# PRACTICAL WORK BOOK

**For Academic Session _____**

**Semester _____**

## DIGITAL LOGIC DESIGN

**(TC-201)**

**For**

**SE (TC)**

**Name:** _____

**Roll Number:** _____

**Batch:** _____

**Department:** _____

**Year:** _____


**Department of Electronic Engineering**

**NED University of Engineering & Technology, Karachi**

# LABORATORY WORK BOOK

## For The Course

# TC-201 DIGITAL LOGIC DESIGN

**Prepared By:**
**Mr. Rizwan Aslam Butt (Lecturer)**

**Reviewed By:**
**Dr. Irfan Ahmed (Associate Professor)**

**Approved By:**
**The Board of Studies of Department of Electronic Engineering**

# INTRODUCTION

Digital Logic Design Practical Workbook covers those practical that are very knowledgeable and quite beneficial in grasping the core objective of the subject. These practical solidify the theoretical and practical concepts that are very essential for the engineering students.

This work book comprise of practical covering the topics of Digital Logic Design that are arranged on modern concepts. Above all this workbook contains a relevant theory about the Lab session.

# CONTENTS

| | | | | |
|---|---|---|---|---|
| 13 | | (i) To understand 4 to 1 MUX working principle<br>(ii) To understand ModelSim Software for Development of Verilog HDL<br>(iii) To implement and Test 4 to 1 MUX on Verilog HDL by<br>• Gate Level Modeling<br>• Data Flow Modeling<br>• Behavioral Modeling | | |
| 14 | | (i) To understand Quartus-II Software for Development of Verilog HDL codes.<br>(ii) To implement and test Verilog HDL code of a given function.<br>(iii) To test the given function program on ALTERA DE2 board. | | |

# LAB NO.1

## OBJECTIVE: To become familiar with basic logic gates and their functions.

## BRIEF OVERVIEW:

A logic gate is an elementary building block of a digital circuit. Most logic gates have two inputs and one output. At any given moment, every terminal is in one of the two binary conditions *low* (0) or *high* (1), represented by different voltage levels. The logic state of a terminal can, and generally does, change often, as the circuit processes data. In most logic gates, the low state is approximately zero volts (0 V), while the high state is approximately five volts positive (+5 V).

There are seven basic logic gates: AND, OR, XOR, NOT, NAND, NOR, and XNOR.

### AND GATE:

The *AND gate* is so named because, if 0 is called "false" and 1 is called "true," the gate acts in the same way as the logical "and" operator. The following illustration and table show the circuit symbol and logic combinations for an AND gate. (In the symbol, the input terminals are at left and the output terminal is at right.) The output is "true" when both inputs are "true." Otherwise, the output is "false."

**AND gate**

| Input 1 | Input 2 | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

### OR GATE:

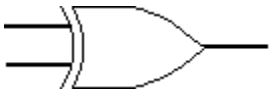
The *OR gate* gets its name from the fact that it behaves after the fashion of the logical inclusive "or." The output is "true" if either or both of the inputs are "true." If both inputs are "false," then the output is "false."

**OR gate**

| Input 1 | Input 2 | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

**XOR GATE:**

The *XOR* (*exclusive-OR* ) *gate* acts in the same way as the logical "either/or." The output is "true" if either, but not both, of the inputs are "true." The output is "false" if both inputs are "false" or if both inputs are "true." Another way of looking at this circuit is to observe that the output is 1 if the inputs are different, but 0 if the inputs are the same.



**XOR gate**

| Input 1 | Input 2 | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

## NOT GATE:

A logical *inverter*, sometimes called a *NOT gate* to differentiate it from other types of electronic inverter devices, has only one input. It reverses the logic state.

**Inverter or NOT gate**

| Input | Output |
|---|---|
| 1 | 0 |
| 0 | 1 |

## NAND GATE:

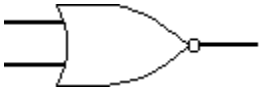
The *NAND gate* operates as an AND gate followed by a NOT gate. It acts in the manner of the logical operation "and" followed by negation. The output is "false" if both inputs are "true." Otherwise, the output is "true."

**NAND gate**

| Input 1 | Input 2 | Output |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**NOR GATE:**

The *NOR gate* is a combination OR gate followed by an inverter. Its output is "true" if both inputs are "false." Otherwise, the output is "false."

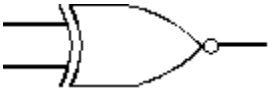


**NOR gate**

| Input 1 | Input 2 | Output |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

**XNOR GATE:**

The *XNOR (exclusive-NOR) gate* is a combination XOR gate followed by an inverter. Its output is "true" if the inputs are the same and "false" if the inputs are different.



**XNOR gate**

| Input 1 | Input 2 | Output |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Using combinations of logic gates, complex operations can be performed. In theory, there is no limit to the number of gates that can be arrayed together in a single device. But in practice, there is a limit to the number of gates that can be packed into a given physical space. Arrays of logic gates are found in digital integrated circuits (ICs). As IC technology advances, the required physical volume for each individual logic gate decreases and digital devices of the same or smaller size become capable of performing ever-more-complicated operations at ever-increasing speeds.

## Common Gate ICs:

| Part number | Description |
|---|---|
| 7400 | quad 2-input NAND gate |
| 7402 | quad 2-input NOR gate |
| 7408 | quad 2-input AND gate |
| 7410 | triple 3-input NAND gate |
| 7432 | quad 2-input OR gate |
| 7486 | quad 2-input XOR gate |

## PROCEDURE:

1) Power up the 2-input AND , OR and NOT TTl ICs on a bread board.
2) Apply inputs using push-to-on/off switches and observe the output via LEDs.
3) Fill the Table provided in the observation area.

## OBSERVATIONS:

| A | B | A.B | A+B | A' |
|---|---|---|---|---|
| 0 | 0 | | | |
| 0 | 1 | | | |
| 1 | 0 | | | |

| | | | | |
|---|---|---|---|---|
| 1 | 1 | | | |

**RESULT:**

# LAB NO.2

## OBJECTIVE: To implement a Half adder circuit.

## BRIEF OVERVIEW:

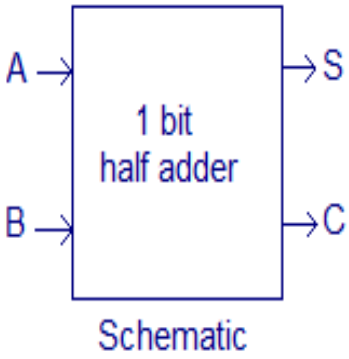
To understand what is a half adder you need to know what is an adder first. Adder circuit is a combinational digital circuit that is used for adding two numbers. A typical adder circuit produces a sum bit (denoted by S) and a carry bit (denoted by C) as the output. Typically adders are realized for adding binary numbers but they can be also realized for adding other formats like BCD (binary coded decimal, XS-3 etc. Besides addition, adder circuits can be used for a lot of other applications in digital electronics like address decoding, table index calculation etc. Adder circuits are of two types: Half adder ad Full adder.

Half adder is a combinational arithmetic circuit that adds two numbers and produces a sum bit (S) and carry bit (C) as the output. If A and B are the input bits, then sum bit (S) is the X-OR of A and B and the carry bit (C) will be the AND of A and B. From this it is clear that a half adder circuit can be easily constructed using one X-OR gate and one AND gate. Half adder is the simplest of all adder circuit, but it has a major disadvantage. The half adder can add only two input bits (A and B) and has nothing to do with the carry if there is any in the input. So if the input to a half adder have a carry, then it will be neglected it and adds only the A and B bits. That means the binary addition process is not complete and that's why it is called a half adder. The truth table, schematic representation and XOR//AND realization of a half adder are shown in the figure below.

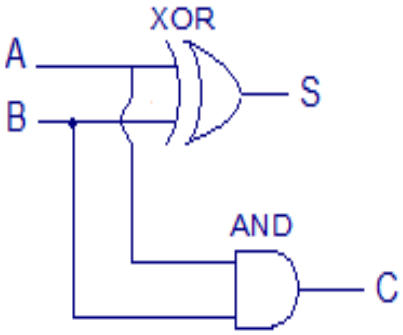
## TRUTH TABLE:

| Inputs | | Outputs | |
|---|---|---|---|
| A | B | S | C |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 |

Truth table




Schematic




Realization

## OBSERVATIONS:

| A | B | Sum | Carry Out |
|---|---|---|---|
| 0 | 0 | | |
| 0 | 1 | | |
| 1 | 0 | | |
| 1 | 1 | | |

## RESULT:

The half adder circuit was implemented on a bread board using ICs.

# LAB NO.3

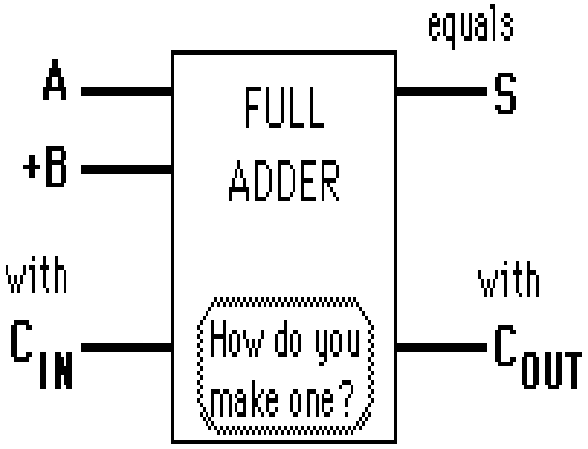
**Objective: To implement a Full adder circuit.**

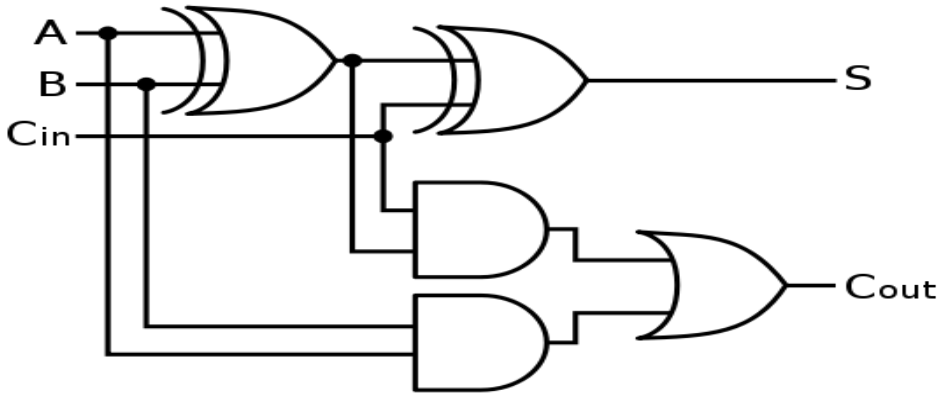
## BRIEF OVERVIEW:

A full adder adds binary numbers and accounts for values carried in as well as out. A one-bit full adder adds three one-bit numbers, often written as A, B, and $C_{in}$;A and B are the operands, and $C_{in}$ is a bit carried in from the next less significant stage. The full-adder is usually a component in a cascade of adders, which add 8, 16, 32, etc. binary numbers. The circuit produces a two-bit output, output carry and sum typically represented by the signals $C_{out}$ and S.

## TRUTH TABLE:

| Input bit for number A | Input bit for number B | Carry bit input $C_{IN}$ | Sum bit output S | Carry bit output $C_{OUT}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |






## OBSERVATIONS:

The required outputs observed as described in the truth table for sum and carry out are as follows.

| A | B | Carry In | Sum | Carry Out |
|---|---|---|---|---|
| 0 | 0 | 0 | | |
| 0 | 0 | 1 | | |
| 0 | 1 | 0 | | |
| 0 | 1 | 1 | | |
| 1 | 0 | 0 | | |
| 1 | 0 | 1 | | |
| 1 | 1 | 0 | | |
| 1 | 1 | 1 | | |

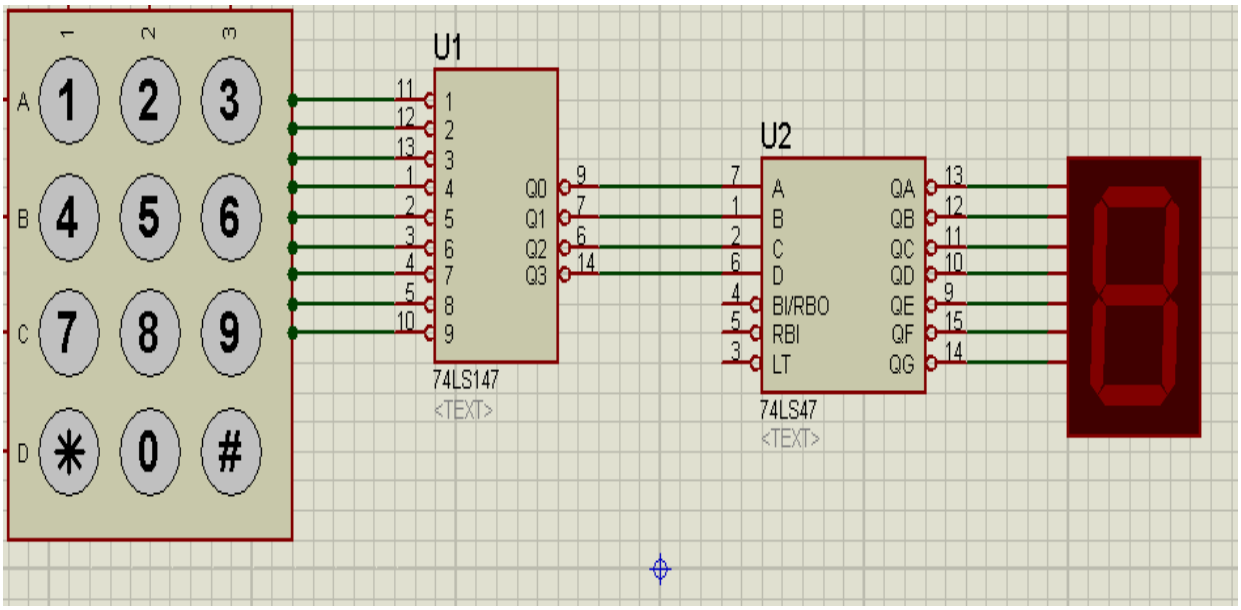
## RESULT:

The Full Adder circuit was implemented using 74LS83 discrete IC and the outputs of sum and carry out were observed on Led's.

# LAB NO.4

**OBJECTIVE: To learn the operation of Encoder and Decoder by implementing a counter using seven segment display and keypad.**

## CIRCUIT DIAGRAM:




## BRIEF OVERVIEW:

An encoder converts a non binary code to binary code like decimal numbers to binary. These are often needed as the human machine interface like a keypad generates such non-binary codes. But to apply digital operations on these codes it is often needed to convert these codes to binary language.

Decoders on the other hand are the inverse function of the encoders. We need these again to represent the processed information in an human understandable form like on seven segment displays.

## OBSERVATIONS:

Press all buttons on the keyboard and record the outputs observed on the seven segment display.

**RESULT:**

The above circuit was implemented using 74LS147 encoder and a 74LS47 BCD to seven segment decoder.

# LAB NO. 5

**OBJECTIVE : To learn programming with 8051 microcontrollers using keil environment.**

## BRIEF OVERVIEW:

Microcontroller is a programmable digital logic device that has on-board micro-processor, RAM, ROM and many other peripheral functions available on a single ship. Famous general propose microcontroller families are 8051, PIC and AVR microcontrollers. These microcontrollers can be programmed in Assembly, C and Basic languages via specific development environment. The most famous environment for 8051 family program development is Keil uVision. It has provision of programming both in C and Assembly language.

## Task#1:

**To write a code on Keil using 8051 microcontroller for blinking led's and simulate it on proteus.**

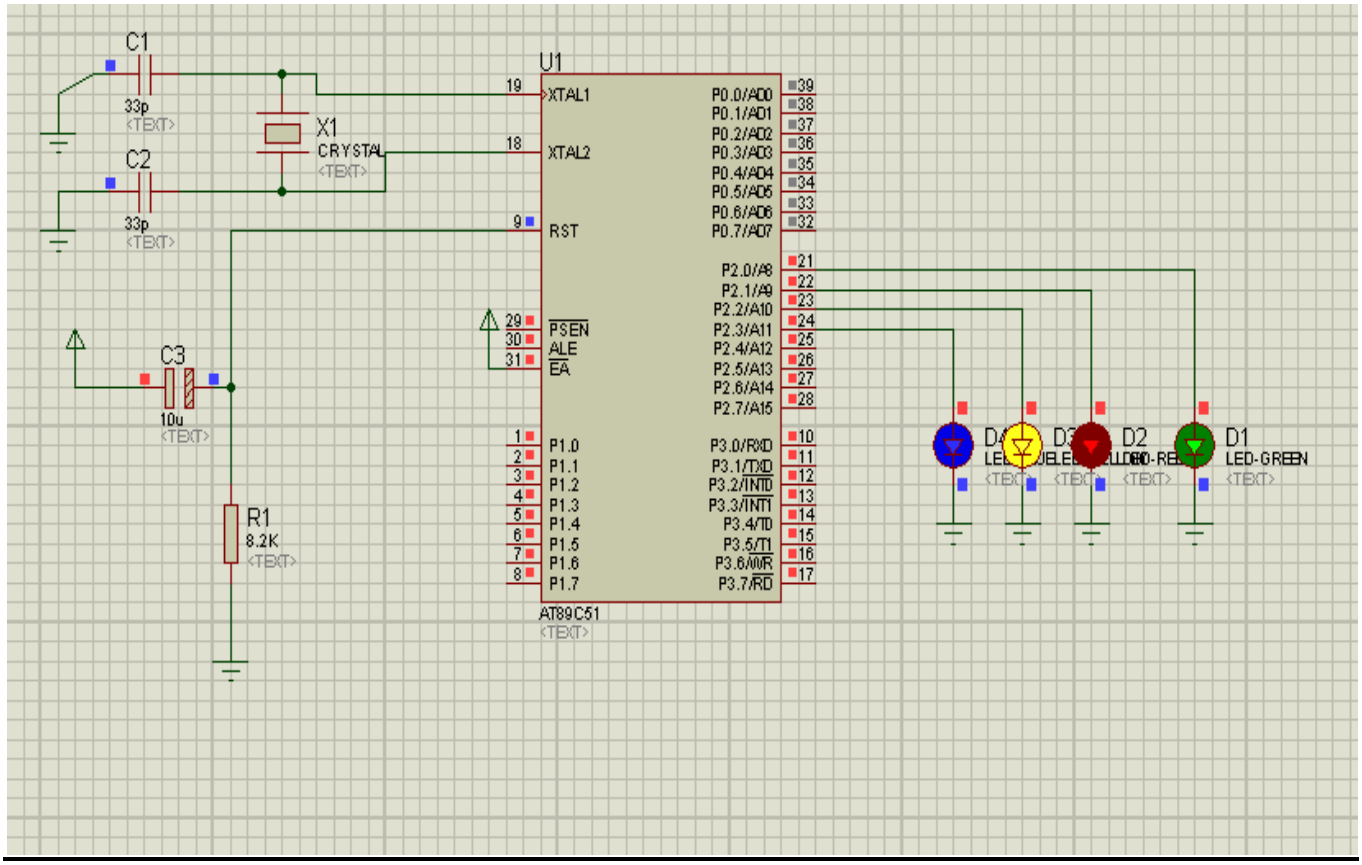
## Program:

```
/*To blink Leds using Microcontroller*/

#include <REG51.h>

void delay(unsigned int sec)

{

Unsigned int i,j;

for (i=0;i<sec;i++)

for(j=0;j<1500;j++);

}

void main()

{
```

```
int i=0;
P2=0x00;
while(1)
{
P2=0xFF;
delay(100);
P2=0x00;
delay(100);
}
}
```

**Simulation in Proteus:**

Implement the following circuit in proteus and burn the above code in it.




## Task#2:

**Write an 8051 based code to design a counting system for a security gate. The count should be displayed on a seven segment.**

## Program:

```
/*Design a security gate to increment the count of persons entering uptil 9
via switch using 8051 microcontroller*/

#include <REG51.h>

void delay(unsigned int sec)

{

Unsigned int i,j;

for(i=0;i<sec;i++)
```

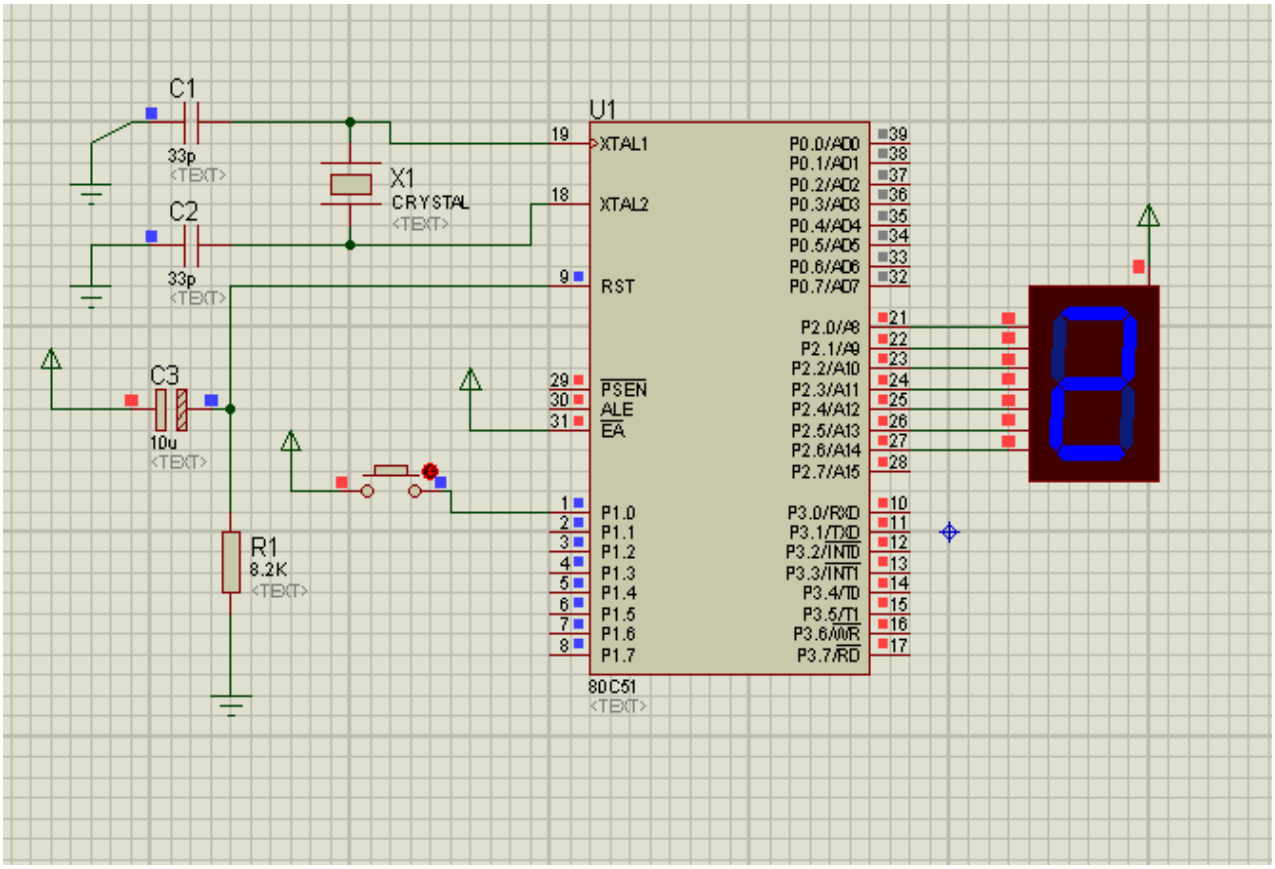
```
for(j=0;j<1500;j++);
}
void main()
{
int i=0;
P1=0x00; //Declaring input port
while(1)
{
if(P1^0!=0)  //Condition to check whether the switch is pressed or not
{i++;
}
if(i>9)  //To reset the value of count to 0 if increment increases 9
{
i=0;
}
if(i==0)
{P2=0xC0;
delay(200);
}
else if(i==1)
{P2=0xF9;
delay(200);
}
```

```
else if(i==2)
{
P2=0xA4;
delay(200);
}
else if(i==3)
{
P2=0xB0;
delay(200);
}
else if(i==4)
{
P2=0x99;
delay(200);
}
else if(i==5)
{
P2=0x92;
delay(200);
}
else if(i==6)
{
P2=0x82;
```

```
delay(200);
}
else if(i==7)
{
P2=0xF8;
delay(200);
}
else if(i==8)
{
P2=0x80;
delay(200);
}
else if(i==9)
{
P2=0x90;
delay(200);
}
}
}
```

**Simulation in Proteus:**

Implement the following circuit in proteus and burn the above code in it.




## RESULT:

Write your achievements and experiences here.

# Lab No.6

**OBJECTIVE: To blink LEDs connected to port A with a delay of 500 millisec using PIC 16F877A.**

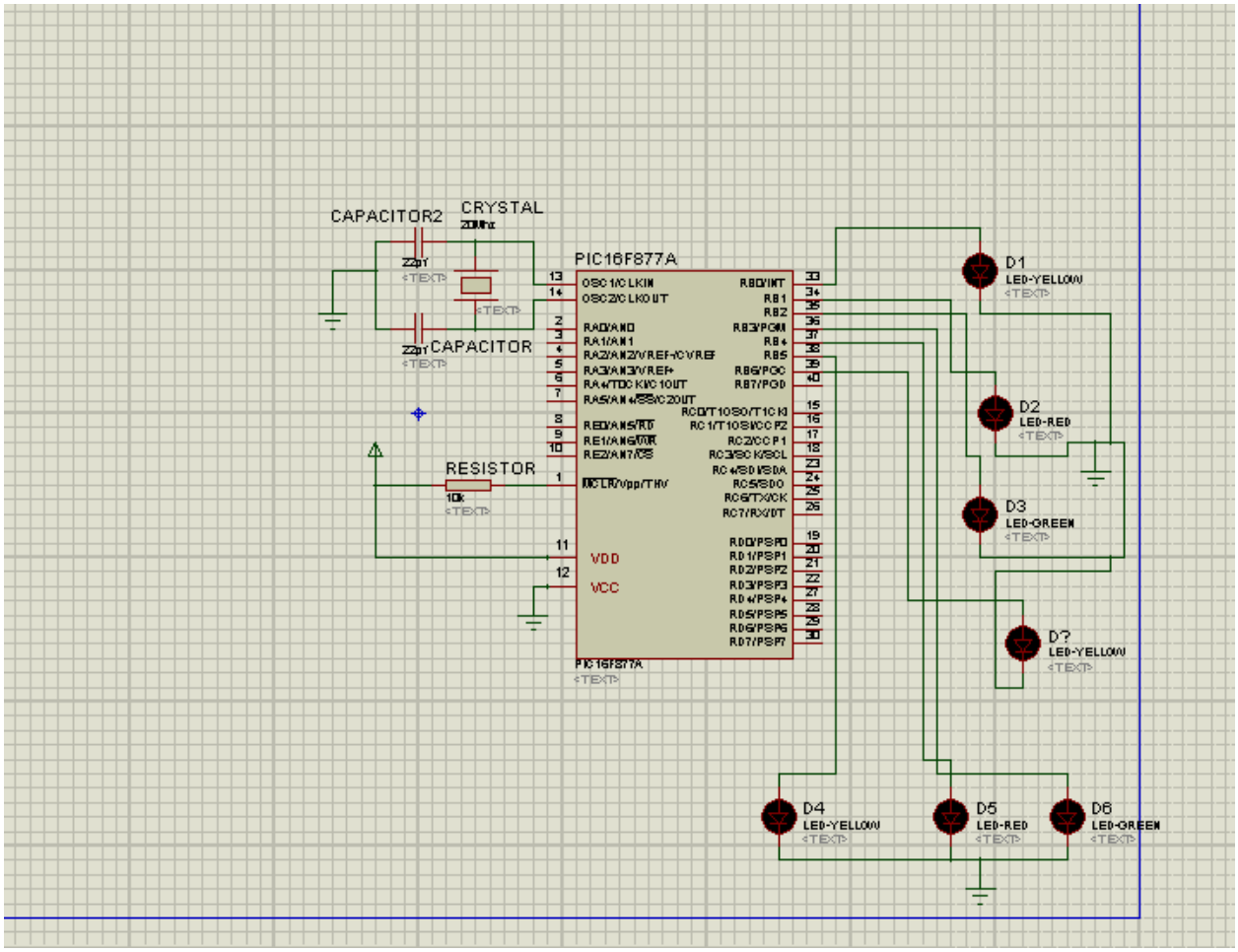
## BRIEF OVERVIEW:

Microcontroller is a programmable digital logic device that has on-boad micro-processor, RAM, ROM and many other peripheral functions available on a single ship. Famous general propose microcontroller families are 8051 , PIC and AVR microcontrollers.

## Program:

```
void main()
 {
 TRISB = 0 ;
 TRISC=0;
for(;;)
 {
 PORTB = 0xff ;
 PORTC = 0xff ;
Delay_ms(500) ;
 PORTB = 0 ;
 PORTC = 0 ;
Delay_ms(500) ;
 }
 }
```

## Proteus Simulation:




## RESULT:

Write your experiences and achievements here.

# LAB NO.7

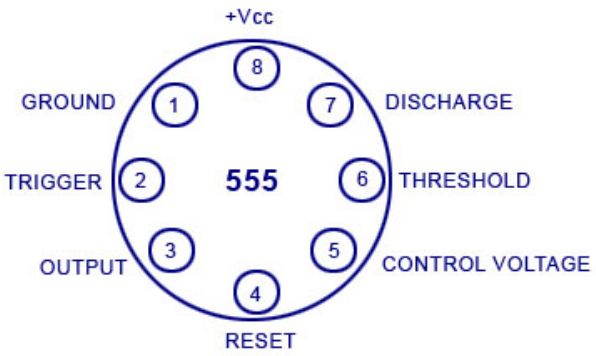
**OBJECTIVE: To design an Astable multi vibrator using 555 timer and to understand Flip Flop operation.**

## BRIEF OVERVIEW:

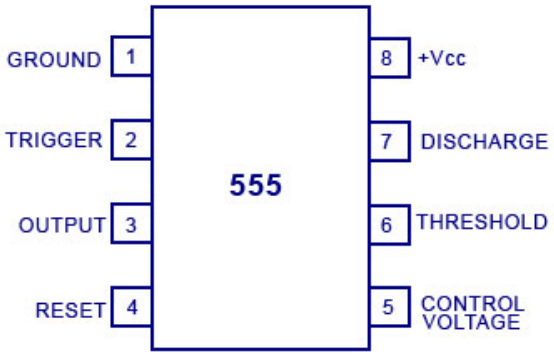
555 IC is a monolithic timing circuit that can produce accurate and highly stable time delays or oscillation. Like other commonly used op-amps, this IC is also very much reliable, easy to use and cheaper in cost. It has a variety of applications including **monostable** and **astable multivibrators**, **dc-dc converters**, digital logic probes, **waveform generators**, analog frequency meters and tachometers, **temperature measurement** and control devices, **voltage regulators** etc. The timer basically operates in one of the two modes either as a monostable (one-shot) multivibrator or as an astable (free-running) multivibrator. The **SE 555** is designed for the operating temperature range from – 55°C to 125° while the **NE 555** operates over a temperature range of 0° to 70°C.

## IC PIN CONFIGURATION:

555 TIMER IC




Top View Of Metal Can Package



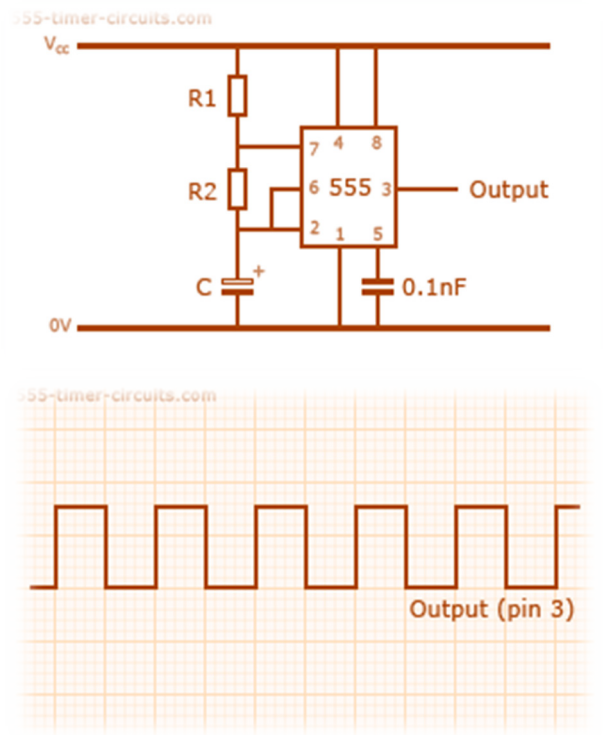

8-Pin DIP

## WORKING MODES:

The 555 has three main operating modes, Monostable, Astable, and Bistable. Each mode represents a different type of circuit that has a particular output.

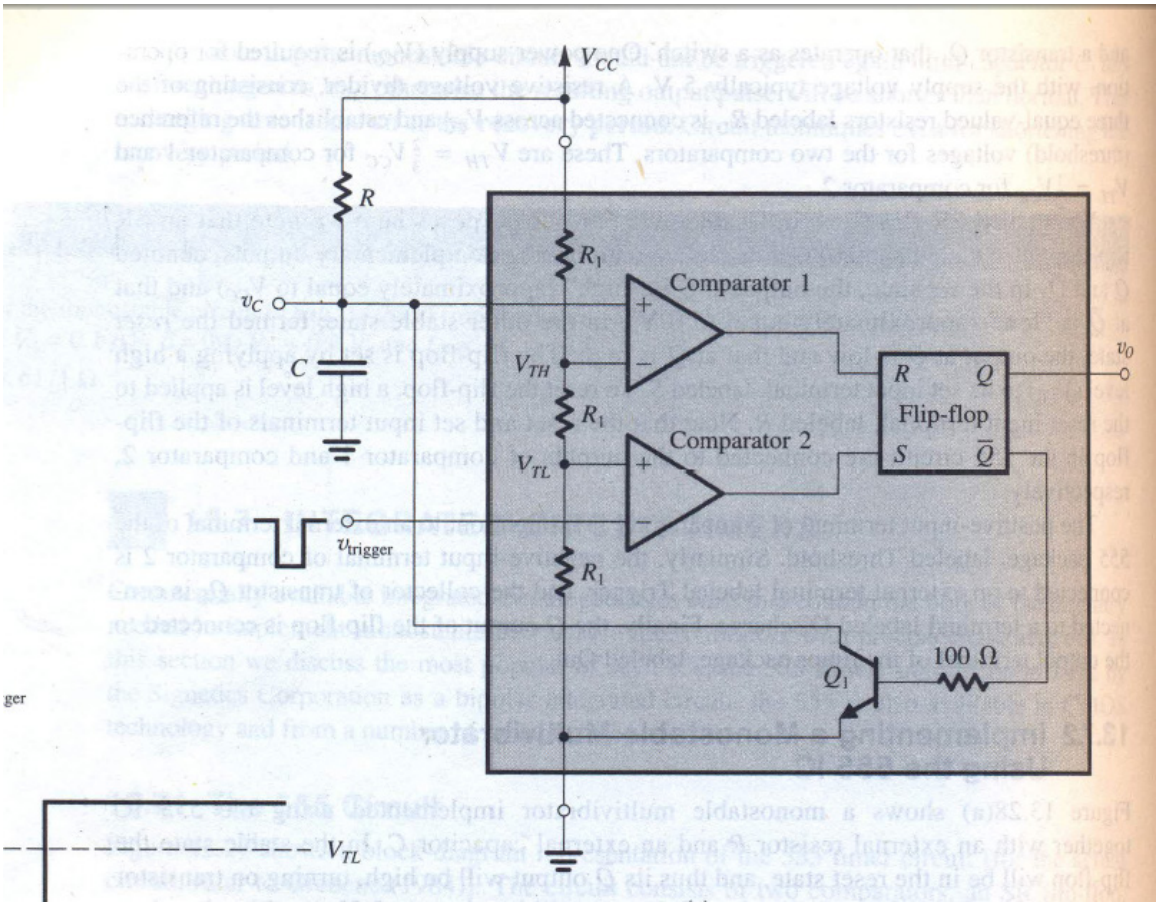
**Astable mode** :

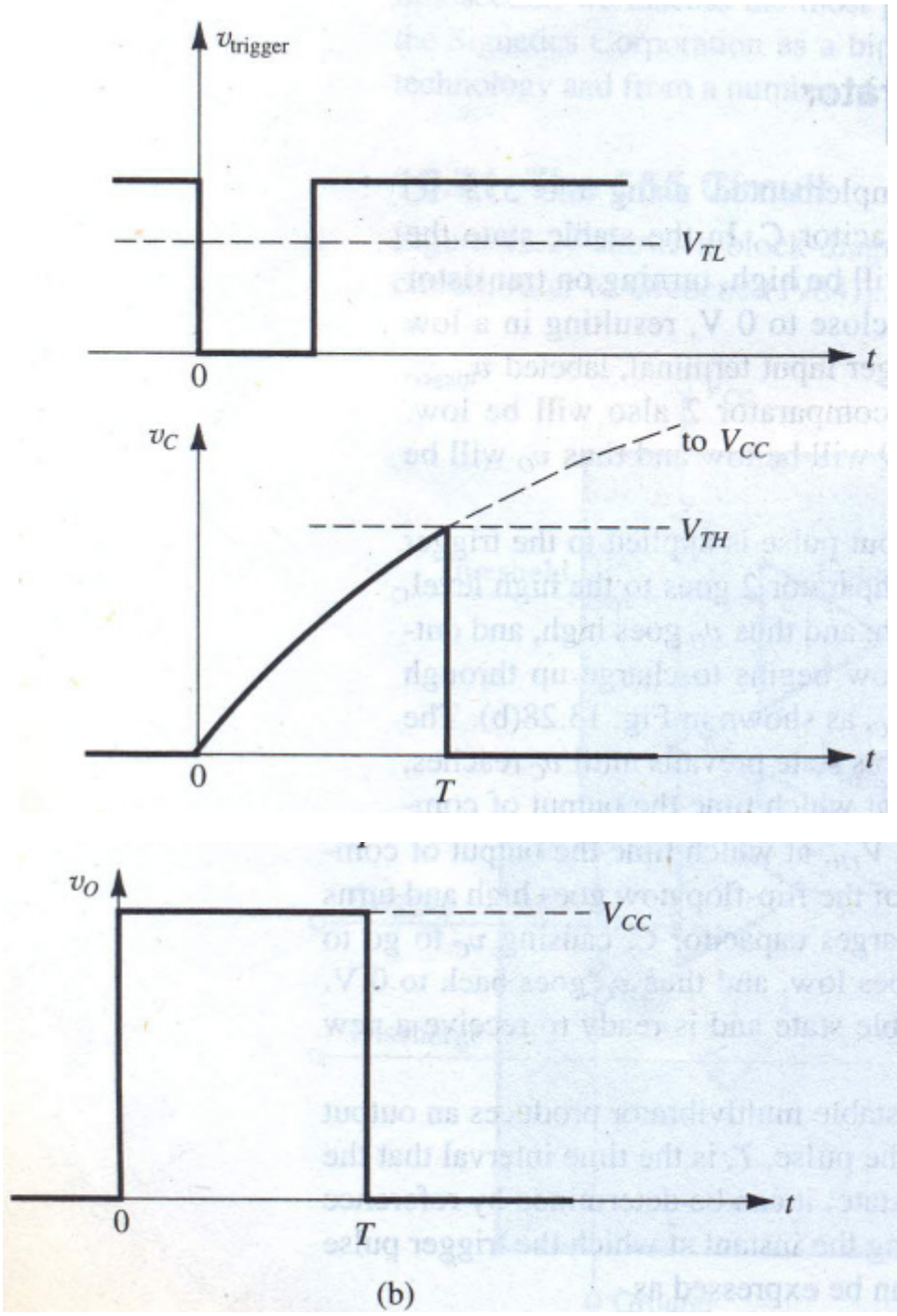
An Astable Circuit has no stable state - hence the name "astable". The output continually switches state between high and low without any intervention from the user, called a 'square' wave. This type of circuit could be used to give a mechanism intermittent motion by switching a motor on and off at regular intervals. It can also be used to flash lamps and LEDs, and is useful as a 'clock' pulse for other digital ICs and circuits.




**Monostable mode :**

A Monostable Circuit produces one pulse of a set length in response to a trigger input such as a push button. The output of the circuit stays in the low state until there is a trigger input, hence the name "monostable" meaning "one stable state". his type of circuit is ideal for use in a "push to operate" system for a model displayed at exhibitions. A visitor can push a button to start a model's mechanism moving, and the mechanism will automatically switch off after a set time.

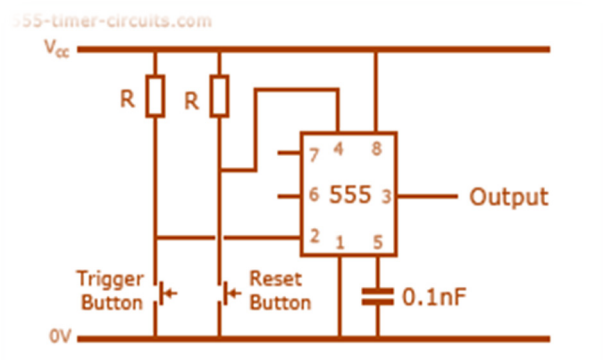

$V_{CC}$
R
$R_1$
Comparator 1
$v_C$
C
$V_{TH}$
$R_1$
Comparator 2
$V_{TL}$
R Q
Flip-flop
S $\bar{Q}$
$v_0$
$v_{trigger}$
$R_1$
$Q_1$
100 Ω
ger
$V_{TL}$

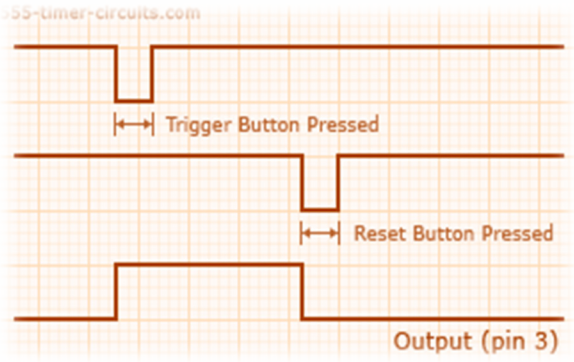



**Bistable Mode (or Schmitt Trigger)**:

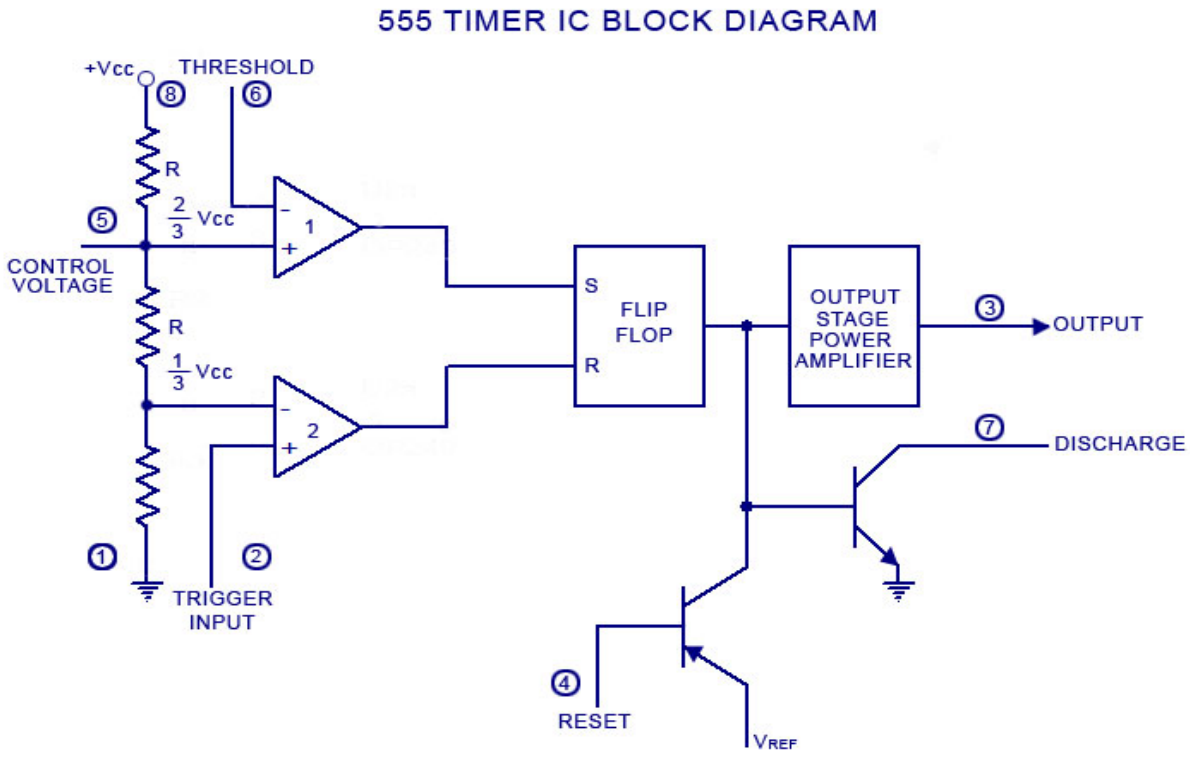
A Bistable Mode or what is sometimes called a Schmitt Trigger, has two stable states, high and low. Taking the Trigger input low makes the output of the circuit go into the high state. Taking the Reset input low makes the output of the circuit go into the low state. This type of circuit is ideal for use

in an automated model railway system where the train is required to run back and forth over the same piece of track. A push button (or reed switch with a magnet on the underside of the train) would be placed at each end of the track so that when one is hit by the train, it will either trigger or reset the bistable. The output of the 555 would control a DPDT relay which would be wired as a reversing switch to reverse the direction of current to the track, thereby reversing the direction of the train.



## FLIP FLOP OPERATION:




The block diagram of a **555 timer** is shown in the above figure. A 555 timer has two comparators, which are basically 2 op-amps), an R-S flip-flop, two transistors and a resistive network.

- Resistive network consists of three equal resistors and acts as a voltage divider.
- Comparator 1 compares threshold voltage with a reference voltage + 2/3 $V_{CC}$ volts.
- Comparator 2 compares the trigger voltage with a reference voltage + 1/3 $V_{CC}$ volts.

Output of both the comparators is supplied to the flip-flop. Flip-flop assumes its state according to the output of the two comparators. One of the two transistors is a discharge transistor of which collector is connected to**pin** 7. This transistor saturates or cuts-off according to the output state of the flip-flop. The saturated transistor provides a discharge path to a capacitor connected externally. Base of another transistor is connected to a reset terminal. A pulse applied to this terminal resets the whole timer irrespective of any input.

## OBSERVATIONS:

Draw here the output wave form obtained from your designed circuit.

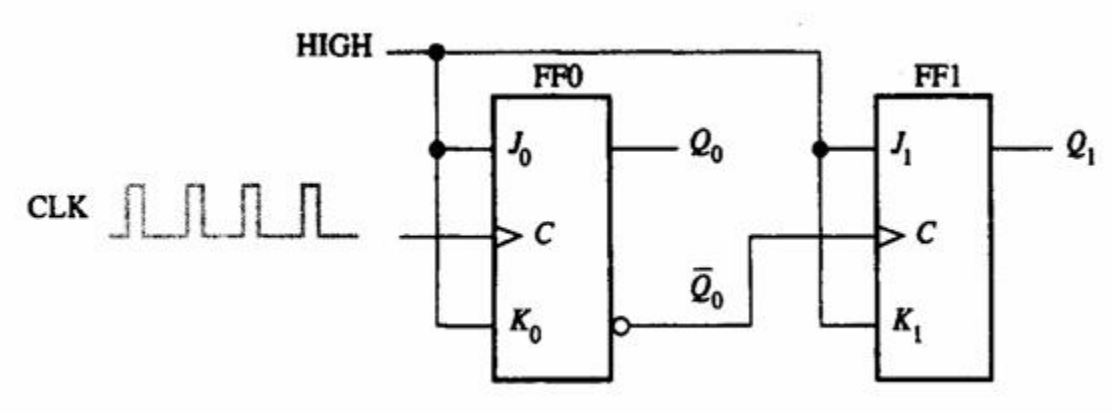
## RESULT:

The circuits were implemented and the required waveforms were observed on an oscilloscope.

# LAB NO.8

**OBJECTIVE : To implement a two bit gray counter and a two bit binary counter using J K flip flops.**

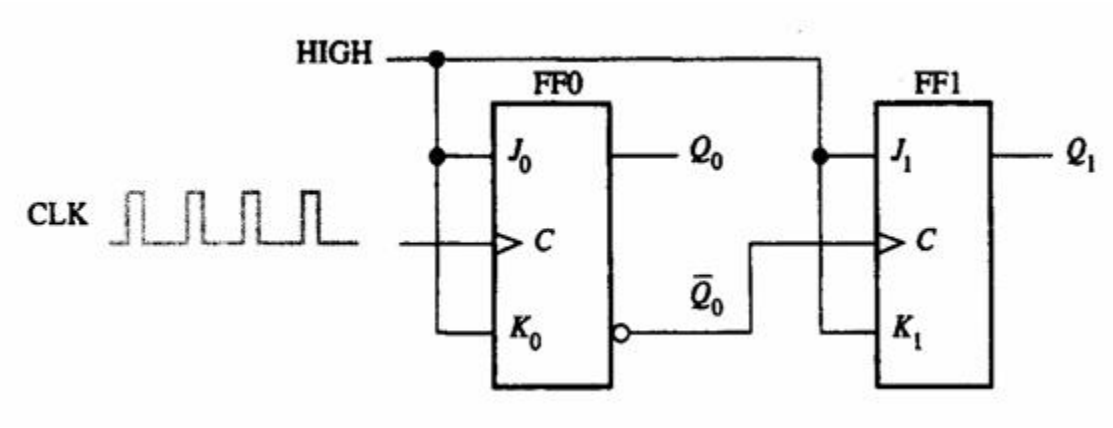
## 2 BIT GRAY COUNTER:

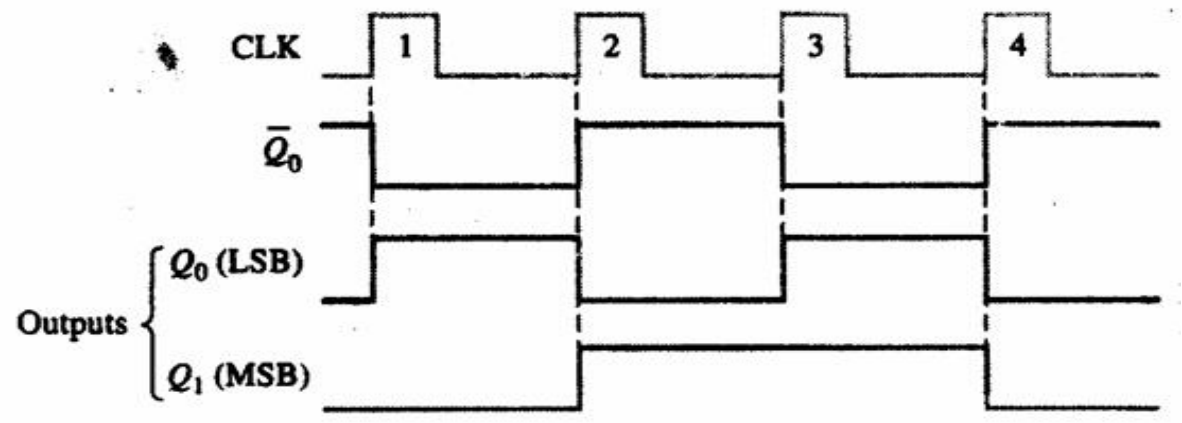



## OBSERVATIONS:

| Clk | Q1 | Qo |
|---|---|---|
| 1↑ | | |
| 1↓ | | |
| 2↑ | | |
| 2↓ | | |

## 2 BIT BINARY COUNTER:




**OBSERVATIONS:**

| Clk | Q1 | Qo |
|---|---|---|
| 1↑ | | |
| 2↑ | | |
| 3↑ | | |
| 4↑ | | |

**RESULT:**

What difference you observed between the two counter outputs?

# LAB NO.09

**OBJECTIVE: To design a combinational circuit and implement it with multiplexers. To use a demultiplexer to implement a multiple output combinational circuit from the same input variables.**

## APPARATUS:

- IC type 7404 HEX inverter
- IC type 7408 quad 2-input AND gate
- IC type 74151 8x1 multiplexer (1)
- IC type 74153 dual 4x1 multiplexer (2)
- IC type 7446 BCD-to-Seven-Segment decoder (1)
- Resistance network (1)
- Seven-Segment Display (1)

## BRIEF OVERVIEW:

74151 is a 8 line-to-1 line multiplexer. It has the schematic representation shown in Fig 1. Selection lines S2, S1 and S0 select the particular input to be multiplexed and applied to the output.
Strobe S acts as an enable signal. If strobe =1, the chip 74151 is disabled and output y = 0. If strobe = 0 then the chip 74151 is enabled and functions as a Multiplexer. Table 1 shows the multiplex function of 74151 in terms of select lines.

Table 1.

| Strobe | Select Lines | | | Output |
|---|---|---|---|---|
| S | $S_2$ | $S_1$ | $S_0$ | Y |
| 1 | X | X | X | 0 |
| 0 | 0 | 0 | 0 | D0 |
| 0 | 0 | 0 | 1 | D1 |
| 0 | 0 | 1 | 0 | D2 |
| 0 | 0 | 1 | 1 | D3 |
| 0 | 1 | 0 | 0 | D4 |
| 0 | 1 | 0 | 1 | D5 |
| 0 | 1 | 1 | 0 | D6 |
| 0 | 1 | 1 | 1 | D7 |

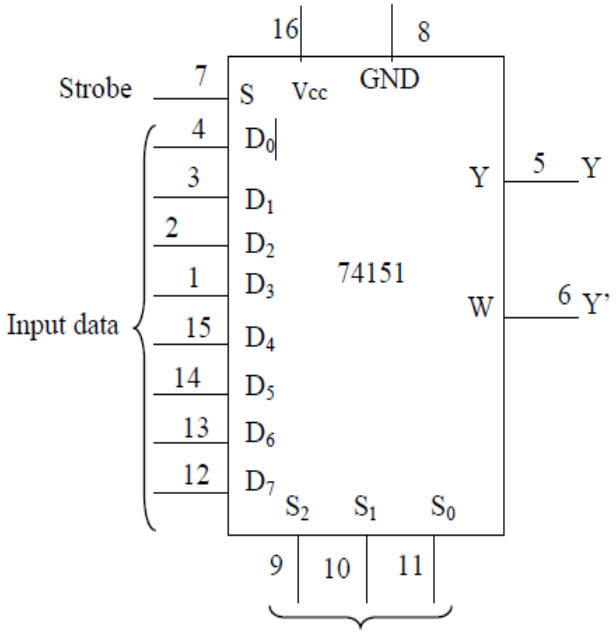



*Fig.1 IC type 74151 Multiplexer 8×1*

74153 is a dual 4 line-to-1 line multiplexer. It has the schematic representation shown in Fig 2. Selection lines S1 and S0 select the particular input to be multiplexed and applied to the output IY{1 = 1, 2}.
Each of the strobe signals *IG* {I = 1, 2} acts as an enable signal for the corresponding multiplexer.

Table 2. shows the multiplex function of 74153 in terms of select lines. Note that each of the on-chip multiplexers act independently from the other, while sharing the same select lines S1 and S0.

Table 2

| | Multiplexer 1 | | |
|---|---|---|---|
| Strobe | Select lines | | Output |
| 1G | $S_1$ | $S_0$ | 1Y |
| 1 | X | X | 0 |
| 0 | 0 | 0 | $1D_0$ |
| 0 | 0 | 1 | $1D_1$ |
| 0 | 1 | 0 | $1D_2$ |
| 0 | 1 | 1 | $1D_3$ |

| | Multiplexer 2 | | |
|---|---|---|---|
| Strobe | Select lines | | Output |
| 2G | $S_1$ | $S_0$ | 2Y |
| 1 | X | X | 0 |
| 0 | 0 | 0 | $2D_0$ |
| 0 | 0 | 1 | $2D_1$ |
| 0 | 1 | 0 | $2D_2$ |
| 0 | 1 | 1 | $2D_3$ |

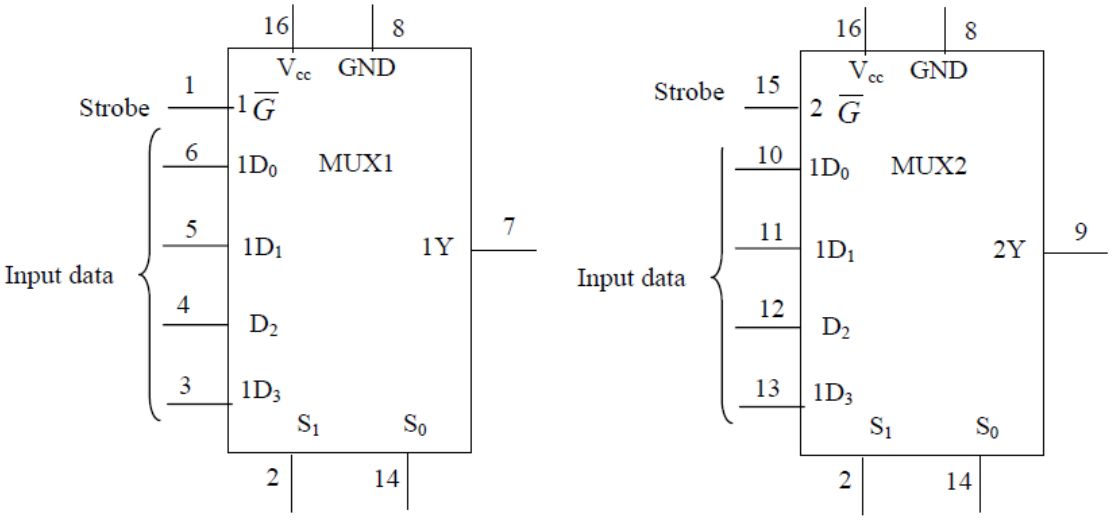



***Fig.2 Pinout of 74153***

IC 7446 is a BCD to seven segment decoder driver. It is used to convert the Combinational circuit outputs in BCD forms into 7 segment digits for the 7 segment LED display units.

## PROCEDURE:

*Part I: Parity Generator:*

a) Design a parity generator by using a 74151 multiplexer. Parity is an extra bit attached to a code to check that the code has been received correctly. Odd parity bit means that the number of 1's in the code including the parity bit is an odd number. Fill the output column of the truth table in Table 2 for a 5-bit code in which four of the bits (A,B,C,D) represents the information to be sent and fifth bit (x), represents the parity bit. The required parity is an odd parity.

The inputs B,C and D correspond to the select inputs of 74151. Complete the truth table in Table 3 by filling in the last column with 0,1,A or A'.

b) Simulate the circuit using proteus , use 74-151 multiplexer and Binary switches for inputs and Binary Probes for outputs. The 74151 has one output for Y and another inverted output W. Use A and A' for providing values for inputs 0-7. The internal values "A, B, C" are used for selection inputs B,C, and D. Simulate the circuit and test each input combination filling in the table shown below. In the Lab connect the circuit and verify the operations. Connect an LED to the multiplexer output so that it represents the parity bit which lights any time when the four bits input have even parity.

| Inputs | | | | Outputs | Connect data to |
|---|---|---|---|---|---|
| A | B | C | D | X | |
| 0 | 0 | 0 | 0 | | |
| 0 | 0 | 0 | 1 | | |
| 0 | 0 | 1 | 0 | | |
| 0 | 0 | 1 | 1 | | |
| 0 | 1 | 0 | 0 | | |
| 0 | 1 | 0 | 1 | | |
| 0 | 1 | 1 | 0 | | |
| 0 | 1 | 1 | 1 | | |
| 1 | 0 | 0 | 0 | | |
| 1 | 0 | 0 | 1 | | |
| 1 | 0 | 1 | 0 | | |
| 1 | 0 | 1 | 1 | | |
| 1 | 1 | 0 | 0 | | |
| 1 | 1 | 0 | 1 | | |
| 1 | 1 | 1 | 0 | | |
| 1 | 1 | 1 | 1 | | |

## Part 2: Vote Counter:

A committee is composed of a chairman (C), a senior member (S), and a member (M). The rules of the committee state that:

- The vote of the member (M) will be counted as 2 votes
- The vote of the senior member will be counted as 3 votes.
- The vote of the chairman will be counted as 5 votes.

Each of these persons has a switch to close ("l") when voting yes and to open ("0") when voting no.

It is necessary to design a circuit that displays the total number of votes for each issue. Use a seven segment display and a decoder to display the required number.

If all members vote no for an issue the display should be blank. (Recall from Experiment #5, that a binary input 15 into the 7446 blanks all seven segments).

If all members vote yes for an issue, the display should be 0. Otherwise the display shows a decimal number equal to the number of 'yes' votes. Use two 74153 units, which include four multiplexers to design the combinational circuit that converts the inputs from the members' switch to the BCD digit for the 7446.

In Proteus use +5V for Logic 1 and ground for Logic 0 and use switches for C, S,and M. Use two chips 74153 and one decoder 7446 verify your design and get a copy of your circuit with the pin numbers to Lab so that you could connect the hardware in exactly the same way.

**OBSERVATIONS:**

**RESULT:**

# Lab N0.10

## OBJECTIVE : To construct and study the operations of the following circuits:

**RS and Clocked RS Flip-Flop**
**D Flip-Flop**

## OVERVIEW:

So far you have encountered with *combinatorial logic*, i.e. circuits for which the output depends only on the inputs. In many instances it is desirable to have the next output depending on the current output. A simple example is a *counter*, where the next number to be output is determined by the current number stored. Circuits that remember their current output or state are often called *sequential logic* circuits. Clearly, sequential logic requires the ability to store the current state. In other words, *memory* is required by sequential logic circuits, which can be created with boolean gates. If you arrange the gates correctly, they will remember an input value. This simple concept is the basis of RAM (random access memory) in computers, and also makes it possible to create a wide variety of other useful circuits.

Memory relies on a concept called **feedback**. That is, the output of a gate is fed back into the input. The simplest possible feedback circuit using two inverters is shown below (Fig.1):

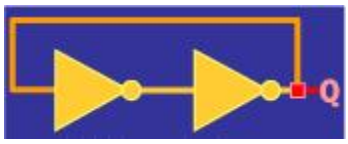


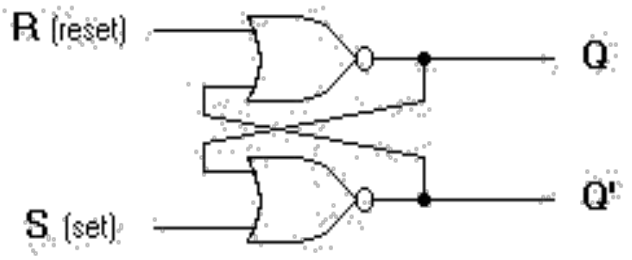

**Fig.1: Simplest realization of feedback circuit**

If you follow the feedback path, you can see that if Q happens to be 1 (or 0), it will always be 1 (or 0) . Since it's nice to be able to control the circuits we create, this one doesn't have much use -- but it does let you see how feedback works. It turns out that in "real" sequential circuits, you can actually use this sort of simple inverter feedback approach. The memory elements in these circuits are called *flip-flops*. A flip-flop circuit has two outputs, one for the normal value and one for the complement value of the stored bit. Binary information can enter a flip-flop in a variety of ways and gives rise to different types of flip-flops.

## RS Flip-Flop

RS flip-flop is the simplest possible memory element. It can be constructed from two NAND gates or two NOR gates. Let us understand the operation of the RS flip-flop using NOR gates as shown below using the truth table for 'A NOR B' gate. The inputs R and S are referred to as the Reset and Set inputs, respectively. The outputs Q and Q' are complements of each other and are referred to as the normal and complement outputs, respectively. The binary state of the flip- flop is taken to be the value of the normal output. When Q=1 and Q'=0, it is in the *set state* (or 1-state). When Q=0 and Q'=1, it is in the *reset/clear state* (or 0-state).

## Circuit Diagram:




| A | B | $\overline{A+B}$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

**S=1 and R=0:** The output of the bottom NOR gate is equal to zero, Q'=0. Hence both inputs to the top NOR gate are equal to 0, thus, Q=1. Hence, the input combination S=1 and R=0 leads to the flip-flop being **set** to Q=1.
**S=0 and R=1:** Similar to the arguments above, the outputs become Q=0 and Q'=1. We say that the flip-flop is **reset**.
**S=0 and R=0:** Assume the flip-flop was previously in set (S=1 and R=0) condition. Now changing S to 0 results Q' still at 0 and Q=1. Similarly, when the flip-flop was previously in a reset state (S=0 and R=1), the outputs do not change. Therefore, with inputs S=0 and R=0, the flip-flop holds its state.
**S=1 and R=1:** This condition violates the fact that both outputs are complements of each

other since each of them tries to go to 0, which is not a stable configuration. It is impossible to predict which output will go to 1 and which will stay at 0. In normal operation this condition must be avoided by making sure that 1's are not applied to both inputs simultaneously, thus making it one of the main disadvantages of RS flip-flop.

All the above conditions are summarized in the characteristic table below:

**Characteristic Table:**

| R | S | Q | Q' | Comment |
|---|---|---|---|---|
| 0 | 0 | Q | Q' | Hold state |
| 0 | 1 | 1 | 0 | Set |
| 1 | 0 | 0 | 1 | Reset |
| 1 | 1 | ? | ? | Indeterminate |

***Debounce circuit***

An elementary example using this flip-flop is the debounce circuit. Suppose a piece of electronics is to change state under the action of a mechanical switch. When this switch is moved from position S to R (S=0, R=1), the contacts make and break several times at R before settling to good contact. It is desirable that the electronics should respond to the first contact and then remain stable, rather than switching back and forth as the circuit makes and breaks. This is achieved by RS flip-flop which is reset to Q=0 by the first signal R=1 and remains in a fixed state until the switch is moved back to position S, when the signal S=1 sets the flip-flop to Q=1.

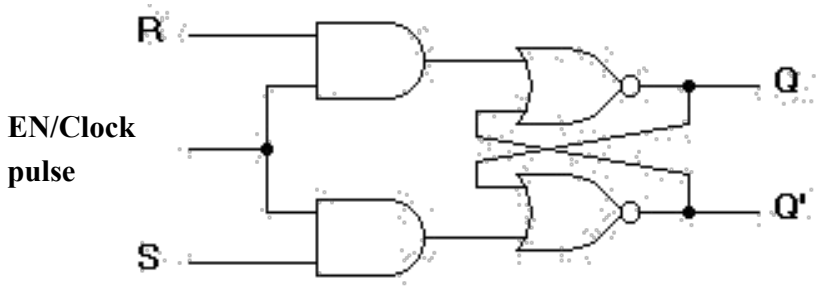
***Gated or Clocked RS Flip-Flop***

It is sometimes desirable in sequential logic circuits to have a bistable RS flip-flop that only changes state when certain conditions are met regardless of the condition of either the Set or the Reset inputs. By connecting a 2-input AND gate in series with each input terminal of the RS NOR Flip-flop a Gated RS Flip-flop can be created. This extra conditional input is called an "Enable" input and is given the prefix of "EN" as shown below. When the Enable input "EN" = 0, the outputs of the two AND gates are also at logic level 0, (AND Gate principles) regardless of the condition of the two inputs S and R, latching the two outputs Q and Q’ into their last known state. When the enable input "EN" = 1, the circuit responds as a normal RS bistable flip-flop with the two AND gates becoming transparent to the Set and Reset signals. This Enable input can also be connected to a clock timing signal adding clock synchronisation to the flip-flop creating what is sometimes called a "Clocked SR Flip-flop".

So a **Gated/Clocked RS Flip- flop** operates as a standard bistable latch but the outputs are only activated when a logic "1" is applied to its EN input and deactivated by a

logic "0". The property of this flip-flop is summarized in its characteristic table where $Q_n$ is the logic state of the previous output and $Q_{n+1}$ is that of the next output and the clock input being at logic 1 for all the R and S input combinations.

## Circuit Diagram:




## Characteristic Table:

| $Q_n$ | R | S | $Q_{n+1}$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 (Hold) |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | Indeterminate |
| 1 | 0 | 0 | 1 (Hold) |
| 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | Indeterminate |

## D FLIP-FLOP

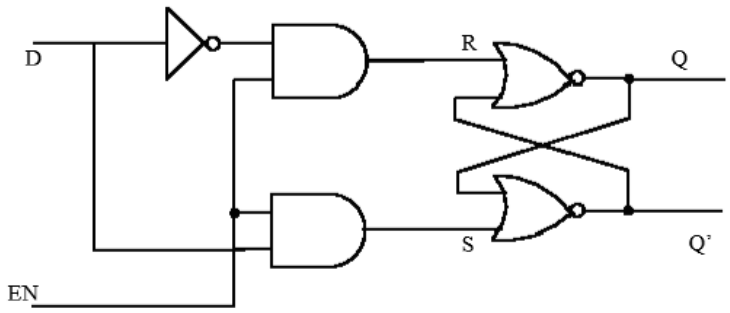
An RS flip-flop is rarely used in actual sequential logic because of its undefined outputs for inputs R= S= 1. It can be modified to form a more useful circuit called D flip-flop, where D stands for data. The D flip-flop has only a single data input D as shown in the circuit diagram. That data input is connected to the S input of an RS flip-flop, while the inverse of D is connected to the R input. To allow the flip-flop to be in a holding state, a D-flip flop has a second input called Enable, EN. The Enable-input is AND-ed with the D-input.

When **EN=0**, irrespective of D-input, the R = S = 0 and the **state is held**.
When **EN= 1**, the S input of the RS flip-flop equals the D input and R is the inverse of D.

Hence, output **Q follows D**, when EN= 1. When **EN returns to 0**, the most recent input **D is 'remembered'**.

The circuit operation is summarized in the characteristic table for **EN=1**.

## Circuit Diagram:




## Characteristic Table:

| $Q_n$ | D | $Q_{n+1}$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**Procedure:**

Assemble the circuits one after another on your breadboard as per the circuit diagrams. Circuit diagrams given here do not show connections to power supply and LEDs assuming that you are already familiar with it from your previous lab experience.
Connect the ICs properly to power supply (pin 14) and ground (pin 7) following the schematics for ICs given above.
Using dip switch and resistors, facilitate all possible combinations of inputs from the power supply. Use the switch also to facilitate pulse input to the circuit.
Turn on power to your experimental circuit.

For each input combination, note the logic state of the normal and complementary outputs as indicated by the LEDs (ON = 1; OFF = 0), and record the results in a table.
Compare your results with the characteristic tables.
When you are done, turn off the power to your experimental circuit.

**Observations:**

Table For RS FF: ___

Table For Gated RS FF: ____

Table For D FF: _____

**Precautions:**

Watch out for loose connections.
While changing the input condition keep the dip switch well pressed

# LAB NO.11

## OBJECTIVE : To construct and study the operations of the following circuits:

**JK and Master-Slave JK Flip-Flop**
**T Flip-Flop**

## OVERVIEW:

So far you have encountered with *combinatorial logic*, i.e. circuits for which the output depends only on the inputs. In many instances it is desirable to have the next output depending on the current output. A simple example is a *counter*, where the next number to be output is determined by the current number stored. Circuits that remember their current output or state are often called *sequential logic* circuits. Clearly, sequential logic requires the ability to store the current state. In other words, *memory* is required by sequential logic circuits, which can be created with boolean gates. If you arrange the gates correctly, they will remember an input value. This simple concept is the basis of RAM (random access memory) in computers, and also makes it possible to create a wide variety of other useful circuits.

Memory relies on a concept called **feedback**. That is, the output of a gate is fed back into the input. The simplest possible feedback circuit using two inverters is shown below (Fig.1):

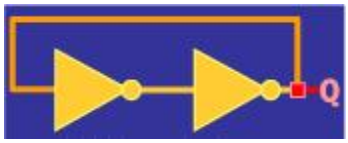
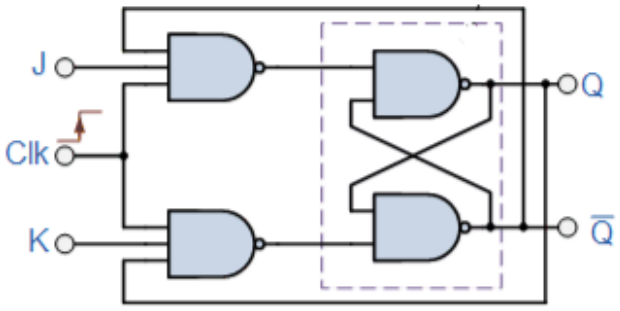



**Fig.1: Simplest realization of feedback circuit**

If you follow the feedback path, you can see that if Q happens to be 1 (or 0), it will always be 1 (or 0) . Since it's nice to be able to control the circuits we create, this one doesn't have much use -- but it does let you see how feedback works. It turns out that in "real" sequential circuits, you can actually use this sort of simple inverter feedback approach. The memory elements in these circuits are called *flip-flops*. A flip-flop circuit has two outputs, one for the normal value and one for the complement value of the stored bit. Binary information can enter a flip-flop in a variety of ways and gives rise to different types of flip-flops.

## JK FLIP-FLOP:

The JK flip flop (JK means Jack Kilby, a Texas instrument engineer, who invented it) is the most versatile flip-flop, and the most commonly used flip flop. Like the RS flip-flop, it has two data inputs, J and K, and an EN/clock pulse input (CP). Note that in the following circuit diagram NAND gates are used instead of NOR gates. It has no undefined states, however. The fundamental difference of this device is the feedback paths to the AND gates of the input, i.e. Q is AND-ed with K and CP and $\bar{Q}$ with J and CP.




| $Q_n$ | J | K | $Q_{n+1}$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1(Toggle,$\bar{Q}_n$) |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0(Toggle,$\bar{Q}_n$) |

The JK flip-flop has the following characteristics:

If one input (J or K) is at logic 0, and the other is at logic 1, then the output is set or reset (by J and K respectively), just like the RS flip-flop.
If both inputs are 0, then it remains in the same state as it was before the clock pulse occurred; again like the RS flip flop. CP has no effect on the output.
If both inputs are high, however the flip-flop changes state whenever a clock pulse occurs; i.e., the clock pulse toggles the flip-flop again and again until the CP goes back to 0 as shown in the shaded rows of the characteristic table above. Since this condition is undesirable, it should be eliminated by an improvised form of this flip-flop as discussed in the next section.

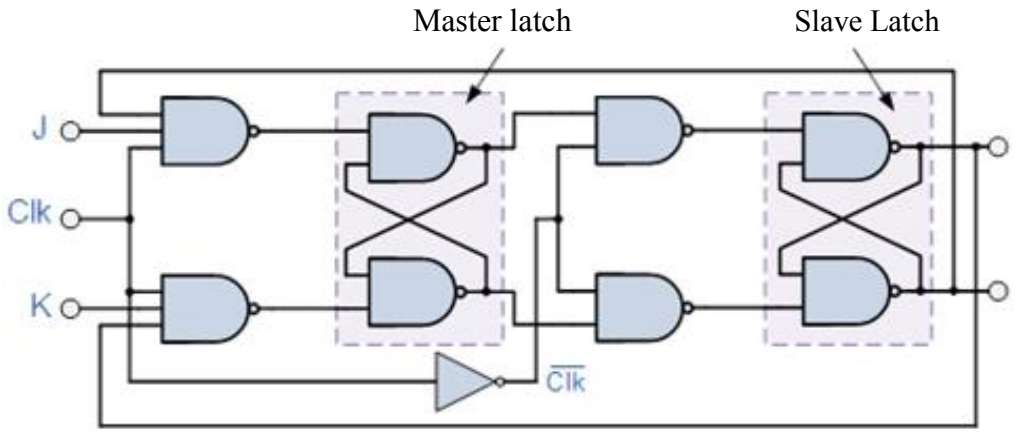
## MASTER-SLAVE JK FLIP-FLOP:

Although JK flip-flop is an improvement on the clocked SR flip-flop it still suffers from timing problems called "race" if the output Q changes state before the timing pulse of the clock input has time to go "OFF", so the timing pulse period (T) must be kept as short as possible (high frequency). As this is sometimes not possible with modern TTL IC's the much improved Master-Slave J-K Flip-Flop was developed. This eliminates all the timing problems by using two SR flip-flops connected together in series, one for the "Master" circuit, which triggers on the leading edge of the clock pulse and the other, the "Slave" circuit, which triggers on the falling edge of the clock pulse.

The master-slave JK flip flop consists of two flip flops arranged so that when the clock

pulse enables the first, or master, it disables the second, or slave. When the clock changes state again (i.e., on its falling edge) the output of the master latch is transferred to the slave latch. Again, toggling is accomplished by the connection of the output with the input AND gates.

## Circuit Diagram:




**Characteristic Table:**

| CP | J | K | $Q_m$ | $\overline{Q}_m$ | $Q_n$ | $\overline{Q}_n$ |
|---|---|---|---|---|---|---|
| 0→1 | 0 | 0 | Hold | | Hold | |
| 1→0 | 0 | 0 | Hold | | Hold | |
| 0→1 | 0 | 1 | 0 | 1 | Hold | |
| 1→0 | 0 | 1 | Hold | | 0 | 1 |
| 0→1 | 1 | 0 | 1 | 0 | Hold | |
| 1→0 | 1 | 0 | Hold | | 1 | 0 |
| 0→1 | 1 | 1 | Toggle | | Hold | |
| 1→0 | 1 | 1 | Hold | | Toggle | |

## T FLIP-FLOP:

The T flip-flop is a single input version of the JK flip-flop. The T flip-flop is obtained from the JK type if both inputs are tied together.

## Circuit Diagram:

Same as Master-Slave JK flip-flop with J=K=1

The toggle, or T, flip-flop is a bistable device, where the output of the T flip-flop "toggles" with each clock pulse.

Till CP=0, the output is in hold state (three input AND gate principle).

When CP=1, for T=0, previous output is memorized by the circuit. When T=1 along with the clock pulse, the output toggles from the previous value as given in the characteristic table below.

**Characteristic Table:**

| $Q_n$ | T | $Q_{n+1}$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**Procedure:**

Assemble the circuits one after another on your breadboard as per the circuit diagrams. Circuit diagrams given here do not show connections to power supply and LEDs assuming that you are already familiar with it from your previous lab experience.

Connect the ICs properly to power supply (pin 14) and ground (pin 7) following the schematics for ICs given above.

Using dip switch and resistors, facilitate all possible combinations of inputs from the power supply. Use the switch also to facilitate pulse input to the circuit.

Turn on power to your experimental circuit.

For each input combination, note the logic state of the normal and complementary outputs as indicated by the LEDs (ON = 1; OFF = 0), and record the results in a table.

Compare your results with the characteristic tables.

When you are done, turn off the power to your experimental circuit.

**Observations:**

Table For JK FF: _____

Table For Master-Slave JK FF:_____

Table For T FF: _____

**Precautions:**

Watch out for loose connections.

While changing the input condition keep the dip switch well pressed

# LAB NO.12

## Objective:

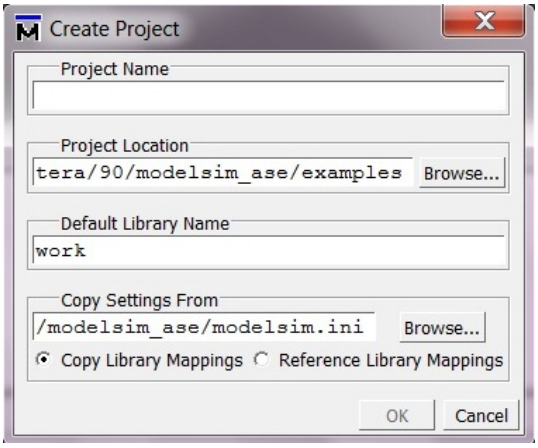
1) Getting familiar with Verilog HDL for digital design.
2) To simulate and verify the verilog code on ModelSim Software.

### Equipment Required

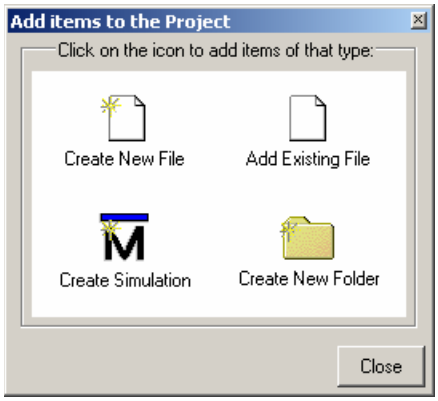
- Modelsim software installed PCs

## Procedure

1. Open the ModelSim software.
2. Create a new project by **File => New => Project** from the Main window.
3. A "**Create Project**" window appears as shown in figure below. Select a suitable name for your project; leave the Default Library Name to work.

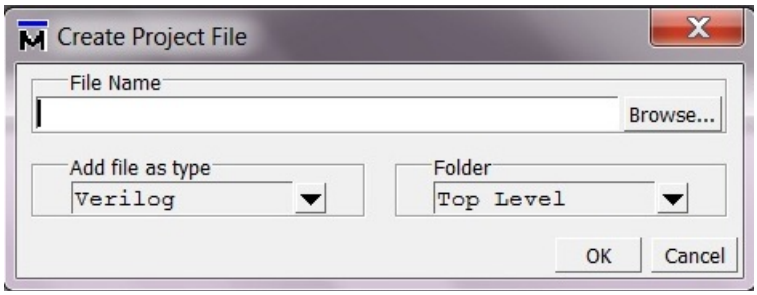



4. After project name, an **Add items to the Project** dialog pops out as shown in figure below.

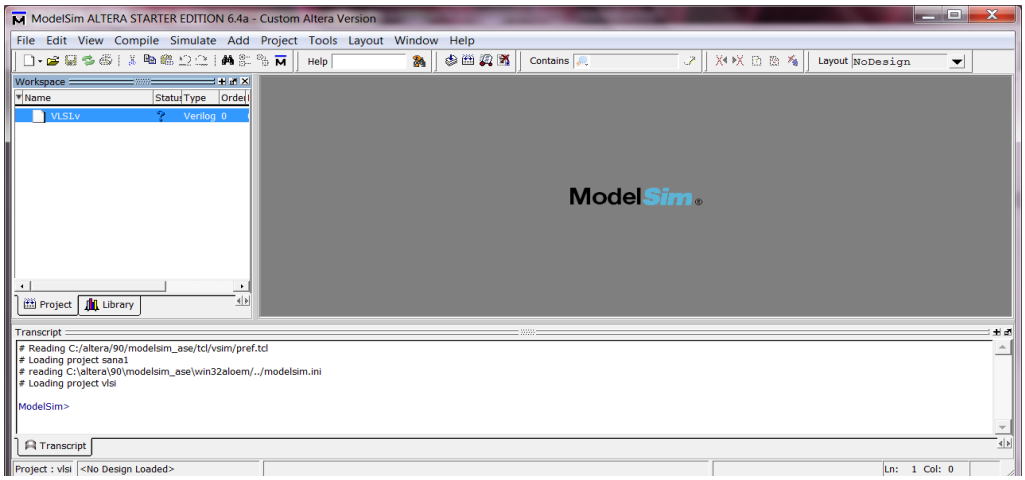


5. From the “Add items to the Project” dialog click on **Create a new file**. If you have closed the “Add items to the Project” dialog, then select **Project => Add to Project => New File** from the main window.

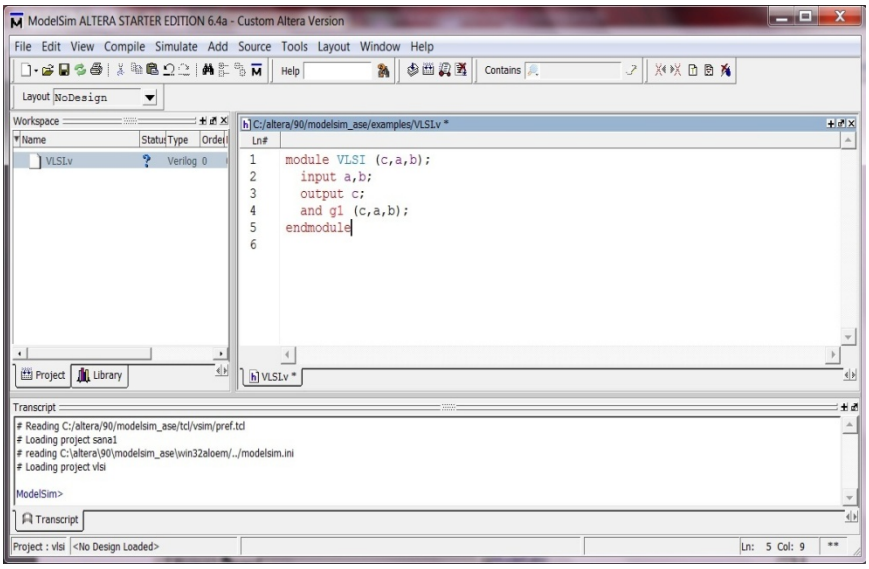
6. A **Create Project File** dialog pops out. Select an appropriate file name for the file you want to add (the name of file must be same as you write in Step 4); choose **Verilog** as the add file as type option and **Top level** as the Folder option (see figure below) and then click on OK.




7. On the workspace section of the **Main Window**, double-click on the file you have just created (VLSI.v in our case).

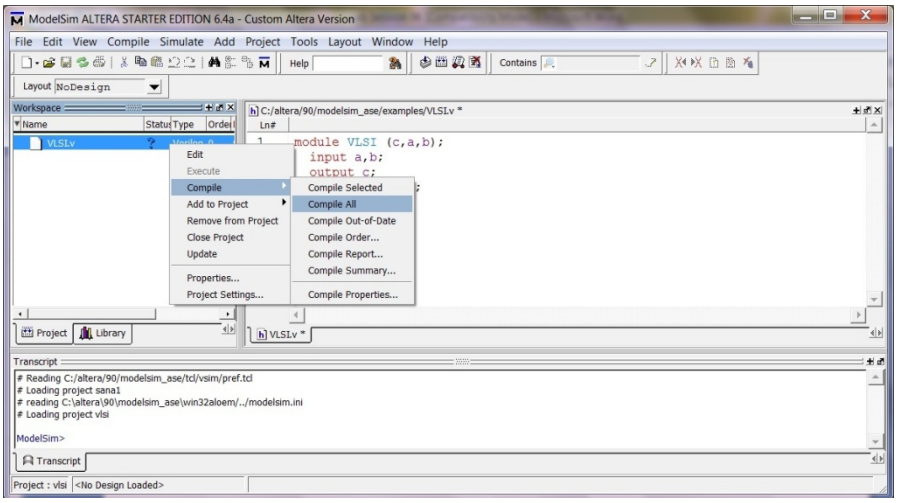



8. Type verilog code of the given task in the new window. For example here we use a simple AND gate code.



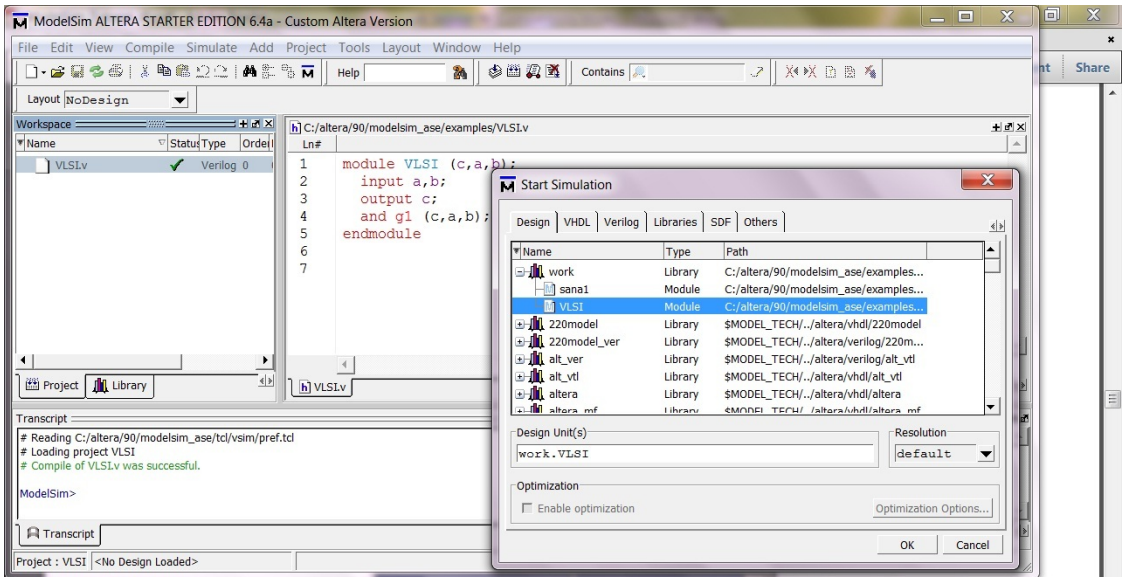

9. Save your code.

10. In workspace window do right click on project name ( i.e. VLSI) select **Compile => Compile All**. A message " Compile of VLSI.v was successful" will appear in message window.

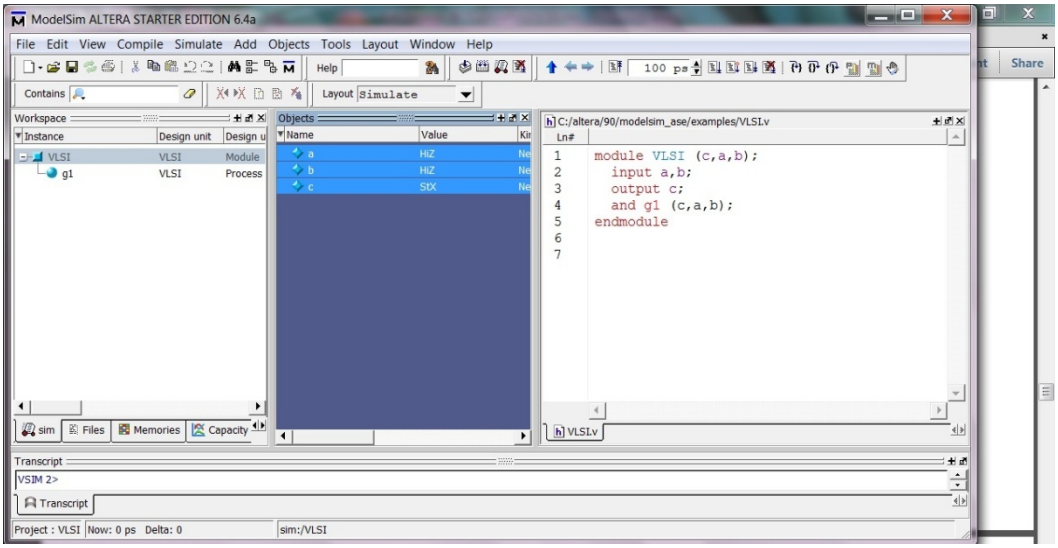



11. For simulating the design click on **Simulation => Start Simulation** in main window, simulation environment will appears as shown in figure below.

12. Click on the (+) sign next to the **work** library. You should see the name of the entity of the code that we have just compiled **"VLSI"** select your desired file.



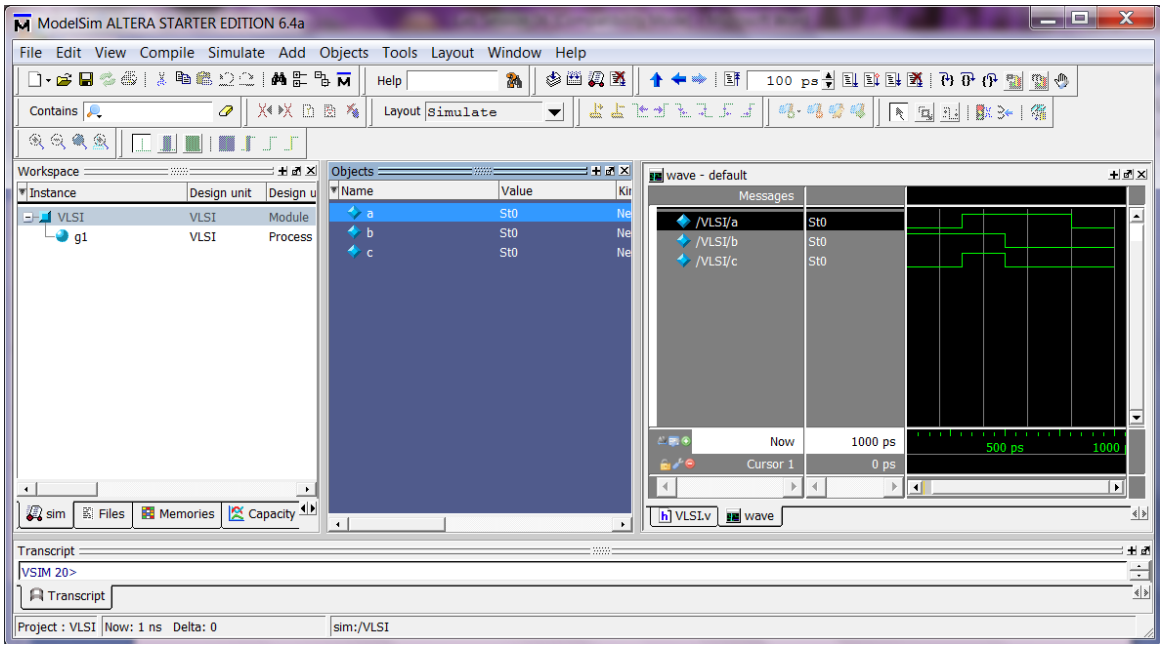

13. Locate the signals window and select the signals that you want to monitor for simulation. For this example of AND gate, select all signals as shown figure below.




14. Drag the above signals by selecting all then right click and select **Add => To Wave => Selected items** to the wave window.

15. Now we are ready to simulate our design. For this purpose we will change the values of inputs (i.e. a and b in above example of AND gate) by right click on input and select **Force** and write either '0' or '1' in value box and repeat same step for changing the value of other inputs.

16. Click **Run** button in main window tool bar and can see the changes in the both the wave and objects windows.




## Results & Observations

# Lab No.13

## Objective

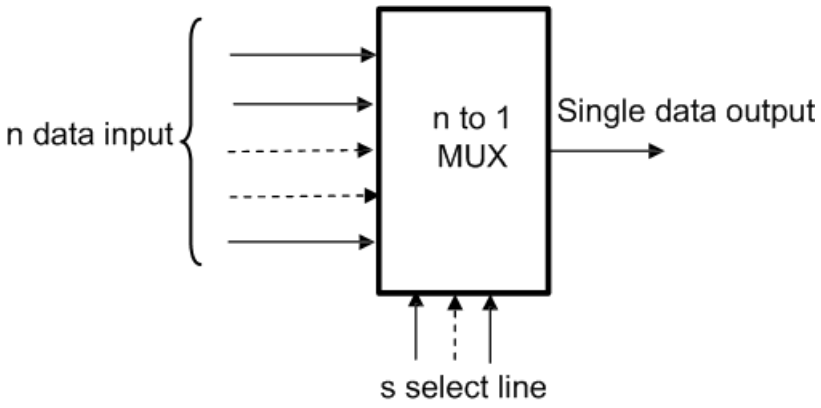
- To understand 4 to 1 MUX working principle
- To understand Quartus-II Software for Development of Verilog HDL Codes.
- To implement and Test 4 to 1 MUX on Verilog HDL by
    - Gate Level Modeling
    - Data Flow Modeling
    - Behavioral Modeling

## Equipment Required

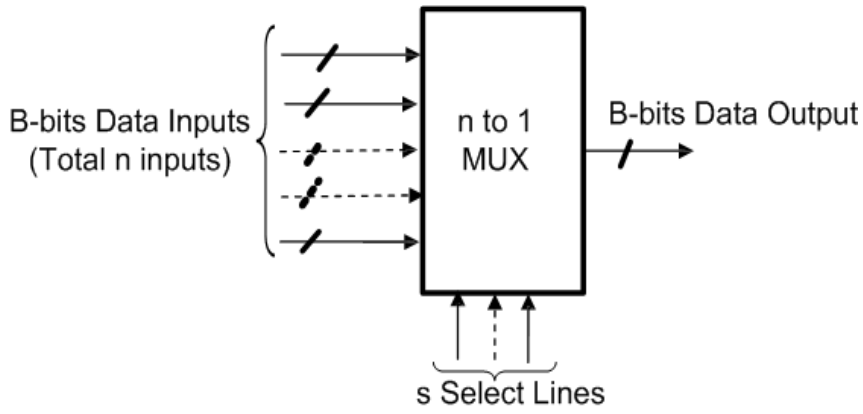
- Quartus-II Modelsim Installed PCs

## Theory

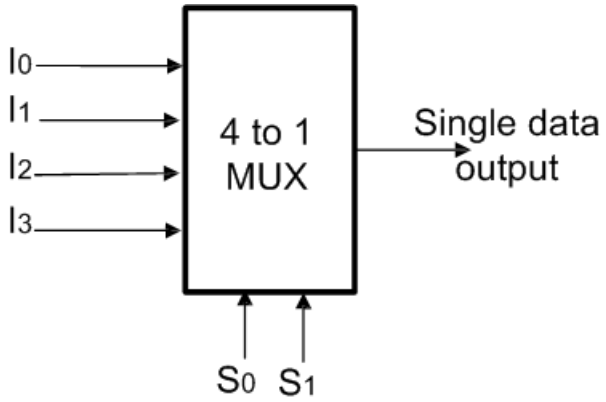
A multiplexer (MUX) is a digital switch which connects data from one of "n" inputs to a single output. A number of "Select Inputs" determine which data input is connected to the output. The Block Diagram of MUX with "n" data inputs and "s" select lines is shown in figure below:




MUX acts like a digitally controlled multi-position switch where the binary code applied to the select inputs controls the input source that will be switched on to the output. At any given point of time only one input gets selected and is connected to output, based on the select input signal. Input can be single bit or multi bits in nature. Following figure shows n to 1 MUX, handling "B" bits of each input and select them to "B" bits output.




A 4 to 1 MUX is shown in figure below. There are four input lines, I0 to I3, and two selections lines, S0 and S1, are decoded to select a particular input to appear at output.




The truth table for the 4:1 MUX is given as:

| S1 | S0 | Output |
|---|---|---|
| 0 | 0 | I0 |
| 0 | 1 | I1 |
| 1 | 0 | I2 |
| 1 | 1 | I3 |

## Procedure

1. Understand given Gate level modeling code.
2. Create new Quartus-II project for writing the code.
3. Open new Verilog file and write given code in it.
4. Include Verilog file in your project and compile your project.
5. Open Modelsim and simulate your project and verify results.
6. Understand given Data flow modeling code.
7. Repeat steps 3 to 5.
8. Understand given Behavioral Modeling code.
9. Repeat steps 3 to 5.

## Gate Level Modeling

```
// Module 4-to-1 multiplexer using gate level modeling. module mux4_to_1 (output out,
                    input       i0, i1, i2, i3, input      s1,
                    s0);
// Internal wire declarations wire s1n, s0n;
wire y0, y1, y2, y3;
// Gate instantiations
// Create s1n and s0n signals
not (s1n, s1);
not (s0n, s0);
// 3-input and gates instantiated
and (y0, i0, s1n, s0n);
and (y1, i1, s1n, s0);
and (y2, i2, s1, s0n);
and (y3, i3, s1, s0);
// 4-input or gate instantiated
```

```
or (out, y0, y1, y2, y3);
endmodule
```

## Data Flow Modeling

```
// Module 4-to-1 multiplexer using data-flow modeling module mux4_to_1 ( output
                                        out,
                    input               i0, i1, i2, i3,
                    input               s1, s0);
//Logic equation for out
assign out =        (~s1 & ~s0 & i0)| (~s1 & s0 &
                    i1) | (s1 & ~s0 & i2) | (s1 & s0
                    & i3) ;
endmodule
```

## Behavioral Modeling

```
// Module 4-to-1 multiplexer using behavioral modeling module mux4_to_1 ( output
                                        out,
                    input               i0, i1, i2, i3, input     s1,
                    s0);
always @(s1 or s0 or i0 or i1 or i2 or i3)
begin
    case ({s1, s0})
    2'b00: out = i0;
    2'b01: out = i1;
```

```
    2'b10: out = i2;
    2'b11: out = i3; default: out = 1'bx;
    endcase
 end endmodule
```

## Results

# Lab No.14

## Objective

- To understand Quartus-II Software for Development of Verilog HDL codes.
- To implement and test Verilog HDL code of a given function.

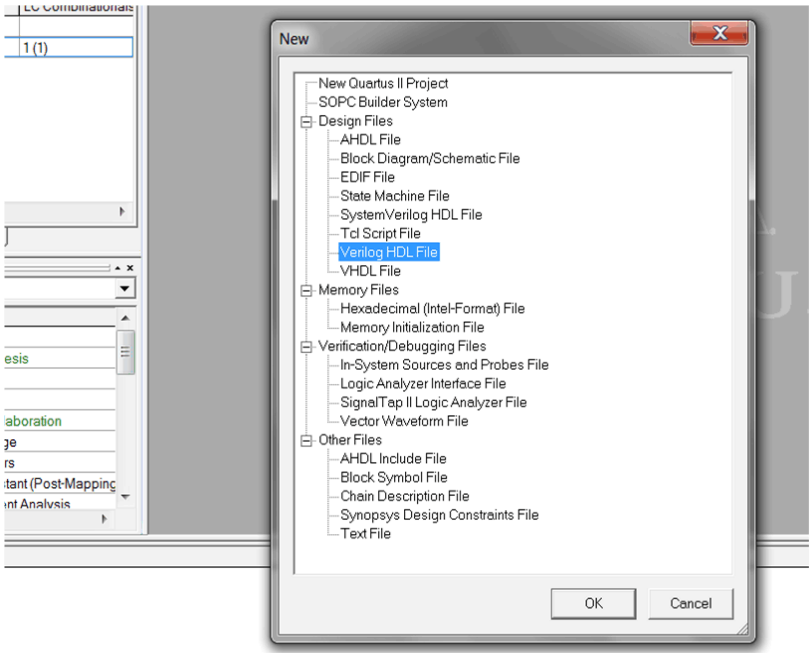
## Equipment Required

- Quartus II software installed PCs
- ALTERA DE2 Board

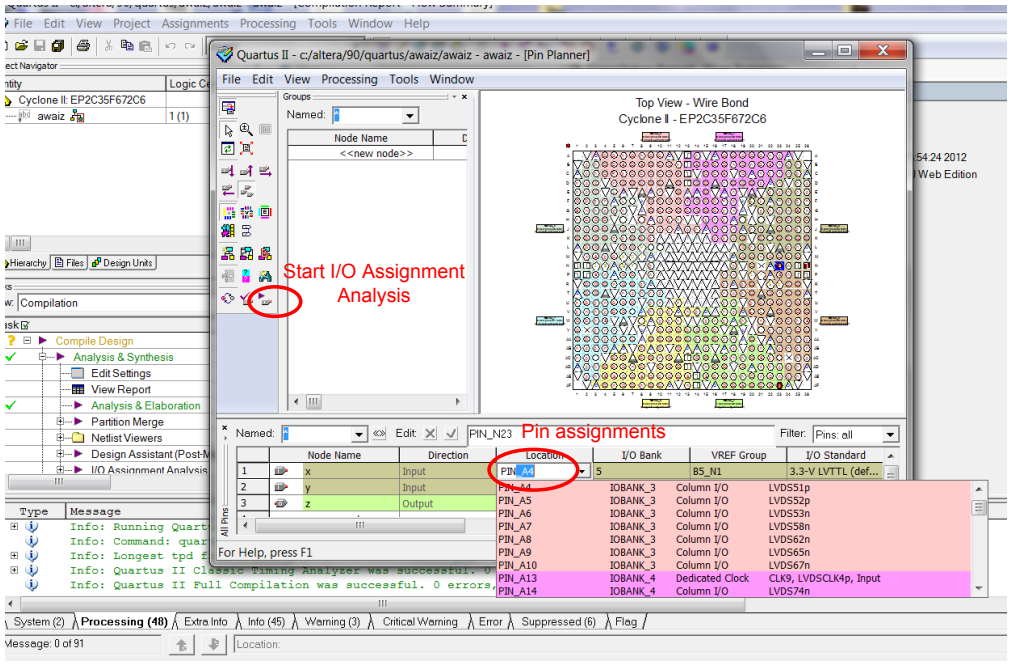
## Procedure

1. Open the Quartus II software.
2. Create a new project by selecting "**Create a New Project (New Project Wizard)**" as shown in Figure below.


3. Select a suitable name for your new directory (or you can use the existing one) and also the name of the project and click on **next** option.
4. After creating new directory and project, create a new file by selecting **File** => **New** and select **Verilog HDL File** type as shown in Figure below.




5. A command window will appear. Write your program and save it with the same name as given in module command. (Make sure that file should be saved in the same project directory mentioned in step **2**)
6. Now compile your program by selecting **Processing** => **start compilation.**
7. After completion of compilation a message will appear "**full compilation was successful**".
8. To verify your verilog code on ALTERA board, assign suitable pins/switches/LEDs to your input/output terminals by **Assignments** => **Pin Planner.** A pin planner window will appear.



9. Assign switches and LEDs to all input and output terminals respectively and start I/O assignment analysis as shown in above figure. The location of ALTERA DE2 board can be selected from "**ALTERA DE2 user manual**".

10. After I/O assignment analysis, now code is ready to be dumped in ALTERA DE2 board. Select **Tools** => **Programmer** and after selection of USB blaster option select **Start**. A 100% completion message will appear, when program is completely dump.

11. Now, you can test your program on ALTERA DE2 board.

## Results and Observations