# Final Project FPGA

Course: EE 4953 FPGA and HDL

Instructor: Savithra Eratne

Submitted by: Javier Uribe

**Problem Statement:**

The goal of the project tis to design and implement a 16 nit reduced instruction set computer on the FPGA board. The processor has to be able to execute small intruictions using custom data paths and control logic.

The processor follows a 4 step process for each instruction

1. Fetch – read the instttructuon from instruction memory using the program counter
2. Decode – goes through the instruction to determine the op code and ooeprands
3. Execute- perform the ALU operation or mem access
4. Update PC- move on to the next instruction

A custom contr4oller generates the signals based on the op code and cycles these 4 steps. Data is stores in mem and registers and the results are shown on the display on the FPGA board.

The memory locations are 203, 204, and 205 and must be updated during program execution and then displayed as decimal on the display.

**Approach:**

To make a 16 bit RISC processor I used a top down design where each block in the datapath was written as its own module so I could easily debug and go back and fix. I also created a top module called rsic16_top where I integrated all previous modules together.

Several modules in this project were from earlier labs to complete the RISC Processor like the PC, IR, ALU, IM, and the digital display converter.

**Problems Encountered:**

During the development initially my 7 segment display was super unstable flicking and showing incorrect values, Fixed this by changing the multiplexing logic. Another issue was during the synthesis when Vidado threw errors on modules being undefined or unassigned, these were all fixed by verifying one by one the assignments to my board, since I was not using the same capital letters and spelling errors everywhere. Overall just general errors I have every lab that I had to fix, most being just spelling.

Here's the breakdown and code for each module

1. Instruction Memory
    a. It has the 10 instructions that perform the operations
    b. All the instructions are 16 bits

c. Instructions include LW, SW, ADD, SUB, LI, SRA, XOR

```
module instr_mem (
    input  wire [15:0] addr,
    output reg  [15:0] instr
);
    always @(*) begin
        case (addr)
            16'd0:  instr = 16'b1001_0101_1100_1001; // LW
            16'd1:  instr = 16'b1001_0110_1100_1010; // LW
            16'd2:  instr = 16'b0000_0111_0101_0110; // ADD
            16'd3:  instr = 16'b1010_0111_1100_1011; // SW
            16'd4:  instr = 16'b1000_1000_1111_1010; // LI
            16'd5:  instr = 16'b0001_0100_1000_0101; // SUB
            16'd6:  instr = 16'b1010_0100_1100_1100; // SW
            16'd7:  instr = 16'b0111_0011_0111_0000; // SRA
            16'd8:  instr = 16'b0100_0010_0011_0100; // XOR
            16'd9:  instr = 16'b1010_0010_1100_1101; // SW
            default: instr = 16'b0000_0000_0000_0000; //default case
        endcase
    end
```

2. Program Counter
    a. 16 bit reg that keeps track of the instruction address
    b. Auto increments by 1 after the instruction

```
module pc (
    input wire clk,
    input wire rst,
    input wire load,
    input wire [15:0] pc_in,
    output reg [15:0] pc_out
);
    always @(posedge clk or posedge rst) begin
        if (rst)
            pc_out <= 16'b0;
        else if (load)
            pc_out <= pc_in;
    end
endmodule
```

3. Instruction Register
    a. Stores the current instruction from the memory
    b. Used to decode opcode and operands

```
module ir (
    input  wire        clk,
    input  wire        rst,
    input  wire        load,
    input  wire [15:0] instr_in,
    output reg  [15:0] instr_out
);
    always @(posedge clk or posedge rst) begin
        if (rst)
            instr_out <= 16'b0;
        else if (load)
            instr_out <= instr_in;
    end
endmodule
```

4. Register File
    a. Contains 16 bit registers

b. Can read two regisrters and write to them in a cycle

```
module reg_file (
    input  wire        clk,
    input  wire        rst,
    input  wire        reg_write,
    input  wire [3:0]  read_reg1,     // Rs
    input  wire [3:0]  read_reg2,     // Rt
    input  wire [3:0]  write_reg,     // Rd
    input  wire [15:0] write_data,
    output wire [15:0] read_data1,    // Rs data
    output wire [15:0] read_data2     // Rt data
);


    (* mark_debug = "true" *) reg [15:0] registers [15:0];
    integer i;

    initial begin
        for (i = 0; i < 16; i = i + 1)
            registers[i] = 16'd0;
    end


    assign read_data1 = registers[read_reg1];
    assign read_data2 = registers[read_reg2];


    always @(posedge clk or posedge rst) begin
        if (rst) begin
            for (i = 0; i < 16; i = i + 1)
                registers[i] <= 16'd0;
        end else if (reg_write) begin
            registers[write_reg] <= write_data;
            $display("Reg[%0d] <= %d at time %0t", write_reg, write_data, $time);
        end
    end

endmodule
```

5. ALU
   a. Executes the operations based on the control signal

```
module alu (
    input  wire [15:0] A,
    input  wire [15:0] B,
    input  wire [3:0]  alu_control,
    output reg  [15:0] result,
    output wire        zero
);

    always @(*) begin
        case (alu_control)
            4'b0000: result = A + B;          // ADD
            4'b0001: result = A - B;          // SUB
            4'b0010: result = A & B;          // AND
            4'b0011: result = A | B;          // OR
            4'b0100: result = A ^ B;          // XOR
            4'b0101: result = ~A;             // NOT
            4'b0110: result = A <<< 1;        // SLA
            4'b0111: result = $signed(A) >>> 1; // SRA
            default: result = 16'h0000;
        endcase
    end

    assign zero = (result == 16'b0);

endmodule
```

6. Data Memory

a. Used LW and SW
b. Memeory addresses 201 and 202 are initialized
c. Values stored at 203,204, and 205 are the final results

```
module data_mem (
    input  wire        clk,
    input  wire        mem_read,
    input  wire        mem_write,
    input  wire [15:0] addr,
    input  wire [15:0] write_data,
    output reg  [15:0] read_data,


    output wire [15:0] mem203,
    output wire [15:0] mem204,
    output wire [15:0] mem205
);

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

  initial begin
    memory[201] = 16'd12;
    memory[202] = 16'd4;
    memory[203] = 16'd1000;
    memory[204] = 16'd2345;
    memory[205] = 16'd4321;
end


    always @(*) begin
        if (mem_read)
            read_data = memory[addr];
        else
            read_data = 16'd0;
    end


    always @(posedge clk) begin
        if (mem_write)
            memory[addr] <= write_data;
    end

    assign mem203 = memory[203];
    assign mem204 = memory[204];
    assign mem205 = memory[205];

endmodule
```

7. Controller
    a. FSM with 4 states

b. Generates control signals to be able to control the data and modules

```
module controller (
    input  wire        clk,
    input  wire        rst,
    input  wire [3:0]  opcode,
    input  wire        zero_flag,

    output reg         pc_write,
    output reg         ir_load,
    output reg         reg_write,
    output reg         mem_read,
    output reg         mem_write,
    output reg  [3:0]  alu_control,
    output reg  [1:0]  alu_src_sel,
    output reg         mem_to_reg
);


reg [1:0] state, next_state;
parameter FETCH = 2'b00,
          DECODE = 2'b01,
          EXECUTE = 2'b10,
          WB_PC = 2'b11;



    always @(posedge clk or posedge rst) begin
        if (rst)
            state <= FETCH;
        else
            state <= next_state;
    end


    always @(*) begin
        case (state)
            FETCH:   next_state = DECODE;
            DECODE:  next_state = EXECUTE;
            EXECUTE: next_state = WB_PC;
            WB_PC:   next_state = FETCH;
            default: next_state = FETCH;
        endcase
    end

    always @(*) begin

        pc_write     = 0;
        ir_load      = 0;
        reg_write    = 0;
        mem_read     = 0;
        mem_write    = 0;
        alu_control  = 4'b0000;
        alu_src_sel  = 2'b00;
        mem_to_reg   = 0;

        case (state)
            FETCH: begin
                pc_write = 1;
                ir_load  = 1;
                mem_read = 1;
            end

            DECODE: begin
            end
            EXECUTE: begin
                case (opcode)
                    4'b0000: alu_control = 4'b0000; // ADD
                    4'b0001: alu_control = 4'b0001; // SUB
                    4'b0010: alu_control = 4'b0010; // AND
                    4'b0011: alu_control = 4'b0011; // OR
                    4'b0100: alu_control = 4'b0100; // XOR
                    4'b0101: alu_control = 4'b0101; // NOT
                    4'b0110: alu_control = 4'b0110; // SLA
                    4'b0111: alu_control = 4'b0111; // SRA
                    4'b1000: alu_control = 4'b0000; // LI
                    4'b1001: begin
                        alu_control = 4'b0000;
                        mem_read    = 1;
                        alu_src_sel = 2'b01;
                    end
                    4'b1010: begin
                        alu_control = 4'b0000;
                        mem_write   = 1;
                        alu_src_sel = 2'b01;
                    end
                    default: alu_control = 4'b0000;
                endcase
            end

            WB_PC: begin
                pc_write = 1;
                case (opcode)
                    4'b0000, 4'b0001, 4'b0010, 4'b0011,
                    4'b0100, 4'b0101, 4'b0110, 4'b0111: reg_write = 1;
                    4'b1000: begin
                        reg_write   = 1;
                        alu_src_sel = 2'b10;
                    end
                    4'b1001: begin
                        reg_write   = 1;
                        mem_to_reg  = 1;
                    end
                    default: ;
                endcase
            end
        endcase
    end

endmodule
```

8. Display System
    a. From previous labs I reused code and was able to implekent it again
    b. Bin 16 to bcd concverts 16 bit binary to BCD for display
    c. Seg driver drives the 7 segment display to 8 digits

d. Controlled by switches 1 and 0 to switch between the memoery outputs

```
module seven_seg_display(
    input clk,
    input [15:0] value,
    output reg [6:0] seg,
    output reg [3:0] an
);
    reg [3:0] digit;
    reg [15:0] value_reg;
    reg [1:0] digit_idx = 0;
    reg [19:0] counter = 0;

    always @(posedge clk) begin
        counter <= counter + 1;
        if (counter == 0) begin
            digit_idx <= digit_idx + 1;
        end
    end

    always @(*) begin
        value_reg = value;
        case (digit_idx)
            2'b00: begin
                digit = value_reg[3:0];
                an = 4'b1110;
            end
            2'b01: begin
                digit = value_reg[7:4];
                an = 4'b1101;
            end
            2'b10: begin
                digit = value_reg[11:8];
                an = 4'b1011;
            end
            2'b11: begin
                digit = value_reg[15:12];
                an = 4'b0111;
            end
        endcase

        case (digit)
            4'h0: seg = 7'b1000000;
            4'h1: seg = 7'b1111001;
            4'h2: seg = 7'b0100100;
            4'h3: seg = 7'b0110000;
            4'h4: seg = 7'b0011001;
            4'h5: seg = 7'b0010010;
            4'h6: seg = 7'b0000010;
            4'h7: seg = 7'b1111000;
            4'h8: seg = 7'b0000000;
            4'h9: seg = 7'b0010000;
            4'hA: seg = 7'b0001000;
            4'hB: seg = 7'b0000011;
            4'hC: seg = 7'b1000110;
            4'hD: seg = 7'b0100001;
            4'hE: seg = 7'b0000110;
            4'hF: seg = 7'b0001110;
            default: seg = 7'b1111111;
        endcase
    end
endmodule
```

```
module bin16_to_bcd5 (
    input wire clk,
    input wire [15:0] bin,
    output reg [3:0] d0, d1, d2, d3, d4
);
    integer i;
    reg [35:0] shift;

    always @(posedge clk) begin
        shift = {20'd0, bin};

        for (i = 0; i < 16; i = i + 1) begin
            if (shift[19:16] > 4) shift[19:16] = shift[19:16] + 3;
            if (shift[23:20] > 4) shift[23:20] = shift[23:20] + 3;
            if (shift[27:24] > 4) shift[27:24] = shift[27:24] + 3;
            if (shift[31:28] > 4) shift[31:28] = shift[31:28] + 3;
            if (shift[35:32] > 4) shift[35:32] = shift[35:32] + 3;
            shift = shift << 1;
        end

        {d4, d3, d2, d1, d0} = {
            shift[35:32], shift[31:28],
            shift[27:24], shift[23:20],
            shift[19:16]
        };
    end
endmodule
```

```
module seg_driver #(
    parameter N_DIG = 8
)(
    input wire clk,
    input wire [4*N_DIG-1:0] digits,
    output reg [7:0] an,
    output reg [7:0] seg
);
    reg [2:0] digit_idx = 0;
    reg [19:0] clkdiv = 0;
    wire [3:0] curr_digit;

    assign curr_digit = digits >> (digit_idx * 4);

    always @(posedge clk) begin
        clkdiv <= clkdiv + 1;
        if (clkdiv[15:0] == 16'd0)
            digit_idx <= digit_idx + 1;
    end


    always @(*) begin
        case (digit_idx)
            3'd0: an = 8'b11111110;
            3'd1: an = 8'b11111101;
            3'd2: an = 8'b11111011;
            3'd3: an = 8'b11110111;
            3'd4: an = 8'b11101111;
            3'd5: an = 8'b11011111;
            3'd6: an = 8'b10111111;
            3'd7: an = 8'b01111111;
            default: an = 8'b11111111;
        endcase

        case (curr_digit[3:0])
            4'h0: seg = 8'b11000000;
            4'h1: seg = 8'b11111001;
            4'h2: seg = 8'b10100100;
            4'h3: seg = 8'b10110000;
            4'h4: seg = 8'b10011001;
            4'h5: seg = 8'b10010010;
            4'h6: seg = 8'b10000010;
            4'h7: seg = 8'b11111000;
            4'h8: seg = 8'b10000000;
            4'h9: seg = 8'b10010000;
            4'hA: seg = 8'b10001000;
            4'hB: seg = 8'b10000011;
            4'hC: seg = 8'b11000110;
            4'hD: seg = 8'b10100001;
            4'hE: seg = 8'b10000110;
            4'hF: seg = 8'b10001110;
            default: seg = 8'b11111111;
        endcase
    end
endmodule
```

9. Top Module
    a. Integrates everything together

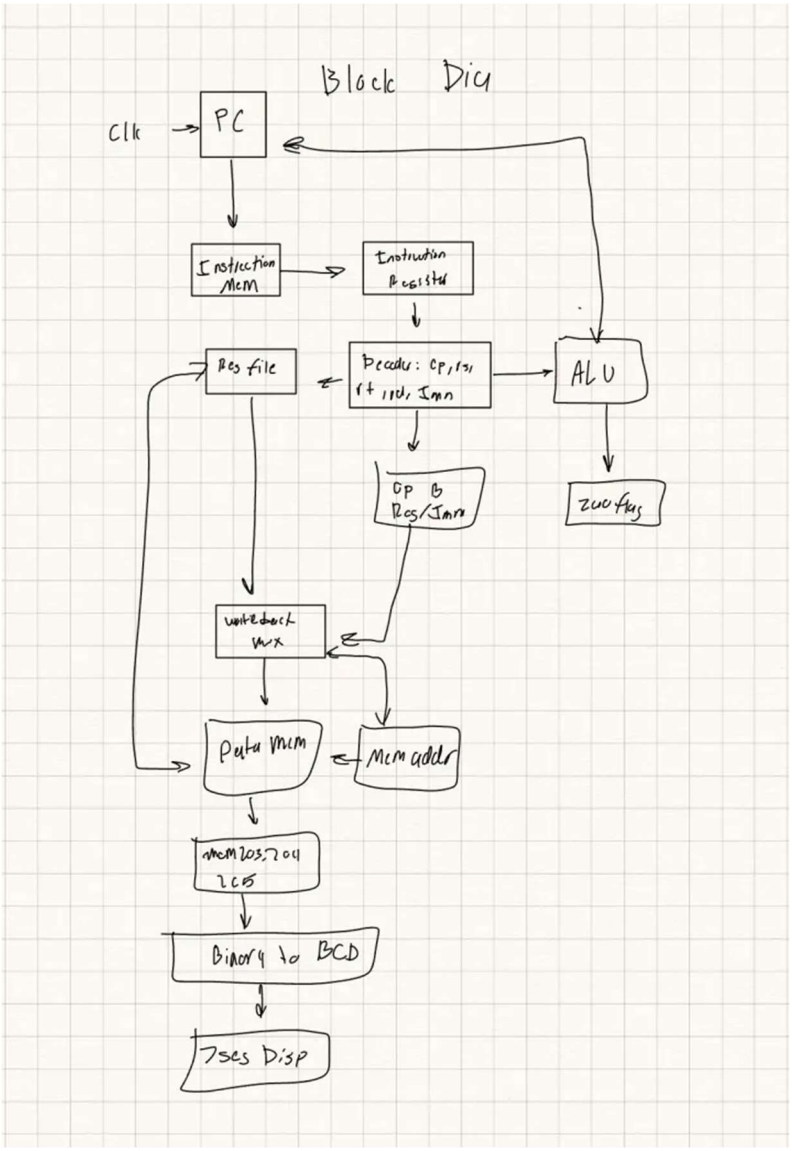
b. Drives the 7-segment display

c. Accepts switch input and displays results on display

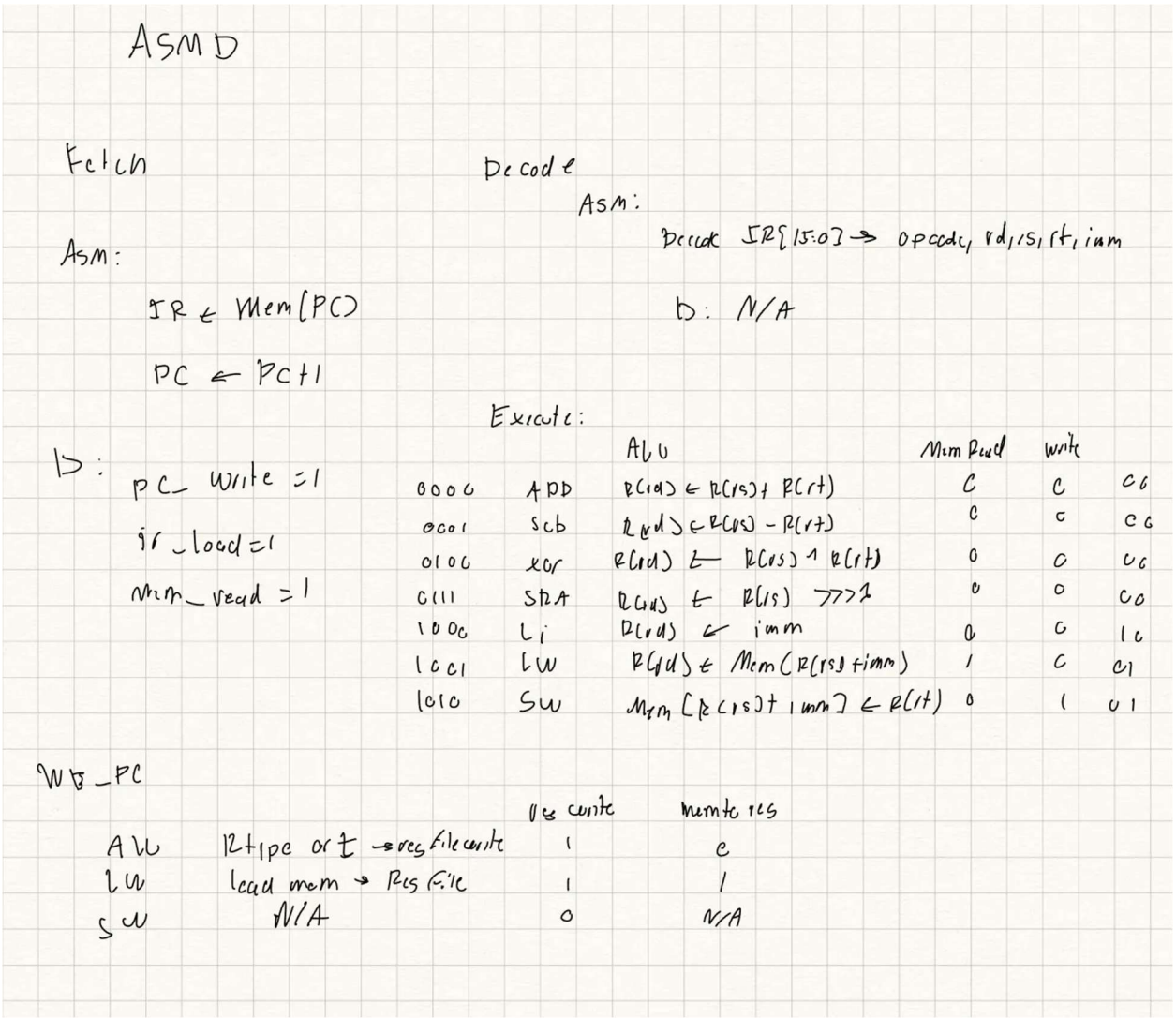
```
module risc16_top (
    input  wire        CLK100MHZ,
    input  wire        rst,
    input  wire [15:0] SW,
    output wire [7:0]  seg,
    output wire [7:0]  an,
    output wire [15:0] LED
);

    wire [15:0] pc_out;
    wire [15:0] pc_next;
    wire [15:0] instr;
    wire [15:0] ir_out;
    wire [3:0]  opcode;
    wire [3:0]  rs, rt, rd;
    wire [7:0]  imm;
    wire [15:0] reg_data1, reg_data2, write_data;
    wire [15:0] alu_result;
    wire [15:0] mem_data;
    wire [15:0] mem203, mem204, mem205;
    wire        zero;

    wire        pc_write, ir_load, reg_write;
    wire        mem_read, mem_write, mem_to_reg;
    wire [3:0]  alu_control;
    wire [1:0]  alu_src_sel;


    assign opcode = ir_out[15:12];
    assign rd     = ir_out[11:8];
    assign rs     = ir_out[7:4];
    assign rt     = ir_out[3:0];
    assign imm    = ir_out[7:0];


    reg [15:0] operand_b;
    always @(*) begin
        case (alu_src_sel)
            2'b00: operand_b = reg_data2;
            2'b01: operand_b = {8'b0, imm};
            2'b10: operand_b = {8'b0, imm};
            default: operand_b = 16'd0;
        endcase
    end


    assign write_data = (mem_to_reg) ? mem_data : alu_result;


    assign pc_next = pc_out + 1;

    controller u_ctrl (
        .clk(CLK100MHZ),
        .rst(rst),
        .opcode(opcode),
        .zero_flag(zero),
        .pc_write(pc_write),
        .ir_load(ir_load),
        .reg_write(reg_write),
        .mem_read(mem_read),
        .mem_write(mem_write),
        .alu_control(alu_control),
        .alu_src_sel(alu_src_sel),
        .mem_to_reg(mem_to_reg)
    );

    wire [3:0] d0, d1, d2, d3, d4;

    bin16_to_bcd5 u_bcd (
        .clk(CLK100MHZ),
        .bin(selected_value),
        .d0(d0), .d1(d1), .d2(d2), .d3(d3), .d4(d4)
    );

    seg_driver #(.N_DIG(8)) u_disp (
    .clk(CLK100MHZ),
    .digits({4'd0, 4'd0, 4'd0, d4, d3, d2, d1, d0}),

    .seg(seg),
    .an(an)
);

endmodule
```

```
reg [15:0] selected_value;
always @(*) begin
    case (SW[1:0])
        2'b00: selected_value = mem203;
        2'b01: selected_value = mem204;
        2'b10: selected_value = mem205;
        default: selected_value = 16'd0;
    endcase
end


pc u_pc (
    .clk(CLK100MHZ),
    .rst(rst),
    .load(pc_write),
    .pc_in(pc_next),
    .pc_out(pc_out)
);

instr_mem u_imem (
    .addr(pc_out),
    .instr(instr)
);

ir u_ir (
    .clk(CLK100MHZ),
    .rst(rst),
    .load(ir_load),
    .instr_in(instr),
    .instr_out(ir_out)
);

reg_file u_regfile (
    .clk(CLK100MHZ),
    .rst(rst),
    .reg_write(reg_write),
    .read_reg1(rs),
    .read_reg2(rt),
    .write_reg(rd),
    .write_data(write_data),
    .read_data1(reg_data1),
    .read_data2(reg_data2)
);

alu u_alu (
    .A(reg_data1),
    .B(operand_b),
    .alu_control(alu_control),
    .result(alu_result),
    .zero(zero)
);

data_mem u_dmem (
    .clk(CLK100MHZ),
    .mem_read(mem_read),
    .mem_write(mem_write),
    .addr(alu_result),
    .write_data(reg_data2),
    .read_data(mem_data),
    .mem203(mem203),
    .mem204(mem204),
    .mem205(mem205)
);
```

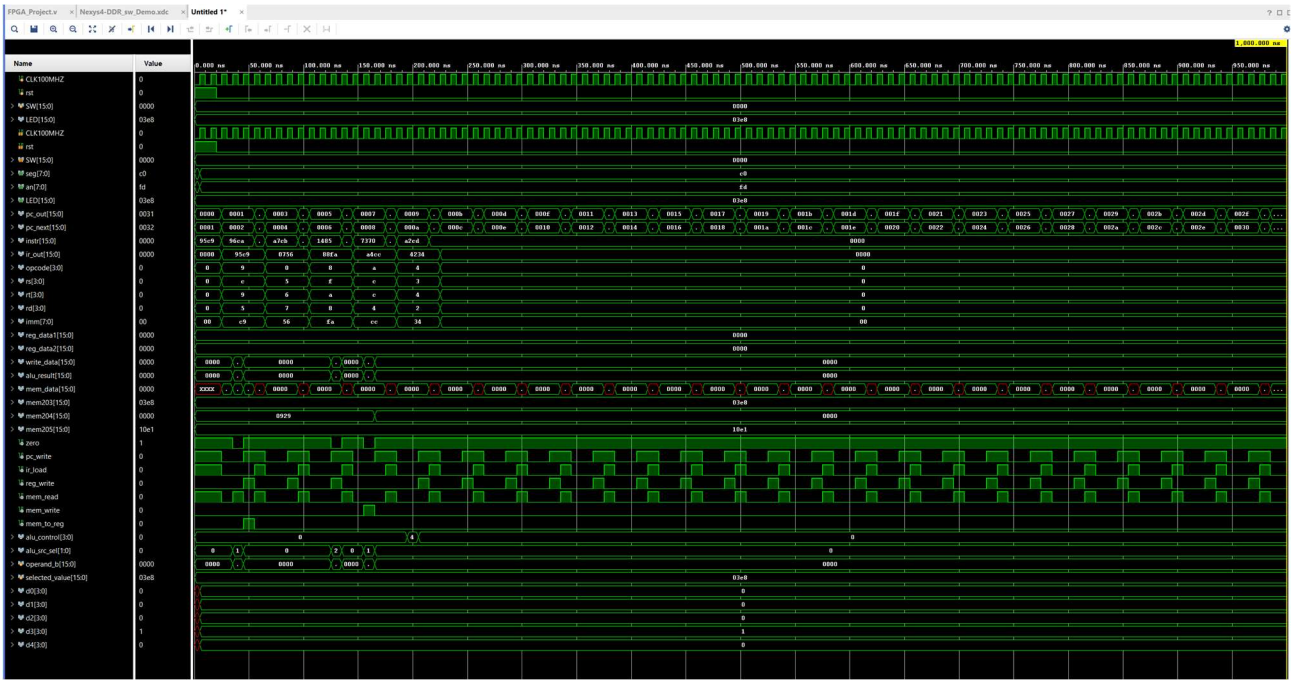
Block Diagram:



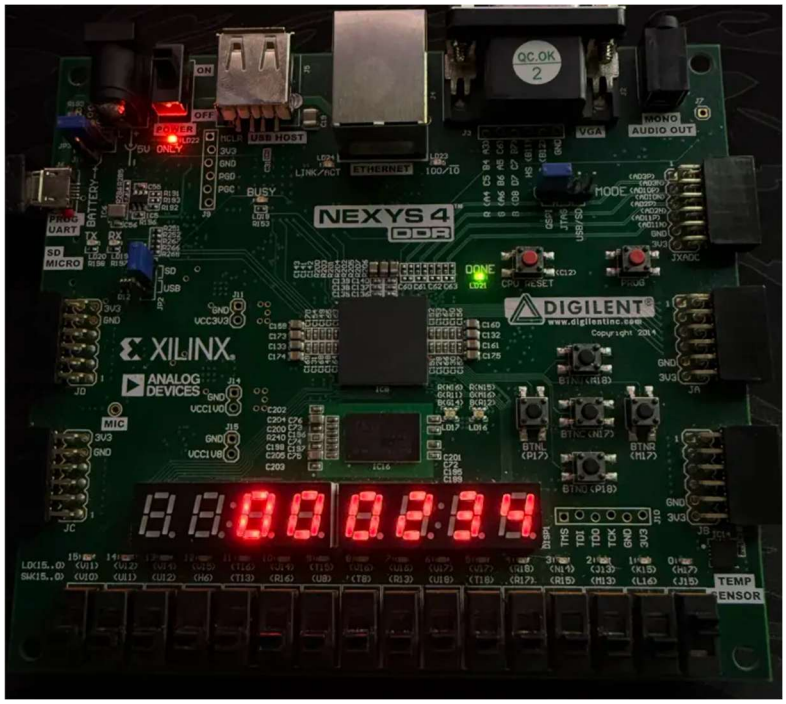
ASM-D:

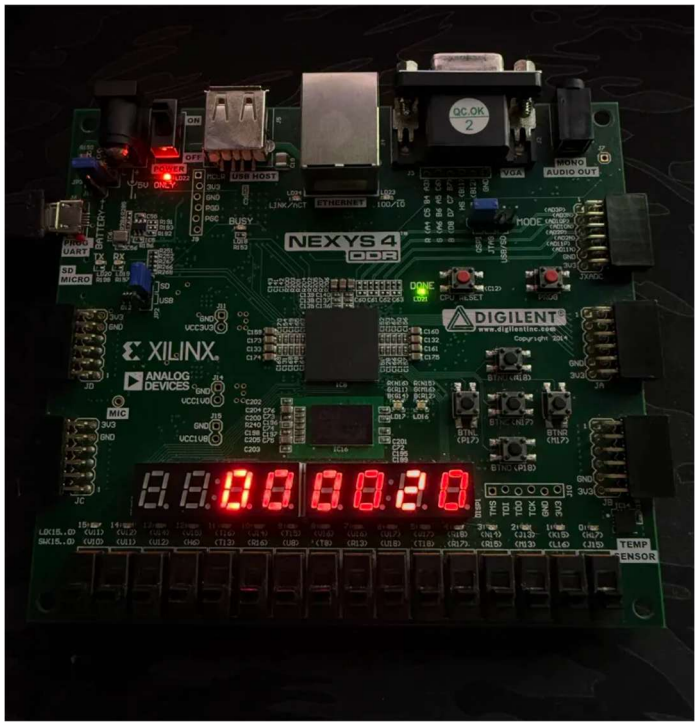


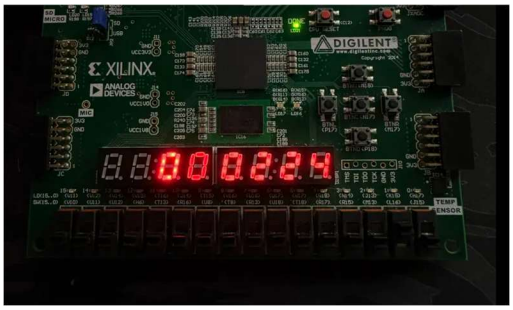

Simulation:



On Board Pictures:


NEXYS 4
DDR
DIGILENT
XILINX
ANALOG DEVICES
TEMP SENSOR


NEXYS 4
DDR
DIGILENT
XILINX
ANALOG DEVICES
TEMP SENSOR


DIGILENT
XILINX
ANALOG DEVICES
TEMP SENSOR