Project report on

Single-Cycle RISC-V Processor

Submitted in partial fulfilment of the requirements for the award of degree of

Bachelor of Technology

in

Electronics and Communication Engineering

Submitted by:

ROHIL S **PES2UG22EC111**

Under the guidance of
Dr Mahesh Awati
Associate Professor
PES University

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
Electronic City, Hosur Road, Bengaluru – 560 100, Karnataka, India

Table of Contents

Description

The **RISC-V processor** is a lightweight, modular, open-source Instruction Set Architecture (ISA) designed for teaching, research, and scalable commercial use. In this project, I implemented a **single-cycle RISC-V processor** using System Verilog, capable of executing basic instructions from the **RV32I** base instruction set.

This processor executes one instruction per clock cycle (single-cycle architecture), and supports:

- **R-type** (e.g. add, sub, and, or)
- **I-type** (e.g. addi, lw)
- **S-type** (e.g. sw)
- **B-type** (e.g. beq)
- **U-type** (e.g. lui)
- **J-type** (e.g. jal)

RTL Code (System Verilog)

```
module riscv_core (
    input logic clk,
    input logic reset
);
    logic [31:0] pc, next_pc;
    logic [31:0] instr;
    logic [4:0] rs1, rs2, rd;
    logic [31:0] reg_data1, reg_data2, alu_result, imm;
    logic [31:0] write_data;
    logic [6:0] opcode;
    logic [2:0] funct3;
    logic [6:0] funct7;
```

```
// Fetch
pc_register pc_reg (
    .clk(clk),
    .reset(reset),
    .next_pc(next_pc),
    .pc(pc)
);
instruction_memory imem (
    .addr(pc),
    .instruction(instr)
);

// Decode
assign opcode = instr[6:0];
assign rd     = instr[11:7];
assign funct3 = instr[14:12];
assign rs1    = instr[19:15];
assign rs2    = instr[24:20];
assign funct7 = instr[31:25];

register_file rf (
    .clk(clk),
    .rs1(rs1),
    .rs2(rs2),
    .rd(rd),
    .write_data(write_data),
    .reg_write(reg_write),
    .data1(reg_data1),
    .data2(reg_data2)
);

immediate_generator imm_gen (
    .instr(instr),
    .imm_out(imm)
);

alu alu_unit (
```

```
        .a(reg_data1),
        .b(alu_src ? imm : reg_data2),
        .alu_control(alu_control),
        .result(alu_result)
    );

    data_memory dmem (
        .clk(clk),
        .addr(alu_result),
        .write_data(reg_data2),
        .mem_read(mem_read),
        .mem_write(mem_write),
        .read_data(mem_read_data)
    );

    control_unit control (
        .opcode(opcode),
        .reg_write(reg_write),
        .alu_src(alu_src),
        .mem_to_reg(mem_to_reg),
        .mem_read(mem_read),
        .mem_write(mem_write),
        .alu_op(alu_op),
        .branch(branch)
    );

    alu_control alu_ctrl (
        .alu_op(alu_op),
        .funct3(funct3),
        .funct7(funct7),
        .alu_control(alu_control)
    );

    assign write_data = mem_to_reg ? mem_read_data : alu_result;
    assign next_pc = pc + 4;  // Basic PC increment; to be replaced
with branch logic
```

```
endmodule
--------------------------------------------------------------------------------
module pc_register (
    input  logic clk,
    input  logic reset,
    input  logic [31:0] next_pc,
    output logic [31:0] pc
);
    always_ff @(posedge clk or posedge reset) begin
        if (reset)
            pc <= 32'h00000000;
        else
            pc <= next_pc;
    end
endmodule
--------------------------------------------------------------------------------
module instruction_memory (
    input  logic [31:0] addr,
    output logic [31:0] instruction
);
    logic [31:0] memory [0:255];

    initial begin
        // Example: addi x1, x0, 10
        memory[0] = 32'h00a00093;  // addi x1, x0, 10
        memory[1] = 32'h00100113;  // addi x2, x0, 1
        // Add more instructions here
    end

    assign instruction = memory[addr[9:2]];  // word aligned
endmodule
--------------------------------------------------------------------------------
module register_file (
    input  logic clk,
    input  logic [4:0] rs1, rs2, rd,
    input  logic [31:0] write_data,
    input  logic reg_write,
    output logic [31:0] data1, data2
```

```
);
    logic [31:0] registers [0:31];

    always_ff @(posedge clk) begin
        if (reg_write && rd != 0)
            registers[rd] <= write_data;
    end

    assign data1 = registers[rs1];
    assign data2 = registers[rs2];
endmodule
--------------------------------------------------------------------------------
module immediate_generator (
    input  logic [31:0] instr,
    output logic [31:0] imm_out
);
    logic [6:0] opcode;
    assign opcode = instr[6:0];

    always_comb begin
        case (opcode)
            7'b0010011, 7'b0000011: // I-type (addi, lw)
                imm_out = {{20{instr[31]}}, instr[31:20]};
            7'b0100011: // S-type (sw)
                imm_out = {{20{instr[31]}}, instr[31:25], instr[11:7]};
            7'b1100011: // B-type (beq, bne)
                imm_out = {{19{instr[31]}}, instr[31], instr[7],
instr[30:25], instr[11:8], 1'b0};
            7'b0110111: // U-type (lui)
                imm_out = {instr[31:12], 12'b0};
            7'b1101111: // J-type (jal)
                imm_out = {{11{instr[31]}}, instr[31], instr[19:12],
instr[20], instr[30:21], 1'b0};
            default:
                imm_out = 32'd0;
        endcase
    end
endmodule
```

```
--------------------------------------------------------------------------------
module alu (
    input  logic [31:0] a, b,
    input  logic [3:0] alu_control,
    output logic [31:0] result
);
    always_comb begin
        case (alu_control)
            4'b0000: result = a & b;
            4'b0001: result = a | b;
            4'b0010: result = a + b;
            4'b0110: result = a - b;
            4'b0111: result = (a < b) ? 32'd1 : 32'd0;
            4'b1100: result = a ^ b;
            default: result = 32'd0;
        endcase
    end
endmodule
--------------------------------------------------------------------------------
module data_memory (
    input  logic clk,
    input  logic [31:0] addr,
    input  logic [31:0] write_data,
    input  logic mem_read,
    input  logic mem_write,
    output logic [31:0] read_data
);
    logic [31:0] memory [0:255];

    always_ff @(posedge clk) begin
        if (mem_write)
            memory[addr[9:2]] <= write_data;
    end

    assign read_data = mem_read ? memory[addr[9:2]] : 32'd0;
endmodule
--------------------------------------------------------------------------------
module control_unit (
```

```
    input  logic [6:0] opcode,
    output logic reg_write,
    output logic alu_src,
    output logic mem_to_reg,
    output logic mem_read,
    output logic mem_write,
    output logic [1:0] alu_op,
    output logic branch
);

    always_comb begin
        case (opcode)
            7'b0110011: begin // R-type
                alu_src    = 0;
                mem_to_reg  = 0;
                reg_write   = 1;
                mem_read    = 0;
                mem_write   = 0;
                branch      = 0;
                alu_op      = 2'b10;
            end
            7'b0010011: begin // I-type (addi)
                alu_src    = 1;
                mem_to_reg  = 0;
                reg_write   = 1;
                mem_read    = 0;
                mem_write   = 0;
                branch      = 0;
                alu_op      = 2'b00;
            end
            7'b0000011: begin // Load
                alu_src    = 1;
                mem_to_reg  = 1;
                reg_write   = 1;
                mem_read    = 1;
                mem_write   = 0;
                branch      = 0;
                alu_op      = 2'b00;
```

```
            end
            7'b0100011: begin // Store
               alu_src    = 1;
               mem_to_reg  = 0; // X
               reg_write   = 0;
               mem_read    = 0;
               mem_write   = 1;
               branch      = 0;
               alu_op      = 2'b00;
            end
            7'b1100011: begin // Branch
               alu_src    = 0;
               mem_to_reg  = 0;
               reg_write   = 0;
               mem_read    = 0;
               mem_write   = 0;
               branch      = 1;
               alu_op      = 2'b01;
            end
            default: begin
               alu_src    = 0;
               mem_to_reg  = 0;
               reg_write   = 0;
               mem_read    = 0;
               mem_write   = 0;
               branch      = 0;
               alu_op      = 2'b00;
            end
         endcase
      end
endmodule
--------------------------------------------------------------------------------
module alu_control (
   input  logic [1:0] alu_op,
   input  logic [2:0] funct3,
   input  logic [6:0] funct7,
   output logic [3:0] alu_control
);
```

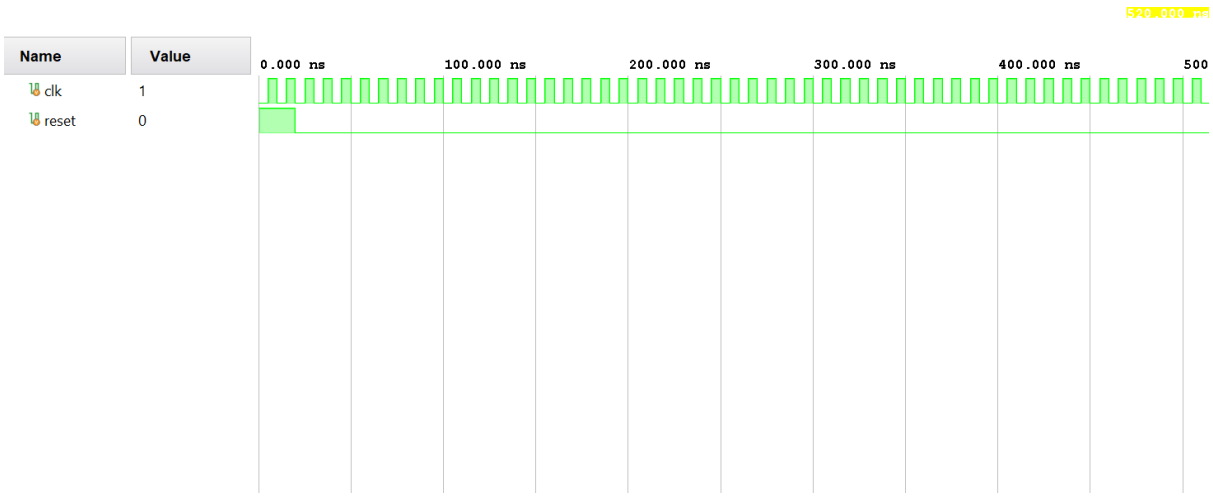
```
    always_comb begin
        case (alu_op)
            2'b00: alu_control = 4'b0010; // ADD (lw/sw/addi)
            2'b01: alu_control = 4'b0110; // SUB (branch)
            2'b10: begin // R-type
                case ({funct7, funct3})
                    10'b0000000000: alu_control = 4'b0010; // ADD
                    10'b0100000000: alu_control = 4'b0110; // SUB
                    10'b0000000111: alu_control = 4'b0000; // AND
                    10'b0000000110: alu_control = 4'b0001; // OR
                    10'b0000000100: alu_control = 4'b1100; // XOR
                    default:        alu_control = 4'b1111;
                endcase
            end
            default: alu_control = 4'b0000;
        endcase
    end
endmodule
--------------------------------------------------------------------------------
```

Testbench Code

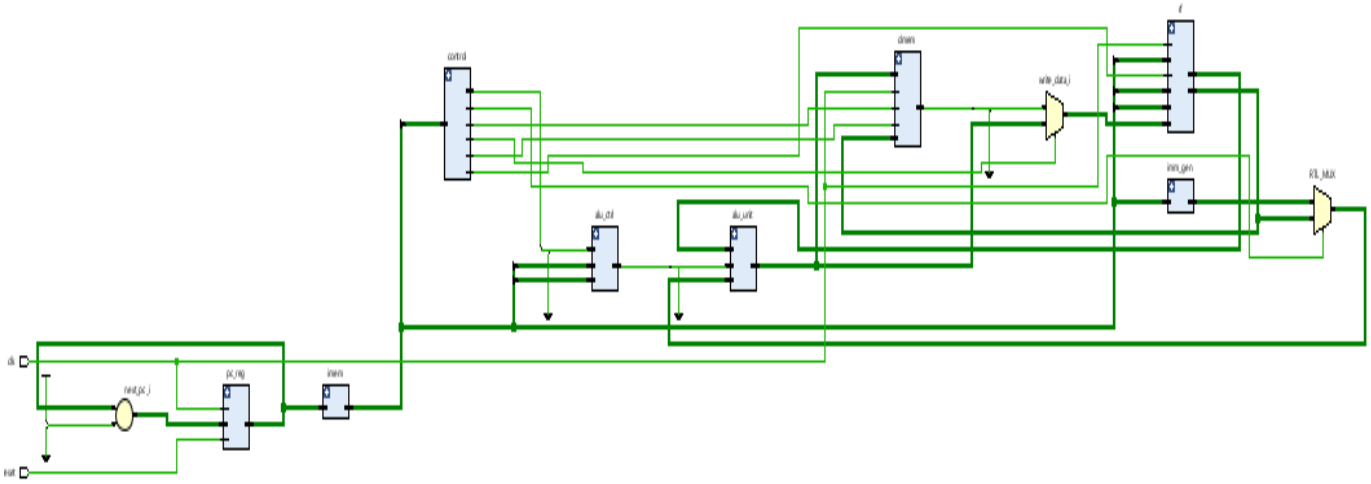
```
`timescale 1ns / 1ps

module tb_riscv_core;
    logic clk;
    logic reset;

    // Instantiate the DUT
    riscv_core uut (
        .clk(clk),
        .reset(reset)
    );

    // Clock generation: 10ns period (100 MHz)
    always #5 clk = ~clk;
```

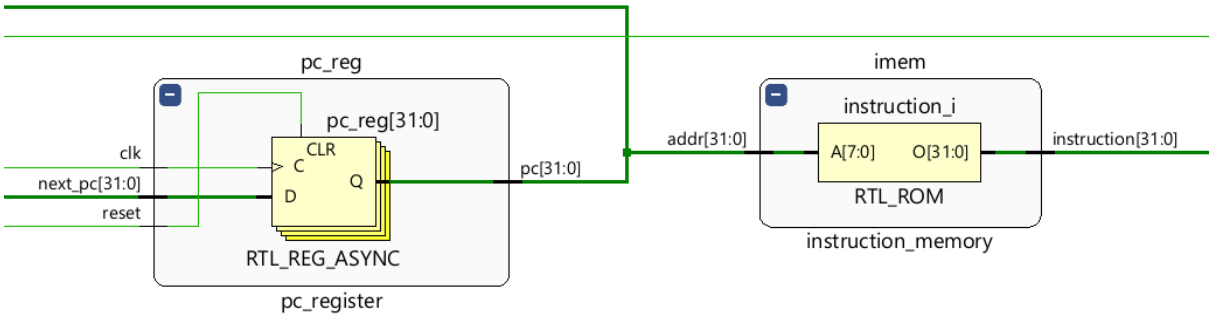
```
    initial begin
        $display("Starting RISC-V Processor Simulation...");

        // Initialize signals
        clk = 0;
        reset = 1;

        // Hold reset for a few cycles
        #20;
        reset = 0;

        // Let the processor run for a while
        #500;

        $display("Simulation Finished.");
        $finish;
    end

endmodule
```

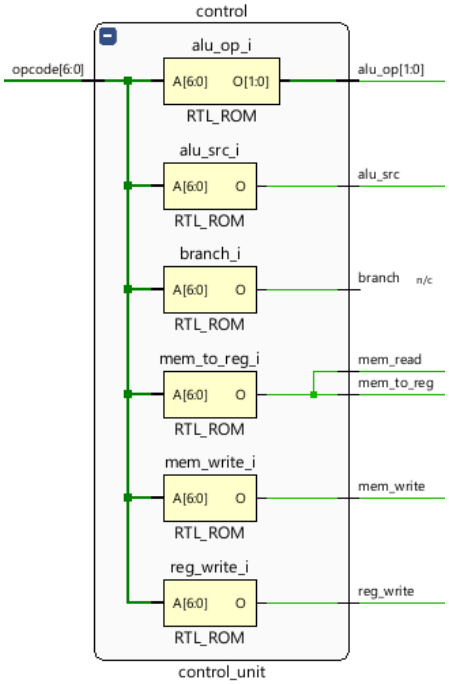
Simulation

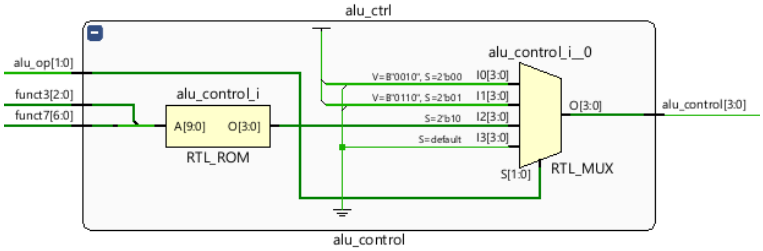


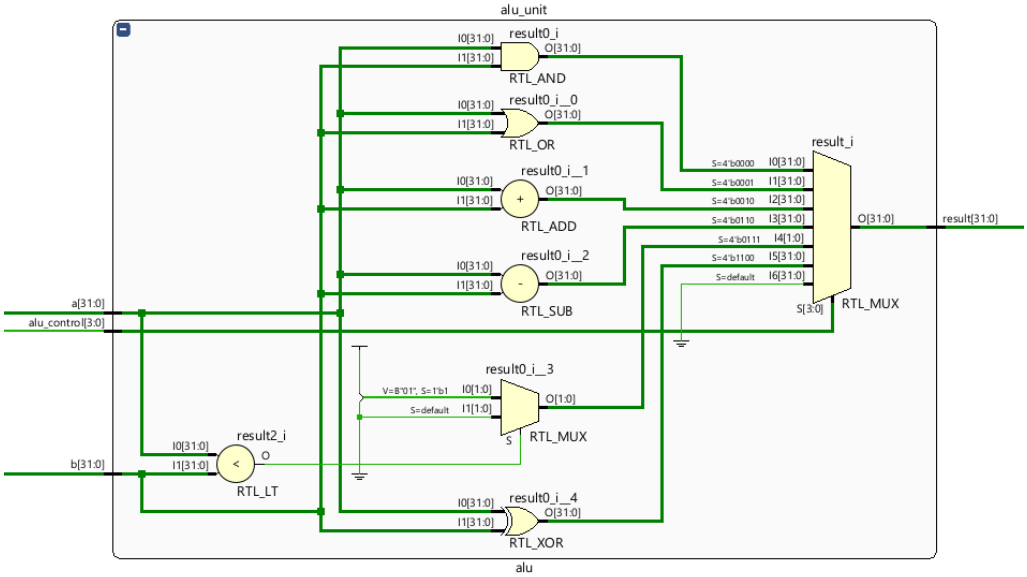
Schematic

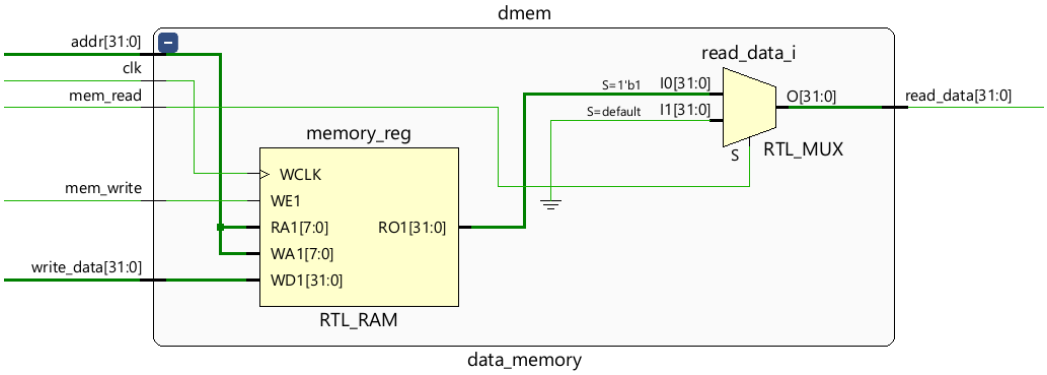


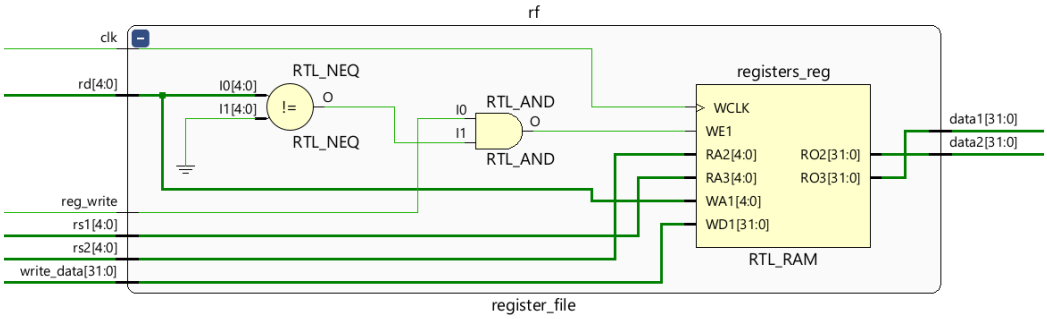

pc_reg
pc_reg[31:0]
clk
next_pc[31:0]
reset
CLR
C
D
Q
RTL_REG_ASYNC
pc_register
pc[31:0]
imem
addr[31:0]
instruction_i
A[7:0]
O[31:0]
RTL_ROM
instruction_memory
instruction[31:0]

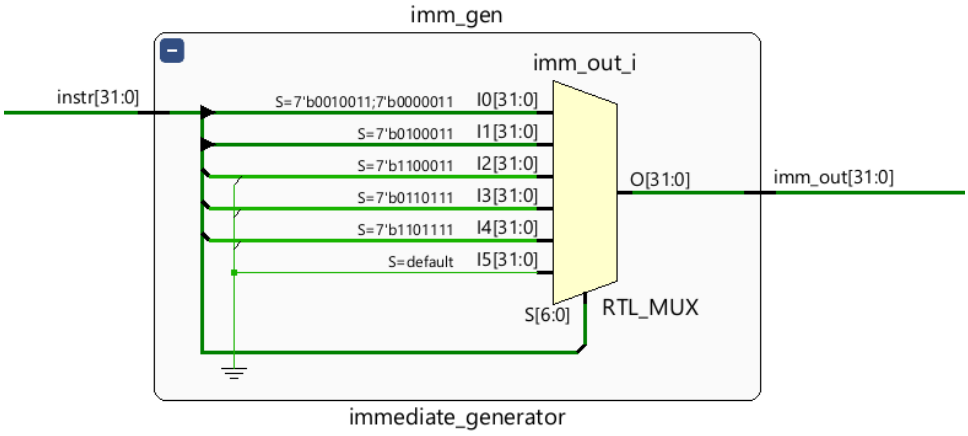

control
opcode[6:0]
alu_op_i
A[6:0]
O[1:0]
RTL_ROM
alu_op[1:0]
alu_src_i
O
alu_src
branch_i
branch
n/c
mem_to_reg_i
mem_read
mem_to_reg
mem_write_i
mem_write
reg_write_i
reg_write
control_unit


alu_ctrl
alu_op[1:0]
funct3[2:0]
funct7[6:0]
alu_control_i
A[9:0]
O[3:0]
RTL_ROM
alu_control_i__0
I0[3:0]
I1[3:0]
I2[3:0]
I3[3:0]
S[1:0]
RTL_MUX
O[3:0]
alu_control[3:0]
alu_control


alu_unit
result0_i
RTL_AND
result0_i__0
RTL_OR
result0_i__1
RTL_ADD
result0_i__2
RTL_SUB
result_i
RTL_MUX
result0_i__3
RTL_MUX
result2_i
RTL_LT
result0_i__4
RTL_XOR
a[31:0]
alu_control[3:0]
b[31:0]
result[31:0]
alu


dmem
addr[31:0]
clk
mem_read
mem_write
write_data[31:0]
memory_reg
WCLK
WE1
RA1[7:0]
RO1[31:0]
WA1[7:0]
WD1[31:0]
RTL_RAM
read_data_i
RTL_MUX
read_data[31:0]
data_memory


rf
clk
rd[4:0]
RTL_NEQ
RTL_NEQ
RTL_AND
RTL_AND
registers_reg
WCLK
WE1
RA2[4:0]
RO2[31:0]
RA3[4:0]
RO3[31:0]
WA1[4:0]
WD1[31:0]
RTL_RAM
data1[31:0]
data2[31:0]
reg_write
rs1[4:0]
rs2[4:0]
write_data[31:0]
register_file


imm_gen
imm_out_i
instr[31:0]
RTL_MUX
imm_out[31:0]
immediate_generator

Applications

- Understanding instruction-level execution
- Educational VLSI/FPGA projects
- Foundation for pipelined or multi-cycle CPU design
- Useful in embedded systems with custom ISA extensions

Advantages

- Modular and scalable RTL design
- Fully synthesizable
- Instruction-driven behavior via RISC-V format
- Easily extendable to support more instructions or pipelining

Disadvantages

- No hazard detection (yet)
- Single-cycle limits performance
- No forwarding, pipelining, or branch prediction
- Limited instruction support (subset of RV32I)

Conclusion

Implementing a single-cycle RISC-V processor deepened my understanding of the **hardware-software interface**, **control signals**, and **instruction execution pipelines**. This foundational design sets the stage for adding pipelining, memory-mapped IO, or even custom instructions in the future.