EXPERIMENT-1

Objective: -

To Design and Implementation of Combinational Circuits (adder, decoder) in Gate Level

Software Used: -

Cadence Incisive

Theory: -

Half Adder

Half adder is a combinational logic circuit with two inputs and two outputs. The half adder circuit is designed to add two single bit binary number A and B. It is the basic building block for addition of two **single** bit numbers. This circuit has two outputs **carry** and **sum**.

Decoder

A decoder is a combinational circuit. It has n input and to a maximum m = 2n outputs. Decoder is identical to a demultiplexer without any data input. It performs operations which are exactly opposite to those of an encoder. Examples: - Code converters, BCD to seven segment decoders, Nixie tube decoders, Relay actuator.

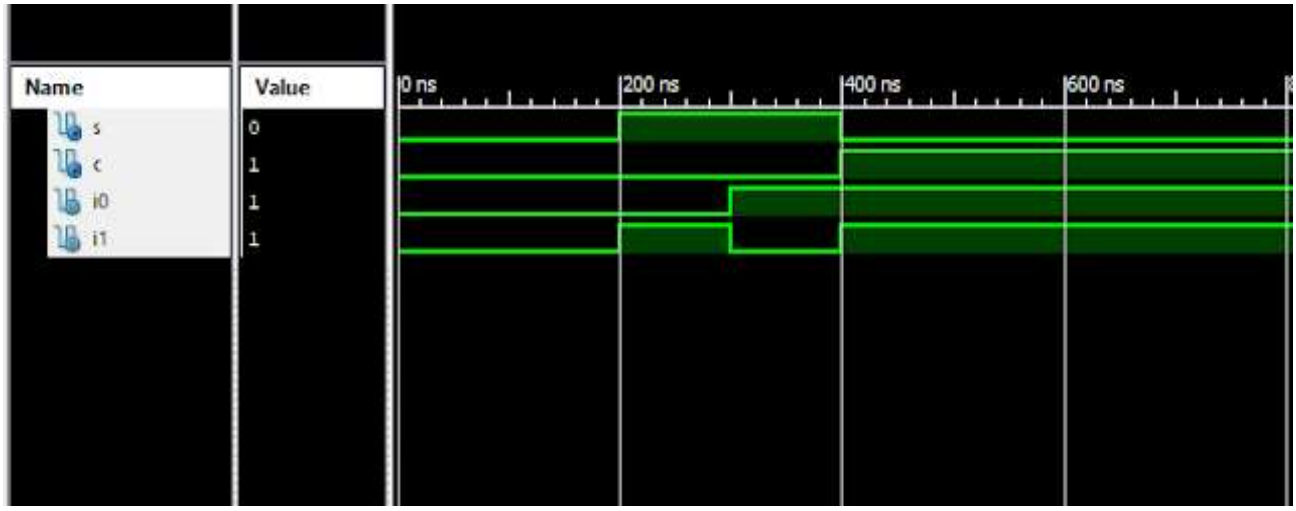
Half Adder Gate Level Modelling with Testbench: -

```
module half_adder(
   input i0,
   input i1,
   output s,
   output c
   );
            and(c,i0,i1);
            xor(s,i0,i1);
endmodule
```

Testbench: -

```
initial begin
                    // Initialize Inputs
                    i0 = 0;
                    i1 = 0;
                    // Wait 100 ns for global reset to finish
                    #100;
                    // Add stimulus here
                    #100; i0 = 0; i1 = 1;
                    #100; i0 = 1; i1 = 0;
                    #100; i0 = 1; i1 = 1;
end
```

Output: -



Decoder Gate Level Modelling with Testbench: -

```
module decoder24_gate(en,a,b,y);
    // declare input and output ports
    input en,a,b;
    output [3:0]y;
```

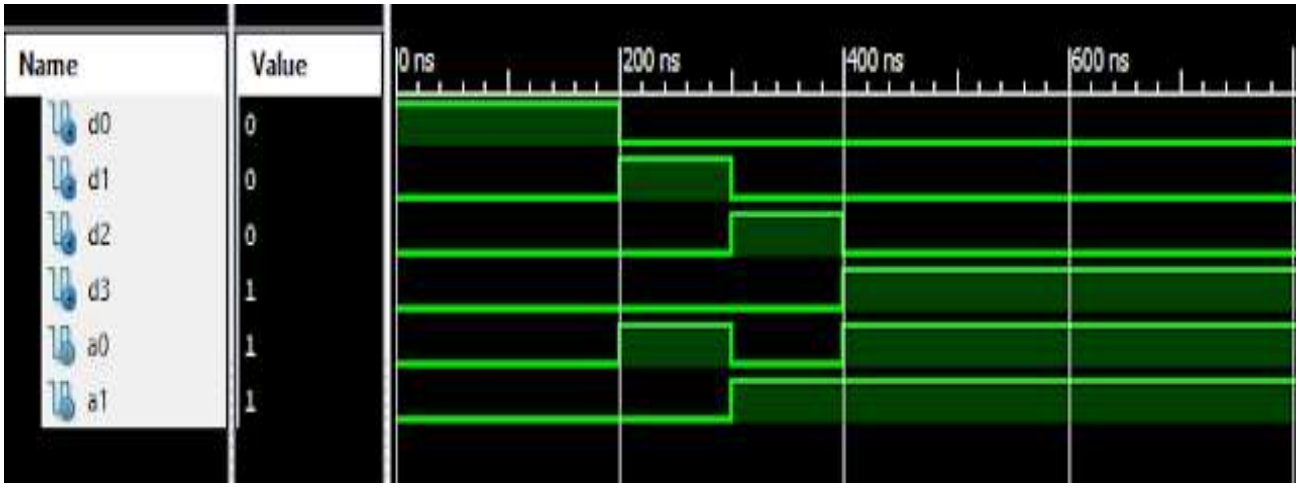
```
    // supportive connections required
    // to connect nand gates
    wire enb,na,nb;
    // instantiate 4 nand gates and 3 not gates
    // make connections by referring the above logic diagram
    not n0(enb,en);
    not n1(na,a);
    not n2(nb,b);
    nand n3(y[0],enb,na,nb);
    nand n4(y[1],enb,na,b);
    nand n5(y[2],enb,a,nb);
    nand n6(y[3],enb,a,b);
endmodule
```

Testbench: -

```
module tb;
  // input port are declared in reg(register)
  reg a,b,en;
  // output port are declared in wire(net)
  wire [3:0]y;
  // instantiate design block
  decoder24_gate dut(en,a,b,y);
  initial
    begin
      $monitor("en=%b a=%b b=%b y=%b",en,a,b,y);
      // with reference to truth table provide input values
```

```
    en=1;a=1'bx;b=1'bx;#5
    en=0;a=0;b=0;#5
    en=0;a=0;b=1;#5
    en=0;a=1;b=0;#5
    en=0;a=1;b=1;#5
    // terminate simulation using $finish system task
    $finish;
  end
endmodule
```

Output: -



EXPERIMENT-2

Objective: -

To Design and Implementation of Multiplexer and Comparator circuits in data-flow level

Software Used: -

Cadence Incisive

Theory: -

Multiplexer

A multiplexer is a device that selects one output from multiple inputs. It is also known as a data selector. We refer to a multiplexer with the terms ***MUX*** and ***MPX***. Multiplexers are used in communication systems to increase the amount of data sent over a network within a certain amount of time and bandwidth. It allows us to squeeze multiple data lines into one data line. It switches between one of the many input lines and combines them one by one to the output. It decides which input line to switch using a control signal.

Comparator

A decoder is a combinational circuit. It has n input and to a maximum m = 2n outputs. Decoder is identical to a demultiplexer without any data input. It performs operations which are exactly opposite to those of an encoder. Examples: - Code converters, BCD to seven segment decoders, Nixie tube decoders, Relay actuator.

Multiplexer Data Flow Level Modelling with Testbench: -

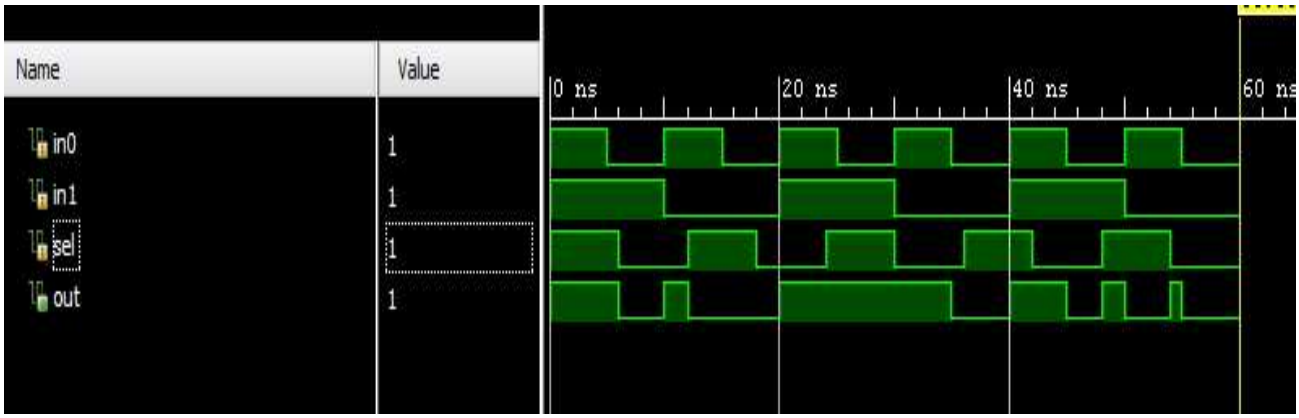
```
module mux2X1 (in0,in1,sel,out);
input in0, in1, sel;
output out;
```

```
assign out=(sel)?in1:in0;
endmodule
```

Testbench: -

```
module mux2X1_tb;
reg in0, in1;
reg sel;
wire out;
mux2X1 uut(.in0(in0), .in1(in1),.sel(sel),.out(out));
initial begin
    $monitor( "in0=%d, in1=%d, sel= %d, out=%d", in0,in1,sel,out);
    for (int i=0; i<8; i=i+1) begin
        {in0,in1,sel} = i;
        #10;
     end
end
endmodule
```

Output: -



Comparator Data Flow Level Modelling with Testbench: -

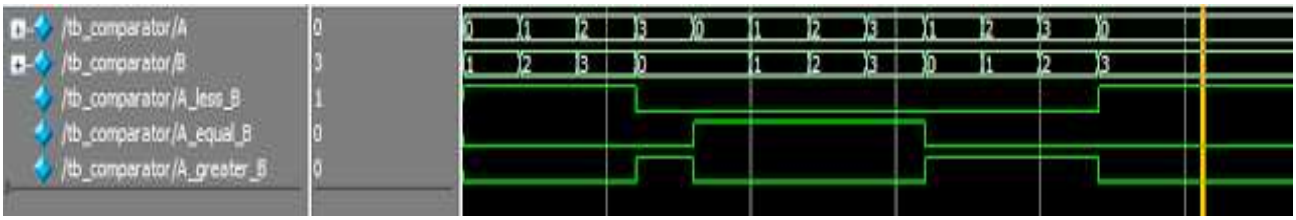
```
// Verilog code for 2-bit comparator
 module comparator(input [1:0] A,B, output A_less_B, A_equal_B,
A_greater_B);
 wire tmp1,tmp2,tmp3,tmp4,tmp5, tmp6, tmp7, tmp8;
 // A = B output
 xnor u1(tmp1,A[1],B[1]);
 xnor u2(tmp2,A[0],B[0]);
 and u3(A_equal_B,tmp1,tmp2);
 // A less than B output
 assign tmp3 = (~A[0])& (~A[1])& B[0];
 assign tmp4 = (~A[1])& B[1];
 assign tmp5 = (~A[0])& B[1]& B[0];
 assign A_less_B = tmp3 | tmp4 | tmp5;
 // A greater than B output
 assign tmp6 = (~B[0])& (~B[1])& A[0];
 assign tmp7 = (~B[1])& A[1];
 assign tmp8 = (~B[0])& A[1]& A[0];
 assign A_greater_B = tmp6 | tmp7 | tmp8;
 endmodule
 `timescale     10 ps/ 10 ps
```

Testbench: -

```
module tb_comparator;
 reg [1:0] A, B;
 wire A_less_B, A_equal_B, A_greater_B;
 integer i;
 // device under test
 comparator dut(A,B,A_less_B, A_equal_B, A_greater_B);
 initial begin
     for (i=0;i<4;i=i+1)
     begin
         A = i;
```

```
        B = i + 1;
        #20;
    end
    for (i=0;i<4;i=i+1)
    begin
        A = i;
        B = i;
        #20;
    end
    for (i=0;i<4;i=i+1)
    begin
        A = i+1;
        B = i;
        #20;
    end
end
endmodule
```

Output: -



EXPERIMENT-3

Objective: -

To Implement multiplexer, decoder in Behavioural Level

Software Used: -

Cadence Incisive

Theory: -

Multiplexer

A multiplexer is a device that selects one output from multiple inputs. It is also known as a data selector. We refer to a multiplexer with the terms ***MUX*** and ***MPX***. Multiplexers are used in communication systems to increase the amount of data sent over a network within a certain amount of time and bandwidth. It allows us to squeeze multiple data lines into one data line. It switches between one of the many input lines and combines them one by one to the output. It decides which input line to switch using a control signal.

Decoder

A decoder is a combinational circuit. It has n input and to a maximum m = 2n outputs. Decoder is identical to a demultiplexer without any data input. It performs operations which are exactly opposite to those of an encoder. Examples: - Code converters, BCD to seven segment decoders, Nixie tube decoders, Relay actuator.

Multiplexer Behavioural Level Modelling with Testbench: -

```
module m41 ( a, b, c, d, s0, s1, out);
input wire a, b, c, d;
input wire s0, s1;
output reg out;
```

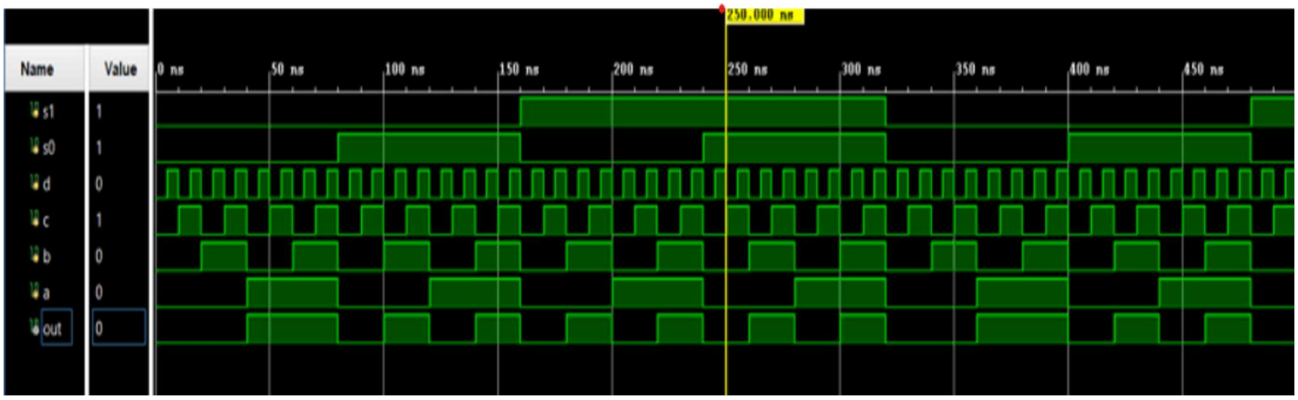
```
always @ (a or b or c or d or s0, s1)
begin
case (s0 | s1)
2'b00 : out <= a;
2'b01 : out <= b;
2'b10 : out <= c;
2'b11 : out <= d;
endcase
end
endmodule
```

Testbench: -

```
module top;
wire  out;
reg  a;
reg  b;
reg  c;
reg  d;
reg s0, s1;
m41 name(.out(out), .a(a), .b(b), .c(c), .d(d), .s0(s0), .s1(s1));
   initial
   begin
   a=1'b0; b=1'b0; c=1'b0; d=1'b0;
   s0=1'b0; s1=1'b0;
   #500 $finish;
end
always #40 a=~a;
always #20 b=~b;
```

```
always #10 c=~c;
always #5 d=~d;
always #80 s0=~s0;
always #160 s1=~s1;
always@(a or b or c or d or s0 or s1)
$monitor("At time = %t, Output = %d", $time, out);
endmodule;
```

Output: -



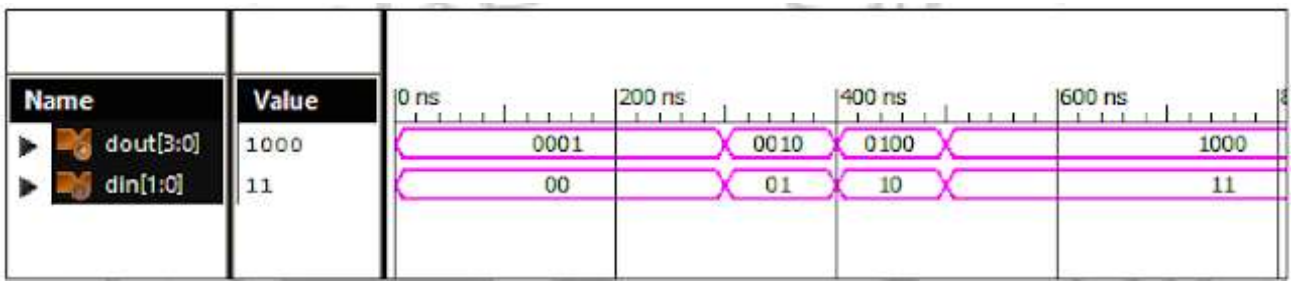
Decoder Behavioural Level Modelling with Testbench: -

```
module 2_4_DEC( input [1:0]din, output [3:0]dout );
reg [3:0]dout;
always @ (din)
case (din)
      0 : dout[0] = 1;
      1 : dout[1] = 1;
      2 : dout[2] = 1;
      3 : dout[3] = 1;
    default : dout = 4'bxxxx;
endcase
endmodule
```

Testbench: -

```
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;
end
initial begin
#100
$monitor("din=%b, dout=%b", din, dout);
end
endmodule
```

Output: -



EXPERIMENT- 4

Objective: -

To Design and Implementation of ALU

Software Used: -

Cadence Incisive

Theory: -

ALU

An arithmetic logic unit (ALU) is a digital circuit used to perform arithmetic and logic operations. It represents the fundamental building block of the central processing unit (CPU) of a computer. Modern CPUs contain very powerful and complex ALUs. In addition to ALUs, modern CPUs contain a control unit (CU). Most of the operations of a CPU are performed by one or more ALUs, which load data from input registers. A register is a small amount of storage available as part of a CPU. The control unit tells the ALU what operation to perform on that data, and the ALU stores the result in an output register. The control unit moves the data between these registers, the ALU, and memory.

Multiplexer Behavioural Level Modelling with Testbench: -

```
module alu(
        input [7:0] A,B,  // ALU 8-bit Inputs
        input [3:0] ALU_Sel,// ALU Selection
        output [7:0] ALU_Out, // ALU 8-bit Output
        output CarryOut // Carry Out Flag
    );
    reg [7:0] ALU_Result;
    wire [8:0] tmp;
    assign ALU_Out = ALU_Result; // ALU out
    assign tmp = {1'b0,A} + {1'b0,B};
    assign CarryOut = tmp[8]; // Carryout flag
```

```
always @(*)
begin
    case(ALU_Sel)
    4'b0000: // Addition
       ALU_Result = A + B ;
    4'b0001: // Subtraction
       ALU_Result = A - B ;
    4'b0010: // Multiplication
       ALU_Result = A * B;
    4'b0011: // Division
       ALU_Result = A/B;
    4'b0100: // Logical shift left
       ALU_Result = A<<1;
     4'b0101: // Logical shift right
       ALU_Result = A>>1;
     4'b0110: // Rotate left
       ALU_Result = {A[6:0],A[7]};
     4'b0111: // Rotate right
       ALU_Result = {A[0],A[7:1]};
      4'b1000: //  Logical and
       ALU_Result = A & B;
      4'b1001: //  Logical or
       ALU_Result = A | B;
      4'b1010: //  Logical xor
       ALU_Result = A ^ B;
      4'b1011: //  Logical nor
       ALU_Result = ~(A | B);
      4'b1100: // Logical nand
       ALU_Result = ~(A & B);
      4'b1101: // Logical xnor
       ALU_Result = ~(A ^ B);
      4'b1110: // Greater comparison
       ALU_Result = (A>B)?8'd1:8'd0 ;
      4'b1111: // Equal comparison
        ALU_Result = (A==B)?8'd1:8'd0 ;
      default: ALU_Result = A + B ;
```

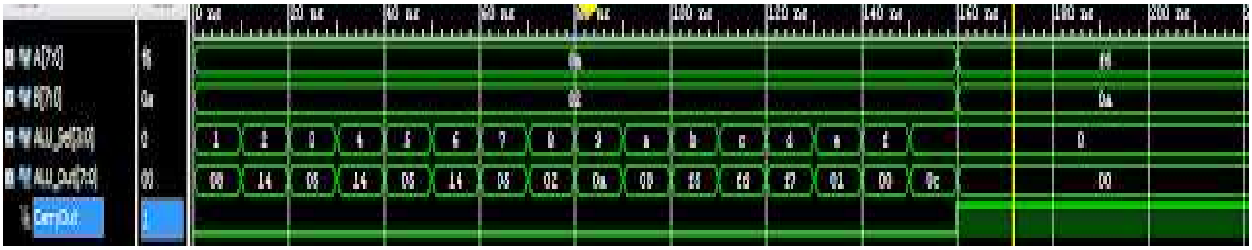
```
        endcase
    end

endmodule
```

Testbench: -

```
`timescale 1ns / 1ps

module tb_alu;
//Inputs
 reg[7:0] A,B;
 reg[3:0] ALU_Sel;

//Outputs
 wire[7:0] ALU_Out;
 wire CarryOut;
 // Verilog code for ALU
 integer i;
 alu test_unit(
          A,B,  // ALU 8-bit Inputs
          ALU_Sel,// ALU Selection
          ALU_Out, // ALU 8-bit Output
          CarryOut // Carry Out Flag
     );
    initial begin
    // hold reset state for 100 ns.
      A = 8'h0A;
      B = 4'h02;
      ALU_Sel = 4'h0;
      for (i=0;i<=15;i=i+1)
      begin
       ALU_Sel = ALU_Sel + 8'h01;
       #10;
      end;
```

```
    A = 8'hF6;
    B = 8'h0A;
  end
endmodule
```

Output: -



EXPERIMENT- 5

Objective: -

To Implement D-FF and JK FF in behavioral level

Software Used: -

Cadence Incisive

Theory: -

A flip flop can store one bit of data. Hence, it is known as a memory cell. Flip-flops are synchronous circuits since they use a clock signal. Using flip flops, we build complex circuits such as RAMs, Shift Registers, etc.

A D flip-flop stands for data or delay flip-flop. The outputs of this flip-flop are equal to the inputs.

The J-K flip-flop is the most versatile of the basic flip flops. The JK flip flop is a gated SR flip-flop with the addition of a clock input circuitry that prevents the illegal or invalid output condition that can occur when both inputs S and R are equal to logic 1. Due to this additional clocked input, a JK flip-flop has four possible input combinations, “logic 1”, “logic 0”, “no change” and “toggle”.

D Flip Flop Behavioral Level Modelling with Testbench: -

```
module dff_behavioral(d,clk,clear,q,qbar); // Synchronous
input d, clk, clear;
output reg q, qbar;
always@(posedge clk)
begin
if(clear== 1)
q <= 0;
qbar <= 1;
```

```
else
q <= d;
qbar = !d;
end
endmodule
module dff_behavioral(d,clk,clear,q,qbar); // Asynchronous
input d, clk, clear;
output reg q, qbar;
always@(posedge clk or posedge clear)
begin
if(clear== 1)
q <= 0;
qbar <= 1;
else
q <= d;
qbar = !d;
end
endmodule
```

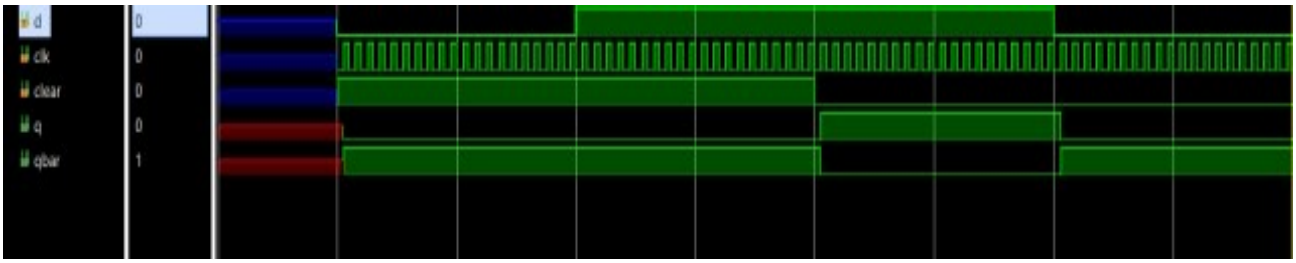
Testbench: -

```
module dff_test;
reg D, CLK,reset;
wire Q, QBAR;
dff_behavior dut(.q(Q), .qbar(QBAR), .clear(reset), .d(D), .clk(CLK)); //
instantiation by port name.
$monitor("simtime = %g, CLK = %b, D = %b,reset = %b, Q = %b, QBAR =
%b", $time, CLK, D, reset, Q, QBAR);
initial begin
  clk=0;
```

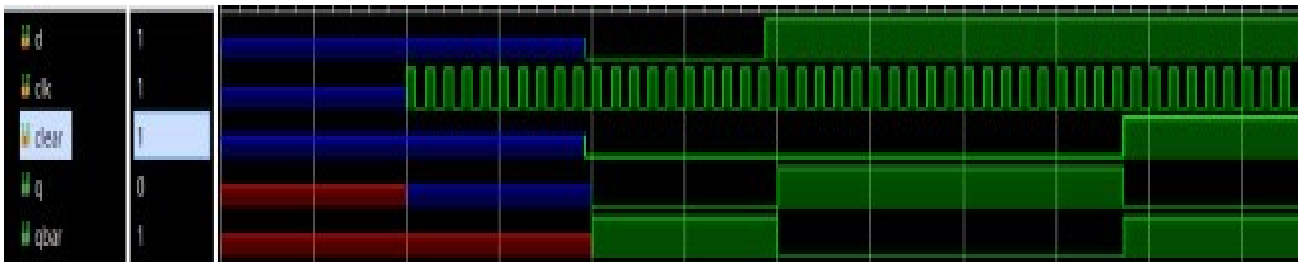
```
    forever #10 clk = ~clk;
end
initial begin
 reset=1; D <= 0;
 #100; reset=0; D <= 1;
 #100; D <= 0;
 #100; D <= 1;
end
endmodule
```

Output: -

D Flip Flop (Synchronous)



D Flip Flop (Asynchronous)



JK Flip Flop Behavioral Level Modelling with Testbench: -

```
module jkff_behave(clk,j,knq,qbar);
input clk,j,k;
output reg q,qbar;
always@(posedge clk)
```

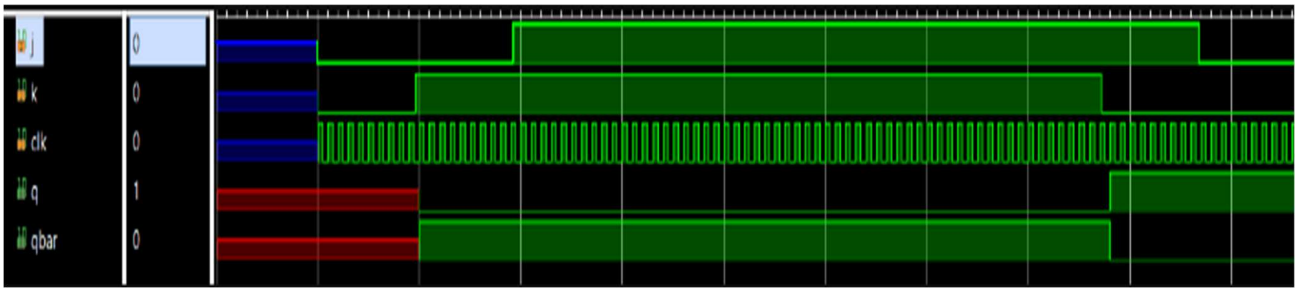
```
begin
if(k = 0)
begin
q <= 0;
qbar <= 1;
end
always@(posedge clk)
begin
  if(k = 0)
   begin
    q <= 0;
    qbar <= 1;
   end
  else if(j = 1)
   begin
    q <= 0;
    qbar <= 0;
   end
  else if(j = 0 & k = 0)
   begin
    q <= q;
    qbar <= qbar;
   end
  else if(j = 1 & k = 1)
   begin
    q <= ~q;
    qbar <= ~qbar;
```

```
   end
end
endmodule
```

Testbench: -

```
module jkff_test;
reg J,K, CLK;
wire Q, QBAR;
jkff_behavior dut(.q(Q), .qbar(QBAR), .j(J), .k(K), .clk(CLK));
$monitor("simtime = %g, CLK = %b, J = %b, K = %b, Q = %b, QBAR =
%b", $time, CLK, J, K, Q, QBAR);
initial begin
  clk=0;
     forever #10 clk = ~clk;
end
initial begin
 J= 1; K= 0;
 #100; J= 0; K= 1;
 #100; J= 0; K= 0;
 #100; J= 1; K=1;
end
endmodule
```

Output: -



EXPERIMENT- 6

Objective: -

To Design and Implementation of Shift Register

Software Used: -

Cadence Incisive

Theory: -

In digital electronics, a shift register is a cascade of flip-flops where the output pin q of one flop is connected to the data input pin (d) of the next. Because all flops work on the same clock, the bit array stored in the shift register will shift by one position. For example, if a 5-bit right shift register has an initial value of 10110 and the input to the shift register is tied to 0, then the next pattern will be 01011 and the next 00101.

Shift Register Modelling with Testbench: -

```
module free_run_shift_reg
  #(parameter N=8)
  (
   input wire clk, reset,
   input wire s_in,
   output wire s_out
  );
  reg [N-1:0] r_reg;
  wire [N-1:0] r_next;
  always @(posedge clk, negedge reset)
  begin
    if (~reset)
      r_reg <= 0;
```

```
    else
      r_reg <= r_next;
      end
      assign r_next = {s_in, r_reg[N-1:1]};
      assign s_out = r_reg[0];
endmodule
```

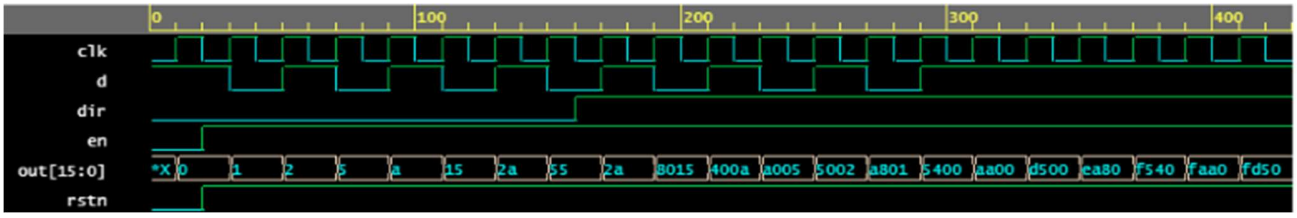
Testbench: -

```
`timescale 1ns / 1ps
module stimulus;
      // Inputs
      reg clk ;
      reg reset;
      // Outputs
      reg s_in;
      wire s_out;
      // Instantiate the Unit Under Test (UUT)
      free_run_shift_reg  #(2) s1  (
            .clk(clk),
            .reset(reset),
            .s_in(s_in),
            .s_out(s_out)
      );

 integer i, j;
 initial
 begin
```

```
 clk = 0;
 for(i =0; i<=40; i=i+1)
 begin
  #10 clk = ~clk;
 end
 end

initial
begin

$dumpfile("test.vcd");
$dumpvars(0,stimulus);

s_in = 0; reset =1;
#2 s_in = 0 ; reset = 0;
#2 reset =1;
 for(i =0; i<=10; i=i+1)
 begin
  #20 s_in = ~s_in;
end
#20 s_in =1;
#20 s_in = 1;
#20 s_in =0;
#20 s_in =1;
#20 s_in = 1;
#20 s_in =0;
```

```
#20 s_in =1;
#20 s_in = 1;
#20 s_in =0;
 end

                initial begin
                 $monitor("clk=%d s_in=%d,s_out=%d",clk,s_in, s_out);
                 end

endmodule
```

Output: -



EXPERIMENT- 7

Objective: -

To Design and Implementation of Asynchronous Counter.

Software Used: -

Cadence Incisive

Theory: -

Asynchronous counters are those whose output is free from the clock signal. Because the flip flops in asynchronous counters are supplied with different clock signals, there may be delay in producing output. The required number of logic gates to design asynchronous counters is very less. So they are simple in design.

Asynchronous Counter with Testbench: -

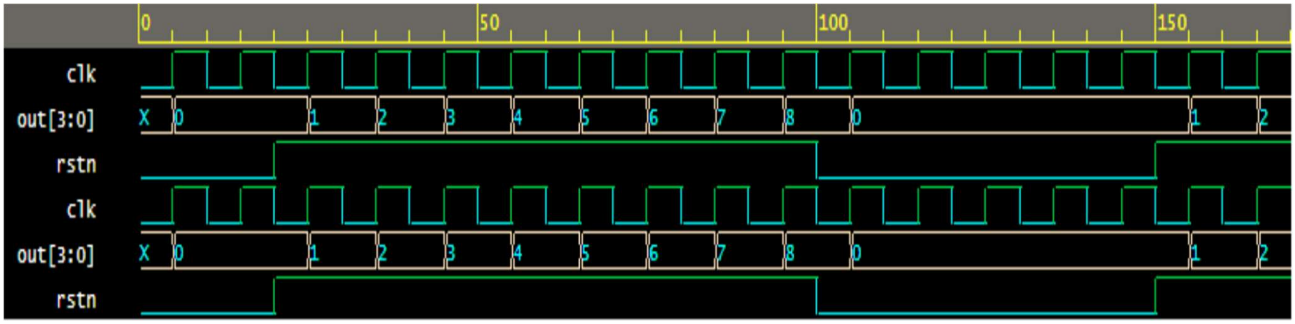
```
module counter ( input clk, input rstn, output reg[3:0] out);
always @ (posedge clk) begin
  if (! rstn)
    out <= 0;
  else
    out <= out + 1;
  end
endmodule
```

Testbench: -

```
module tb_counter; reg clk; reg rstn; wire [3:0] out;
counter c0 ( .clk (clk), .rstn (rstn), .out (out));
always #5 clk = ~clk;
initial begin
   clk <= 0;
   rstn <= 0;
```

```
  #20 rstn <= 1;
  #80 rstn <= 0;
  #50 rstn <= 1;
  #20 $finish;
 end
endmodule
```

Output:-



EXPERIMENT- 8

Objective: -

To Design and Implementation of Synchronous Counter.

Software Used: -

Cadence Incisive

Theory: -

The synchronous counter also referred to as a parallel counter is the one in which each establishing flip flops are clocked with the similar clock input at the same time. In the synchronous counter, all the flip-flops in the cascade network are independently linked to an external clock.

Synchronous Counter with Testbench: -

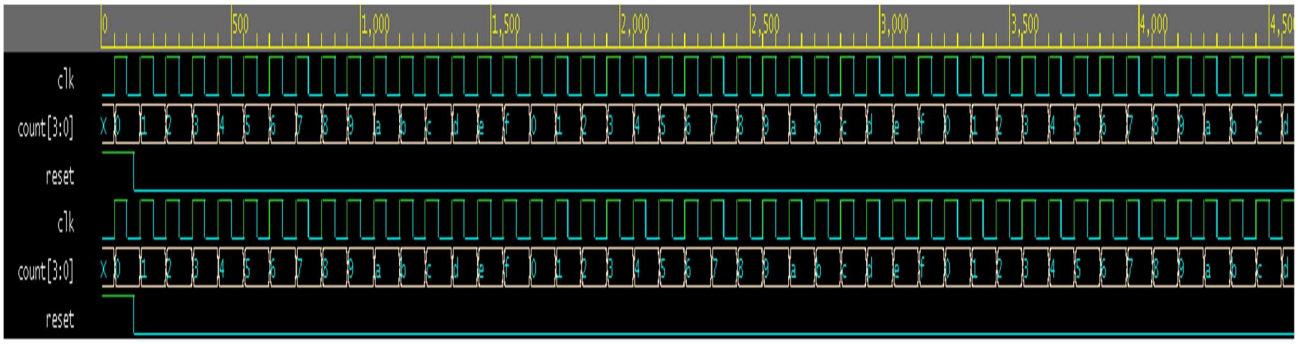
```
module synchronouscounter(clk,reset,count);
        input clk,reset;
        output reg [3:0] count;
       always@(posedge clk)
       begin
             if(reset)
                   count <= 4'b0000;
             else
                   count <= count + 1;
       end
endmodule
```

Testbench: -

```
module synchronouscountertb;
        reg clk, reset;
        wire [3:0] count;
        synchronouscounter synccntr(clk,reset,count);
```

```
        initial
        begin
                clk = 0;
                reset = 1;
                #125
                reset = 0;
                #4500;
                $finish;
        end
        always #50 clk=~clk;
        always@(posedge clk)
        begin
                $display ("value is %d", count);
        end
endmodule
```

Output: -



EXPERIMENT- 9

Objective: -

To Design and Implementation of Sequence Detector in FSM.

Software Used: -

Cadence Incisive

Theory: -

A sequence detector is a sequential circuit that outputs 1 when a particular pattern of bits sequentially arrives at its data input. The figure below shows a block diagram of a sequence detector. It has two inputs and one output.

Sequence Detector Verilog code with Testbench: -

```
module det_1011 ( input clk,
                  input rstn,
                  input in,
                  output out );

 parameter IDLE      = 0,
                     S1          = 1,
                     S10   = 2,
                     S101        = 3,
                     S1011       = 4;

 reg [2:0] cur_state, next_state;

 assign out = cur_state == S1011 ? 1 : 0;

 always @ (posedge clk) begin
   if (!rstn)
```

```
        cur_state <= IDLE;
    else
        cur_state <= next_state;
 end

 always @ (cur_state or in) begin
   case (cur_state)
     IDLE : begin
       if (in) next_state = S1;
       else next_state = IDLE;
     end

     S1: begin
`ifdef BUG_FIX
       if (in) next_state = S1;
`else
       if (in) next_state = IDLE;
`endif
       else    next_state = S10;
     end

     S10 : begin
       if (in) next_state = S101;
       else    next_state = IDLE;
     end

     S101 : begin
       if (in) next_state = S1011;
`ifdef BUG_FIX
               else   next_state = S10;
`else
```

```
      else    next_state = IDLE;
`endif
    end

    S1011: begin
      next_state = IDLE;
    end
  endcase
 end
endmodule
```

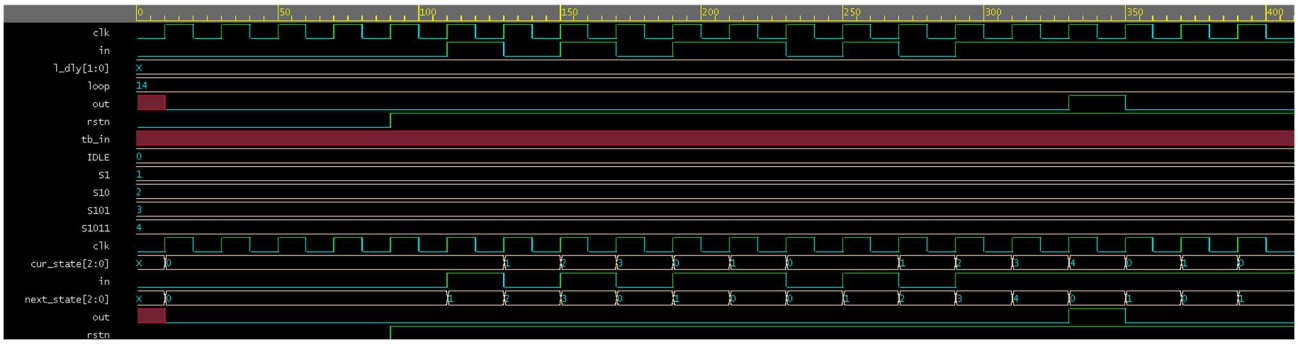
Testbench: -

```
module tb;
 reg                clk, in, rstn;
 wire                     out;
 reg [1:0] l_dly;
 reg                tb_in;
 integer    loop = 20;

 always #10 clk = ~clk;

 det_1011 u0 ( .clk(clk), .rstn(rstn), .in(in), .out(out) );

 initial begin
      clk <= 0;
  rstn <= 0;
  in <= 0;

  repeat (5) @ (posedge clk);
  rstn <= 1;
```

```
`ifndef RANDOM
        // Generate a directed pattern
    @(posedge clk) in <= 1;
    @(posedge clk) in <= 0;
    @(posedge clk) in <= 1;
    @(posedge clk) in <= 0;
    @(posedge clk) in <= 1;
    @(posedge clk) in <= 1;
    @(posedge clk) in <= 0;
    @(posedge clk) in <= 1;
    @(posedge clk) in <= 0;
    @(posedge clk) in <= 1;
    @(posedge clk) in <= 1;

`else
    // Or random stimulus using a for loop that drives a random
    // value of input N times, but there is lesser chance to hit
    // this pattern
    for (int i = 0 ; i < loop; i ++) begin
      l_dly = $random;
      repeat (l_dly) @ (posedge clk);
      tb_in = $random;
      in <= tb_in;
    end
`endif

    #100 $finish;
  end

  always @ (posedge clk) begin
    $strobe ("T=%0t in=%0b out=%0b", $time, in, out);
```

```
  end
endmodule
```

Output: -



EXPERIMENT- 10

Objective: -

To Design and Implementation of Traffic Light Signal System in FSM.

Software Used: -

Cadence Incisive

Theory: -

Traffic light simulation is a classic text book problem used to demonstrate a finite state machine. A FSM is a system with finite states, finite inputs, finite outputs. It has a known number of states, inputs, and outputs.

Traffic Light Signal Verilog code with Testbench: -

```
module traffic_light(light_highway, light_farm, C, clk, rst_n);
parameter HGRE_FRED=2'b00, // Highway green and farm red
   HYEL_FRED = 2'b01,// Highway yellow and farm red
   HRED_FGRE=2'b10,// Highway red and farm green
   HRED_FYEL=2'b11;// Highway red and farm yellow
input C, // sensor
   clk, // clock = 50 MHz
   rst_n; // reset active low
output reg[2:0] light_highway, light_farm; // output of lights
// fpga4student.com FPGA projects, VHDL projects, Verilog projects
reg[27:0] count=0,count_delay=0;
reg delay10s=0,
delay3s1=0,delay3s2=0,RED_count_en=0,YELLOW_count_en1=0,YELLO
W_count_en2=0;
wire clk_enable; // clock enable signal for 1s
```

```
reg[1:0] state, next_state;
// next state
always @(posedge clk or negedge rst_n)
begin
if(~rst_n)
 state <= 2'b00;
else
 state <= next_state;
end
// FSM
always @(*)
begin
case(state)
HGRE_FRED: begin // Green on highway and red on farm way
 RED_count_en=0;
 YELLOW_count_en1=0;
 YELLOW_count_en2=0;
 light_highway = 3'b001;
 light_farm = 3'b100;
 if(C) next_state = HYEL_FRED;
 // if sensor detects vehicles on farm road,
 // turn highway to yellow -> green
 else next_state =HGRE_FRED;
end
HYEL_FRED: begin// yellow on highway and red on farm way
  light_highway = 3'b010;
  light_farm = 3'b100;
```

```
 RED_count_en=0;
 YELLOW_count_en1=1;
 YELLOW_count_en2=0;
  if(delay3s1) next_state = HRED_FGRE;
  // yellow for 3s, then red
  else next_state = HYEL_FRED;
end
HRED_FGRE: begin// red on highway and green on farm way
 light_highway = 3'b100;
 light_farm = 3'b001;
 RED_count_en=1;
 YELLOW_count_en1=0;
 YELLOW_count_en2=0;
 if(delay10s) next_state = HRED_FYEL;
 // red in 10s then turn to yello -> green again for high way
 else next_state =HRED_FGRE;
end
HRED_FYEL:begin// red on highway and yellow on farm way
 light_highway = 3'b100;
 light_farm = 3'b010;
 RED_count_en=0;
 YELLOW_count_en1=0;
 YELLOW_count_en2=1;
 if(delay3s2) next_state = HGRE_FRED;
 // turn green for highway, red for farm road
 else next_state =HRED_FYEL;
end
```

```
default: next_state = HGRE_FRED;
endcase
end
// fpga4student.com FPGA projects, VHDL projects, Verilog projects
// create red and yellow delay counts
always @(posedge clk)
begin
if(clk_enable==1) begin
 if(RED_count_en||YELLOW_count_en1||YELLOW_count_en2)
  count_delay <=count_delay + 1;
  if((count_delay == 9)&&RED_count_en)
  begin
   delay10s=1;
   delay3s1=0;
   delay3s2=0;
   count_delay<=0;
  end
  else if((count_delay == 2)&&YELLOW_count_en1)
  begin
   delay10s=0;
   delay3s1=1;
   delay3s2=0;
   count_delay<=0;
  end
  else if((count_delay == 2)&&YELLOW_count_en2)
  begin
   delay10s=0;
```

```
   delay3s1=0;
   delay3s2=1;
   count_delay<=0;
  end
  else
  begin
   delay10s=0;
   delay3s1=0;
   delay3s2=0;
  end
 end
end
// create 1s clock enable
always @(posedge clk)
begin
 count <=count + 1;
 //if(count == 50000000) // 50,000,000 for 50 MHz clock running on real FPGA
 if(count == 3) // for testbench
  count <= 0;
end
 assign clk_enable = count==3 ? 1: 0; // 50,000,000 for 50MHz running on FPGA
endmodule
```

Testbench: -

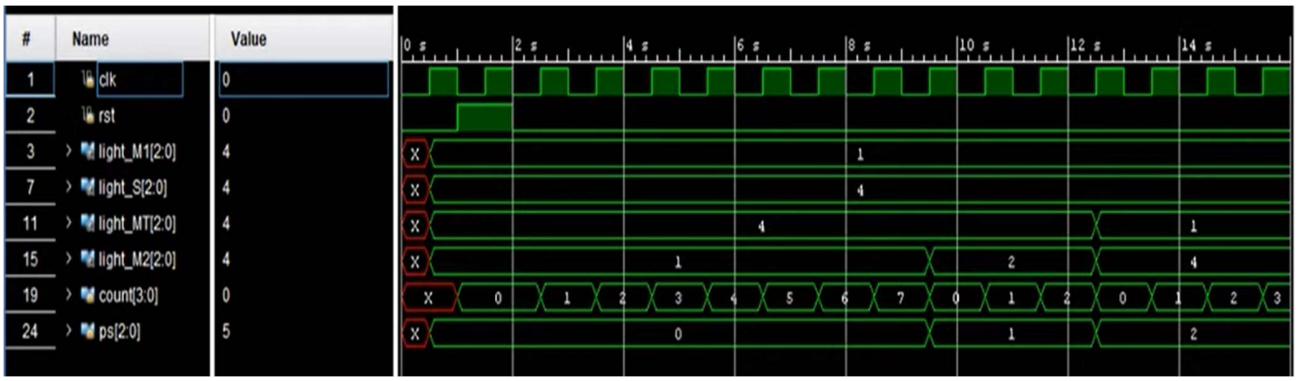
```
`timescale 10 ns/ 1 ps
// 2. Preprocessor Directives
```

```
`define DELAY 1
// 3. Include Statements
//`include "counter_define.h"
module tb_traffic;
// 4. Parameter definitions
parameter ENDTIME  = 400000;
// 5. DUT Input regs
//integer count, count1, a;
reg clk;
reg rst_n;
reg sensor;
wire [2:0] light_farm;
// 6. DUT Output wires
wire [2:0] light_highway;

// fpga4student.com FPGA projects, VHDL projects, Verilog projects
// 7. DUT Instantiation
traffic_light tb(light_highway, light_farm, sensor, clk, rst_n);

// 8. Initial Conditions
initial
 begin
 clk = 1'b0;
 rst_n = 1'b0;
 sensor = 1'b0;
 // count = 0;
//// count1=0;
// a=0;
 end
// 9. Generating Test Vectors
initial
```

```
 begin
 main;
 end
task main;
 fork
 clock_gen;
 reset_gen;
 operation_flow;
 debug_output;
 endsimulation;
 join
endtask
task clock_gen;
 begin
 forever #`DELAY clk = !clk;
 end
endtask
task reset_gen;
 begin
 rst_n = 0;
 # 20
 rst_n = 1;
 end
endtask
// fpga4student.com FPGA projects, VHDL projects, Verilog projects
task operation_flow;
 begin
 sensor = 0;
 # 600
 sensor = 1;
 # 1200
```

```
 sensor = 0;
 # 1200
 sensor = 1;
 end
endtask
// 10. Debug output
task debug_output;
 begin
 $display("-----------------------------------------");
       $display("-----------------     ----------------------");
 $display("---------- SIMULATION RESULT ---------------");
 $display("-------------           -------------------");
 $display("---------------         --------------------");
 $display("-----------------------------------------");
 $monitor("TIME = %d, reset = %b, sensor = %b, light of highway = %h,
light of farm road = %h",$time,rst_n ,sensor,light_highway,light_farm );
 end
endtask
// fpga4student.com FPGA projects, VHDL projects, Verilog projects
//12. Determines the simulation limit
task endsimulation;
 begin
 #ENDTIME
 $display("------------- THE SIMUALTION END ------------");
 $finish;
 end
endtask
endmodule
```

Output: -


#
Name
Value
1
clk
0
2
rst
0
3
light_M1[2:0]
4
7
light_S[2:0]
4
11
light_MT[2:0]
4
15
light_M2[2:0]
4
19
count[3:0]
0
24
ps[2:0]
5
0 s
2 s
4 s
6 s
8 s
10 s
12 s
14 s