Dept. of Electronics and Electrical Communication Engineering
Indian Institute of Technology Kharagpur

VLSI LABORATORY (EC39004)

Experiment No: 5

Date of submission: 28 February 2024

Name: Samineni Rahul
Roll Number: 21EC30045

Objectives:

• Design a sample microprocessor with 6 registers and 5 operations. Input bus and Output bus need to be of 8-bit width. The instruction set consists of 8-bit instructions. The allowed operations are ADD, SUB, MOV, IN and OUT.

• Assume the 8 registers as A, B, C, D, E and F. Say R, R1 and R2 is used to represent one of the registers. The operations are defined as follows.

- ➢ ADD R: Add the value in R to value in A and update A to the obtained sum.
- ➢ MOV R1 R2: Copy the value of R2 into R1.
- ➢ IN R: Input an 8-bit number to register R.
- ➢ OUT R: Output the 8-bit number in register R.

• Assign 8-bit instructions to each of the above 5 operations as per your convenience and proceed with the design. Design using Verilog Hardware Description Language (HDL) behaviorally. Design appropriate testbench and observe and report the obtained waveforms. Show the output waveforms, RTL Schematic

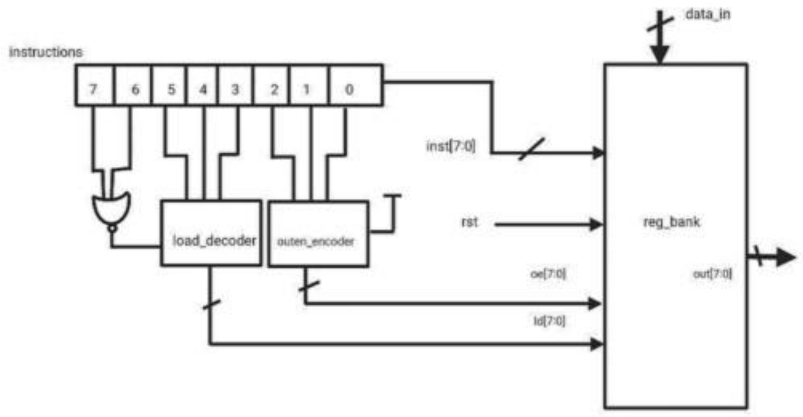


Figure: Main Architecture

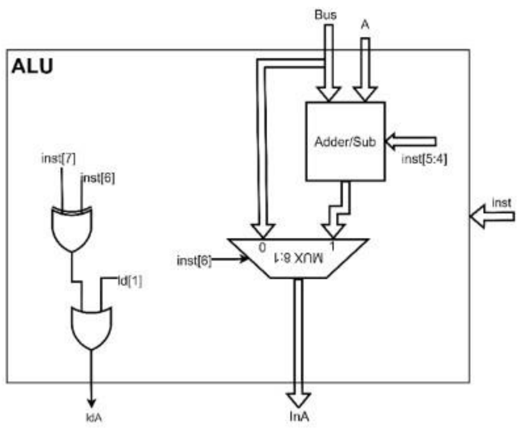


Figure: Architecture for ALU

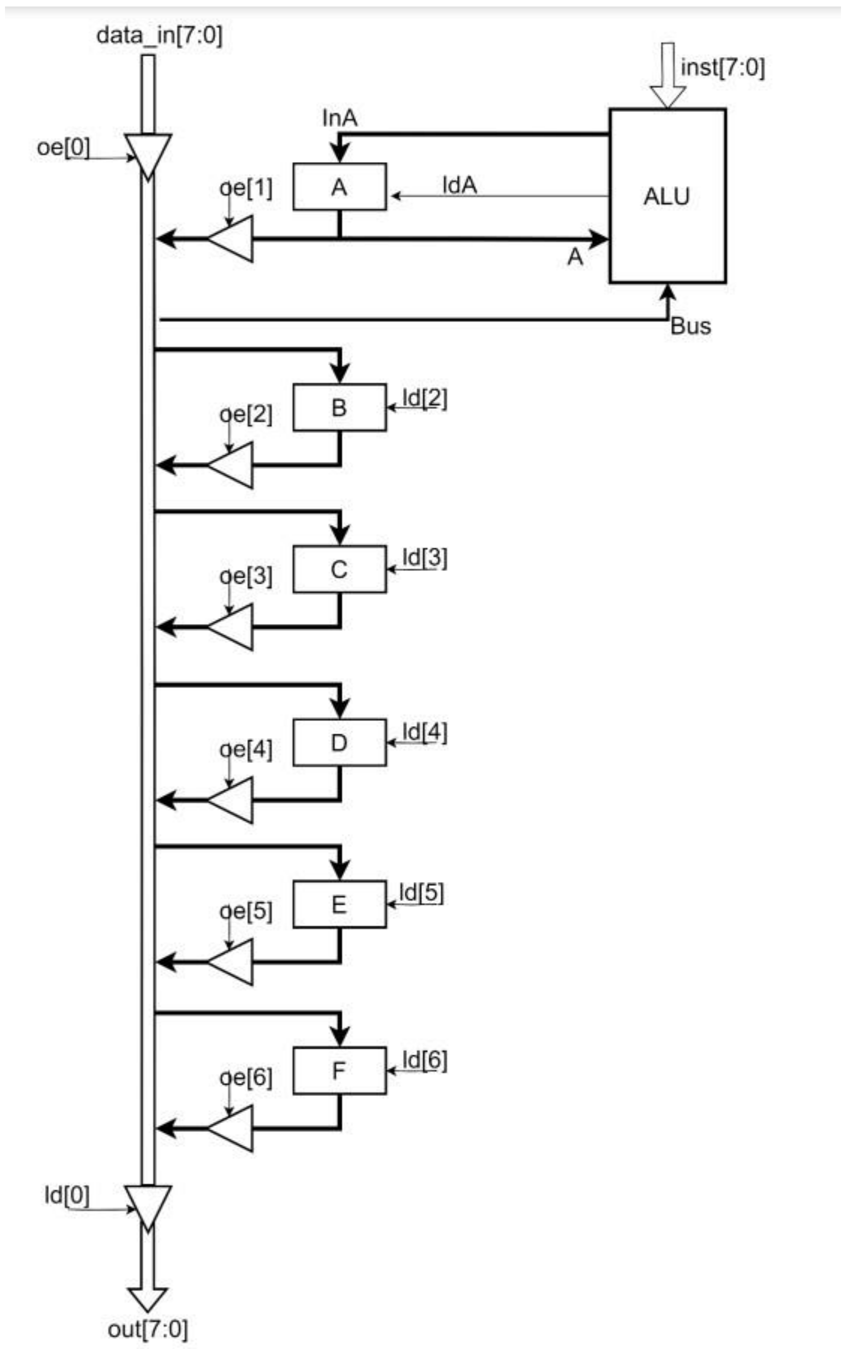
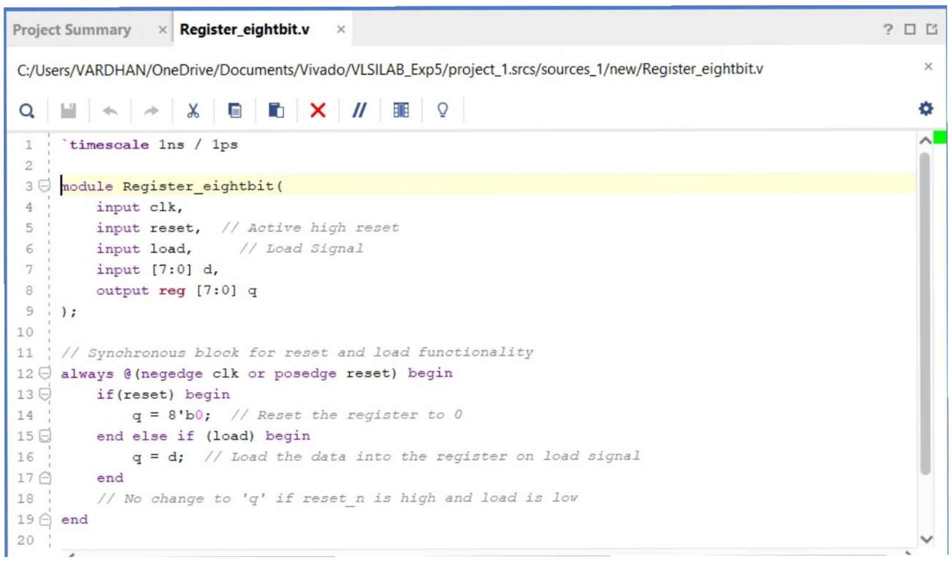


Figure: Architecture for Register bank

Design Code:

Implementation of 8-bit <u>Register</u>:


Project Summary
Register_eightbit.v
C:/Users/VARDHAN/OneDrive/Documents/Vivado/VLSILAB_Exp5/project_1.srcs/sources_1/new/Register_eightbit.v

```verilog
`timescale 1ns / 1ps

module Register_eightbit(
    input clk,
    input reset,  // Active high reset
    input load,     // Load Signal
    input [7:0] d,
    output reg [7:0] q
);

// Synchronous block for reset and load functionality
always @(negedge clk or posedge reset) begin
    if(reset) begin
        q = 8'b0;  // Reset the register to 0
    end else if (load) begin
        q = d;  // Load the data into the register on load signal
    end
    // No change to 'q' if reset_n is high and load is low
end
```

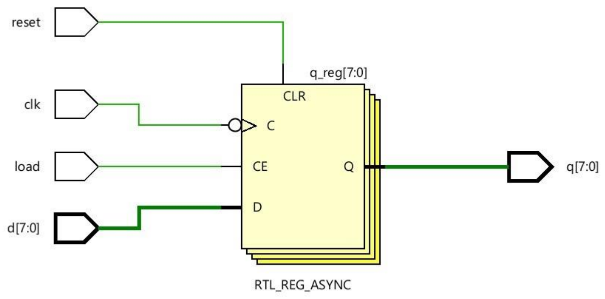
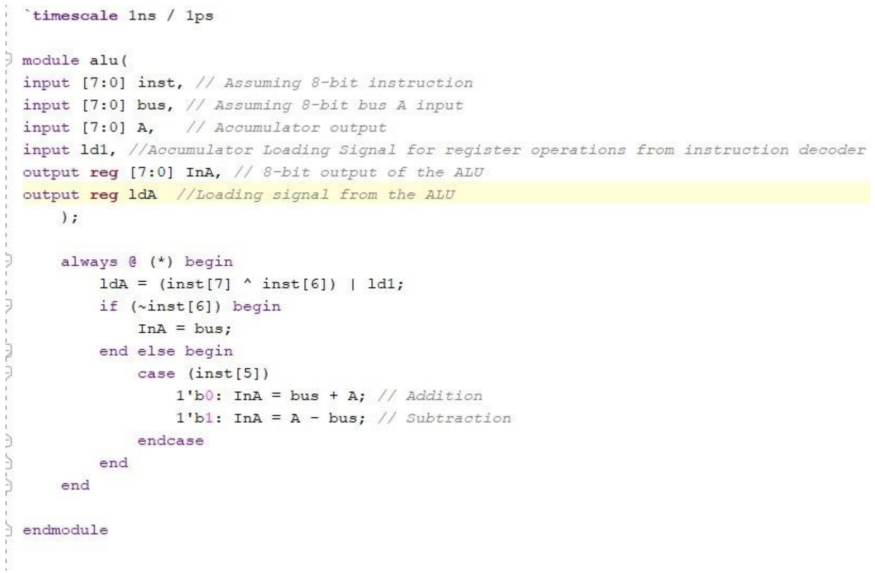


Figure: Schematic for 8-bit Register

Implementation of ALU:



```verilog
`timescale 1ns / 1ps

module alu(
input [7:0] inst, // Assuming 8-bit instruction
input [7:0] bus, // Assuming 8-bit bus A input
input [7:0] A,   // Accumulator output
input ld1, //Accumulator Loading Signal for register operations from instruction decoder
output reg [7:0] InA, // 8-bit output of the ALU
output reg ldA  //Loading signal from the ALU
    );

    always @ (*) begin
        ldA = (inst[7] ^ inst[6]) | ld1;
        if (~inst[6]) begin
            InA = bus;
        end else begin
            case (inst[5])
                1'b0: InA = bus + A; // Addition
                1'b1: InA = A - bus; // Subtraction
            endcase
        end
    end

endmodule
```

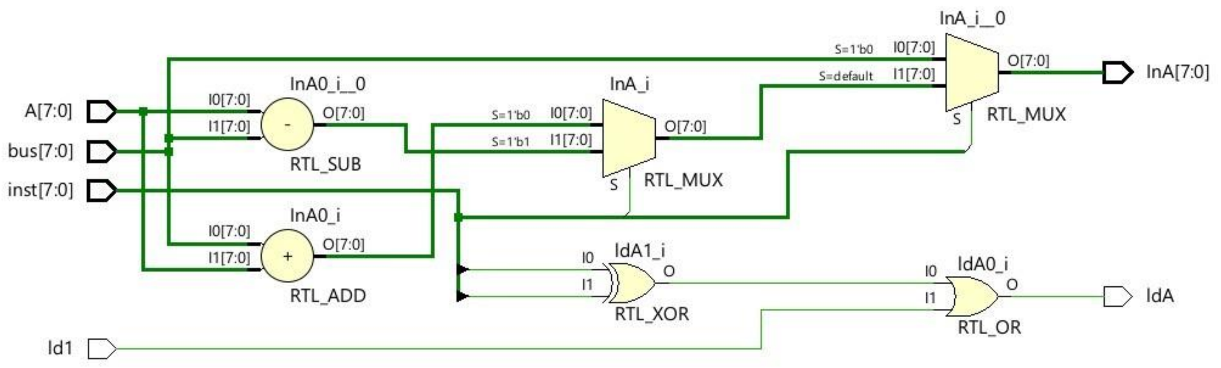
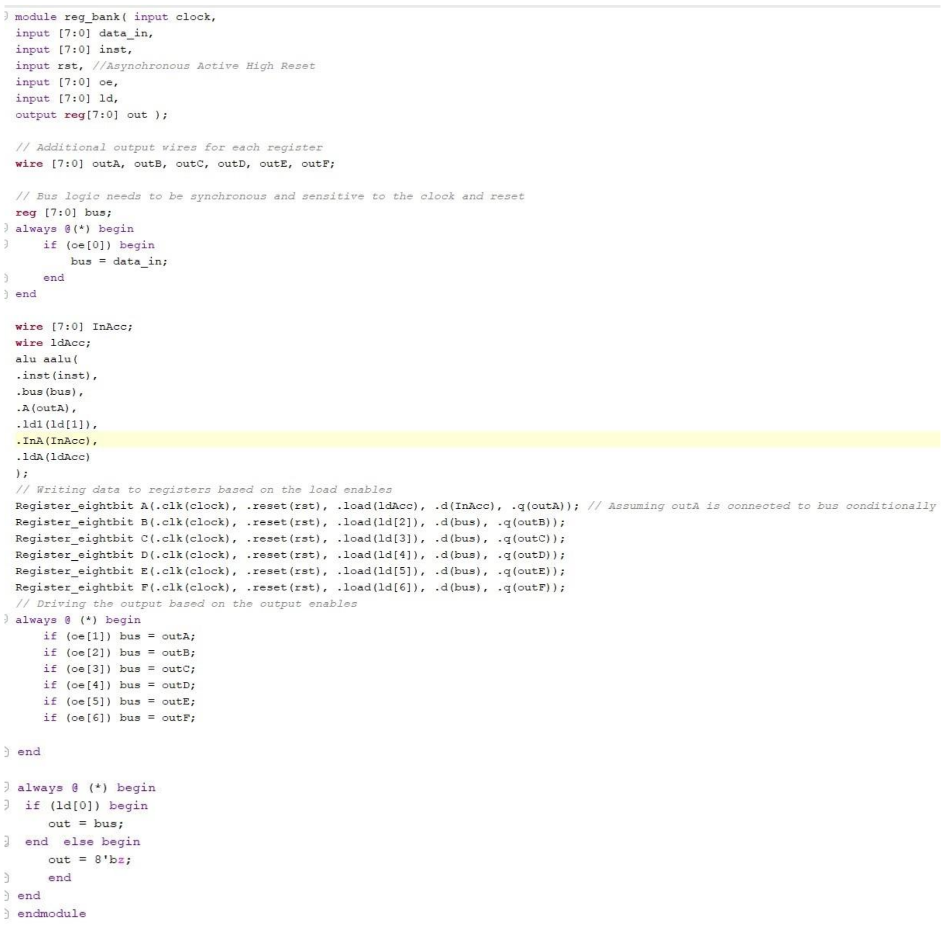


Figure: Schematic for ALU

Implementation of Register Bank:



```
module reg_bank( input clock,
input [7:0] data_in,
input [7:0] inst,
input rst, //Asynchronous Active High Reset
input [7:0] oe,
input [7:0] ld,
output reg[7:0] out );

// Additional output wires for each register
wire [7:0] outA, outB, outC, outD, outE, outF;

// Bus logic needs to be synchronous and sensitive to the clock and reset
reg [7:0] bus;
always @(*) begin
    if (oe[0]) begin
        bus = data_in;
    end
end

wire [7:0] InAcc;
wire ldAcc;
alu aalu(
.inst(inst),
.bus(bus),
.A(outA),
.ld1(ld[1]),
.InA(InAcc),
.ldA(ldAcc)
);
// Writing data to registers based on the load enables
Register_eightbit A(.clk(clock), .reset(rst), .load(ldAcc), .d(InAcc), .q(outA)); // Assuming outA is connected to bus conditionally
Register_eightbit B(.clk(clock), .reset(rst), .load(ld[2]), .d(bus), .q(outB));
Register_eightbit C(.clk(clock), .reset(rst), .load(ld[3]), .d(bus), .q(outC));
Register_eightbit D(.clk(clock), .reset(rst), .load(ld[4]), .d(bus), .q(outD));
Register_eightbit E(.clk(clock), .reset(rst), .load(ld[5]), .d(bus), .q(outE));
Register_eightbit F(.clk(clock), .reset(rst), .load(ld[6]), .d(bus), .q(outF));
// Driving the output based on the output enables
always @ (*) begin
    if (oe[1]) bus = outA;
    if (oe[2]) bus = outB;
    if (oe[3]) bus = outC;
    if (oe[4]) bus = outD;
    if (oe[5]) bus = outE;
    if (oe[6]) bus = outF;

 end

 always @ (*) begin
  if (ld[0]) begin
     out = bus;
  end  else begin
     out = 8'bz;
     end
 end
 endmodule
```

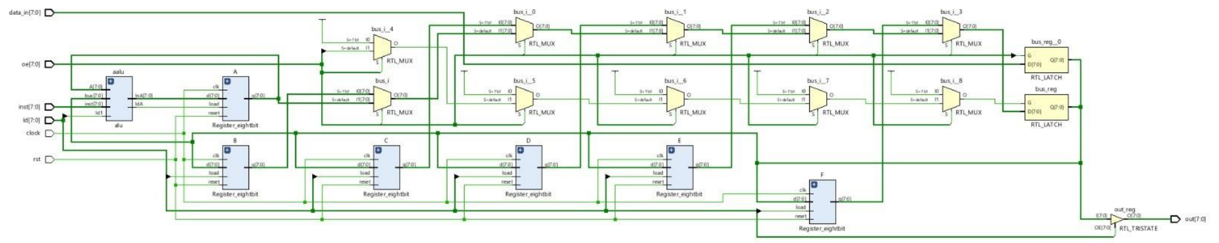
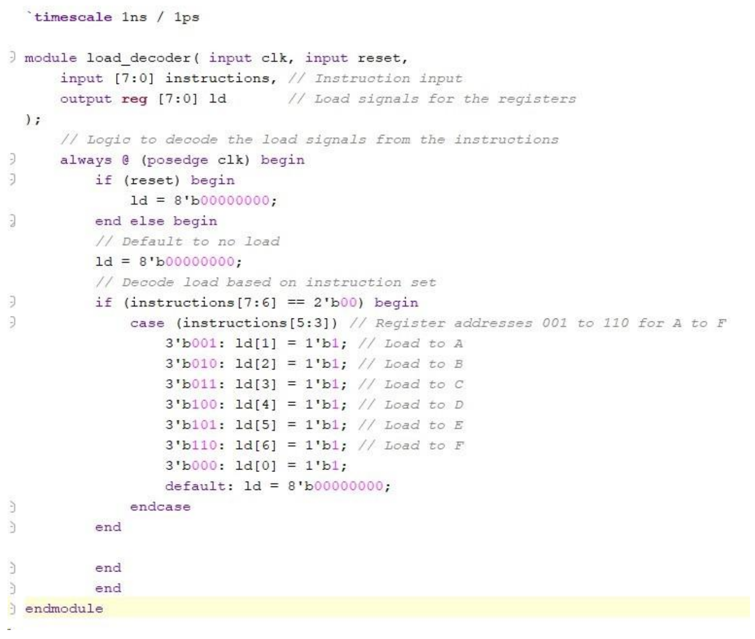


Figure: Schematic for Register Bank

Implementation of Load Decoder Signals:



```
`timescale 1ns / 1ps

module load_decoder( input clk, input reset,
    input [7:0] instructions, // Instruction input
    output reg [7:0] ld       // Load signals for the registers
);
    // Logic to decode the load signals from the instructions
    always @ (posedge clk) begin
        if (reset) begin
            ld = 8'b00000000;
        end else begin
        // Default to no load
        ld = 8'b00000000;
        // Decode load based on instruction set
        if (instructions[7:6] == 2'b00) begin
            case (instructions[5:3]) // Register addresses 001 to 110 for A to F
                3'b001: ld[1] = 1'b1; // Load to A
                3'b010: ld[2] = 1'b1; // Load to B
                3'b011: ld[3] = 1'b1; // Load to C
                3'b100: ld[4] = 1'b1; // Load to D
                3'b101: ld[5] = 1'b1; // Load to E
                3'b110: ld[6] = 1'b1; // Load to F
                3'b000: ld[0] = 1'b1;
                default: ld = 8'b00000000;
            endcase
        end

        end
        end
endmodule
```

Implementation of Output Decoder Signals:

```
module output_encoder( input clk, input reset,
    input [7:0] instructions, // Instruction input
    output reg [7:0] oe       // Output enable signals for the registers
);
     // Logic to decode the load signals from the instructions
    always @ (posedge clk) begin
        if (reset) begin
            oe = 8'b00000000;
        end else begin
        // Default to no load
        oe = 8'b00000000;
        // Decode load based on instruction set
        if (instructions[7:6] == 2'b00 || instructions[7:6] == 2'b01) begin
            case (instructions[2:0]) // Register addresses 001 to 110 for A to F
                3'b001: oe[1] = 1'b1;
                3'b010: oe[2] = 1'b1;
                3'b011: oe[3] = 1'b1;
                3'b100: oe[4] = 1'b1;
                3'b101: oe[5] = 1'b1;
                3'b110: oe[6] = 1'b1;
                3'b000: oe[0] = 1'b1;
                default: oe = 8'b00000000;
            endcase
        end
        end
        end
endmodule
```

Implementation of Instruction Decoder:

```
module inst_decoder( input clk, input reset,
    input [7:0] instructions,  // Instruction input
    output [7:0] ld,           // Load signals for the registers
    output [7:0] oe            // Output enable signals for the registers
);
    // Instantiate the load decoder
    load_decoder ld_dec ( .clk(clk), .reset(reset),
        .instructions(instructions),
        .ld(ld)
    );

    // Instantiate the output encoder
    output_encoder oe_enc (  .clk(clk), .reset(reset),
        .instructions(instructions),
        .oe(oe)
    );

endmodule
```

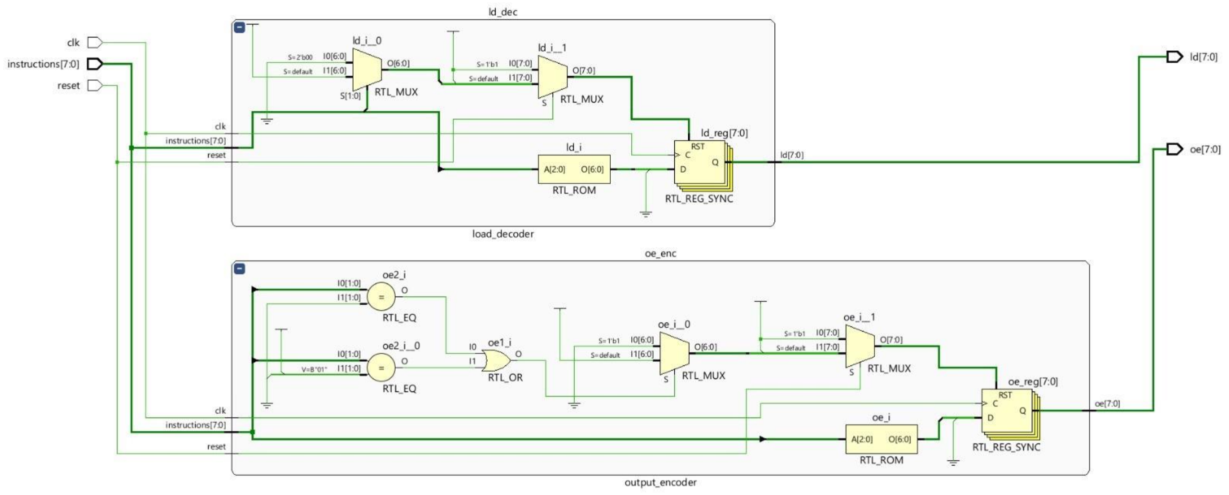
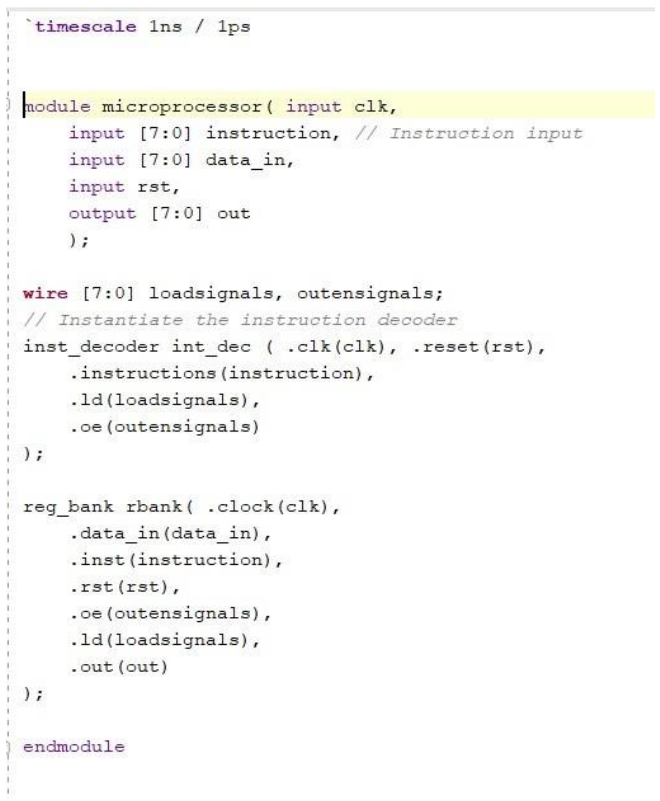


Figure: Schematic for instruction decoder

Implementation of Final Architecture for microprocessor:


```
`timescale 1ns / 1ps


module microprocessor( input clk,
    input [7:0] instruction, // Instruction input
    input [7:0] data_in,
    input rst,
    output [7:0] out
    );

wire [7:0] loadsignals, outensignals;
// Instantiate the instruction decoder
inst_decoder int_dec ( .clk(clk), .reset(rst),
    .instructions(instruction),
    .ld(loadsignals),
    .oe(outensignals)
);

reg_bank rbank( .clock(clk),
    .data_in(data_in),
    .inst(instruction),
    .rst(rst),
    .oe(outensignals),
    .ld(loadsignals),
    .out(out)
);

endmodule
```

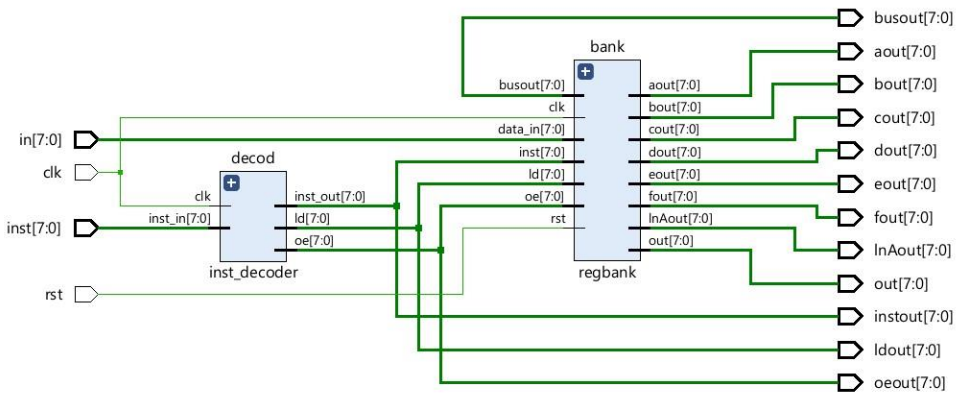
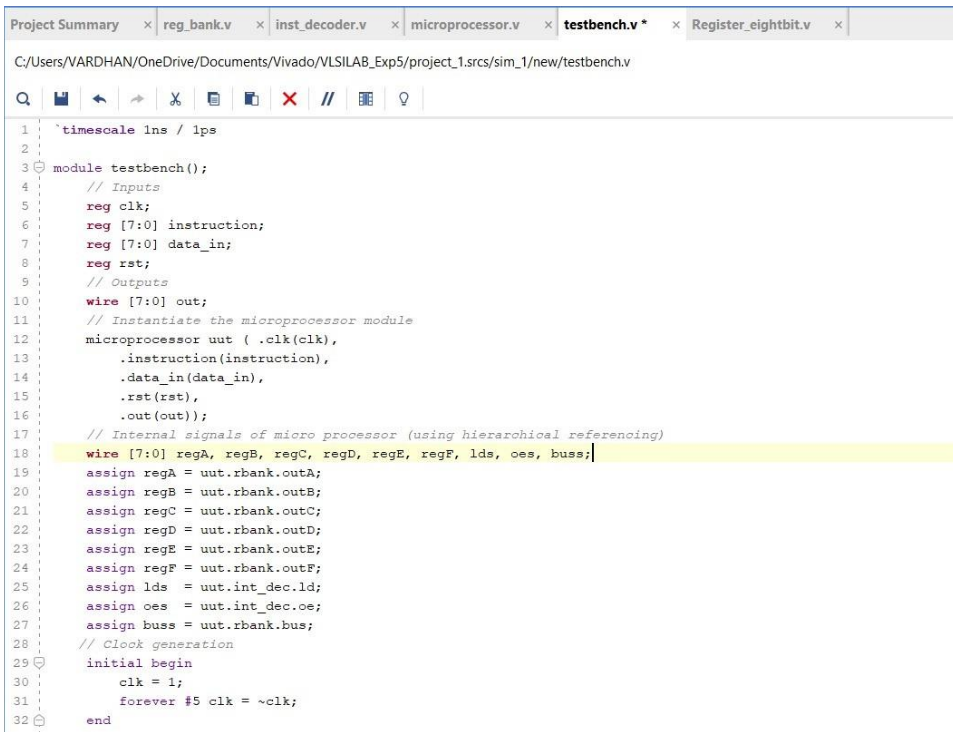


Figure: Schematic for Microprocessor

Testbench Code:

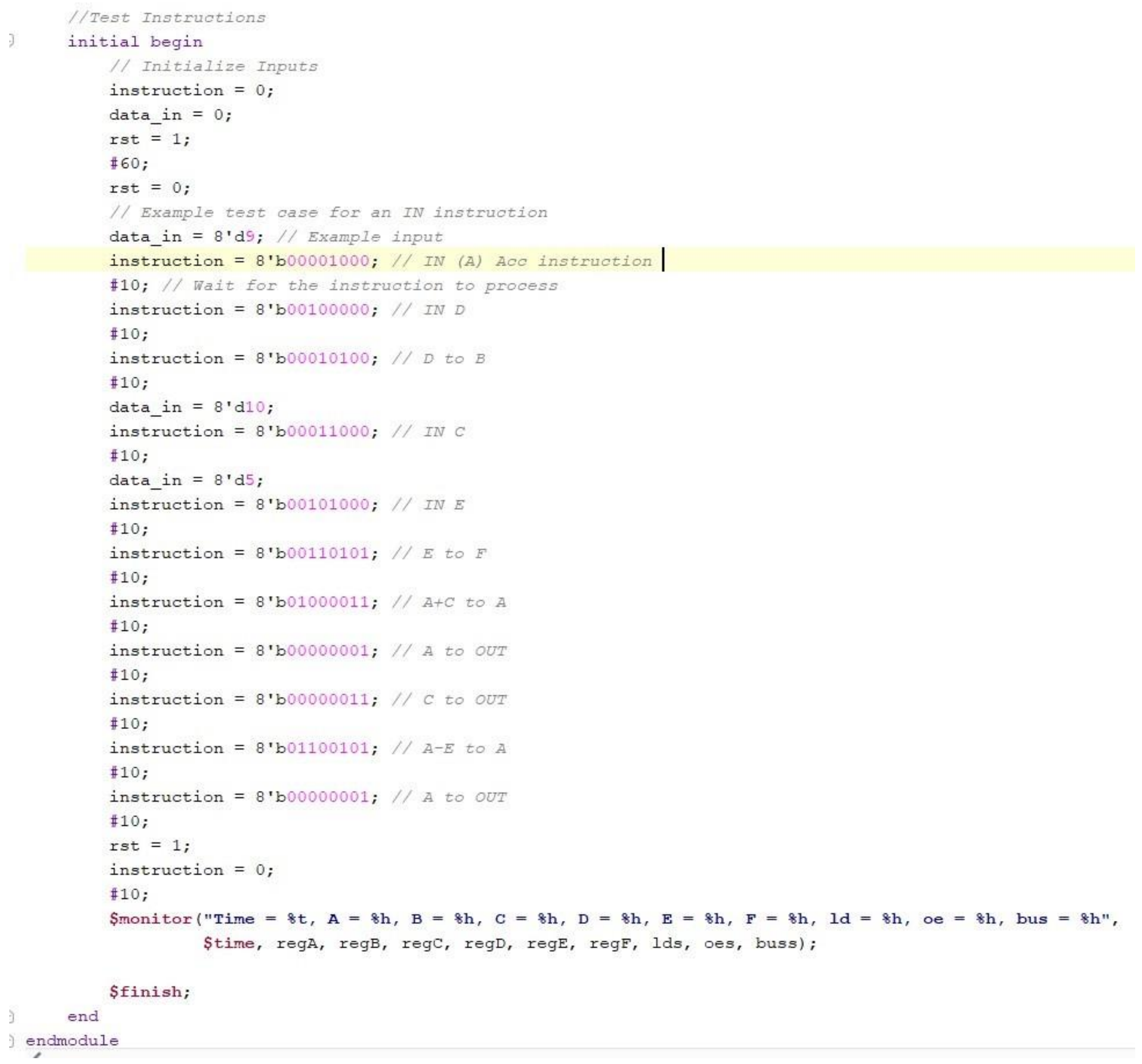
Code for Testbench:


Project Summary × reg_bank.v × inst_decoder.v × microprocessor.v × testbench.v * × Register_eightbit.v ×

C:/Users/VARDHAN/OneDrive/Documents/Vivado/VLSILAB_Exp5/project_1.srcs/sim_1/new/testbench.v

```
`timescale 1ns / 1ps

module testbench();
    // Inputs
    reg clk;
    reg [7:0] instruction;
    reg [7:0] data_in;
    reg rst;
    // Outputs
    wire [7:0] out;
    // Instantiate the microprocessor module
    microprocessor uut ( .clk(clk),
        .instruction(instruction),
        .data_in(data_in),
        .rst(rst),
        .out(out));
    // Internal signals of micro processor (using hierarchical referencing)
    wire [7:0] regA, regB, regC, regD, regE, regF, lds, oes, buss;
    assign regA = uut.rbank.outA;
    assign regB = uut.rbank.outB;
    assign regC = uut.rbank.outC;
    assign regD = uut.rbank.outD;
    assign regE = uut.rbank.outE;
    assign regF = uut.rbank.outF;
    assign lds  = uut.int_dec.ld;
    assign oes  = uut.int_dec.oe;
    assign buss = uut.rbank.bus;
   // Clock generation
    initial begin
        clk = 1;
        forever #5 clk = ~clk;
    end
```



```
    //Test Instructions
    initial begin
        // Initialize Inputs
        instruction = 0;
        data_in = 0;
        rst = 1;
        #60;
        rst = 0;
        // Example test case for an IN instruction
        data_in = 8'd9; // Example input
        instruction = 8'b00001000; // IN (A) Acc instruction
        #10; // Wait for the instruction to process
        instruction = 8'b00100000; // IN D
        #10;
        instruction = 8'b00010100; // D to B
        #10;
        data_in = 8'd10;
        instruction = 8'b00011000; // IN C
        #10;
        data_in = 8'd5;
        instruction = 8'b00101000; // IN E
        #10;
        instruction = 8'b00110101; // E to F
        #10;
        instruction = 8'b01000011; // A+C to A
        #10;
        instruction = 8'b00000001; // A to OUT
        #10;
        instruction = 8'b00000011; // C to OUT
        #10;
        instruction = 8'b01100101; // A-E to A
        #10;
        instruction = 8'b00000001; // A to OUT
        #10;
        rst = 1;
        instruction = 0;
        #10;
        $monitor("Time = %t, A = %h, B = %h, C = %h, D = %h, E = %h, F = %h, ld = %h, oe = %h, bus = %h",
                 $time, regA, regB, regC, regD, regE, regF, lds, oes, buss);

        $finish;
    end
endmodule
```

Observations:

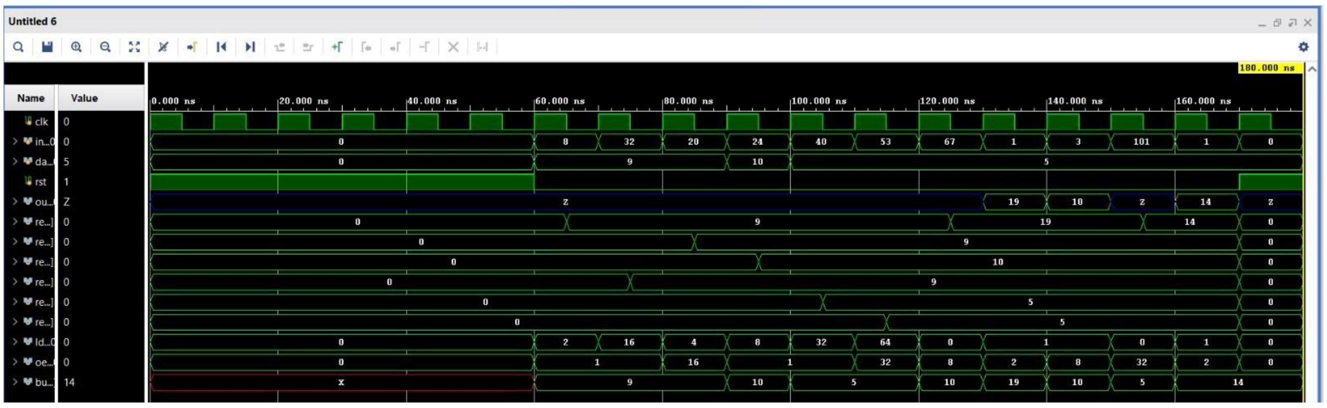


Figure: Output Waveform

Results:

Register Addresses:

A – 001

B – 010

C – 011

D – 100

E – 101

F – 110*Instructions Given in Test Bench*:

##data_in = 9##

IN A

IN D

MOV D B