

# Lab Manual

Course code - **EC655**

## HDL Programming Lab

Batch 2025-2027

---

**DEPARTMENT OF ELECTRONICS AND COMMUNICATION  
ENGINEERING**

**NATIONAL INSTITUTE OF TECHNOLOGY  
TIRUCHIRAPALLI - 620015**

---

**Course Faculty: Dr. B. Naresh Kumar Reddy**





## Contents

<b>Preface</b>	iii
<b>List of Listings</b>	v
<b>List of Figures</b>	vi
<b>List of Tables</b>	vi
<b>Introduction</b>	vii
<b>1 Experiment 1 : Full Adder and Full Subtractor</b>	1
1.1 Theory . . . . .	1
1.2 Verilog code . . . . .	2
1.3 Testbench code . . . . .	2
1.4 Implementation . . . . .	3
<b>2 Experiment 2 : Mux, Demux and Encoders</b>	4
2.1 Theory . . . . .	4
2.2 Verilog code . . . . .	6
2.3 Testbench code . . . . .	8
2.4 Implementation . . . . .	10
<b>3 Experiment 3 : Code Converters</b>	11
3.1 Theory . . . . .	11
3.2 Verilog code . . . . .	12
3.3 Testbench code . . . . .	12
3.4 Implementation . . . . .	13
<b>4 Experiment 4 : Flipflops</b>	14
4.1 Theory . . . . .	14
4.2 Verilog code . . . . .	15
4.3 Testbench code . . . . .	16
4.4 Implementation . . . . .	18
<b>5 Experiment 5 : Universal Shift Register</b>	19
5.1 Theory . . . . .	19
5.2 Verilog code . . . . .	19
5.3 Testbench code . . . . .	20
5.4 Implementation . . . . .	21
<b>6 Experiment 6 : Comparators</b>	22
6.1 Theory . . . . .	22
6.2 Verilog code . . . . .	22
6.3 Testbench code . . . . .	23
6.4 Implementation . . . . .	23

<b>7 Experiment 7 : Up/Down counters</b>	<b>24</b>
7.1 Theory . . . . .	24
7.2 Verilog code . . . . .	24
7.3 Testbench code . . . . .	25
7.4 Implementation . . . . .	25
<b>8 Experiment 8 : Array Multiplier</b>	<b>27</b>
8.1 Theory . . . . .	27
8.2 Verilog code . . . . .	27
8.3 Testbench code . . . . .	28
8.4 Implementation . . . . .	29
<b>9 Experiment 9 : RAM</b>	<b>31</b>
9.1 Theory . . . . .	31
9.2 Verilog code . . . . .	31
9.3 Testbench code . . . . .	32
9.4 Implementation . . . . .	33

## Preface

### Dos and Dont's in the Lab

- Do not install any software on any lab computer. Only lab operators and technical support personnel are authorized to carry out these tasks.
- Do not touch any exposed wires or sockets.
- Avoid all horseplay in the laboratory.
- Look away from the screen once in a while to give your eyes a rest.
- Do not attempt to open any machines, and do not touch the backs of machines when they are switched on.
- Turn off the machine you were using when you are done using it.
- Do not access external devices without scanning them for computer viruses.
- Always maintain an extra copy of all your important data.
- Keep the laboratory and work area clean and uncluttered.
- Prohibit unauthorized individuals from entering the laboratory.

## Safety

Safety in the electronics lab, like anywhere else, relies on knowing the hazards, following precautions, and using common sense. Electrical labs have serious risks—currents as low as 0.1 amp through the chest can be fatal, especially for those with heart conditions. Factors like wet skin, body resistance, and voltage affect danger levels, so never underestimate “low” voltage, as even 30 volts can be deadly. Besides electrical shock, injuries like burns, broken bones, sprains, and eye damage can occur, so take precautions to avoid them. Keep emergency phone numbers nearby and ask lab staff if you have any safety concerns. Following proper safety measures protects you and others from harm, with electric shock being the most common and dangerous risk.

### Electric Shock

Electric shock happens when current passes through the body, with severity mainly depending on the current amount rather than voltage. Around 1 mA causes tingling, above 10 mA causes painful muscle spasms making it hard to let go, and 100 to 200 mA can trigger fatal heart fibrillation. The voltage needed for a dangerous current depends on skin resistance, which varies widely from about  $150\ \Omega$  when wet to  $15\ k\Omega$  when dry. For example, 240 V with wet skin can push 500 mA through the body, which is deadly. Since skin resistance drops quickly at contact points, it’s crucial to break contact immediately to avoid lethal currents.

Always observe the following safety precautions when working in the laboratory:

1. Power must be switched off whenever an experiment or project is being assembled, disassembled, or modified.

- 2.** Never touch electrical equipment while standing on a damp or metal floor.
- 3.** In an emergency all power in the laboratory can be switched off by depressing the large red button on the main breaker panel. Locate it. It is to be used for emergencies only.
- 4.** Horseplay, running, or practical jokes must not occur in the laboratory.
- 5.** Never use water on an electrical fire. If possible switch power off, and then use CO<sub>2</sub> or a dry type fire extinguisher. Locate extinguishers and read operating instructions before an emergency occurs.

## Listings

1	FA Module	2
2	FS Module	2
3	Testbench for FA	2
4	Testbench for FS	3
5	Constraints for full adder	3
6	Mux Module	6
7	Demux Module	7
8	Encoder Module	7
9	Priority Encoder Module	7
10	Mux testbench	8
11	Demux testbench	8
12	Demux testbench	9
13	Encoder testbench	9
14	Priority Encoder testbench	9
15	Constraints for 2:1 Mux	10
16	Btg Module	12
17	Gtb Module	12
18	Btg testbench	12
19	gtb testbench	13
20	Constraints for 2:1 Mux	13
21	SR FF Module	15
22	JK FF Module	16
23	D FF Module	16
24	T FF Module	16
25	Testbench for SR FF	16
26	Testbench for JK FF	17
27	Testbench for D FF	17
28	Testbench for T FF	18
29	Constraints for SR FF	18
30	USR Module	19
31	USR testbench	20
32	Constraints for USR	21
33	1 bit comparator Module	22
34	1 bit comparator testbench	23
35	Constraints for 1 bit comparator	23
36	up/down counter Module	24
37	counter testbench	25
38	Constraints for up/down counter	25
39	Array Multiplier Module	27
40	Array Multiplier testbench	28
41	Array Multiplier Implementation	29
42	RAM Module	31
43	RAM testbench	32
44	Array Multiplier Implementation	33

## List of Figures

1.1	Logic circuit for Full Adder . . . . .	1
1.2	Logic circuit for Full Subtractor . . . . .	1
2.1	Mux (2:1) . . . . .	4
2.2	Demux (2:1) . . . . .	5
2.3	Encoder (4:2) . . . . .	5
2.4	Logic circuit for Priority Encoder . . . . .	6
3.1	Logic circuit for Binary to Gray converter . . . . .	11
3.2	Logic circuit for Gray to Binary converter . . . . .	11
4.1	Logic circuit for SR Flip-Flop . . . . .	14
4.2	Logic circuit for JK Flip-Flop . . . . .	14
4.3	Logic circuit for D Flip-Flop . . . . .	15
4.4	Logic circuit for T Flip-Flop . . . . .	15
5.1	Block diagram for Universal Shift Register . . . . .	19
6.1	Block diagram for 1 bit comparator . . . . .	22
7.1	Block diagram for up/down counter . . . . .	24
8.1	Block diagram for Array Multiplier . . . . .	27
9.1	Block diagram for RAM . . . . .	31

## List of Tables

1.1	Truth Table — Full Adder . . . . .	1
1.1	Truth Table - Full Subtractor . . . . .	2
2.1	Truth Table - Multiplexer . . . . .	4
2.2	Truth Table - Demultiplexer . . . . .	5
2.3	Truth Table - Encoder . . . . .	6
2.4	Truth Table - Priority Encoder . . . . .	6
4.2	Truth Table — SR Flip-Flop . . . . .	14
4.3	Truth Table — JK Flip-Flop . . . . .	14
4.4	Truth Table — D Flip-Flop . . . . .	15
4.5	Truth Table — T Flip-Flop . . . . .	15
6.1	Truth Table - 1 bit magnitude comparator . . . . .	22

# Introduction

## Intention of Lab Manual

The intention of this Lab manual is to provide a structured approach on how to tackle simple verilog experiments as succinctly as possible so as to enable the reader to explore more complex experiments on their own.

## Sections of each experiment

Briefly each section of the experiments are explained to give an idea of what to expect from them:

- Theory:** Brief theory on the experiment at hand
- Verilog code:** The HDL code in Verilog to simulate the experiment is given
- Testbench code:** The HDL code in Verilog used for the testbench is given
- Implementation:** The constraints (IO mapping) are provided for one of the sub-experiments to give an idea of how to proceed with implementation in FPGA

## Tools/Skills required

It is pertinent that the reader has at least a precursory knowledge on the following:

### Tools used:

S.No	Category	Tool / Resource
1.	Hardware Description Language	Verilog
2.	Software	Xillinx Vivado Design Suite
3.	FPGA	Basys 3 (part no. XC7A35TICPG296-1L)

If the reader requires a tutorial on Xillinx Vivado and basics of implementation on FPGA then they may head to [Course website tutorials](#) where they can find the resources including video tutorials required to bring them up to speed.

## Brief instructions on Vivado and FPGA Implementation

This tutorial will help you get started with using Vivado Design Suite for synthesizing, implementing, and programming designs on an FPGA development board.

### Step-by-Step Procedure:

#### 1. Create a New Vivado Project

- Open Vivado and select *Create New Project*.
- Name your project and set the location.
- Choose RTL project and add your Verilog files.
- Select the appropriate FPGA part or board (e.g., XC7A35TICPG296-1L).

#### 2. Add Design Files and Constraints

- Add your top-level Verilog file.
- Add the XDC constraints file (for pin mapping).

#### 3. Run Synthesis and Implementation

- Click on *Run Synthesis*.
- After synthesis, click on *Run Implementation*.

#### 4. Generate Bitstream

- Once implementation is complete, click *Generate Bitstream*.

#### 5. Program the FPGA

- Open *Hardware Manager*.
- Connect to the target board
- Program the device with the generated bitstream (.bit file).

### Recommended Tutorial Videos:

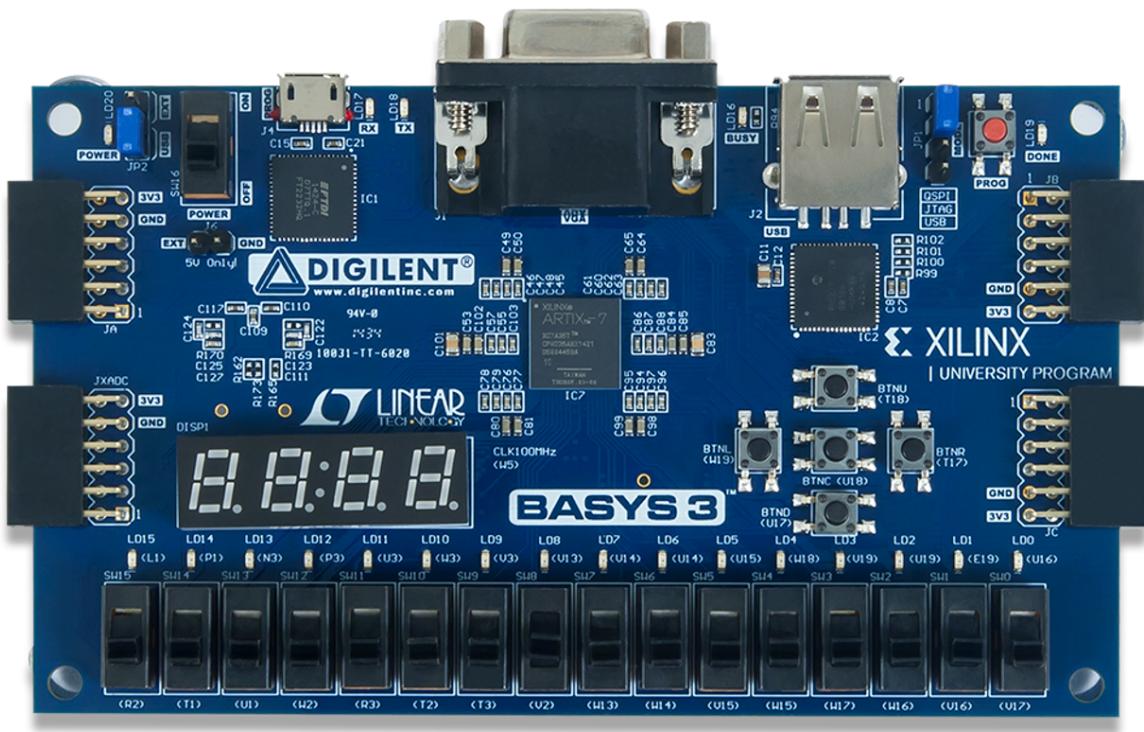
- How to Create First Xilinx FPGA Project in Vivado? - YouTube ([Link 1](#))
- FPGA - Implementation of basic logic gates on Xilinx Artix - 7 - YouTube ([Link 2](#))

## Basys 3 Board pin elaboration

For the ease of access during implementation, a picture of Basys 3 board along with the list of pin numbers and their onboard functions is provided in this section:

Note that the LEDs and Switches are listed out in the order of their actual location on the board i.e. from left to right as in the image of the board as follows:

## Basys 3 FPGA Board



### Basys 3 Pin Mapping:

LEDs		Switches	
Pin	Name	Pin	Name
L1	LED15	R2	SW15
P1	LED14	T1	SW14
N3	LED13	U1	SW13
P3	LED12	W2	SW12
U3	LED11	R3	SW11
W3	LED10	T2	SW10
V3	LED9	T3	SW9
V13	LED8	V2	SW8
V14	LED7	W13	SW7
U14	LED6	W14	SW6
U15	LED5	V15	SW5
W18	LED4	W15	SW4
V19	LED3	W17	SW3
U19	LED2	W16	SW2
E19	LED1	V16	SW1
U16	LED0	V17	SW0

Push Buttons / Clock	
Pin	Name
T18	BTN0 (Center)
W19	BTN1 (Up)
T17	BTN2 (Left)
U18	BTN3 (Right)
U17	BTN4 (Down)
W5	100 MHz Clock

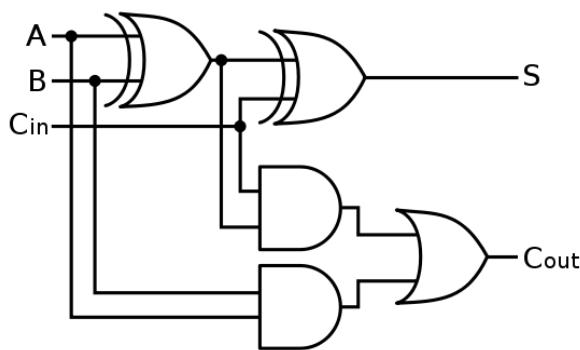
# 1 | Experiment 1 : Full Adder and Full Subtractor

## 1.1 | Theory

### 1.1.1 | Full Adder

Full adder is used to add three 1-bit inputs (say A,B,C) and provides 2 outputs namely sum and carry.

The Truth table and logic circuits for full adder is as follows:



A	B	C	Sum	Carry
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

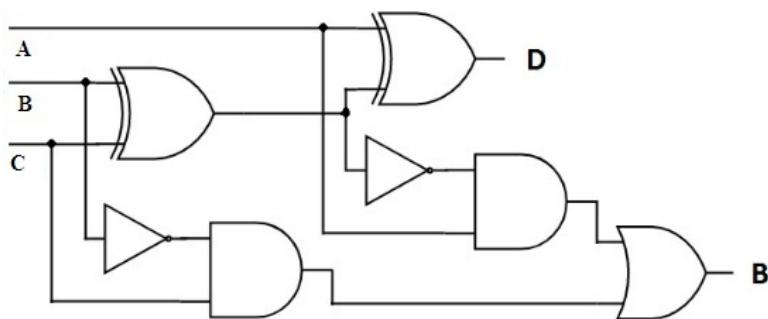
**Figure 1.1:** Logic circuit for Full Adder

**Table 1.1:** Truth Table — Full Adder

### 1.1.2 | Full Subtractor

Full subtractor is used to subtract three inputs (say A,B,C) and provides 2 outputs namely difference and carry.

The Truth table and logic circuits for full subtractor is as follows:



**Figure 1.2:** Logic circuit for Full Subtractor

A	B	C	Difference	Carry
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	0	0
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

**Table 1.1:** Truth Table - Full Subtractor

## 1.2 | Verilog code

### 1.2.1 | Full Adder

```

1 module Full_adder(
2     input wire A,B,C;
3     output wire sum, carry);
4     assign sum = A^B^C;
5     assign carry = A&B | B&C | C&A;
6 endmodule

```

**Listing 1:** FA Module

### 1.2.2 | Full Subtractor

```

1 module Full_subtractor(
2     input wire A,B,Bin;
3     output wire difference, borrow);
4     assign difference = A^B^Bin;
5     assign carry = (~A & B) | (~A ^ B) & Bin );
6 endmodule

```

**Listing 2:** FS Module

## 1.3 | Testbench code

### 1.3.1 | Full Adder

```

1 module FA_tb;
2     reg A,B,C;
3     wire sum,carry;
4     Full_adder dut(A,B,C,sum,carry);
5     initial
6         begin
7             A = 0; B = 1; C = 0; #10;

```

```

8      A = 1; B = 0; C = 1; #10;
9      end
10 endmodule

```

**Listing 3:** Testbench for FA

### 1.3.2 | Full Subtractor

```

1 module FS_tb;
2   reg A,B,C;
3   wire difference, borrow;
4   Full_subtractor dut(A,B,Bin,difference,borrow);
5   initial
6     begin
7       A = 0; B = 1; Bin = 0; #10;
8       A = 1; B = 0; Bin = 1; #10;
9     end
10 endmodule

```

**Listing 4:** Testbench for FS

## 1.4 | Implementation

### 1.4.1 | Constraints

```

1 set_property IOSTANDARD LVCMOS33 [get_ports A]
2 set_property IOSTANDARD LVCMOS33 [get_ports B]
3 set_property IOSTANDARD LVCMOS33 [get_ports C]
4 set_property IOSTANDARD LVCMOS33 [get_ports sum]
5 set_property IOSTANDARD LVCMOS33 [get_ports carry]
6 set_property PACKAGE_PIN U1 [get_ports A]
7 set_property PACKAGE_PIN U2 [get_ports B]
8 set_property PACKAGE_PIN U2 [get_ports C]
9 set_property PACKAGE_PIN T1 [get_ports sum]
10 set_property PACKAGE_PIN T2 [get_ports carry]

```

**Listing 5:** Constraints for full adder

- U1, U2, U3 are switches
- T1, T2 are LEDS.

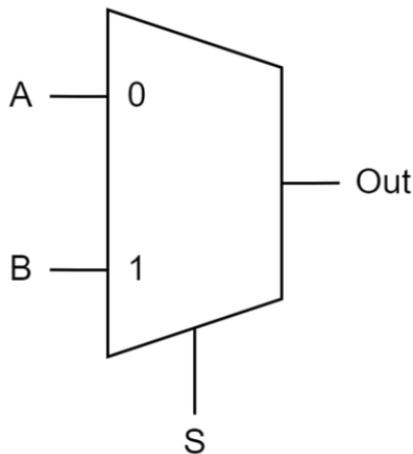
Note : The pins are to be assigned based on the FPGA and reader's requirement. The constraints given above is just for example.

## 2 | Experiment 2 : Mux, Demux and Encoders

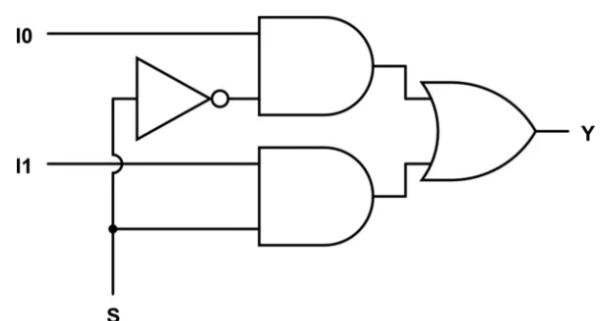
### 2.1 | Theory

#### 2.1.1 | Multiplexer

1. Multiplexer : A multiplexer (MUX), also known as a data selector, is a digital logic circuit that selects one of several input lines and directs it to a single output line. The selection is controlled by a set of select lines. The truth table and logic circuit for 2:1 mux is as follows:



(a) Mux (2:1) block diagram



(b) Mux (2:1) circuit diagram

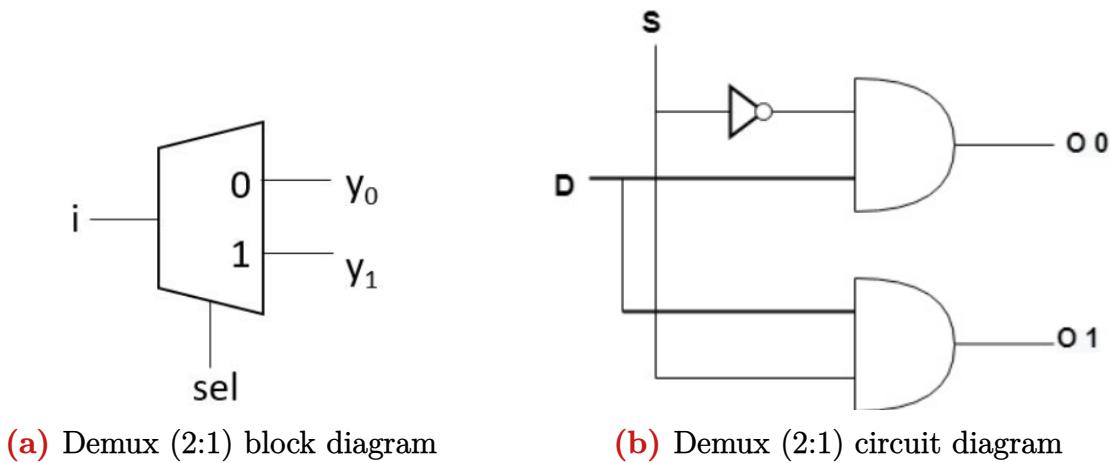
Figure 2.1: Mux (2:1)

Select (X)	A	B	Output (Y)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
0	1	1	1
1	0	0	0
1	1	0	0
1	1	1	1

Table 2.1: Truth Table - Multiplexer

#### 2.1.2 | Demultiplexer

A demultiplexer (or demux) is a digital circuit that performs the reverse operation of a multiplexer. It takes a single input line and distributes it to one of several output lines, based on the values of select lines. The truth table and logic circuit for 1:2 demux is as follows.



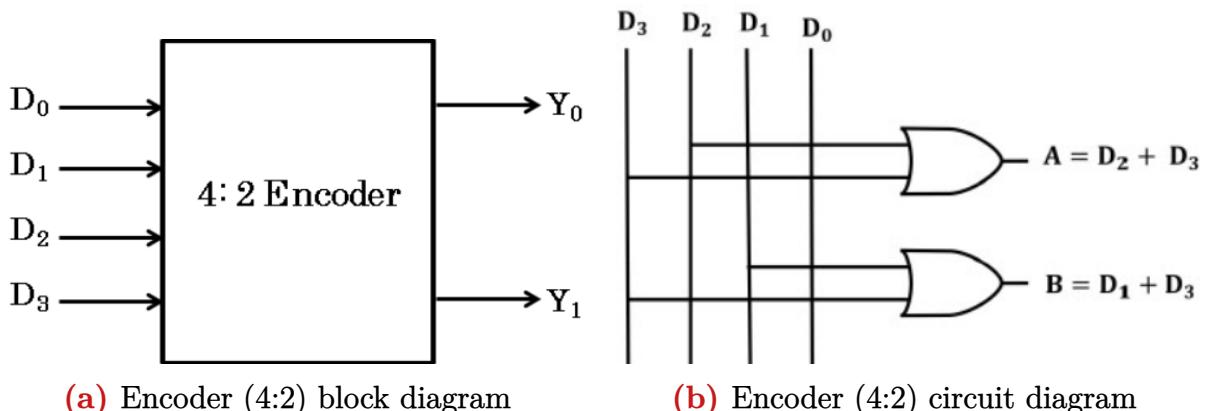
**Figure 2.2:** Demux (2:1)

Select (S)	Input (D)	Output 1 (O0)	Output (O1)
0	0	0	0
0	1	0	1
1	0	0	0
1	1	1	0

**Table 2.2:** Truth Table - Demultiplexer

### 2.1.3 | Encoder

An encoder is a digital circuit that converts a set of binary inputs into a unique binary code. The logic circuit and truth table for 4 to 2 encoder is given below:



**Figure 2.3:** Encoder (4:2)

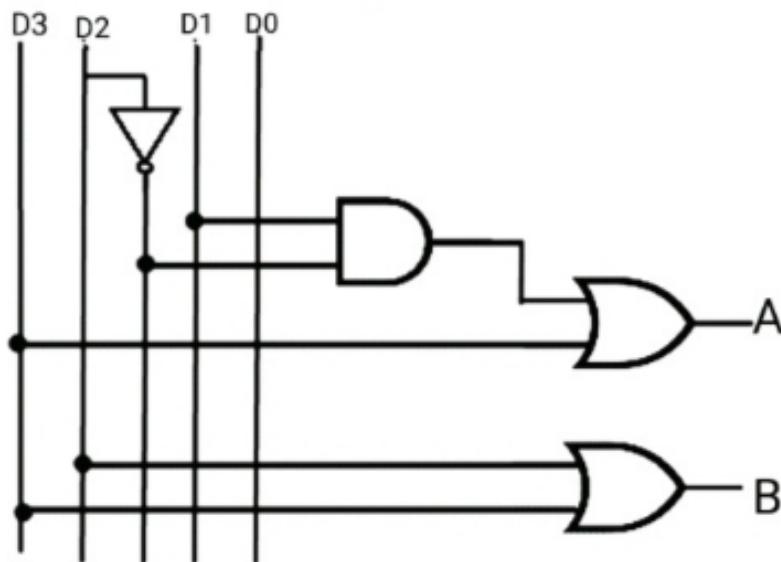
### 2.1.4 | Priority Encoder

A 4 to 2 priority encoder has 4 inputs:  $D_3, D_2, D_1, D_0$ , and 2 outputs:  $A, B$ . Here, the input,  $D_0$  has the highest priority, whereas the input,  $D_3$  has the lowest priority. In this

D3	D2	D1	D0	Output (Y1)	Output (Y0)
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

**Table 2.3:** Truth Table - Encoder

case, even if more than one input is ‘1’ at the same time, the output will be the (binary) code corresponding to the input, which is having higher priority.

**Figure 2.4:** Logic circuit for Priority Encoder

D3	D2	D1	D0	A	B
0	0	0	1	X	X
1	0	1	0	0	0
X	1	0	0	0	1
X	X	1	0	1	0
X	X	X	1	1	1

**Table 2.4:** Truth Table - Priority Encoder

## 2.2 | Verilog code

### 2.2.1 | Mux

```

1 module mux2to1 (
2     input wire a,b,s,
3     output wire y
4 );

```

```

5      assign y = s ? b : a;
6  endmodule

```

**Listing 6:** Mux Module

## 2.2.2 | Demux

```

1 module demux1to2(
2     input i,sel,
3     output reg y0,y1
4 );
5     always @(*)
6     begin
7         case(sel)
8             1 b0 : y0=i;
9             1 b1 : y1=i;
10            endcase
11        end
12    endmodule

```

**Listing 7:** Demux Module

## 2.2.3 | Encoder

```

1 module encoder4to2 (
2     input wire [3:0] in,
3     output reg [1:0] out
4 );
5     always @(*) begin
6         case (in)
7             4'b0001: out = 2'b00;
8             4'b0010: out = 2'b01;
9             4'b0100: out = 2'b10;
10            4'b1000: out = 2'b11;
11            default: out = 2'b00;
12        endcase
13    end
14 endmodule

```

**Listing 8:** Encoder Module

## 2.2.4 | Priority Encoder

```

1 module priority_encoder4to2 (
2     input wire [3:0] in,
3     output reg [1:0] out
4 );

```



```
5
6 always @(*) begin
7   casex (in)
8     4'b1xxx: out = 2'b11;
9     4'b01xx: out = 2'b10;
10    4'b001x: out = 2'b01;
11    4'b0001: out = 2'b00;
12    default: out = 2'b00;
13   endcase
14 end
15 endmodule
```

**Listing 9:** Priority Encoder Module

## 2.3 | Testbench code

### 2.3.1 | Mux

```
1 module mux_tb;
2   reg a,b,s;
3   wire y;
4   mux2to1 dut (a,b,s,y);
5   initial
6     begin
7       a=0;b=0;s=0;
8       #10 a=1;b=0;s=0;
9       #10 a=1;b=0;s=1;
10      #10 $finish;
11    end
12 endmodule
```

**Listing 10:** Mux testbench

### 2.3.2 | Demux

```
1 module tb;
2   reg i,sel;
3   wire yo,y1;
4   demux1to2 dut (i,sel,y0,y1);
5   initial
6   begin
7     i=0;sel=0;
8     #10 i=1;sel=0;
9     #10 i=0;sel=1;
10    #10 $finish;
11  end
12 endmodule
```

---

**Listing 11:** Demux testbench**2.3.3 | Encoder**

```
1 module tb;
2   reg i,sel;
3   wire yo,y1;
4   demux1to2 dut (i,sel,y0,y1);
5   initial
6   begin
7     i=0;sel=0;
8     #10 i=1;sel=0;
9     #10 i=0;sel=1;
10    #10 $finish;
11  end
12 endmodule
```

**Listing 12:** Demux testbench**2.3.4 | Encoder**

```
1 module tb;
2   reg [3:0] in;
3   wire [1:0] out;
4   encoder4to2 dut (in,out);
5   initial begin
6     in = 4'b0001; #10;
7     in = 4'b0010; #10;
8     in = 4'b0100; #10;
9     in = 4'b1000; #10;
10    in = 4'b0000; #10;
11    in = 4'b1100; #10;
12    $finish;
13  end
14 endmodule
```

**Listing 13:** Encoder testbench**2.3.5 | Priority Encoder**

```
1 module tb;
2   reg [3:0] in;
3   wire [1:0] out;
4   priority_encoder4to2 dut (in,out);
5   initial begin
6     in = 4'b0001; #10;
```

```
7      in = 4'b0010; #10;
8      in = 4'b0100; #10;
9      in = 4'b1000; #10;
10     in = 4'b0000; #10;
11     in = 4'b1100; #10;
12     $finish;
13 end
14 endmodule
```

**Listing 14:** Priority Encoder testbench

## 2.4 | Implementation

### 2.4.1 | Constraints

```
1 set_property IOSTANDARD LVCMOS33 [get_ports a]
2 set_property IOSTANDARD LVCMOS33 [get_ports b]
3 set_property IOSTANDARD LVCMOS33 [get_ports s]
4 set_property IOSTANDARD LVCMOS33 [get_ports y]
5 set_property PACKAGE_PIN V16 [get_ports a]
6 set_property PACKAGE_PIN V17 [get_ports b]
7 set_property PACKAGE_PIN R2 [get_ports s]
8 set_property PACKAGE_PIN U16 [get_ports y]
```

**Listing 15:** Constraints for 2:1 Mux

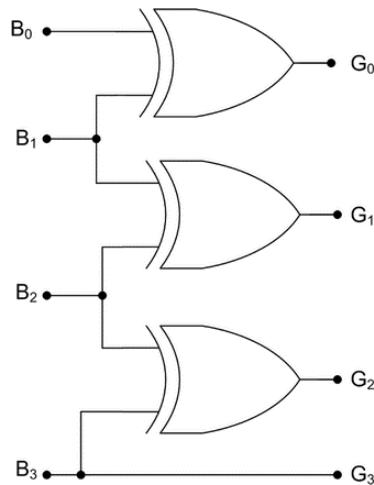
- V16, V17, W2 are switches
- U16 is LED.

## 3 | Experiment 3 : Code Converters

### 3.1 | Theory

#### 3.1.1 | Binary to Gray

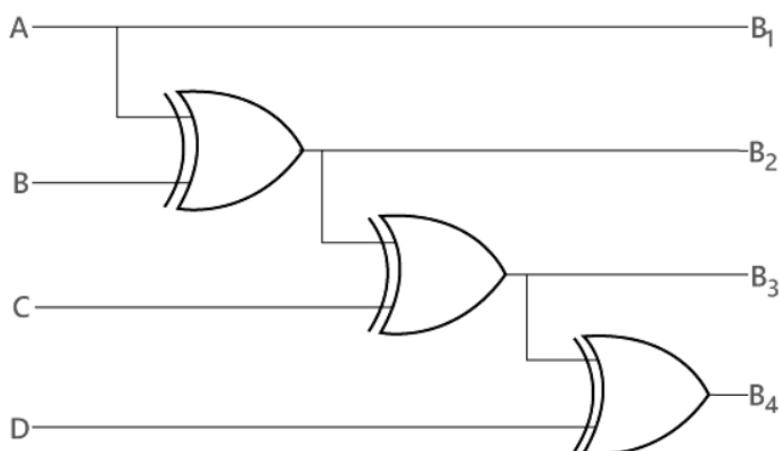
To convert a binary number to its Gray code equivalent, the most significant bit (MSB) remains the same. The subsequent bits are obtained by XORing each binary bit with its preceding bit. The circuit diagram for 4 bit binary to gray is given below.



**Figure 3.1:** Logic circuit for Binary to Gray converter

#### 3.1.2 | Gray to Binary

To convert Gray code to binary, copy the Gray code's MSB. Then, XOR the binary MSB with the next Gray code bit to get the next binary bit. Repeat this XOR process for all remaining bits, using the previously calculated binary bit. The circuit for 4 bit gray to binary is given below.



**Figure 3.2:** Logic circuit for Gray to Binary converter

## 3.2 | Verilog code

### 3.2.1 | Binary to Gray code converter

```

1 module binary_to_gray (
2     input wire [3:0] b,
3     output wire [3:0] g
4 );
5     assign g[3] = b[3];
6     assign g[2] = b[3] ^ b[2];
7     assign g[1] = b[2] ^ b[1];
8     assign g[0] = b[1] ^ b[0];
9 endmodule

```

**Listing 16:** Btg Module

### 3.2.2 | Gray to Binary code converter

```

1 module gray_to_binary (
2     input wire [3:0] g,
3     output wire [3:0] b
4 );
5     assign b[3] = g[3];
6     assign b[2] = b[3] ^ g[2];
7     assign b[1] = b[2] ^ g[1];
8     assign b[0] = b[1] ^ g[0];
9 endmodule

```

**Listing 17:** Gtb Module

## 3.3 | Testbench code

### 3.3.1 | Binary to Gray code converter

```

1 module tb;
2     reg [3:0] b;
3     wire [3:0] g;
4     binary_to_gray dut (b,g);
5     initial begin
6         b = 4'b0000; #10;
7         b = 4'b0001; #10;
8         b = 4'b0010; #10;
9         b = 4'b0011; #10;
10        $finish;
11    end
12 endmodule

```

**Listing 18:** Btg testbench

### 3.3.2 | Gray to Binary code converter

```
1 module tb;
2     reg [3:0] g;
3     wire [3:0] b;
4     gray_to_binary dut (g,b);
5     initial begin
6         g = 4'b0000; #10;
7         g = 4'b0001; #10;
8         g = 4'b0011; #10;
9         g = 4'b0010; #10;
10        $finish;
11    end
12 endmodule
```

**Listing 19:** gtb testbench

## 3.4 | Implementation

### 3.4.1 | Constraints

```
1 set_property IOSTANDARD LVCMOS33 [get_ports {b[3]}]
2 set_property IOSTANDARD LVCMOS33 [get_ports {b[2]}]
3 set_property IOSTANDARD LVCMOS33 [get_ports {b[1]}]
4 set_property IOSTANDARD LVCMOS33 [get_ports {b[0]}]
5 set_property IOSTANDARD LVCMOS33 [get_ports {g[3]}]
6 set_property IOSTANDARD LVCMOS33 [get_ports {g[2]}]
7 set_property IOSTANDARD LVCMOS33 [get_ports {g[1]}]
8 set_property IOSTANDARD LVCMOS33 [get_ports {g[0]}]
9 set_property PACKAGE_PIN W17 [get_ports {b[3]}]
10 set_property PACKAGE_PIN W16 [get_ports {b[2]}]
11 set_property PACKAGE_PIN V17 [get_ports {b[1]}]
12 set_property PACKAGE_PIN V16 [get_ports {b[0]}]
13 set_property PACKAGE_PIN V19 [get_ports {g[3]}]
14 set_property PACKAGE_PIN U19 [get_ports {g[2]}]
15 set_property PACKAGE_PIN E19 [get_ports {g[1]}]
16 set_property PACKAGE_PIN U16 [get_ports {g[0]}]
```

**Listing 20:** Constraints for 2:1 Mux

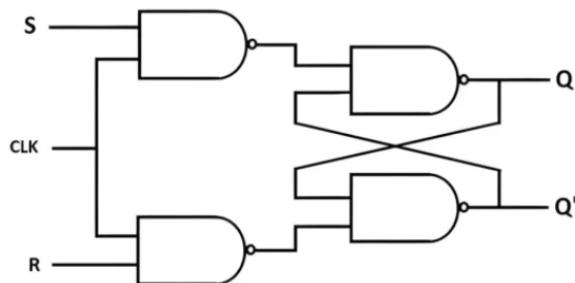
- W17, W16, V17, V16 are switches
- V19, U19, E19, U16 are LEDs.

## 4 | Experiment 4 : Flipflops

### 4.1 | Theory

#### 4.1.1 | SR Flipflop

The SR (Set-Reset) flip-flop is a basic bistable circuit with two inputs, S and R, and one output Q. It stores one bit of data. When S=1 and R=0, it sets the output to 1. When S=0 and R=1, it resets the output to 0. S=R=0 holds the current state, while S=R=1 is an invalid condition.



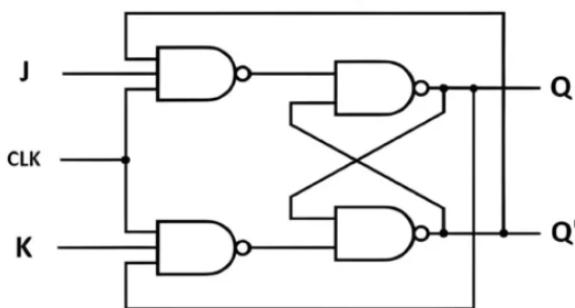
Clk	S	R	Q(t+1)
0	X	X	Q(t)
1	0	0	Q(t)
1	0	1	1
1	1	0	1
1	1	1	invalid

**Figure 4.1:** Logic circuit for SR Flip-Flop

**Table 4.2:** Truth Table — SR Flip-Flop

#### 4.1.2 | JK Flipflop

The JK flip-flop is a refinement of the SR flip-flop that avoids the invalid state. It has two inputs: J (set) and K (reset). When both are high, the output toggles. When J=K=0, it holds its state.



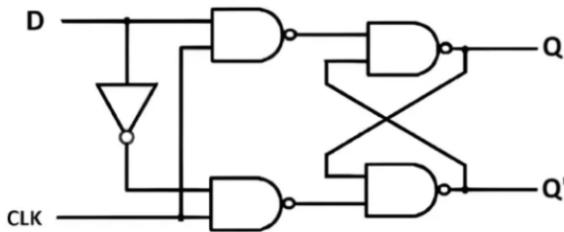
Clk	J	K	Q(t+1)
0	X	X	Q(t)
1	0	0	Q(t)
1	0	1	0
1	1	0	1
1	1	1	$\neg Q(t)$

**Figure 4.2:** Logic circuit for JK Flip-Flop

**Table 4.3:** Truth Table — JK Flip-Flop

#### 4.1.3 | D Flipflop

The D (Data or Delay) flip-flop has a single data input D and captures its value at the rising edge of the clock. It is the most commonly used flip-flop in digital systems for data storage and synchronization.

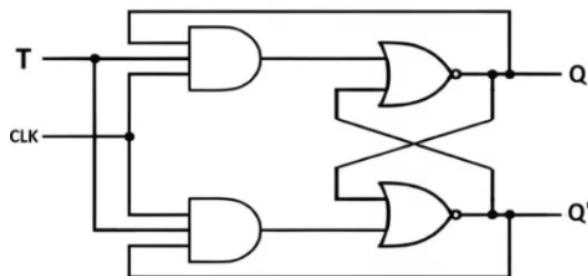
**Figure 4.3:** Logic circuit for D Flip-Flop

Clk	D	$Q(t+1)$
0	X	$Q(t)$
1	0	0
1	1	1

**Table 4.4:** Truth Table — D Flip-Flop

#### 4.1.4 | T Flipflop

The T (Toggle) flip-flop toggles its output on each rising clock edge if T=1. If T=0, it holds its state. It is often used in counters and frequency division.

**Figure 4.4:** Logic circuit for T Flip-Flop

Clk	T	$Q(t+1)$
0	X	$Q(t)$
1	0	$Q(t)$
1	1	$\neg Q(t)$

**Table 4.5:** Truth Table — T Flip-Flop

## 4.2 | Verilog code

### 4.2.1 | SR Flipflop

```

1 module sr_flipflop(input clk, s, r, output reg q, output qbar
2 );
3   always @ (posedge clk)
4   begin
5     case ({s, r})
6       2'b00: q <= q;
7       2'b01: q <= 0;
8       2'b10: q <= 1;
9       2'b11: q <= 1'bx; // invalid
10      endcase
11    end
12    assign qbar = ~q;
13  endmodule

```

**Listing 21:** SR FF Module

#### 4.2.2 | JK Flipflop

```

1 module jk_flipflop(input clk, j, k, output reg q, output qbar
2   );
3   always @ (posedge clk)
4   begin
5     case ({j, k})
6       2'b00: q <= q;
7       2'b01: q <= 0;
8       2'b10: q <= 1;
9       2'b11: q <= ~q;
10    endcase
11  end
12  assign qbar = ~q;
endmodule

```

**Listing 22:** JK FF Module

#### 4.2.3 | D Flipflop

```

1 module d_flipflop(input clk, d, output reg q, output qbar);
2   always @ (posedge clk)
3   q <= d;
4   assign qbar = ~q;
5 endmodule

```

**Listing 23:** D FF Module

#### 4.2.4 | T Flipflop

```

1 module t_flipflop(input clk, t, output reg q=0, output qbar);
2   always @ (posedge clk)
3   if (t)
4     q <= ~q;
5   else
6     q <= q;
7   assign qbar = ~q;
8 endmodule

```

**Listing 24:** T FF Module

### 4.3 | Testbench code

#### 4.3.1 | SR Flipflop

```

1 module tb_sr_flipflop;
2   reg clk, s, r;
3   wire q, qbar;

```



```
4  sr_flipflop uut (clk, s, r, q, qbar);
5  always #5 clk = ~clk;
6  initial begin
7    clk=0;
8    s=0; r=0; #10;
9    s=1; r=0; #10;
10   s=0; r=1; #10;
11   s=1; r=1; #10;
12   s=0; r=0; #10;
13   $finish;
14 end
15 endmodule
```

**Listing 25:** Testbench for SR FF

#### 4.3.2 | JK Flipflop

```
1 module tb_jk_flipflop;
2   reg clk, j, k;
3   wire q, qbar;
4   jk_flipflop uut (clk, j, k, q, qbar);
5   always #5 clk = ~clk;
6   initial begin
7     clk=0;
8     j=0; k=0; #10;
9     j=1; k=0; #10;
10    j=0; k=1; #10;
11    j=1; k=1; #50;
12    $finish;
13  end
14 endmodule
```

**Listing 26:** Testbench for JK FF

#### 4.3.3 | D Flipflop

```
1 module tb_d_flipflop;
2   reg clk,d;
3   wire q, qbar;
4   d_flipflop uut (clk, d, q, qbar);
5   always #5 clk = ~clk;
6   initial begin
7     clk=0;
8     d = 0; #10;
9     d = 1; #10;
10    d = 0; #10;
11    $finish;
```



```
12 end  
13 endmodule
```

**Listing 27:** Testbench for D FF

#### 4.3.4 | T Flipflop

```
1 module tb_t_flipflop;  
2   reg clk,t;  
3   wire q, qbar;  
4   t_flipflop uut (clk, t, q, qbar);  
5   always #5 clk = ~clk;  
6   initial begin  
7     clk=0;  
8     t = 0; #10;  
9     t = 1; #10;  
10    t = 1; #10;  
11    t = 0; #10;  
12    $finish;  
13  end  
14 endmodule
```

**Listing 28:** Testbench for T FF

### 4.4 | Implementation

#### 4.4.1 | Constraints

```
1 set_property IOSTANDARD LVCMOS33 [get_ports clk]  
2 set_property IOSTANDARD LVCMOS33 [get_ports q]  
3 set_property IOSTANDARD LVCMOS33 [get_ports qbar]  
4 set_property IOSTANDARD LVCMOS33 [get_ports s]  
5 set_property IOSTANDARD LVCMOS33 [get_ports r]  
6 set_property PACKAGE_PIN W5 [get_ports clk]  
7 set_property PACKAGE_PIN E19 [get_ports q]  
8 set_property PACKAGE_PIN U16 [get_ports qbar]  
9 set_property PACKAGE_PIN V16 [get_ports s]  
10 set_property PACKAGE_PIN V17 [get_ports r]
```

**Listing 29:** Constraints for SR FF

- W5 is clock generator
- V17, V16 are switches
- E19, U16 are LEDs.

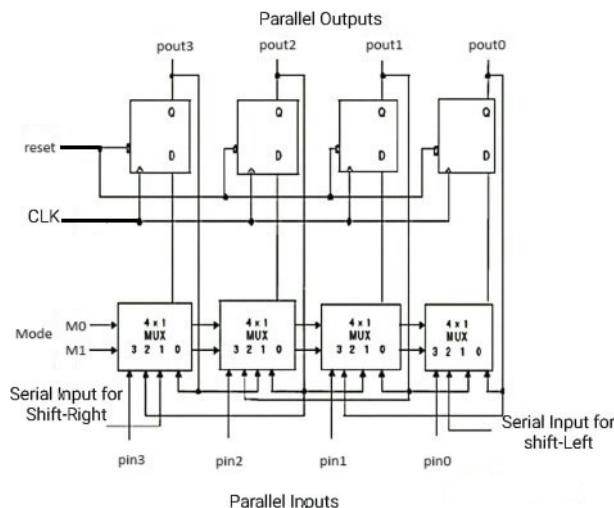
**Note:** W5 is a clock of 100MHz so the LEDs switching might occur too fast for it to make any difference. Clock divider is to implemented to divide the clock from 100MHz to say 1Hz to actually observe the output of sequential circuits in LED and the implementation of clock divider is left as an exercise to the reader.

## 5 | Experiment 5 : Universal Shift Register

### 5.1 | Theory

#### 5.1.1 | Universal Shift Register

A Universal Shift Register is a versatile 4-bit register that can perform multiple data handling operations such as Serial-In Serial-Out (SISO), Serial-In Parallel-Out (SIPO), Parallel-In Serial-Out (PISO), and Parallel-In Parallel-Out (PIPO) based on the control mode inputs. It is widely used in digital systems for data transfer and manipulation.



**Figure 5.1:** Block diagram for Universal Shift Register

### 5.2 | Verilog code

#### 5.2.1 | Universal Shift Register

```

1 module universal_shift_register (
2     input clk, rst,
3     input [1:0] mode,           // 00: Hold, 01: Shift Right,
4                                // 10: Shift Left, 11: Parallel Load
5     input sin,                 // Serial Input for
6         Shift Right & Left (SISO, SIPO)
7     input [3:0] pin,            // Parallel input (PIPO,
8         PISO)
9     output reg sout,           // Serial Output (PISO, SISO)
10    output reg [3:0] pout     // Parallel Output (SIPO, PIPO)
11 );
12    always @ (posedge clk or posedge rst) begin
13        if (rst)
14            {sout, pout} <= 0;
15        else begin
16            case (mode)
17                2'b00: begin

```



```
15      pout <= pout; // Hold
16  end
17  2'b01: begin
18      sout <= pout[0];
19      pout <= {sin, pout[3:1]}; // Shift Right
20  end
21  2'b10: begin
22      sout <= pout[3];
23      pout <= {pout[2:0], sin}; // Shift Left
24  end
25  2'b11: begin
26      pout <= pin; // Parallel Load
27  end
28  endcase
29 end
30 end
31 endmodule
```

Listing 30: USR Module

## 5.3 | Testbench code

### 5.3.1 | Universal Shift Register

```
1 module tb_universal_shift_register;
2   reg clk, rst;
3   reg [1:0] mode;
4   reg sin;
5   reg [3:0] pin;
6   wire sout;
7   wire [3:0] pout;
8   universal_shift_register uut (clk, rst, mode, sin, pin,
9     sout, pout);
10  always #5 clk = ~clk;
11  initial begin
12    clk=0;
13    rst = 1; mode = 2'b00; sin = 0; pin = 4'b0000; #10;
14    rst = 0; mode = 2'b11; pin = 4'b1010; #10; // Parallel
15    load (PIPO)
16    mode = 2'b01; sin = 1; #10; // Shift Right (SIPO)
17    mode = 2'b01; sin = 0; #10;
18    mode = 2'b10; sin = 1; #10; // Shift Left
19    mode = 2'b10; sin = 0; #10;
20    mode = 2'b11; pin = 4'b1100; #10; // Reload Parallel
21    mode = 2'b00; #10; // Hold
22    $finish;
23  end
```

```
22 endmodule
```

**Listing 31:** USR testbench

## 5.4 | Implementation

### 5.4.1 | Constraints

```

1 set_property IOSTANDARD LVCMOS33 [get_ports {mode[1]}]
2 set_property IOSTANDARD LVCMOS33 [get_ports {mode[0]}]
3 set_property IOSTANDARD LVCMOS33 [get_ports {pin[3]}]
4 set_property IOSTANDARD LVCMOS33 [get_ports {pin[2]}]
5 set_property IOSTANDARD LVCMOS33 [get_ports {pin[1]}]
6 set_property IOSTANDARD LVCMOS33 [get_ports {pin[0]}]
7 set_property IOSTANDARD LVCMOS33 [get_ports {pout[3]}]
8 set_property IOSTANDARD LVCMOS33 [get_ports {pout[2]}]
9 set_property IOSTANDARD LVCMOS33 [get_ports {pout[1]}]
10 set_property IOSTANDARD LVCMOS33 [get_ports {pout[0]}]
11 set_property PACKAGE_PIN T1 [get_ports {mode[0]}]
12 set_property PACKAGE_PIN W17 [get_ports {pin[3]}]
13 set_property PACKAGE_PIN W16 [get_ports {pin[2]}]
14 set_property PACKAGE_PIN V16 [get_ports {pin[1]}]
15 set_property PACKAGE_PIN V17 [get_ports {pin[0]}]
16 set_property PACKAGE_PIN V19 [get_ports {pout[3]}]
17 set_property PACKAGE_PIN U19 [get_ports {pout[2]}]
18 set_property PACKAGE_PIN E19 [get_ports {pout[1]}]
19 set_property PACKAGE_PIN U16 [get_ports {pout[0]}]
20 set_property IOSTANDARD LVCMOS33 [get_ports clk]
21 set_property IOSTANDARD LVCMOS33 [get_ports rst]
22 set_property IOSTANDARD LVCMOS33 [get_ports sin]
23 set_property IOSTANDARD LVCMOS33 [get_ports sout]
24 set_property PACKAGE_PIN W5 [get_ports clk]
25 set_property PACKAGE_PIN U1 [get_ports rst]
26 set_property PACKAGE_PIN R2 [get_ports {mode[1]}]
27 set_property PACKAGE_PIN W2 [get_ports sin]
28 set_property PACKAGE_PIN L1 [get_ports sout]
```

**Listing 32:** Constraints for USR

- W5 is clock generator
- T1, R2, W17, W16, V16, V17, U1, W2 are switches
- V19, U19, E19, U16, L1 are LEDs.

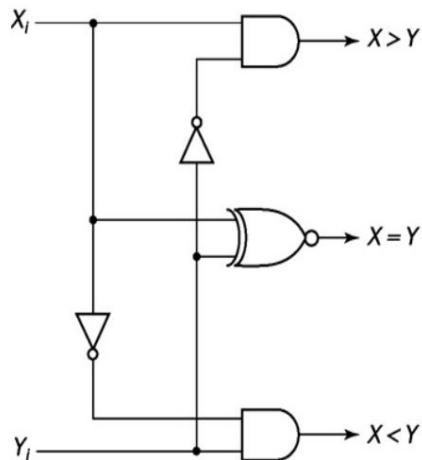
**Note:** W5 is a clock of 100MHz so the LEDs switching might occur too fast for it to make any difference. Clock divider is to implemented to divide the clock from 100MHz to say 1Hz to actually observe the output of sequential circuits in LED and the implementation of clock divider is left as an exercise to the reader.

## 6 | Experiment 6 : Comparators

### 6.1 | Theory

#### 6.1.1 | 1 bit comparator

A comparator is a combinational circuit that compares two binary numbers and determines their relative magnitudes. It outputs signals indicating whether one number is greater than, less than, or equal to the other. For example: 1 bit comparator block diagram / truth table is as follows:



**Figure 6.1:** Block diagram for 1 bit comparator

A	B	$A < B$	$A = B$	$A > B$
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

**Table 6.1:** Truth Table - 1 bit magnitude comparator

### 6.2 | Verilog code

```

1 module comparator_1bit (
2     input A,
3     input B,
4     output A_gt_B,
5     output A_eq_B,
6     output A_lt_B
7 );
8
9     assign A_gt_B = A & ~B;
10    assign A_eq_B = ~(A ^ B);

```

```

11 assign A_lt_B = ~A & B;
12
13 endmodule

```

**Listing 33:** 1 bit comparator Module

## 6.3 | Testbench code

```

1 module tb_comparator_1bit();
2   reg A, B;
3   wire A_gt_B, A_eq_B, A_lt_B;
4   comparator_1bit DUT(A, B, A_gt_B, A_eq_B, A_lt_B);
5
6   initial
7     begin
8       #10; A = 0; B = 0;
9       #10; A = 0; B = 1;
10      #10; A = 1; B = 0;
11      #10; A = 1; B = 1;
12      $finish;
13   end
14
15 endmodule

```

**Listing 34:** 1 bit comparator testbench

## 6.4 | Implementation

### 6.4.1 | Constraints

```

1 set_property IOSTANDARD LVCMOS33 [get_ports A]
2 set_property IOSTANDARD LVCMOS33 [get_ports A_gt_B]
3 set_property IOSTANDARD LVCMOS33 [get_ports A_eq_B]
4 set_property IOSTANDARD LVCMOS33 [get_ports A_lt_B]
5 set_property IOSTANDARD LVCMOS33 [get_ports B]
6 set_property PACKAGE_PIN V16 [get_ports A]
7 set_property PACKAGE_PIN V17 [get_ports B]
8 set_property PACKAGE_PIN U19 [get_ports A_eq_B]
9 set_property PACKAGE_PIN E19 [get_ports A_gt_B]
10 set_property PACKAGE_PIN U16 [get_ports A_lt_B]

```

**Listing 35:** Constraints for 1 bit comparator

- V17, V16 are switches
- U19, E19, U16 are LEDs.

## 7 | Experiment 7 : Up/Down counters

### 7.1 | Theory

#### 7.1.1 | up / down counter

An Up/Down Counter is a sequential digital circuit that counts in either ascending (up) or descending (down) order based on a control input. For example: up/down counter block diagram is as follows

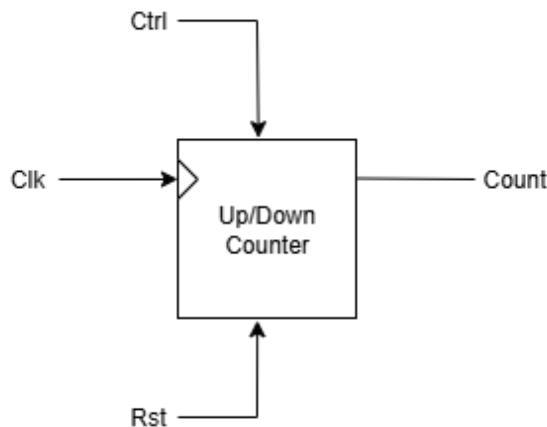


Figure 7.1: Block diagram for up/down counter

### 7.2 | Verilog code

```
1 module counter_up_down(
2     input clk,rst,ctrl,
3     output reg [3:0] count
4 );
5
6     always @ (posedge clk)
7     begin
8         //count <=0;
9         if(rst)
10             count <= 4'd0;
11         else
12             begin
13                 if(ctrl)begin
14                     count <= count + 1'b1; end
15                 else
16                     begin
17                         count <= count - 1'b1; end
18                 end
19             end
20 endmodule
```

Listing 36: up/down counter Module

### 7.3 | Testbench code

```
1 module counter_up_down_tb;
2 reg clk,rst,ctrl;
3 wire [3:0] count;
4
5 counter_up_down c1(clk,rst,ctrl,count);
6
7 initial begin
8 clk=1'b0;
9 //rst=1'b1;
10 end
11
12 always #5 clk = ~clk;
13
14 initial begin
15 ctrl = 0; #6;
16 ctrl = 1; #40;
17 ctrl = 0; #30;
18 ctrl=1;
19 end
20
21 initial begin
22
23 rst = 1'b1; #7;
24 rst = 1'b0; #40;
25 rst = 1'b0;
26
27 #100;
28 $finish;
29 end
30 endmodule
```

**Listing 37:** counter testbench

### 7.4 | Implementation

#### 7.4.1 | Constraints

```
1 set_property IOSTANDARD LVCMOS33 [get_ports clk]
2 set_property IOSTANDARD LVCMOS33 [get_ports rst]
3 set_property IOSTANDARD LVCMOS33 [get_ports ctrl]
4 set_property IOSTANDARD LVCMOS33 [get_ports {count[3]}]
5 set_property IOSTANDARD LVCMOS33 [get_ports {count[2]}]
6 set_property IOSTANDARD LVCMOS33 [get_ports {count[1]}]
7 set_property IOSTANDARD LVCMOS33 [get_ports {count[0]}]
8 set_property PACKAGE_PIN V19 [get_ports {count[3]}]
```



```
9 set_property PACKAGE_PIN U19 [get_ports {count[2]}]
10 set_property PACKAGE_PIN E19 [get_ports {count[1]}]
11 set_property PACKAGE_PIN U16 [get_ports {count[0]}]
12 set_property PACKAGE_PIN W5 [get_ports clk]
13 set_property PACKAGE_PIN R2 [get_ports ctrl]
14 set_property PACKAGE_PIN T1 [get_ports rst]
```

**Listing 38:** Constraints for up/down counter

- W5 is clock generator
- R2, T1 are switches
- V19, U19, E19, U16 are LEDs.

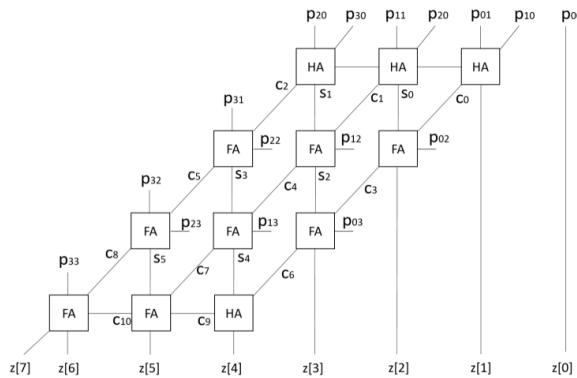
**Note:** W5 is a clock of 100MHz so the LEDs switching might occur too fast for it to make any difference. Clock divider is to implemented to divide the clock from 100MHz to say 1Hz to actually observe the output of sequential circuits in LED and the implementation of clock divider is left as an exercise to the reader.

## 8 | Experiment 8 : Array Multiplier

### 8.1 | Theory

#### 8.1.1 | Array Multiplier

Array multiplier is the traditional multiplier architecture that is used to compute the product of two numbers using a pen, paper. It takes in the two operands, namely the Multiplicand and the Multiplier and gives the product of them as the result.



**Figure 8.1:** Block diagram for Array Multiplier

### 8.2 | Verilog code

#### 8.2.1 | Array Multiplier

```

1 module array_multiplier (
2     input  [3:0] A, B,
3     output [7:0] z
4 );
5
6 // Partial product matrix
7 wire p[3:0][3:0];
8
9 // Intermediate carries and sums
10 wire [10:0] c;
11 wire [5:0] s;
12
13 // Partial product generation using AND gates
14 genvar i, j;
15 generate
16     for (i = 0; i < 4; i = i + 1) begin : row
17         for (j = 0; j < 4; j = j + 1) begin : col
18             and pp_gen(p[i][j], A[i], B[j]);
19         end
20     end
21 endgenerate

```

```
22 // Assign LSB directly to the output
23 assign z[0] = p[0][0];
24
25
26 half_adder ha0 (p[0][1], p[1][0], z[1], c[0]);
27 half_adder ha1 (p[1][1], p[2][0], s[0], c[1]);
28 half_adder ha2 (p[2][1], p[3][0], s[1], c[2]);
29
30 full_adder fa0 (p[0][2], c[0], s[0], z[2], c[3]);
31 full_adder fa1 (p[1][2], c[1], s[1], s[2], c[4]);
32 full_adder fa2 (p[2][2], c[2], p[3][1], s[3], c[5]);
33
34 full_adder fa3 (p[0][3], c[3], s[2], z[3], c[6]);
35 full_adder fa4 (p[1][3], c[4], s[3], s[4], c[7]);
36 full_adder fa5 (p[2][3], c[5], p[3][2], s[5], c[8]);
37
38 half_adder ha3 (c[6], s[4], z[4], c[9]);
39 full_adder fa6 (c[9], c[7], s[5], z[5], c[10]);
40 full_adder fa7 (c[10], c[8], p[3][3], z[6], z[7]);
41
42 endmodule
43
44 // Full Adder
45 module full_adder (
46     input  a, b, cin,
47     output s0, c0
48 );
49     assign s0 = a ^ b ^ cin;
50     assign c0 = (a & b) | (b & cin) | (a & cin);
51 endmodule
52
53 // Half Adder
54 module half_adder (
55     input  a, b,
56     output s0, c0
57 );
58     assign s0 = a ^ b;
59     assign c0 = a & b;
60 endmodule
```

**Listing 39:** Array Multiplier Module

## 8.3 | Testbench code

### 8.3.1 | Array Multiplier

```
1 module array_multiplier_tb();
```

```

2
3 reg [3:0] A, B;
4 wire [7:0] z;
5
6 array_multiplier uut (
7   .A(A),
8   .B(B),
9   .z(z)
10);
11
12 initial begin
13   $display("A = %d, B = %d, A*B = %d", A, B, z);
14   A = 4'd3; B = 4'd2; #10; // 3 * 2 = 6
15   A = 4'd5; B = 4'd4; #10; // 5 * 4 = 20
16   A = 4'd7; B = 4'd3; #10; // 7 * 3 = 21
17   A = 4'd9; B = 4'd8; #10; // 9 * 8 = 72
18   $finish;
19 end
20 endmodule

```

**Listing 40:** Array Multiplier testbench

## 8.4 | Implementation

### 8.4.1 | Constraints

```

1 ## Input Switches for A[3:0] (SW0-SW3)
2 set_property PACKAGE_PIN V17 [get_ports {A[0]}]
3 set_property PACKAGE_PIN V16 [get_ports {A[1]}]
4 set_property PACKAGE_PIN W16 [get_ports {A[2]}]
5 set_property PACKAGE_PIN W17 [get_ports {A[3]}]
6
7 set_property IOSTANDARD LVCMOS33 [get_ports {A[*]}]
8
9 ## Input Switches for B[3:0] (SW4-SW7)
10 set_property PACKAGE_PIN W15 [get_ports {B[0]}]
11 set_property PACKAGE_PIN V15 [get_ports {B[1]}]
12 set_property PACKAGE_PIN W14 [get_ports {B[2]}]
13 set_property PACKAGE_PIN W13 [get_ports {B[3]}]
14
15 set_property IOSTANDARD LVCMOS33 [get_ports {B[*]}]
16 ## Output LEDs for z[7:0] (LEDO-LED7)
17 set_property PACKAGE_PIN U16 [get_ports {z[0]}];
18 set_property PACKAGE_PIN E19 [get_ports {z[1]}];
19 set_property PACKAGE_PIN U19 [get_ports {z[2]}];
20 set_property PACKAGE_PIN V19 [get_ports {z[3]}];
21 set_property PACKAGE_PIN W18 [get_ports {z[4]}];

```

```
22 set_property PACKAGE_PIN U15 [get_ports {z[5]}] ;  
23 set_property PACKAGE_PIN U14 [get_ports {z[6]}] ;  
24 set_property PACKAGE_PIN V14 [get_ports {z[7]}] ;  
25 set_property IO_STANDARD LVCMOS33 [get_ports {z[*]}]
```

**Listing 41:** Array Multiplier Implementation

- All the pins allocated to A and B are switches
- All the pins allocated to z are LEDs.

## 9 | Experiment 9 : RAM

### 9.1 | Theory

#### 9.1.1 | RAM

RAM (Random Access Memory) is a storage element that supports both read and write operations at specified addresses using control signals like clk, we, and reset. It allows direct access to any memory location via an address bus, making it suitable for fast and flexible data handling.

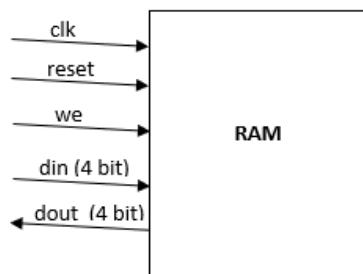


Figure 9.1: Block diagram for RAM

### 9.2 | Verilog code

#### 9.2.1 | RAM

```
1 module ram(
2     input clk, we, reset,
3     input [2:0] addr,
4     input [3:0] din,
5     output reg [3:0] dout
6 );
7
8     reg [3:0] mem [0:7] ;
9
10    always @ (posedge clk) begin
11        if (reset) begin
12            dout <= 8'd0;
13        end
14        else if (we) begin
15            mem [addr] <= din;
16        end
17        else begin
18            dout <= mem [addr];
19        end
20    end
21 endmodule
```

---

**Listing 42:** RAM Module**9.3 | Testbench code****9.3.1 | RAM**

```
1 module ram_tb();
2   reg clk, we, reset;
3   reg [2:0] addr;
4   reg [3:0] din;
5   wire [3:0] dout;
6
7
8   ram r1(clk, we, reset, addr, din, dout);
9
10  always #5 clk = ~clk;
11
12 initial begin
13   clk = 0;
14   we = 0;
15   reset = 1;
16   addr = 3'b000;
17   din = 4'b0000;
18
19
20   #12;
21   reset = 0;
22
23   //Write phase
24   we = 1;
25
26
27   addr = 3'b000; din = 4'hA; #10; // Write A at addr 0
28   addr = 3'b001; din = 4'hB; #10; // Write B at addr 1
29   addr = 3'b010; din = 4'hC; #10; // Write C at addr 2
30   addr = 3'b011; din = 4'hD; #10; // Write D at addr 3
31
32   // Read phase
33   we = 0;
34
35   addr = 3'b000; #10; // Expect A
36   addr = 3'b001; #10; // Expect B
37   addr = 3'b010; #10; // Expect C
38   addr = 3'b011; #10; // Expect D
39
40   // Finish simulation
```

```

41      $finish;
42
43  end
44 endmodule

```

**Listing 43:** RAM testbench

## 9.4 | Implementation

### 9.4.1 | Constraints

```

## Clock
set_property PACKAGE_PIN W5 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]

## Reset
set_property PACKAGE_PIN V17 [get_ports reset]
set_property IOSTANDARD LVCMOS33 [get_ports reset]

## Write Enable
set_property PACKAGE_PIN V16 [get_ports we]
set_property IOSTANDARD LVCMOS33 [get_ports we]

## Address [2:0]
set_property PACKAGE_PIN W16 [get_ports addr[2]]
set_property PACKAGE_PIN W17 [get_ports addr[1]]
set_property PACKAGE_PIN W15 [get_ports addr[0]]
set_property IOSTANDARD LVCMOS33 [get_ports {addr[*]}]

## Data Input [3:0]
set_property PACKAGE_PIN V15 [get_ports din[3]]
set_property PACKAGE_PIN W14 [get_ports din[2]]
set_property PACKAGE_PIN W13 [get_ports din[1]]
set_property PACKAGE_PIN V2 [get_ports din[0]]
set_property IOSTANDARD LVCMOS33 [get_ports {din[*]}]

## Data Output [3:0]
set_property PACKAGE_PIN U16 [get_ports dout[3]]
set_property PACKAGE_PIN E19 [get_ports dout[2]]
set_property PACKAGE_PIN U19 [get_ports dout[1]]
set_property PACKAGE_PIN W18 [get_ports dout[0]]
set_property IOSTANDARD LVCMOS33 [get_ports {dout[*]}]

```

**Listing 44:** Array Multiplier Implementation

- W5 is clock generator

- All the pins allocated to enable, reset, input and address are switches
- All the pins allocated to output are LEDs.

**Note:** W5 is a clock of 100MHz so the LEDs switching might occur too fast for it to make any difference. Clock divider is to implemented to divide the clock from 100MHz to say 1Hz to actually observe the output of sequntial circuits in LED and the implementation of clock divider is left as an exercise to the reader.