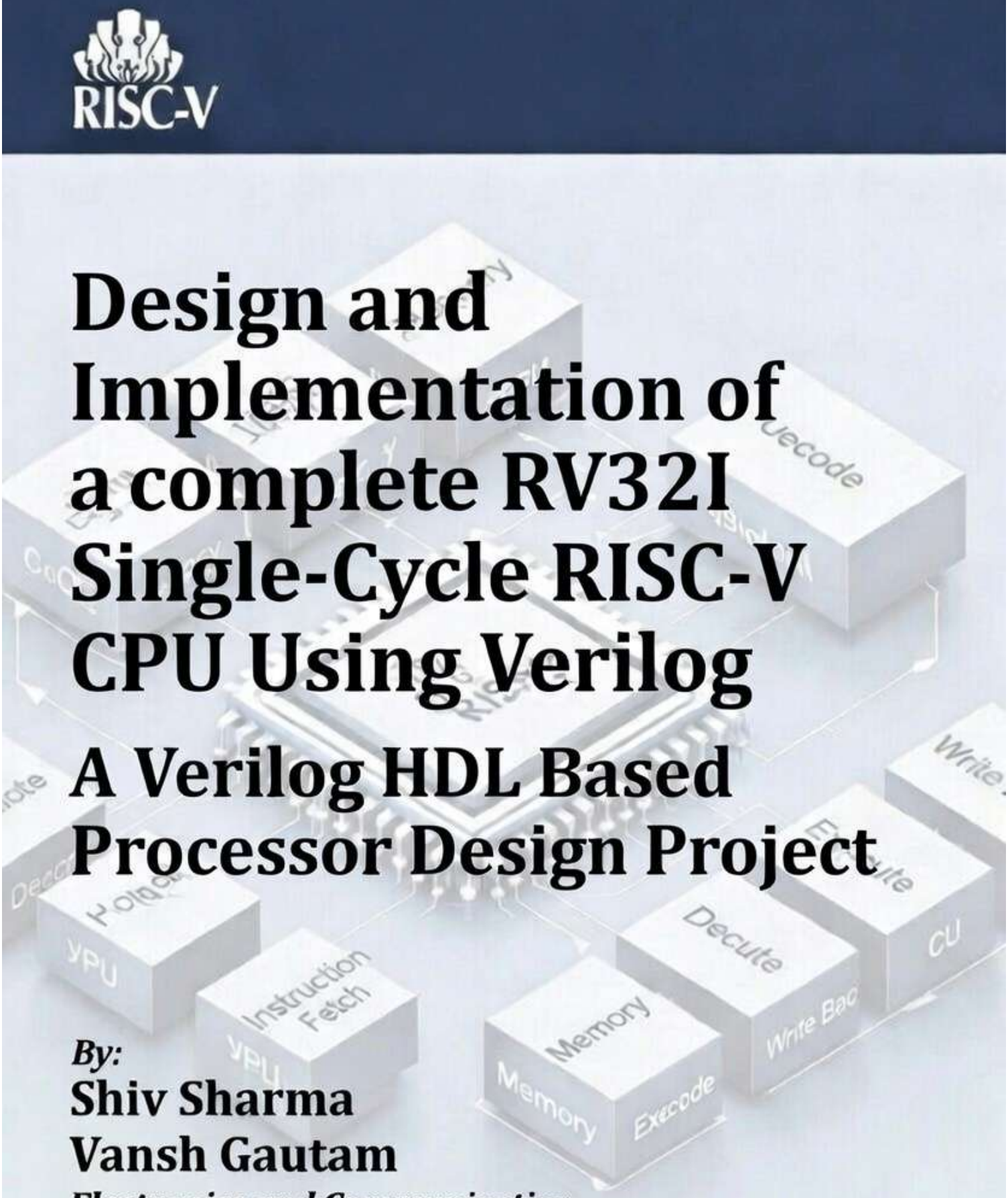


Design and Implementation of a complete RV32I Single-Cycle RISC-V CPU Using Verilog

A Verilog HDL Based Processor Design Project

By:
Shiv Sharma
Vansh Gautam
Electronics and Communication Engineering / VLSI / Digital Design

https://www.youtube.com/@siliconcircle

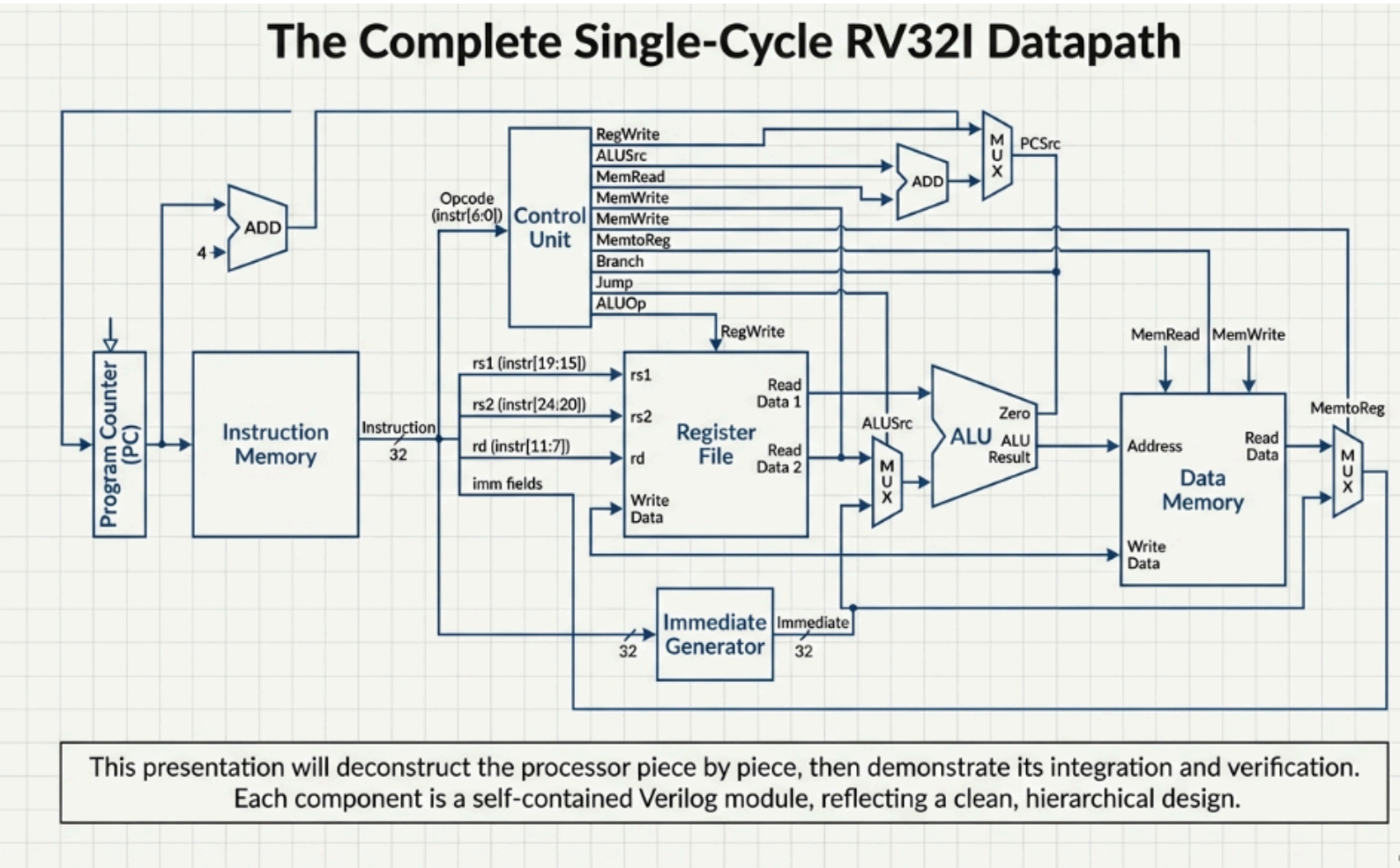
Introduction

RISC-V is an open-standard ISA gaining traction in academia, industry, and open-source hardware due to its modularity and license-free nature. As an aspiring hardware engineer, I undertook this project to implement a complete RV32I CPU from scratch in Verilog. This hands-on experience helped me master datapath design, control logic, and instruction decoding while exploring edge cases like signed arithmetic and branch prediction.

Objectives

- Implement all RV32I instructions as per the RISC-V specification.
- Support advanced features like variable-width memory operations and performance monitoring.
- Include a self-contained test suite for verification.

EDA Playground Link: https://www.edaplayground.com/x/D_uk



. Features of the CPU

- Full RV32I instruction set support
- Single-cycle CPU architecture
- Step-by-step execution mode
- Byte, halfword, and word memory access
- Sign and zero extension support
- Branch and jump instructions (BEQ, BNE, BLT, BGE, JAL, JALR)
- Support for LUI and AUIPC instructions
- ECALL and EBREAK handling with halt mechanism
- Performance counters (cycle count and instruction count)
- Memory-mapped I/O for LED output
- Fully modular RTL design

CPU Architecture Overview

The CPU follows a single-cycle architecture, where each instruction is completed in one clock cycle. The main blocks of the CPU are:

- Program Counter (PC)
- Instruction Memory
- Control Unit
- Register File
- Immediate Generator
- Arithmetic Logic Unit (ALU)
- Data Memory
- Writeback Unit
- Performance Counter

PC → Instruction Memory → Decode → Register File → ALU → Data Memory → Writeback

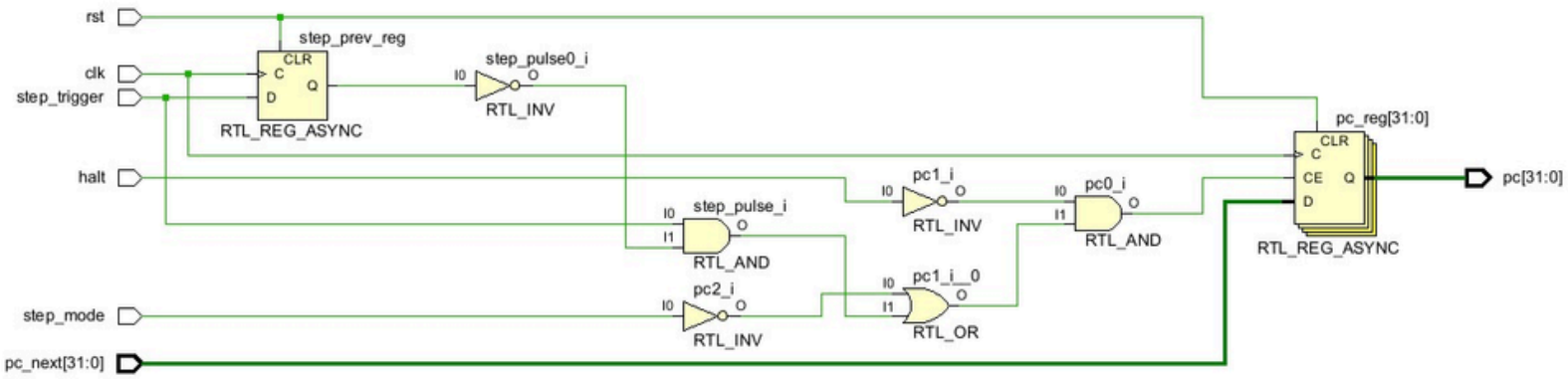
Module-Wise Design

1.Program Counter (PC)

The Program Counter holds the address of the current instruction. It supports normal execution, step-by-step execution mode, and halt control. The PC updates to PC+4, branch target, or jump target based on control signals.

code:

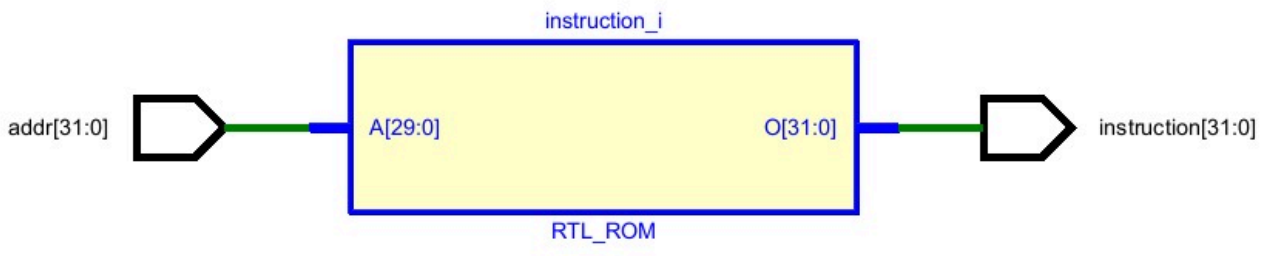
```
// ==================== Program Counter ====================
module PC(
    input clk, rst, step_mode, step_trigger, halt,
    input [31:0] pc_next,
    output reg [31:0] pc
);
    reg step_prev;
    wire step_pulse;

    assign step_pulse = step_trigger && !step_prev;

    always @(posedge clk or posedge rst) begin
        if(rst) begin
            pc <= 0;
            step_prev <= 0;
        end else begin
            step_prev <= step_trigger;
            if(!halt && (!step_mode || step_pulse))
                pc <= pc_next;
        end
    end
endmodule
```



Instruction Memory

Instruction memory stores the program instructions. It is addressed using PC[31:2] since instructions are word-aligned. A test program covering most RV32I instructions is preloaded into memory.

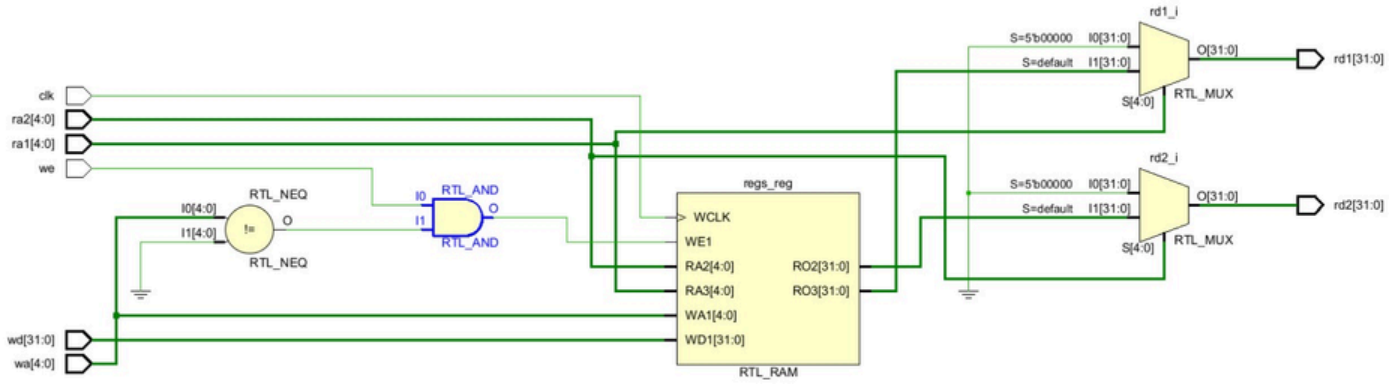
```
// ==================== Instruction Memory ====================
module InstructionMemory(
    input [31:0] addr,
    output [31:0] instruction
);
    reg [31:0] mem [0:127];

    initial begin
        // Comprehensive test program with ALL RV32I instructions

        // Basic arithmetic & logical
        mem[0]  = 32'h00500093; // addi x1, x0, 5     (x1 = 5)
        mem[1]  = 32'h00300113; // addi x2, x0, 3     (x2 = 3)
        mem[2]  = 32'h002081B3; // add  x3, x1, x2    (x3 = 8)
        mem[3]  = 32'h40208233; // sub  x4, x1, x2    (x4 = 2)
        mem[4]  = 32'h0020A2B3; // slt  x5, x1, x2    (x5 = 0)
        mem[5]  = 32'h0020F333; // and  x6, x1, x2    (x6 = 1)
        mem[6]  = 32'h0020E3B3; // or   x7, x1, x2    (x7 = 7)
        mem[7]  = 32'h0020C433; // xor  x8, x1, x2    (x8 = 6)

        // Shifts
        mem[8]  = 32'h00209493; // slli x9, x1, 2     (x9 = 20)
        mem[9]  = 32'h0010D513; // srli x10, x1, 1    (x10 = 2)
        mem[10] = 32'hFFF00593; // addi x11, x0, -1   (x11 = -1)
        mem[11] = 32'h4015D613; // srai x12, x11, 1   (x12 = -1, arith shift)

        // Upper immediate instructions
        mem[12] = 32'h12345B7; // lui  x13, 0x12345  (x13 = 0x12345000)
        mem[13] = 32'h00000717; // auipc x14, 0       (x14 = PC = 52)

        // Memory operations - Word
        mem[14] = 32'h00412223; // sw   x4, 4(x2)     (mem[7] = 2)
        mem[15] = 32'h00412783; // lw   x15, 4(x2)    (x15 = 2)

        // Memory operations - Byte
        mem[16] = 32'h00410823; // sb   x4, 16(x2)    (mem[19] byte = 2)
        mem[17] = 32'h01010803; // lb   x16, 16(x2)   (x16 = 2, sign-ext)
        mem[18] = 32'h01014883; // lbu  x17, 16(x2)   (x17 = 2, zero-ext)

        // Memory operations - Halfword
        mem[19] = 32'hFFE00913; // addi x18, x0, -2   (x18 = -2)
        mem[20] = 32'h01211923; // sh   x18, 18(x2)   (mem[21] half = -2)
        mem[21] = 32'h01211983; // lh   x19, 18(x2)   (x19 = -2, sign-ext)
        mem[22] = 32'h01215A03; // lhu  x20, 18(x2)   (x20 = 0xFFFE, zero-ext)

        // Branch instructions
        mem[23] = 32'h00208863; // beq  x1, x2, 16    (not taken)
        mem[24] = 32'h00209463; // bne  x1, x2, 8     (taken, skip next)
        mem[25] = 32'h00000013; // nop (skipped)
        mem[26] = 32'h00114A63; // blt  x2, x1, 20    (taken, 3 < 5)
        mem[27] = 32'h00000013; // nop (skipped)
        mem[28] = 32'h00000013; // nop (skipped)
        mem[29] = 32'h00000013; // nop (skipped)
        mem[30] = 32'h00115463; // bge  x2, x1, 8     (not taken)
        mem[31] = 32'h00A00A93; // addi x21, x0, 10   (x21 = 10)

        // Jump instructions
        mem[32] = 32'h008000EF; // jal  x1, 8         (x1 = PC+4=132, jump to 136)
        mem[33] = 32'h00000013; // nop (skipped)
        mem[34] = 32'h00C00B13; // addi x22, x0, 12   (x22 = 12, at PC=136)
        mem[35] = 32'h00008067; // jalr x0, x1, 0     (jump to x1=132, return)
        mem[36] = 32'h00D00B93; // addi x23, x0, 13   (x23 = 13, back at 140)

        // System calls
        mem[37] = 32'h00000073; // ecall              (environment call)
        mem[38] = 32'h00100073; // ebreak             (breakpoint)

        // End program
        mem[39] = 32'h00000013; // nop
        mem[40] = 32'h00000013; // nop
    end

    assign instruction = mem[addr[31:2]];
endmodule
```



Register File

The register file contains 32 registers of 32-bit width. Register x0 is always hardwired to zero. The design supports synchronous write and asynchronous read operations.

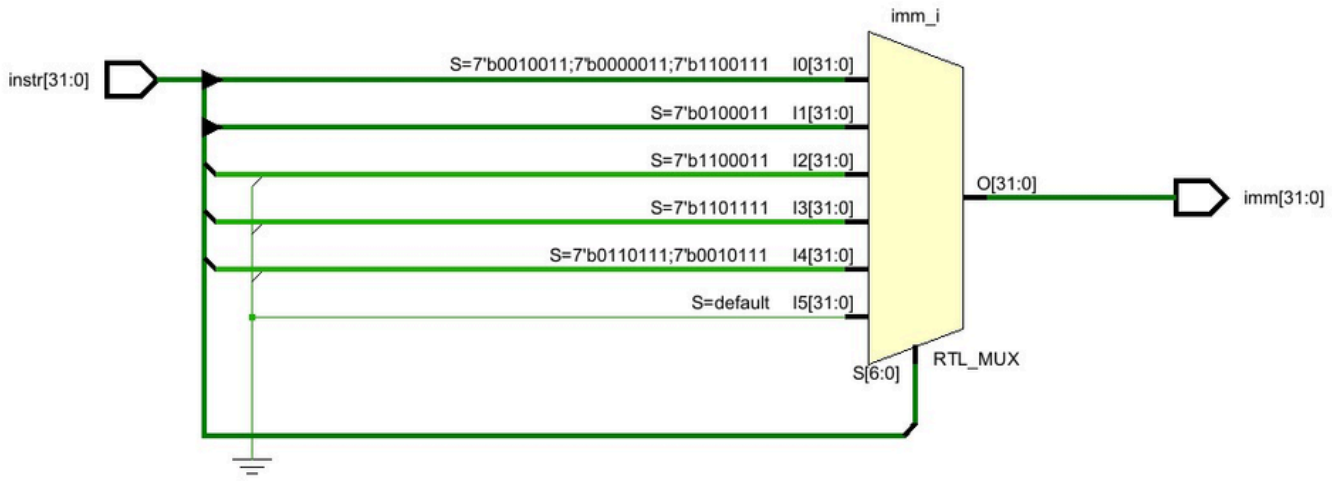
```
// ==================== Register File ====================
module RegisterFile(
    input clk, we,
    input [4:0] ra1, ra2, wa,
    input [31:0] wd,
    output [31:0] rd1, rd2
);
    reg [31:0] regs [0:31];
    integer i;

    initial begin
        for(i = 0; i < 32; i = i + 1)
            regs[i] = 0;
    end

    assign rd1 = (ra1 == 0) ? 0 : regs[ra1];
    assign rd2 = (ra2 == 0) ? 0 : regs[ra2];

    always @(posedge clk) begin
        if(we && wa != 0)
            regs[wa] <= wd;
    end
endmodule
```



Immediate Generator

The immediate generator extracts and sign-extends or zero-extends the immediate field from the instruction. It supports all RISC-V immediate formats: I-type, S-type, B-type, U-type, and J-type

```
// ==================== Immediate Generator ====================
module ImmGen(
    input [31:0] instr,
    output reg [31:0] imm
);
    always @(*) begin
        case(instr[6:0])
            7'b0010011, 7'b0000011, 7'b1100111: // I-type (ALU, Load, JALR)
                imm = {{20{instr[31]}}, instr[31:20]};
            7'b0100011: // S-type (Store)
                imm = {{20{instr[31]}}, instr[31:25], instr[11:7]};
            7'b1100011: // B-type (Branch)
                imm = {{19{instr[31]}}, instr[31], instr[7], instr[30:25], instr[11:8], 1'b0};
            7'b1101111: // J-type (JAL)
                imm = {{11{instr[31]}}, instr[31], instr[19:12], instr[20], instr[30:21], 1'b0};
            7'b0110111, 7'b0010111: // U-type (LUI, AUIPC)
                imm = {instr[31:12], 12'b0};
            default: imm = 0;
        endcase
    end
endmodule
```

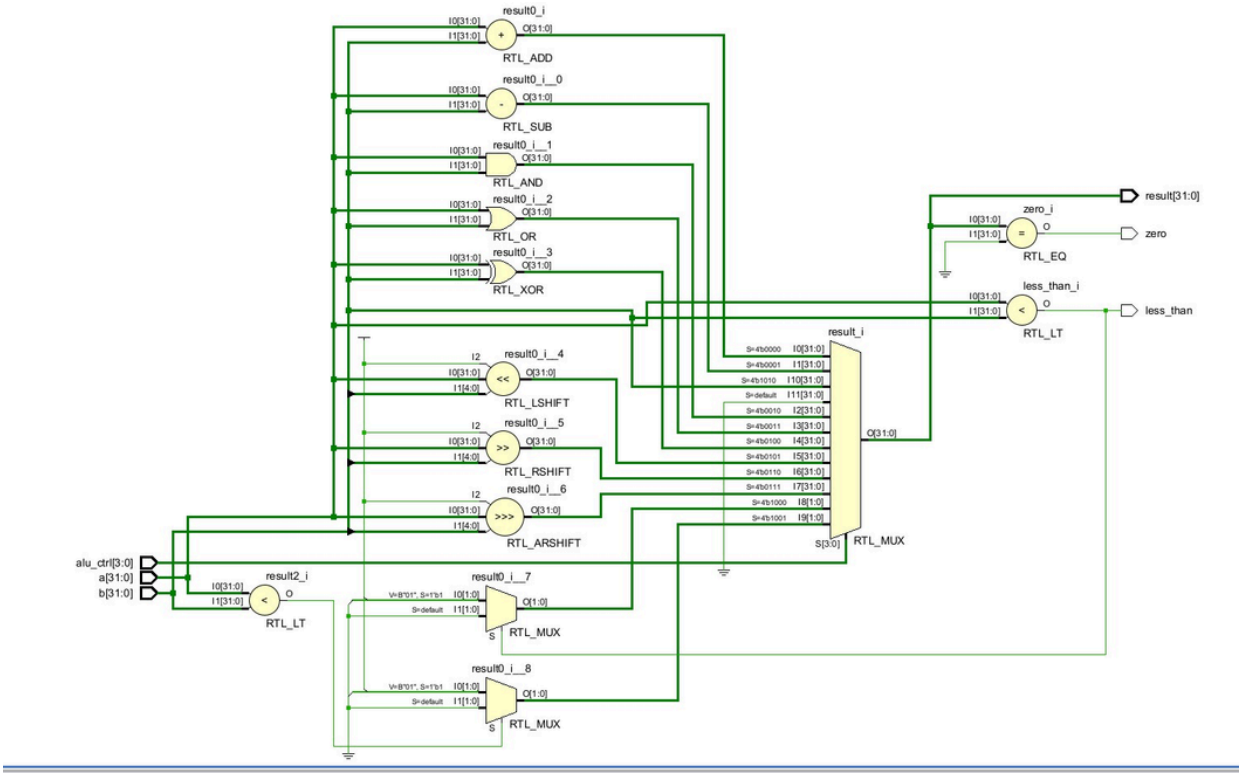


Arithmetic Logic Unit (ALU)

The ALU performs arithmetic and logical operations such as:

- ADD, SUB
- AND, OR, XOR
- SLL, SRL, SRA
- SLT, SLTU

It also generates zero and less-than flags used for branch decisions.



```
// ==================== ALU ====================
module ALU(
    input [31:0] a, b,
    input [3:0] alu_ctrl,
    output reg [31:0] result,
    output zero, less_than
);
    always @(*) begin
        case(alu_ctrl)
            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 << b[4:0];      // SLL
            4'b0110: result = a >> b[4:0];      // SRL
            4'b0111: result = $signed(a) >>> b[4:0]; // SRA
            4'b1000: result = ($signed(a) < $signed(b)) ? 1 : 0; // SLT
            4'b1001: result = (a < b) ? 1 : 0; // SLTU
            4'b1010: result = b;                // PASS B (for LUI, AUIPC)
            default: result = 0;
        endcase
    end

    assign zero = (result == 0);
    assign less_than = ($signed(a) < $signed(b));
endmodule
```

Control Unit

The control unit decodes the instruction opcode, funct3, and funct7 fields and generates all required control signals such as:

- reg_write, mem_write, mem_read
- alu_src, mem_to_reg
- branch, jump, jalr
- mem_size, mem_sign_ext
- alu_ctrl

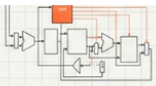
It also detects ECALL and EBREAK instructions.

```
// ==================== Control Unit (Complete RV32I) ====================
module ControlUnit(
    input [6:0] opcode,
    input [2:0] funct3,
    input [6:0] funct7,
    input [11:0] imm12,
    output reg reg_write, mem_write, mem_read,
    output reg alu_src, mem_to_reg, branch, jump, jalr,
    output reg [1:0] branch_type, mem_size,
    output reg mem_sign_ext, auipc_sel, ecall, ebreak,
    output reg [3:0] alu_ctrl
);
    always @(*) begin
        // Defaults
        reg_write = 0; mem_write = 0; mem_read = 0;
        alu_src = 0; mem_to_reg = 0; branch = 0; jump = 0; jalr = 0;
        branch_type = 2'b00; mem_size = 2'b10; mem_sign_ext = 1;
        auipc_sel = 0; ecall = 0; ebreak = 0;
        alu_ctrl = 4'b0000;

        case(opcode)
            7'b0110011: begin // R-type
                reg_write = 1;
                alu_src = 0;
                case(funct3)
                    3'b000: alu_ctrl = (funct7[5]) ? 4'b0001 : 4'b0000; // SUB : ADD
                    3'b111: alu_ctrl = 4'b0010; // AND
                    3'b110: alu_ctrl = 4'b0011; // OR
                    3'b100: alu_ctrl = 4'b0100; // XOR
                    3'b010: alu_ctrl = 4'b1000; // SLT
                    3'b011: alu_ctrl = 4'b1001; // SLTU
                    3'b001: alu_ctrl = 4'b0101; // SLL
                    3'b101: alu_ctrl = (funct7[5]) ? 4'b0111 : 4'b0110; // SRA : SRL
                endcase
            end

            7'b0010011: begin // I-type ALU
                reg_write = 1;
                alu_src = 1;
                case(funct3)
                    3'b000: alu_ctrl = 4'b0000; // ADDI
                    3'b100: alu_ctrl = 4'b0100; // XORI
                    3'b110: alu_ctrl = 4'b0011; // ORI
                    3'b111: alu_ctrl = 4'b0010; // ANDI
                    3'b010: alu_ctrl = 4'b1000; // SLTI
                    3'b011: alu_ctrl = 4'b1001; // SLTIU
                    3'b001: alu_ctrl = 4'b0101; // SLLI
                    3'b101: alu_ctrl = (funct7[5]) ? 4'b0111 : 4'b0110; // SRAI : SRLI
                endcase
            end

```

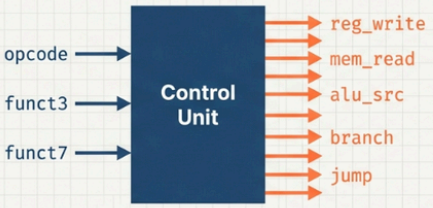
```
            7'b0000011: begin // Load
                reg_write = 1;
                mem_read = 1;
                alu_src = 1;
                mem_to_reg = 1;
                alu_ctrl = 4'b0000;
                case(funct3)
                    3'b000: begin mem_size = 2'b00; mem_sign_ext = 1; end // LB
                    3'b001: begin mem_size = 2'b01; mem_sign_ext = 1; end // LH
                    3'b010: begin mem_size = 2'b10; mem_sign_ext = 1; end // LW
                    3'b100: begin mem_size = 2'b00; mem_sign_ext = 0; end // LBU
                    3'b101: begin mem_size = 2'b01; mem_sign_ext = 0; end // LHU
                endcase
            end

            7'b0100011: begin // Store
                mem_write = 1;
                alu_src = 1;
                alu_ctrl = 4'b0000;
                case(funct3)
                    3'b000: mem_size = 2'b00; // SB
                    3'b001: mem_size = 2'b01; // SH
                    3'b010: mem_size = 2'b10; // SW
                endcase
            end

            7'b1100011: begin // Branch
                branch = 1;
                alu_ctrl = 4'b0001; // SUB
                case(funct3)
                    3'b000: branch_type = 2'b00; // BEQ
                    3'b001: branch_type = 2'b01; // BNE
                    3'b100: branch_type = 2'b10; // BLT
                    3'b101: branch_type = 2'b11; // BGE
                    3'b110: branch_type = 2'b10; // BLTU (reuse BLT logic with
                    3'b111: branch_type = 2'b11; // BGEU
                endcase
            end

            7'b1101111: begin // JAL
                reg_write = 1;
                jump = 1;
            end

            7'b1100111: begin // JALR
                reg_write = 1;
                jalr = 1;
                alu_src = 1;
                alu_ctrl = 4'b0000; // ADD rs1 + imm
            end
```

```
            7'b1101111: begin // JAL
                reg_write = 1;
                jump = 1;
            end

            7'b1100111: begin // JALR
                reg_write = 1;
                jalr = 1;
                alu_src = 1;
                alu_ctrl = 4'b0000; // ADD rs1 + imm
            end

            7'b0110111: begin // LUI
                reg_write = 1;
                alu_src = 1;
                alu_ctrl = 4'b1010; // PASS immediate
            end

            7'b0010111: begin // AUIPC
                reg_write = 1;
                alu_src = 1;
                auipc_sel = 1;
                alu_ctrl = 4'b0000; // ADD PC + imm
            end

            7'b1110011: begin // SYSTEM
                if(funct3 == 3'b000) begin
                    if(imm12 == 12'h000) ecall = 1;
                    else if(imm12 == 12'h001) ebreak = 1;
                end
            end
        endcase
    end
endmodule
```

The Command Center: Decoding Instructions with the Control Unit



Concept & Diagram

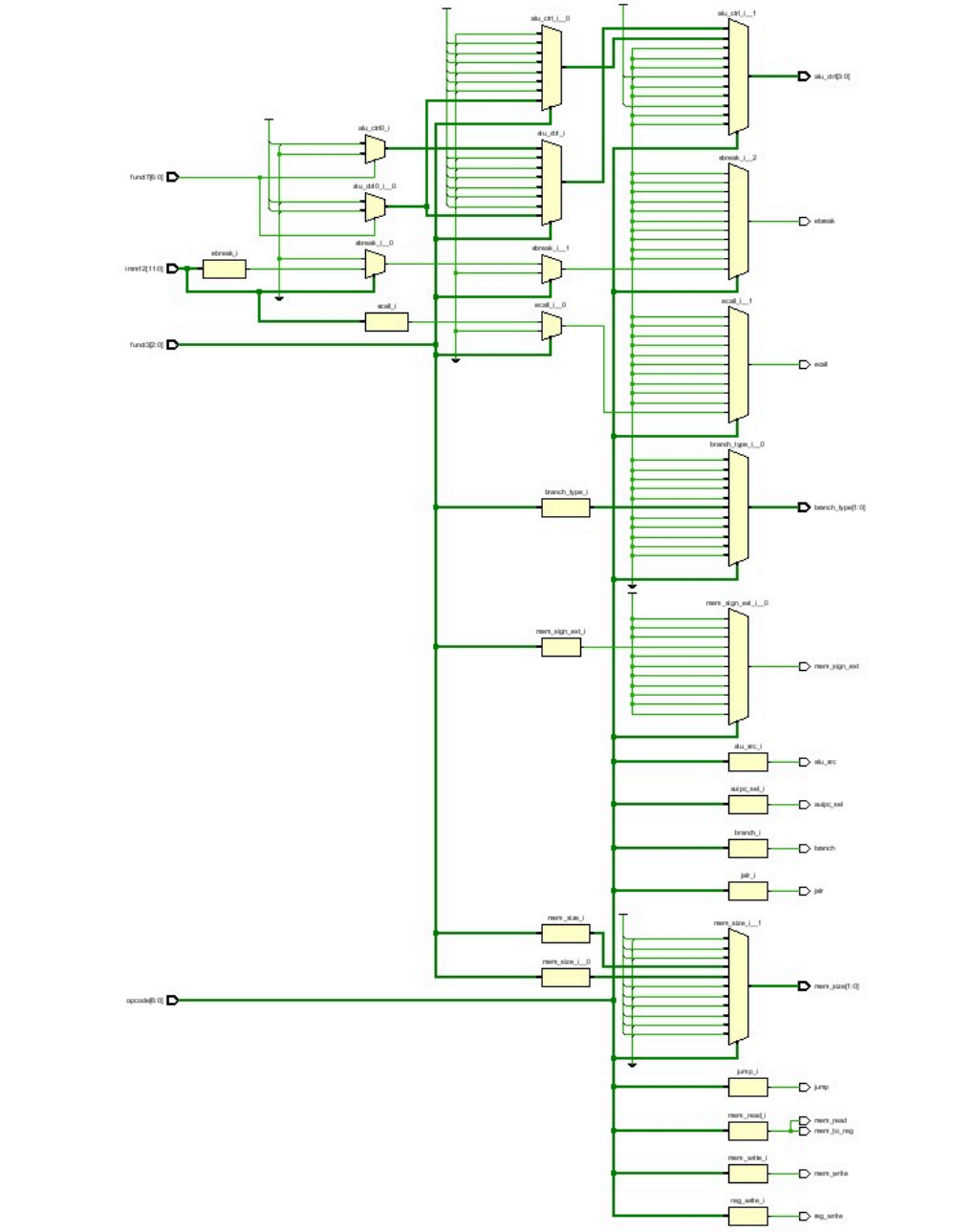


- A purely combinational logic block that translates the instruction's opcode into control signals.
- Generates all signals required to configure the datapath for the specific instruction type (R, I, S, B, U, J).
- Sets default values for all signals to ensure a known state.

Verilog Decoding Logic

```
case(opcode)
    7'b0110011: begin // R-type
                  reg_write = 1;
                  alu_src = 0;
                ... end
    7'b0000011: begin // Load
                  reg_write = 1;
                  mem_read = 1;
                  alu_src = 1;
                  mem_to_reg = 1;
                ... end
    7'b1100011: begin // Branch
                  branch = 1;
                  alu_ctrl = 4'b0001;
                  ... // SUB for compare
    end
endcase
```

The `case` statement forms the core of the decoder, mapping each opcode to a unique configuration of the processor's datapath.



Data Memory

The data memory is byte-addressable and supports:

- Load: LB, LH, LW, LBU, LHU
- Store: SB, SH, SW

It handles sign extension and zero extension during load operations.

```
// ==================== Data Memory (with byte/halfword support) ====================
     module DataMemory(
    input clk, we, re,
    input [1:0] size,        // 00=byte, 01=half, 10=word
    input sign_ext,
    input [31:0] addr, wd,
    output reg [31:0] rd
);
    reg [7:0] mem [0:255];   // Byte-addressable memory
    wire [1:0] byte_offset;
    wire [31:0] aligned_addr;

    assign byte_offset = addr[1:0];
    assign aligned_addr = {addr[31:2], 2'b00};

    // Read logic
    always @(*) begin
        case(size)
            2'b00: begin // Byte
                if(sign_ext)
                    rd = {{24{mem[addr][7]}}, mem[addr]};
                else
                    rd = {24'b0, mem[addr]};
            end
            2'b01: begin // Halfword
                if(sign_ext)
                    rd = {{16{mem[addr+1][7]}}, mem[addr+1], mem[addr]};
                else
                    rd = {16'b0, mem[addr+1], mem[addr]};
            end
            2'b10: begin // Word
                rd = {mem[aligned_addr+3], mem[aligned_addr+2],
                      mem[aligned_addr+1], mem[aligned_addr]};
            end
            default: rd = 0;
        endcase
    end

    // Write logic
    always @(posedge clk) begin
        if(we) begin
            case(size)
                2'b00: mem[addr] <= wd[7:0];              // Byte
                2'b01: begin                               // Halfword
                    mem[addr] <= wd[7:0];
                    mem[addr+1] <= wd[15:8];
                end
                2'b10: begin                               // Word
                    mem[aligned_addr] <= wd[7:0];
                    mem[aligned_addr+1] <= wd[15:8];
                    mem[aligned_addr+2] <= wd[23:16];
                    mem[aligned_addr+3] <= wd[31:24];
                end
            endcase
        end
    end
endmodule
```

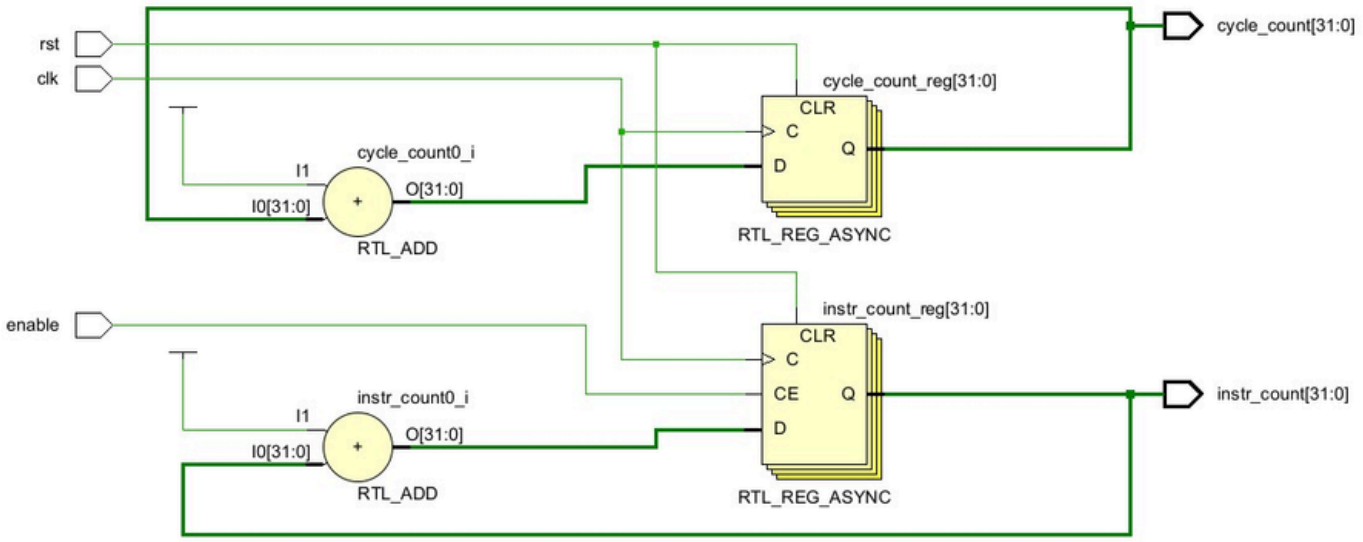
Performance Counter

The performance counter keeps track of:

- Total number of clock cycles
- Total number of executed instructions

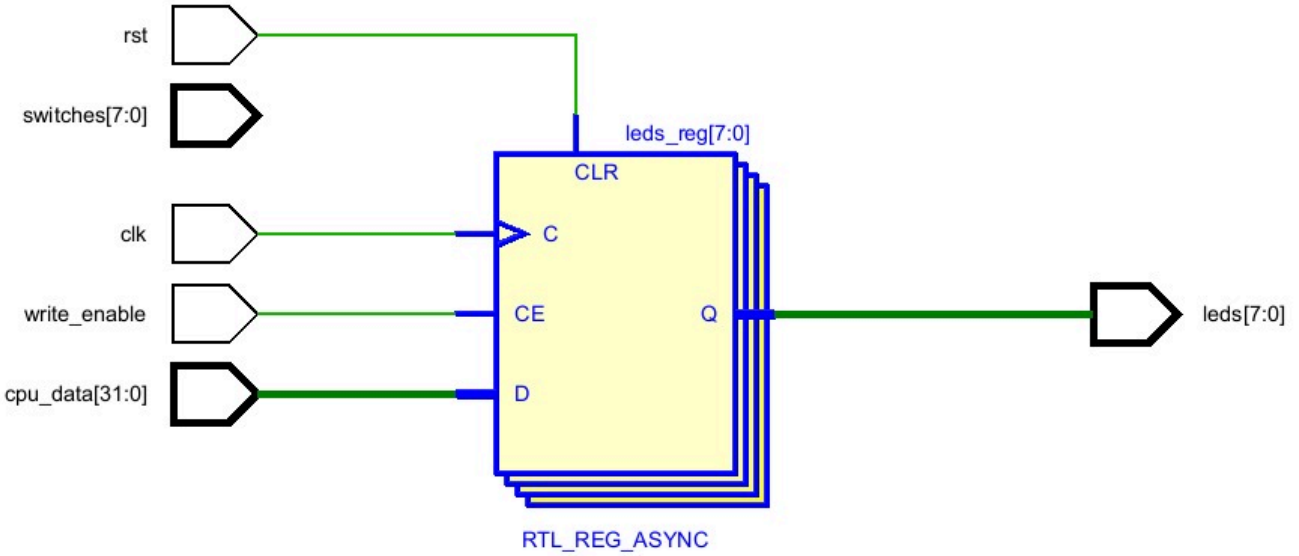
This helps in analyzing CPU performance.

```
// ==================== Performance Counter ====================
module PerformanceCounter(
    input clk, rst, enable,
    output reg [31:0] cycle_count,
    output reg [31:0] instr_count
);
    always @(posedge clk or posedge rst) begin
        if(rst) begin
            cycle_count <= 0;
            instr_count <= 0;
        end else begin
            cycle_count <= cycle_count + 1;
            if(enable)
                instr_count <= instr_count + 1;
        end
    end
endmodule
```



Simple I/O Module

The Simple I/O module provides a basic way to connect the CPU with external LEDs using memory-mapped I/O. When the CPU writes data to a fixed address (0xFFFF0000), the lower 8 bits of the data are sent to the LEDs. This allows the CPU to display values on the LEDs and helps in testing store instructions and I/O interfacing.



```
// ==================== Simple I/O Module ====================
module simpleIO(
    input clk, rst,
    input [7:0] switches,
    input [31:0] cpu_data,
    input write_enable,
    output reg [7:0] leds
);
    always @(posedge clk or posedge rst) begin
        if(rst)
            leds <= 0;
        else if(write_enable)
            leds <= cpu_data[7:0];
    end
endmodule
```

Top Module (CPU Integration)

The top module integrates all the major blocks of the CPU, including the Program Counter, Instruction Memory, Control Unit, Register File, ALU, Data Memory, Immediate Generator, Performance Counter, and Simple I/O module. It connects these blocks to form the complete RV32I single-cycle processor datapath and control path. The top module also handles PC update logic, branch and jump selection, writeback selection, and halt control for ECALL and EBREAK instructions. This module represents the complete working CPU.

```
// ==================== Top Module: Complete RV32I CPU ====================
module CPU(
    input clk, rst,
    input [7:0] switches,
    input step_mode,
    input step_trigger,
    output [7:0] leds,
    output [31:0] cycle_count,
    output [31:0] instr_count,
    output [31:0] current_pc,
    output halt_flag
);
    // Wires
    wire [31:0] pc, pc_next, pc_plus4, pc_branch, pc_jump;
    wire [31:0] instr;
    wire [31:0] rd1, rd2, imm, alu_a, alu_b, alu_result, mem_data, write_data;
    wire [3:0] alu_ctrl;
    wire [1:0] branch_type, mem_size;
    wire reg_write, mem_write, mem_read, alu_src, mem_to_reg;
    wire branch, jump, jalr, mem_sign_ext, auipc_sel;
    wire ecall, ebreak, halt;
    wire zero, less_than, branch_taken;

    // Halt on ECALL/EBREAK
    assign halt = ecall | ebreak;
    assign halt_flag = halt;

    // Branch logic
    reg branch_condition;
    always @(*) begin
        case(branch_type)
            2'b00: branch_condition = zero;         // BEQ
            2'b01: branch_condition = ~zero;        // BNE
            2'b10: branch_condition = less_than;    // BLT/BLTU
            2'b11: branch_condition = ~less_than;   // BGE/BGEU
        endcase
    end

    assign branch_taken = branch & branch_condition;

```

```
    // PC logic
    assign pc_plus4 = pc + 4;
    assign pc_branch = pc + imm;
    assign pc_jump = jalr ? (alu_result & 32'hFFFFFFFE) : (pc + imm);
    assign pc_next = (jump | jalr) ? pc_jump : (branch_taken ? pc_branch : pc_plu
    assign current_pc = pc;

    // Instantiate modules
    PC pc_reg(
        .clk(clk), .rst(rst),
        .step_mode(step_mode), .step_trigger(step_trigger),
        .halt(halt),
        .pc_next(pc_next), .pc(pc)
    );

    InstructionMemory imem(.addr(pc), .instruction(instr));

    ControlUnit ctrl(
        .opcode(instr[6:0]), .funct3(instr[14:12]), .funct7(instr[31:25]),
        .imm12(instr[31:20]),
        .reg_write(reg_write), .mem_write(mem_write), .mem_read(mem_read),
        .alu_src(alu_src), .mem_to_reg(mem_to_reg),
        .branch(branch), .jump(jump), .jalr(jalr),
        .branch_type(branch_type), .mem_size(mem_size),
        .mem_sign_ext(mem_sign_ext), .auipc_sel(auipc_sel),
        .ecall(ecall), .ebreak(ebreak),
        .alu_ctrl(alu_ctrl)
    );

    RegisterFile regfile(
        .clk(clk), .we(reg_write),
        .ra1(instr[19:15]), .ra2(instr[24:20]), .wa(instr[11:7]),
        .wd(write_data), .rd1(rd1), .rd2(rd2)
    );

    ImmGen immgen(.instr(instr), .imm(imm));

    // ALU input selection
    assign alu_a = auipc_sel ? pc : rd1;
    assign alu_b = alu_src ? imm : rd2;

    ALU alu(
        .a(alu_a), .b(alu_b), .alu_ctrl(alu_ctrl),
        .result(alu_result), .zero(zero), .less_than(less_than)
    );

    DataMemory dmem(
        .clk(clk), .we(mem_write), .re(mem_read),
        .size(mem_size), .sign_ext(mem_sign_ext),
        .addr(alu_result), .wd(rd2), .rd(mem_data)
    );
```

```
    // Writeback logic
    assign write_data = (jump | jalr) ? pc_plus4 : (mem_to_reg ? mem_data : alu_result);

    // Performance Counter
    PerformanceCounter perf_counter(
        .clk(clk), .rst(rst),
        .enable(!halt && (!step_mode || step_trigger)),
        .cycle_count(cycle_count),
        .instr_count(instr_count)
    );

    // Simple I/O
    SimpleIO io_module(
        .clk(clk), .rst(rst),
        .switches(switches),
        .cpu_data(rd1),
        .write_enable(mem_write && (alu_result == 32'hFFFF0000)),
        .leds(leds)
    );

    // Monitor
    always @(posedge clk) begin
        if(!rst && (!step_mode || step_trigger) && !halt) begin
            if(ecall) $display(">>> ECALL at PC=%0d", pc);
            if(ebreak) $display(">>> EBREAK at PC=%0d", pc);
        end
    end
endmodule
```

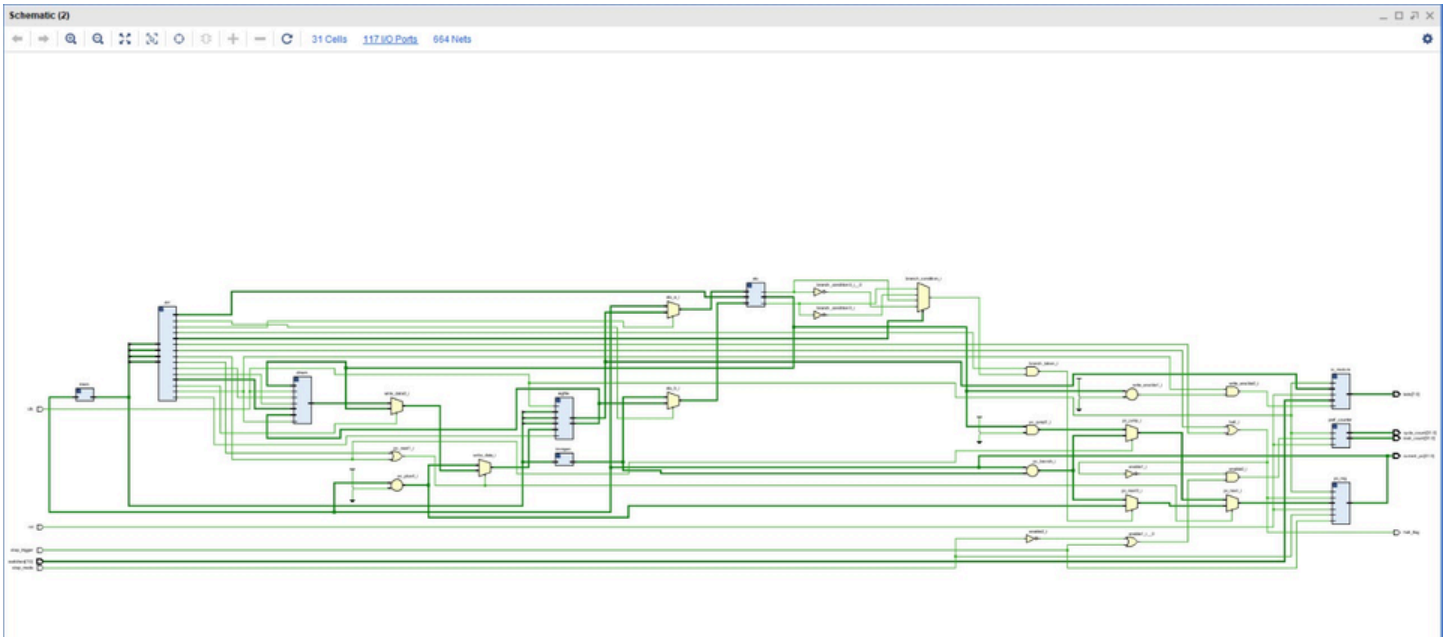
Testbench

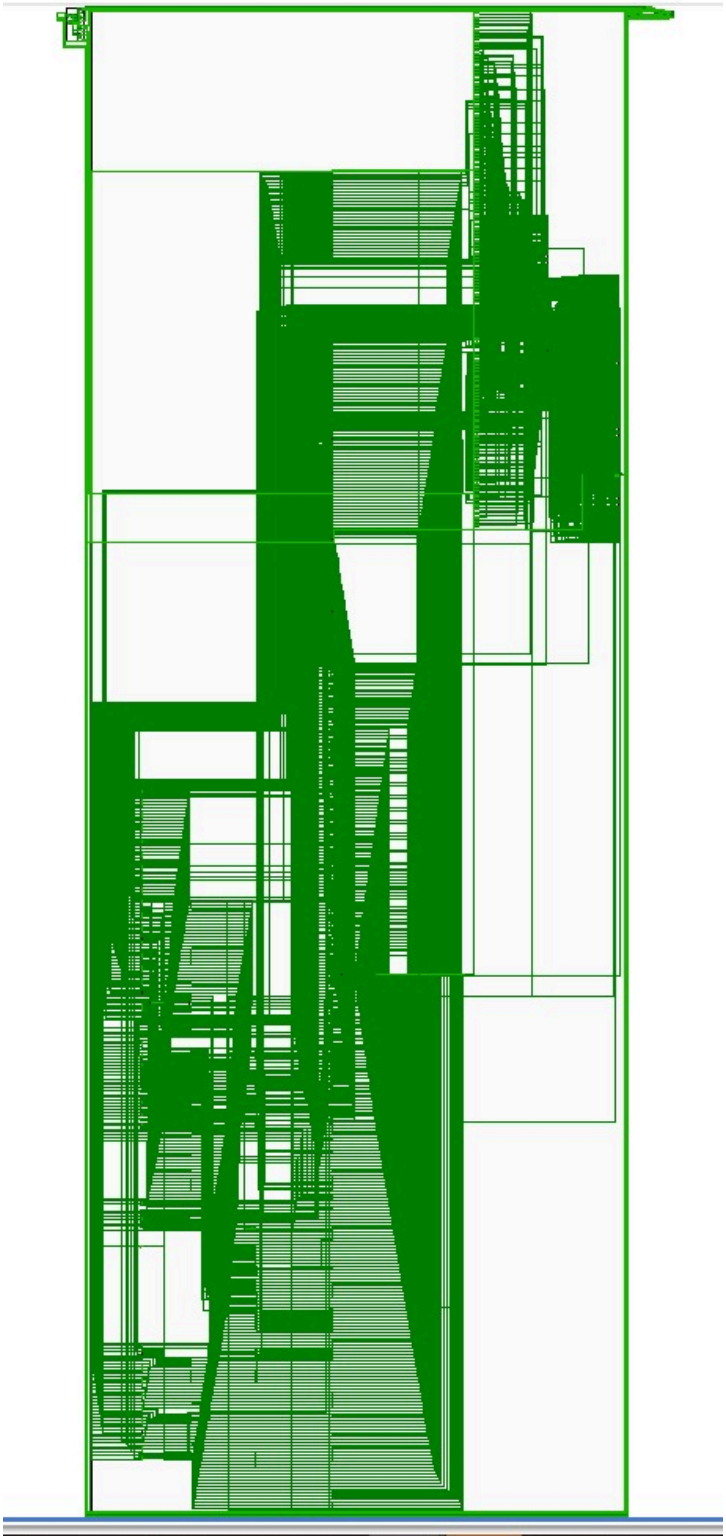
The testbench is used to verify the correct working of the CPU design through simulation. It generates the clock and reset signals and observes the internal signals such as Program Counter, register values, ALU result, and memory data. A test program is already loaded in the instruction memory, and the simulation checks whether the CPU executes all instructions correctly. The testbench helps in validating the functionality of the complete CPU before hardware implementation.

```
module tb;
    reg clk, rst;
    reg [7:0] switches;
    reg step_mode, step_trigger;
    wire [7:0] leds;
    wire [31:0] cycle_count, instr_count, current_pc;
    wire halt_flag;

    CPU cpu(
        .clk(clk), .rst(rst),
        .switches(switches),
        .step_mode(step_mode),
        .step_trigger(step_trigger),
        .leds(leds),
        .cycle_count(cycle_count),
        .instr_count(instr_count),
        .current_pc(current_pc),
        .halt_flag(halt_flag)
    );

    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    initial begin
        $dumpfile("dump.vcd");
        $dumpvars(0, tb);

        $display("\n========== Complete RV32I CPU Test ==========\n");

        rst = 1; step_mode = 0; step_trigger = 0; switches = 8'b10101010;
        #10 rst = 0;

        #500;

        $display("\n========== Performance Summary ==========");
        $display("Total Cycles:       %0d", cycle_count);
        $display("Instructions Exec:  %0d", instr_count);
        $display("CPI:                %.2f", $itor(cycle_count)/$itor(instr_count));
        $display("Halted:             %s", halt_flag ? "YES" : "NO");

        $display("\n========== Register Dump ==========");
        $display("x1  = %0d (x1 = JAL return)", cpu.regfile.regs[1]);
        $display("x3  = %0d (ADD)", cpu.regfile.regs[3]);
        $display("x4  = %0d (SUB)", cpu.regfile.regs[4]);
        $display("x5  = %0d (SLT)", cpu.regfile.regs[5]);
        $display("x8  = %0d (XOR)", cpu.regfile.regs[8]);
        $display("x9  = %0d (SLLI)", cpu.regfile.regs[9]);
        $display("x12 = %0d (SRAI)", cpu.regfile.regs[12]);
        $display("x13 = 0x%h (LUI)", cpu.regfile.regs[13]);
        $display("x14 = %0d (AUIPC)", cpu.regfile.regs[14]);
        $display("x15 = %0d (LW)", cpu.regfile.regs[15]);
        $display("x16 = %0d (LB)", cpu.regfile.regs[16]);
        $display("x17 = %0d (LBU)", cpu.regfile.regs[17]);
        $display("x19 = %0d (LH)", cpu.regfile.regs[19]);
        $display("x20 = 0x%h (LHU)", cpu.regfile.regs[20]);
        $display("x21 = %0d (after BLT)", cpu.regfile.regs[21]);
        $display("x22 = %0d (after JAL)", cpu.regfile.regs[22]);
        $display("x23 = %0d (after JALR)", cpu.regfile.regs[23]);

        $finish;
    end
endmodule
```

Schematic (Block Diagram)

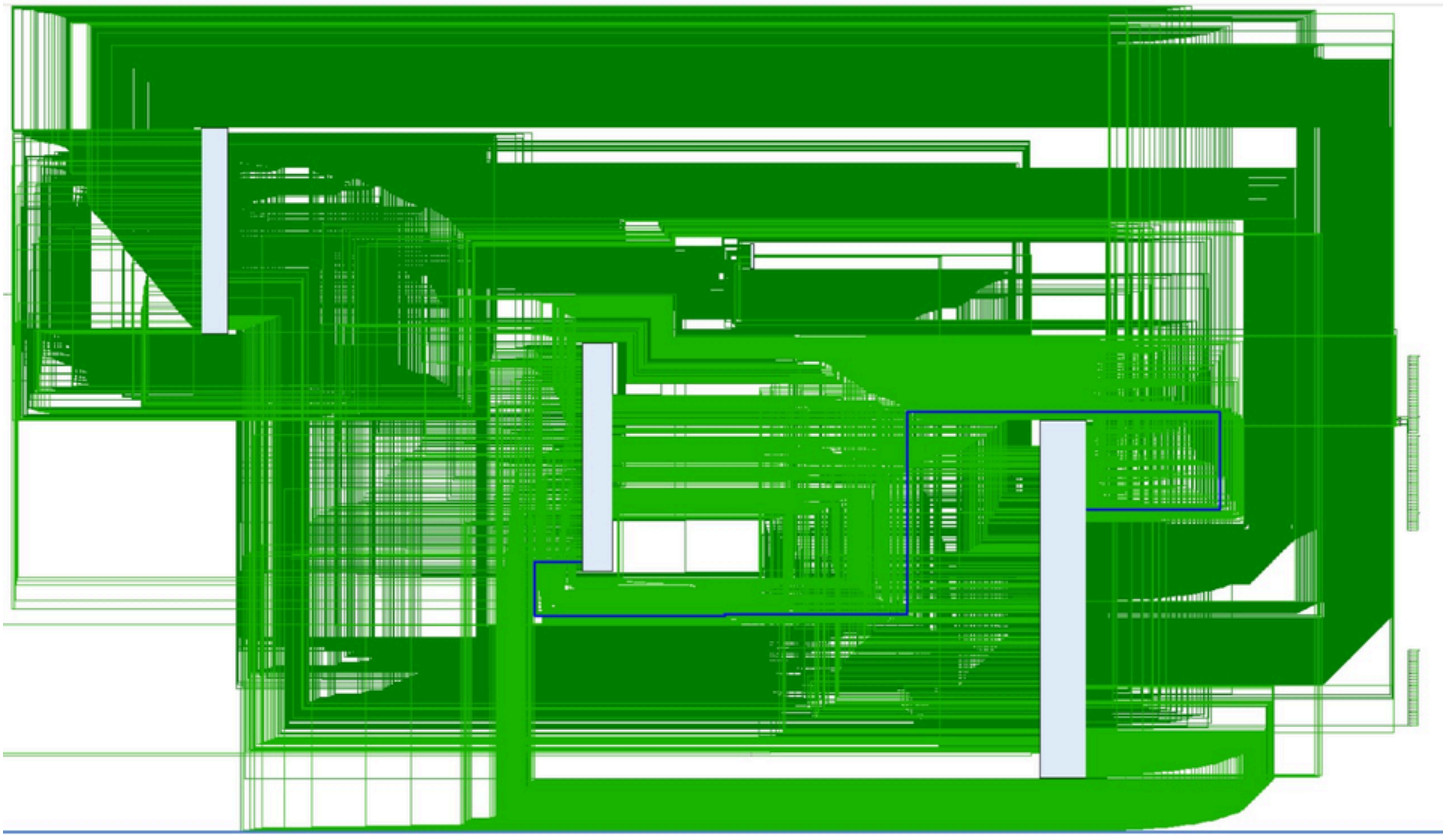
The schematic shows the overall block-level structure of the RV32I CPU and how all modules are connected with each other. It includes the Program Counter, Instruction Memory, Control Unit, Register File, ALU, Data Memory, Immediate Generator, and Writeback path. The schematic helps in understanding the data flow and control flow inside the CPU from instruction fetch to writeback.





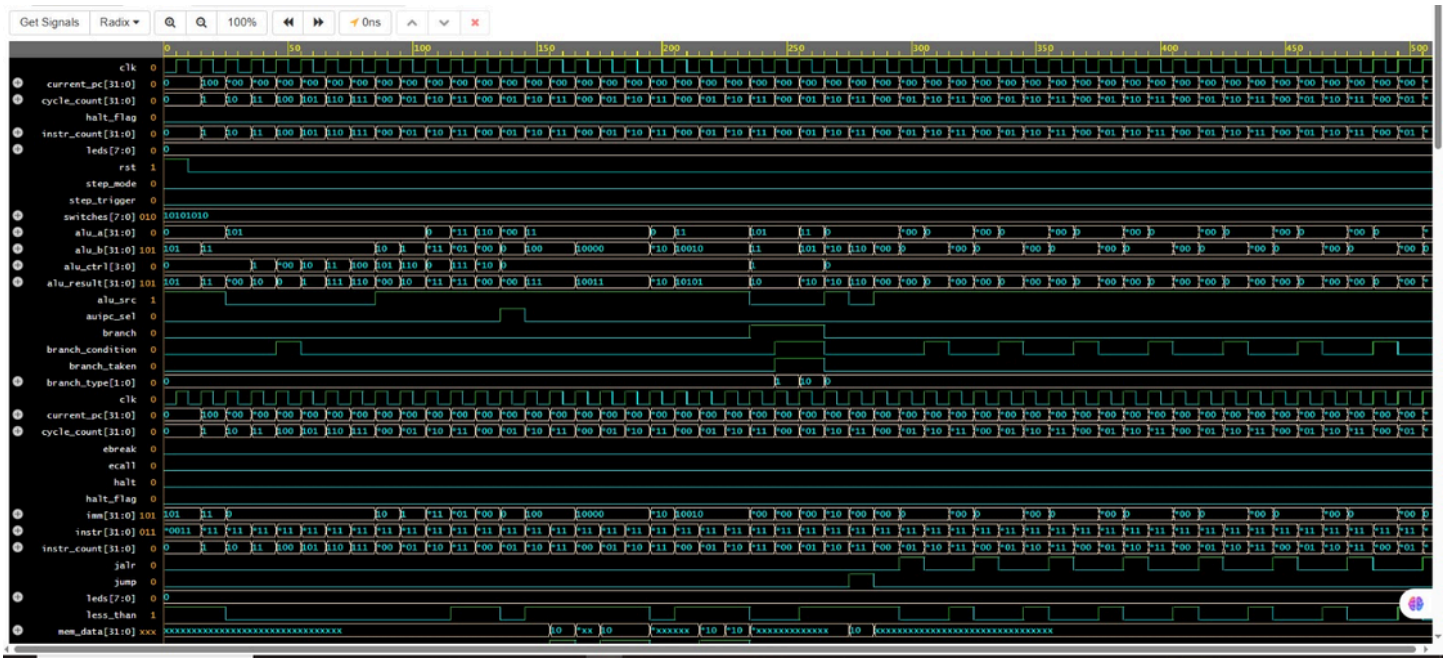
Synthesis

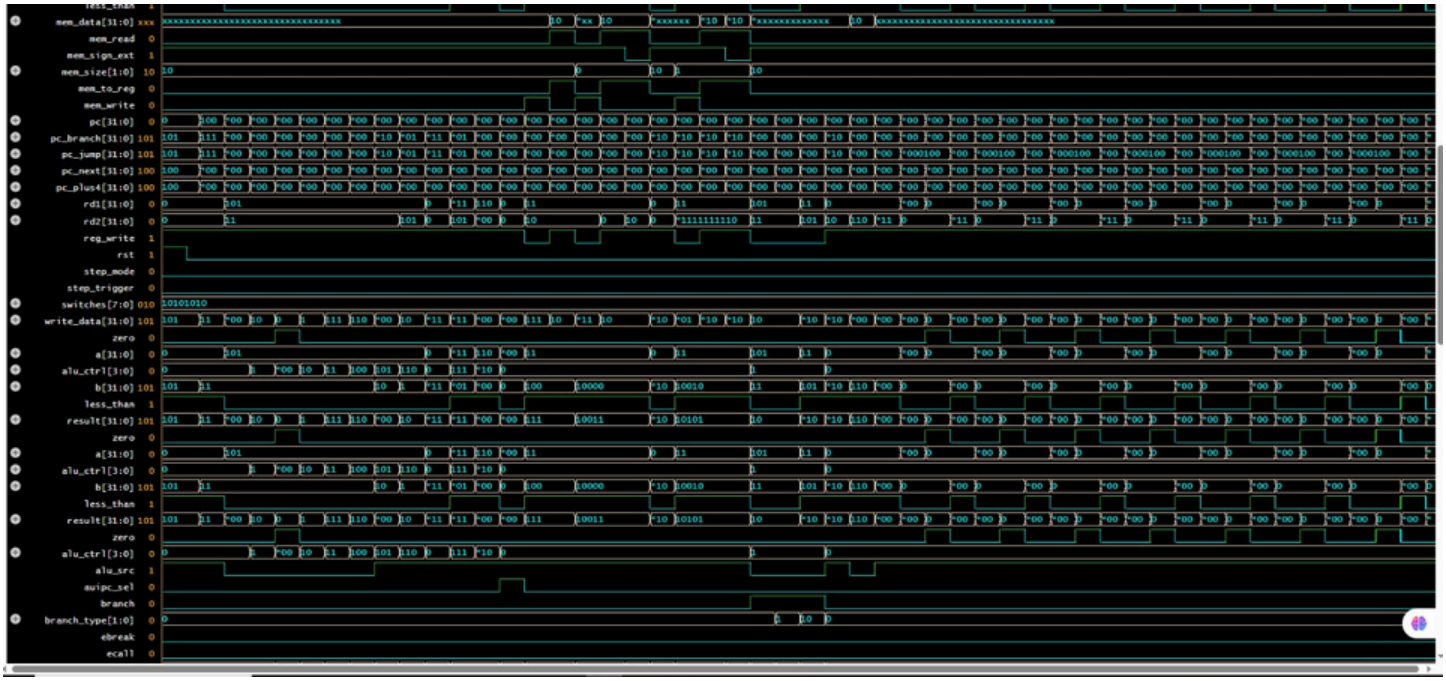
Synthesis is the process of converting the Verilog RTL design into a gate-level netlist. In this project, the CPU design is synthesized to check that the code is hardware realizable and free from synthesis errors. The synthesis report shows the usage of logic resources such as registers and combinational logic. Successful synthesis confirms that the designed RV32I CPU can be implemented on hardware like an FPGA.

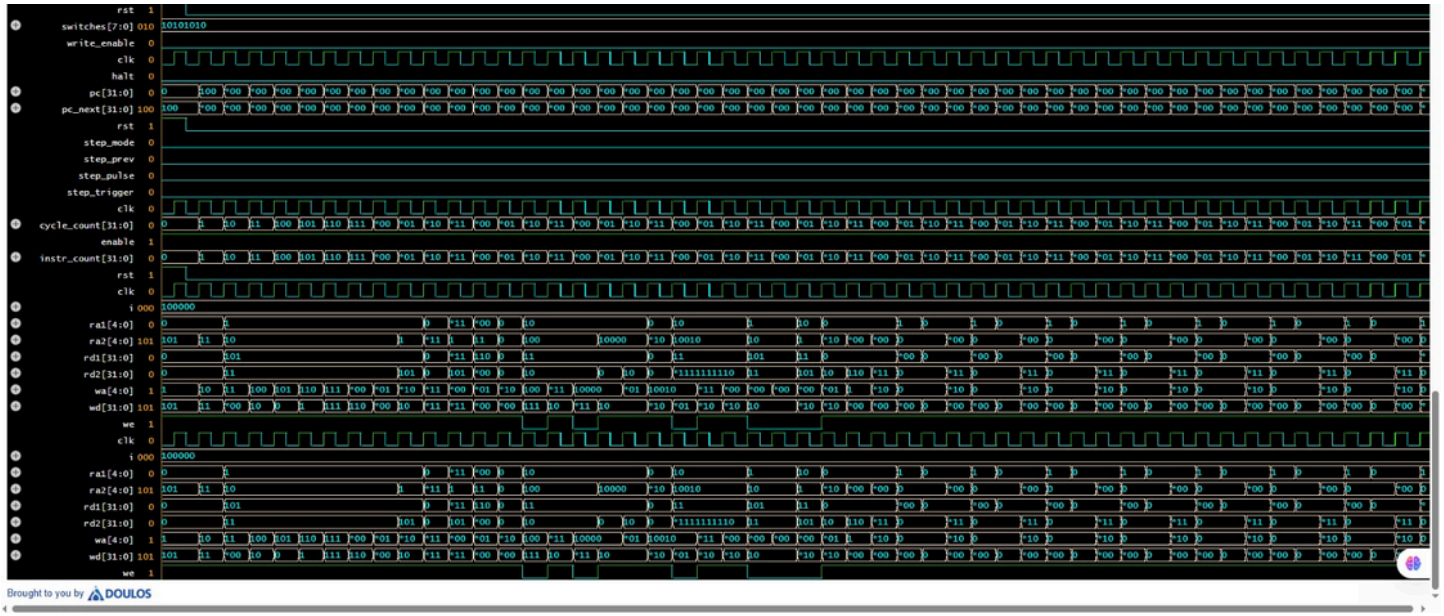


Waveform Analysis

Waveforms are used to verify the behavior of the CPU during simulation. By observing signals such as Program Counter, instruction, register write data, ALU result, and memory data, it can be confirmed that each instruction is executed correctly. The waveform view helps in debugging the design and validating the complete instruction flow from fetch to writeback.







Output

The output of the CPU is observed through simulation waveforms and the Simple I/O LED output. The CPU successfully executes the loaded program and produces correct results for arithmetic, logical, memory, branch, and jump instructions. The LED output changes according to the data written by the CPU, which confirms that the memory-mapped I/O and overall CPU operation are working correctly.

```
# KERNEL:
# KERNEL: ========== Complete RV32I CPU Test ==========
# KERNEL:
# KERNEL:
# KERNEL: ========== Performance Summary ==========
# KERNEL: Total Cycles:       50
# KERNEL: Instructions Exec:  50
# KERNEL: CPI:                1.00
# KERNEL: Halted:              NO
# KERNEL:
# KERNEL: ========== Register Dump ==========
# KERNEL: x1  = 132 (x1 = JAL return)
# KERNEL: x3  = 8 (ADD)
# KERNEL: x4  = 2 (SUB)
# KERNEL: x5  = 0 (SLT)
# KERNEL: x8  = 6 (XOR)
# KERNEL: x9  = 20 (SLLI)
# KERNEL: x12 = 4294967295 (SRAI)
# KERNEL: x13 = 0x12345000 (LUI)
# KERNEL: x14 = 52 (AUIPC)
# KERNEL: x15 = 2 (LW)
# KERNEL: x16 = 2 (LB)
# KERNEL: x17 = 2 (LBU)
# KERNEL: x19 = 4294967294 (LH)
# KERNEL: x20 = 0x0000fffe (LHU)
# KERNEL: x21 = 10 (after BLT)
# KERNEL: x22 = 12 (after JAL)
# KERNEL: x23 = 0 (after JALR)
# RUNTIME: Info: RUNTIME_0068 testbench.sv (63): $finish called.
# KERNEL: Time: 510 ns,  Iteration: 0,  Instance: /tb,  Process: @INITIAL#27_1@.
# KERNEL: stopped at time: 510 ns
# VSIM: Simulation has finished. There are no more test vectors to simulate.
# VSIM: Simulation has finished.
Finding VCD file...
./dump.vcd
[2026-01-16 06:55:09 UTC] Opening EPWave...
Done
```

Instruction Execution Flow

Example: ADD x3, x1, x2

The execution of an instruction in the CPU follows a well-defined sequence of steps. For the instruction ADD x3, x1, x2, the flow is as follows:

1. The instruction is fetched from the instruction memory using the Program Counter (PC).
2. The Control Unit decodes the instruction and generates the required control signals.
3. The Register File provides the contents of registers x1 and x2 as operands.
4. The ALU performs the addition operation on the two operands.
5. The result produced by the ALU is written back into the destination register x3.

This sequence represents the standard instruction flow from fetch to writeback in the designed CPU.

Supported Instruction Set

The designed CPU supports the complete RV32I base instruction set, which includes the following instruction categories:

- **R-type:** ADD, SUB, AND, OR, XOR, SLL, SRL, SRA, SLT, SLTU
- **I-type:** ADDI, ANDI, ORI, XORI, SLTI, SLTIU, SLLI, SRLI, SRAI
- **Load Instructions:** LB, LH, LW, LBU, LHU
- **Store Instructions:** SB, SH, SW
- **Branch Instructions:** BEQ, BNE, BLT, BGE, BLTU, BGEU
- **Jump Instructions:** JAL, JALR
- **Upper Immediate Instructions:** LUI, AUIPC
- **System Instructions:** ECALL, EBREAK

.Simulation and Verification

The complete CPU design was verified using simulation and waveform analysis. The following aspects were thoroughly tested:

- Program Counter update logic
- Register file writeback operation
- ALU operations for different instruction types
- Data memory read and write operations
- Branch and jump behavior

The simulation waveforms confirm that the CPU executes all instructions correctly and follows the expected datapath and control flow.

Results

The following results were obtained from simulation and testing:

- All RV32I instruction types were successfully executed.
- The step-by-step execution mode works correctly and allows controlled instruction execution.
- The performance counters correctly count the number of cycles and executed instructions.
- Memory access works correctly for byte, halfword, and word operations.

Limitations

Although the CPU is fully functional, it has the following limitations:

- The design is a single-cycle CPU and is not pipelined.
- There is no hazard detection or forwarding unit.
- CSR (Control and Status Register) support is not implemented.
- Interrupt and exception handling (other than ECALL/EBREAK halt) is not supported.

Future Scope

The project can be extended in several ways to make it more advanced and closer to an industry-level processor:

- Design and implementation of a pipelined RISC-V CPU
- Addition of hazard detection and forwarding unit
- Integration of cache memory
- Support for CSR registers and interrupt handling
- Integration of AXI or other external bus interfaces

Conclusion

In this project, a complete RV32I single-cycle RISC-V CPU was successfully designed and implemented using Verilog HDL. The processor supports all major instruction types including arithmetic, logical, memory, branch, jump, and system instructions. The design integrates all essential blocks such as the Program Counter, Instruction Memory, Control Unit, Register File, ALU, Data Memory, and Writeback logic into a fully functional CPU.

This project provided deep practical insight into CPU datapath design, instruction decoding, control signal generation, and RTL-level integration of multiple hardware modules. Features such as step-by-step execution mode, performance counters, and memory-mapped I/O further enhanced the usefulness of the design for debugging and learning purposes.

Overall, this work significantly strengthened the understanding of computer architecture and digital system design concepts and laid a strong foundation for developing more advanced processors such as pipelined and superscalar RISC-V cores in the future

.

References

1. RISC-V Instruction Set Architecture Specification
2. Patterson and Hennessy, *Computer Organization and Design*
3. Official RISC-V Documentation

Contact Us:

shiv sharma

linkdin id = https://www.linkedin.com/in/shiv-sharma-a73655328/

YouTube channel = https://www.youtube.com/@siliconcircle

telegram channel = https://t.me/+HT885gjxK1c0Yzll

vansh gautam

linkdin id = https://www.linkedin.com/in/vanshhgautam