VLSI COE - VLSI Design Methodologies (LAB 1-LAB 6)

LAB 1

HALF ADDER:

RTL CODE:

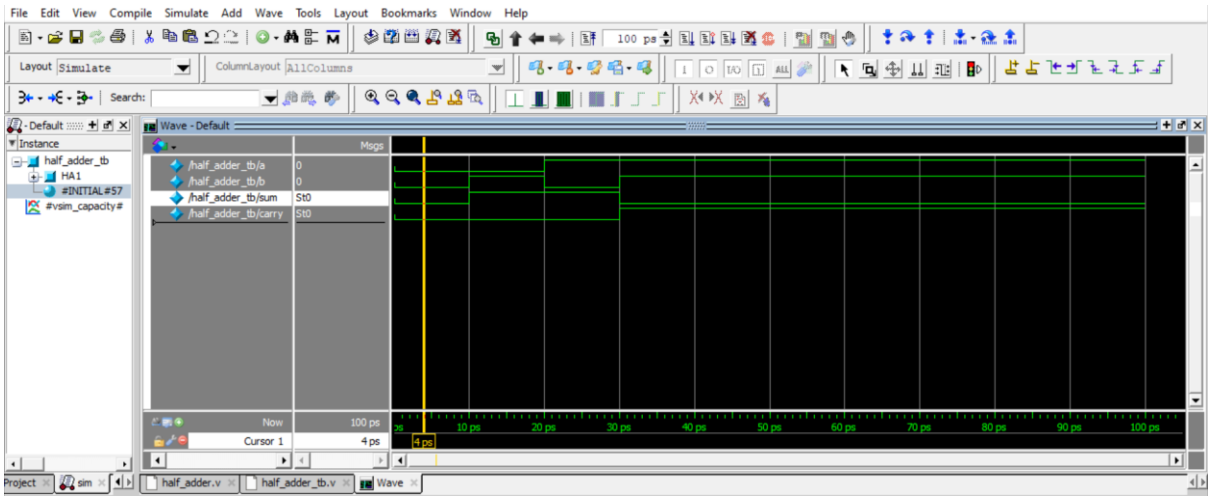
```
module half_adder(a,
                  b,
                  sum,
                  carry);
  input  a,
         b;
  output sum,
         carry;
  assign sum = a ^ b;
  assign carry = a & b;
endmodule
```

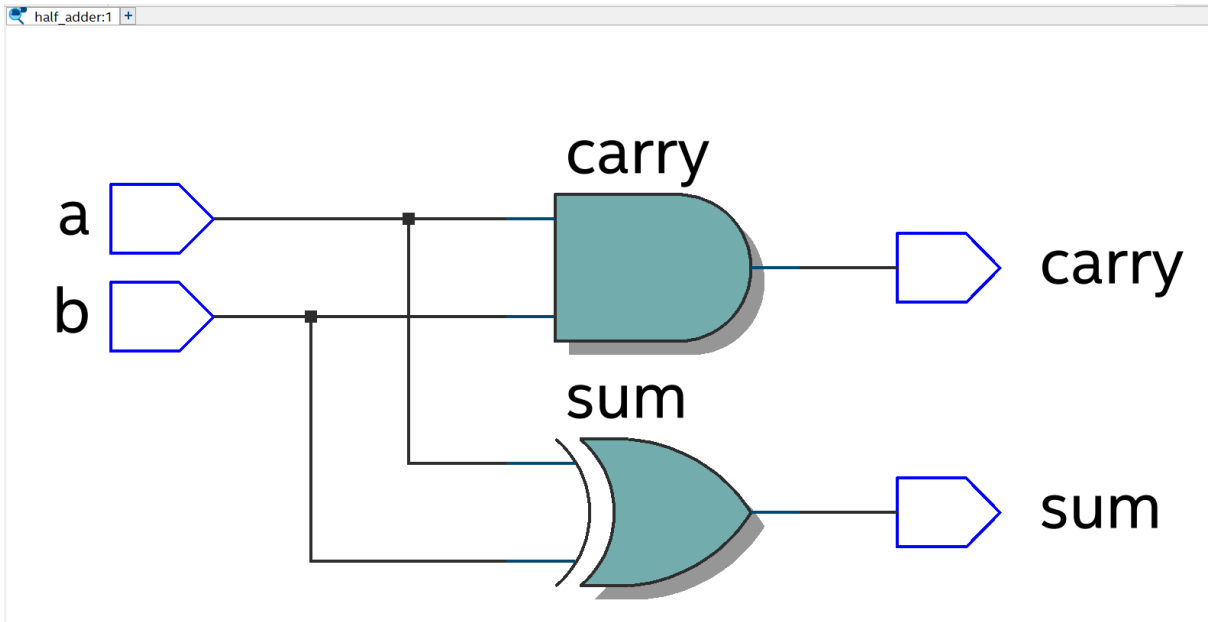
TESTBENCH:

```
module half_adder_tb();
  reg   a,b;
  wire  sum,carry;
  integer i;
  //Instantiating the half adder
  half_adder HA1(.a(a), .b(b), .sum(sum), .carry(carry));
  initial
  begin
    a   = 1'b0;
    b   = 1'b0;
  end
  initial
  begin
```

```
    for (i=0;i<4;i=i+1)
    begin
      {a,b}=i;
      #10;
    end
  end
  initial $monitor("Input a=%b, b=%b, Output sum =%b, carry=%b",
                   a,b,sum,carry);
  initial #100 $finish;
endmodule
```

OUTPUT:





FULL ADDER:

RTL CODE:

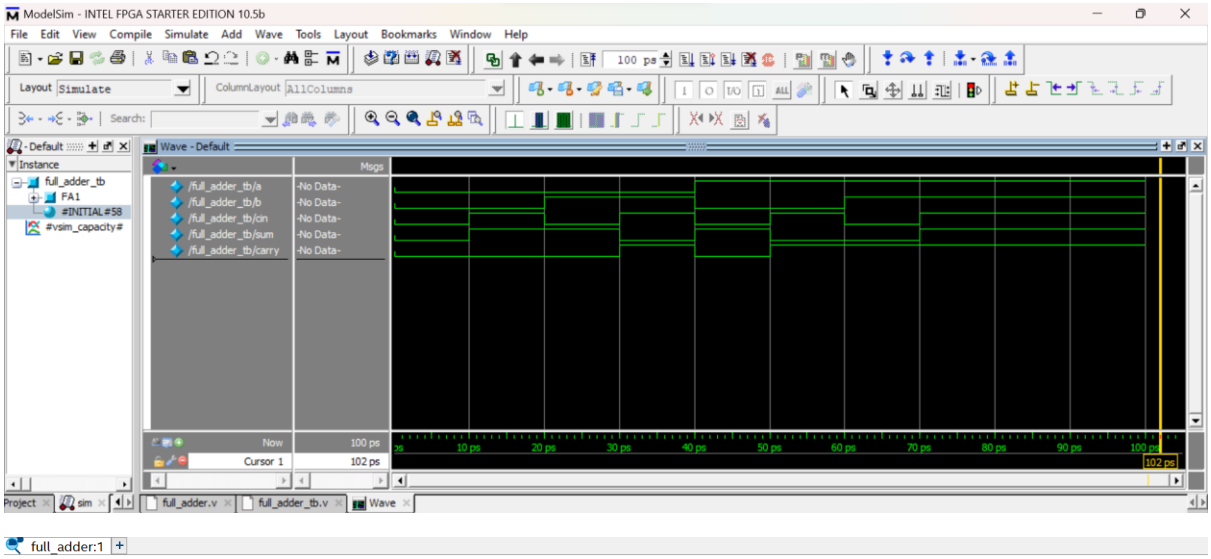
```
module full_adder(a,
                  b,
                  c,
                  sum,
                  carry);
  // Step 1. Write down the directions for the ports
input a,b,c;
output sum,carry;
  // Step 2. Declare the internal wires
wire w1,w2,w3;
  // Step 3. Instantiate two Half-Adders
half_adder HA1 (.a(a),.b(b),.sum(w1),.carry(w2));
half_adder HA2 (.a(w1),.b(c),.sum(sum),.carry(w3));
  // Step 4. Instantiate the OR gate
or or1 (carry,w2,w3);
endmodule
```

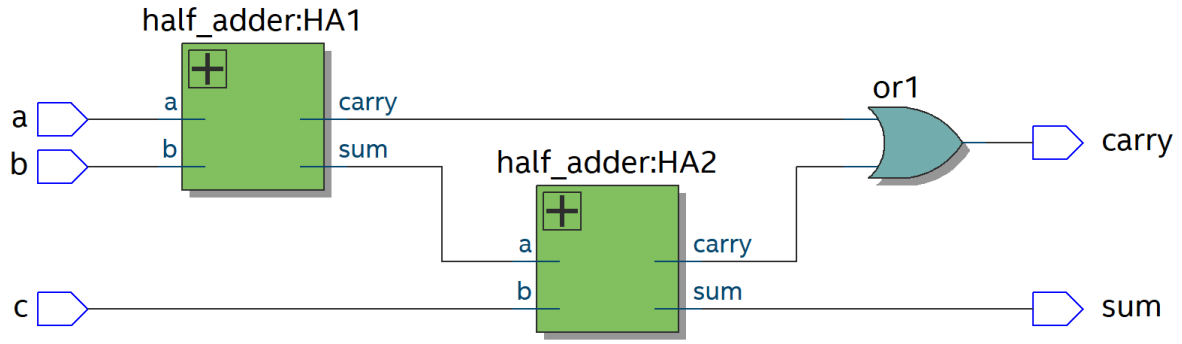
TESTBENCH:

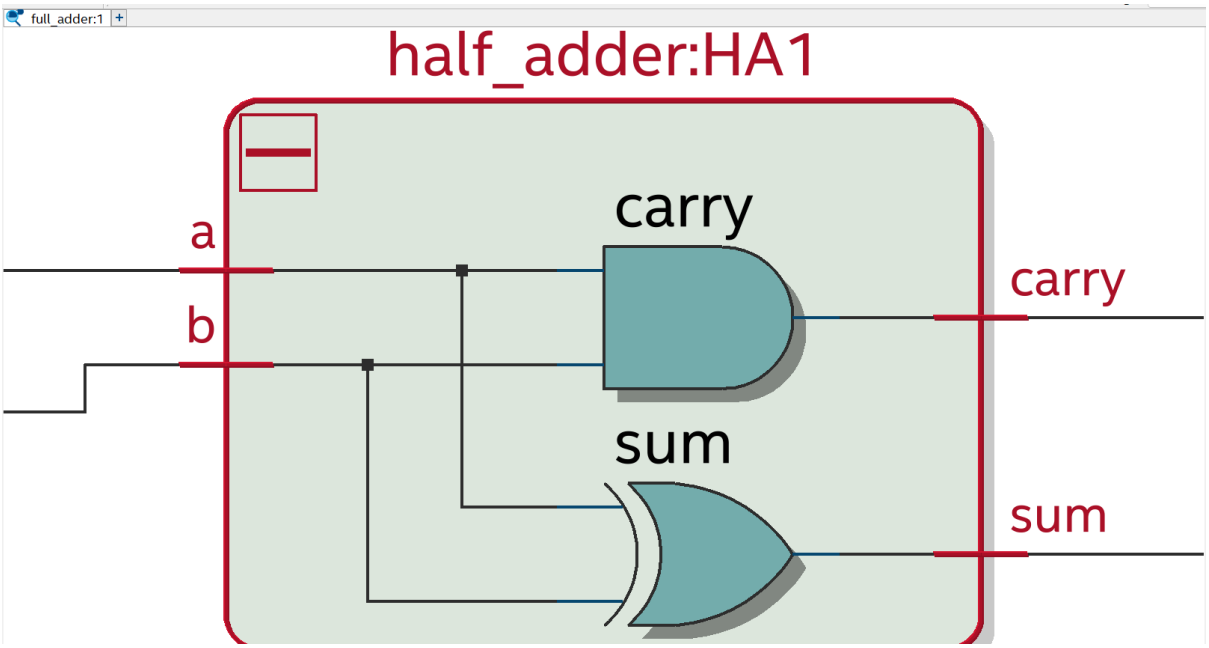
```
module full_adder_tb();
  reg   a,b,cin;
  wire  sum,carry;
  integer i;
  //Instantiate the full adder
full_adder FA1 (.a(a),.b(b),.c(cin),.sum(sum),.carry(carry));
  initial
  begin
    a   = 1'b0;
    b   = 1'b0;
    cin = 1'b0;
  end
  initial
```

```
  begin
    for (i=0;i<8;i=i+1)
    begin
      {a,b,cin}=i;
      #10;
    end
  end
  initial $monitor("Input a=%b, b=%b, c=%b, Output sum =%b,
carry=%b",a,b,cin,sum,carry);
  initial #100 $finish;
endmodule
```

OUTPUT:







LAB 2:

Arithmetic logic unit executing 16 operations:

RTL CODE:

```
module alu(a,
           b,
           command,
           oe,
           y);
  parameter ADD  = 4'b0000, // Add two 4 bit numbers a and b.
            INC  = 4'b0001, // Increment a by 1.
            SUB  = 4'b0010, // Subtracts b from a.
            DEC  = 4'b0011, // Decrement a by 1.
            MUL  = 4'b0100, // Multiply 4 bit numbers a and b.
            DIV  = 4'b0101, // Divide a by b.
            SHL  = 4'b0110, // Shift a to left side by 1 bit.
            SHR  = 4'b0111, // Shift a to right by 1 bit.
            AND  = 4'b1000, // Logical AND
            OR   = 4'b1001, // Logical OR
            INV  = 4'b1010, // Compement
            NAND = 4'b1011, // Logical NAND
```

```
            NOR  = 4'b1100, // Logical NOR
            XOR  = 4'b1101, // Logical EXOR
            XNOR = 4'b1110, // Logical EXNOR
            BUF  = 4'b1111; // Buffer
  input [7:0] a,
              b;
  input [3:0] command;
  input oe;
  output [15:0] y;
  reg    [15:0] out;
// Step 1. Write down the functionality of ALU based on commands
given above.
//         *Use arithmetic and logical operators
    always@(*)
    begin
       case (command)
      //--------- write the functionality here -------
      ADD  :     out  =a+b;
      INC  :     out  =a+1;
      SUB  :     out  =a-b;
      DEC  :     out  =a-1;
      MUL  :     out  =a*b;
      DIV  :     out  =a/b;
      SHL  :     out  =a<<b;
      SHR  :     out  =a>>b;
      AND  :     out  =a&b;
      OR   :     out  =a|b;
      INV  :     out  =~a;
      NAND :     out  =~(a&b);
      NOR  :     out  =~(a|b);
      XOR  :     out  =a^b;
      XNOR :     out  =~(a^b);
      BUF  :     out  =a;
```

```
        default :       out     =16'hxxxx;
       endcase
    end
// Step 2. Understand the tri-state output logic
  assign y = (oe) ? out : 16'hzzzz;
endmodule
```

TESTBENCH:

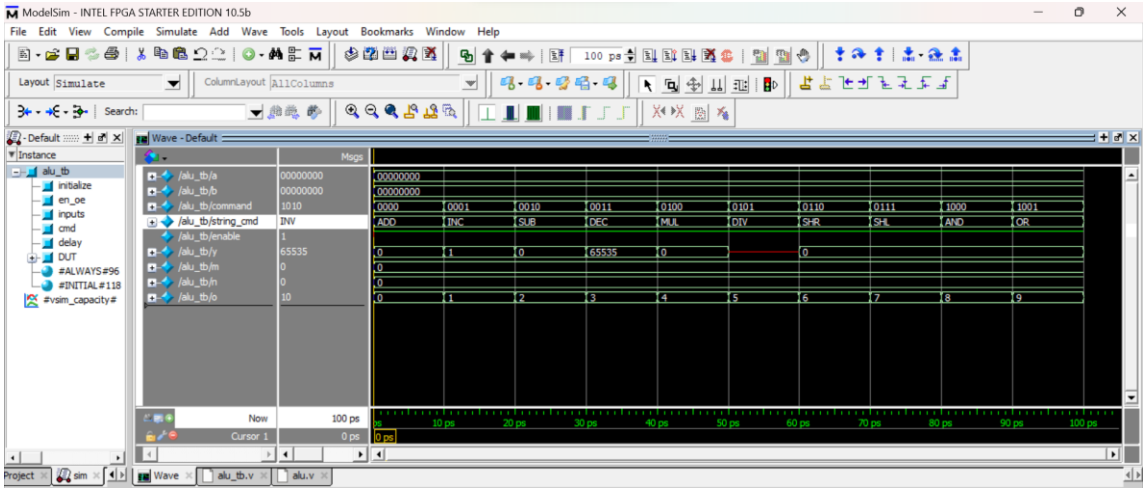
```
module alu_tb();
  reg [7:0]  a,
             b;
  reg [3:0]  command;
  reg enable;
  wire [15:0] y;
  integer m,n,o;
  parameter ADD  = 4'b0000, // 0  Add two 4 bit numbers a and b.
            INC  = 4'b0001, // 1  Increment a by 1.
            SUB  = 4'b0010, // 2  Subtracts b from a.
            DEC  = 4'b0011, // 3  Decrement a by 1.
            MUL  = 4'b0100, // 4  Multiply 4 bit numbers a and b.
            DIV  = 4'b0101, // 5  Divide a by b.
            SHR  = 4'b0110, // 6  Shift a to left side by 1 bit.
            SHL  = 4'b0111, // 7  Shift a to right by 1 bit.
            AND  = 4'b1000, // 8  Logical AND
            OR   = 4'b1001, // 9  Logical OR
            INV  = 4'b1010, // 10 Compement
            NAND = 4'b1011, // 11 Logical NAND
            NOR  = 4'b1100, // 12 Logical NOR
            XOR  = 4'b1101, // 13 Logical EXOR
            XNOR = 4'b1110, // 14 Logical EXNOR
            BUF  = 4'b1111; // 15 Buffer
  reg [(4*8)-1:0] string_cmd;
// Step 1. Instantiate the design ALU
  alu DUT(a,b,command,enable,y);
```

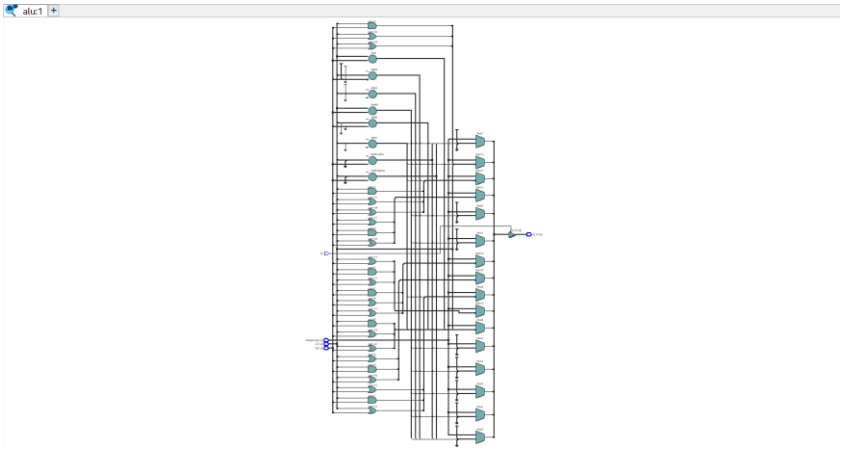
```
// Step 2. Write a task named "initialize" to initialize the inputs
of DUT
  task initialize;
  begin
   {a,b,command,enable}=0;
  end
  endtask
// Step 3. Understand the complete test bench and various tasks
defined below.
  task en_oe(input i);
  begin
    enable=i;
  end
  endtask
  task inputs(input [7:0] j,k);
  begin
    a=j;
    b=k;
  end
  endtask
  task cmd (input [3:0] l);
  begin
    command=l;
  end
  endtask
  task delay();
  begin
    #10;
  end
  endtask
  always@(command)
  begin
    case (command)
```

```
    ADD   :  string_cmd = "ADD";
    INC   :  string_cmd = "INC";
    SUB   :  string_cmd = "SUB";
    DEC   :  string_cmd = "DEC";
    MUL   :  string_cmd = "MUL";
    DIV   :  string_cmd = "DIV";
    SHR   :  string_cmd = "SHR";
    SHL   :  string_cmd = "SHL";
    AND   :  string_cmd = "AND";
    OR    :  string_cmd = "OR";
    INV   :  string_cmd = "INV";
    NAND  :  string_cmd = "NAND";
    NOR   :  string_cmd = "NOR";
    XOR   :  string_cmd = "XOR";
    XNOR  :  string_cmd = "XNOR";
    BUF   :  string_cmd = "BUF";
  endcase
end
initial
begin
  initialize;
  en_oe(1'b1);
  for (m=0;m<16;m=m+1)
  begin
    for (n=0;n<16;n=n+1)
    begin
      inputs(m,n);
      for (o=0;o<16;o=o+1)
      begin
        command=o;
        delay;
      end
```

```
      end
    end
    en_oe(0);
    inputs(8'd20,8'd10);
    cmd(ADD);
    delay;
    en_oe(1);
    inputs(8'd25,8'd17);
    cmd(ADD);
    delay;
    $finish;
  end
  initial $monitor("Input oe=%b, a=%b, b=%b, command=%s, Output y=%b",enable,a,b,string_cmd,y);
endmodule
```

OUTPUT:





LAB 3:

4:1 MUX Design

RTL CODE:

```
module mux4_1(a,
              sel,
              y);
// Step 1 : Define the port directions with proper datatypes and
ranges.
  input [3:0] a;
  input [1:0] sel;
  output reg y;
// Step 2 : Write the MUX behaviour as a parallel logic using case
statement in behaviour modelling.
  always@(*)
  begin
    case(sel)
      2'd0:y=a[0];
      2'd1:y=a[1];
      2'd2:y=a[2];
      2'd3:y=a[3];
    endcase
  end
endmodule
```

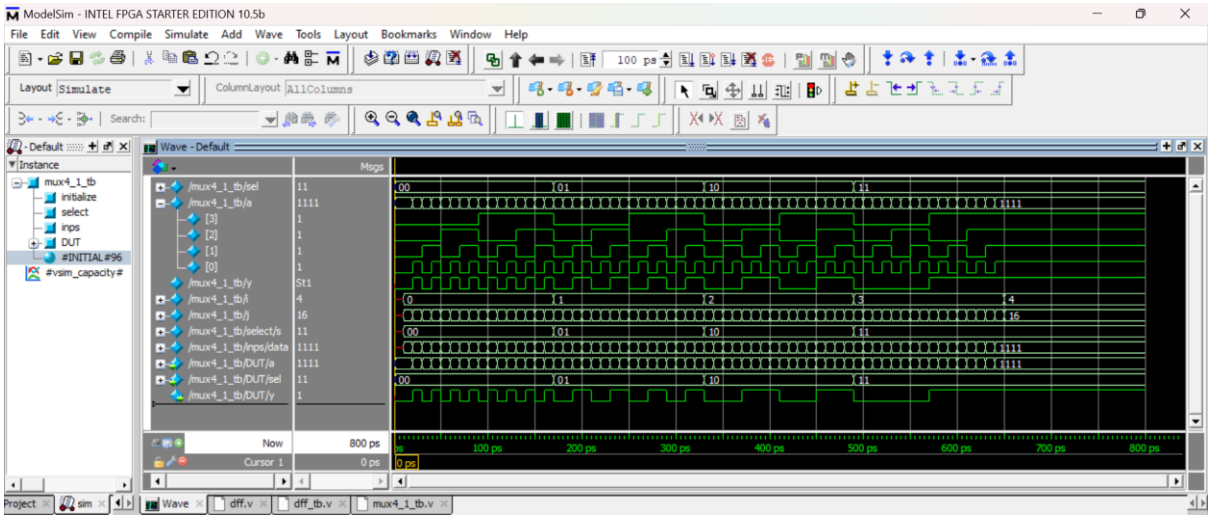
TESTBENCH:

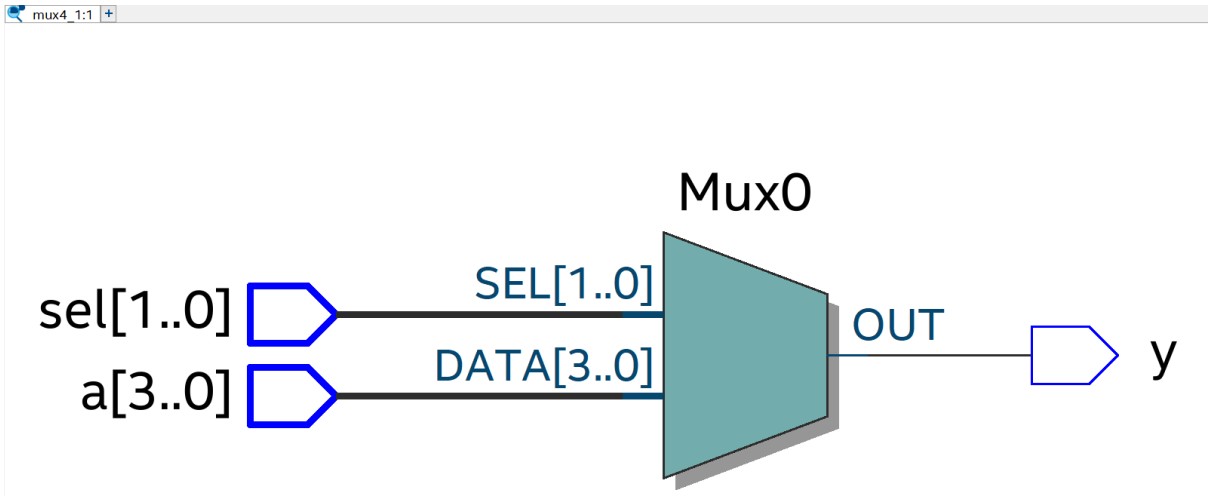
```
module mux4_1_tb();
// Declaration of the variables required for testbench
  reg [1:0]sel;
  reg [3:0]a;
  wire y ;
// Declaration of internal variables required for testbench
  integer i,j;
// Step 1. Instantiate the Design
```

```
  mux4_1 DUT(.a(a),.sel(sel),.y(y));
// Step 2. Define the body for the initialize task to initialize the
variables of DUT to 0
  task initialize;
  begin
    a=0;
    sel=0;
  end
  endtask
// Step 3. Declare  tasks with arguments for driving stimulus to DUT
  task select(input [1:0]s);
  begin
   sel=s;
  end
  endtask
  task inps(input [3:0] data);
  begin
    a=data;
  end
  endtask
// Step 4. Call the tasks from procedural block
  initial
  begin
    initialize;
    #10;
    for(i=0;i<4;i=i+1)
    begin
      select(i);
      for(j=0;j<16;j=j+1)
      begin
     inps(j);
     #10;
      end
```

```
    end
  end
// Step 5. Use $monitor task in a parallel initial block to display
inputs and outputs
  initial
    $monitor("a[3:0]=%b--select=%b--y=%d\n",a,sel,y);
// Step 6. Use $finish task to finish the simulation in a parallel
initial
// block with appropriate delay
  initial
    #800 $finish;
endmodule
```

OUTPUT:





LAB 4:

Delay Flipflop

RTL CODE:

```
module d_ff(clk,
            reset,
            d,
            q,
            qb);
// Step1 : Declare Port Directions
  input clk,reset,d;
  output reg q;
  output qb;
// Step2 : Write the behavioral logic for D flip-flop functionality.
  always@(posedge clk)
  begin
    if(reset)
      q=0;
    else
      q=d;
  end
//Step3. Assign complement of q to qb.
  assign qb=~q;
endmodule
```

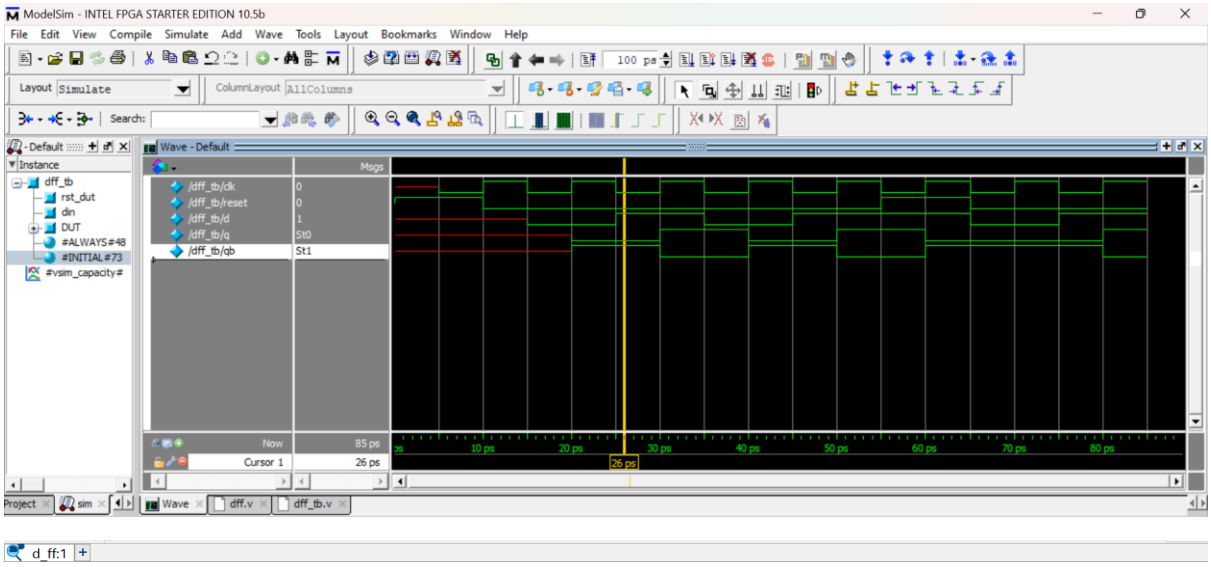
TESTBENCH:

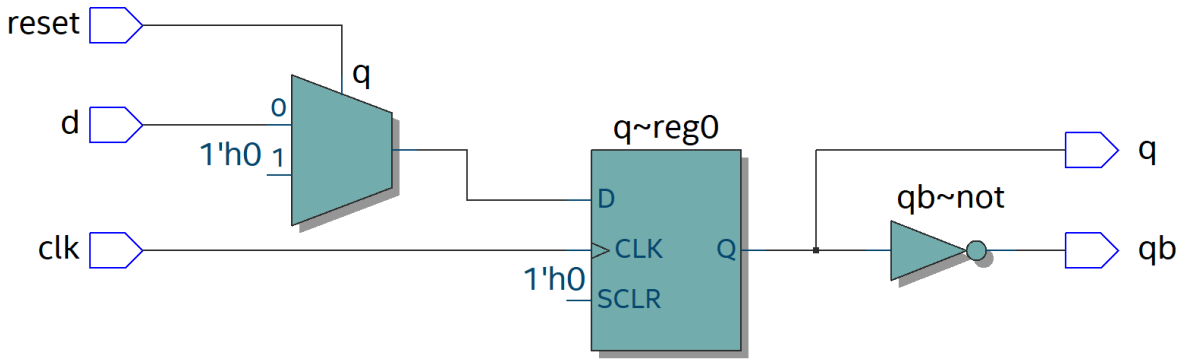
```
module dff_tb();
// Step 1. Define a parameter with name "cycle" which is equal to 10
  parameter cycle=10;
  reg clk,
      reset,
      d;
  wire q,
       qb;
```

```
// Step 2. Instantiate the dff design
  d_ff DUT (.clk(clk),.reset(reset),.d(d),.q(q),.qb(qb));
// Step 3. Understand the clock generation logic
  always
  begin
    #(cycle/2);
      clk = 1'b0;
    #(cycle/2);
      clk=~clk;
  end
//Step4. Understand the various tasks used and also how to use tasks
in testbench.
  task rst_dut();
  begin
    reset=1'b1;
    #10;
    reset=1'b0;
  end
  endtask
  task din(input i);
  begin
    @(negedge clk);
    d=i;
  end
  endtask
  initial
  begin
    rst_dut;
    din(0);
    din(1);
    din(0);
    din(1);
    din(1);
```

```
    rst_dut;
    din(0);
    din(1);
    #10;
    $finish;
  end
// Step 5. Use $monitor task in a parallel initial block to display
inputs and outputs
  initial
    $monitor("d=%d--q=%d--qb=%d",d,q,qb);
endmodule
```

OUTPUT:





LAB 5:

Asynchronous Single port Random access memory (16x8)

RTL CODE:

```
module ram(data,
           we,
           enable,
           addr);
  input we,enable;
  input [3:0] addr;
  inout [7:0] data;
// Step 1. Declare a 8 bit wide memory having 16 locations.
  reg [7:0]mem[15:0];
// Step 2. Understand the logic for writing data into a memory
location
  always@(data,we,enable,addr)
  if (we && !enable)
      mem[addr]=data;
// Step 3. Understand the logic of reading data from a memory
location
  assign data= (enable && !we) ? mem[addr] : 8'hzz;
endmodule
```

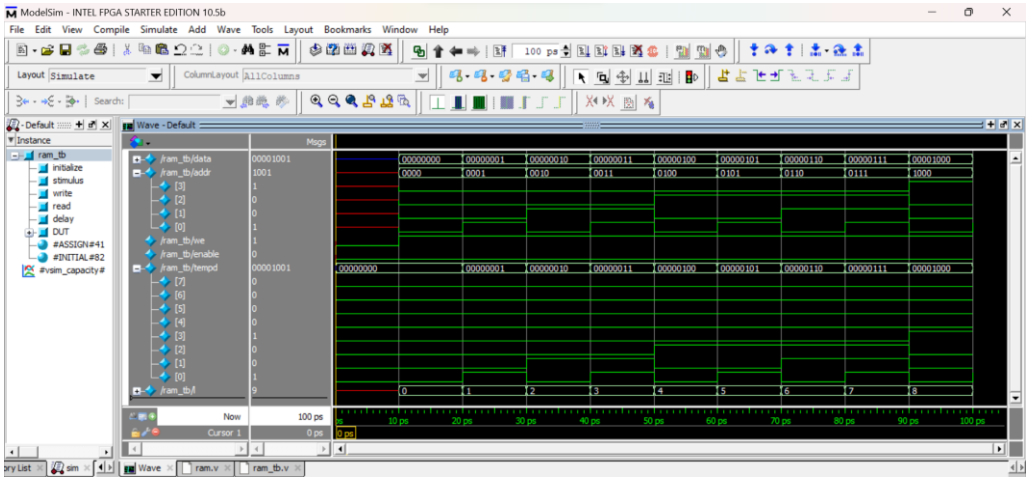
TESTBENCH:

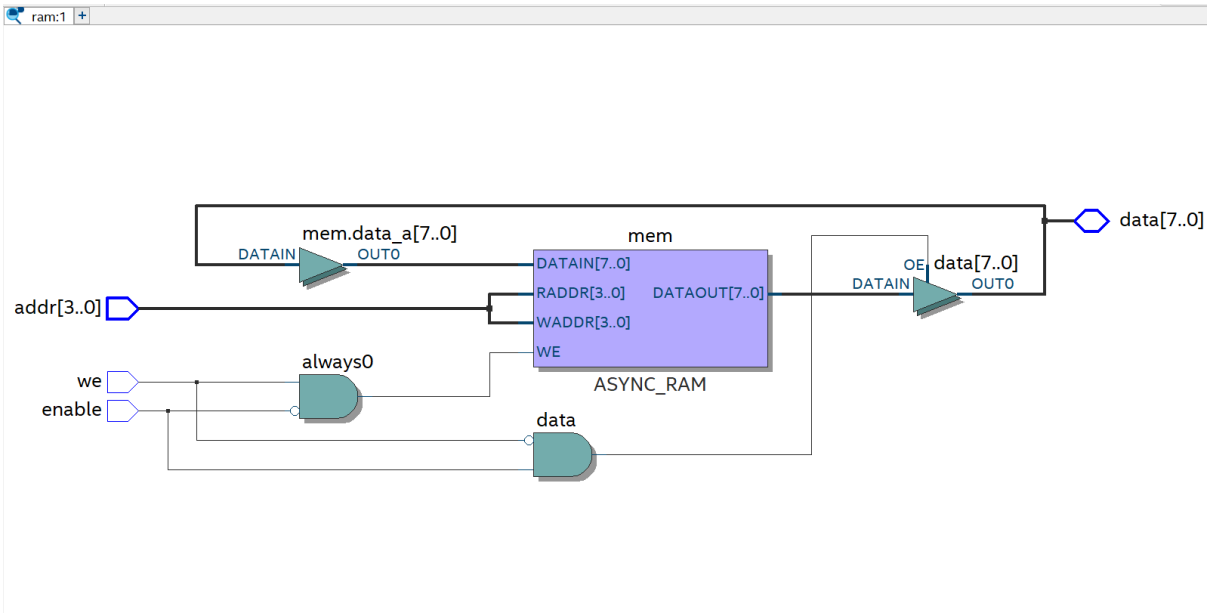
```
module ram_tb;
  wire [7:0] data;
  reg  [3:0] addr;
  reg  we,enable;
  reg  [7:0] tempd;
  integer l;
// Step 1. Instantiate the RAM module and connect the ports
  ram DUT(data,we,enable,addr);
  assign data=(we && !enable) ? tempd : 8'hzz;
  task initialize();
```

```
  begin
    we=1'b0; enable=1'b0; tempd=8'h00;
  end
  endtask
  // Step 2. define body of the task named "stimulus" to initialize
the
  //         "addr" and "tempd" inputs through i and j variables.
  //         use i initialization for "addr" and j initialization
for "tempd".
  task stimulus(input [3:0]i,input [7:0]j);
  begin
  //---------- define the body of the task here-----------//
    addr=i;
    tempd=j;
  end
   endtask
  // Step 3. Understand the various tasks defined in this testbench
  task write();
  begin
    we=1'b1;
    enable=1'b0;
  end
  endtask
  task read();
  begin
    we=1'b0;
    enable=1'b1;
  end
  endtask
  task delay;
  begin
    #10;
  end
```

```
  endtask
  initial
  begin
    initialize;
    delay;
    write;
    for(l=0;l<16;l=l+1)
    begin
      stimulus(l,l);
      delay;
    end
    initialize;
    delay;
    read;
    for(l=0;l<16;l=l+1)
    begin
      stimulus(l,l);
      delay;
    end
    delay;
    $finish;
  end
endmodule
```

OUTPUT:





LAB 6:

Sequence detector detecting "101"

RTL CODE:

```
module seq_det(din,
               clock,
               reset,
               dout);
// Step 1. Declare the sates as parameter
"IDLE","STATE1","STATE2","STATE3",
//         and use binary encoding for encoding these states.
  parameter IDLE = 2'b00,
            STATE1 = 2'b01,
            STATE2 = 2'b10,
            STATE3 = 2'b11;
// Step 2. Write down the port direction with proper directions.
  input din,clock,reset;
  output dout;
  reg [1:0] present_state,next_state;
// Step 3. Write down the sequential logic for present state
  always@(posedge clock)
  begin
```

```
    if(reset)
      present_state <= IDLE;
    else
     present_state <= next_state;
  end
// Step 4. Understand the combinational logic for next state
  always@(present_state,din)
  begin
    case (present_state)
      IDLE   : if (din==1)
                 next_state=STATE1;
               else
                 next_state=IDLE;
      STATE1 : if (din==0)
                  next_state=STATE2;
               else
                  next_state=STATE1;
      STATE2 : if (din==1)
                   next_state=STATE3;
               else
                   next_state=IDLE;
      STATE3 : if (din==1)
                   next_state=STATE1;
               else
                   next_state=STATE2;
      default : next_state=IDLE;
    endcase
  end
// Step 5. Write down the logic for output "dout".
  assign dout = (present_state == STATE3);
endmodule
```

TESTBENCH:

```
module seq_det_tb();
  parameter cycle=10;
  reg  din,
       clock,
       reset;
  wire dout;
  seq_det SQD(.din(din),
              .clock(clock),
              .reset(reset),
              .dout(dout));
// Step 1. Generate clock, using parameter "cycle"
  initial
  begin
    clock=0;
    forever #(cycle/2) clock=~clock;
  end
// Step 2. Write a task named "initialize" to initialize
//         the input din of sequence detector.
  task initialize;
  begin
    din = 0;
  end
  endtask
  task delay(input integer i);
  begin
    #i;
  end
  endtask
// Step 3. Write a task named "RESET" to reset the design,
//         use the above delay task for adding delay
  task RESET;
```

```
  begin
    reset=1;
    delay(10);
    reset=0;
  end
  endtask
// Step 4. Write a task named "stimulus" which provides input to
//         design on negedge of clock
  task stimulus(input j);
  begin
    @(negedge clock)
    din = j;
  end
  endtask
// Step 5 : understand the remaing logic defind below.
  initial $monitor("Reset=%b, state=%b, Din=%b, Output Dout=%b",
                   reset,SQD.present_state,din,dout);
  always@(SQD.present_state or dout)
  begin
     if (SQD.present_state==2'b11 && dout==1)
         $display("Correct output at state %b", SQD.present_state);
  end
  initial
  begin
    initialize;
    RESET;
    stimulus(0);
    stimulus(1);
    stimulus(0);
    stimulus(1);
    stimulus(0);
    stimulus(1);
```

```
    stimulus(1);
    RESET;
    stimulus(1);
    stimulus(0);
    stimulus(1);
    stimulus(1);
    delay(10);
    $finish;
  end
endmodule
```

OUTPUT: