
}}}
```

Output waveform:

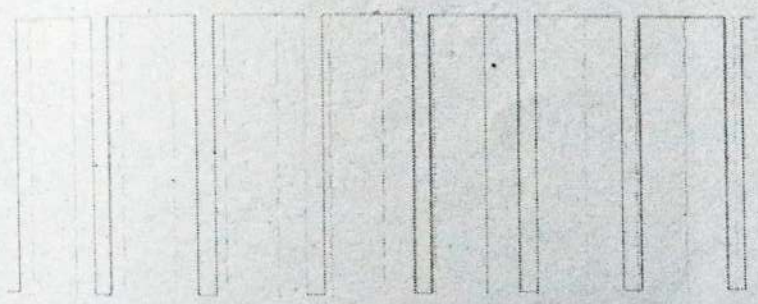


Rectangular waveform:

```
#include<reg51.h>
void delay(unsigned int); //function prototype
```

```
void main ()
{
while(1)
{
P0=0xFF;
delay(5); //delay of 5ms
P0=0x00;
delay(1); //delay of 1ms
}
}
void delay(unsigned int itime)
{
unsigned int i,j;
for(i=0;i<itime;i++)
for(j=0;j<1275;j++);
}
```

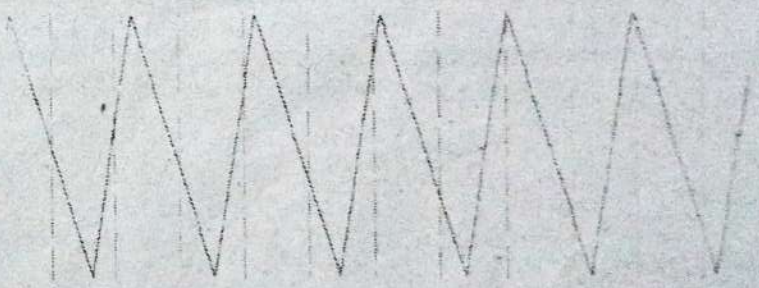
Output waveform:



Triangular waveform:

```
#include<reg51.h>
void main()
{
unsigned char x;
while(1)
{
for(x=0;x<255;x++) //positive ramp
P0=x;
for(x=255;x>0;x--) // negative ramp
P0=x;
}}
```

Output waveform:



Review Questions :

1. Write the schematic for interfacing a 4-digit multiplexed 7 segment display to 8051.
2. Write the schematic for interfacing ADC 0804 to 8051.
3. Sketch the schematic for interfacing DC motor to 8051.
4. Sketch the schematic for interfacing a stepper motor to 8051.
5. Write the schematic, algorithm and a program to interface 4 digit multiplexed 7 segment display to 8051 and display 2022.
6. Write the schematic, algorithm and a program to interface a DAC to 8051 and to generate a triangular waveform.
7. Write the schematic, algorithm and a program to interface a ADC 0804 to 8051.
8. Discuss the need of L293D in interfacing DC motor with 8051.
9. Write the schematic, algorithm and a program to interface a DAC to 8051 and to generate atrapezoidal waveform.