# EXPERIMENT NO : 01

## Half Adder, Full Adder using three modelling styles

## Aim :

Develop and verify a Verilog code, exercise a testbench, synthesize, and do the initial timing verification with gate level simulation.

## Half Adder :

### Theory:

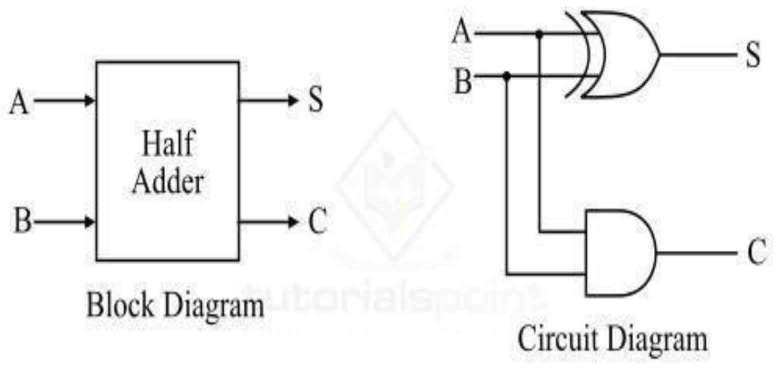


Figure 1 - Half Adder

A half adder is a digital circuit that adds two single binary digits and produces a sum and a carry output. It has two inputs, typically labeled A and B, and two outputs: the sum (S) and the carry (C). The sum is calculated using the XOR operation ($S = A \oplus B$), while the carry is calculated using the AND operation ($C = A \cdot B$).

### Truth Table :

Half Adder Truth table

| INPUTS | | OUTPUTS | |
|---|---|---|---|
| A | B | SUM | CARRY |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

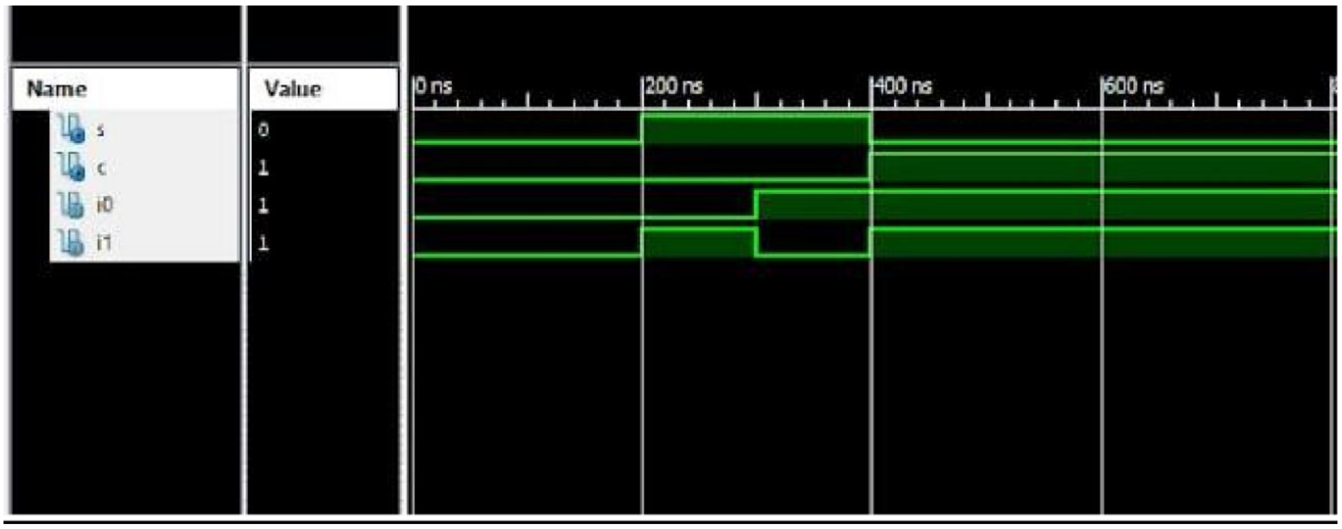
## Half Adder:-

```
module hsfh(a,b, sum,carry);
   input a,b;
   output reg sum,carry;
          always@(a,b)
          begin
          if(a==b)
          begin
          sum=0;
          carry=b;
          end
          else
          begin
          sum=1;
          carry=0;
          end
          end
endmodule
```

## Test Bench :-

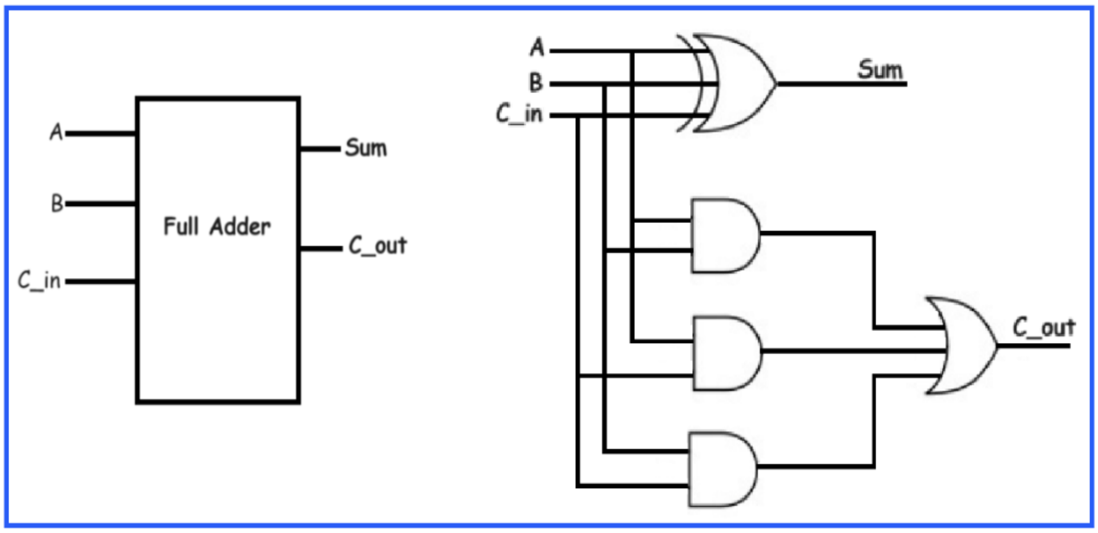
```
module hfgj_v;

        // Inputs
        reg a;
        reg b;

        //   Outputs
        wire   sum;
        wire carry;

        // Instantiate the Unit Under Test (UUT)
        hsfh uut (
                .a(a),
                .b(b),
                .sum(sum),
                .carry(carry)
        );

        initial begin
                // Initialize Inputs
                a = 0;
                b = 0;

                // Wait 100 ns for global reset to finish
                #100;
                a = 0;
                b = 1;
                #100;
                a =1;
                b =0;
                #100;
                a =1;
                b = 1;
                #100;

                // Add stimulus here
        end
endmodule
```

## Output :-



## Full Adder :

### Theory :



A full adder is a digital circuit that adds three binary digits: two significant bits and a carry-in bit from a previous addition. It produces a sum and a carry-out, using the equations S = A ⊕ B ⊕ Cin for the sum and C = (A · B) + (Cin · (A ⊕ B)) for the carry-out. This allows it to handle multi-bit binary addition effectively.

### Truth Table :

| Inputs | | | Outputs | |
|---|---|---|---|---|
| A | B | $C_{in}$ | 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 |

## Full Adder :-

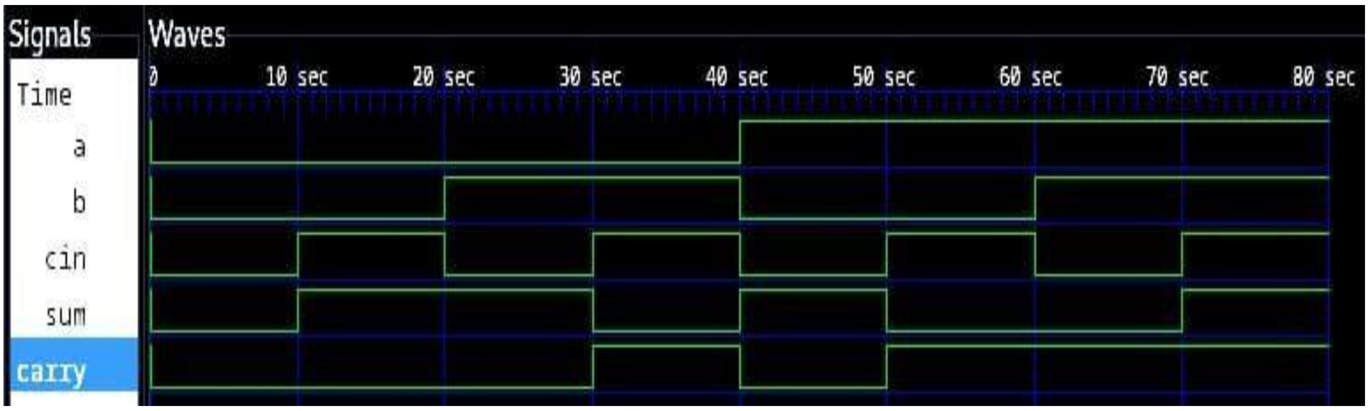
```
module fgvc(a,b,cin,sum,carry);
   input a,b,cin;
   output reg sum,carry;
          always@(a,b,cin)
          begin
          if(a==b)
          begin
          sum=cin;
          carry=b;
          end
          else
          begin
          sum=~cin;
          carry=cin;
          end
          end
endmodule
```

## Test Bench :-

```
module gff_v;

        // Inputs
        reg a;
        reg b;
        reg cin;

        //   Outputs
        wire   sum;
        wire carry;
        // Instantiate the Unit Under Test (UUT)
        fgvc uut (
                .a(a),
                .b(b),
                .cin(cin),
                .sum(sum),
                .carry(carry)
        );

        initial begin
                // Initialize Inputs
                a = 0;
                b = 0;
                cin = 0;
```

```
		// Wait 100 ns for global reset to finish
		#100;
		a = 0;
		b = 0;
		cin =1;
		#100;
		a = 0;
		b = 1;
		cin =0;
		#100;
		// Add stimulus here

	end
endmodule
```

**Output :-**



# EXPERIMENT NO : 02

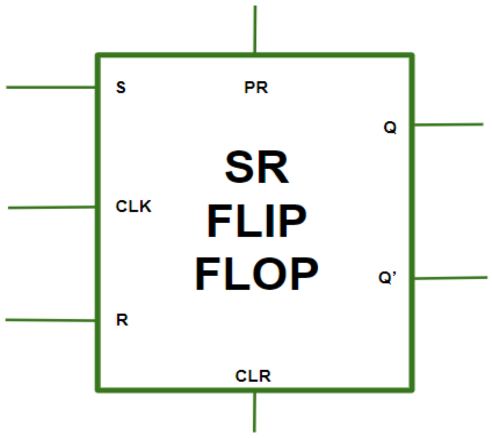
## Flip Flops

## AIM :

Develop and verify a Verilog code, exercise a testbench, synthesize, and do the initial timing verification with gate level simulation.

## SR Flip Flops :

### Theory :



An SR flip-flop is a bistable multivibrator that has two inputs, Set (S) and Reset (R), and two outputs, Q and $\bar{Q}$. When S is activated, Q is set to high, while activating R sets Q to low; if both inputs are low, the output retains its previous state. This flip-flop is used for storing binary information and is fundamental in memory storage and sequential logic circuits.

### Truth Table :

| S | R | $Q_{n+1}$ |
|---|---|---|
| 0 | 0 | $Q_n$(No Change) |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | x |

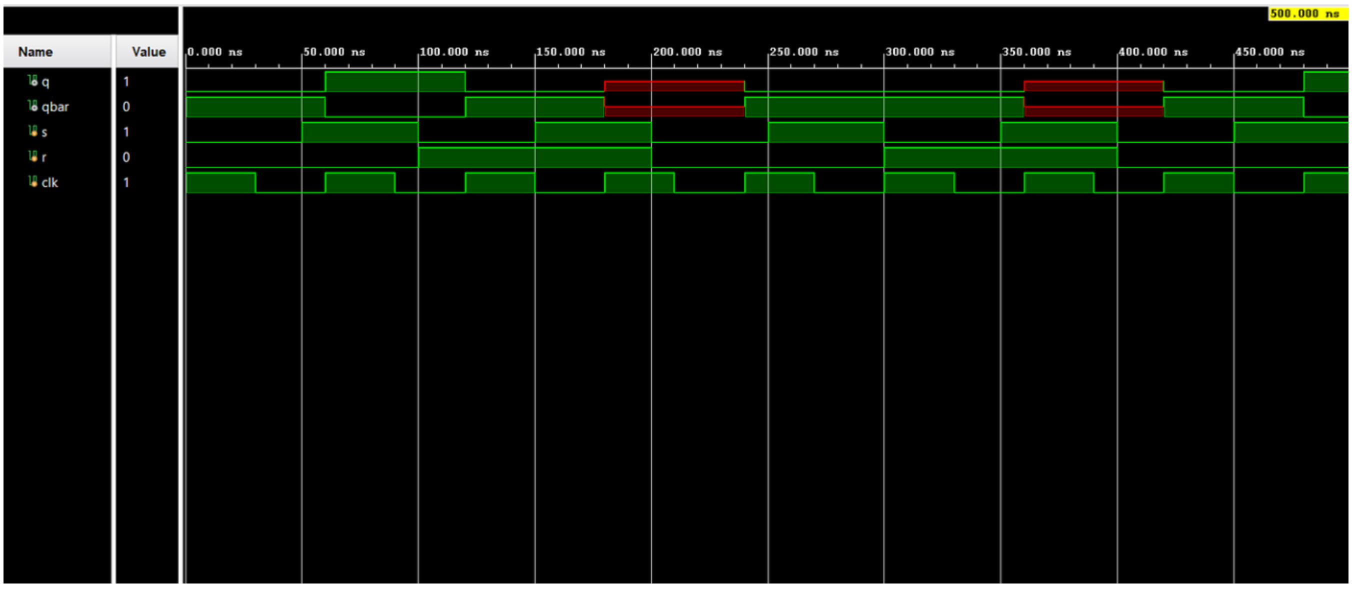
## SR Flip-Flop :-

```
module rfdc(clk,s,r, q,qb);
   input clk,s,r;
   output reg q,qb;
        always@(posedge clk)
        begin
        if(s==0 &&r==1)
        q=1'b0;
        else if(s==1 && r==0)
        q=1'b1;
        qb=~q;
        end
endmodule
```

## Test Bench :-

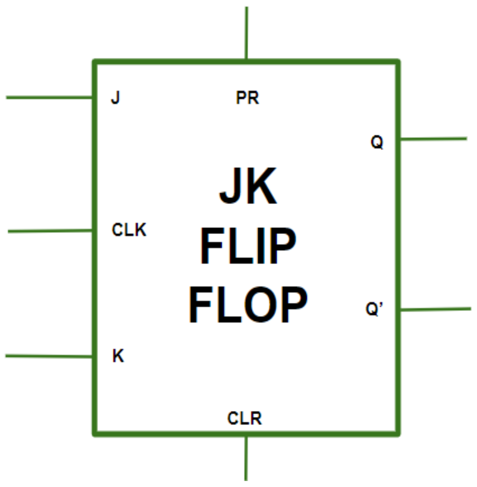
```
module thfg_v;

        // Inputs
        reg clk;
        reg s;
        reg r;

        // Outputs
        wire q;
        wire qb;

        // Instantiate the Unit Under Test (UUT)
        rfdc uut (
                .clk(clk),
                .s(s),
                .r(r),
                .q(q),
                .qb(qb)
        );
        always#10 clk=~clk;


        initial begin
                // Initialize Inputs
                clk=0;
                s=0;
                r=0;
// Wait 100 ns for global reset to finish
                #100;
                clk=1;
                s=1;
                r=0;
                #100;
                clk=0;
                s=0;

                r=1;
                #100;
                // Add stimulus here
        end
endmodule
```

## Output :-



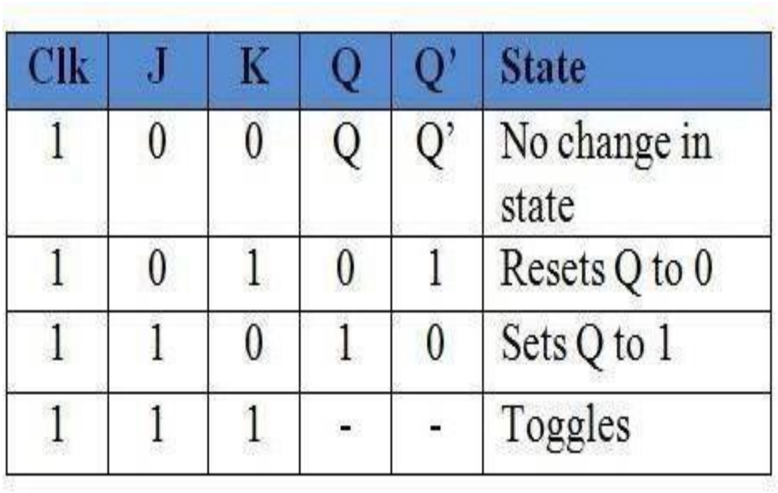
## JK Flip Flops:

### Theory :



A JK flip-flop is a type of bistable multivibrator with two inputs, J and K, and two outputs, Q and $\bar{Q}$ . It toggles its output state when both J and K are high, while setting Q high occurs when J is high and K is low, and resetting Q occurs when J is low and K is high. This versatility allows JK flip-flops to be used in counters and various sequential logic applications.

### Truth Table :



| Clk | J | K | Q | Q' | State |
|---|---|---|---|---|---|
| 1 | 0 | 0 | Q | Q' | No change in state |
| 1 | 0 | 1 | 0 | 1 | Resets Q to 0 |
| 1 | 1 | 0 | 1 | 0 | Sets Q to 1 |
| 1 | 1 | 1 | - | - | Toggles |

**JK Flip-Flop :-**

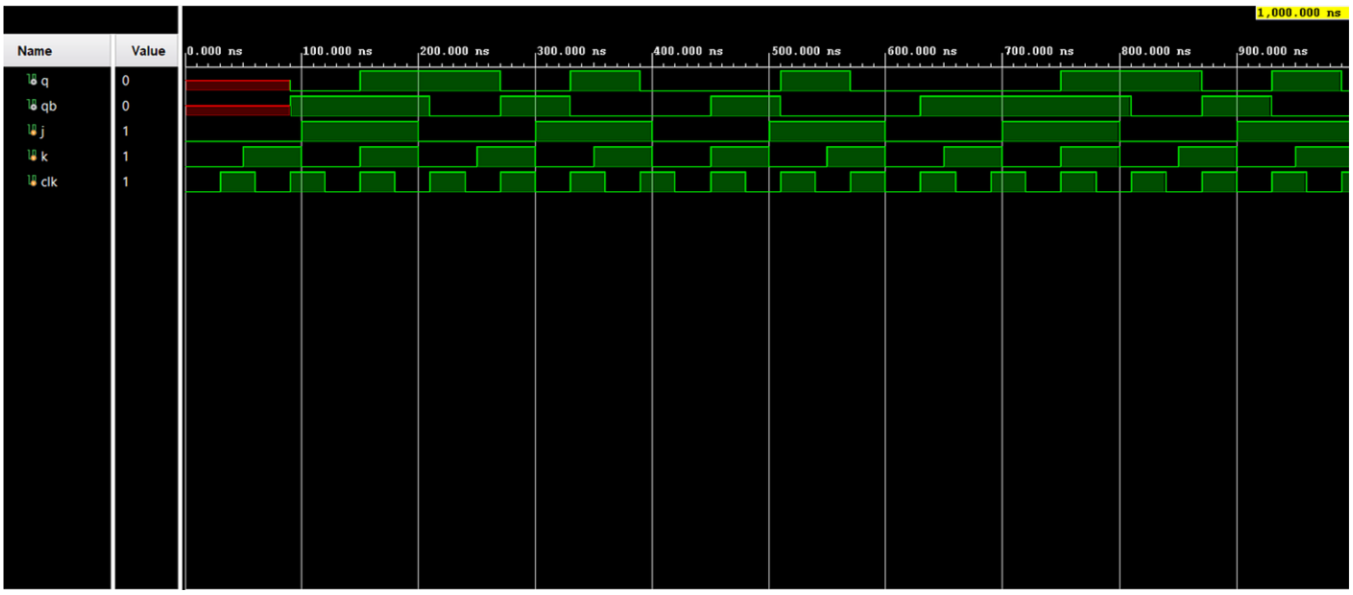
```
module frdvc(clk,j,k, q,qb);
   input clk,j,k;
   output  reg q,qb;
         always@(posedge clk)
         begin
         if(j==0&&k==0)
         q=0;
         else if(j==0&&k==0)
         q=1'b0;
         else if(j==1&&k==1)
         q=1'b1;
         else
         q=~q;
         qb=~q;
         end
endmodule
```

## **Test Bench :-**

```
module trgf_v;
        // Inputs
        reg clk;
        reg j;
        reg k;
        // Outputs
        wire q;
        wire qb;
        // Instantiate the Unit Under Test (UUT)
        frdvc uut (
                .clk(clk),
                .j(j),
                .k(k),
                .q(q),
                .qb(qb)
        );
        always#10 clk=~clk;
        initial begin
                // Initialize Inputs
                clk = 0;
                j = 0;
                k = 0;

                // Wait 100 ns for global reset to finish
                #100;
```

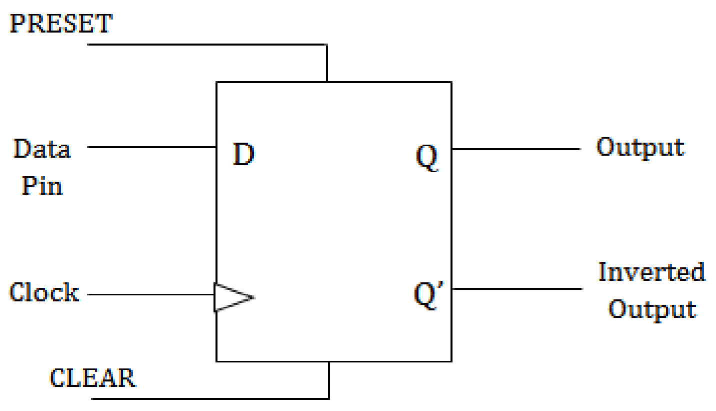
```
                    clk = 1;
                    j = 0;
                    k = 1;
                    #100;
                    clk = 1;
                    j = 1;
                    k = 1;
                    #100;
                    // Add stimulus here
    end
endmodule
```

## **Output :-**



## D Flip Flops :

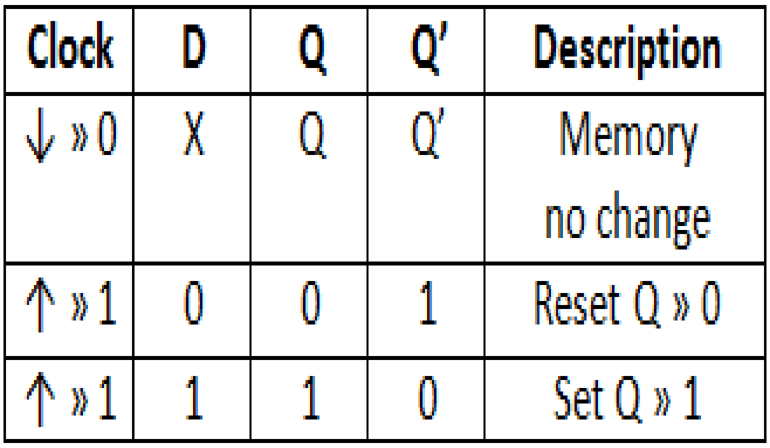
## Theory :



**Symbol: D Flip-flop**

A D flip-flop is a bistable device that captures the value of the input data (D) on the rising (or falling) edge of a clock signal, transferring it to the output Q. When the clock edge occurs, Q takes on the value of D, while $\overline{Q}$ becomes its complement. This simplicity makes D flip-flops essential for data storage, timing applications, and building registers in digital circuits.

## Truth Table :



| Clock | D | Q | Q' | Description |
|---|---|---|---|---|
| ↓ » 0 | X | Q | Q' | Memory no change |
| ↑ » 1 | 0 | 0 | 1 | Reset Q » 0 |
| ↑ » 1 | 1 | 1 | 0 | Set Q » 1 |

## D Flip-Flop :-

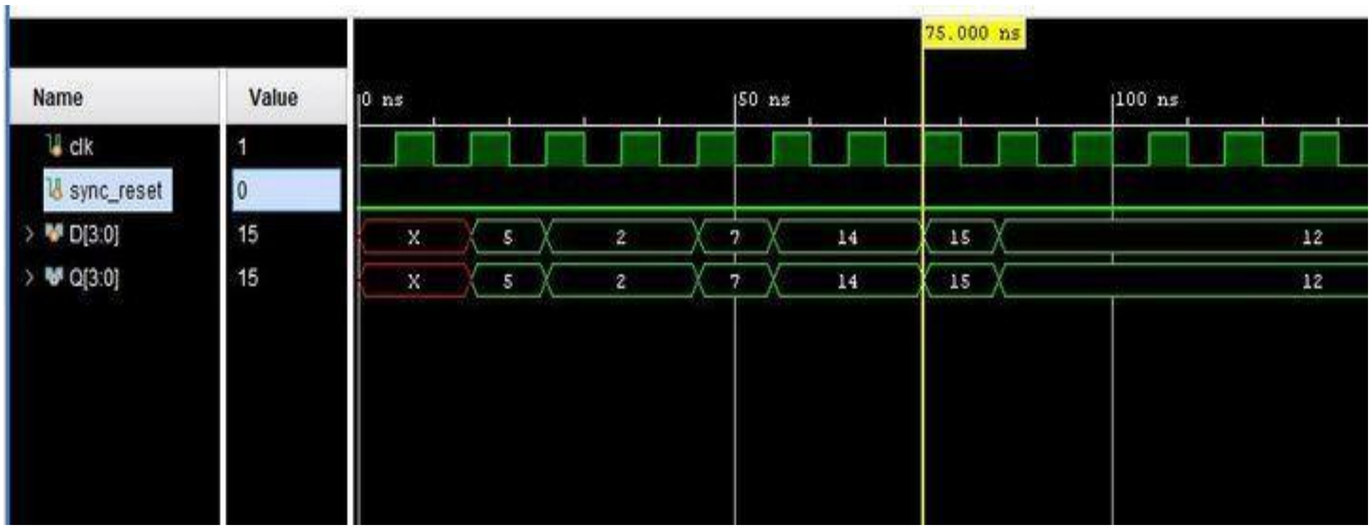
```
module gtfbv(clk,rst_n,d, q);
   input clk,rst_n,d;
   output reg q;
            always@(posedge clk or negedge rst_n)
            begin
            if(!rst_n)q<=0;
            else
            q<=d;
            end
endmodule
```

## Test Bench :-

```
module yhg_v;

        // Inputs
        reg clk;
        reg rst_n;
        reg d;

        // Outputs
        wire q;

        // Instantiate the Unit Under Test (UUT)
        gtfbv uut (
                .clk(clk),
                .rst_n(rst_n),
                .d(d),
                .q(q)
        );
        always#10 clk=~clk;

        initial begin
                // Initialize Inputs
                clk = 0;
                rst_n = 0;
                d = 0;

                // Wait 100 ns for global reset to finish
                #100;
                clk =1;
                rst_n =1;
                d = 1;
```

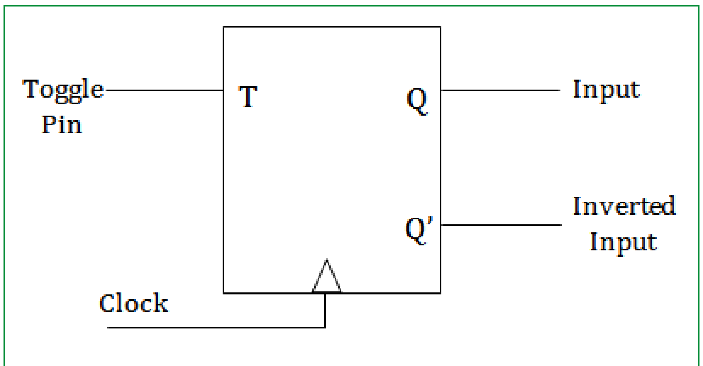
```
                    // Add stimulus here

          end
endmodule
```

## Output :-



## T Flip Flops :

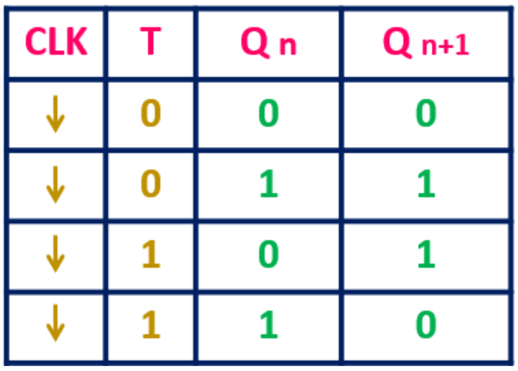
### Theory :



Symbol: T Flip-flop

A T flip-flop, or toggle flip-flop, is a bistable device that changes its output state (toggles) when the T input is high at the clock edge. If T is low, the output remains unchanged. This characteristic makes T flip-flops useful for counting applications and building frequency dividers in digital circuits.

### Truth Table :



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

## T Flip-Flop :-

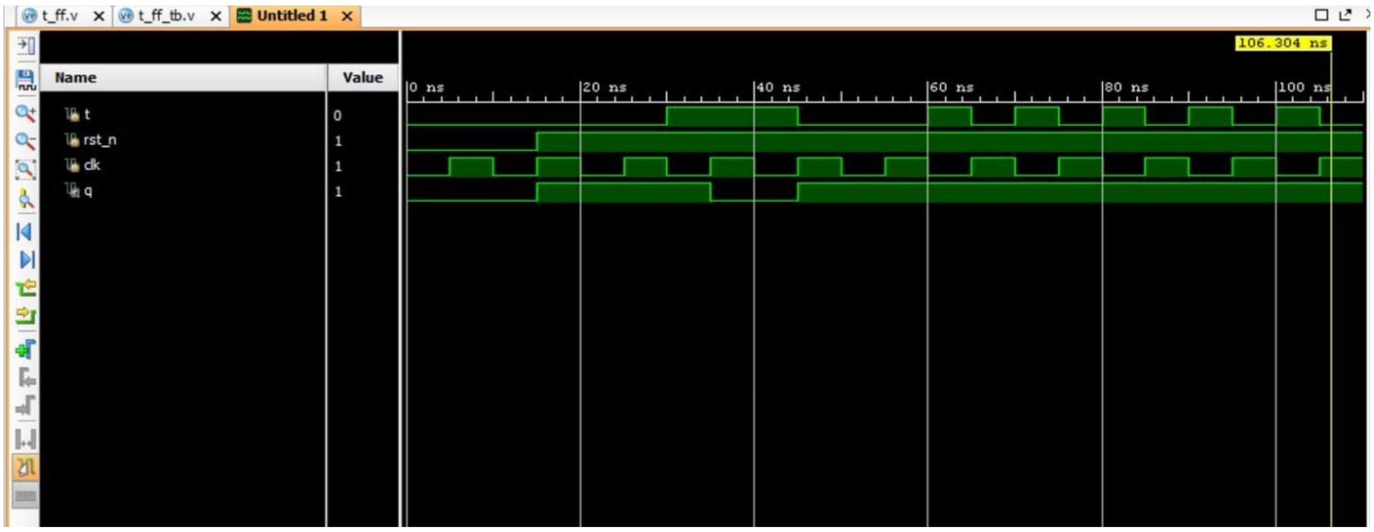
```
module refdvc(clk,rst,t, q);
   input clk,rst,t;
   output reg q;
            always@(posedge clk)
            begin
            if(!rst)
            q<=0;
            else
            if(t)
            q<=~q;
            else
            q<=q;
            end
endmodule
```

## Test Bench :-

```
module fgh_v;

        // Inputs
        reg clk;
        reg rst;
        reg t;

        // Outputs
        wire q;

        // Instantiate the Unit Under Test (UUT)
        refdvc uut (
                .clk(clk),
                .rst(rst),
                .t(t),
                .q(q)
        );
        always#10 clk=~clk;

        initial begin
                // Initialize Inputs
                clk = 0;
                rst = 0;
                t = 0;

                // Wait 100 ns for global reset to finish
                #100;
```

```
            clk =1;
            rst =1;
            t=1;
            #100;


            // Add stimulus here

        end
endmodule
```

## <u>Output :-</u>



| Name | Value |
|---|---|
| t | 0 |
| rst_n | 1 |
| clk | 1 |
| q | 1 |

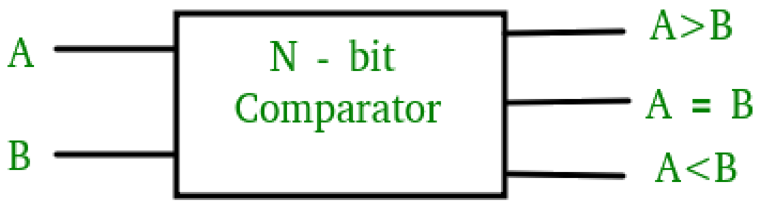
**EXPERIMENT NO : 03**

# N-B it Comparator

## Aim :

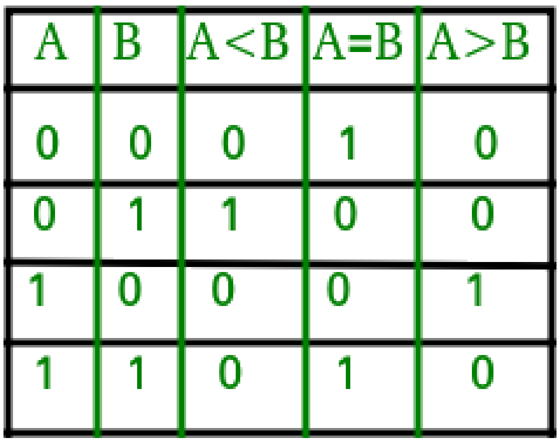
Develop and verify a Verilog code, exercise a testbench, synthesize, and do the initial timing verification with gate level simulation.

## Theory :



An n-bit comparator is a digital circuit that compares two n-bit binary numbers and determines their relative equality or order. It produces outputs indicating whether the first number is less than, equal to, or greater than the second number. Comparators are essential in decision-making circuits, arithmetic operations, and digital systems for sorting and control functions.

## Truth Table :



| 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 |

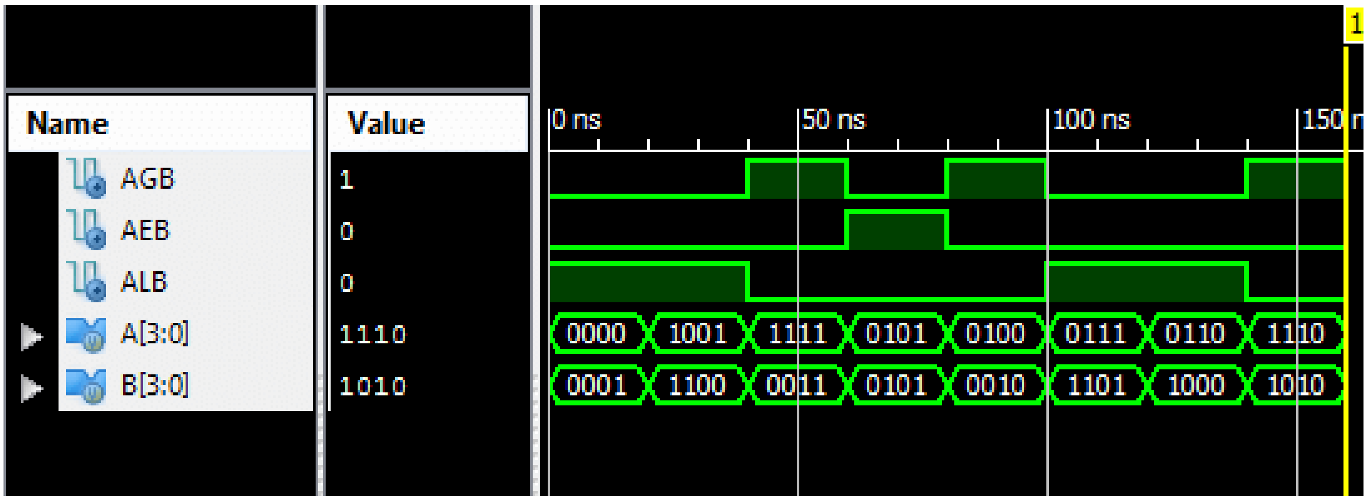
**N-Bit Comparator :-**

```
module fgd(a,b, eq,lt,gt);
   input [3:0] a,b;
   output reg eq,lt,gt;
            always@(a,b)
           begin
           if(a==b)
           begin
           eq=1'b1;
           lt=1'b0;
           gt=1'b0;
           end
           else if(a>b)
           begin
           eq=1'b0;
           lt=1'b0;
           gt=1'b1;
           end
           else
           begin
           eq=1'b0;
           lt=1'b1;
           gt=1'b0;
           end
           end
endmodule
```

**Test Bench :-**

```
module gfvc_v;

	// Inputs
	reg [3:0] a;
	reg [3:0] b;

	// Outputs
	wire eq;
	wire lt;
	wire gt;

	// Instantiate the Unit Under Test (UUT)
	fgd uut (
		.a(a),
		.b(b),
		.eq(eq),
		.lt(lt),
		.gt(gt)
	);

	initial begin
		// Initialize Inputs

		a =1000;
		b =1000;

		// Wait 100 ns for global reset to finish
		#100;
		a =1000;
		b =0001;

		// Wait 100 ns for global reset to finish
		#100;
		a =0001;
		b =1000;

		// Wait 100 ns for global reset to finish
		#100;

		// Add stimulus here

	end
endmodule
```

**Output :-**



## EXPERIMENT NO : 04

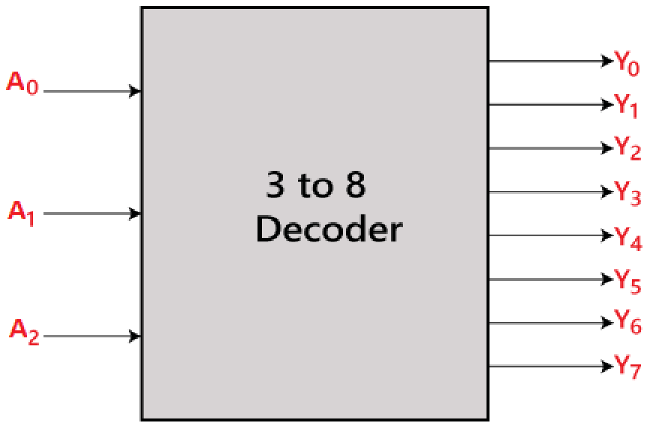
## DECODER AND ENCODER

### Aim:

Develop and verify a Verilog code, exercise a testbench, synthesize, and do the initial timing verification with gate level simulation.

### a) 3:8 Decoder:

#### Theory :



A 3:8 decoder is a digital circuit that converts a 3-bit binary input into one of eight unique outputs, with only one output being activated (high) for each combination of input values. It achieves this by using the input bits to select the corresponding output line based on the binary representation of the input. This type of decoder is commonly used in memory addressing, data demultiplexing, and signal routing in digital systems.

### Truth Table :

| Enable | INPUTS | | | Outputs | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| E | $A_2$ | $A_1$ | $A_0$ | $Y_7$ | $Y_6$ | $Y_5$ | $Y_4$ | $Y_3$ | $Y_2$ | $Y_1$ | $Y_0$ |
| 0 | x | x | x | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

## Decoder :-

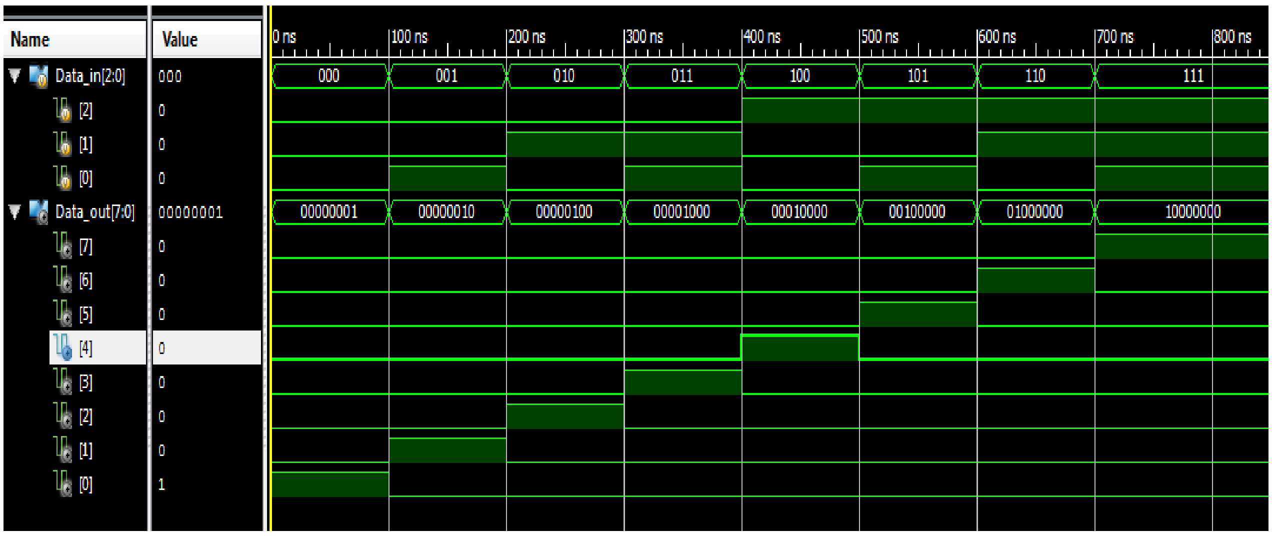
```
Module decoder 3_8(in,out,en);
input[2:0] in;
input reg en;
output[7:0] out;
always@ (in or en)
begin
if (en) begin
out = 8’d0;
case(in)
3’b000:out[0]=1’b1;

3’b001:out[1]=1’b1;

3’b010:out[2]=1’b1;

3’b011:out[3]=1’b1;

3’b100:out[4]=1’b1;

3’b101:out[5]=1’b1;

3’b110:out[6]=1’b1;

3’b111:out[7]=1’b1;

default:out=8’d0;
endcase
end
else
out=8’d0;
end
endmodule
```

## Test Bench:-

```
 module ghnhjn_v;

 eg [2:0]in;

reg en;

wire [7:0] out;

initial begin

in=000;

en=0;
```

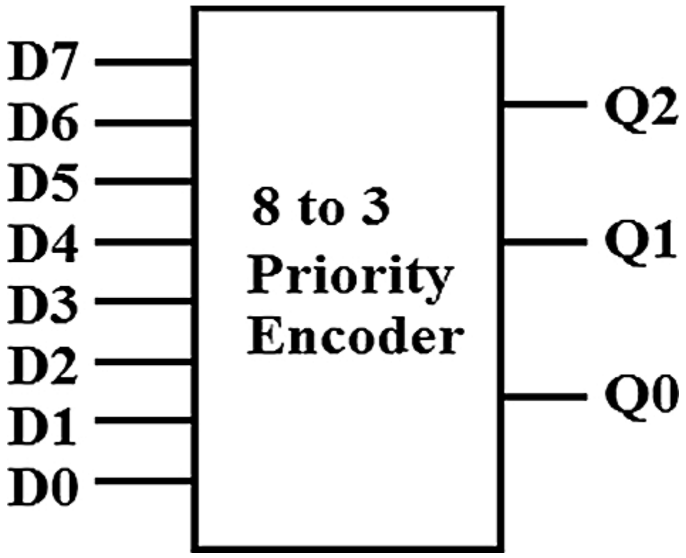
```
#100;
in=000;
en=1;
#100;
in=001;
en=1;
#100;
in=010;
en=1;
#100;
in=011;
en=1;
#100;
in=100;
en=1;
#100;
in=101;
en=1;
#100;
in=110;
en=1;
#100;
in=111;
en=1;
#100
```

## Output :-


Name
Value
Data_in[2:0]
000
[2]
0
[1]
0
[0]
0
Data_out[7:0]
00000001
[7]
0
[6]
0
[5]
0
[4]
0
[3]
0
[2]
0
[1]
0
[0]
1
0 ns
100 ns
200 ns
300 ns
400 ns
500 ns
600 ns
700 ns
800 ns
000
001
010
011
100
101
110
111
00000001
00000010
00000100
00001000
00010000
00100000
01000000
10000000

## b) 8:3 Priority Encoder

### Theory :



An 8:3 priority encoder is a digital circuit that encodes eight input signals into a three-bit output, prioritizing the highest-valued active input. When multiple inputs are active, the encoder outputs the binary representation of the highest priority input while ensuring that lower-priority inputs do not affect the output. This device is commonly used in applications requiring signal management, such as data routing and resource allocation in digital systems.

### Truth Table :

| Digital Inputs | | | | | | | | Binary Output | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ | $Q_2$ | $Q_1$ | $Q_0$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | X | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 1 | X | X | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 1 | X | X | X | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | X | X | X | X | 1 | 0 | 0 |
| 0 | 0 | 1 | X | X | X | X | X | 1 | 0 | 1 |
| 0 | 1 | X | X | X | X | X | X | 1 | 1 | 0 |
| 1 | X | X | X | X | X | X | X | 1 | 1 | 1 |

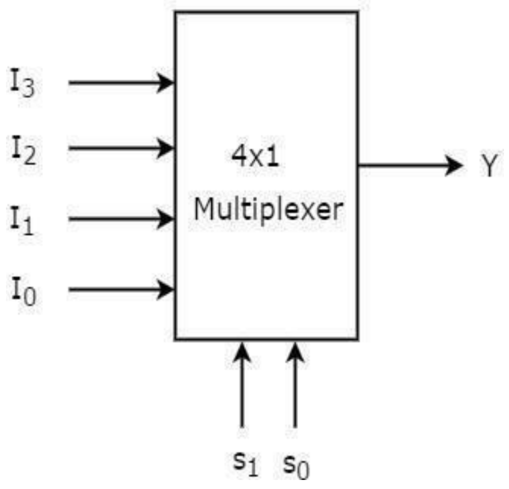
## Priority Encoder:-

```
module Priority_encoder (input [7:0] in; Output reg [2:0]
out;
always @ (in)
begin
caseX(in)
8'b1xxxxxxx : out=3’b111;
8'b01xxxxxx : out = 3’b110;
8' b001xxxxx: out = 3'b101;
8' b0001xxxx : out =3’b100;
8' b00001xxx : out =3’b011;
8’b000001xx: out =3;b0120;
8'b000000x: out =3’b001;
8'b0000001: out =3’b000;
default = out = 0;
endcase
end
endmodule
```

# EXPERIMENT NO : 05

## a) 4:1 multiplexer :-

### Theory :



A 4:1 multiplexer (MUX) is a digital device that selects one of four input signals based on two control or selection lines. It has four data inputs (D0, D1, D2, D3) and two selection lines (S1, S0). The output, denoted as Y, corresponds to one of the inputs depending on the values of the selection lines. For instance, if S1S0 = 00, the output is D0, if S1S0 = 01, the output is D1, and so on. The selection lines determine which input is passed to the output through logic gates. A 4:1 MUX can be implemented using AND, OR, and NOT gates. It is commonly used in applications like data routing, multiplexing in communication systems, and as a selector in digital circuits.

### Truth Table :

| INPUTS | | Output |
|---|---|---|
| $S_1$ | $S_0$ | Y |
| 0 | 0 | $A_0$ |
| 0 | 1 | $A_1$ |
| 1 | 0 | $A_2$ |
| 1 | 1 | $A_3$ |

## <u>4:1 multiplixer:-</u>

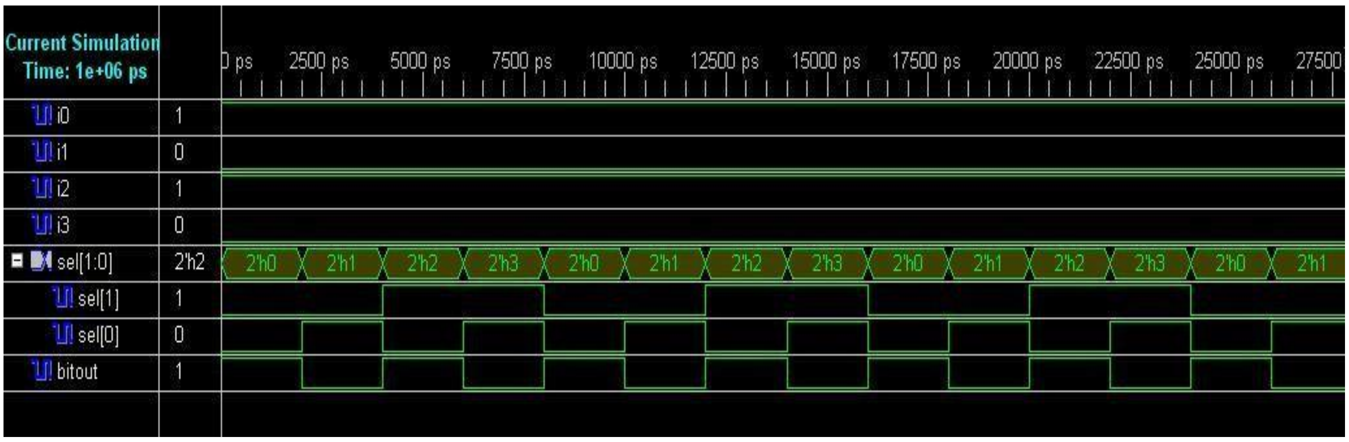
```
module mux(i0,i1,i2,i3,sel,y)
input i0,i1,i2,i3;
input[1:0] sel;
output reg y;
always@(i0 or i1 or i2 or i3 or sel)
begin
case (sel)
2’b00:y=io;
2’b01:y=i1;
2’b10:y=i2;
2’b11:y=i3;
default:y=0;
endcase
end
endmodule
```

## <u>TestBench:-</u>

```
module rrg_v;
reg [1:0] sel;
reg i0,i1,i2,i3;
wire y;
initial begin
i0=0;
i1=0;
i2=0;
i3=0;
sel=00;
#100;
i0=1;
```

```
i1=0;
i2=0;
i3=0;
sel=01;
#100;
i0=0;
i1=1;
i2=0;
i3=0;
sel=10;
#100;
i0=0;
i1=0;
i2=1;
i3=1;
sel=11;
#100;
  end
  endmodule
```

**<u>Output:-</u>**



## b) **1:4 Demultiplexer :-**

### **Theory :**

A 1:4 demultiplexer (DEMUX) is a digital circuit that takes a single input and routes it to one of four output lines based on the values of two selection inputs. It has one data input (D), two selection lines (S1, S0), and four output lines (Y0, Y1, Y2, Y3). The selection lines determine which output the input will be directed to. For example, if S1S0 = 00, the input is routed to Y0; if S1S0 = 01, it goes to Y1, and so on. The demultiplexer works by using logic gates, such as AND gates, to direct the input signal to the correct output. It is commonly used in applications where a single data source needs to be distributed to multiple destinations, such as in communication and data processing systems.

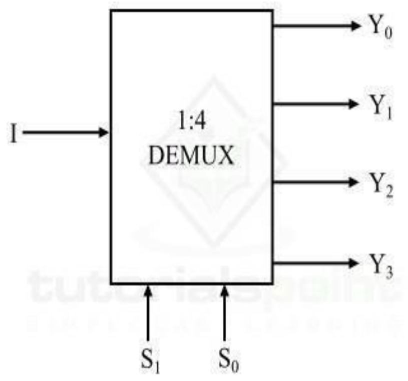
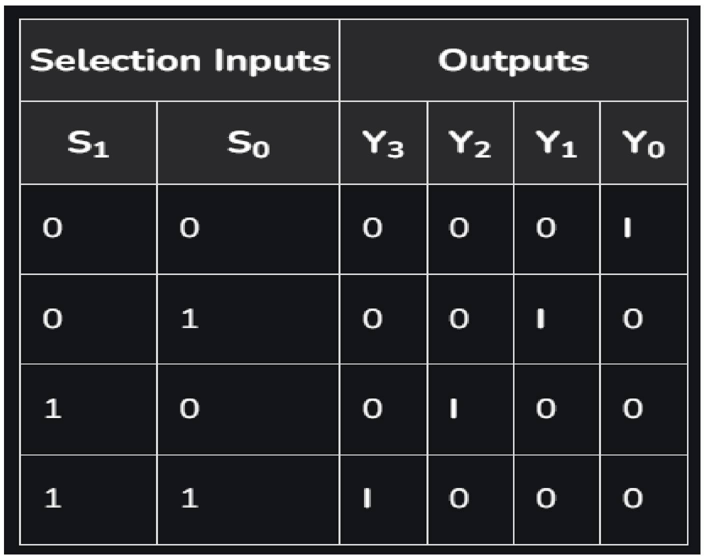


Figure 3 - 1:4 Demultiplexer

### **Truth Table :-**



| Selection Inputs | | Outputs | | | |
|---|---|---|---|---|---|
| $S_1$ | $S_0$ | $Y_3$ | $Y_2$ | $Y_1$ | $Y_0$ |
| 0 | 0 | 0 | 0 | 0 | I |
| 0 | 1 | 0 | 0 | I | 0 |
| 1 | 0 | 0 | I | 0 | 0 |
| 1 | 1 | I | 0 | 0 | 0 |

## Demux[1:4]:-

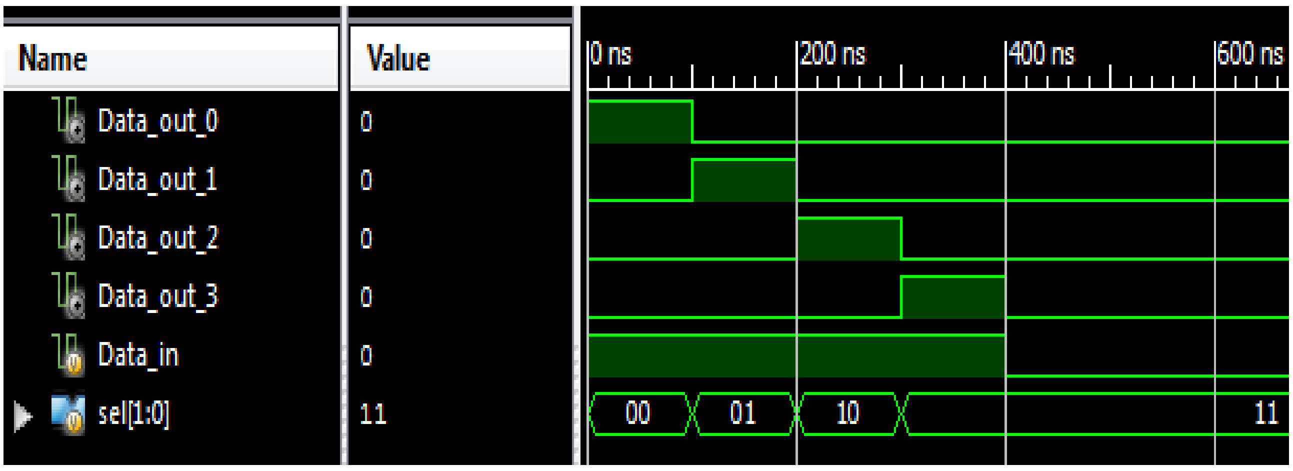
```
module demux(input[1:0] sel,input i,
Output reg (y0,y1,y2,y3);
always@ (sel,i)
begin
case(sel)
2’b00:{y0,y1,y2,y3}={i,3’b0};
2’b01:{y0,y1,y2,y3}={1’b0,i,2’b0};
2’b10:{y0,y1,y2,y3}={2’b0,i,1’b0};
2’b11:{y0,y1,y2,y3}={3’b0,i};
default:{y0,y1,y2,y3}={0,0,0,0};
endcase
end
endmodule
```

## TestBench:-

```
module ggh_k;
reg[1:0]sel;
reg[1:0] i;
wire y0;
wire y1;
wire y2;
wire y3;
sdf uut(
.sel(sel),
.i(i),
.y0(y0),
.y1(y1),
.y2(y2),
.y3(y3),);
```

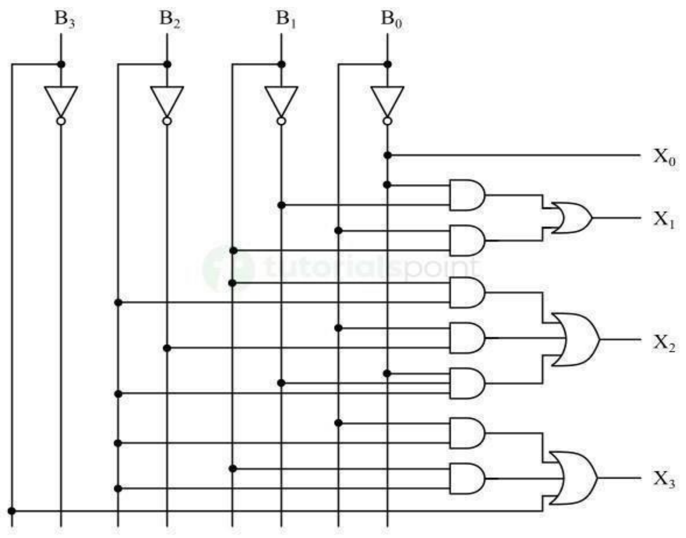
```
Initial begin

Sel=00;
i=1;
#100;
Sel=01;
i=1;
#100;
Sel=10;
i=1;
#100;
Sel=11;
i=1;
#100;
 end
 endmodule
```

**Output:-**



## c) BCD to Excess 3 converter and vice versa :

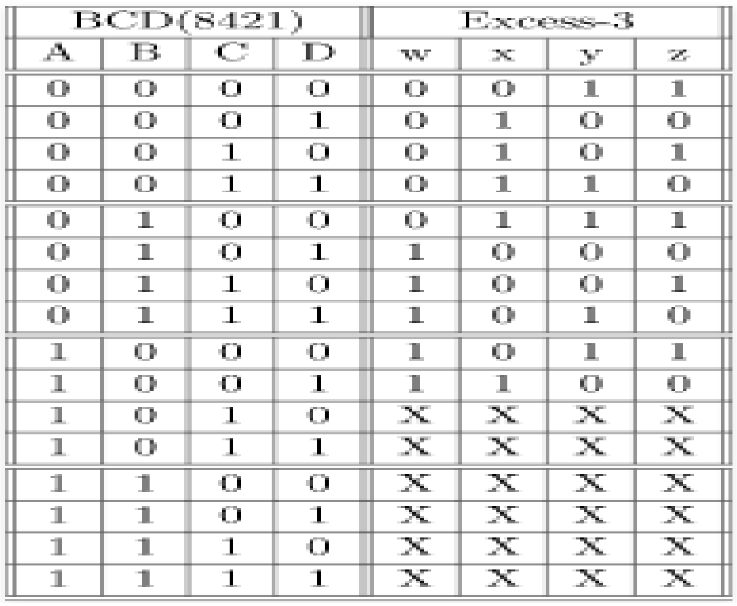
### Theory :



A **BCD to Excess-3 converter** is a digital circuit that converts a Binary Coded Decimal (BCD) inputinto an Excess-3 code. In BCD, each decimal digit is represented by a 4-bit binary number, while Excess-3 adds 3 to each decimal digit before encoding it. For example, the BCD representation of thedigit "2" is 0010, and in Excess-3, it becomes 0101 (which is 2 + 3). The conversion is achieved by adding the binary value of 3 (0011) to the BCD input.

Conversely, an **Excess-3 to BCD converter** takes an Excess-3 input and subtracts 3 from it to return the original BCD code. For instance, the Excess-3 code for "5" is 1000, and subtracting 3 (0011) gives 0101, which is the BCD for "2". These converters are useful in digital systems that handle decimal numbers, especially in applications like digital clocks, calculators, and other devices that require numeric processing.

### Truth Table :



| BCD(8421) | | | | Excess-3 | | | |
|---|---|---|---|---|---|---|---|
| A | B | C | D | w | x | y | z |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | X | X | X | X |
| 1 | 0 | 1 | 1 | X | X | X | X |
| 1 | 1 | 0 | 0 | X | X | X | X |
| 1 | 1 | 0 | 1 | X | X | X | X |
| 1 | 1 | 1 | 0 | X | X | X | X |
| 1 | 1 | 1 | 1 | X | X | X | X |

## BCD to Excess :-

W=A+BC+BD

X=B'C+B'D+BC'D'

Y=CD+C'D'

Z=D'

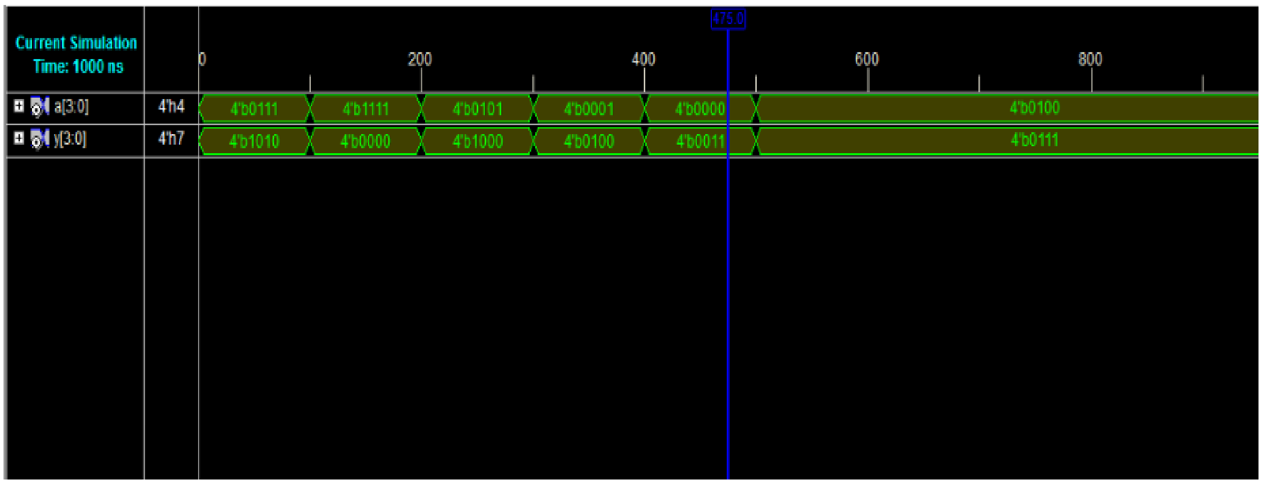
```
module w=(a w BCD_EX3(
Input b,
Input c,
Input d,
Output w,
Output x,
Output y,
Output z);
assign w=(a|(b&c)|(b&d);
assign x=(((~b)&c)&((~b)&d)|(b&(~c)&~d)));
assign y=((c&d)|((~c)&(~d)));
 assign z=(~d);
 endmodule
```

## Test Bench:-

```
 module rrg_x;
 reg a;
reg b;
reg c;
reg d;
wire w;
wire x;
wire y;
wire z;
rdt uut(
.a(a),
.b(b),
.c(c),
.d(d),
.w(w),
.x(x),
Y(y),
Z(z),
  );
Initial begin
a=0;
b=0;
c=0;
d=0;
#100;
a=0;
b=0;
c=0;
d=1;
#100;
a=0;
b=0;
c=1;
d=0;
#100;
a=0;
b=0;
c=1;
d=1;
#100;
a=0;
b=1;
c=0;
```

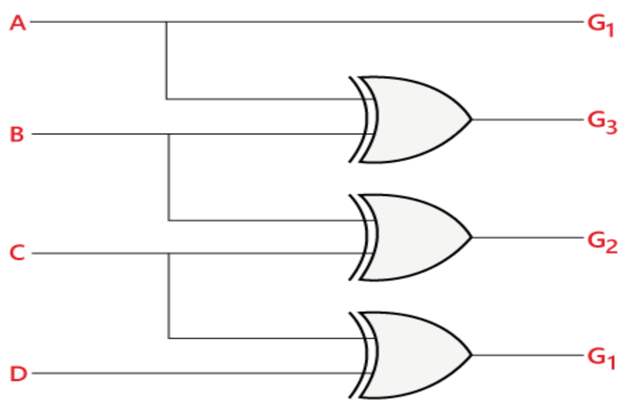
```
d=0;
#100;
endmodule
```

## Output :-



## d) Binary to Gray Converter and vice versa :

### Theory :

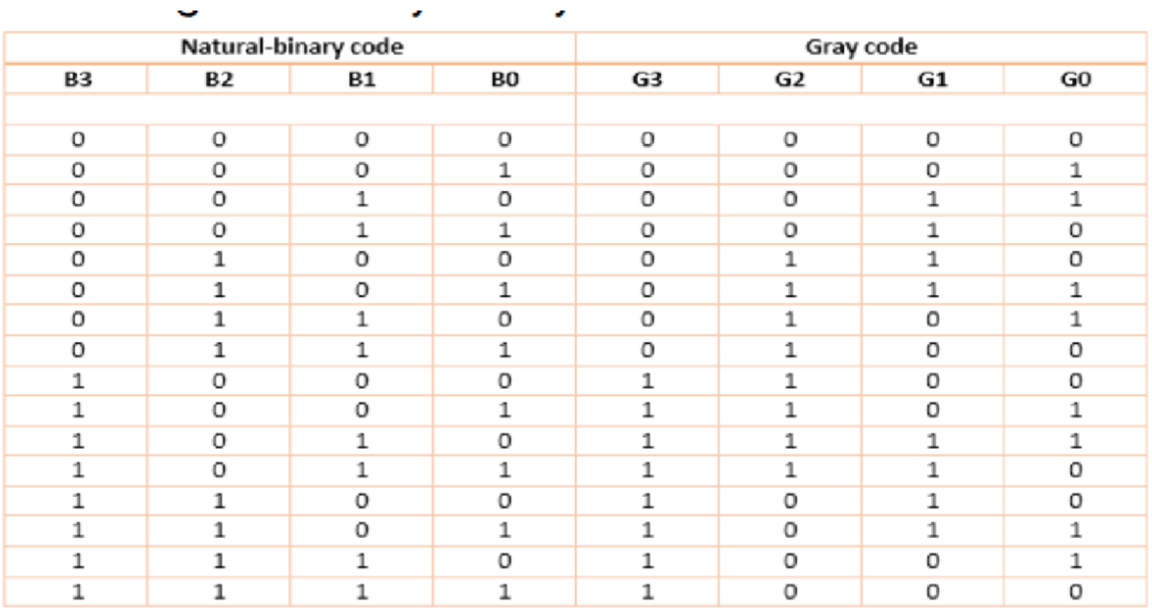


Logic Circuit for Binary to Gray Code Converter

**Binary to Gray code conversion** is a process where a binary number is transformed into its equivalent Gray code. In Gray code, only one bit changes at a time between consecutive numbers, which helps minimize errors in digital communication and electronics. To convert from binary to Gray code, the most significant bit (MSB) remains the same, and each subsequent bit is obtained by XOR-ing the corresponding binary bit with the previous Gray code bit. For example, the binary number 1101 converts to Gray code as 1011.

The **Gray to Binary conversion** is the reverse process, where a Gray code is converted back to its original binary form. To convert from Gray code to binary, the MSB remains unchanged, and each subsequent binary bit is derived by XOR-ing the previous binary bit with the current Gray code bit. This method ensures the accurate recovery of the original binary number. Both conversions are used in applications like error correction, rotary encoders, and digital communication systems.

## Truth Table :



| Natural-binary code | | | | Gray code | | | |
|---|---|---|---|---|---|---|---|
| B3 | B2 | B1 | B0 | G3 | G2 | G1 | G0 |
| | | | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |

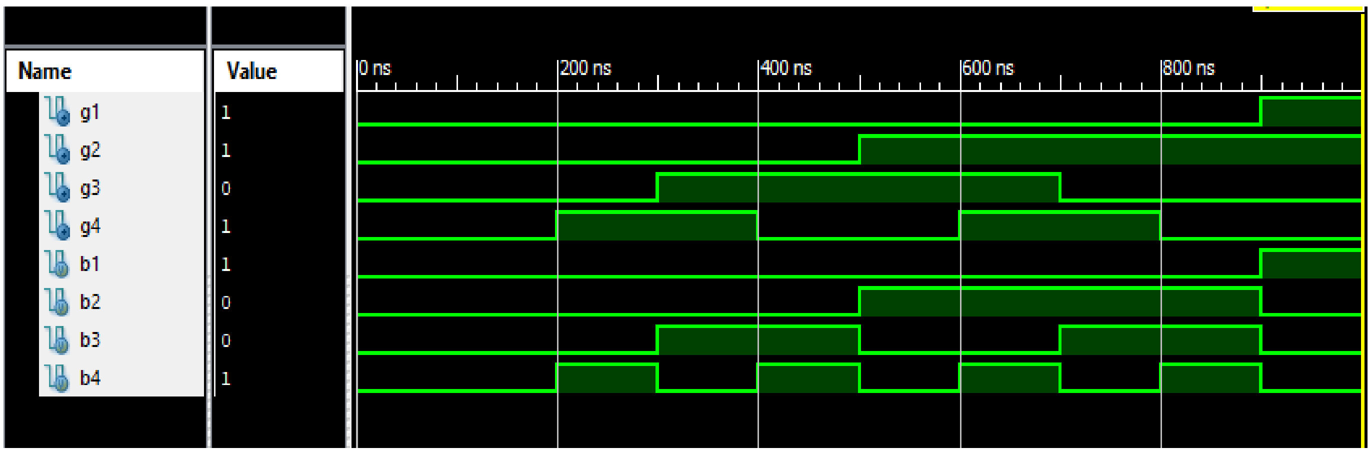
## BINARY TO GRAY:-

```
module bin2gray(input[3:0] bin,output[3:0]G);

assign G[3]=bin[3];
assign  G[2]=bin[3]^bin[2];
assign G[1]=bin[2]^bin[1] ;
assign  G[0]=bin[1]^bin[0];
endmodule
```

## Test Bench:-

```
 Module sst_s; reg [3:0] bin;
wire[3:0]G; sdf uut(
.bin(bin),
.G(G),
 );
Initial begin

bin=0100;

#100;
bin=1001;
#100;
Endmodule
```

## Output :-



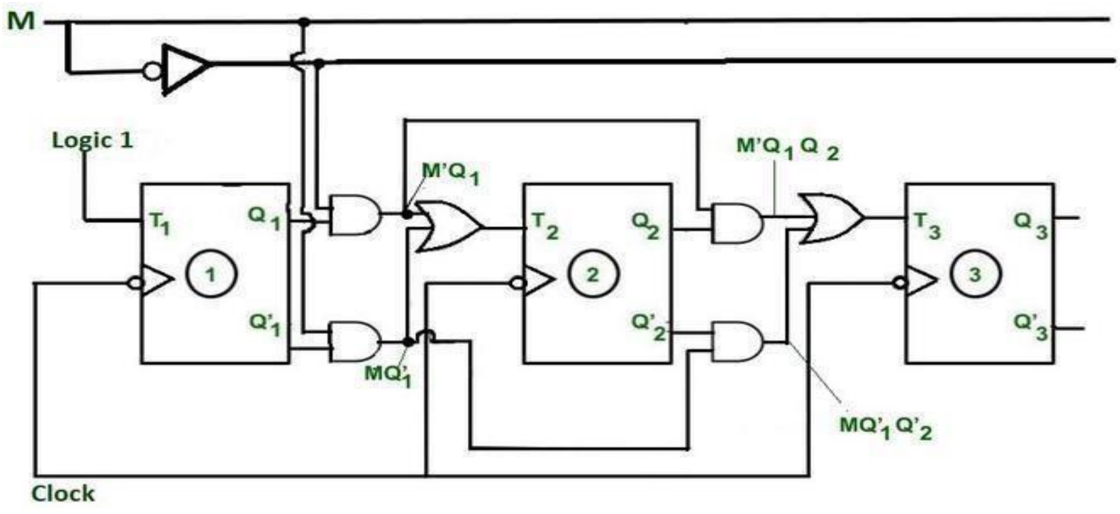
| Name | Value |
|---|---|
| g1 | 1 |
| g2 | 1 |
| g3 | 0 |
| g4 | 1 |
| b1 | 1 |
| b2 | 0 |
| b3 | 0 |
| b4 | 1 |

## EXPERIMENT NO :-06

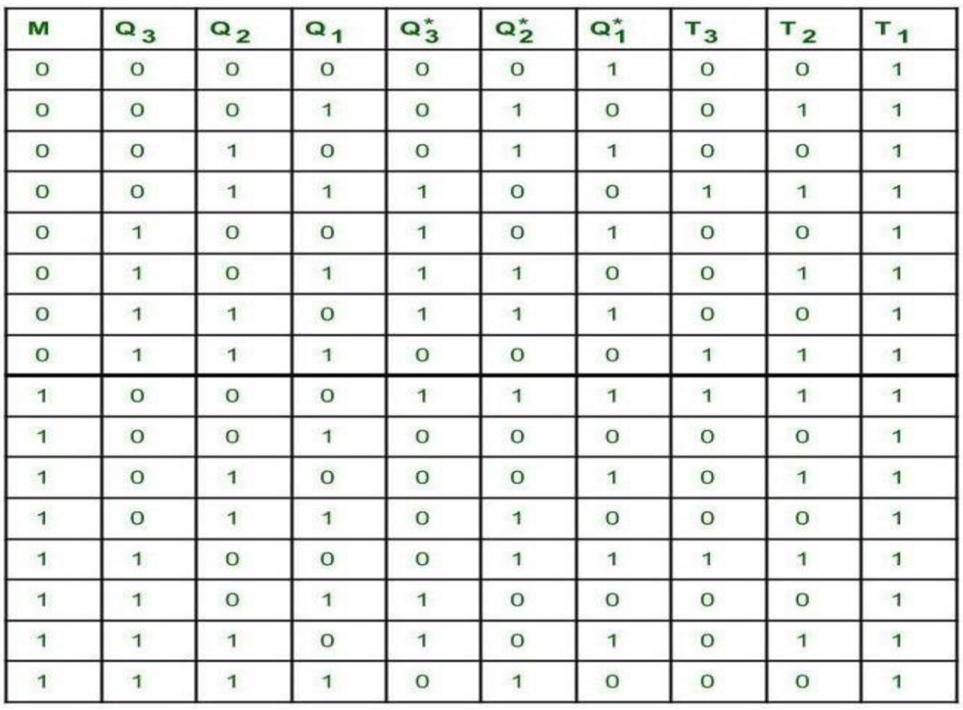
### a) N-bit Synchronous Up-Down Counter :-

#### Theory :

An **N-bit Synchronous Up-Down Counter** is a digital counter that can count both upwards and downwards based on a control input. It consists of N flip-flops, typically T or JK flip- flops, that store the current count in binary form. The counter increments (counts up) when the control input is set to "up," and it decrements (counts down) when the control input is set to "down." All flip-flops are driven by a common clock signal, making it a synchronous counter. The state of the counter changes on each clock pulse, with the direction of counting determined by the up/down control signal. Such counters are widely used in applications that require counting in both directions, like in digital clocks, frequency dividers, and event counters.



### Truth Table :-



| M | $Q_3$ | $Q_2$ | $Q_1$ | $Q_3^*$ | $Q_2^*$ | $Q_1^*$ | $T_3$ | $T_2$ | $T_1$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |

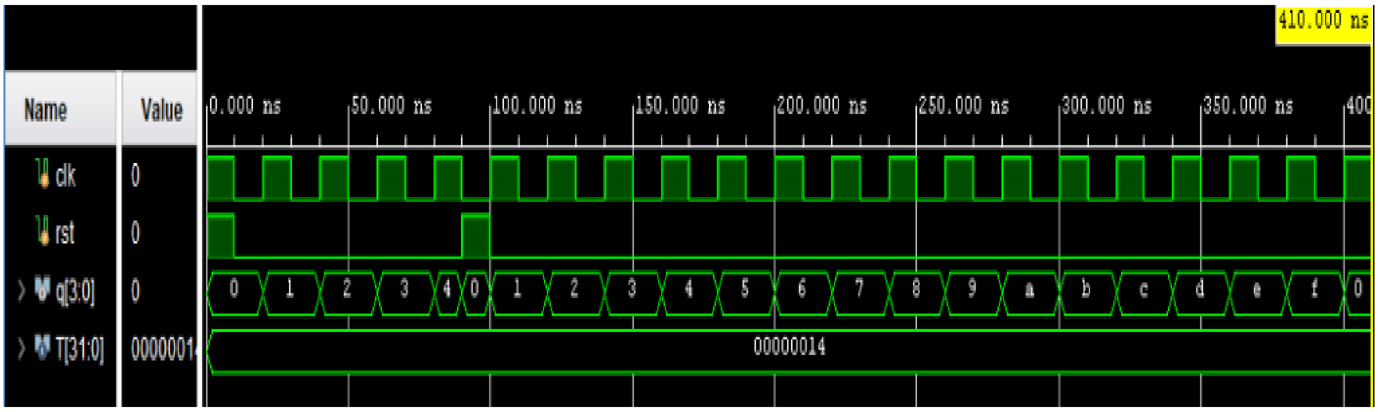
## SYNCHRONOUS UPORDOWN COUNTER:-

```
module upordown_counter(clk,rst,upordown,count);

input clk,rst,upordown;
output [3:0] count;
reg [3:0] count=0;
always@(posedge (clk) or posedge(rst))
begin
if (reset==1)
count<=0;
else
if (upordown==1)
if (count==15)
count<=0;
else
count<=count+1;
else
if (count==0)
count<=15;
else
count<=count-1;
end
endmodule
```

## TestBench:-

```
 Module  tdc_r;
 reg clk;
reg rst;
reg upordown;
wire [3:0] count;
pqr uut(
.clk(clk),
.rst(rst);
.upordown(upordown),
.count(count),
  );
always#10 clk=~clk;
Initial begin

clk=0;  rst=0;
upordown=0;
#100;
clk=0;
upordown=1;
#100;
Clk=0;
upordown=0;
#100;
endmodule
```

## Output:-

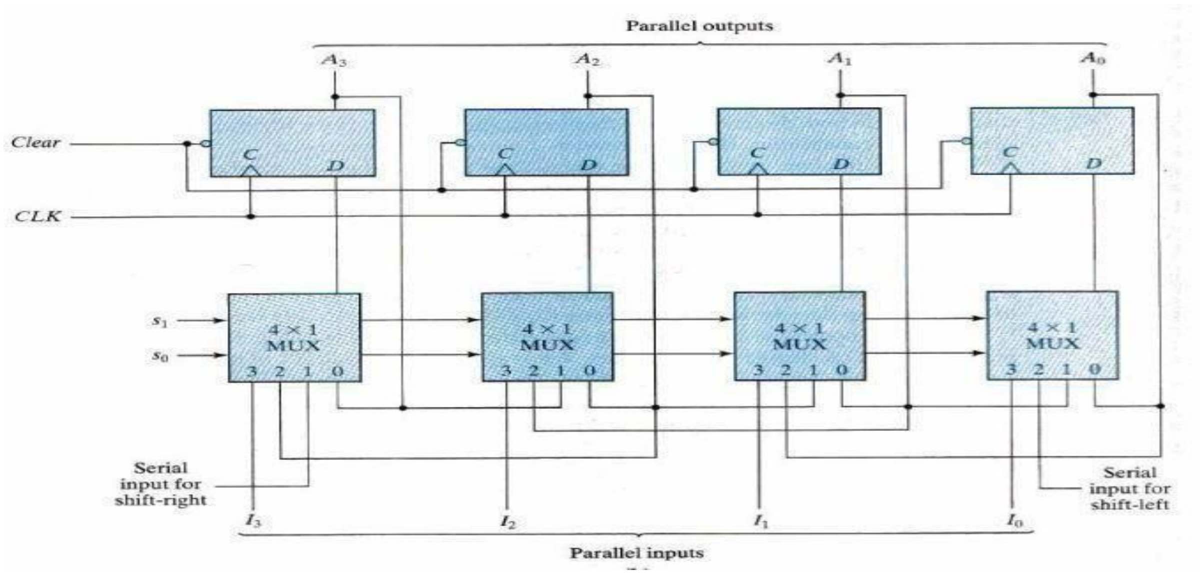


## EXPERIMENT NO :- 07

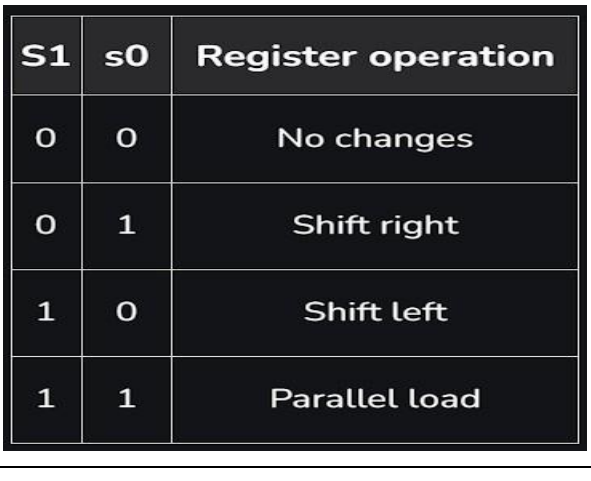
### a) 4-Bit Universal Shift Register :-

### Theory :-

A **4-bit Universal Shift Register** is a versatile digital storage device capable of shifting data in both directions (left and right), loading data in parallel, and holding data. It consists of fourflip-flops, and each flip-flop can store one bit of data. The register allows for different operations based on control inputs: data can be shifted left or right with each clock pulse, newdata can be loaded in parallel, or the current data can be retained without change. The shift operations are controlled by shift direction and shift enable signals, while the parallel load operation is controlled by a load signal. This flexibility makes the universal shift register ideal for applications requiring data storage, transfer, and manipulation, such as in data conversion, temporary storage in digital circuits, and serial-to-parallel or parallel-to-serial conversion.



### Truth Table :-



| S1 | s0 | Register operation |
|---|---|---|
| 0 | 0 | No changes |
| 0 | 1 | Shift right |
| 1 | 0 | Shift left |
| 1 | 1 | Parallel load |

## UNIVERSAL SHIFT REGISTER:-

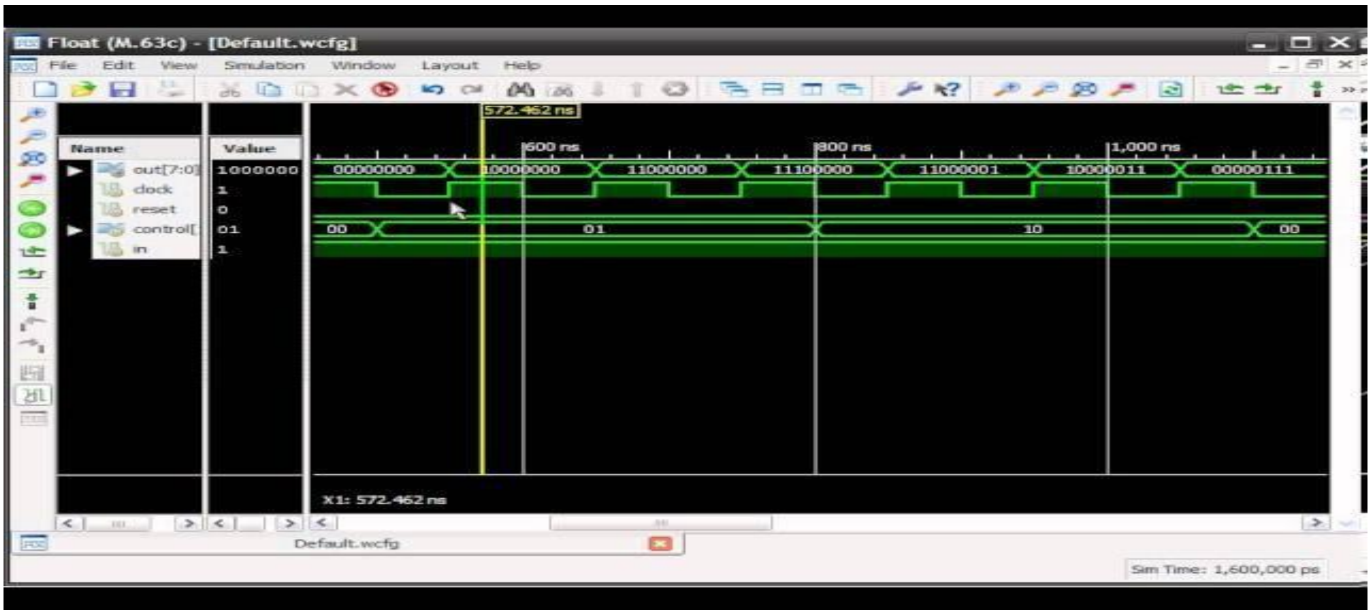
```
module universal_shift_reg(
input cllk,rst_n,
input[1:0] select,
input[3:0] p_in;
input s_left_in,
input s_right_in,
output reg[3:0] p_out,
output s_left_out,
output s_right_out);
always@(posedge clk)
begin
if(!rst_n)
p_out<=0;
else begin
case(select)
2’b01:p_out<={s_right_in,p_out[3:1]};
2’b10:p_out<={p_out[2:0],s_left_in};
2’b11:p_out<=p_in; default:p_out<=p_out;
endcase
end
end
assign s_left_out=p_out[0];
assign s_right_out=p_out[3];
endmodule
```

## Test Bench:-

```
 module ccf_i;
 reg clk;
reg rst_n;
reg [1:0]select;
reg[3:0]p_in; reg
s_left_in; reg
s_right_in; wire
[3:0] p_out; wire
s_left_out;
wire s_right_out);
always#10 clk=~clk
initial begin
clk=0;
rst_n=0;
select=0;
p_in=0;
s_left_in=0;
s_right_in=0;
#100;
clk=0; rst_n=1;
select=2’b11;
p_in=4’b1101;
s_left_in=1’b1;
s_right_in=1’b0;
#100;
clk=0;
rst_n=1;
select=2’b10;
```

```
p_in=4'b1101;

s_left_in=1'b1;

s_right_in=1'b0;

#100;
endmodule
```

**Output:-**

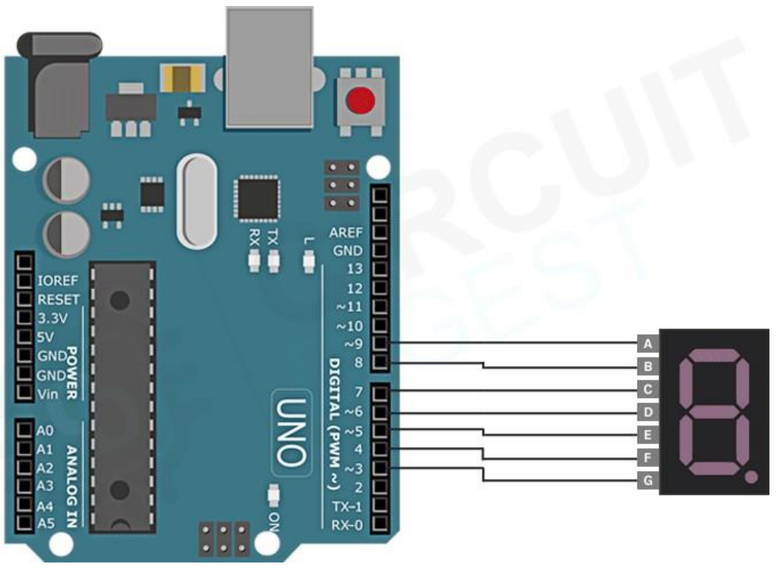


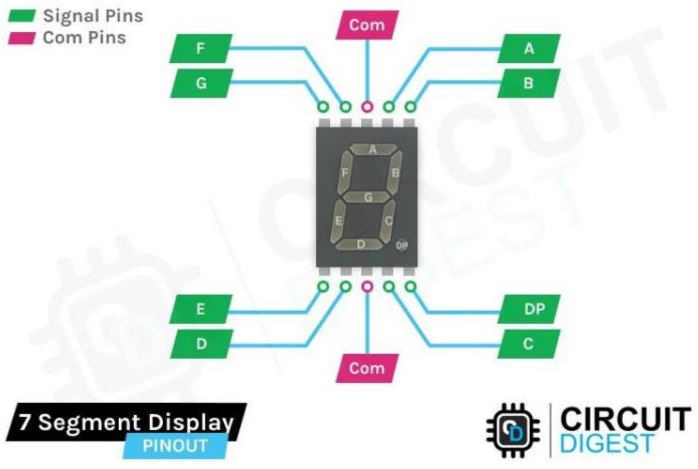
## EXPERIMENT NO :- 08

## Seven-Segment Light-Emitting Diode (LED) Display

### Theory :-

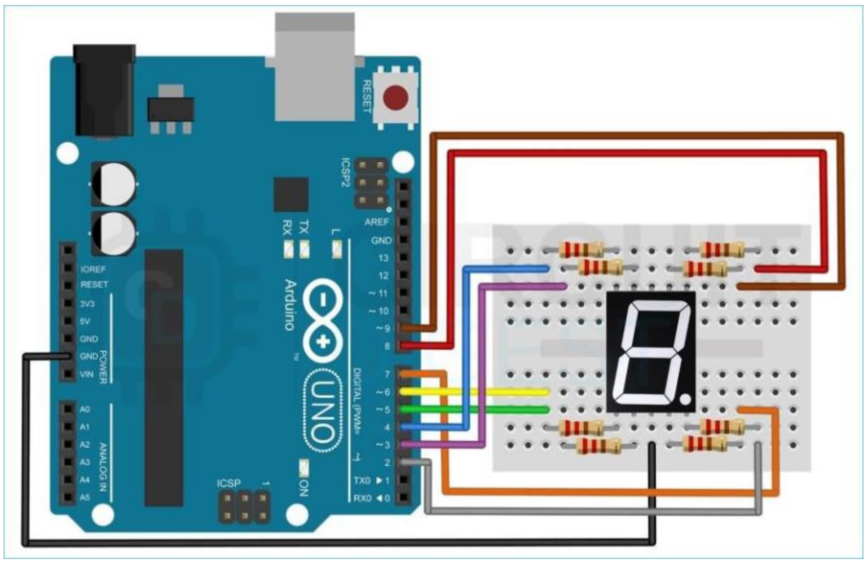
The seven-segment displays are designed for displaying numeric values. You can find them anywhere from instruments to space shuttles. They are the most practical way to display numeric values. They are cheap and easy to use. Not only that, they are highly readable in any light condition, unlike LCDs. You can find them in many everyday usages like counters or token systems, etc. And here is a small simulation showing the Seven Segment Interface. The Arduino will count from 0 to 9 and repeat. The value will be displayed on the Seven Segment display. Seven-segment displays are available in various sizes and colours. They are available from 0.28 inches to 18 inches, and even bigger sizes are available for industrial usage. The **most commonly used display size is 0.56 inches**.

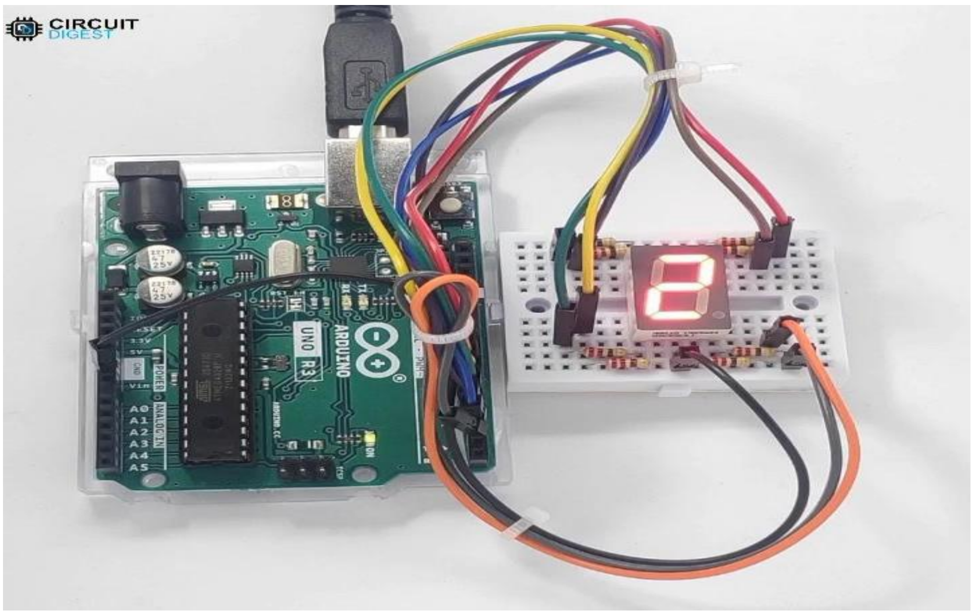




Now let's see how we can drive a seven-segment display with an Arduino. For that let's start by placing the display module on a breadboard with the decimal point facing downwards. Then wire up each pin as per the connection diagram below. In this tutorial, we will be using a Common Cathode display. Connect either of the common pins (seven-segment pin 3 or 8) to the ground. And connect the rest of the pins to D2 to D9 of the Arduino through a current limiting resistor, as per the connection diagram. Just like driving an LED, it is preferable to use these current limiting resistors to increase the display life. Choose these values depending on the display colour and size. Here we will be using a 330 ohms resistor which will provide around 10mA current for the LEDs.

And here is the actual circuit.





```
#include "SevSeg.h"
SevSeg sevseg;
void setup()
{
    //Set to 1 for single-digit display
    byte numDigits = 1;
    //defines common pins while using multi-digit display. Left for single digit display
    byte digitPins[] = {};
    //Defines Arduino pin connections in order: A, B, C, D, E, F, G, DP
    byte segmentPins[] = {9,8, 7, 6, 5, 4, 3, 2};
    byte displayType = COMMON_CATHODE; //Use COMMON_ANODE for Common Anode display
    bool resistorsOnSegments = true; //'false' if resistors are connected to common pin
    //Initialize sevseg object. Use COMMON_ANODE instead of COMMON_CATHODE for CA display
    sevseg.begin(displayType, numDigits, digitPins, segmentPins, resistorsOnSegments);
    sevseg.setBrightness(90);
}
void loop()
{
  //Display numbers 0-9 with 1 seconds delay
  for(int i = 0; i <= 10; i++)
  {
    if (i == 10)
{
 i = 0;
}
    sevseg.setNumber(i);
    sevseg.refreshDisplay();
    delay(1000);
  }
}
```

## EXPERIMENT NO:9

## Data Flow and Structural Modeling

### Data flow modeling of Half Adder:

```
module half_adder(a, b, sum, carry);

input a, b;

output sum, carry;

assign sum = a ^ b;

assign carry = a & b;

endmodule
```

### Structural modeling of Half Adder:

```
module half_adder(a, b, sum, carry);

input a, b;

output sum, carry;

xor (sum, a, b);

and (carry, a, b);

endmodule
```

## Data flow modeling of Full Adder:

```
module full_adder(a, b, cin, sum, carry);
input a, b, cin;
output sum, carry;
assign sum = a ^ b ^ cin;
assign carry = (a & b) | (b & cin) | (cin & a);
endmodule
```

## Structural modeling of Full Adder:

```
module fulladder(a, b, cin, sum, carry);
input a, b, cin;
output sum, carry;
wire w1, w2, w3, s1;
xor x1(s1, a, b);
xor x2(sum, s1, cin);
and a1(w1, a, b);
and a2(w2, b, cin);
and a3(w3, a, cin);
or o1(carry, w1, w2, w3);
endmodule
```