



## Department of Electronics and Communication Engineering

### HARDWARE DESCRIPTION LABORATORY MANUAL

**Subject Code: 18ECL58**

**Prepared by**

**Mrs. Vijaya Dalawai**

**Assistant Professor, Dept. Of ECE**

**Dr. H B Bhuvaneswari**

**HOD, Dept. Of ECE**



**Affiliated to Visvesvaraya Technological University,**

**Belagavi, Karnataka - 590018**

**2022-23**

## SYLLABUS

### (ACCORDING TO VTU SYLLABUS)

#### **PRACTICAL COMPONENT OF IPCC**

Using suitable simulation software, demonstrate the operation of the following circuits:

#### **Experiments**

To simplify the given Boolean expressions and realize using Verilog program.

To realize Adder/Subtractor (Full/half) circuits using Verilog data flow description.

To realize 4-bit ALU using Verilog program.

To realize the following Code converters using Verilog Behavioral description

- a) Gray to binary and vice versa b) Binary to excess3 and vice versa

To realize using Verilog Behavioral description: 8:1 mux, 8:3 encoder, Priority encoder

To realize using Verilog Behavioral description: 1:8 Demux, 3:8 decoder, 2-bit Comparator

To realize using Verilog Behavioral description:

- Flip-flops: a) JK type b) SR type c) T type and d) D type

To realize Counters - up/down (BCD and binary) using Verilog Behavioral description.

#### **Demonstration Experiments (For CIE only - not to be included for SEE)**

Use FPGA/CPLD kits for downloading Verilog codes and check the output for interfacing experiments.

Verilog Program to interface a Stepper motor to the FPGA/CPLD and rotate the motor in the specified direction (by N steps).

Verilog programs to interface a Relay or ADC to the FPGA/CPLD and demonstrate its working.

Verilog programs to interface DAC to the FPGA/CPLD for Waveform generation.

Verilog programs to interface Switches and LEDs to the FPGA/CPLD and demonstrate its working.

**To realize adder/substractor (Full/Half) Circuits usimg verilog dataflow description.**

**PROGRAM:-**

```
module full_adder(
    input a, b, cin,
    output sum, cout
);
    assign {sum, cout} = {a^b^cin, ((a & b) | (b & cin) | (a & cin))};
    //or
    //assign sum = a^b^cin;
    //assign cout = (a & b) | (b & cin) | (a & cin);
endmodule

module ripple_carry_adder_subtractor #(parameter SIZE = 4) (
    input [SIZE-1:0] A, B,
    input CTRL,
    output [SIZE-1:0] S, Cout);
    bit [SIZE-1:0] Bc;
    genvar g;
    assign Bc[0] = B[0] ^ CTRL;
    full_adder fa0(A[0], Bc[0], CTRL, S[0], Cout[0]);
    generate // This will instantial full_adder SIZE-1 times
        for(g = 1; g<SIZE; g++) begin
            assign Bc[g] = B[g] ^ CTRL;
            full_adder fa(A[g], Bc[g], Cout[g-1], S[g], Cout[g]);
        end
    endgenerate
endmodule
```

**TEST BENCH PROGRAM:-**

```
module RCAS_TB;
    wire [3:0] S, Cout;
    reg [3:0] A, B;
    reg ctrl;
    ripple_carry_adder_subtractor rcas(A, B, ctrl, S, Cout);
    initial begin
        $monitor("CTRL=%b: A = %b, B = %b --> S = %b, Cout[3] = %b", ctrl, A, B, S, Cout[3]);
        ctrl = 0;
```

```

A = 1; B = 0;
#3 A = 2; B = 4;
#3 A = 4'hb; B = 4'h6;
#3 A = 5; B = 3;
ctrl = 1;
A = 1; B = 0;
#3 A = 2; B = 4;
#3 A = 4'hb; B = 4'h6;
#3 A = 5; B = 3;
#3 $finish;
end
initial begin
$dumpfile("waves.vcd");
$dumpvars;
end
endmodule

```

### **OUTPUT:-**

```

CTRL=0: A = 0001, B = 0000 --> S = 0001, Cout[3] = 0
CTRL=0: A = 0010, B = 0100 --> S = 0110, Cout[3] = 0
CTRL=0: A = 1011, B = 0110 --> S = 0001, Cout[3] = 1
CTRL=1: A = 0001, B = 0000 --> S = 0001, Cout[3] = 1
CTRL=1: A = 0010, B = 0100 --> S = 1110, Cout[3] = 0
CTRL=1: A = 1011, B = 0110 --> S = 0101, Cout[3] = 1
CTRL=1: A = 0101, B = 0011 --> S = 0010, Cout[3] = 1

```

### **TO Realize 4-bit ALU using verilog Program**

### **PROGRAM:-**

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity alu is
Port ( inp_a : in signed(3 downto 0);
inp_b : in signed(3 downto 0);
sel : in STD_LOGIC_VECTOR (2 downto 0);
out_alu : out signed(3 downto 0));
end alu;

```

architecture Behavioral of alu is  
begin

```

process(inp_a, inp_b, sel)
begin
case sel is
when "000" =>
out_alu<= inp_a + inp_b; -- addition
when "001" =>
out_alu<= inp_a - inp_b; -- subtraction
when "010" =>
out_alu<= inp_a - 1; -- sub 1
when "011" =>
out_alu<= inp_a + 1; -- add 1
when "100" =>
out_alu<= inp_a and inp_b; -- AND gate
when "101" =>
out_alu<= inp_a or inp_b; -- OR gate
when "110" =>
out_alu<= not inp_a ; -- NOT gate
when "111" =>
out_alu<= inp_a xor inp_b; -- XOR gate
when others =>
NULL;
end case;
end process;
end Behavioral;

```

### **TEST BENCH PROGRAM:-**

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

```

```

ENTITY Tb_alu IS
END Tb_alu;

```

ARCHITECTURE behavior OF Tb\_alu IS

– Component Declaration for the Unit Under Test (UUT)

```

COMPONENT alu
PORT(
inp_a : IN signed(3 downto 0);
inp_b : IN signed(3 downto 0);
sel : IN std_logic_vector(2 downto 0);
out_alu : OUT signed(3 downto 0)
);
END COMPONENT;

```

– Inputs

```

signal inp_a : signed(3 downto 0) := (others => '0');
signal inp_b : signed(3 downto 0) := (others => '0');
signal sel : std_logic_vector(2 downto 0) := (others => '0');

```

```

– Outputs
signal out_alu : signed(3 downto 0);

BEGIN
    – Instantiate the Unit Under Test (UUT)
    uut: alu PORT MAP (
        inp_a => inp_a,
        inp_b => inp_b,
        sel => sel,
        out_alu => out_alu
    );
    – Stimulus process
    stim_proc: process
    begin
        – hold reset state for 100 ns.
        wait for 100 ns;

        – insert stimulus here

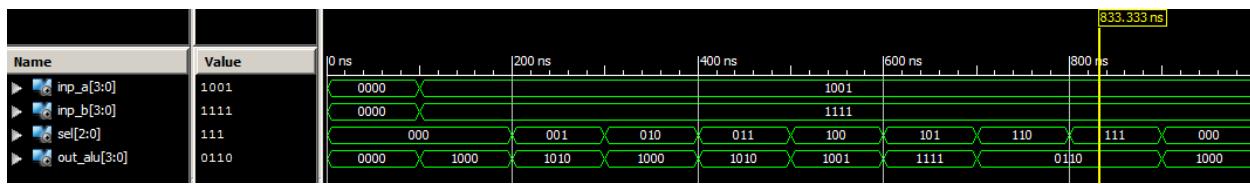
        inp_a <= "1001";
        inp_b <= "1111";

        sel <= "000";
        wait for 100 ns;
        sel <= "001";
        wait for 100 ns;
        sel <= "010";
        wait for 100 ns;
        sel <= "011";
        wait for 100 ns;
        sel <= "100";
        wait for 100 ns;
        sel <= "101";
        wait for 100 ns;
        sel <= "110";
        wait for 100 ns;
        sel <= "111";
        end process;

    END;

```

## OUTPUT:-



To realize the following Code Converters using Verilog Behavioral description

**a)Gray to Binary and vice versa**

**Program(Binary to gray):-**

```
module Bin_Gry(
```

```
    input [3:0]din,
```

```
    output [3:0]dout
```

```
);
```

```
reg [3:0]dout;
```

```
always @ (din)
```

```
begin
```

```
case(din)
```

```
    0 : dout = 0;
```

```
    1 : dout = 1;
```

```
    2 : dout = 3;
```

```
    3 : dout = 2;
```

```
    4 : dout = 6;
```

```
    5 : dout = 7;
```

```
    6 : dout = 5;
```

```
    7 : dout = 4;
```

```
    8 : dout = 12;
```

```
    9 : dout = 13;
```

```
    10 : dout = 15;
```

```
    11 : dout = 14;
```

```
    12 : dout = 10;
```

```
    13 : dout = 11;
```

```
    14 : dout = 9;
```

```
    15 : dout = 8;
```

```
    default: dout = 4'b xxxx;
```

```
endcase
```

```
end
```

```
endmodule
```

**TESTBENCH :-**

```
initial begin
```

```
// Initialize Inputs
```

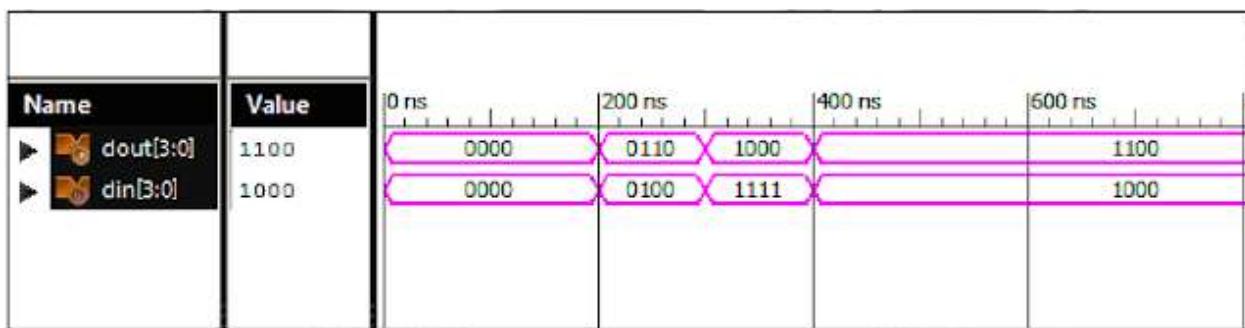
```
din = 0;
```

```

// Wait 100 ns for global reset to finish#100;
// Add stimulus here
#100; din = 4;
#100; din = 15;
#100; din = 8;
end
initial begin
#100
$monitor("din = %b, dout = %b, din, dout");
end
endmodule

```

### OUTPUT:-



### Program(GRAY to BINARY):-

```

module Gry_Bin(
    input [3:0]din,
    output [3:0]dout
);
reg [3:0]dout;

always @ (din)
begin
    case(din)
        0 : dout = 0;
        1 : dout = 1;
        2 : dout = 3;
        3 : dout = 2;
        4 : dout = 7;
        5 : dout = 6;
        6 : dout = 4;
        7 : dout = 5;
        8 : dout = 15;
    endcase
end
endmodule

```

```

9 : dout = 14;
10 : dout = 12;
11 : dout = 13;
12 : dout = 8;
13 : dout = 9;
14 : dout = 11;
15 : dout = 10;
default: dout = 4'b xxxx;
endcase
end
endmodule

```

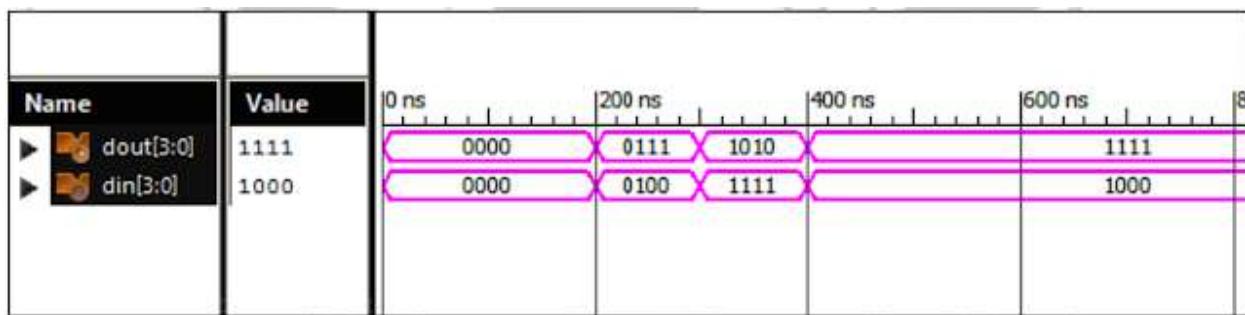
### TESTBENCH :-

```

initial begin
// Initialize Inputs
din = 0;
// Wait 100 ns for global reset to finish#100;
// Add stimulus here
#100; din = 4;
#100; din = 15;
#100; din = 8;
end
initial begin
#100
$monitor("din = %b, dout = %b, din, dout");
end
endmodule

```

### Output:-



b)Binary to excess and vice versa

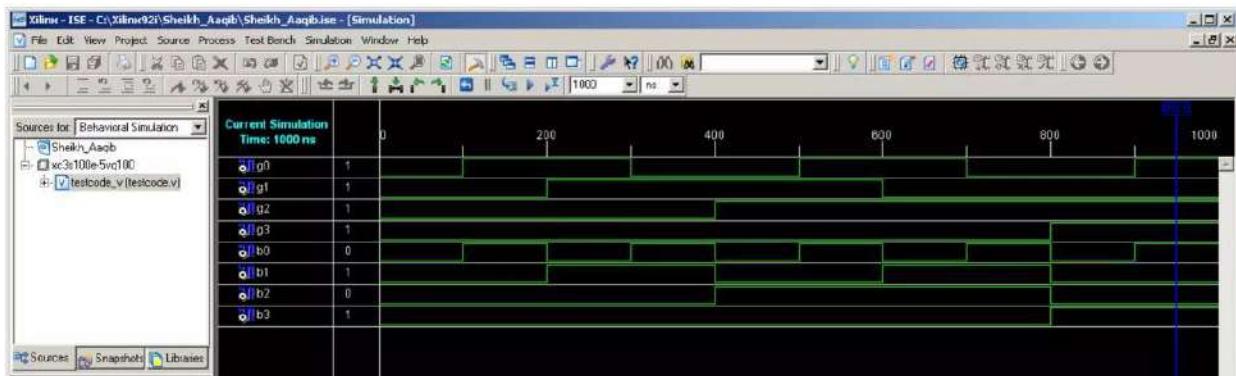
Program(binary to excess3)

```

module binary2ex3(b3,b2,b1,b0,e3,e2,e1,e0);
input b3,b2,b1,b0;
output e3,e2,e1,e0;
assign e3=b3|b0&b2|b2&b1;
assign e2=(~b1)&(~b0)&(b2)|(~b2)&(b0)|(~b2)&(b1);
assign e1=(~b1)&(~b0)|b1&b2;
assign e0=(~b1)&(~b0)|b1&(~b0);
endmodule

```

### Output:-



### Program(excess3 to binary)

NOT FOUND

To realize using verilog behavioral description

#### 8:1 mux

```

module m81(out, D0, D1, D2, D3, D4, D5, D6, D7, S0, S1, S2);
input wire D0, D1, D2, D3, D4, D5, D6, D7, S0, S1, S2;
output reg out;
always@(*)
begin
case(S0 & S1 & S2)
3'b000: out=D0;
3'b001: out=D1;
3'b010: out=D2;
3'b011: out=D3;
3'b100: out=D4;
3'b101: out=D5;
3'b110: out=D6;

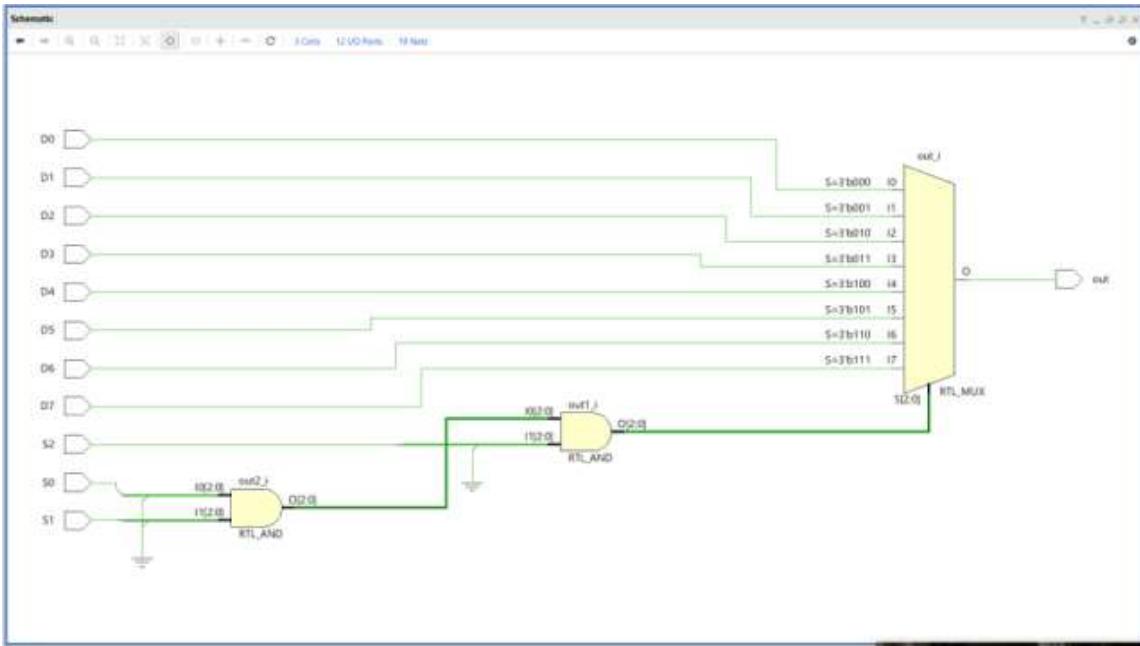
```

```

3'b111: out=D7;
default: out=1'b0;
endcase
end
endmodule

```

## Output



## 3:8 decoder

### **DESIGN OF 8-TO-3 ENCODER (WITHOUT AND WITH PRIORITY)**

| BehaviouralModel | BehaviouralModel with Enable |
|------------------|------------------------------|
|------------------|------------------------------|

```

module encoder (din, dout);
input [7:0] din;
output [2:0] dout;
reg [2:0] dout;
always @(din) begin
if (din == 8'b00000001) dout=3'b000; else
if (din==8'b00000010) dout=3'b001; else if
(din==8'b00000100) dout=3'b010; else if
(din==8'b00001000) dout=3'b011; else if
(din==8'b00010000) dout=3'b100; else if
(din == 8'b00100000) dout=3'b101; else if
(din==8'b01000000) dout=3'b110; else if
(din==8'b10000000) dout=3'b111; else
dout=3'bX;
end endmodule

```

```

module encwtoutprio(a,en,y); input
[7:0] a;
input en;
output reg [2:0] y;
always@(a or en)
begin
if(!en)
y<=1'b0;
else
case(a)
8'b00000001:y<=3'b000;
8'b00000010:y<=3'b001;
8'b00000100:y<=3'b010;
8'b00001000:y<=3'b011;
8'b00010000:y<=3'b100;
8'b00100000:y<=3'b101;
8'b01000000:y<=3'b110;
8'b10000000:y<=3'b111;
endcase end
endmodule

```

## OUTPUT:-

**Simulation output: Waveform window:** Displays **output** waveform for verification.



3:8 decoder

```

module 3_8_DEC(
  input [3:0]din,
  output [7:0]dout
);
  reg [7:0]dout;
  always @ (din)
    case (din)

```

case (din)

```

0 : dout[0] = 1;
1 : dout[1] = 1;
2 : dout[2] = 1;
3 : dout[3] = 1;
4 : dout[4] = 1;
5 : dout[5] = 1;
6 : dout[6] = 1;
7 : dout[7] = 1;
default : dout = 8'bxxxxxxxx;

```

endcase

endmodule

**Testbench** code for 3 to 8 Decoder Behavioral Modelling using Case Statement

initial begin

// Initialize Inputs

din = 0;

// Wait 100 ns for global reset to finish

#100;

// Add stimulus here

#100; din=0;

#100; din=1;

#100; din=2;

#100; din=3;

#100; din=4;

#100; din=5;

#100; din=6;

#100; din=7;

end

initial begin

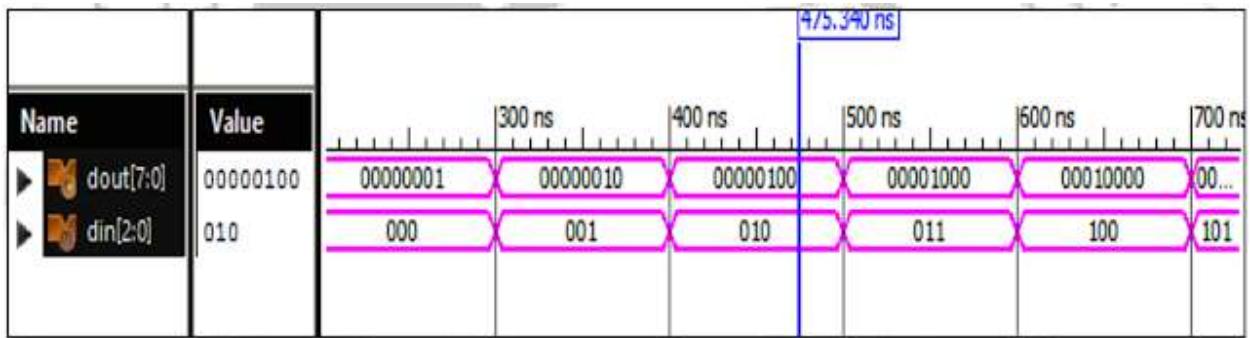
#100

\$monitor("din=%b, dout=%b", din, dout);

end

endmodule

**OUTPUT**



### Verilog Code for 2 Bit Magnitude Comparator Behavioral Modelling using If Else Statement with Testbench Code

```

module 2_Mag_Comp(
    input [1:0]a,b,
    output equal, greater, lower
);
reg greater, equal, lower;
initial greater = 0, equal = 0, lower = 0;
always @ (a or b)
begin
    if (a < b)
        begin
            greater = 0; equal = 0; lower = 1;
        end
    else if (a == b)
        begin
            greater = 0; equal = 1; lower = 0;
        end
    else
        begin
            greater = 1; equal = 0; lower = 0;
        end
end
endmodule

```

**Testbench code for 2 Bit Magnitude Comparator Behavioral Modelling using If Else Statement**

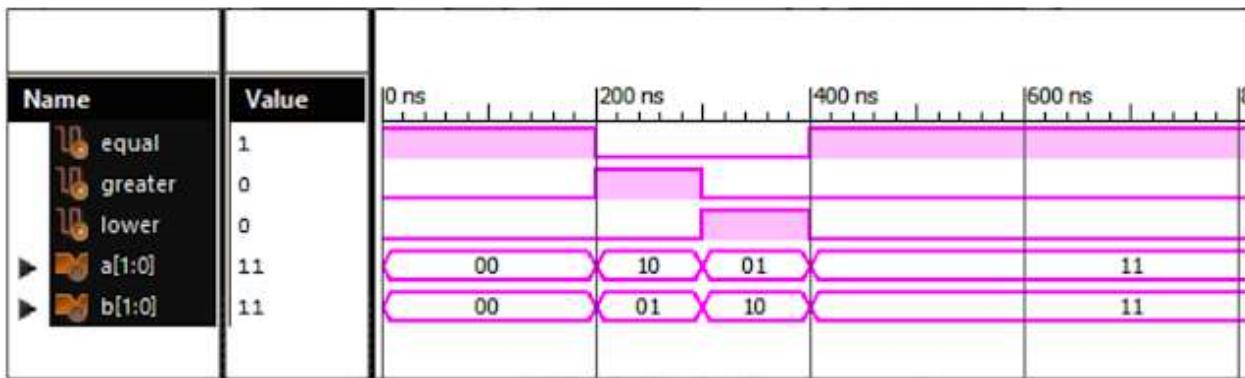
```
initial begin
```

```

// Initialize Inputs
a = 0; b = 0;
// Wait 100 ns for global reset to finish
#100;
// Add stimulus here
#100; a = 2; b = 1;
#100; a = 1; b = 2;
#100; a = 3; b = 3;
end
initial begin
#100
$monitor("a = %b, b = %b, lower = %b, greater = %b, equal = %b", a, b, lower, greater, equal);
end
endmodule

```

## OUTPUT



## JK FLIP FLOP

### Behavioral Description:

```

`timescale 1ns / 1ps

module jkff( input clk, input rst, input j, input k, output reg q, output reg
qb );
reg[1:0] jk;
always@(posedge clk)
begin
if(rst==1'b1)
q=1'b0;
else
begin
jk={j,k};
case (jk)
2'b00: q=q;

```

```
2'b01: q=1'b0;  
2'b10: q=1'b1;  
2'b11: q=~q;  
default: q=1'bx;  
endcase  
end  
qb=~q;  
end  
endmodule
```

### **Test Bench Program:**

```
`timescale 1ns / 1ps  
module jkff_test;  
reg clk;  
reg rst;  
reg j;  
reg k;  
wire q;  
wire qb;  
jkff uut (.clk(clk), .rst(rst), .j(j), .k(k), .q(q), .qb(qb));  
initial  
begin  
clk = 0;  
rst = 1;  
j = 0;  
k = 0;  
#30 j=0;k=0;rst=0;  
#30j=0;k=1;  
#30 j=1;k=0;  
#30 j=1;k=1;  
#200 $finish;  
end  
always  
#10 clk=~clk;  
endmodule
```

### **User Constraint File:**

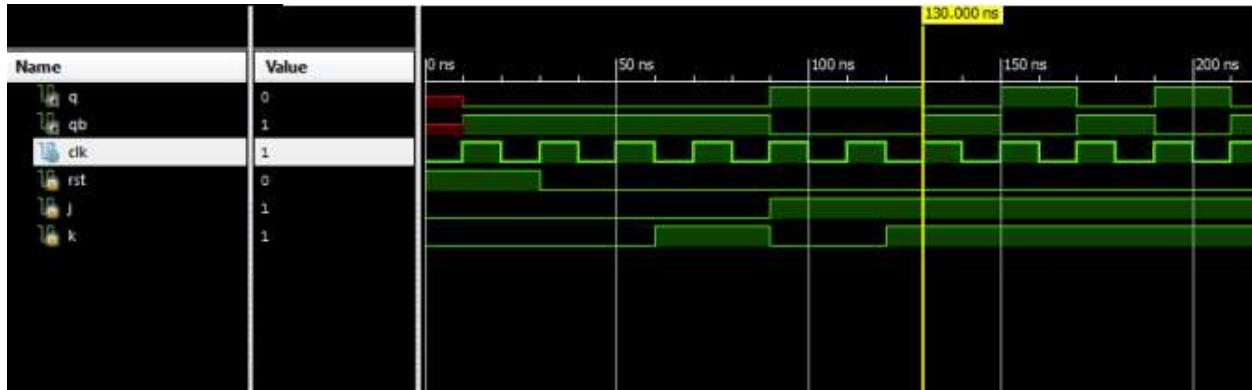
```
#PINLOCK_BEGIN  
NET "clk" LOC= "S:PIN1";
```

```

NET "rst" LOC = "S:PIN2";
NET "j" LOC= "S:PIN3";
NET "k" LOC= "S:PIN4";
NET "q" LOC= "S:PIN5";
NET "qb" LOC = "S:PIN6";
#PINLOCK_END

```

## OUTPUT



## SR FLIP FLOP

Behavioral Description:

```

`timescale 1ns / 1ps

module srff( input clk, input rst, input s, input r, output reg q, output regqb );
reg[1:0] sr;

always@(posedge clk)
begin
if(rst==1'b1)
q=1'b0;
else
begin
sr={s,r};
case (sr)
2'b00: q=q;
2'b01: q=1'b0;
2'b10: q=1'b1;
2'b11: q=1'bZ;
default: q=1'bx;
endcase
qb=~q;
end
endmodule

```

## Test Bench Program:

```

`timescale 1ns / 1ps
module srff_test;
reg clk;
reg rst;
reg s;
reg r;
wire q;
wire qb;
srff uut (.clk(clk), .rst(rst), .s(s), .r(r), .q(q), .qb(qb));
initial
begin
end
always
#10 clk=~clk;
endmodule

```

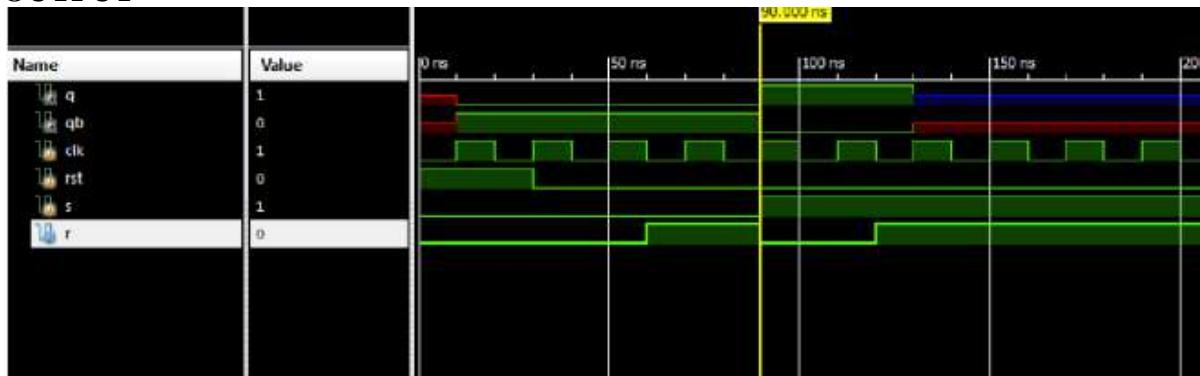
### User Constraint File:

```

#PINLOCK_BEGIN
NET "clk" LOC = "S:PIN1";
NET "rst" LOC = "S:PIN2";
NET "s" LOC = "S:PIN3";
NET "r" LOC = "S:PIN4";
NET "q" LOC = "S:PIN5";
NET "qb" LOC = "S:PIN6";
#PINLOCK_END

```

### OUTPUT



### T FLIP FLOP

```

module tff ( input clk, input rstn, input t, output reg q);
always @ (posedge clk) begin
  if (!rstn)

```

```

q <= 0;
else
  if (t)
    q <= ~q;
  else
    q <= q;
end
endmodule

```

## TESTBENCH

```

module tb;
reg clk;
reg rstn;
reg t;

tff u0 (.clk(clk),
         .rstn(rstn),
         .t(t),
         .q(q));

always #5 clk = ~clk;

initial begin
{rstn, clk, t} <= 0;

$monitor ("T=%0t rstn=%0b t=%0d q=%0d", $time, rstn, t, q);
repeat(2) @(posedge clk);
rstn <= 1;

for (integer i = 0; i < 20; i = i+1) begin
  reg [4:0] dly = $random;
  #(dly) t <= $random;
end
#20 $finish;
end
endmodule
D Flipflop
`timescale 1ns / 1ps

moduledff(clk,rst,d, q,qb);
input clk,rst,d;
output q,qb;
reg q,qb;
always@(posedge clk)
begin
if(rst==1'b1)
q=1'b0;
else
q=d;
qb=~q;

```

```
end  
endmodule
```

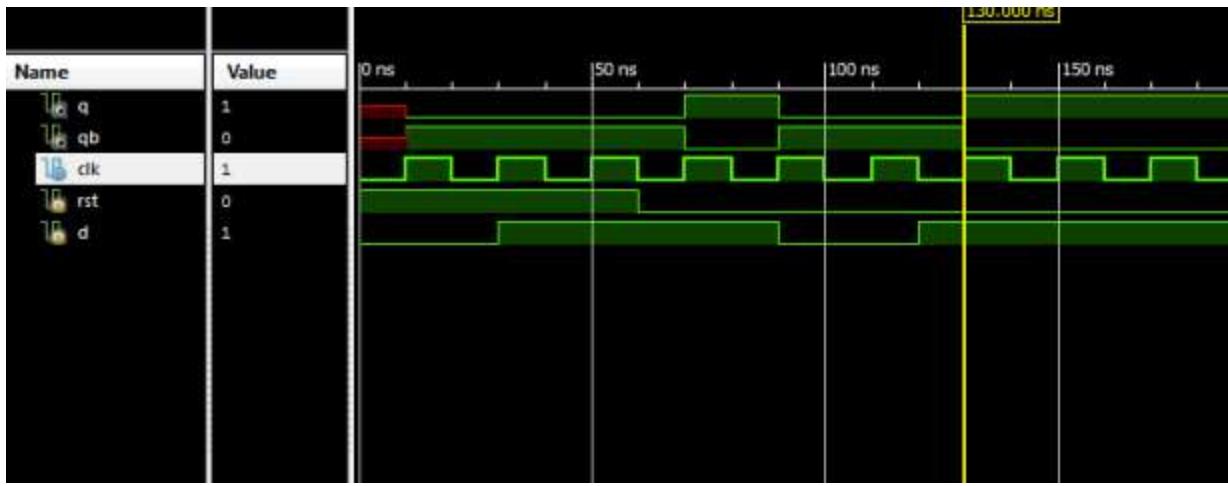
### **Test Bench Program:**

```
`timescale 1ns / 1ps  
  
module dFF_test;  
  
reg clk;  
reg rst;  
reg d;  
wire q;  
wire qb;  
  
dFF uut (.clk(clk),.rst(rst),.d(d),.q(q),.qb(qb));  
  
initial  
begin  
clk = 0;  
rst = 1;  
d = 0;  
#30 d=1;  
#30 rst=0;  
#30 d=0;  
#30 d=1;  
#100 $finish;  
end  
  
always  
#10 clk=~clk;  
endmodule
```

### **User Constraint File:**

```
#PINLOCK_BEGIN  
  
NET "clk" LOC = "S:PIN1";  
NET "rst" LOC = "S:PIN2";  
NET "d" LOC = "S:PIN3";  
NET "q" LOC = "S:PIN4";  
NET "qb" LOC = "S:PIN5";  
#PINLOCK_END
```

### **Simulation Result:**



### up/down(BCD and Binary) counters

```

module upordown_counter(
    Clk,
    reset,
    UpOrDown, //high for UP counter and low for Down counter
    Count
);

//input ports and their sizes
input Clk,reset,UpOrDown;
//output ports and their size
output [3 : 0] Count;
//Internal variables
reg [3 : 0] Count = 0;

always @(posedge(Clk) or posedge(reset))
begin
    if(reset == 1)
        Count <= 0;
    else
        if(UpOrDown == 1) //Up mode selected
            if(Count == 15)
                Count <= 0;
            else
                Count <= Count + 1; //Incremend Counter
        else //Down mode selected
            if(Count == 0)

```

```

    Count <= 15;
else
    Count <= Count - 1; //Decrement counter
end

endmodule

```

**Testbench for counter:**

```

module tb_counter;

// Inputs
reg Clk;
reg reset;
reg UpOrDown;

// Outputs
wire [3:0] Count;

// Instantiate the Unit Under Test (UUT)
upordown_counter uut (
    .Clk(Clk),
    .reset(reset),
    .UpOrDown(UpOrDown),
    .Count(Count)
);

//Generate clock with 10 ns clk period.
initial Clk = 0;
always #5 Clk = ~Clk;

initial begin
    // Apply Inputs
    reset = 0;
    UpOrDown = 0;
    #300;
    UpOrDown = 1;

```

```

#300;
    reset = 1;
    UpOrDown = 0;
#100;
    reset = 0;
end

```

```
endmodule
```

### Simulation waveform:



**verilog program to interface a stepper motor to the FGPA/CPLD and rotate the motor in the specified direction(by N steps)**

### PROGRAM:

```

`timescale 1ns / 1ps

module stepper( input clk,rst, inout [7:0] kb, output reg[7:0] sldata );
reg tclk;
reg[15:0] clkdiv=16'd0;
reg[1:0]sts=2'd0;
reg[3:0]coil=4'b00001;
reg[7:0] i1=8'd0;
reg[7:0] i2=8'd0;
reg[7:0] i3=8'd0;
reg[3:0] N=4'd15;
reg[2:0] stdir;
assign kb[7:3]=5'b00001;
always@(posedge clk)
begin
clkdiv=clkdiv+1;
tclk=clkdiv[15];
end
always@(posedge tclk)
begin

```

```
if(rst==1)
begin
end
i1=0;
i2=0;
i3=0;
case (kb[2:0])
3'b110:
if(i1!=N)
begin
sts=sts+1;
i1=i1+1;
end
3'b101:
if(i2!=N)
begin
sts=sts
-1;
i2=i2+1;
end
3'b011:
if(i3!=(N/2))
begin
sts=sts+1;
i3=i3+1;
end
endcase
end
always@(sts)
begin
case(sts)
2'b00: coil=4'b0001;
2'b01: coil=4'b0010;
2'b10: coil=4'b0100;
2'b11: coil=4'b1000;
default: coil=4'b0001;
endcase
```

```
smdata={4'b0000,coil};
```

```
end
```

```
endmodule
```

### User Constraint File:

```
NET "clk" LOC = "P53" ;  
NET "kb<0>" LOC = "P1" ;  
NET "kb<1>" LOC = "P2" ;  
NET "kb<2>" LOC = "P3" ;  
NET "kb<3>" LOC = "P4" ;  
NET "kb<4>" LOC = "P5" ;  
NET "kb<5>" LOC = "P6" ;  
NET "kb<6>" LOC = "P7" ;  
NET "kb<7>" LOC = "P9" ;  
NET "rst" LOC = "P54" ;  
NET "smdata<0>" LOC = "P10" ;  
NET "smdata<1>" LOC = "P11" ;  
NET "smdata<2>" LOC = "P12" ;  
NET "smdata<3>" LOC = "P13" ;  
NET "smdata<4>" LOC = "P14" ;  
NET "smdata<5>" LOC = "P15" ;  
NET "smdata<6>" LOC = "P17" ;  
NET "smdata<7>" LOC = "P18" ;
```

### EXPECTED RESULT:

1. Set the number of required steps (N) in the program.
2. By pressing sw1 of 4x4 hexa keypad, stepper motor will rotate in clockwise direction for N steps.
3. By pressing sw2 of 4x4 hexakeypad, stepper motor will rotate in anticlockwise direction for N steps.
4. By pressing sw3 of 4x4 hexakeypad, stepper motor will rotate in clockwise direction for (N/2) steps.

**verilog program to interface a relay or ADC to the FPGA/CPLD and demonstrate its working**

**DAC to the FPG/CPLD**

**VHDL code for SPI DAC**

```
library ieee;
```

```
use ieee.std_logic_1164.all;
use IEEE.NUMERIC_STD.ALL;
```

entity DAC is

```
port(
    CLK: in std_logic;
    CLK2: inout std_logic;
    SCK: out std_logic;
    CS: out std_logic;
    MOSI: out std_logic
);
end DAC;
```

architecture behavioral of DAC is

```
signal SEND: std_logic:='1';
```

```
signal i : integer range 0 to 255:=0;
signal VALUE: std_logic_vector (15 downto 0);
type memory is array (0 to 255) of std_logic_vector(15 downto 0);
constant sine : memory := (
X"8000", X"8324", X"8647", X"896a", X"8c8b", X"8fab", X"92c7", X"95e1",
X"98f8", X"9c0b", X"9f19", X"a223", X"a527", X"a826", X"ab1e", X"ae10",
X"b0fb", X"b3de", X"b6b9", X"b98c", X"bc56", X"bf16", X"c1cd", X"c47a",
X"c71c", X"c9b3", X"cc3f", X"cebf", X"d133", X"d39a", X"d5f4", X"d842",
X"da81", X"dcb3", X"ded6", X"e0eb", X"e2f1", X"e4e7", X"e6ce", X"e8a5",
X"ea6c", X"ec23", X"edc9", X"ef5e", X"f0e1", X"f254", X"f3b5", X"f503",
X"f640", X"f76b", X"f883", X"f989", X"fa7c", X"fb5c", X"fc29", X"fce2",
X"fd89", X"fe1c", X"fe9c", X"ff08", X"ff61", X"ffa6", X"ffd7", X"fff5",
X"ffff", X"fff5", X"ffd7", X"ffa6", X"ff61", X"ff08", X"fe9c", X"fe1c",
X"fd89", X"fce2", X"fc29", X"fb5c", X"fa7c", X"f989", X"f883", X"f76b",
X"f640", X"f503", X"f3b5", X"f254", X"f0e1", X"ef5e", X"edc9", X"ec23",
X"ea6c", X"e8a5", X"e6ce", X"e4e7", X"e2f1", X"e0eb", X"ded6", X"dcb3",
X"da81", X"d842", X"d5f4", X"d39a", X"d133", X"cebf", X"cc3f", X"c9b3",
X"c71c", X"c47a", X"c1cd", X"bf16", X"bc56", X"b98c", X"b6b9", X"b3de",
X"b0fb", X"ae10", X"ab1e", X"a826", X"a527", X"223", X"9f19", X"9c0b",
X"98f8", X"95e1", X"92c7", X"8fab", X"8c8b", X"896a", X"8647", X"8324",
X"8000", X"7cdb", X"79b8", X"7695", X"7374", X"7054", X"6d38", X"6a1e",
```

```
X"6707", X"63f4", X"60e6", X"5ddc", X"5ad8", X"57d9", X"54e1", X"51ef",
X"4f04", X"4c21", X"4946", X"4673", X"43a9", X"40e9", X"3e32", X"3b85",
X"38e3", X"364c", X"33c0", X"3140", X"2ecc", X"2c65", X"2a0b", X"27bd",
X"257e", X"234c", X"2129", X"1f14", X"1d0e", X"1b18", X"1931", X"175a",
X"1593", X"13dc", X"1236", X"10a1", X"0f1e", X"0dab", X"0c4a", X"0afc",
X"09bf", X"0894", X"077c", X"0676", X"0583", X"04a3", X"03d6", X"031d",
X"0276", X"01e3", X"0163", X"00f7", X"009e", X"0059", X"0028", X"000a",
X"0001", X"000a", X"0028", X"0059", X"009e", X"00f7", X"0163", X"01e3",
X"0276", X"031d", X"03d6", X"04a3", X"0583", X"0676", X"077c", X"0894",
X"09bf", X"0afc", X"0c4a", X"0dab", X"0f1e", X"10a1", X"1236", X"13dc",
X"1593", X"175a", X"1931", X"1b18", X"1d0e", X"1f1`4", X"2129", X"234c",
X"257e", X"27bd", X"2a0b", X"2c65", X"2ecc", X"3140", X"33c0", X"364c",
X"38e3", X"3b85", X"3e32", X"40e9", X"43a9", X"4673", X"4946", X"4c21",
X"4f04", X"51ef", X"54e1", X"57d9", X"5ad8", X"5ddc", X"60e6", X"63f4",
X"6707", X"6a1e", X"6d38", X"7054", X"7374", X"7695", X"79b8", X"7cdb"
```

```
);
```

```
signal SENDING : std_logic := '0';
```

```
signal reg : std_logic_vector (15 downto 0);
```

```
constant DELAY:integer := 3; -- 50 MHz / 3 == 16.667 MHz
```

```
constant IGNORE:std_logic := '0'; -- 0:use, 1:ignore
```

```
constant BUFFERED:std_logic := '0'; -- 0:unbuffered, 1:buffered
```

```
constant GAIN:std_logic := '1'; -- 0:2X, 1:1X
```

```
constant ACTIVE:std_logic := '1'; -- 0:shutdown, 1:active
```

```
begin
```

```
clkDiv : entity work.ClockDivider(Behavioral)
```

```
generic map(DELAY => DELAY)
```

```
port map (CLK, CLK2);
```

```
process(clk2)
```

```
begin
```

```
if(rising_edge(clk2)) then
```

```
    VALUE(15 downto 0) <= sine(i);
```

```

    i <= i+ 1;
    if(i = 255) then
        i <= 0;
    end if;
end if;
end process;

process(CLK2, SEND)

variable counter : integer range 0 to 15 := 0;

begin

if falling_edge(CLK2) then

    if SEND = '1' then
        reg <= IGNORE & BUFFERED & GAIN & ACTIVE & VALUE(15 downto 4);
        counter := 0;
        CS <= '0';
        SENDING <= '1';
        SEND <= '0';

    elsif SENDING = '1' then
        reg <= reg(14 downto 0) & '0';

            if counter = 15 then
                counter := 0;
                CS <= '1';
                SENDING <= '0';
                SEND <= '1';
            else
                counter := counter + 1;
            end if;
        end if;

    end if;
end process;

```

```
SCK <= CLK2 AND SENDING;
```

```
MOSI <= reg(15);
```

```
end behavioral;
```

Seven segment displays

```
module Seven_segment_LED_Display_Controller(
    input clock_100Mhz, // 100 Mhz clock source on Basys 3 FPGA
    input reset, // reset
    output reg [3:0] Anode_Activate, // anode signals of the 7-segment LED display
    output reg [6:0] LED_out// cathode patterns of the 7-segment LED display
);
reg [26:0] one_second_counter; // counter for generating 1 second clock enable
wire one_second_enable;// one second enable for counting numbers
reg [15:0] displayed_number; // counting number to be displayed
reg [3:0] LED_BCD;
reg [19:0] refresh_counter; // 20-bit for creating 10.5ms refresh period or 380Hz refresh rate
    // the first 2 MSB bits for creating 4 LED-activating signals with 2.6ms digit period
wire [1:0] LED_activating_counter;
    // count 0 -> 1 -> 2 -> 3
    // activates LED1 LED2 LED3 LED4
    // and repeat
always @(posedge clock_100Mhz or posedge reset)
begin
    if(reset==1)
        one_second_counter <= 0;
    else begin
        if(one_second_counter>=99999999)
            one_second_counter <= 0;
        else
            one_second_counter <= one_second_counter + 1;
    end
end
assign one_second_enable = (one_second_counter==99999999)?1:0;
always @(posedge clock_100Mhz or posedge reset)
begin
    if(reset==1)
```

```

displayed_number <= 0;
else if(one_second_enable==1)
    displayed_number <= displayed_number + 1;
end
always @(posedge clock_100Mhz or posedge reset)
begin
    if(reset==1)
        refresh_counter <= 0;
    else
        refresh_counter <= refresh_counter + 1;
end
assign LED_activating_counter = refresh_counter[19:18];
// anode activating signals for 4 LEDs, digit period of 2.6ms
// decoder to generate anode signals
always @(*)
begin
    case(LED_activating_counter)
        2'b00: begin
            Anode_Activate = 4'b0111;
            // activate LED1 and Deactivate LED2, LED3, LED4
            LED_BCD = displayed_number/1000;
            // the first digit of the 16-bit number
        end
        2'b01: begin
            Anode_Activate = 4'b1011;
            // activate LED2 and Deactivate LED1, LED3, LED4
            LED_BCD = (displayed_number % 1000)/100;
            // the second digit of the 16-bit number
        end
        2'b10: begin
            Anode_Activate = 4'b1101;
            // activate LED3 and Deactivate LED2, LED1, LED4
            LED_BCD = ((displayed_number % 1000)%100)/10;
            // the third digit of the 16-bit number
        end
        2'b11: begin
            Anode_Activate = 4'b1110;
            // activate LED4 and Deactivate LED2, LED3, LED1
            LED_BCD = ((displayed_number % 1000)%100)%10;
            // the fourth digit of the 16-bit number
        end
    endcase
end

```

```
    end
  endcase
end

// Cathode patterns of the 7-segment LED display
always @(*)
begin
  case(LED_BCD)
    4'b0000: LED_out = 7'b0000001; // "0"
    4'b0001: LED_out = 7'b1001111; // "1"
    4'b0010: LED_out = 7'b0010010; // "2"
    4'b0011: LED_out = 7'b0000110; // "3"
    4'b0100: LED_out = 7'b1001100; // "4"
    4'b0101: LED_out = 7'b0100100; // "5"
    4'b0110: LED_out = 7'b0100000; // "6"
    4'b0111: LED_out = 7'b0001111; // "7"
    4'b1000: LED_out = 7'b0000000; // "8"
    4'b1001: LED_out = 7'b0000100; // "9"
    default: LED_out = 7'b0000001; // "0"
  endcase
end
endmodule
```