KLE Technological University | Creating Value, Leveraging Knowledge

Dr. M. S. Sheshgiri Campus, Belagavi

Department of Electronics & Communication Engineering

Architectural Design of Integrated Circuits

23EECE302

Submitted for the partial fulfillment of the course

Submitted by:

Mr. Pratik P Patil

SRN:02FE21BEC063

Sem=6th, Div=A

KLE TU, Dr. M S Sheshgiri Campus, Belagavi.

Mentored by:

Dr. Kunjan D. Shinde

Assistant Professor, Dept. of E&CE,

KLE Technological University, Dr. M S Sheshgiri Campus, Belagavi.

Academic Year 2023-24

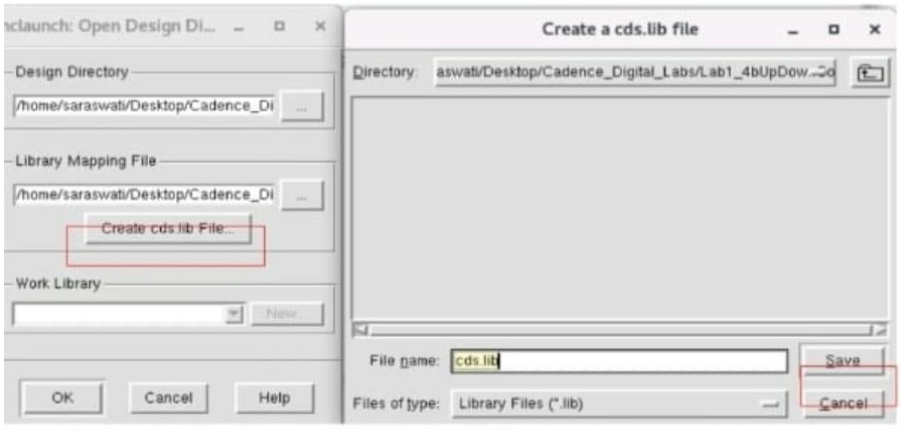
List of experiments

Demonstration	1	Introduction to Cadence Geneus, Questa Sim/ Model Sim/ Xilinx ISE/Cadence Geneus to simulate, synthesis and verify the design
Exercise	2	Design, Simulate and synthesize digital building blocks like Adder, Substractor, Mux,Demux
	3	Modeling data path and controller. Design and implement a a. Booth Multiplier b. GCD
	4	Design and Implement overlapping and non-overlapping Mealy and Moore state machine design.
	5	Design and implement Dual port RAM of 100Mb capacity Implementation of functions (FA) using ROM
	6	Design and implement A pipeline that carries out the following stage wise operation. Inputs: Three Register address (rs1,rs2,rd), ALU function, memory address addr. Stage 1: Read two 16 bit numbers from register specified and store them in A and B Stage 2: Perform ALU operation specified by func and store then in Z Stage 3: Write the value of Z in specified register rd. Stage 4: Also write the value of register in memory location addr
Open Ended	7	Design and implementation of 5 stage pipeline RISC V Processor.

Experiment: Demonstration	1	Introduction to Cadence Geneus, Questa Sim/ Model Sim/ Xilinx ISE/Cadence Geneus to simulate, synthesis and verify the design

Cadence Geneus (steps)

- In the Terminal, type gedit.v
- A Blank Document opens up into which the following source code can be typed
- Similarly, create your test bench using gedit.v
- Invoke the cadence environment by type the below commands
- csh
- source /home/install/cshrc
- To Launch Simulation tool ◦ linux:/> nclaunch -new&

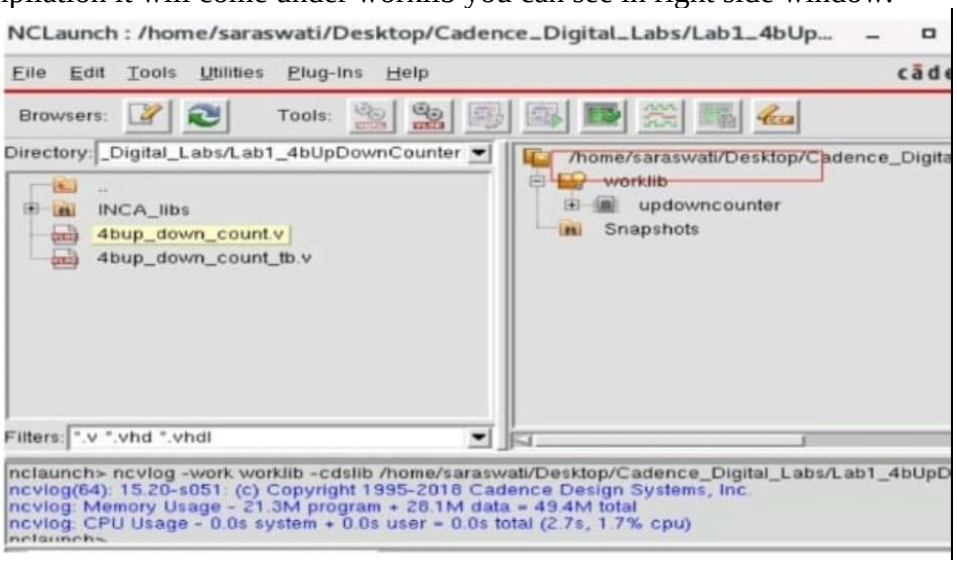


- Select Multiple Step and then select “Create cds.lib File” as shown in below figure
- Click the cds.lib file and save the file by clicking on Save option
- A ‘NCLaunch window’ appears.
- Left side you can see the HDL files. Right side of the window has worklib and snapshots directories listed.

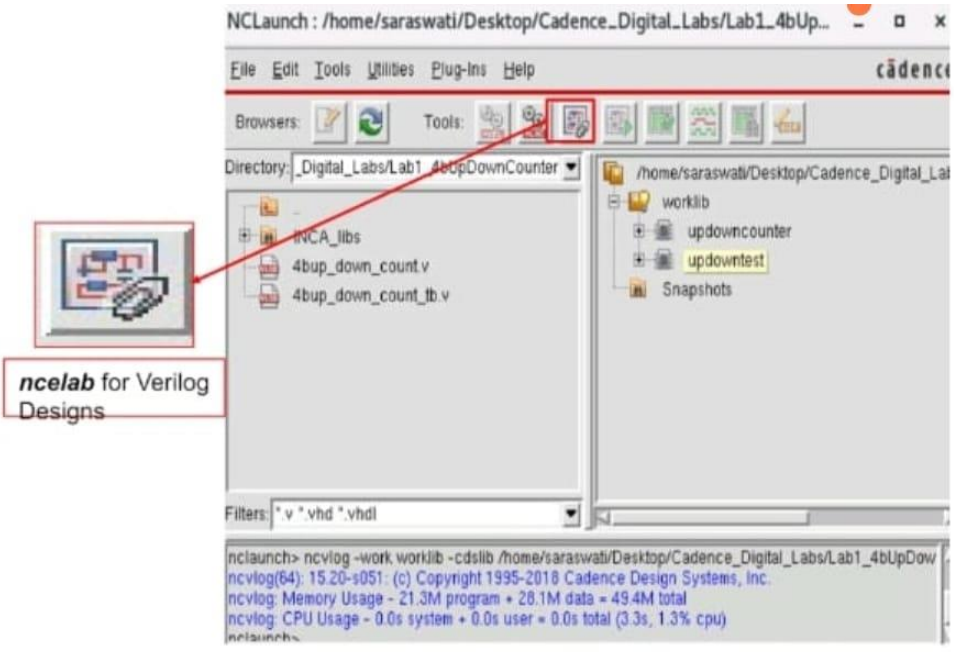
- Worklib is the directory where all the compiled codes are stored while Snapshot will have output of elaboration which in turn goes for simulation

Steps to compilation

- Left side select the file and in Tools : launch verilog compiler with current selection will get enable. Click it to compile the code.
- Worklib is the directory where all the compiled codes are stored while Snapshot will have output of elaboration which in turn goes for simulation.
- After compilation it will come under worklib you can see in right side window.



- Elaborate database updated in mapped library if successful, generates report else error reported in log file.



- Simulate with the given test vectors over a period of time to observe the output behaviour.

Xilinx ISE (steps)

Launch Xilinx ISE
Create New Project
Click on File > New Project... in the menu.
Project Name and Location:
Enter a Project Name and choose a Location to save your project.
Click Next.
Select **Project Type** (e.g., HDL, schematic).
Choose the **Top-Level Source Type** (e.g., HDL or schematic).
Click **Next.**
Add/**Create Source:**
Add existing sources or create new ones.
Save the File
Simulating Your Design
Create a New Simulation Source:
Right-click on **Simulation Sources** in the **Hierarchy pane.**
Select **New Source**... and choose **VHDL Test Bench** or **Verilog Test Fixture**.
Write your testbench code to test your design.
Run Simulation:
Click on **Behavioral Simulation** in the Processes pane.
Simulate your design to verify its correctness.
Run Synthesis
Configure FPGA

Experiment: Exercise	2	Design, Simulate and synthesize digital building blocks like Adder, Substractor, Mux,Demux

FULL ADDER

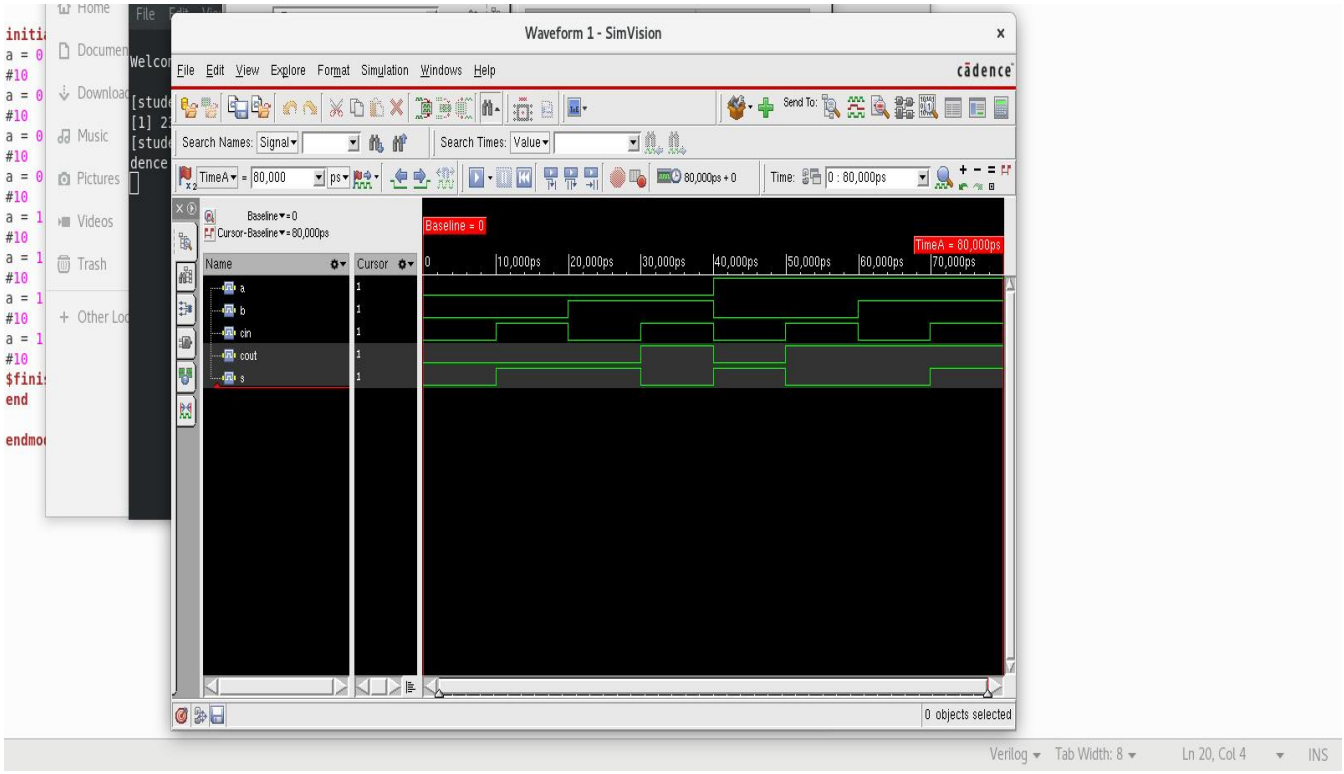
CODE:

```
`timescale 1ns/1ps
module full(a,b,cin,s,cout);
input a,b,cin;
output s,cout;

assign s = a^b^cin;
assign cout = (a & b ) | (b & cin ) | (cin & a ) ;

endmodule
```

TEST BENCH:

```
timescale 1ns/1ps
module full_test;
reg a,b,cin;
wire s,cout;

full uut(a,b,cin,s,cout);

initial begin
a = 0; b = 0; cin = 0;
#10
a = 0; b = 0; cin = 1;
#10
a = 0; b = 1; cin = 0;
#10
a = 0; b = 1; cin = 1;
#10
a = 1; b = 0; cin = 0;
#10
a = 1; b = 0; cin = 1;
#10
a = 1; b = 1; cin = 0;
#10
a = 1; b = 1; cin = 1;
#10
$finish();
end

endmodule
```

OUTPUT:



REFLECTION:

1. Understood how to add single bits (0 or 1) together, including the concept of a carry bit.

2. understood how full adders form the building block for more complex arithmetic circuits like multi-bit adders.

Subtractor

Code:

```
module subtractor_4bit (
    input wire [3:0] a,  // Minuend
    input wire [3:0] b,  // Subtrahend
    output wire [3:0] diff,  // Difference
    output wire borrow   // Borrow output
);
```

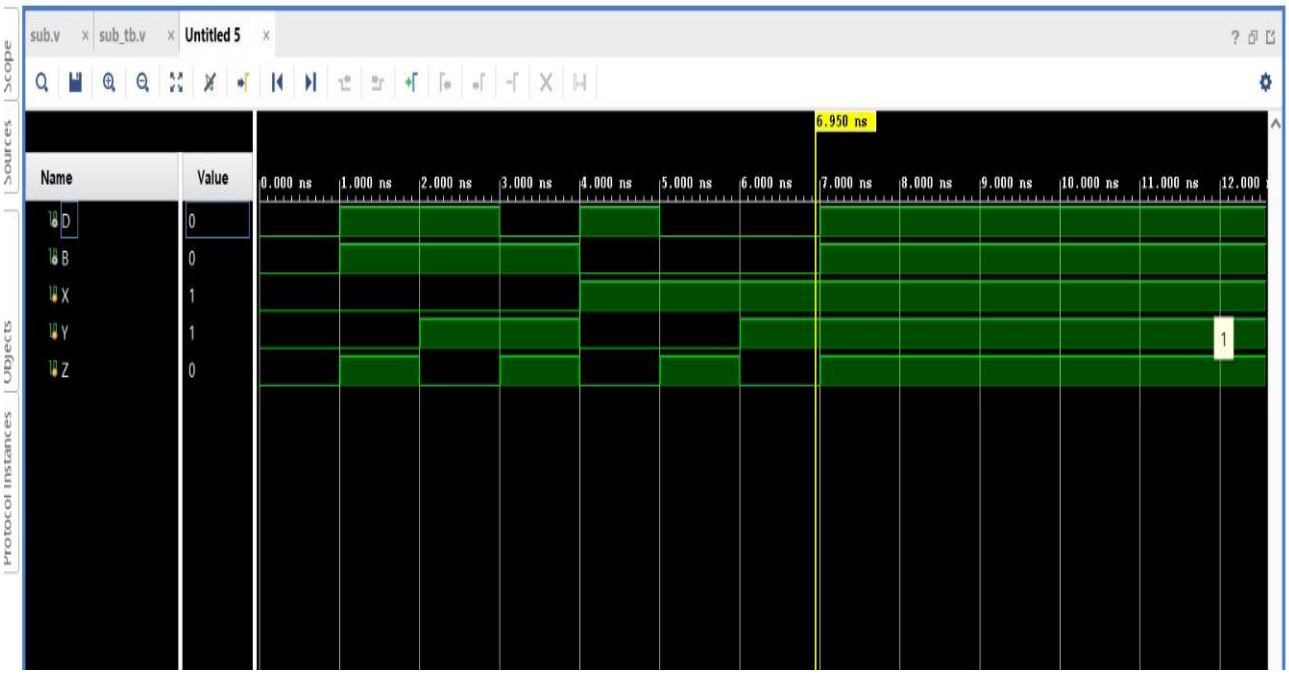
```
    wire [3:0] b_complement;  // Two's complement of b
    wire [4:0] temp_result;   // Temporary result to handle borrow

    // Calculate two's complement of b
    assign b_complement = ~b + 1;

    // Perform addition of a and two's complement of b
    assign temp_result = a + b_complement;

    // Assign the difference and the borrow
    assign diff = temp_result[3:0];
    assign borrow = temp_result[4];

endmodule
```

Testbench:

```
module tb_subtractor_4bit;

    reg [3:0] a;  // Minuend
    reg [3:0] b;  // Subtrahend
    wire [3:0] diff;  // Difference
    wire borrow;  // Borrow output

    // Instantiate the subtractor
    subtractor_4bit uut (
        .a(a),
        .b(b),
        .diff(diff),
        .borrow(borrow)
    );

    initial begin
        // Test case 1
        a = 4'b0110;  // 6
        b = 4'b0011;  // 3
        #10;
        $display("a = %b, b = %b, diff = %b, borrow = %b", a, b, diff, borrow);
```

```
        // Test case 2
        a = 4'b1001;  // 9
        b = 4'b1010;  // 10
        #10;
        $display("a = %b, b = %b, diff = %b, borrow = %b", a, b, diff, borrow);

        // Test case 3
        a = 4'b1111;  // 15
        b = 4'b0001;  // 1
        #10;
        $display("a = %b, b = %b, diff = %b, borrow = %b", a, b, diff, borrow);

        // Test case 4
        a = 4'b0000;  // 0
        b = 4'b0001;  // 1
        #10;
        $display("a = %b, b = %b, diff = %b, borrow = %b", a, b, diff, borrow);

        // End simulation
        $stop;
    end
endmodule
```

Output:



Reflection:

1. Learnt about vivado
2. understood the logic of substractor
3. implement a full subtractor, a combinational circuit that performs subtraction of two bits with borrow.

4_to_1_MUX

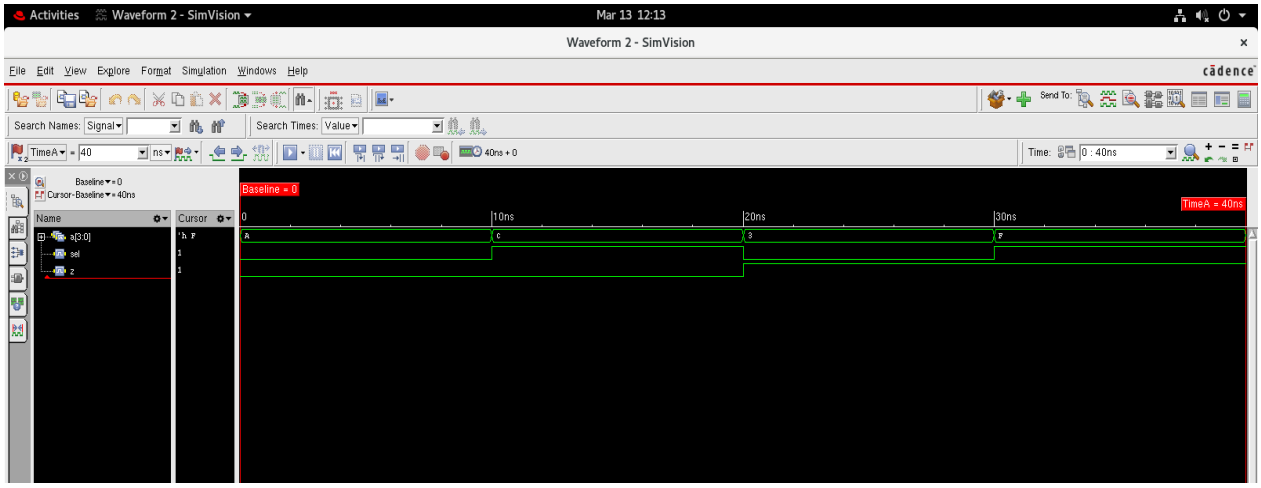
Code:

```
module multiplexer_4_to_1 ( output reg z, input [3:0] a, input sel );
 always @*
case (sel)
2'b00: z = a[0];
2'b01: z = a[1];
2'b10: z = a[2];
2'b11: z = a[3];
endcase
Endmodule
```

Test Bench:

```
`timescale 1ns/1ns
module multiplexer_4_to_1 (
  output reg z,
  input [3:0] a,
  input sel
);
  always @*
          case (sel)
          2'b00: z = a[0];
          2'b01: z = a[1];
          2'b10: z = a[2];
          2'b11: z = a[3];
          endcase
Endmodule
```

Output:



Reflection:

1.It is the best practice to explicitly specify whether a variable is a register or a wire..
2. The "reg" keyword is essential for declaring a variable as a register, signaling that it can hold values within procedural blocks such as "always" or "initial" blocks.

DEMUX

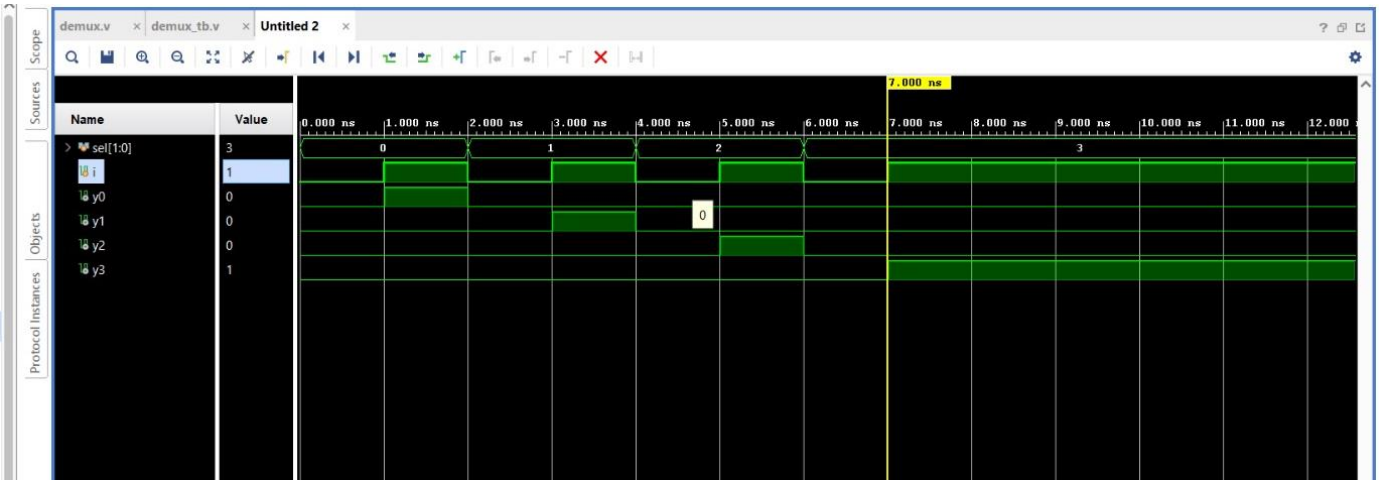
Code:

```
module demux_1_4(
  input [1:0] sel,
  input  i,
  output reg y0,y1,y2,y3);

  always @(*) begin
    case(sel)
      2'h0: {y0,y1,y2,y3} = {i,3'b0};
      2'h1: {y0,y1,y2,y3} = {1'b0,i,2'b0};
      2'h2: {y0,y1,y2,y3} = {2'b0,i,1'b0};
      2'h3: {y0,y1,y2,y3} = {3'b0,i};
      default: $display("Invalid sel input");
    endcase
  end
endmodule
```

Testbench

```
module tb;
  reg [1:0] sel;
  reg i;
  wire y0,y1,y2,y3;

  demux_1_4 demux(sel, i, y0, y1, y2, y3);

  initial begin
    $monitor("sel = %b, i = %b -> y0 = %0b, y1 = %0b ,y2 = %0b, y3 = %0b", sel,i, y0,y1,y2,y3);
    sel=2'b00; i=0; #1;
    sel=2'b00; i=1; #1;
    sel=2'b01; i=0; #1;
    sel=2'b01; i=1; #1;
    sel=2'b10; i=0; #1;
    sel=2'b10; i=1; #1;
    sel=2'b11; i=0; #1;
    sel=2'b11; i=1; #1;
  end
endmodule
```

Output



Reflection

1. Learnt about vivado
2. understood the logic of 1:4 demux
3. Simulate waveform and get output

Experiment: Exercise	3	Modeling data path and controller. Design and implement a a. Booth Multiplier b. GCD

Booth Multiplier

CODE:

```
module BoothMul(clk,rst,start,X,Y,valid,Z);

input clk;
input rst;
input start;
input signed [3:0]X,Y;
output signed [7:0]Z;
output valid;

reg signed [7:0] Z,next_Z,Z_temp;
reg next_state, pres_state;
reg [1:0] temp,next_temp;
reg [1:0] count,next_count;
reg valid, next_valid;

parameter IDLE = 1'b0;
parameter START = 1'b1;

always @ (posedge clk or negedge rst)
begin
if(!rst)
begin
  Z          <= 8'd0;
  valid      <= 1'b0;
  pres_state <= 1'b0;
  temp       <= 2'd0;
  count      <= 2'd0;
end
else
begin
  Z          <= next_Z;

  valid      <= next_valid;
  pres_state <= next_state;
  temp       <= next_temp;
  count      <= next_count;
end
always @ (*)
```

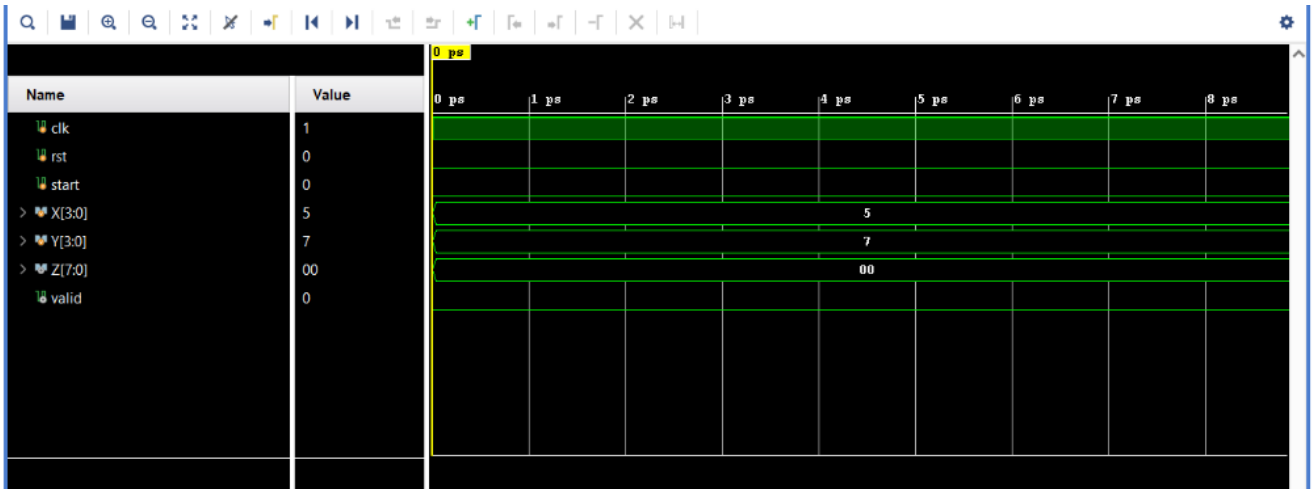
```
begin
case(pres_state)
IDLE:
begin
next_count = 2'b0;
next_valid = 1'b0;
if(start)
begin
   next_state = START;
   next_temp  = {X[0],1'b0};
   next_Z     = {4'd0,X};
end
else
begin
   next_state = pres_state;
   next_temp  = 2'd0;
   next_Z     = 8'd0;
end
end

START:
begin
   case(temp)
   2'b10:   Z_temp = {Z[7:4]-Y,Z[3:0]};
   2'b01:   Z_temp = {Z[7:4]+Y,Z[3:0]};
   default: Z_temp = {Z[7:4],Z[3:0]};
   endcase
next_temp  = {X[count+1],X[count]};
next_count = count + 1'b1;
next_Z     = Z_temp >>> 1;
next_valid = (&count) ? 1'b1 : 1'b0;
next_state = (&count) ? IDLE : pres_state;
end
endcase
end
endmodule
```

Test bench:

```
module booth_tb;

reg clk,rst,start;
reg signed [3:0]X,Y;
wire signed [7:0]Z;
wire valid;

always #5 clk = ~clk;

BoothMul inst (clk,rst,start,X,Y,valid,Z);

initial
$monitor($time,"X=%d, Y=%d, valid=%d, Z=%d ",X,Y,valid,Z);
```

```
initial
begin
X=5;Y=7;clk=1'b1;rst=1'b0;start=1'b0;
#10 rst = 1'b1;
#10 start = 1'b1;
#10 start = 1'b0;
@valid
#10 X=-4;Y=6;start = 1'b1;
#10 start = 1'b0;
end
endmodule
```

Output:



Reflection :

1. Booth Multipliers offer significant improvements in efficiency compared to traditional multiplication algorithms, particularly in hardware implementations.
2. Booth Multipliers handle signed multiplication efficiently by encoding signed numbers in a way that simplifies the multiplication process.
3. Booth Multipliers handle signed multiplication efficiently by encoding signed numbers in a way that simplifies the multiplication process

Greatest Common Divisor **(GCD)**

Code:

```
module GCD(out,done,clk,rst,in1,in2,go);
input [31:0]in1,in2;
input clk,rst,go;
output [31:0]out;
output done;
wire a_gt_b,a_lt_b,a_eq_b;
wire a_ld,b_ld,a_sel,b_sel;
wire output_en;
controller c1(a_ld,b_ld,a_sel,b_sel,output_en,done,clk,rst,go,a_gt_b,a_lt_b,a_eq_b);
datapath d1(a_gt_b,a_lt_b,a_eq_b,out,output_en,clk,rst,a_ld,b_ld,a_sel,b_sel,in1,in2);
 endmodule
```

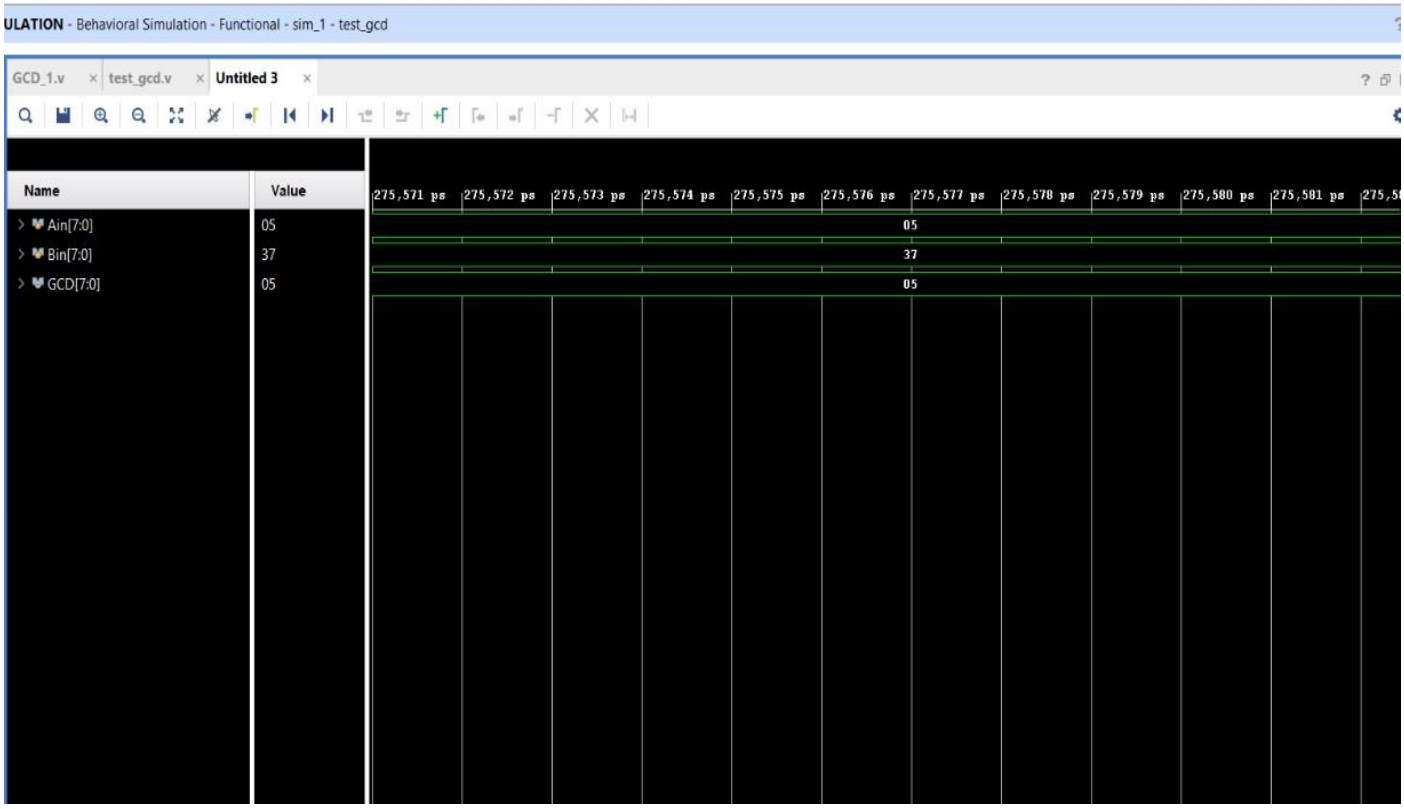
Test bench:

```
module testbench_GCD;
        reg clk;
         reg rst;
         reg [31:0] in1;
         reg [31:0] in2;
         reg go;
wire [31:0] out;
         wire done;
```

```
        //integer i;
        initial clk=0;
        always@(clk) #20 clk <= ~clk;
        GCD uut (
                .out(out),
                .done(done),
                .clk(clk),
                .rst(rst),
                .in1(in1),
                .in2(in2),
                .go(go));
task initialize;
        begin
          in1=0;
          in2=0;
          go=0;
        end
        endtask
        task reset;
          begin
            @(negedge clk)
                rst <=1;
                go<=0;
                @(negedge clk)
                rst <=0;
                go <=1;
```

```
            end endtask
        task calculate(input [31:0]a,input [31:0]b);
          begin
            @(negedge clk)
                go<=1;
                in1 <= a;
                    in2 <= b;
                    #50 go<=0;
                    #10000;
          end
          endtask
        initial begin
          $monitor($time," %d %d %d",in1,in2,out);
          reset;
          initialize;
          //calculate(48123,628163);
          //reset;
          //calculate(45,90);
          //reset;
          //calculate(2000,10000);
          //reset;
          calculate(17111,149495);
  end
endmodule
```

Output:



Reflection

1. Calculating the GCD involves finding the largest integer that divides both numbers without leaving a remainder.
2. GCD computations have practical applications in computer science, such as in cryptography, where it is used in key generation and other cryptographic protocols.

Experiment: Exercise	4	Design and Implement overlapping and non-overlapping Mealy and Moore state machine design.

Sequence detector

Code

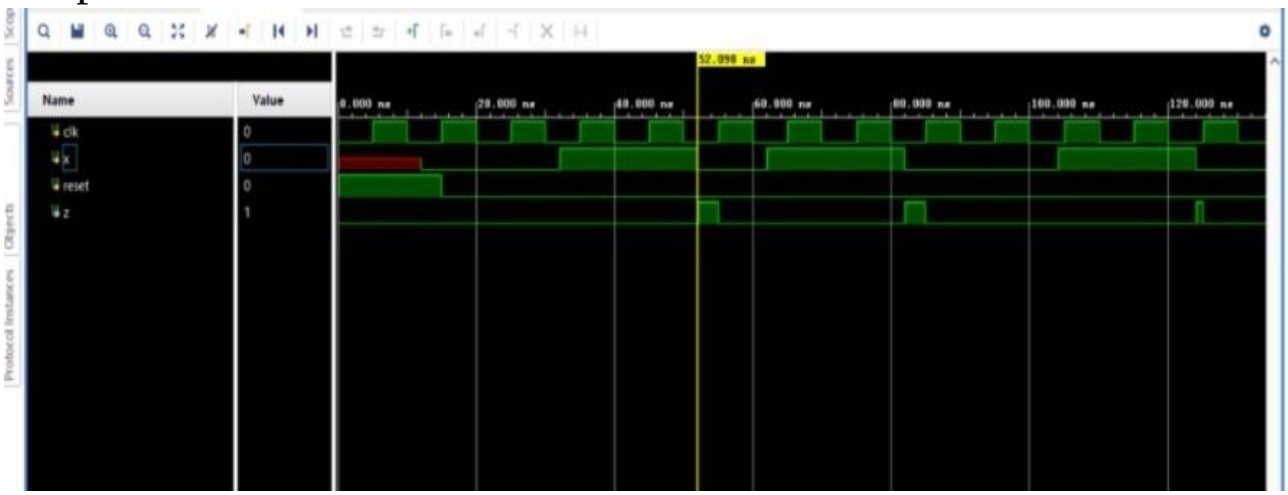
```
module detector(
  input x,
  input clk,
  input reset,
  output reg z
);
parameter s0=0,s1=1,s2=2,s3=3;
reg[0:1]Ps,Ns;
always@(posedge clk or posedge reset)
if(reset)
Ps<=s0;
else
Ps<=Ns;
always@(Ps,x)
case(Ps)
s0:begin
z= x? 0:0;
Ns= x? s0:s1;
end
s1:begin
z= x? 0:0;
Ns = x? s2:s1;
end
s2:begin
z= x ? 0: 0;
Ns = x? s3:s1;
end
s3:begin
z= x? 0:1;
Ns = x? s0:s1;
end
endcase
Endmodule
```

Test bench

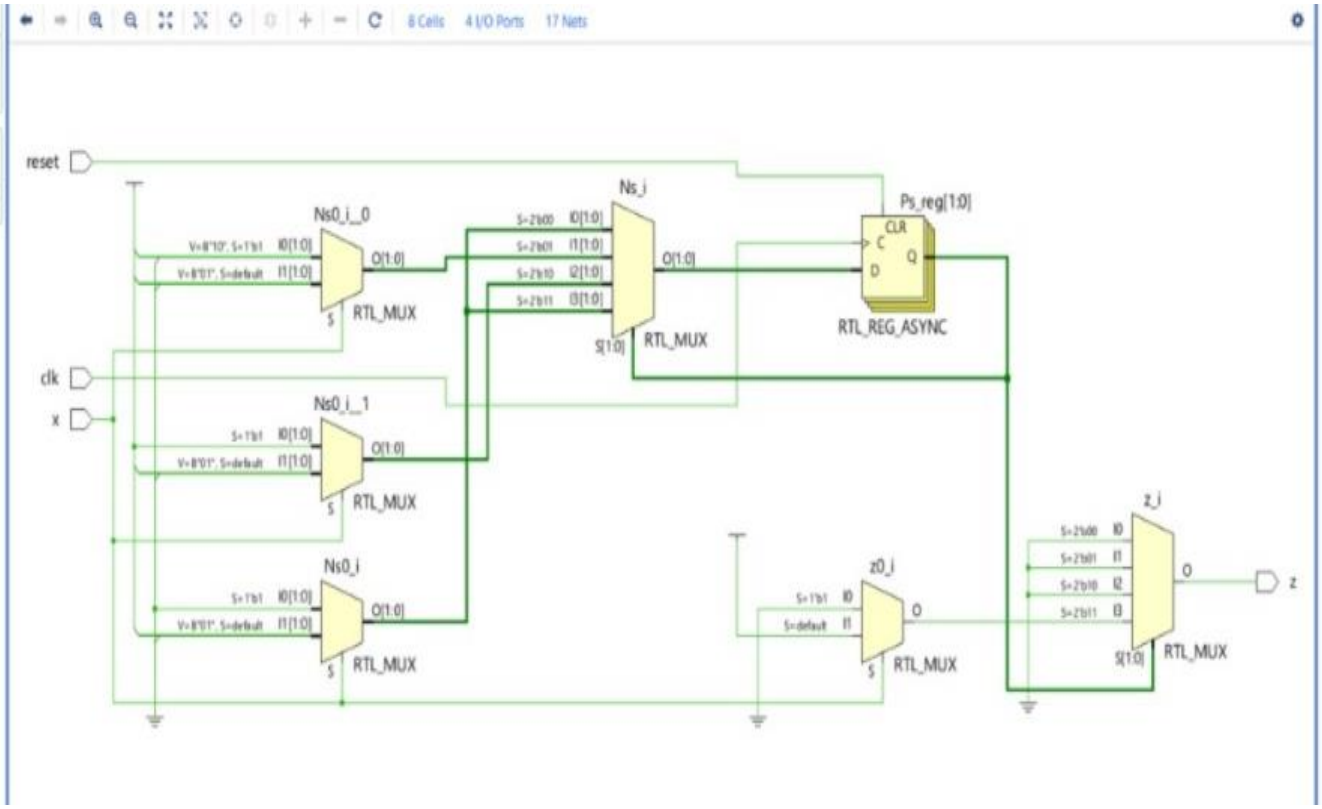
```
module seq_tb;
```

```
reg clk,x,reset;
wire z;
detector SEQ(x,clk,reset,z);
initial
begin
$dumpfile("sequence.vcd");
$dumpvars(0,seq_tb);
clk=1'b0;
reset =1'b1;
#15 reset = 1'b0;
end
always  #5 clk = ~clk;
initial
begin
#12 x=0;#10 x=0;#10 x=1; #10 x=1;
#10 x=0;#10 x=1;#10 x=1; #10 x=0;
#12 x=0;#10 x=1;#10 x=1; #10 x=0;
#10 $finish;
end
endmodule
```

Output



Schematic



Reflection

1. implementing FSMs for sequence detection. FSMs are a fundamental concept in digital design and are widely used in various applications

2. The sequence detector essentially performs edge detection by checking the current state and input to determine the next state and output. This principle is useful in many digital design scenarios, such as debouncing switches and serial data communication

Experiment: Exercise	5	Design and implement Dual port RAM of 100Mb capacity Implementation of functions (FA) using ROM

Design port RAM of 100MB capacity Implementation of functions(FA) using ROM.

Code

```
module dual_port_ram(
input [7:0] data_a, data_b, //input data


input [5:0] addr_a, addr_b, //Port A and Port B address
input we_a, we_b, //write enable for Port A and Port B
input clk, //clk
output reg [7:0] q_a, q_b //output data at Port A and Port B
);


reg [7:0] ram [63:0]; //8*64 bit ram


always @ (posedgeclk)
begin
if(we_a)
 ram[addr_a] <= data_a;
else
q_a<= ram[addr_a];
end


always @ (posedgeclk)
begin
 if(we_b)
 ram[addr_b] <= data_b; else
q_b<= ram[addr_b];
 end
endmodule
```

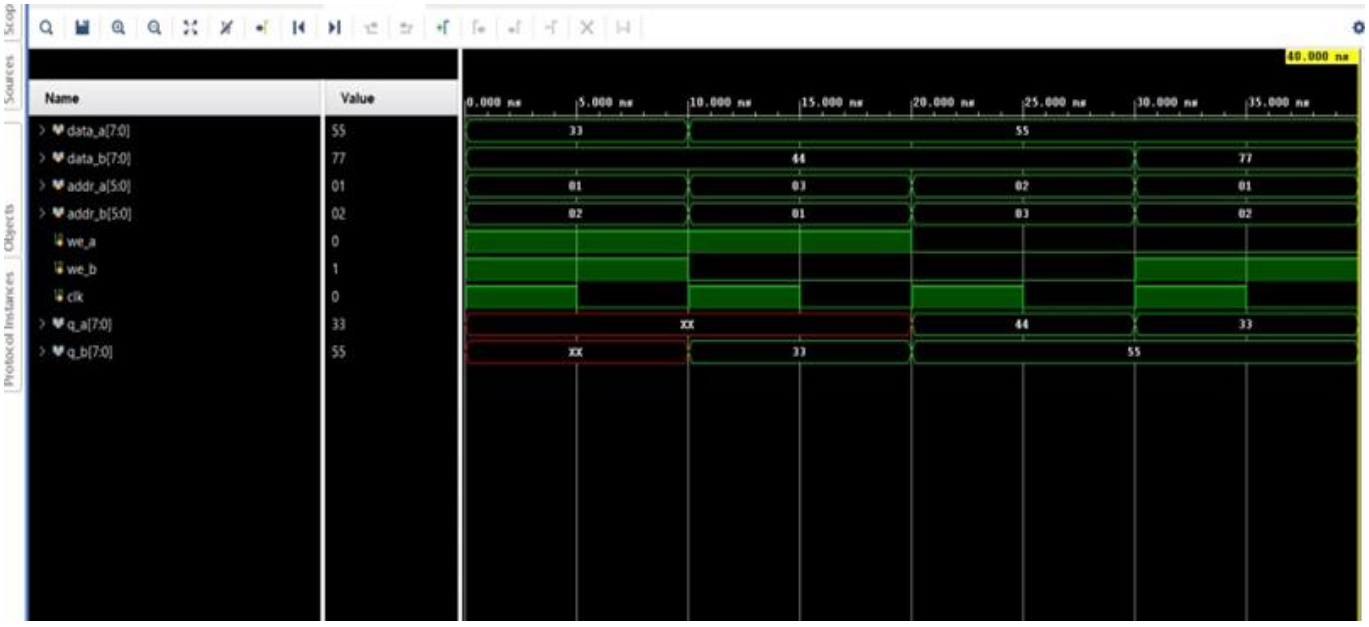
Testbench

```
module dual_port_ram_tb;
 reg [7:0] data_a, data_b; //input data
 reg [5:0] addr_a, addr_b; //Port A and Port B address
```

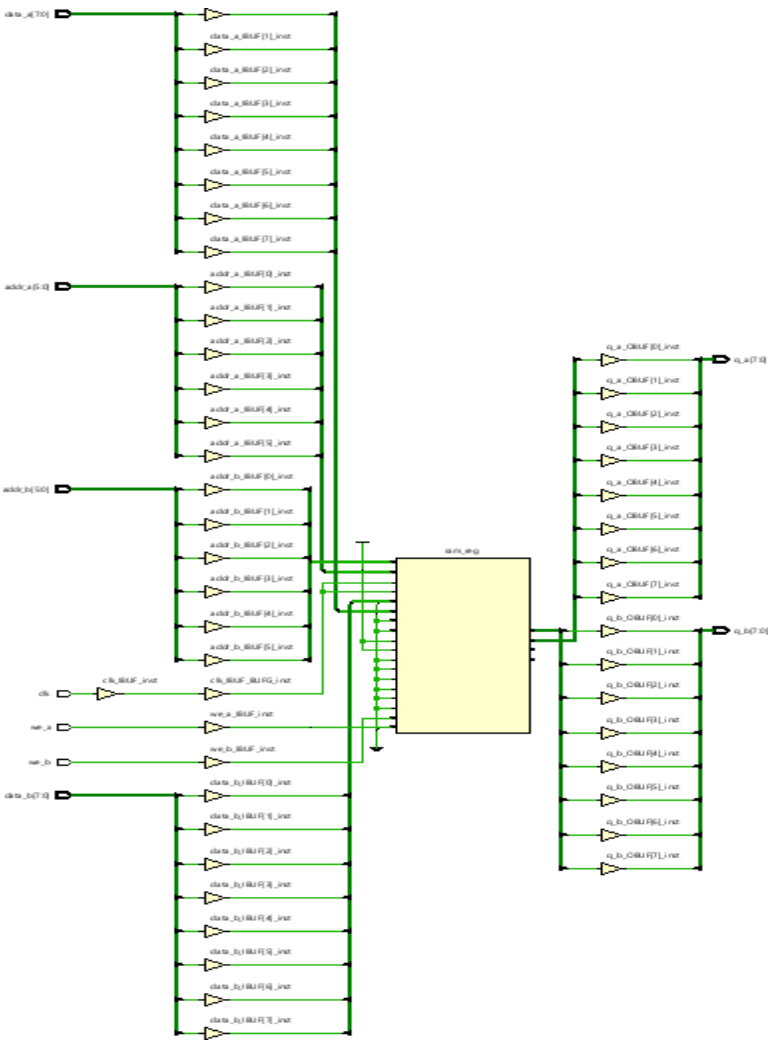
```
 reg we_a, we_b; //write enable for Port A and Port B
 reg clk; //clk
 wire [7:0] q_a, q_b; //output data at Port A and Port B

dual_port_ram dpr1(
 .data_a(data_a),
 .data_b(data_b),
.addr_a(addr_a),
.addr_b(addr_b),
.we_a(we_a),

.we_b(we_b),
.clk(clk),
.q_a(q_a),
.q_b(q_b)
);

initial begin
 $dumpfile("dump.vcd");
 $dumpvars(1, dual_port_ram_tb);

clk=1'b1;
forever #5 clk = ~clk;
end

initial begin
data_a = 8'h33;
addr_a = 6'h01;

data_b = 8'h44;
addr_b = 6'h02;

we_a = 1'b1;
we_b = 1'b1;

#10;

data_a = 8'h55;
addr_a = 6'h03;

addr_b = 6'h01;
we_b = 1'b0;
#10;

addr_a = 6'h02;
addr_b = 6'h03;
```

```
we_a = 1'b0;
#10;

addr_a = 6'h01;

data_b = 8'h77;
addr_b = 6'h02;
we_b = 1'b1; #10;
end

initial
#40 $stop;

Endmodule
```

Output



Schematic:



Reflection:

1. Dual-port RAM allows simultaneous read/write access to different addresses, which is critical for high-performance systems where multiple data operations need to be conducted concurrently.
2. ROM-Based Function Implementation.

Experiment: Exercise	6	Design and implement A pipeline that carries out the following stage wise operation. Inputs: Three Register address (rs1,rs2,rd), ALU function, memory address addr. Stage 1: Read two 16 bit numbers from register specified and store them in A and B Stage 2: Perform ALU operation specified by func and store then in Z Stage 3: Write the value of Z in specified register rd. Stage 4: Also write the value of register in memory location addr

Code

```
module ALU_16bit_3s_Pipelined(
   input clk,                // Clock signal
   input reset,              // Reset signal
   input [15:0] a,           // First operand
   input [15:0] b,           // Second operand
   input [2:0] sel,          // Operation selector
   output reg [15:0] y,      // Result of the operation
   output reg cout,          // Carry out for addition
   output reg zero           // Zero flag
);

// Pipeline registers
reg [15:0] a_reg1, b_reg1;
reg [2:0] sel_reg1;
reg [16:0] result_reg2;
reg [15:0] y_reg3;
reg cout_reg3, zero_reg3;

// Fetch/Decode Stage
always @(posedge clk or posedge reset) begin
   if (reset) begin
      a_reg1 <= 16'b0;
      b_reg1 <= 16'b0;
      sel_reg1 <= 3'b0;
   end else begin
      a_reg1 <= a;
      b_reg1 <= b;
      sel_reg1 <= sel;
   end
end

// Execute Stage
always @(posedge clk or posedge reset) begin
   if (reset) begin
      result_reg2 <= 17'b0;
   end else begin
      case (sel_reg1)
         3'b000: begin // Addition
            result_reg2 <= a_reg1 + b_reg1;
         end
         3'b001: begin // Subtraction
            result_reg2 <= a_reg1 - b_reg1;
```

```
        end
        3'b010: begin // AND
           result_reg2 <= {1'b0, a_reg1 & b_reg1};
        end
        3'b011: begin // OR
           result_reg2 <= {1'b0, a_reg1 | b_reg1};
        end
        3'b100: begin // XOR
           result_reg2 <= {1'b0, a_reg1 ^ b_reg1};
        end
        3'b101: begin // Left Shift
           result_reg2 <= {1'b0, a_reg1 << 2};
        end
        3'b110: begin // Right Shift
           result_reg2 <= {1'b0, a_reg1 >> 2};
        end
        default: begin // Default to zero
           result_reg2 <= 17'b0;
        end
      endcase
   end
end

// Write Back Stage
always @(posedge clk or posedge reset) begin
   if (reset) begin
      y_reg3 <= 16'b0;
      cout_reg3 <= 1'b0;
      zero_reg3 <= 1'b0;
   end else begin
      y_reg3 <= result_reg2[15:0];
      cout_reg3 <= result_reg2[16];
      zero_reg3 <= (result_reg2[15:0] == 16'b0);
   end
end

// Output assignments
always @(*) begin
   y = y_reg3;
   cout = cout_reg3;
   zero = zero_reg3;
end

endmodule
```

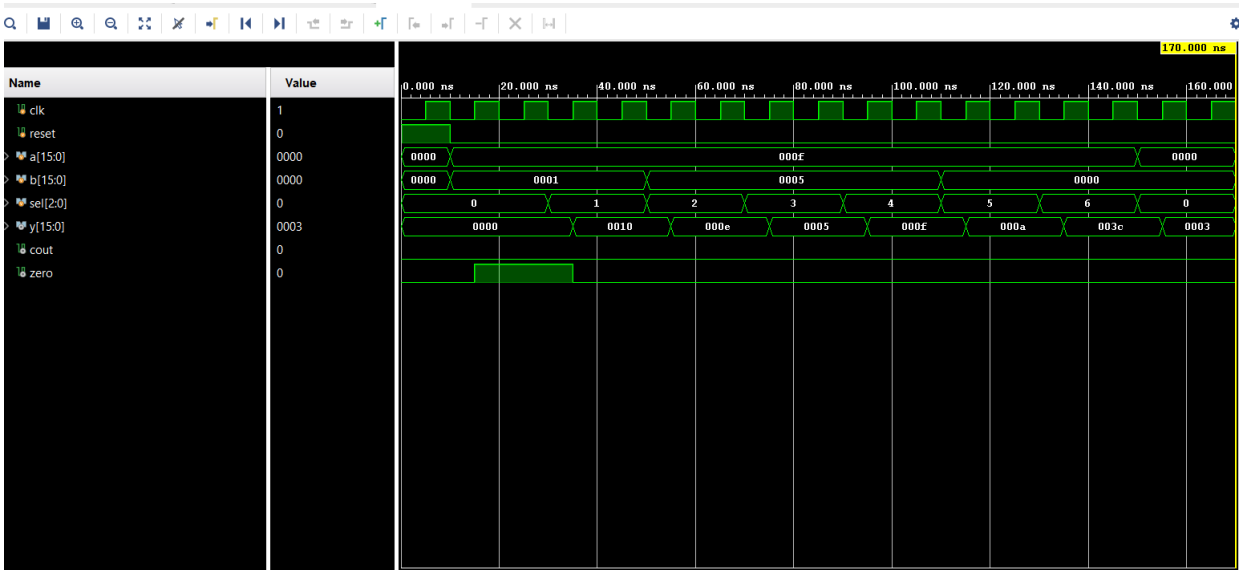
Testbench

```
module ALU_16bit_3s_Pipelined_TB;

reg clk;          // Clock signal
reg reset;        // Reset signal
```

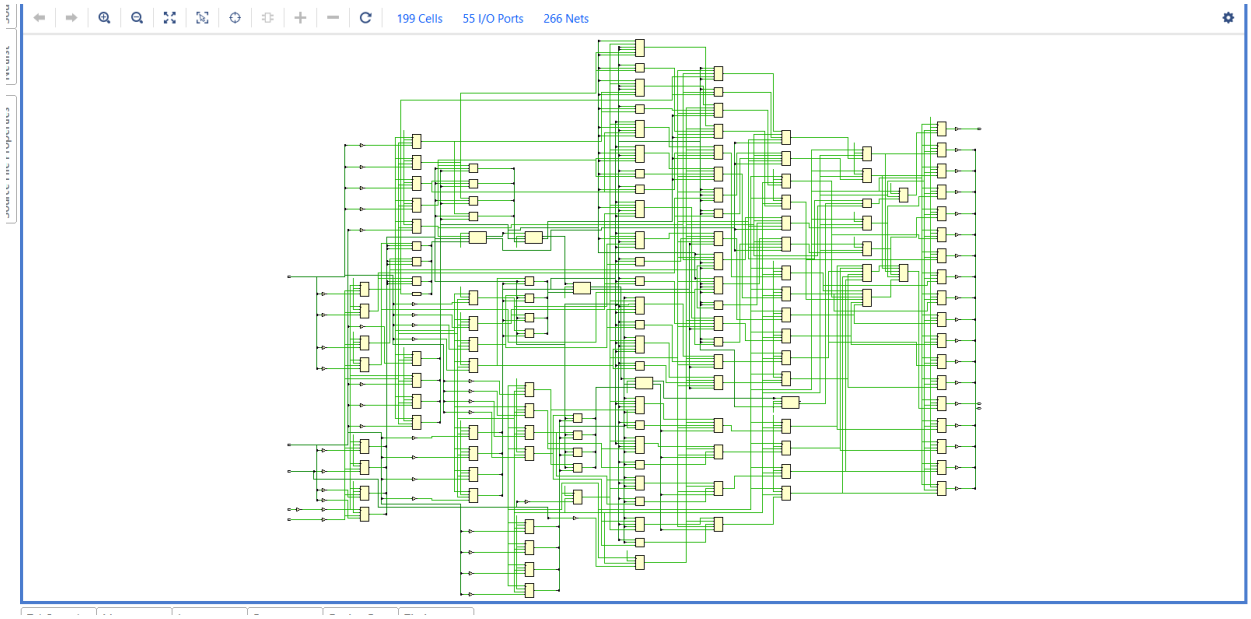
```
reg [15:0] a;      // First operand
reg [15:0] b;      // Second operand
reg [2:0] sel;     // Operation selector
wire [15:0] y;     // Result of the operation
wire cout;         // Carry out for addition
wire zero;         // Zero flag

// Instantiate the ALU
ALU_16bit_3s_Pipelined uut (
   .clk(clk),
   .reset(reset),
   .a(a),
   .b(b),
   .sel(sel),
   .y(y),
   .cout(cout),
   .zero(zero)
);

// Clock generation
always begin
   #5 clk = ~clk;
end

initial begin
   // Initialize signals
   clk = 0;
   reset = 1;
   a = 0;
   b = 0;
   sel = 0;
   #10;
   reset = 0;

   // Monitor the signals
   $monitor("Time = %d: a = %b, b = %b, sel = %b, y = %b, cout = %b, zero = %b", $time, a, b, sel, y,
cout, zero);

   // Test addition
   a = 16'b0000000000001111; b = 16'b0000000000000001; sel = 3'b000;
   #20;

   // Test subtraction
   a = 16'b0000000000001111; b = 16'b0000000000000001; sel = 3'b001;
   #20;

   // Test AND
   a = 16'b0000000000001111; b = 16'b0000000000000101; sel = 3'b010;
   #20;

   // Test OR
   a = 16'b0000000000001111; b = 16'b0000000000000101; sel = 3'b011;
   #20;
```

```
    // Test XOR
    a = 16'b0000000000001111; b = 16'b0000000000000101; sel = 3'b100;
    #20;

    // Test Left Shift
    a = 16'b0000000000001111; b = 16'b0000000000000000; sel = 3'b101;
    #20;

    // Test Right Shift
    a = 16'b0000000000001111; b = 16'b0000000000000000; sel = 3'b110;
    #20;

    // Test Zero Flag
    a = 16'b0000000000000000; b = 16'b0000000000000000; sel = 3'b000;
    #20;

    // End of simulation
    $stop;
end

endmodule
```

Output



Schematic



Reflection

1. Understand various arithmetic and logic operations like addition, subtraction, AND, OR, etc.
2. Learn to design a pipelined digital system with multiple stages to improve performance.
3. Gain proficiency in writing and simulating Verilog code to design and test digital systems.

Open Ended	7	Design and implementation of 5 stage pipeline RISC V Processor.

CODE:

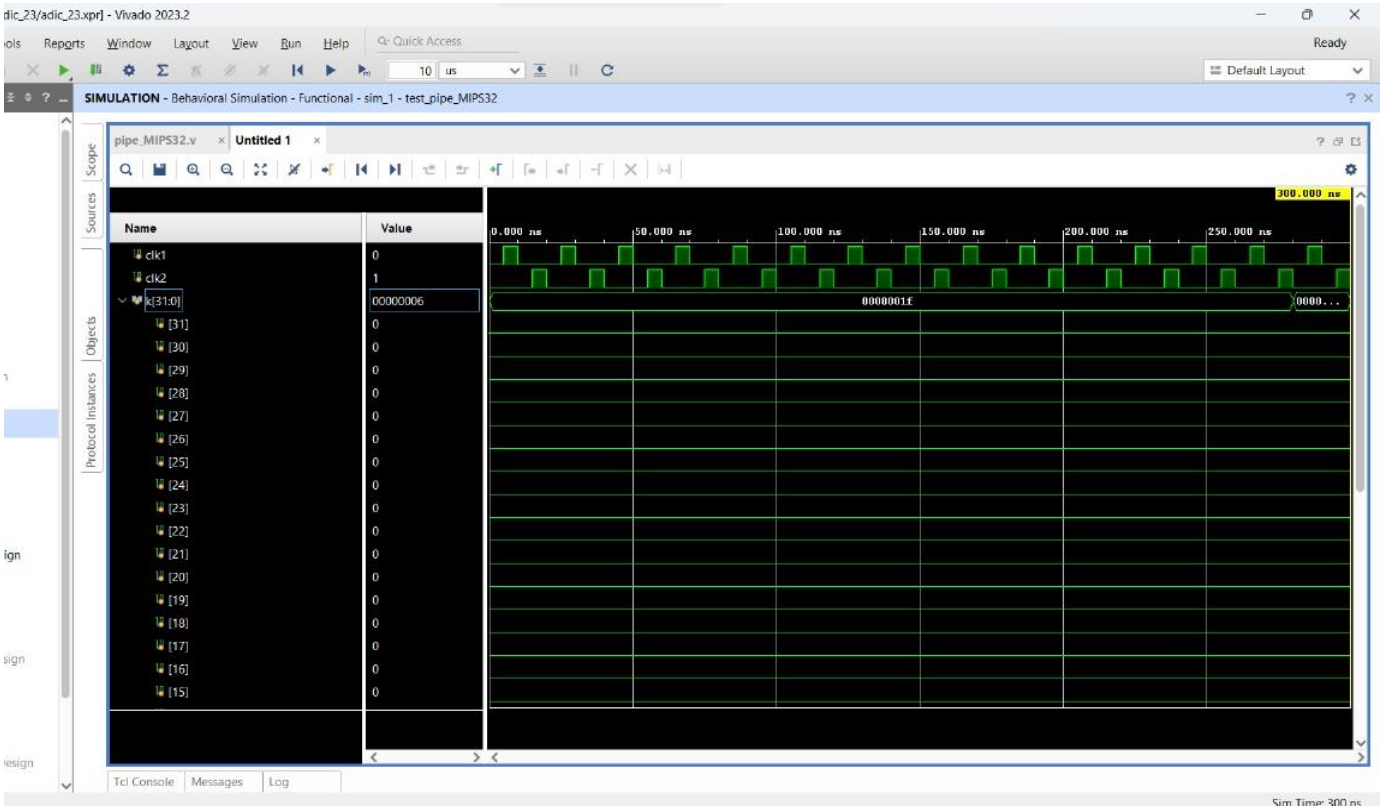
```
module pipe_MIPS32(clk1, clk2);
input clk1, clk2;

reg [31:0] PC, IF_ID_IR, IF_ID_NPC;
reg [31:0] ID_EX_IR, ID_EX_NPC, ID_EX_A, ID_EX_B, ID_EX_Imm, ID_EX_type;
reg [31:0] EX_MEM_IR, EX_MEM_ALUout, EX_MEM_type, EX_MEM_B;
reg        EX_MEM_cond;
reg [31:0] MEM_WB_IR, MEM_WB_ALUout, MEM_WB_LMD, MEM_WB_type;
reg [31:0] Reg [0:31];
reg [31:0] MEM [0:1023];

parameter ADD=6'b000000, SUB=6'b000001, AND=6'b000010, OR=6'b000011, SLT=6'b000100,
MUL=6'b000101, HLT=6'b111111,
LM=6'b001000, SW=6'b001001, ADDI=6'b001010, SUBI=6'b001011, SLTI=6'b001100, BNEQZ=6'b001101,
BEQZ=6'b001110;

parameter RR_ALU=3'b000, RM_ALU=3'b001, LOAD=3'b010, STORE=3'b011, BRANCH=3'b100,
HALT=3'b101;

reg HALTED;

reg TAKEN_BRANCH;

always @(posedge clk1) //Instruction Fetch stage 1
begin
if(HALTED == 0)
begin
if(((EX_MEM_IR [31:26]== BEQZ) && (EX_MEM_cond == 1)) || ((EX_MEM_IR [31:26]== BNEQZ) &&
(EX_MEM_cond == 0)))
begin
IF_ID_IR     <= #2 MEM[EX_MEM_ALUout];
TAKEN_BRANCH <= #2 1'b1;
IF_ID_NPC    <= #2 EX_MEM_ALUout + 1;
PC           <= #2 EX_MEM_ALUout + 1;
end

else
begin
IF_ID_IR     <= #2 MEM[PC];
IF_ID_NPC    <= #2 PC + 1;
PC           <= #2 PC + 1;
end
end
end

always @(posedge clk2) //Instruction Decode stage 2
begin
```

```
if(HALTED == 0)
begin
if(IF_ID_IR[25:21] == 5'b00000)
ID_EX_A <= 0;
else
ID_EX_A <= #2 Reg[IF_ID_IR[25:21]]; //rs

if(IF_ID_IR[20:16] == 5'b00000)
ID_EX_B <= 0;
else
ID_EX_B <= #2 Reg[IF_ID_IR[20:16]]; //rt

ID_EX_NPC <= #2 IF_ID_NPC;
ID_EX_IR <= #2 IF_ID_IR;
ID_EX_Imm <= #2 {{16{IF_ID_IR[15]}}, {IF_ID_IR[15:0]}};

case(IF_ID_IR[31:26])
ADD,SUB,AND,OR,SLT,MUL: ID_EX_type <= #2 RR_ALU;
ADDI,SUBI,SLTI:        ID_EX_type <= #2 RM_ALU;
LM:                    ID_EX_type <= #2 LOAD;
SW:                    ID_EX_type <= #2 STORE;
BNEQZ, BEQZ:           ID_EX_type <= #2 BRANCH;
HLT:                   ID_EX_type <= #2 HALT;
default:           ID_EX_type <= #2 HALT; //invalid opcode
endcase
end
end

always @(posedge clk1) //Instruction Execution stage 3
begin
if(HALTED == 0)
begin
EX_MEM_type  <= #2 ID_EX_type;
EX_MEM_IR    <= #2 ID_EX_IR;
TAKEN_BRANCH <= #2 0;

case(ID_EX_type)
RR_ALU: begin
case(IF_ID_IR[31:26]) //opcode
ADD:     EX_MEM_ALUout <= #2 ID_EX_A + ID_EX_B;
SUB:     EX_MEM_ALUout <= #2 ID_EX_A - ID_EX_B;
AND:     EX_MEM_ALUout <= #2 ID_EX_A && ID_EX_B;
OR:      EX_MEM_ALUout <= #2 ID_EX_A || ID_EX_B;
SLT:     EX_MEM_ALUout <= #2 ID_EX_A < ID_EX_B;
MUL:     EX_MEM_ALUout <= #2 ID_EX_A * ID_EX_B;
default: EX_MEM_ALUout <= #2 32'hxxxxxxxx;
endcase
end
RM_ALU: begin
case(IF_ID_IR[31:26]) //opcode
ADDI:    EX_MEM_ALUout <= #2 ID_EX_A + ID_EX_Imm;
SUBI:    EX_MEM_ALUout <= #2 ID_EX_A - ID_EX_Imm;
SLTI:    EX_MEM_ALUout <= #2 ID_EX_A < ID_EX_Imm;
```

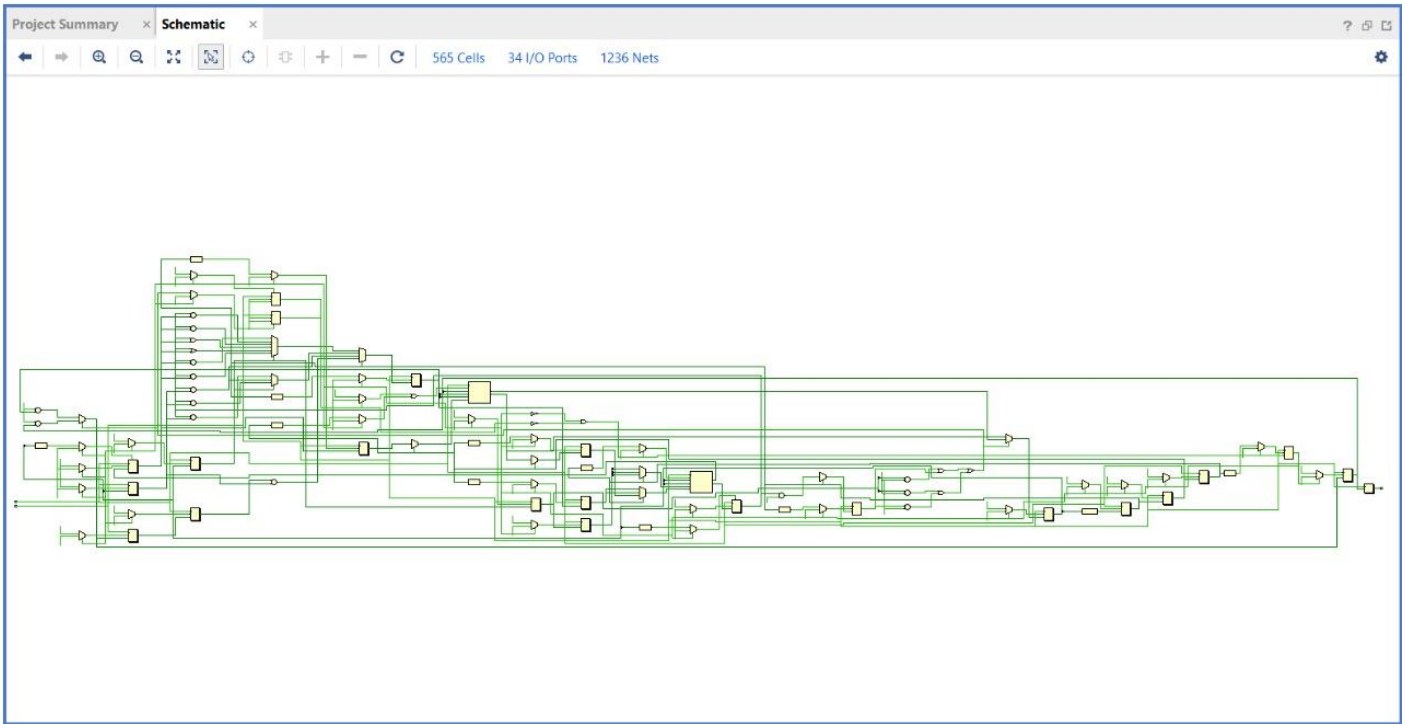
```
default: EX_MEM_ALUout <= #2 32'hxxxxxxxx;
endcase
end
LOAD, STORE: begin
EX_MEM_ALUout <= #2 ID_EX_A + ID_EX_Imm;
EX_MEM_B      <= #2 ID_EX_B;
end
BRANCH: begin
EX_MEM_ALUout <= #2 ID_EX_NPC + ID_EX_Imm;
EX_MEM_cond   <= #2 (ID_EX_A == 0);
end
endcase
end
end

always @(posedge clk2) //Writing back to Memory stage 4
begin
if(HALTED == 0)
begin
MEM_WB_type <= EX_MEM_type;
MEM_WB_IR <= #2 EX_MEM_IR;

case(EX_MEM_type) //opcode
RR_ALU, RM_ALU: MEM_WB_ALUout <= #2 EX_MEM_ALUout;
LOAD:           MEM_WB_LMD    <= #2 MEM[EX_MEM_ALUout];
STORE: if(TAKEN_BRANCH == 0) //DISABLE WRITE
MEM_WB_ALUout <= #2 EX_MEM_B;
endcase
end
end

always @(posedge clk1) //Writing back to Register stage 4
begin
if(TAKEN_BRANCH == 0) //DISABLE WRITE IF BRANCH TAKEN
begin
case(MEM_WB_type) //opcode
RR_ALU: Reg[MEM_WB_IR[15:11]] <= #2 MEM_WB_ALUout; //rd
RM_ALU: Reg[MEM_WB_IR[20:16]] <= #2 MEM_WB_ALUout; //rt
LOAD:   Reg[MEM_WB_IR[20:16]] <= #2 MEM_WB_LMD;    //rt
HALT:   HALTED <= #2 1'b1;
endcase
end
end

endmodule
module test_pipe_MIPS32;

reg clk1, clk2;
integer k;

pipe_MIPS32 mips(clk1, clk2);

initial begin
```

```
clk1 = 0; clk2 = 0;
repeat(20) //generating 2 phase clock
begin
#5 clk1 = 1; #5 clk1 = 0;
#5 clk2 = 1; #5 clk2 = 0;
end
end

initial begin
for(k=0;k<31;k=k+1)
mips.Reg[k] = k;
mips.MEM[0] = 32'h2801000a; //addi r1, r0, 10
mips.MEM[1] = 32'h28020014; //addi r2, r0, 20
mips.MEM[2] = 32'b00101000000000110000000000011110; //addi r3, r0, 30
mips.MEM[3] = 32'h0ce77800; // or  r7, r7, r7
mips.MEM[4] = 32'h0ce77800; // or  r7, r7, r7
mips.MEM[5] = 32'h00222000; //add r4, r1, r2
mips.MEM[6] = 32'h0ce77800; // or  r7, r7, r7
mips.MEM[7] = 32'h00832800; //add r5, r4, r3
mips.MEM[8] = 32'hfc000000; //hlt

mips.HALTED        = 0;
mips.PC            = 0;
mips.TAKEN_BRANCH = 0;

#280;
for(k=0;k<6;k=k+1)
$display("R%1d - %2d", k, mips.Reg[k]);
end

initial begin
$dumpfile ("mips.vcd");
$dumpvars (0, test_pipe_MIPS32);
#300 $finish;
end
endmodule
```

OUTPUT:


dic_23/adic_23.xpr] - Vivado 2023.2
Reports
Window
Layout
View
Run
Help
Quick Access
Ready
Default Layout
SIMULATION - Behavioral Simulation - Functional - sim_1 - test_pipe_MIPS32
pipe_MIPS32.v
Untitled 1
Name
Value
clk1
clk2
k[31:0]
00000006
0000001E
0.000 ns
50.000 ns
100.000 ns
150.000 ns
200.000 ns
250.000 ns
300.000 ns
Tcl Console
Messages
Log

Schematic:


Project Summary
Schematic
565 Cells
34 I/O Ports
1236 Nets

Reflection:

1. Learn about different types of pipeline hazards (data, control, and structural hazards) and techniques to mitigate them, such as forwarding, stalls, and branch prediction.

2.How RISC processors streamline the instruction set to facilitate efficient pipelining.