Class 1 : 2 / 3 / 24

FULL ADDER

CODE:

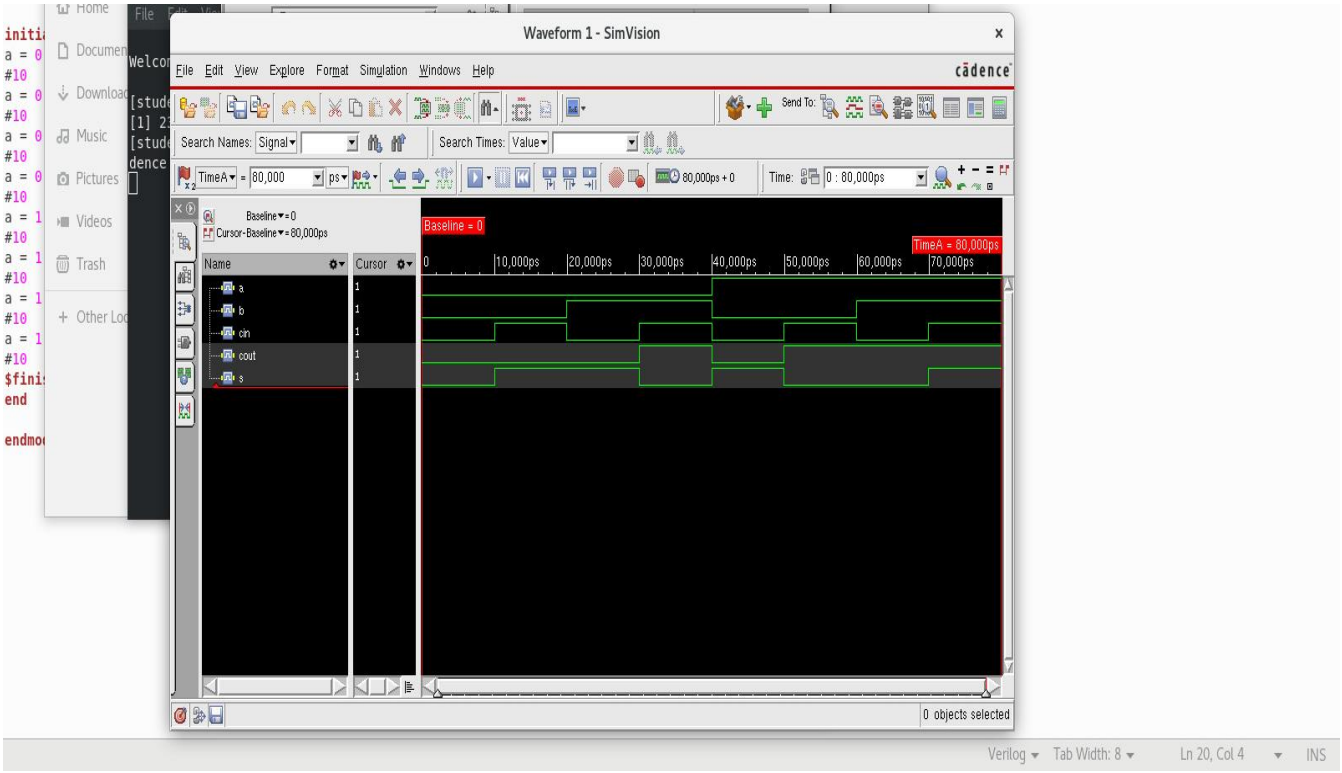
```
`timescale 1ns/1ps
module full(a,b,cin,s,cout);
input a,b,cin;
output s,cout;

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.

Class 2: 6 / 3 / 24

7 SEGMENT

CODE:

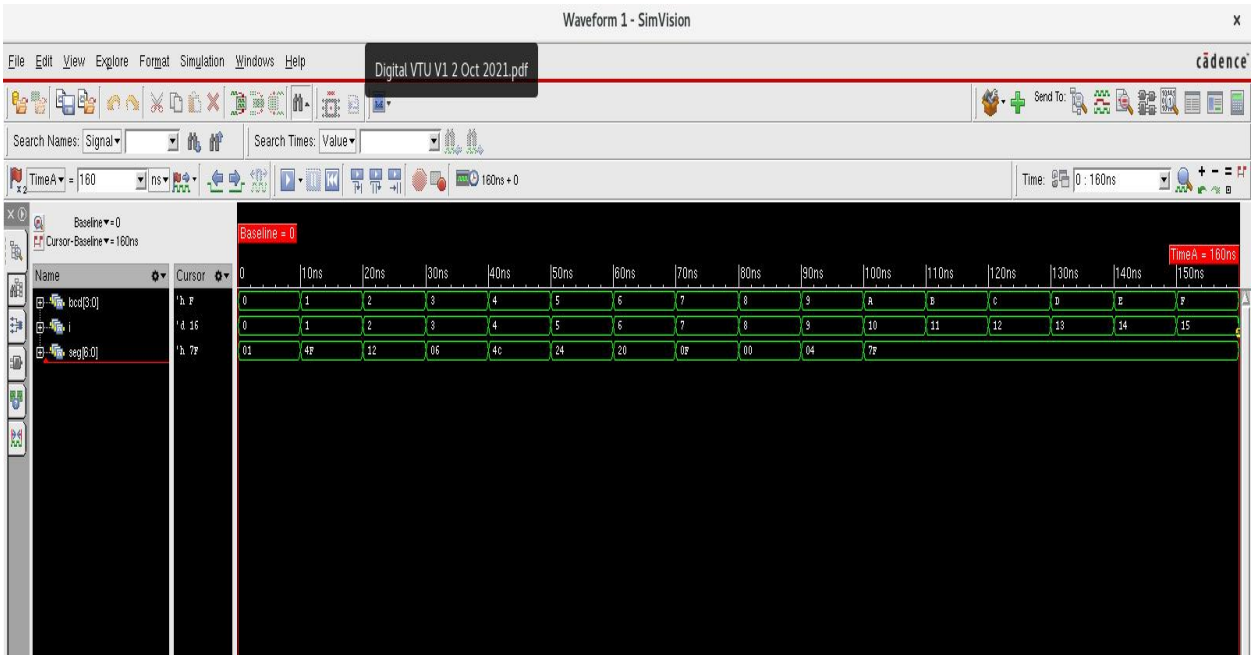
```
module segment7(bcd,seg);

        //Declare inputs,outputs and internal variables.
        input [3:0] bcd;
        output [6:0] seg;
        reg [6:0] seg;

//always block for converting bcd digit into 7 segment format
        always @(bcd)
        begin
        case (bcd) //case statement
        0 : seg = 7'b0000001;
        1 : seg = 7'b1001111;
        2 : seg = 7'b0010010;
        3 : seg = 7'b0000110;
        4 : seg = 7'b1001100;
        5 : seg = 7'b0100100;
        6 : seg = 7'b0100000;
        7 : seg = 7'b0001111;
        8 : seg = 7'b0000000;
        9 : seg = 7'b0000100;
        //switch off 7 segment character when the bcd digit is not a decimal number.
        default : seg = 7'b1111111;
        endcase
        end

Endmodule
```

TEST BENCH:

```
module tb_segment7;
```

```
        reg [3:0] bcd;
        wire [6:0] seg;
        integer i;

        // Instantiate the Unit Under Test (UUT)
        segment7 uut (
        .bcd(bcd),
        .seg(seg)
        );

//Apply inputs
        initial begin
        for(i = 0;i < 16;i = i+1) //run loop for 0 to 15.
        begin
        bcd = i;
        #10; //wait for 10 ns
        end
        end

endmodule
```

OUTPUT:



Reflection:

1. 7-segment display has practical applications in various embedded systems, such as digital clocks, temperature displays, and instrumentation panels, exposing students to real-world engineering challenges.

UP DOWN COUNTER

CODE:

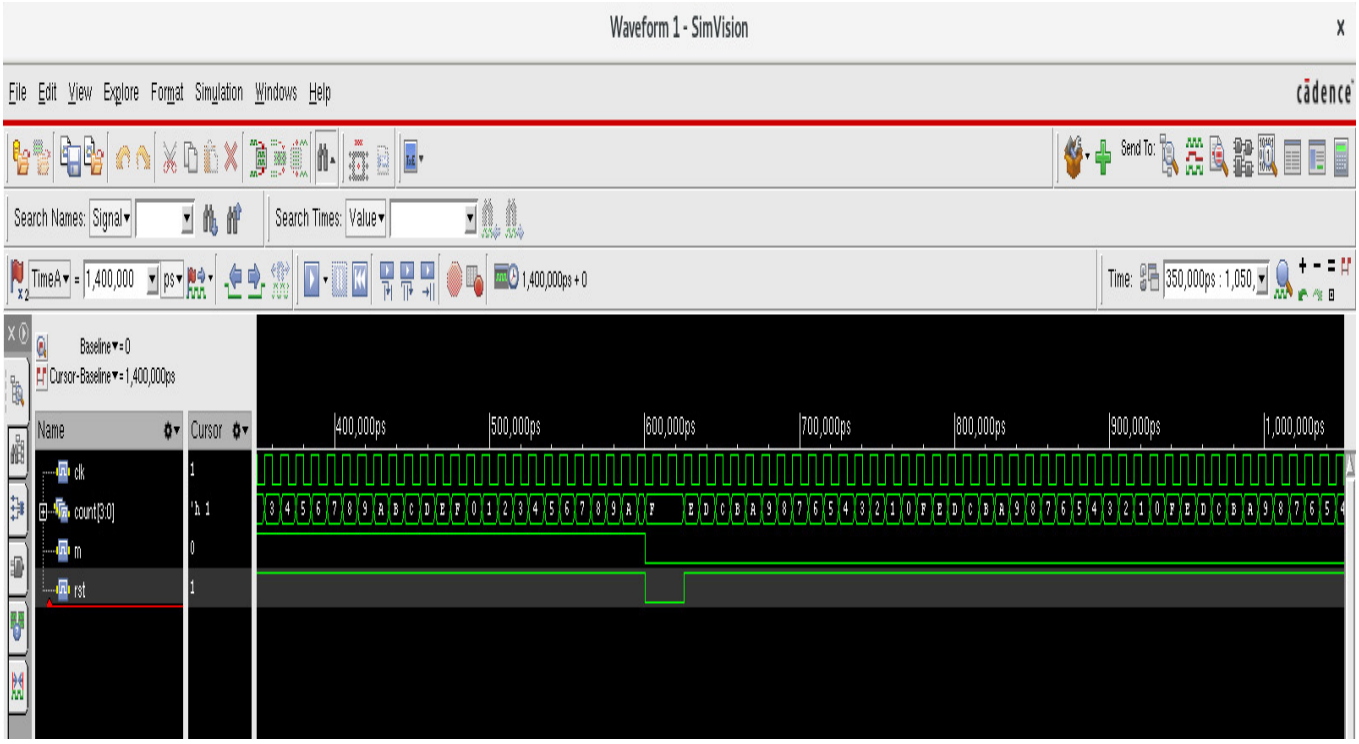
```
`timescale 1ns/1ps
//Defining Module
module counter(clk,rst,m,count);
//Defining Inputs and Outputs (4bit)
input clk,rst,m;
output reg [3:0]count;
//The Block is executed when EITHER of positive edge of clock
//Both are independent events or Neg Edge of Rst arrives
always@(posedge clk or negedge rst)
begin
if(!rst)
count=0;
if(m)
count=count+1;
else
count=count-1;
end
Endmodule
```

TEST BENCH:

```
`timescale 1ns/1ps // Creating Time Scale as in Source Code
module counter_test; // Defining Module Name without Port List
reg clk, rst,m; // Defining I/P as Registers [to Hold Values]
```

```
wire [3:0] count; // Defining O/P as Wires [To Probe Waveforms]
initial
begin
clk=0; // Initializing Clock and Reset
rst=0;#25; // All O/P is 4'b0000 from t=0 to t=25ns.
rst=1; // Up-Down counting is allowed at posedge clk
end
initial
begin
m=1; // Condition for Up-Count
#600 m=0; // Condition for Down-Count
rst=0;#25;
rst=1;
#500 m=0;
end
counter counter1(clk,rst,m,count); // Instantiation of Source Code
always #5 clk=~clk; // Inverting Clk every 5ns
initial
#1400 $finish; // Finishing Simulation at t=1400ns
endmodule
```

OUTPUT:



Reflection:

1. understanding of the process that transforms your written code into a program that your computer can understand and execute.
2. Learned how to create a visual representation of how data changes over time, helping you comprehend and analyze the behavior of programs or circuits that involve signals.

Class 3: 13 / 3 / 24

4_to_1_MUX

Code:

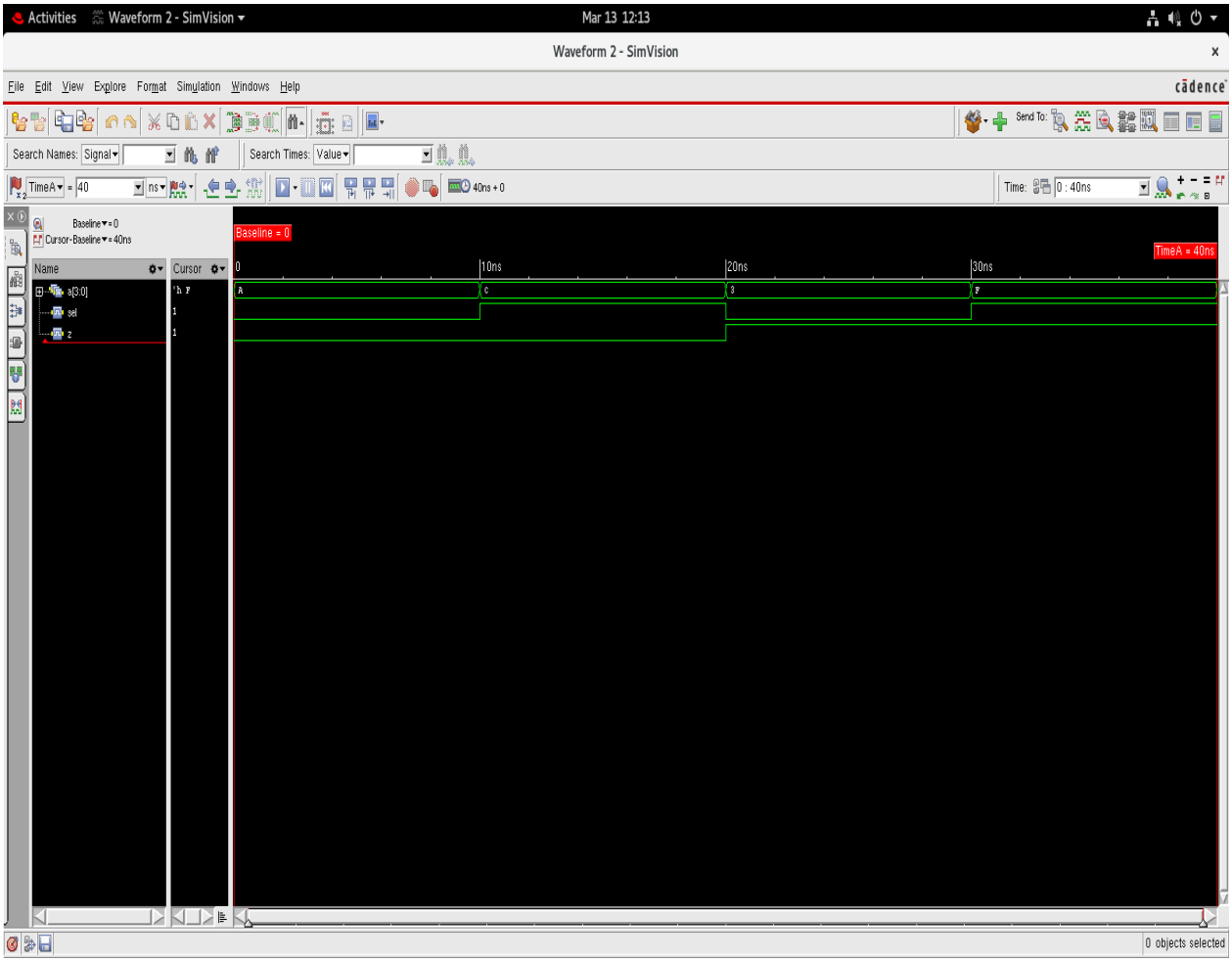
```
module multiplexer_4_to_1 ( output reg z, input [3:0] a, input sel );
 always @*
case (sel)
2'b00: z = a[0];
2'b01: z = a[1];
2'b10: z = a[2];
2'b11: z = a[3];
endcase
Endmodule
```

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.

Class 4: 23 / 3 / 24

Half adder using pipeline

Code:

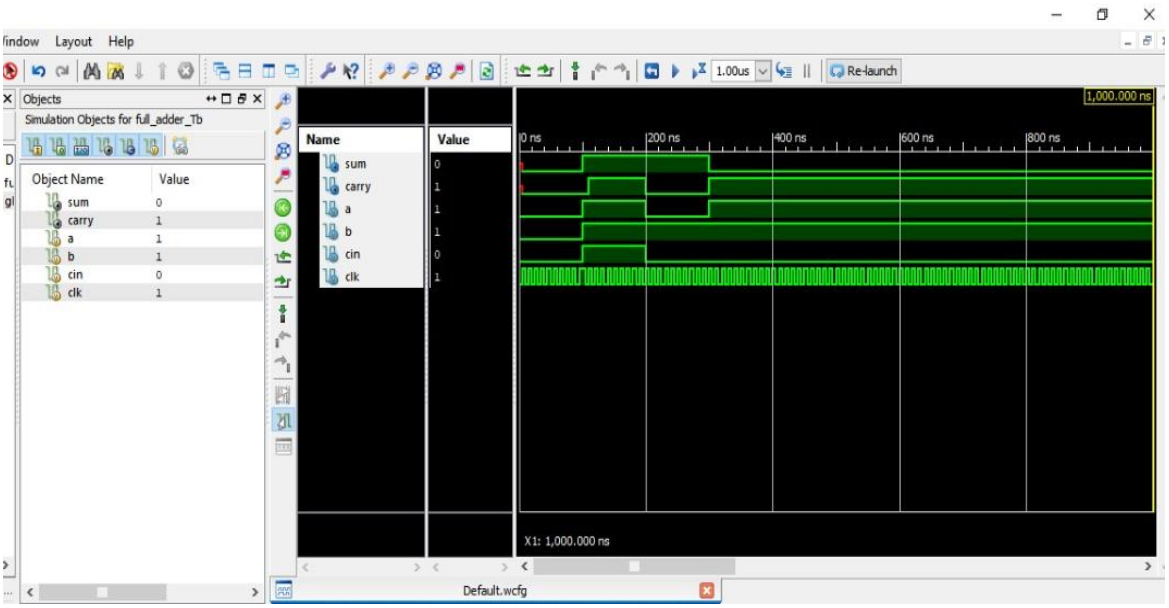
```
module half_adder (input a,b,output sum,carry);
assign sum = a ^ b;
assign carry = a & b;
endmodule

module full_adder2( input a,b,cin,clk, output sum,carry);
wire c1,c2,c3; reg pe1,pe2;
half_adder ha0(a,b,c1,c2);
half_adder ha1(pe1,cin,sum,c3);
assign carry = c3 | pe2 ;
always @(posedge clk)
begin
pe1=c1;
pe2=c2;
end
endmodule
```

Test bench:

```
module full_adder_Tb;

	reg a;
	reg b;
	reg cin;
	reg clk;
	wire sum;
	wire carry;
	full_adder uut (
		.a(a),
		.b(b),
		.cin(cin),
		.clk(clk),
		.sum(sum),
		.carry(carry)
	);
	always #5 clk = ~clk;

	initial begin
		a = 0;
		b = 0;
		cin = 0;
		clk = 0;
		#100;
		a = 1;
		b = 1;
		cin = 1;
		clk = 0;
		#100;
		a = 0;
		b = 1;
		cin = 0;
		clk = 0;
		#100;
		a = 1;
		b = 1;
		cin = 0;
		clk = 0;
```

```
        #100;

    end

endmodule
```

Output :



Reflection :

1. Each stage is implemented using a generate block to create multiple instances of the logic for each bit position**.**
2. This pipeline implementation of a half adder in Verilog demonstrates how the addition operation can be divided into stages to increase throughput and efficiency.

Class 5: / 4 / 24

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 @ (*)
```

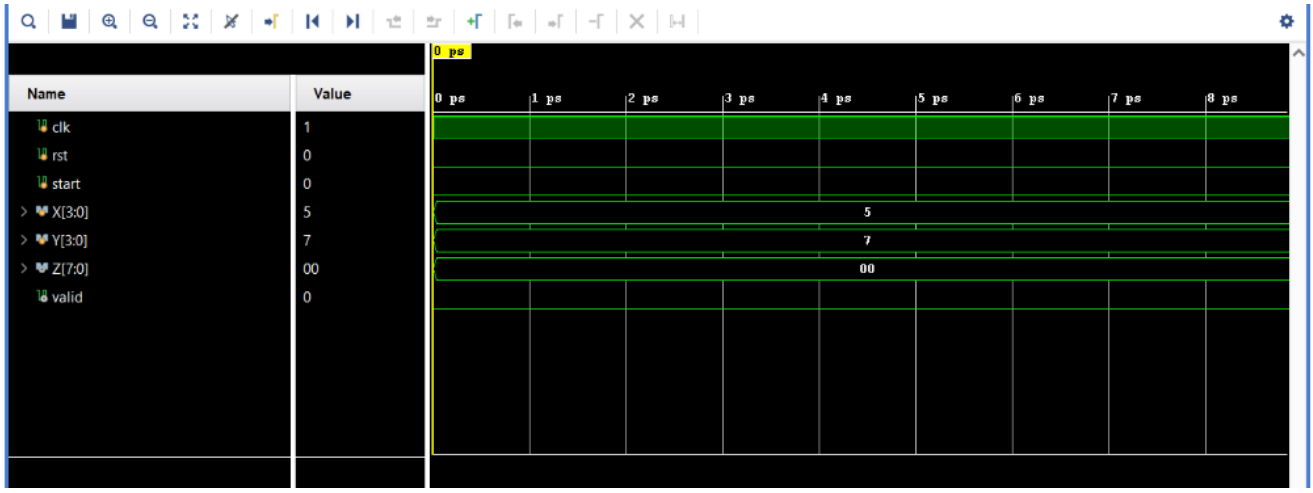
```
begin
case(pres_state)
IDLE:
begin
next_count = 2'b0;
next_valid = 1'b0;
if(start)
begin
   next_state = START;
   next_temp  = {X[0],1'b0};
   next_Z     = {4'd0,X};
end
else
begin
   next_state = pres_state;
   next_temp  = 2'd0;
   next_Z     = 8'd0;
end
end

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:

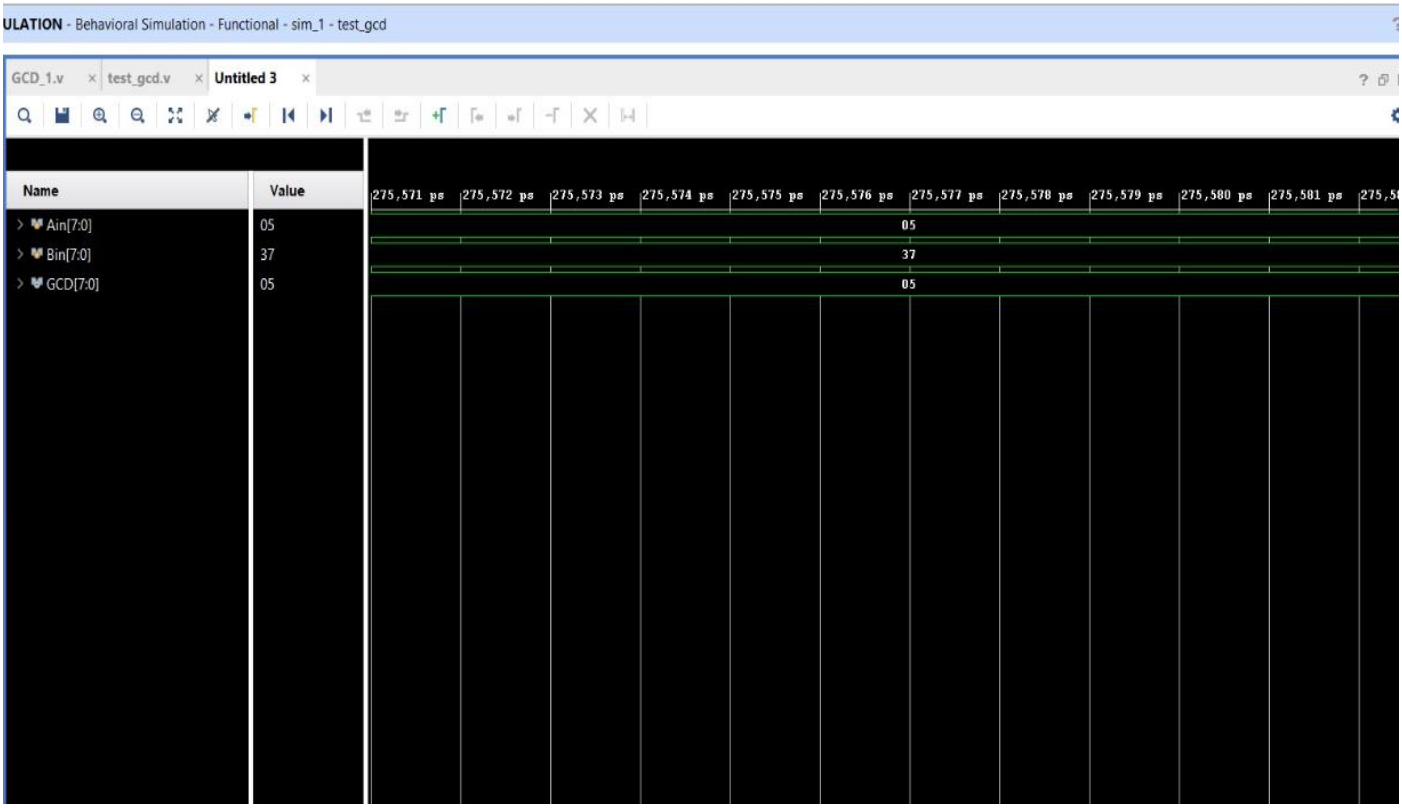
```
module testbench_GCD;

        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.

FIR

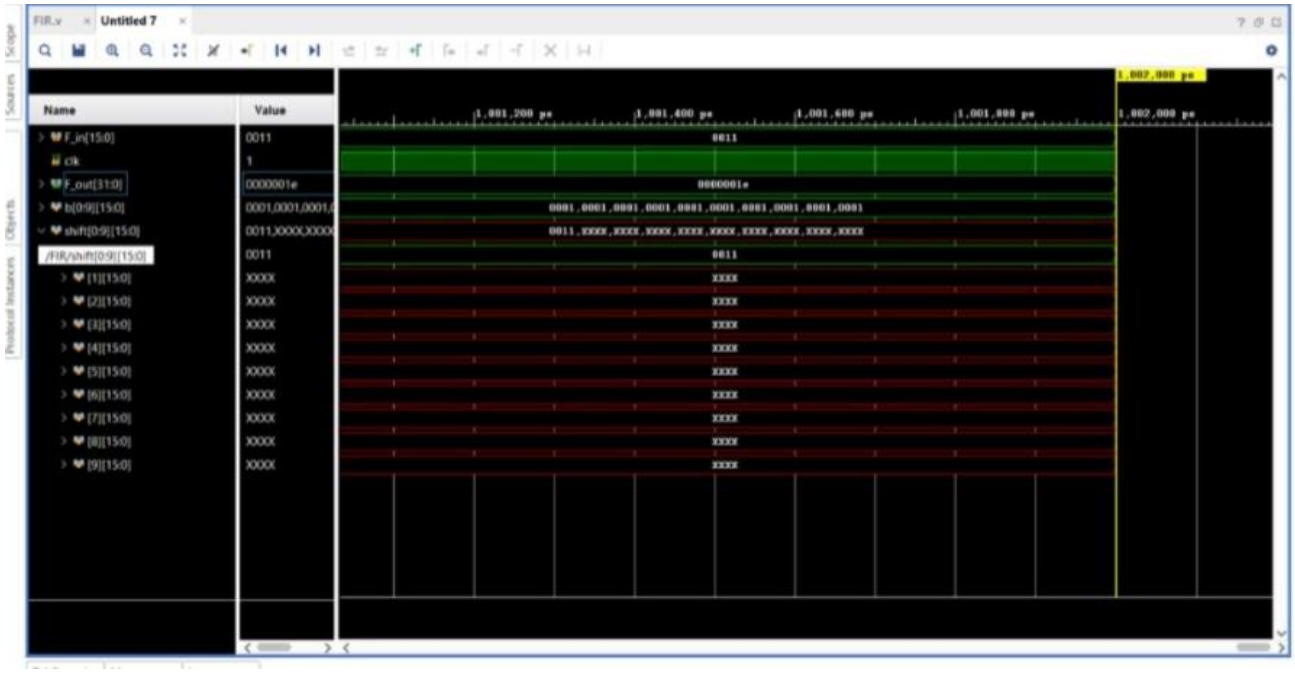
Code

```
module FIR(F_out, F_in, clk);

   input signed [15:0]F_in;

   input clk;

   output reg signed [31:0]F_out;

   reg signed [15:0]b[0:9];

   reg signed [15:0]shift[0:9];

   initial

   begin

   b[0]<=16'd1;

   b[1]<=16'd1;

   b[2]<=16'd1;

   b[3]<=16'd1;

   b[4]<=16'd1;
```

```
b[5]<=16'd1;
b[6]<=16'd1;
b[7]<=16'd1;
b[8]<=16'd1;
b[9]<=16'd1;
end
always@(posedge clk)
begin
shift[0]<=F_in;
shift[1]<=shift[0];
shift[2]<=shift[1];
shift[3]<=shift[2];
shift[4]<=shift[3];
shift[5]<=shift[4];
shift[6]<=shift[5];
shift[7]<=shift[6];
shift[8]<=shift[7];
shift[9]<=shift[8];
end
always@(posedge clk)
begin
F_out<=((shift[0]*b[0])+(shift[1]*b[1])+(shift[2]*b[2])+
      (shift[3]*b[3])+(shift[4]*b[4])+(shift[5]*b[5])+
```

```
        (shift[6]*b[6])+(shift[7]*b[7])+(shift[8]*b[8])+(shift[9]*b[9]));

    end

endmodule
```

Output



Reflection

1. **understood of how FIR filters operate by using a simple 3-tap filter. Each new input sample is multiplied by a coefficient and summed with the previous samples' results.**
2. **The use of shift registers to store previous input samples is crucial in FIR filter design. This example demonstrates the use of a shift register to implement the delay elements needed for FIR filtering**

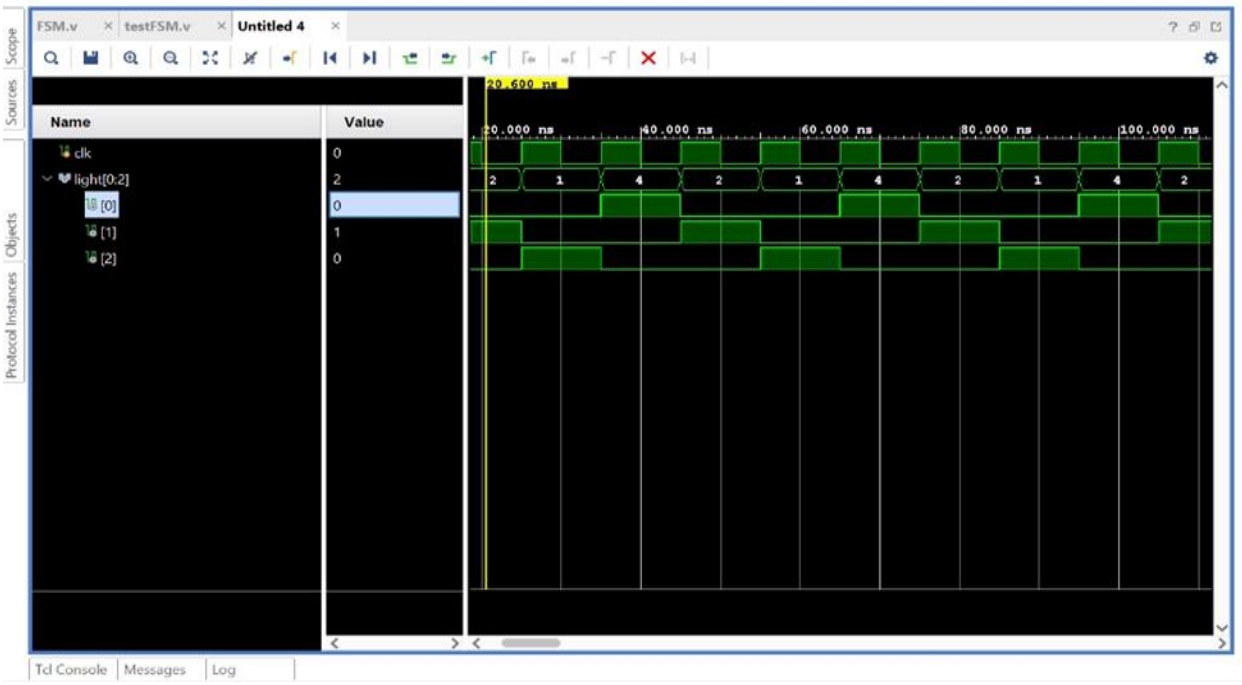
Traffic Light

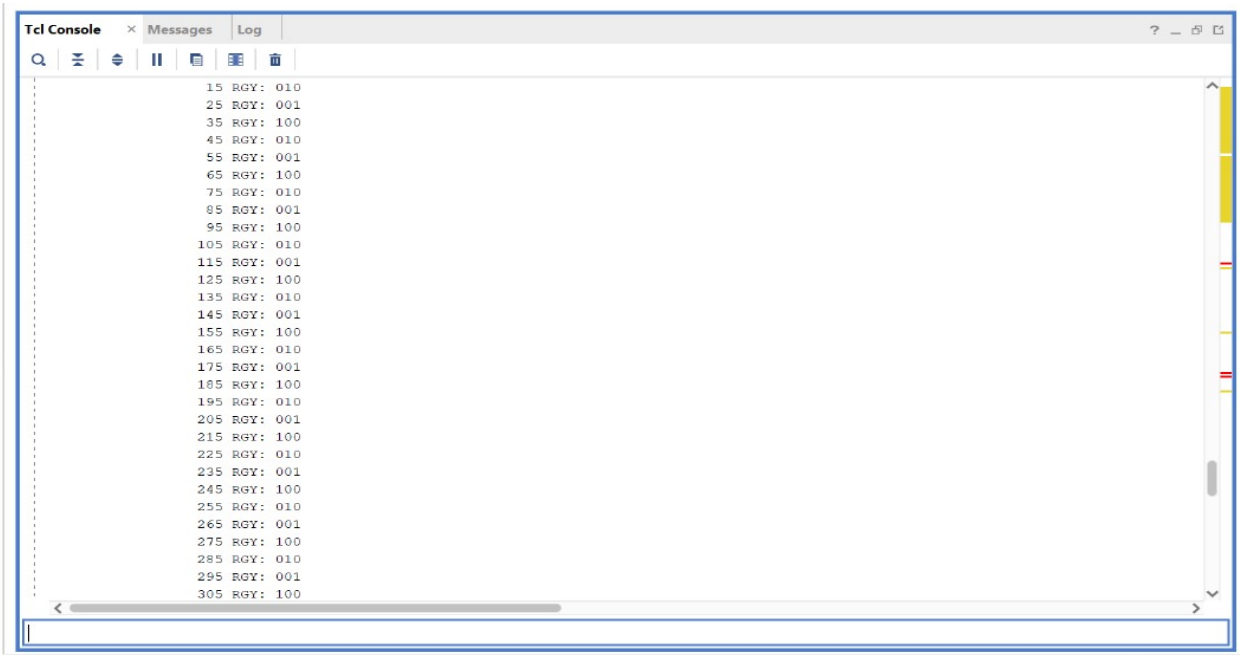
Code:

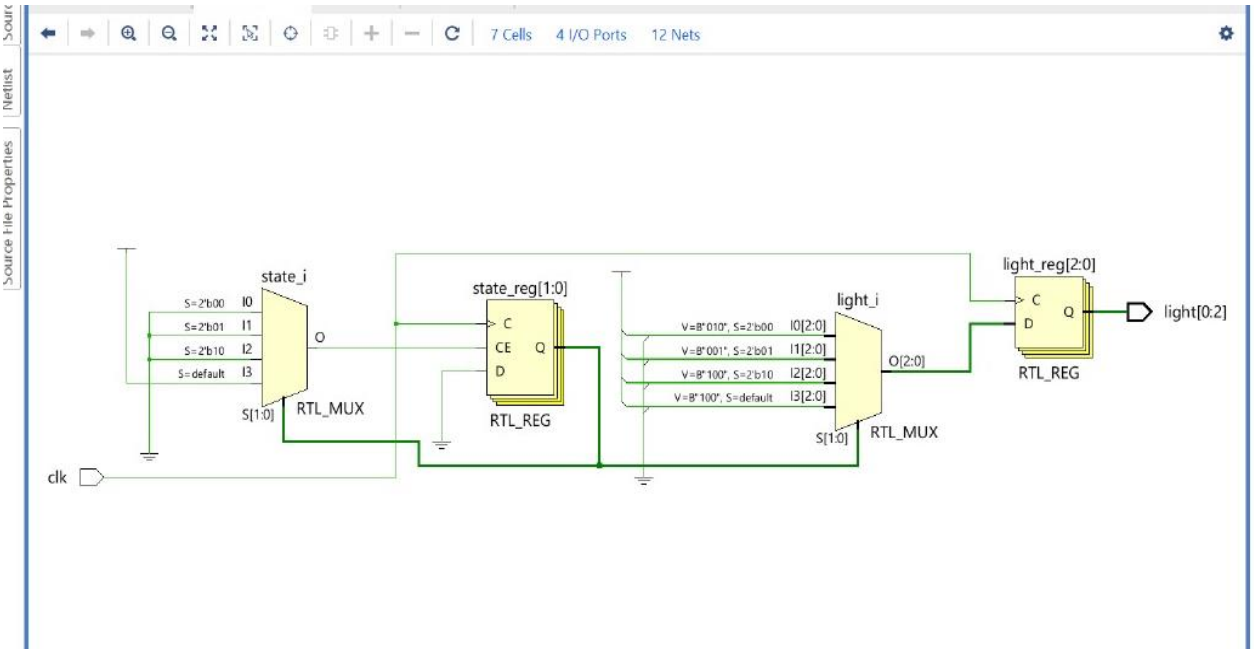
```
module cyclic_lamp (clk, light);
 input clk;
 output reg [2:0] light; // Light output should be 3 bits for RED, GREEN, YELLOW
 parameter s0 = 0; // Initial state
 parameter s1 = 1;
 parameter s2 = 2;
 parameter RED = 3'b100;
 parameter GREEN = 3'b010;
 parameter YELLOW = 3'b001;
 reg [1:0] state = s0; // Initialize state to s0
 always @(posedge clk) begin
  case (state)
   s0: begin
    light <= GREEN;
    state <= s1;
   end
   s1: begin
    light <= YELLOW;
    state <= s2;
   end
```

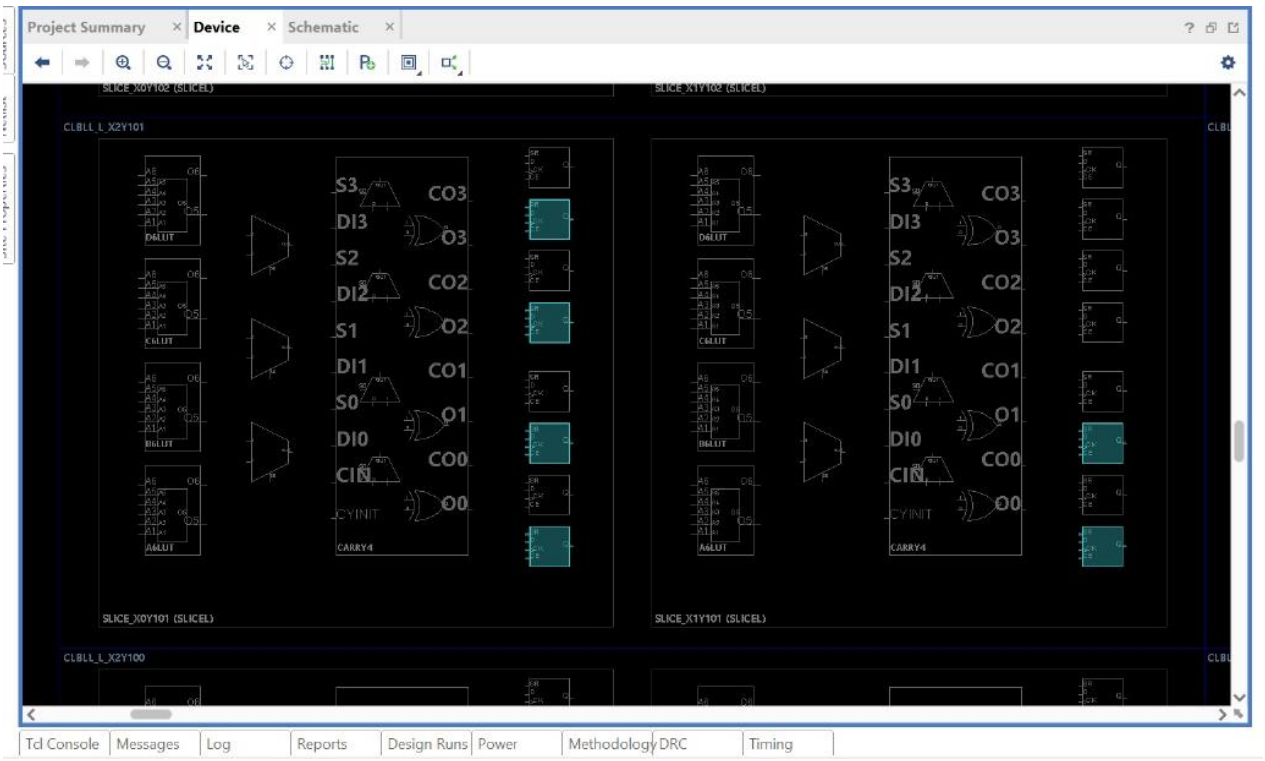
```
        s2: begin
          light <= RED;
          state <= s0;
        end
        default: begin // Explicitly handle undefined states
          light <= RED; // Set light to RED for safety
          state <= s0; // Transition to initial state
        end
      endcase
    end
endmodule
```

Output:




Tcl Console
Messages
Log
15 RGY: 010
25 RGY: 001
35 RGY: 100
45 RGY: 010
55 RGY: 001
65 RGY: 100
75 RGY: 010
85 RGY: 001
95 RGY: 100
105 RGY: 010
115 RGY: 001
125 RGY: 100
135 RGY: 010
145 RGY: 001
155 RGY: 100
165 RGY: 010
175 RGY: 001
185 RGY: 100
195 RGY: 010
205 RGY: 001
215 RGY: 100
225 RGY: 010
235 RGY: 001
245 RGY: 100
255 RGY: 010
265 RGY: 001
275 RGY: 100
285 RGY: 010
295 RGY: 001
305 RGY: 100


Source
Netlist
Source File Properties
7 Cells
4 I/O Ports
12 Nets
state_i
S=2'b00 I0
S=2'b01 I1
S=2'b10 I2
S=default I3
O
S[1:0]
RTL_MUX
state_reg[1:0]
C
CE
D
Q
RTL_REG
clk
light_i
V=8'd10", S=2'b00 I0[2:0]
V=8'd01", S=2'b01 I1[2:0]
V=8'd100", S=2'b10 I2[2:0]
V=8'd100", S=default I3[2:0]
O[2:0]
S[1:0]
RTL_MUX
light_reg[2:0]
C
D
Q
RTL_REG
light[0:2]



Reflection :

1. FSMs facilitate the analysis and verification of system properties. Techniques such as reachability analysis, model checking, and simulation can be applied to FSMs to verify correctness properties, detect errors, and ensure system reliability
2. FSMs can be used to model concurrent and parallel systems by representing concurrent processes as distinct states or state transitions. Techniques such as state synchronization, mutual exclusion, and deadlock avoidance can be applied to manage concurrency and ensure correct system behavior.

Design and implement a pipeline that carries the following stage wise operations.

Code:

```
module pipeline (
input wire clk,
input wire rst,
input wire [3:0] rs1, rs2, rd, rp, // Register addresses
input wire [2:0] alu_fun,  // ALU function code
input wire [15:0] mem_addr,   // Memory address
output reg [15:0] mem_data_out  // Data output to memory
);

// Register file
reg [15:0] reg_file [0:15];

// ALU
reg [15:0] a, b, z;

// Memory
reg [15:0] memory [0:255];

// ALU function definitions localparam ALU_ADD = 3'b000,
ALU_SUB = 3'b001,
ALU_AND = 3'b010,
ALU_OR = 3'b011,
ALU_XOR = 3'b100,
 ALU_NOT = 3'b101;

// Pipeline stages
always @(posedge clk or posedgerst) begin
if (rst) begin
a <= 16'd0;
b <= 16'd0;
z <= 16'd0;
mem_data_out<= 16'd0;
end else begin
// Stage 1: Read from register file
```

```
a <= reg_file[rs1];
b <= reg_file[rs2];

// Stage 2: Perform ALU operation
 case (alu_fun)
ALU_ADD: z <= a + b;
ALU_SUB: z <= a - b;
ALU_AND: z <= a & b;
ALU_OR: z <= a | b;
ALU_XOR: z <= a ^ b;
ALU_NOT: z <= ~a;
default: z <= 16'd0;
 endcase

// Stage 3: Write to register file
 reg_file[rd] <= z;

// Stage 4: Write to memory memory
[mem_addr] <= z;
mem_data_out<=
 z; end
end endmodule
```

Test bench:

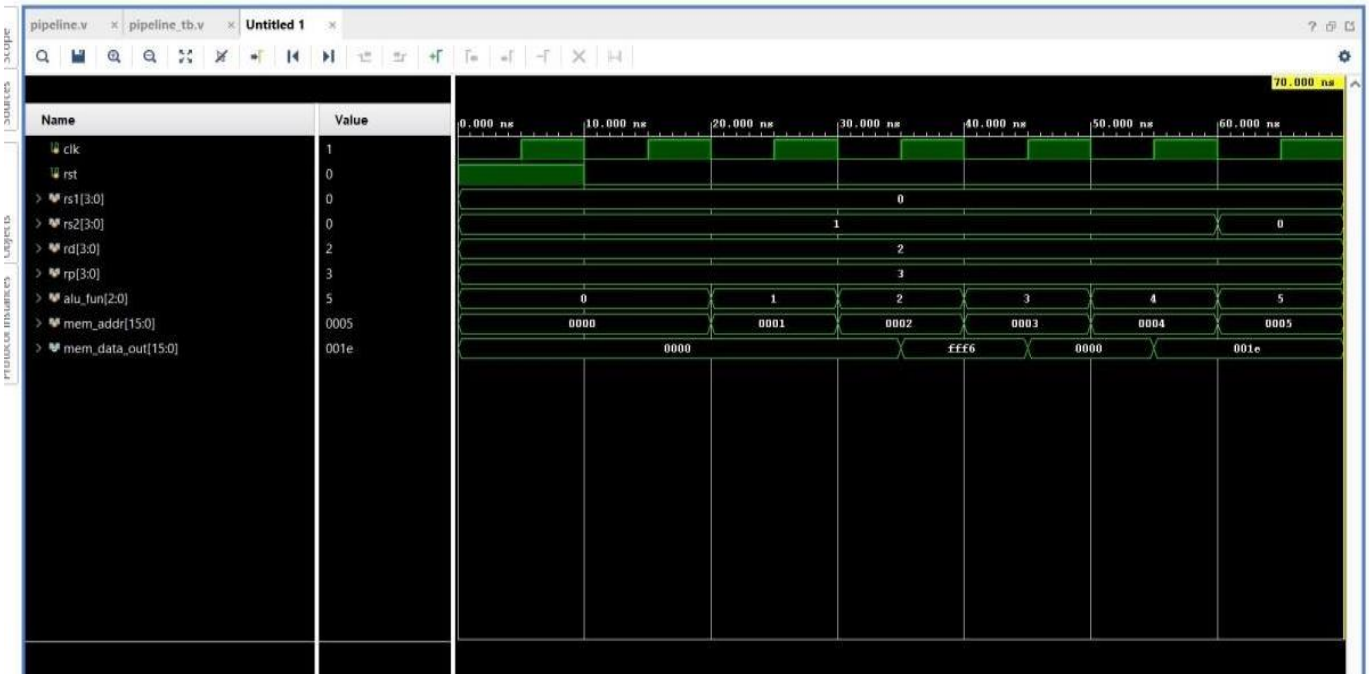
```
module pipeline_tb;
reg clk;
reg rst;
reg [3:0] rs1, rs2, rd, rp; // Register addresses reg [2:0] alu_fun;
// ALU function code reg [15:0] mem_addr;             // Memory address
wire [15:0] mem_data_out;   // Data output to memory

// Instantiate the pipeline module pipeline uut (
.clk(clk),
.rst(rst),
.rs1(rs1),
.rs2(rs2),
.rd(rd),
.rp(rp),
```

```
.alu_fun(alu_fun),
.mem_addr(mem_addr),
.mem_data_out(mem_data_out)
);

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

initial begin
// Initialize signals
clk = 0;
rst = 1;
rs1 = 4'd0;
rs2 = 4'd1;
rd = 4'd2;
rp = 4'd3;
alu_fun = 3'd0;
mem_addr = 16'd0;

// Initialize register file and memory
 uut.reg_file[0] = 16'd10; // Register 0
 uut.reg_file[1] = 16'd20; // Register 1
 uut.reg_file[2] = 16'd0; // Register 2
 uut.reg_file[3] = 16'd0; // Register 3

#10 rst = 0; // Release reset

// Test case 1: ADD rs1 and rs2, store in rd and memory
alu_fun = 3'b000; // ADD operation
rs1 = 4'd0;
rs2 = 4'd1;
rd = 4'd2;
mem_addr = 16'd0;
#10;

// Check results
$display("Test Case 1: ADD");
$display("reg_file[2] = %d, memory[0] = %d", uut.reg_file[2], uut.memory[0]);
$display("mem_data_out = %d", mem_data_out);
  // Test case 2: SUB rs1 and rs2, store in rd and memory
```

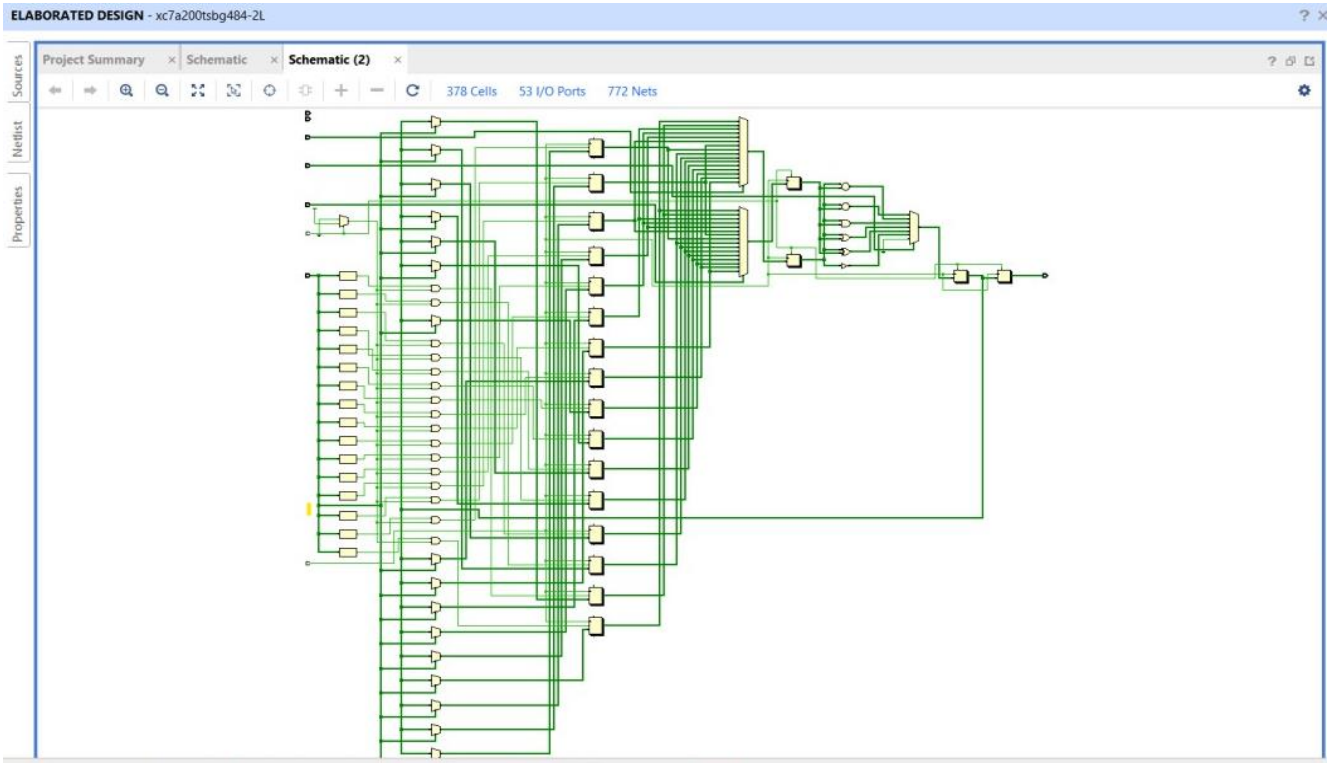
```
alu_fun = 3'b001; // SUB operation
 rs1 = 4'd0;
 rs2 = 4'd1;
rd = 4'd2;
mem_addr = 16'd1;
#10;

// Check results
$display("Test Case 2: SUB");
$display("reg_file[2] = %d, memory[1] = %d", uut.reg_file[2], uut.memory[1]);
$display("mem_data_out = %d", mem_data_out);

 // Test case 3: AND rs1 and rs2, store in rd and memory
alu_fun = 3'b010; // AND operation
 rs1 = 4'd0;
 rs2 = 4'd1;
rd = 4'd2;
mem_addr = 16'd2;
#10;

// Check results
$display("Test Case 3: AND");
$display("reg_file[2] = %d, memory[2] = %d", uut.reg_file[2], uut.memory[2]);
$display("mem_data_out = %d", mem_data_out);

 // Test case 4: OR rs1 and rs2, store in rd and memory
alu_fun = 3'b011; // OR operation
 rs1 = 4'd0;
 rs2 = 4'd1;
rd = 4'd2;
mem_addr = 16'd3;
#10;
// Check results
$display("Test Case 4: OR");
$display("reg_file[2] = %d, memory[3] = %d", uut.reg_file[2], uut.memory[3]);
$display("mem_data_out = %d", mem_data_out);

// Test case 5: XOR rs1 and rs2, store in rd and memory
alu_fun = 3'b100; // XOR operation
 rs1 = 4'd0;
```

```
 rs2 = 4'd1;
rd = 4'd2;
mem_addr = 16'd4;
 #10;
 // Check results
 $display("Test Case 5: XOR");
 $display("reg_file[2] = %d, memory[4] = %d", uut.reg_file[2], uut.memory[4]);
 $display("mem_data_out = %d", mem_data_out);
 // Test case 6: NOT rs1, store in rd and memory
alu_fun = 3'b101; // NOT operation
 rs1 = 4'd0;
 rs2 = 4'd0; // rs2 is irrelevant for NOT operation
rd = 4'd2;
mem_addr = 16'd5;
 #10;
$display("Test Case 6: NOT");
$display("reg_file[2] = %d, memory[5] = %d", uut.reg_file[2], uut.memory[5]);
$display("mem_data_out = %d", mem_data_out);
$stop;
 end
endmodule
```

OUTPUT:


pipeline.v
pipeline_tb.v
Untitled 1
Name
Value
clk
rst
rs1[3:0]
rs2[3:0]
rd[3:0]
rp[3:0]
alu_fun[2:0]
mem_addr[15:0]
mem_data_out[15:0]
1
0
0
0
2
3
5
0005
001e
70.000 ns
0.000 ns
10.000 ns
20.000 ns
30.000 ns
40.000 ns
50.000 ns
60.000 ns
0000
0001
0002
0003
0004
0005
0000
ffff6
0000
001e

Schematic


ELABORATED DESIGN - xc7a200tsbg484-2L
Sources
Netlist
Properties
Project Summary
Schematic
Schematic (2)
378 Cells
53 I/O Ports
772 Nets

Reflections :

1. understanding of pipeline design, specifically how data flows through multiple stages (fetch, decode, execute, memory access, write-back). This is essential for optimizing CPU performance and increasing instruction throughput
2. The implementation of a reset mechanism to initialize pipeline stages and other components is a critical aspect of designing robust digital systems.

NON PIPELINE

Code:

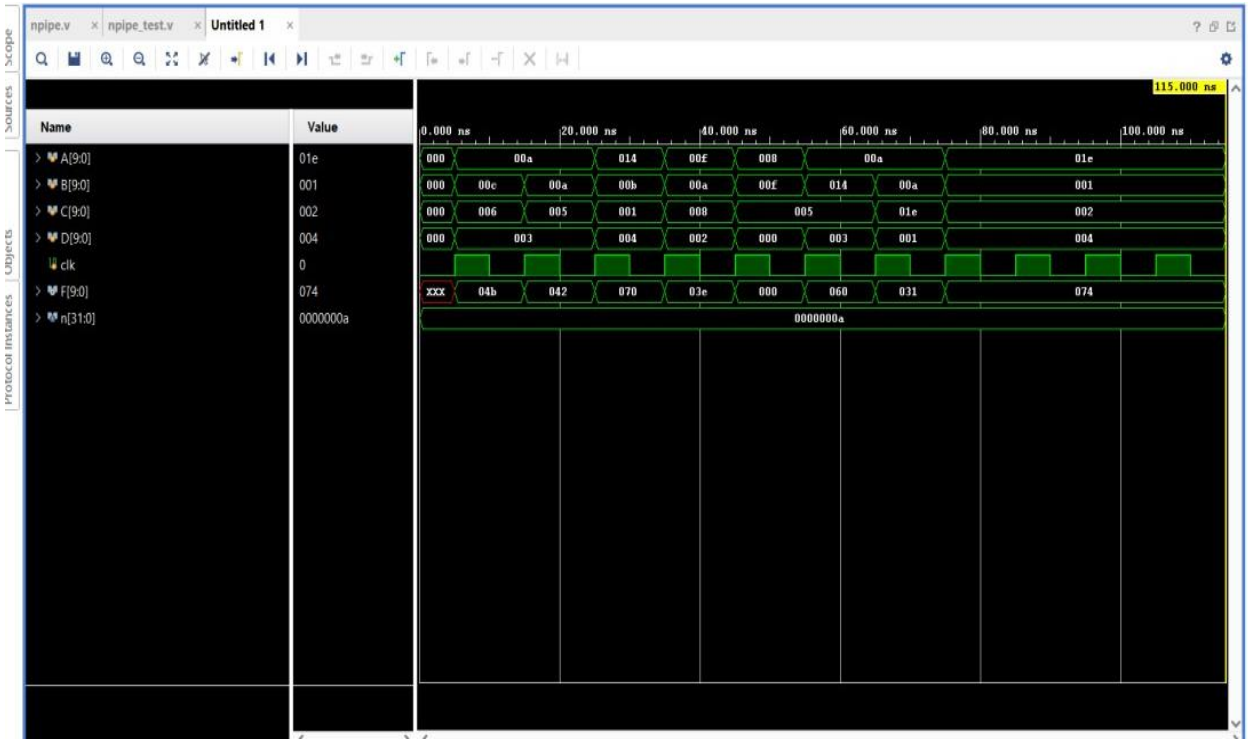
```
module npipe (F, A, B, C, D, clk);
parameter n = 10;
input [n-1:0] A, B, C, D;
input
clk;
output reg [n-1:0] F;
 reg [n-1:0] x1, x2, x3, d;
always @(posedge clk)
begin


// Perform all operations in one clock cycle
//blocking assignment
 x1 = A + B;
 x2 = C - D;
//d = D;
 x3 = x1 + x2;

F = x3 * D;
end
endmodule
```

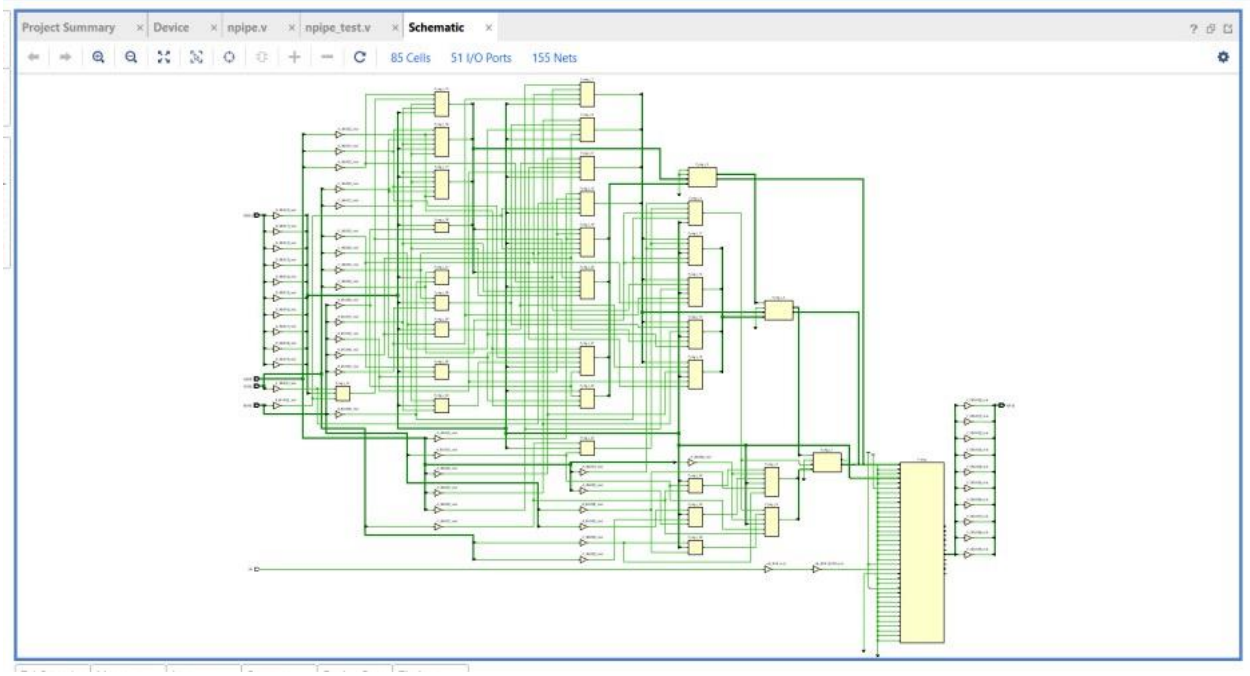
Testbench:

```
`timescale 1ns/1ps
module npipe_test;
// Parameters
parameter n = 10;

// Inputs
reg [n-1:0] A, B, C, D;
reg clk;
//
Outputs wire[
n-1:0] F;

// Instantiate the Unit Under Test (UUT)
npipe #(n) uut (
.A(A),
.B(B),
.C(C),
.D(D),
.clk(clk),
.F(F)
);

initial begin
 // Initialize Inputs
 clk = 0;
 A = 0; B = 0; C = 0; D = 0;

#5 A = 10; B = 12; C = 6; D = 3; // Expected F = (10+12 + 6-3) * 3 = 75
#10 A = 10; B = 10; C = 5; D = 3; // Expected F = (10+10 + 5-3) * 3 = 66
#10 A = 20; B = 11; C = 1; D = 4; // Expected F = (20+11 + 1-4) * 4 = 112
#10 A = 15; B = 10; C = 8; D = 2; // Expected F = (15+10 + 8-2) * 2 = 62
#10 A = 8; B = 15; C = 5; D = 0; // Expected F = (8+15 + 5-0) * 0 = 0
#10 A = 10; B = 20; C = 5; D = 3; // Expected F = (10+20 + 5-3) * 3 = 96
#10 A = 10; B = 10; C = 30; D = 1; // Expected F = (10+10 + 30-1) * 1 = 49
#10 A = 30; B = 1; C = 2; D = 4; // Expected F = (30+1 + 2-4) * 4 = 116

// Finish simulation
 #40 $finish;
end
```

```
// Clock generation
always begin
  #5 clk = ~clk; // Toggle clock every 5 time units end

initial begin
// Monitor changes
$dumpfile("npipe.vcd");
$dumpvars(0, npipe_test);
 $monitor("Time: %3d, A = %d, B = %d, C = %d, D = %d, F = %d", $time, A, B, C, D, F);end
Endmodule
```

OUTPUT





Reflections :

1. Calculating the data path delays from console window
2. Calculating LUT values and comparing with pipelined logic
3. does not implement a true multi-stage pipeline, but the testbench simulates the behavior of a single-stage pipeline where all operations are completed within one clock cycle. This provides a basic understanding of how pipelines can be simulated and verified

Sequence detector

Code

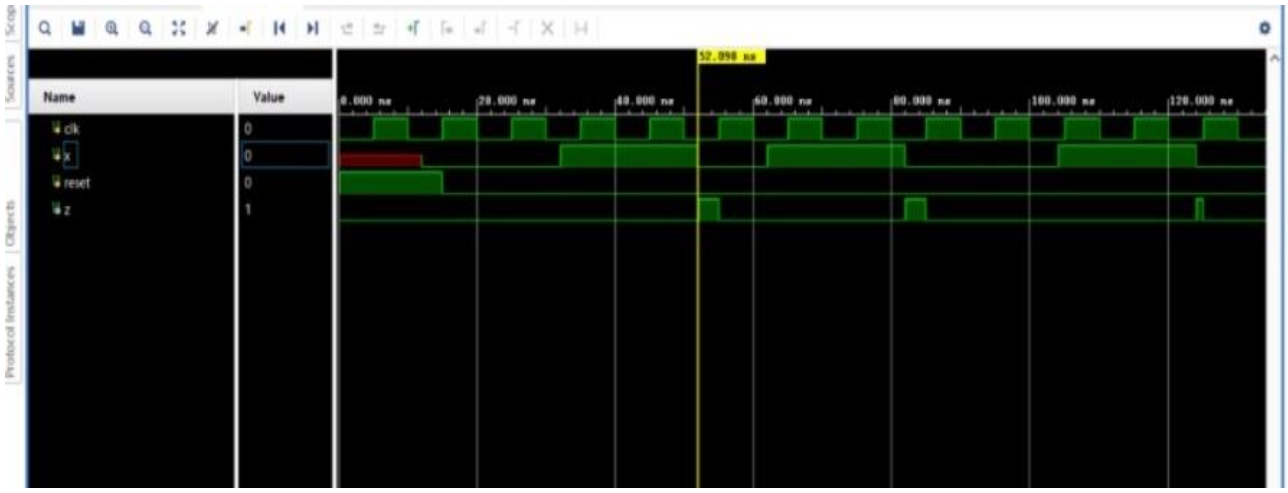
```
module detector(
  input x,
  input clk,
  input reset,
  output reg z
);
parameter s0=0,s1=1,s2=2,s3=3;
reg[0:1]Ps,Ns;
always@(posedge clk or posedge reset)
if(reset)
Ps<=s0;
else
Ps<=Ns;
always@(Ps,x)
case(Ps)
s0:begin
z= x? 0:0;
Ns= x? s0:s1;
end
s1:begin
z= x? 0:0;
Ns = x? s2:s1;
end
s2:begin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

```
endcase
Endmodule
```

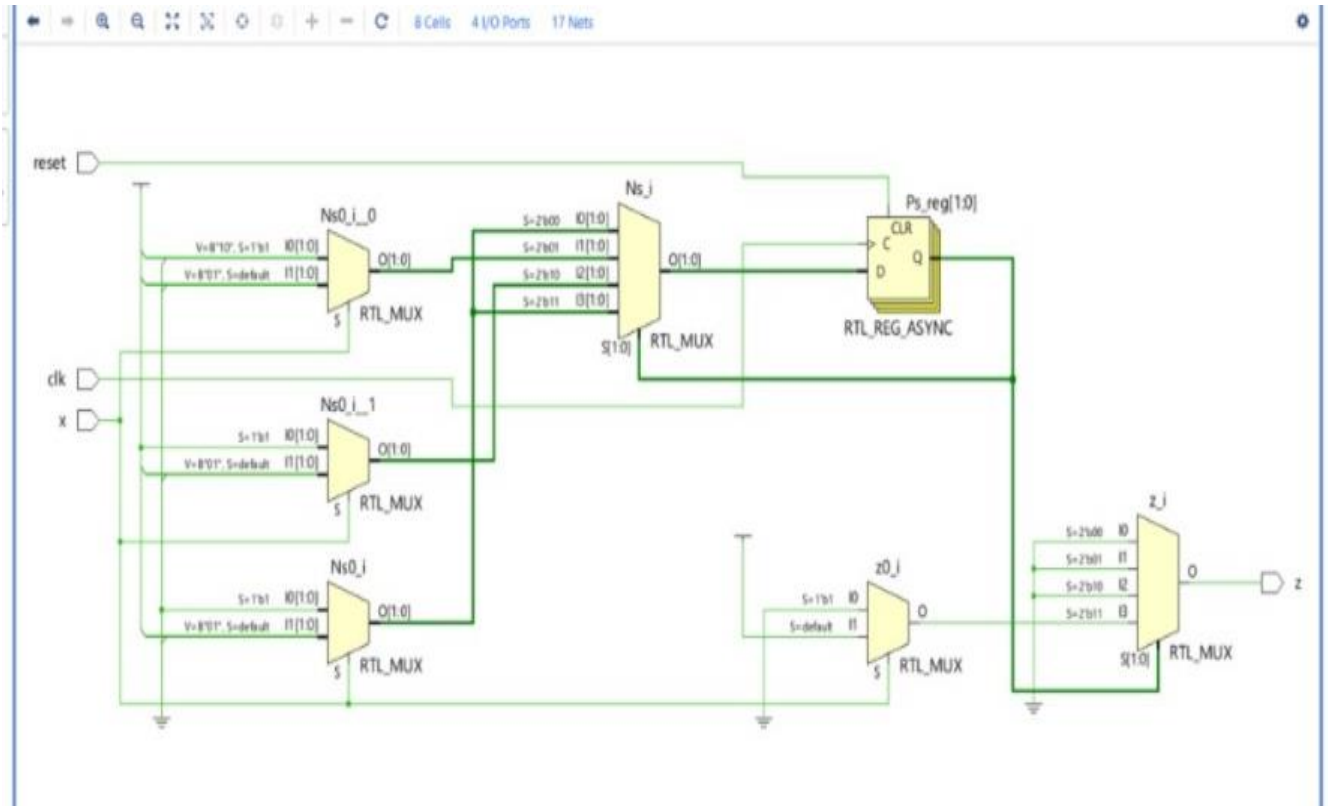
Test bench

```
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

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
```

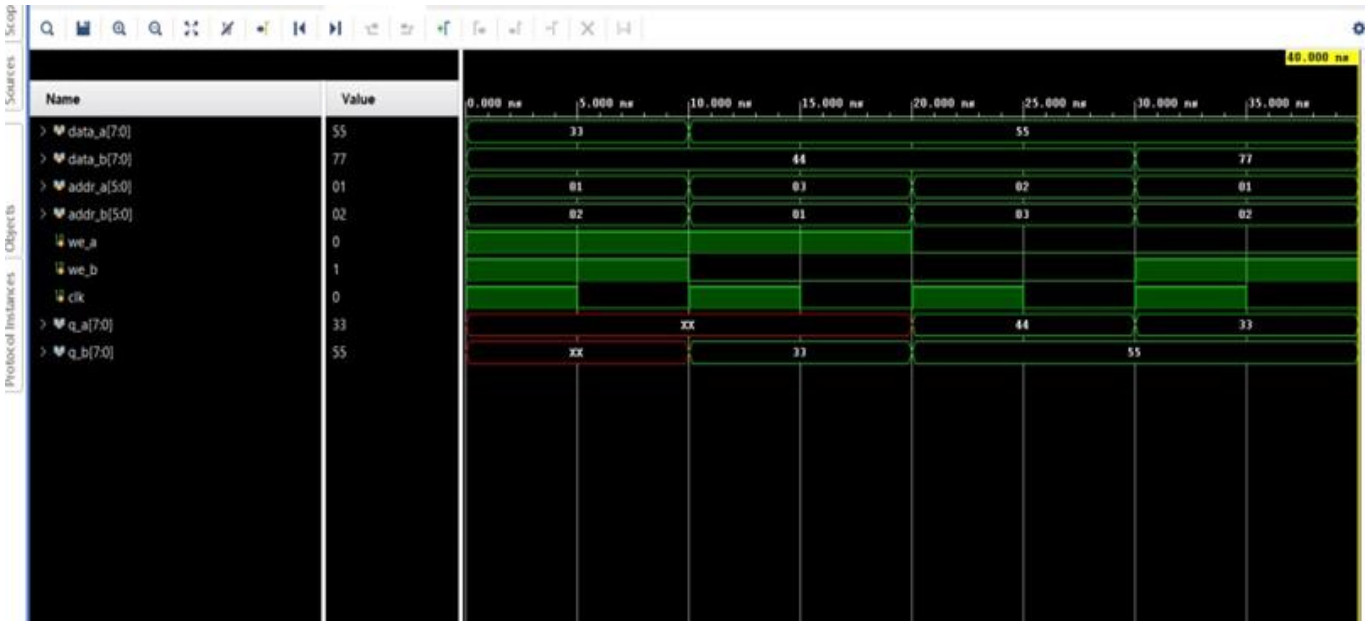
```
always @ (posedgeclk)
begin
 if(we_b)
 ram[addr_b] <= data_b; else
q_b<= ram[addr_b];
 end
endmodule
```

Testbench

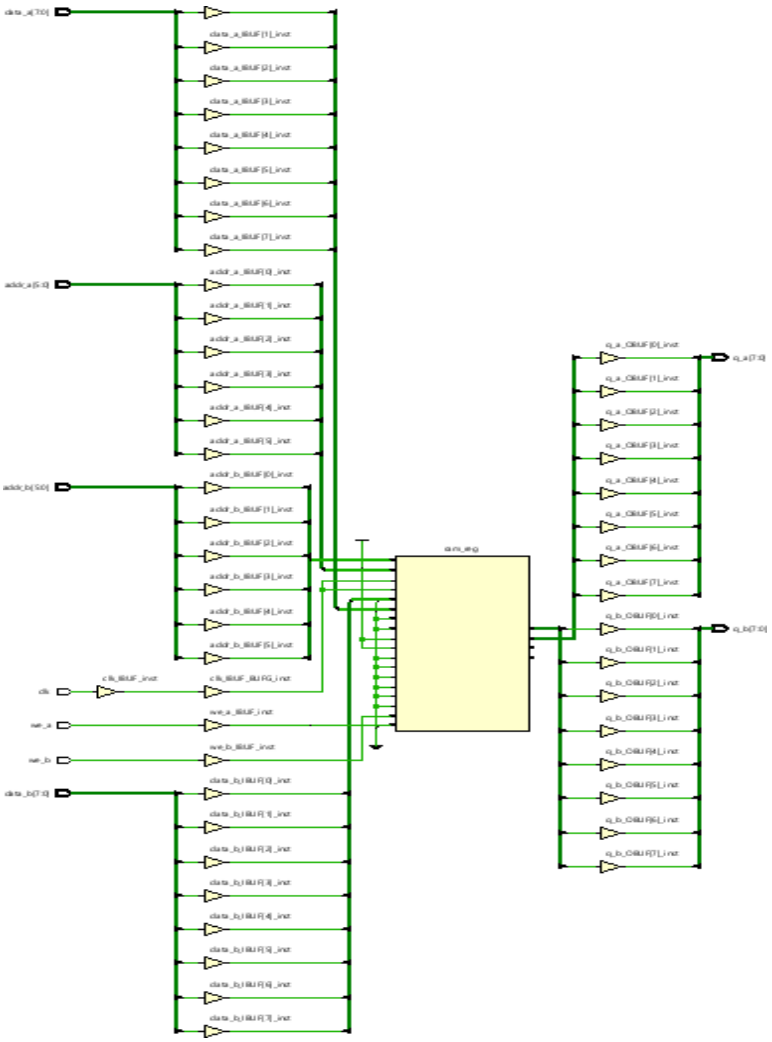
```
module dual_port_ram_tb;
 reg [7:0] data_a, data_b; //input data
 reg [5:0] addr_a, addr_b; //Port A and Port B address
 reg we_a, we_b; //write enable for Port A and Port B
 reg clk; //clk
 wire [7:0] q_a, q_b; //output data at Port A and Port B

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


Scop
Sources
Objects
Protocol Instances
Name
Value
data_a[7:0]
55
data_b[7:0]
77
addr_a[5:0]
01
addr_b[5:0]
02
we_a
0
we_b
1
clk
0
q_a[7:0]
33
q_b[7:0]
55
40.000 ns
0.000 ns
5.000 ns
10.000 ns
15.000 ns
20.000 ns
25.000 ns
30.000 ns
35.000 ns
33
55
44
77
01
03
02
01
02
01
03
02
xx
44
33
xx
33
55

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.

3 Stage ALU

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
```

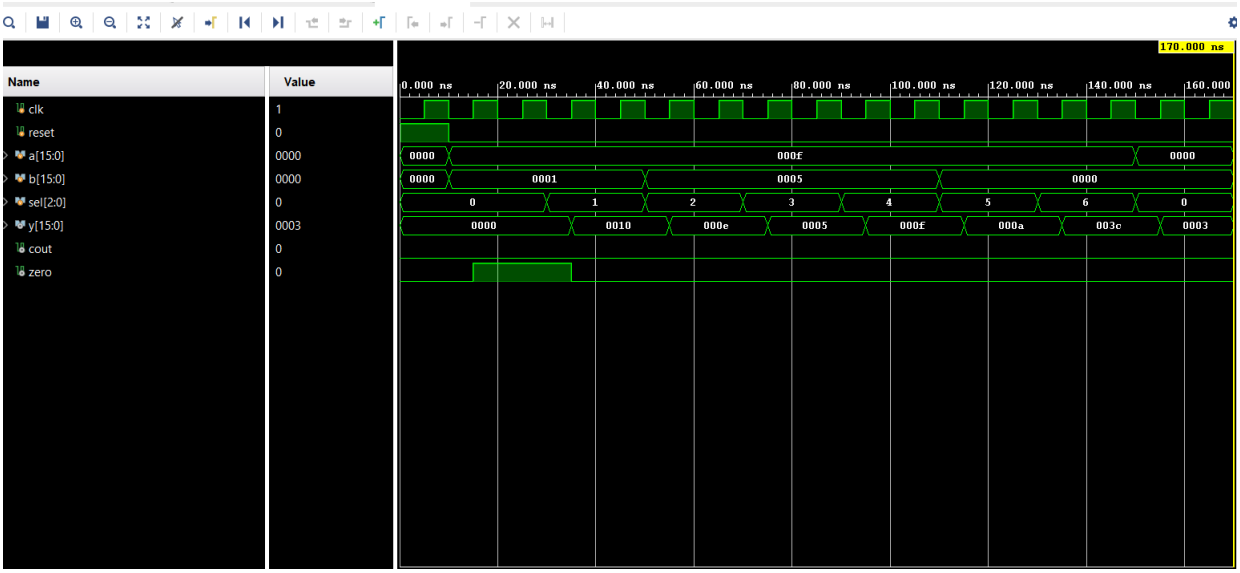
Testbench

```
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
```

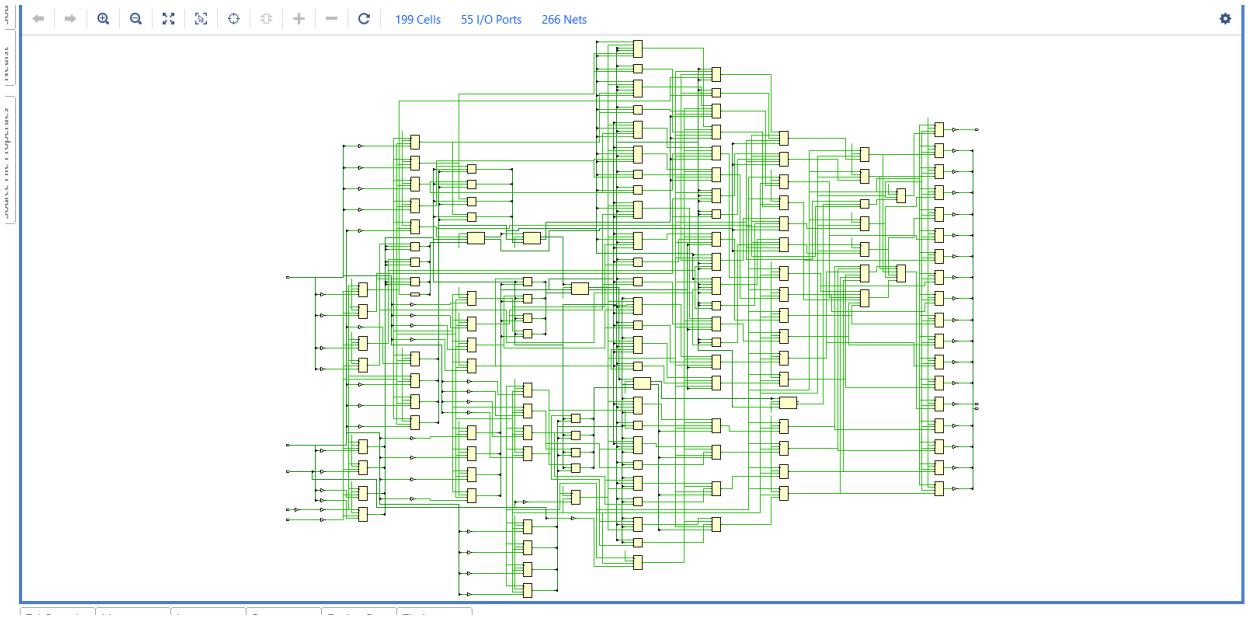
```
always begin
   #5 clk = ~clk;
end

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.