EE3043 Computer Architecture

Semester 242

Design of a Single Cycle RISC-V Processor

Student: Hoàng Sỹ Nhất – 2111915

1. Introduction

Trong bối cảnh kiến trúc mã nguồn mở ngày càng định hình tương lai của điện toán hiện đại, RISC-V nổi bật nhờ tính linh hoạt, modular và khả năng tùy biến không giới hạn. Dự án này tập trung vào việc thiết kế và triển khai một lõi xử lý 32-bit dựa trên tập lệnh cơ bản RV32I của RISC-V bằng ngôn ngữ Verilog, nhằm tạo nền tảng cho các hệ thống nhúng, IoT, hoặc nghiên cứu chuyên sâu về kiến trúc máy tính. Bằng cách mô phỏng các module chính như ALU, bộ đệm thanh ghi, khối điều khiển và giao tiếp bộ nhớ, dự án không chỉ minh họa nguyên lý hoạt động của CPU RISC-V singlecycle/ pipeline mà còn góp phần vào cộng đồng phần cứng mở, đồng thời làm cơ sở để mở rộng sang tập lệnh tùy chỉnh hoặc tối ưu hiệu năng. Với việc kiểm thử nghiêm ngặt thông qua testbench và công cụ mô phỏng, dự án hướng đến một thiết kế đáng tin cậy, cân bằng giữa tính giáo dục và ứng dụng thực tiễn.

2. Design Strategy

2.1. Overview

Thiết kế CPU RISC-V single cycle được xây dựng dựa trên kiến trúc đơn chu kỳ, nơi mỗi lệnh được thực thi hoàn chỉnh trong một chu kỳ xung nhịp.

Các thành phần chính:

- **Program Counter (PC)**: Lưu trữ địa chỉ của lệnh tiếp theo cần thực thi. Tín hiệu **PC_next** xác định địa chỉ mới, có thể được cập nhật thông qua **PS_sel** (chọn giữa PC+4 hoặc địa chỉ nhảy).
- **Regfile**: Bộ thanh ghi đa dụng (32 thanh ghi), thực hiện đọc/ghi dữ liệu từ các port **rs1_addr**, **rs2_addr**, **rd_addr** với tín hiệu điều khiển **rd_wren**.
- **ALU (Arithmetic Logic Unit)**: Xử lý các phép toán số học và logic. Toán hạng được chọn thông qua **opa_sel** và **opb_sel**, kết quả trả về **alu_data**.
- **Control Unit**: Tạo các tín hiệu điều khiển (**alu_op**, **mem_wren**, **wb_sel**, **br_un**, v.v.) dựa trên opcode của lệnh (**instr**).
- **LSU (Load Store Unit)**: Quản lý truy cập bộ nhớ (**data**) và giao tiếp với **Regfile** thông qua **wb_data**.
- **I/O Module**: Xử lý giao tiếp với thiết bị ngoại vi thông qua các tín hiệu như **i_io_sw**, **i_io_btn** (đầu vào), **o_io_left**, **o_io_beta** (đầu ra).

Luồng thực thi lệnh:

- **Fetch:** Lệnh được đọc từ bộ nhớ dựa trên địa chỉ PC, truyền vào instr.
- **Decode**: Giải mã lệnh để xác định rs1_addr, rs2_addr, rd_addr, và các tín hiệu điều khiển từ Control Unit.
- **Execute:** ALU nhận rs1_data và rs2_data, thực hiện phép toán dựa trên alu_op. Kết quả (alu_data) được sử dụng cho nhánh điều kiện (br_less, br_equal) hoặc truy cập bộ nhớ.
- **Memory Access:** LSU đọc/ghi dữ liệu (data) nếu lệnh là load/store.

- **Write-back:** Dữ liệu (wb_data) được chọn từ ALU, bộ nhớ hoặc I/O thông qua wb_sel, ghi vào Regfile nếu rd_wren được kích hoạt.

Đặc điểm nổi bật:

- **Tối ưu hóa đường đi tín hiệu**: Các mạch chọn (**PS_sel**, **opa_sel**, **wb_sel**) giảm độ trễ bằng cách sử dụng multiplexer.
- **Xử lý nhánh**: Điều kiện nhảy (**br_un**, **br_less**, **br_equal**) quyết định **PC_next**, hỗ trợ cả nhánh có điều kiện và vô điều kiện.
- **Giao tiếp I/O**: Module I/O tích hợp cho phép CPU tương tác với phần cứng bên ngoài, phù hợp cho hệ thống nhúng.

Kết luận: Thiết kế đảm bảo tính đơn giản và hiệu quả, phù hợp với mục tiêu triển khai RISC-V single cycle. Các thành phần được kết nối chặt chẽ, tận dụng tối đa tín hiệu điều khiển để tối ưu hóa tốc độ xử lý. Báo cáo chi tiết sẽ phân tích sâu hơn về từng module và kết quả mô phỏng.

2.2. ALU

Bộ Arithmetic Logic Unit (ALU) là thành phần trung tâm của CPU, đảm nhận vai trò thực hiện các phép toán số học và logic cơ bản. Trong thiết kế CPU RISC-V Single Cycle, ALU nhận hai toán hạng đầu vào từ rs1_data và rs2_data, được chọn lọc thông qua các tín hiệu opa_sel và opb_sel. Tín hiệu điều khiển alu_op từ Control Unit quyết định loại phép toán cần thực thi (ví dụ: cộng, trừ, AND, OR, so sánh). Kết quả được xuất qua alu_data, phục vụ cho các thao tác tiếp theo như tính địa chỉ nhảy, truy cập bộ nhớ hoặc ghi lại vào Regfile. ALU cũng hỗ trợ xử lý các điều kiện nhánh thông qua tín

hiệu br_less và br_equal, đóng vai trò then chốt trong việc tối ưu hóa hiệu suất và độ chính xác của luồng xử lý lệnh.

2.2.1. Specification

Signal	Width	Direction	Description
i_op_a i_op_b	32 32	Input Input	Toán hạng A Toán hạng B
i_alu_op	4	Input	Lựa chọn phép toán thực hiện
o_alu_data	32	Output	Kết quả phép toán

2.2.2. Design

alu_op	Description (R-type)	Description (I-type)
ADD	$rd \leftarrow rs1 + rs2$	$rd \leftarrow rs1 + imm$
SUB	$rd \leftarrow rs1 - rs2$	n/a
SLT	$rd \leftarrow (rs1 < rs2)?\ 1 : 0$	$rd \leftarrow (rs1 < imm)?\ 1 : 0$
SLTU	$rd \leftarrow (rs1 < rs2)?\ 1 : 0$	$rd \leftarrow (rs1 < imm)?\ 1 : 0$
XOR	$rd \leftarrow rs1 \oplus rs2$	$rd \leftarrow rs1 \oplus imm$
OR	$rd \leftarrow rs1 \vee rs2$	$rd \leftarrow rs1 \vee imm$
AND	$rd \leftarrow rs1 \wedge rs2$	$rd \leftarrow rs1 \wedge imm$
SLL	$rd \leftarrow rs1 \ll rs2[4:0]$	$rd \leftarrow rs1 \ll imm[4:0]$
SRL	$rd \leftarrow rs1 \gg rs2[4:0]$	$rd \leftarrow rs1 \gg imm[4:0]$
SRA	$rd \leftarrow rs1 \ggg rs2[4:0]$	$rd \leftarrow rs1 \ggg imm[4:0]$

Dựa theo bảng câu lệnh ta chia ALU thành 4 khối chính:

- Khối cộng trừ: Sử dụng 1 bộ cộng Fulladder 32bit cùng với cổng logic Xor 32 bit để thực hiện phép cộng trừ. Dựa vào alu_op của phép cộng và trừ lựa chọn i_alu_op[3] làm bit điều khiển phép cộng từ với 0 là phép cộng và 1 là phép trừ.

```
localparam ADD = 4'b0000;
localparam SUB = 4'b1000;

// ADD+SUB
wire[31:0] temp_i_op_b;
assign temp_i_op_b = i_op_b^{32{i_alu_op[3]}};

// OP code
always @(*) begin
    case (i_alu_op[2:0])
        ADD[2:0]: o_alu_data <= i_op_a + temp_i_op_b + i_alu_op[3]; // ADD+SUB
```

- Khối logic: sử dụng các cổng logic cơ bản để thực hiện phép and, or, xor đồng thời tận dụng logic and để thực hiện câu lệnh LUI tức là i_op_a and với 1 để ngõ ra giữ nguyên là i_op_a (đặt i_alu_op cho câu lệnh LUI là 4'b1111).

```
localparam XOR = 4'b0100;
localparam OR = 4'b0110;
localparam AND = 4'b0111;
localparam LUI = 4'b1111; // use and with 32'hFFFF_FFFF

XOR[2:0]: o_alu_data <= i_op_a ^ i_op_b;
AND[2:0]: o_alu_data <= (i_op_a | {32{i_alu_op[3]}}) & i_op_b;
OR[2:0]: o_alu_data <= i_op_a | i_op_b;
```

- Khối so sánh: Sử dụng kết hợp ~(A ^ B) để so sánh giống nhau giữa các bit và ~A & B để so sánh A < B từ đó xây dựng một bộ so sánh không dấu. Khi đã có kết quả so sánh không dấu cho A và B để so sánh có dấu thay vì sử dụng bộ cộng trừ thì ta có thể đon giản bằng cách sử dụng bit cuối của 2 toán hạng để so sánh bằng phép logic sau A_lt_B ^ (~A[31] & B[31]) ^ (A[31] & ~B[31]) . Ví dụ với 2 trường hợp sau:

101 < 010 (-3 < 2)	010 !< 101 (2 !< -3)
~A&B=010	~A&B=101
~A^B=000	~A^B=000
lt_temp=010	lt_temp=101
o_alu_data=0^0^1=1	o_alu_data=1^1^0=0

```
// Set less than A<B
wire A_lt_B;
wire [31:0] A_xor_B; // XOR result for each bit
wire [31:0] not_A_and_B; // (NOT A) AND B for each bit
genvar i;
generate
    for (i = 0; i < 32; i = i + 1) begin : bit_comparison
        assign A_xor_B[i] = i_op_a[i] ^ i_op_b[i]; // XOR for equality check
        assign not_A_and_B[i] = ~i_op_a[i] & i_op_b[i]; // A < B for this bit
    end
endgenerate

// assign A_eq_B = &(~A_xor_B); // AND of all XOR results (0 if any bit differs)

// Less than check (A < B)
wire [31:0] lt_temp;
assign lt_temp[0] = not_A_and_B[0];
generate
    for (i = 1; i < 32; i = i + 1) begin : lt_check
        assign lt_temp[i] = not_A_and_B[i] | (lt_temp[i-1] & ~A_xor_B[i]);
    end
endgenerate

assign A_lt_B = lt_temp[31];


SLTU[2:0]:  o_alu_data <= {31'd0, A_lt_B};
SLT[2:0]: o_alu_data <= {31'd0, (A_lt_B ^ (~i_op_a[31] & i_op_b[31]) ^ (i_op_a[31] & ~i_op_b[31]))};
```

- Khối shift bit: Sử dụng cấu trúc Barrel shifter để làm bộ shift phải và trái. Đối với lệnh shift SRA nếu bit đầu của i_op_a là 1 thì thêm bit 1 khi shift.

```
// Shift right arithmetic
wire sign_bit_sra;
assign sign_bit_sra = i_op_a[31]&i_alu_op[3];
```

```
localparam SLL = 4'b0001;
localparam SRL = 4'b0101;
localparam SRA = 4'b1101;
```

```
SRL[2:0]: begin // SRL and SRA
    case (i_op_b[4:0])
        1: o_alu_data <= {{1{sign_bit_sra}}, i_op_a[31:1]};
        2: o_alu_data <= {{2{sign_bit_sra}}, i_op_a[31:2]};
        3: o_alu_data <= {{3{sign_bit_sra}}, i_op_a[31:3]};
        4: o_alu_data <= {{4{sign_bit_sra}}, i_op_a[31:4]};
        5: o_alu_data <= {{5{sign_bit_sra}}, i_op_a[31:5]};
        6: o_alu_data <= {{6{sign_bit_sra}}, i_op_a[31:6]};
        7: o_alu_data <= {{7{sign_bit_sra}}, i_op_a[31:7]};
        8: o_alu_data <= {{8{sign_bit_sra}}, i_op_a[31:8]};
```

2.3. BRU

Bộ BRC (Branch Condition Unit) là thành phần quan trọng trong CPU RISC-V Single Cycle, chuyên xử lý các điều kiện nhánh để quyết định việc thay đổi luồng thực thi lệnh. Dựa trên các tín hiệu đầu vào như br_un (xác định phép so sánh không dấu), br_less, và br_equal (kết quả so sánh từ ALU), BRC đánh giá điều kiện nhảy (ví dụ: beq, bne, blt, bge). Kết quả từ BRC kết hợp với PS_sel (tín hiệu chọn địa chỉ nhảy) giúp xác định PC_next (địa chỉ lệnh tiếp theo), hỗ trợ cả nhánh có điều kiện và vô điều kiện. Bộ này đảm bảo tính chính xác của việc rẽ nhánh, tối ưu hiệu suất bằng cách giảm độ trễ trong quyết định cập nhật Program Counter, đồng thời tích hợp chặt chẽ với ALU và Control Unit để duy trì luồng xử lý lệnh liền mạch.

2.3.1. Specification

Signal	**Width**	**Direction**	**Description**
i_rs1_data	32	Input	Dữ liệu thanh ghi 1
i_rs2_data	32	Input	Dữ liệu thanh ghi 2
i_br_un	1	Input	Chọn chế độ so sánh có dấu hoặc không dấu
o_br_less	1	Output	Bằng 1 nếu rs1 < rs2
o_br_equal	1	Output	Bằng 1 nếu rs1 = rs2

2.3.2. Design

Thiết kế tượng từ khối so sánh ở ALU nhưng thêm so sánh bằng bằng cách And tất cả bit ~(A ^ B) nếu có ít nhất 1 bit khác nhau thì o_br_equal = 0.

```
// Set less than A<B
wire [31:0] A_xor_B; // XOR result for each bit
wire [31:0] not_A_and_B; // (NOT A) AND B for each bit
genvar i;
generate
    for (i = 0; i < 32; i = i + 1) begin : bit_comparison
        assign A_xor_B[i] = i_rs1_data[i] ^ i_rs2_data[i]; // XOR for equality check
        assign not_A_and_B[i] = ~i_rs1_data[i] & i_rs2_data[i]; // A < B for this bit
    end
endgenerate

assign o_br_equal = &(~A_xor_B); // AND of all XOR results (0 if any bit differs)

// Less than check (A < B)
wire [31:0] lt_temp;
assign lt_temp[0] = not_A_and_B[0];
generate
    for (i = 1; i < 32; i = i + 1) begin : lt_check
        assign lt_temp[i] = not_A_and_B[i] | (lt_temp[i-1] & ~A_xor_B[i]);
    end
endgenerate

assign o_br_less = lt_temp[31] ^ (~i_br_un & (~i_rs1_data[31] & i_rs2_data[31])) ^ (~i_br_un & (i_rs1_data[31] & ~i_rs2_data[
```

2.4. Regfile

Regfile (Register File) là tập hợp 32 thanh ghi đa dụng (32-bit) trong kiến trúc RISC-V, đóng vai trò lưu trữ dữ liệu tạm thời phục vụ cho quá trình xử lý lệnh. Mỗi lệnh thường truy cập hai thanh ghi nguồn (rs1_addr, rs2_addr) để đọc dữ liệu (rs1_data, rs2_data) và một thanh ghi đích (rd_addr) để ghi kết quả (rd_data) khi tín hiệu rd_wren được kích hoạt.

Regfile tối ưu hóa tốc độ xử lý bằng cách cung cấp dữ liệu trực tiếp cho ALU và các đơn vị khác mà không cần truy cập bộ nhớ chính. Đặc biệt, thanh ghi x0 luôn giữ giá trị 0 và không thể ghi đè, hỗ trợ các phép toán cố định và đơn giản hóa lệnh. Với thiết kế đa port (đọc/ghi đồng thời), Regfile đảm bảo hiệu suất cao, là thành phần không thể thiếu trong pipeline xử lý lệnh của CPU RISC-V.

2.4.1. Specification

Signal	**Width**	**Direction**	**Description**
i_clk	1	Input	Clock
i_reset	1	Input	Tín hiệu reset (High Active)

i_rs1_addr	5	Input	Địa chỉ thanh ghi thứ nhất
i_rs2_addr	5	Input	Địa chỉ thanh ghi thứ hai
o_rs1_data	32	Output	Dữ liệu thanh ghi 1
o_rs2_data	32	Output	Dữ liệu thanh ghi 2
i_rd_addr	5	Input	Địa chỉ thanh ghi đích
i_rd_data	32	Input	Dữ liệu thanh ghi đích
i_rd_wren	1	Input	Tín hiệu cho phép ghi dữ liệu và thanh ghi đích

2.4.2. Design

Thiết kế thanh ghi 32x32-bit theo cấu trúc 2 Read 1 Write, dữ liệu chỉ ghi vào thanh ghi đích khi có xung clock cạnh lên và thanh ghi địa chỉ 0 (zero) luôn nối xuống đất.

2.5. LSU

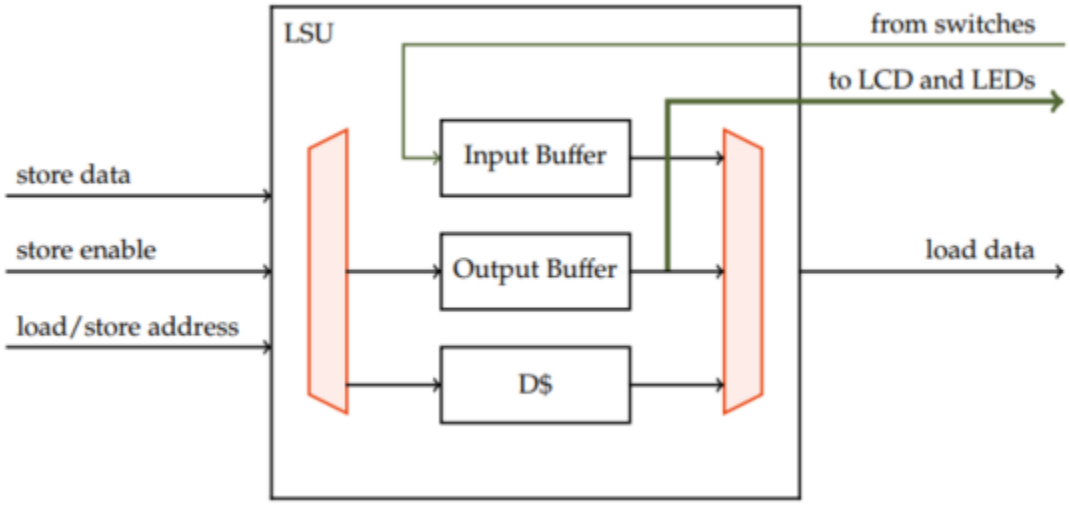
LSU (Load Store Unit) là thành phần chuyên trách quản lý việc truy cập bộ nhớ trong CPU RISC-V Single Cycle, thực thi các lệnh load (đọc dữ liệu từ bộ nhớ) và store (ghi dữ liệu vào bộ nhớ). Đơn vị này nhận địa chỉ bộ nhớ từ alu_data, dữ liệu từ rs2_data (cho lệnh store), và điều khiển bằng tín hiệu mem_wren (ghi nhớ) từ Control Unit.
Khi thực hiện lệnh load, LSU đọc dữ liệu từ bộ nhớ (data) và trả kết quả qua wb_data để ghi vào Regfile. Với lệnh store, LSU ghi rs2_data vào bộ nhớ tại địa chỉ đã tính toán. LSU đảm bảo độ chính xác và hiệu suất trong việc kết nối CPU với bộ nhớ, đồng thời tối ưu hóa luồng xử lý bằng cách phối hợp chặt chẽ với ALU và Control Unit, đặc biệt quan trọng trong các ứng dụng yêu cầu truy cập dữ liệu thường xuyên.

2.5.1. Specification

Tín hiệu	**Độ rộng**	**Hướng**	**Mô tả**
i_clk	1	Input	Xung nhịp toàn cục
i_reset	1	Input	Tín hiệu reset đồng bộ, tích cực mức cao
i_lsu_addr	32	Input	Địa chỉ truy cập bộ nhớ (căn chỉnh theo từ)
i_st_data	32	Input	Dữ liệu cần ghi vào bộ nhớ
i_lsu_wren	1	Input	Tín hiệu cho phép ghi (1 = ghi, 0 = đọc)
o_ld_data	32	Output	Dữ liệu đọc từ bộ nhớ (mở rộng dấu/không dấu tùy theo i_ld_un)
i_lsu_op	2	Input	Kích thước truy cập: 00=word, 10=half-word, 11=byte
i_ld_un	1	Input	Loại mở rộng khi đọc: 0=có dấu, 1=không dấu
i_pc	32	Input	Bộ đếm chương trình (dùng để lấy lệnh)
o_instr	32	Output	Lệnh được lấy về (đầu ra có thanh ghi)
o_io_ledr	32	Output	Thanh ghi điều khiển đèn LED đỏ
o_io_ledg	32	Output	Thanh ghi điều khiển đèn LED xanh lá
o_io_hex0-o_io_hex7	7	Output	Các đèn 7 đoạn (HEX0-HEX7, các đoạn tích cực mức thấp)
o_io_lcd	32	Output	Thanh ghi điều khiển LCD
i_io_sw	32	Input	Đầu vào từ các công tắc (lấy mẫu trực tiếp)

2.5.2. *Design*

Thiết kế bộ LSU giống hình trên và map Memory và các buffer io về các địa chỉ như hình dưới.



Base address	Top address	Mapping
0x1001_1000	0xFFFF_FFFF	(Reserved)
0x1001_0000	0x1001_0FFF	Switches *(required)*
0x1000_5000	0x1000_FFFF	(Reserved)
0x1000_4000	0x1000_4FFF	LCD Control Registers
0x1000_3000	0x1000_3FFF	Seven-segment LEDs 7-4
0x1000_2000	0x1000_2FFF	Seven-segment LEDs 3-0
0x1000_1000	0x1000_1FFF	Green LEDs *(required)*
0x1000_0000	0x1000_0FFF	Red LEDs *(required)*
0x0000_0800	0x0FFF_FFFF	(Reserved)
0x0000_0000	0x0000_07FF	Memory (2KiB) *(required)*

2.6. Datapath

2.6.1. *Specification*

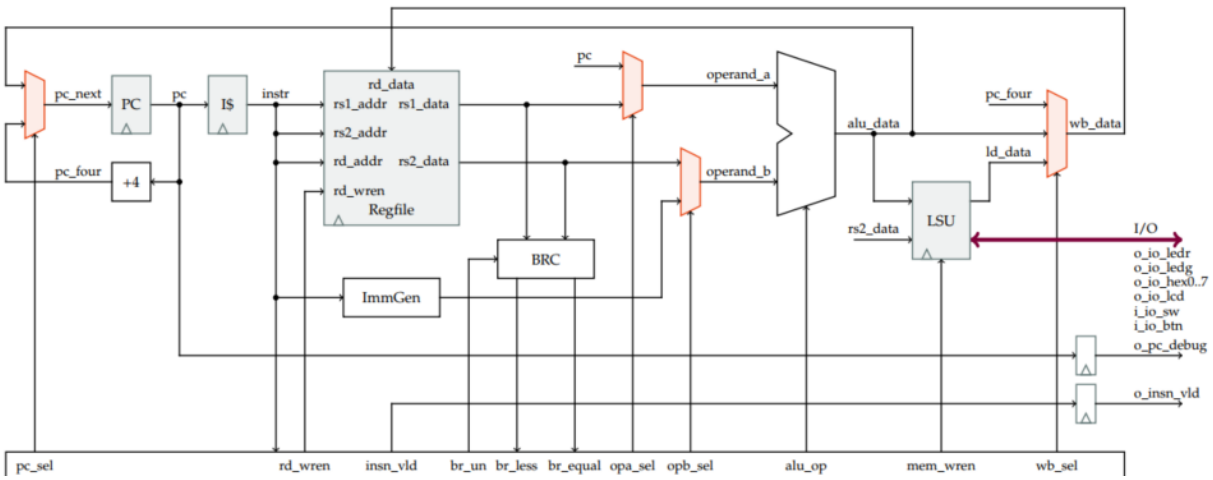
Signal	Width	Direction	Description
i_clk	1	Input	Clock

i_reset	1	Input	Tín hiệu reset (Active High)
i_pc_sel	1	Input	Lựa chọn giá trị PC (0: PC+4, 1: Branch/Jump target)
o_instr	32	Output	Instruction hiện tại
i_insn_vld	1	Input	Tín hiệu instruction hợp lệ
o_insn_vld	1	Output	Xác nhận instruction hợp lệ (debug)
i_imm_sel	3	Input	Lựa chọn kiểu immediate (I-type, S-type, B-type, U-type, J-type)
i_opa_sel	1	Input	Lựa chọn toán hạng A (0: RS1, 1: PC)
i_opb_sel	1	Input	Lựa chọn toán hạng B (0: RS2, 1: Immediate)
i_alu_op	4	Input	Operation code ALU (ADD, SUB, AND, OR, XOR, SLT, etc.)
i_br_un	1	Input	Loại so sánh branch (0: signed, 1: unsigned)
o_br_less	1	Output	Kết quả so sánh "RS1 < RS2"
o_br_equal	1	Output	Kết quả so sánh "RS1 == RS2"
i_mem_wren	1	Input	Tín hiệu ghi bộ nhớ (Active High)
i_lsu_op	2	Input	Loại load/store (00: word, 10: half-word, 11: byte)
i_ld_un	1	Input	Loại load (0: signed, 1: unsigned)
i_wb_sel	2	Input	Lựa chọn dữ liệu writeback (00: ALU, 01: Memory, 10: PC+4)

i_rd_wren	1	Input	Tín hiệu ghi thanh ghi đích (Active High)
o_io_ledr	32	Output	Điều khiển LED đỏ
o_io_ledg	32	Output	Điều khiển LED xanh
o_io_hex0- o_io_hex7	7	Output	Điều khiển 7-segment (HEX0-HEX7)
o_io_lcd	32	Output	Điều khiển LCD
i_io_sw	32	Input	Đọc trạng thái công tắc
o_pc_debug	32	Output	Giá trị PC (debug)

2.6.2. Design

Ghép các khối giống như hình dưới đây.



2.7. Control Unit

Control Unit (Bộ Điều Khiển) là thành phần trung tâm của CPU RISC-V, đảm nhiệm việc giải mã lệnh (**instr**) và tạo các tín hiệu điều khiển để phối hợp hoạt động của toàn

bộ hệ thống. Dựa trên opcode và các trường chức năng trong lệnh, Control Unit xác định:

- Loại phép toán ALU thực hiện thông qua **alu_op**.
- Việc ghi/đọc bộ nhớ bằng **mem_wren** (kích hoạt LSU cho lệnh load/store).
- Nguồn dữ liệu ghi vào Regfile (**wb_sel**) và tín hiệu cho phép ghi (**rd_wren**).
- Toán hạng cho ALU qua **opa_sel** và **opb_sel** (chọn từ PC, Regfile hoặc hằng số).
- Điều kiện rẽ nhánh (**br_un**, kết hợp với **br_less**, **br_equal** từ ALU) và chọn địa chỉ lệnh tiếp theo (**PS_sel** để cập nhật **PC_next**).

Control Unit đảm bảo các lệnh như số học (add, sub), logic (AND, OR), nhảy có điều kiện (beq, bne), hay truy cập bộ nhớ (lw, sw) được thực thi chính xác. Với thiết kế tối ưu, nó tối giản độ trễ bằng cách sinh tín hiệu đồng bộ, giúp CPU RISC-V Single Cycle vận hành hiệu quả, tuân thủ chặt chẽ đặc tả ISA.

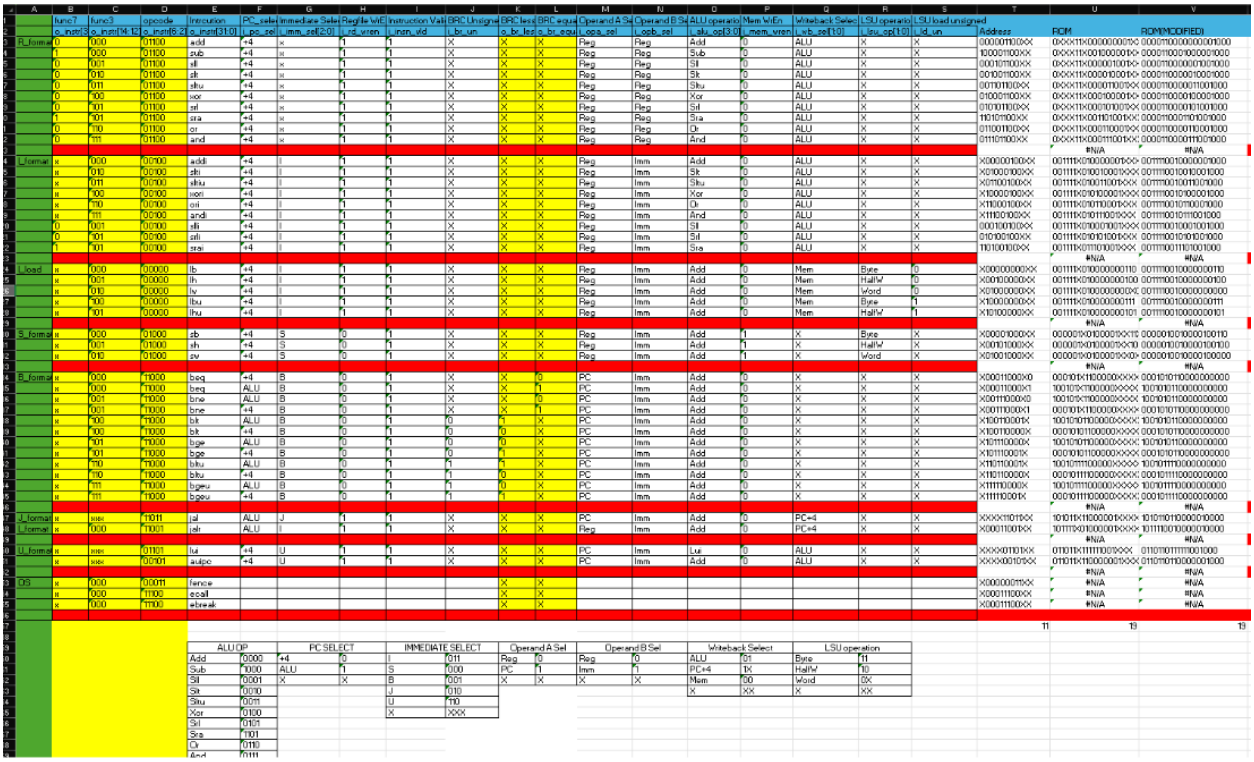
2.7.1. Specification

Signal	**Width**	**Direction**	**Description**
i_clk	1	Input	Clock signal
i_reset	1	Input	Active-high reset signal
i_instr	32	Input	Current instruction to decode
o_imm_sel	3	Output	Immediate type selector (I/S/B/U/J-type)
o_insn_vld	1	Output	Valid instruction indicator
o_pc_sel	1	Output	PC source selector (0: PC+4, 1: Branch target)
o_br_un	1	Output	Branch comparison type (0: signed, 1: unsigned)

i_br_less	1	Input	RS1 < RS2 comparison result
i_br_equal	1	Input	RS1 == RS2 comparison result
o_opa_sel	1	Output	Operand A selector (0: RS1, 1: PC)
o_opb_sel	1	Output	Operand B selector (0: RS2, 1: Immediate)
o_alu_op	4	Output	ALU operation code (e.g., ADD, SUB, AND, OR)
o_mem_wren	1	Output	Memory write enable (Active-high)
o_lsu_op	2	Output	Load/store size (00: word, 10: half-word, 11: byte)
o_ld_un	1	Output	Load type (0: sign-extended, 1: zero-extended)
o_wb_sel	2	Output	Writeback data source (00: ALU, 01: Memory, 10: PC+4)
o_rd_wren	1	Output	Register file write enable (Active-high)

2.7.2. Design

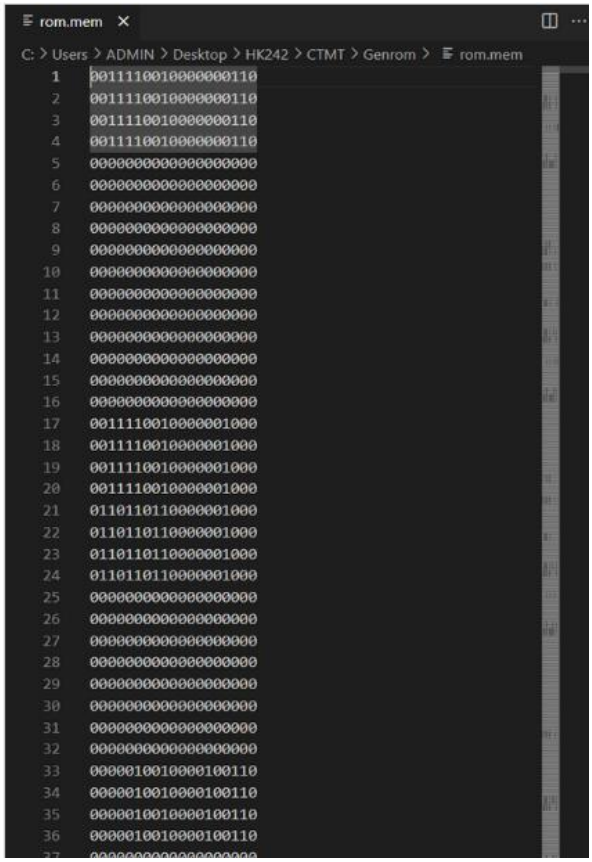
Để đơn giản hóa thiết kế và dễ dàng nâng cấp thêm hoặc thay đổi câu lệnh ta dùng ROM để thiết kế control unit. Từ bảng tập lệnh ISA RiscV ta lập được bảng sau:



ALU OP		PC SELECT		IMMEDIATE SELECT		Operand A Sel		Operand B Sel		Writeback Select		LSU operation	
Add	0000	+4	0	I	011	Reg	0	Reg	0	ALU	01	Byte	11
Sub	0000	ALU	1	S	000	PC	1	Imm	1	PC+4	1X	HalfW	10
Sll	0001	X	X	B	001	X	X	X	X	Mem	00	Word	0X
Slt	0010			J	010					X	XX	X	XX
Sltu	0011			U	110								
Xor	0100			X	XXX								
Srl	0101												
Sra	1101												
Or	0110												
And	0111												

Từ đó ta có được địa chỉ của ROM tương ứng với tổ hợp logic của các câu lệnh và ngõ ra bộ so sánh BRC (các ô tô màu vàng) và dữ liệu tại địa chỉ đó ứng với tín hiệu logic điều khiển các khối chức năng. Ở phần địa chỉ có nhưng bit tùy định để cho control unit hoạt động đúng với mọi trường hợp này ta phải tạo ROM bao quát hết các trường hợp tùy định này ta viết một đoạn script ngắn để tạo dữ liệu cho ROM.

```python
import os
import numpy as np

lenAddr = 11

with open('addr.txt', 'r') as file:
    Addrarray = [line.strip() for line in file]
with open('mem.txt', 'r') as file:
    Memarray = [line.strip() for line in file]


# print(Memarray)
# print(Addrarray)

f = open('rom.mem', 'w')

for i in range(2048):
    tempaddr = np.binary_repr(i, width=lenAddr)
    invalid = 1
    doub = 0
    for k, addr in enumerate(Addrarray):
        count = 0
        for j, t in enumerate(tempaddr):
            if(t == addr[j] or addr[j] == 'X'):
                count += 1
        if (count == lenAddr):
            f.write(Memarray[k]+'\n')
            invalid = 0
            if (doub>0):
                print(tempaddr)
                print(addr)
            doub +=1
            break
    if (invalid):
        f.write(np.binary_repr(0, width=19)+'\n')
f.close()
```



```
rom.mem
C: > Users > ADMIN > Desktop > HK242 > CTMT > Genrom > rom.mem
1   0011110010000000110
2   0011110010000000110
3   0011110010000000110
4   0011110010000000110
5   0000000000000000000
6   0000000000000000000
7   0000000000000000000
8   0000000000000000000
9   0000000000000000000
10  0000000000000000000
11  0000000000000000000
12  0000000000000000000
13  0000000000000000000
14  0000000000000000000
15  0000000000000000000
16  0000000000000000000
17  0011110010000001000
18  0011110010000001000
19  0011110010000001000
20  0011110010000001000
21  0110110110000001000
22  0110110110000001000
23  0110110110000001000
24  0110110110000001000
25  0000000000000000000
26  0000000000000000000
27  0000000000000000000
28  0000000000000000000
29  0000000000000000000
30  0000000000000000000
31  0000000000000000000
32  0000000000000000000
33  0000010010000100110
34  0000010010000100110
35  0000010010000100110
36  0000010010000100110
```

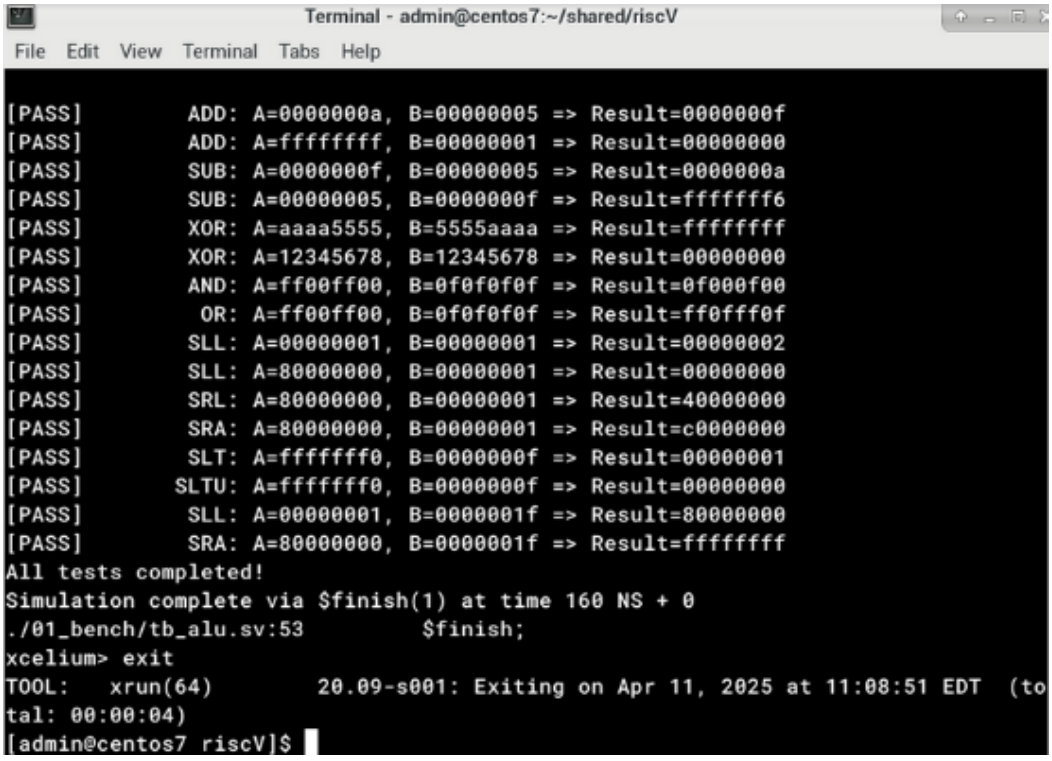
3. Verification Strategy

3.1. ALU

Lệnh	Mô tả Test Case	Input (A, B)	Kết Quả Mong Đợi	Kết Quả Thực Tế	Trạng Thái
ADD	Cộng hai số dương	0x0000000a, 0x00000005	0x0000000f	0x0000000f	PASS
ADD	Cộng tràn số không dấu	0x0fffffff, 0x00000001	0x00000000	0x00000000	PASS

SUB	Trừ hai số dương	0x0000000f, 0x00000005	0x0000000a	0x0000000a	PASS
SUB	Trừ số nhỏ hơn (kết quả âm)	0x00000005, 0x0000000f	0xfffffff6	0xfffffff6	PASS
XOR	XOR hai giá trị khác nhau	0xaaaa5555, 0x5555aaaa	0xffffffff	0xffffffff	PASS
XOR	XOR hai giá trị giống nhau	0x12345678, 0x12345678	0x00000000	0x00000000	PASS
AND	AND hai bitmask	0xff00ff00, 0x0f0ff0ff	0x0f000f00	0x0f000f00	PASS
OR	OR hai bitmask	0xff00ff00, 0x0f0ff0ff	0xffffffff	0xffffffff	PASS
SLL	Dịch trái 1 bit (số dương)	0x00000001, 0x00000001	0x00000002	0x00000002	PASS
SLL	Dịch trái 1 bit (tràn)	0x80000000, 0x00000001	0x00000000	0x00000000	PASS
SRL	Dịch phải logic 1 bit	0x80000000, 0x00000001	0x40000000	0x40000000	PASS
SRA	Dịch phải số học 1 bit (giữ dấu)	0x80000000, 0x00000001	0x00000000	0x00000000	PASS

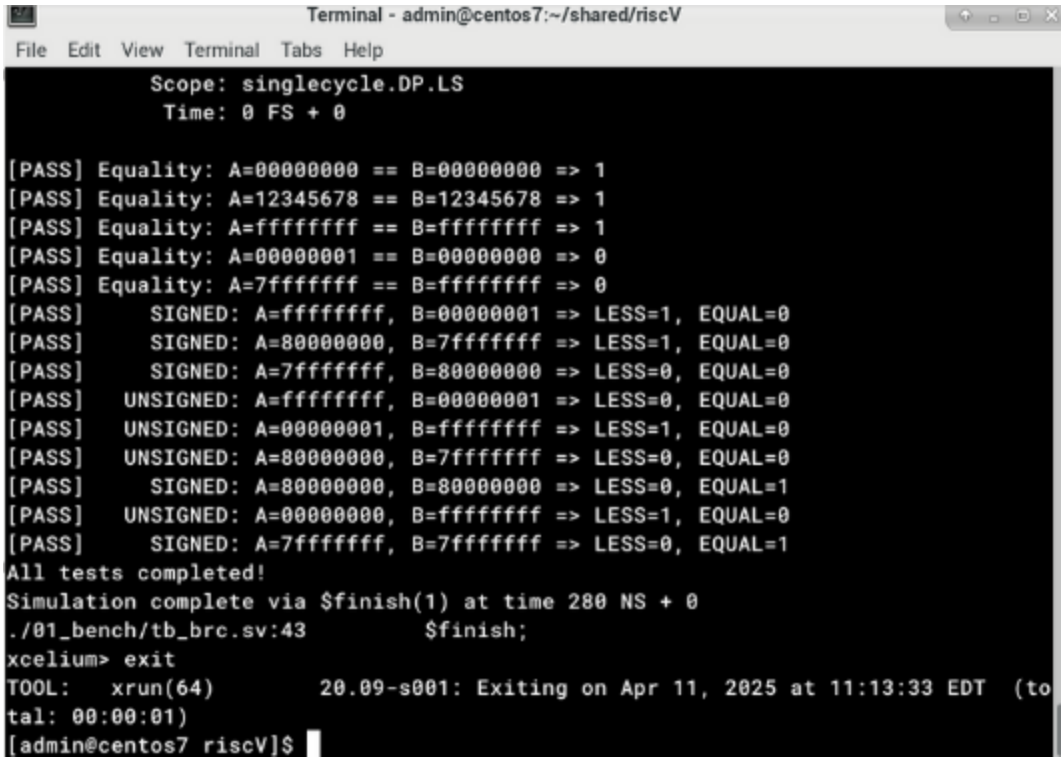
SLT	So sánh nhỏ hơn (có dấu)	0xfffffff0, 0x0000000f	0x00000001	0x00000001	PASS
SLTU	So sánh nhỏ hơn (không dấu)	0xfffffff0, 0x0000000f	0x00000000	0x00000000	PASS
SLL	Dịch trái 31 bit	0x00000001, 0x0000001f	0x80000000	0x80000000	PASS
SRA	Dịch phải số học 31 bit (giữ dấu)	0x80000000, 0x0000001f	0xffffffff	0xffffffff	PASS



3.2. BRC

Kiểm Tra	Mô tả Test Case	Input (A, B)	Kết Quả Mong Đợi	Kết Quả Thực Tế	Trạng Thái
Equality	So sánh bằng nhau (A = 0, B = 0)	0x00000000, 0x00000000	Equal = 1	Equal = 1	PASS
Equality	So sánh bằng nhau (A = 0x12345678, B = 0x12345678)	0x12345678, 0x12345678	Equal = 1	Equal = 1	PASS
Equality	So sánh bằng nhau (A = 0xffffffff, B = 0xffffffff)	0xffffffff, 0xffffffff	Equal = 1	Equal = 1	PASS
Equality	So sánh không bằng nhau (A = 0x00000001, B = 0x00000000)	0x00000001, 0x00000000	Equal = 0	Equal = 0	PASS
SIGNED	So sánh có dấu (A = -1, B = 1)	0xffffffff, 0x00000001	LESS = 1, EQUAL = 0	LESS = 1, EQUAL = 0	PASS
SIGNED	So sánh có dấu (A = 0, B = -1)	0x00000000, 0xffffffff	LESS = 1, EQUAL = 0	LESS = 1, EQUAL = 0	PASS

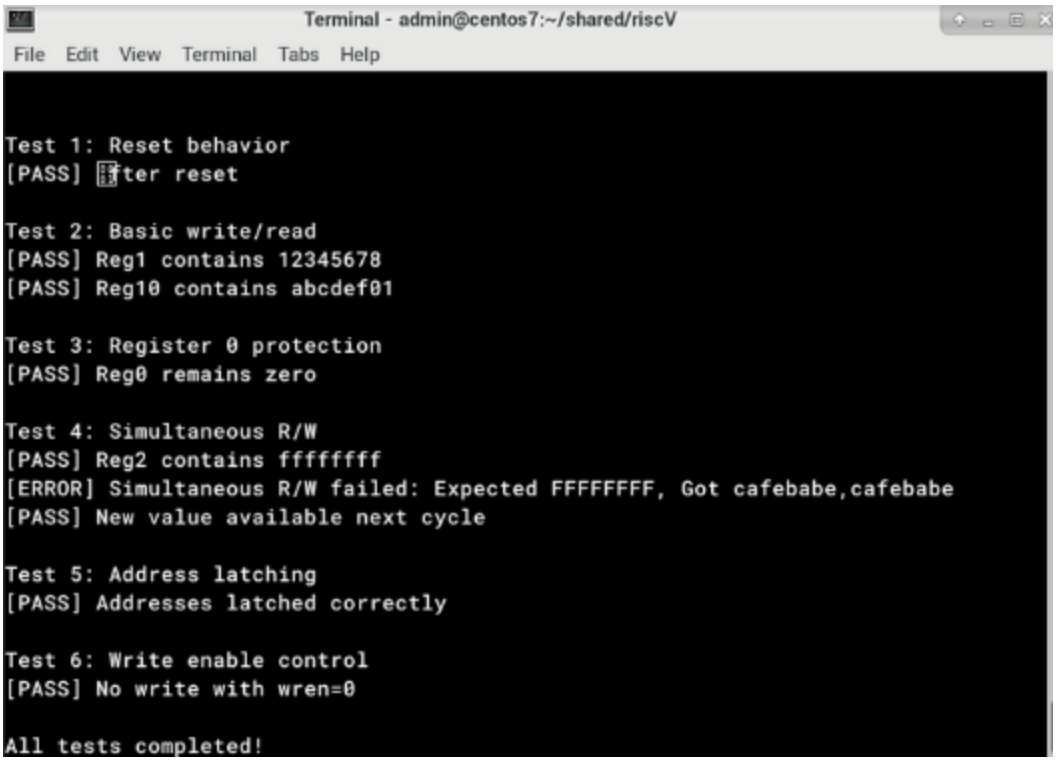
SIGNED	So sánh có dấu (A = -1, B = 0)	0xffffffff, 0x00000000	LESS = 0, EQUAL = 0	LESS = 0, EQUAL = 0	PASS
UNSIGNED	So sánh không dấu (A = 1, B = 4294967295)	0x00000001, 0xffffffff	LESS = 1, EQUAL = 0	LESS = 1, EQUAL = 0	PASS
UNSIGNED	So sánh không dấu (A = 0, B = 4294967295)	0x00000000, 0xffffffff	LESS = 0, EQUAL = 0	LESS = 0, EQUAL = 0	PASS
SIGNED	So sánh có dấu (A = 0, B = 0)	0x00000000, 0x00000000	LESS = 0, EQUAL = 1	LESS = 0, EQUAL = 1	PASS
UNSIGNED	So sánh không dấu (A = 0, B = 4294967295)	0x00000000, 0xffffffff	LESS = 1, EQUAL = 0	LESS = 1, EQUAL = 0	PASS
SIGNED	So sánh có dấu (A = -1, B = -1)	0xffffffff, 0xffffffff	LESS = 0, EQUAL = 1	LESS = 0, EQUAL = 1	PASS



3.3. Regfile

Test Case	**Mô tả**	**Kết quả Mong đợi**	**Kết quả Thực tế**	**Trạng thái**	**Ghi chú**
Test 1	Kiểm tra reset	Các thanh ghi reset về trạng thái mặc định	Reset thành công	**PASS**	-
Test 2	Ghi/đọc cơ bản	Dữ liệu ghi/đọc chính xác	Reg1=12345678, Reg10=abcdef01	**PASS**	Xác nhận chức năng cơ bản
Test 3	Bảo vệ Register 0	Register 0 luôn bằng 0	Giá trị không đổi	**PASS**	Chế độ bảo vệ hoạt động tốt

Test 4	Đọc/ghi đồng thời	Giá trị đồng bộ khi truy cập song song	Reg2=ffffffff (mong đợi) nhưng nhận **cafebabe**	**PARTIAL PASS**	Lỗi truy cập đồng thời (Mong đợi: FFFFFFFF, Thực tế: cafebabe)
		Giá trị mới xuất hiện ở chu kỳ tiếp theo	Hoạt động đúng	**PASS**	-
Test 5	Giữ địa chỉ	Địa chỉ được giữ chính xác	Địa chỉ đúng	**PASS**	-
Test 6	Điều khiển ghi (wren=0)	Không ghi khi wren=0	Không ghi sai	**PASS**	Kiểm soát ghi hoạt động tốt

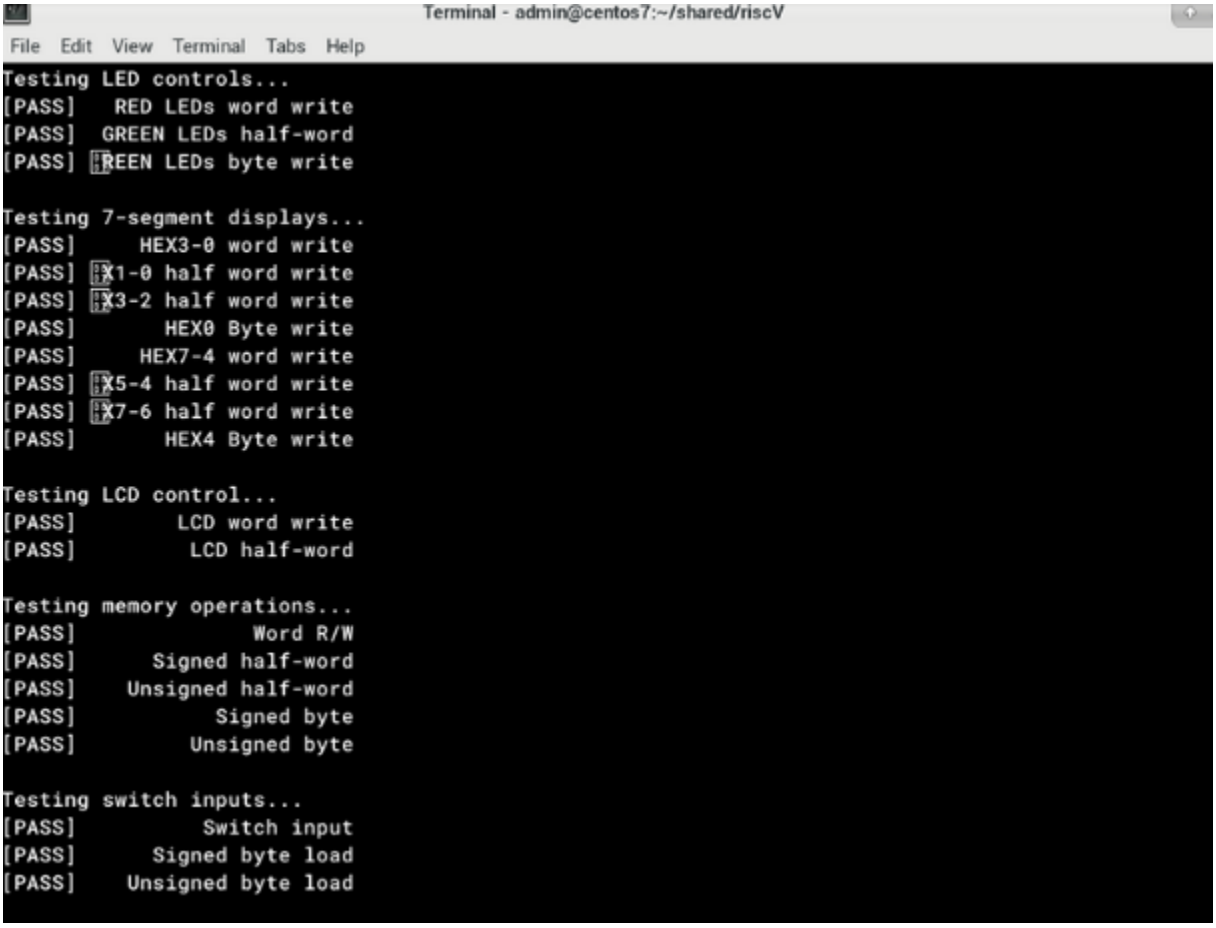


3.4. LSU

Nhóm Kiểm thử	**Test Case**	**Mô tả**	**Kết quả**	**Ghi chú**
LED Controls	RED LEDs word write	Ghi dữ liệu dạng word vào LED đỏ	PASS	
	GREEN LEDs half-word write	Ghi dữ liệu half-word vào LED xanh	PASS	
	GREEN LEDs byte write	Ghi dữ liệu byte vào LED xanh	PASS	
7-Segment Displays	HEX3-0 word write	Ghi word vào HEX3-0	PASS	

	HEX1-0 half-word write	Ghi half-word vào HEX1-0	PASS	
	HEX3-2 half-word write	Ghi half-word vào HEX3-2	PASS	
	HEX0 byte write	Ghi byte vào HEX0	PASS	
	HEX7-4 word write	Ghi word vào HEX7-4	PASS	
	HEX5-4 half-word write	Ghi half-word vào HEX5-4	PASS	
	HEX7-6 half-word write	Ghi half-word vào HEX7-6	PASS	
	HEX4 byte write	Ghi byte vào HEX4	PASS	
LCD Control	LCD word write	Ghi dữ liệu word vào LCD	PASS	
	LCD half-word write	Ghi half-word vào LCD	PASS	
Memory Operations	Word R/W	Đọc/ghi word	PASS	
	Signed half-word R/W	Đọc/ghi half-word có dấu	PASS	
	Unsigned half-word R/W	Đọc/ghi half-word không dấu	PASS	
	Signed byte R/W	Đọc/ghi byte có dấu	PASS	

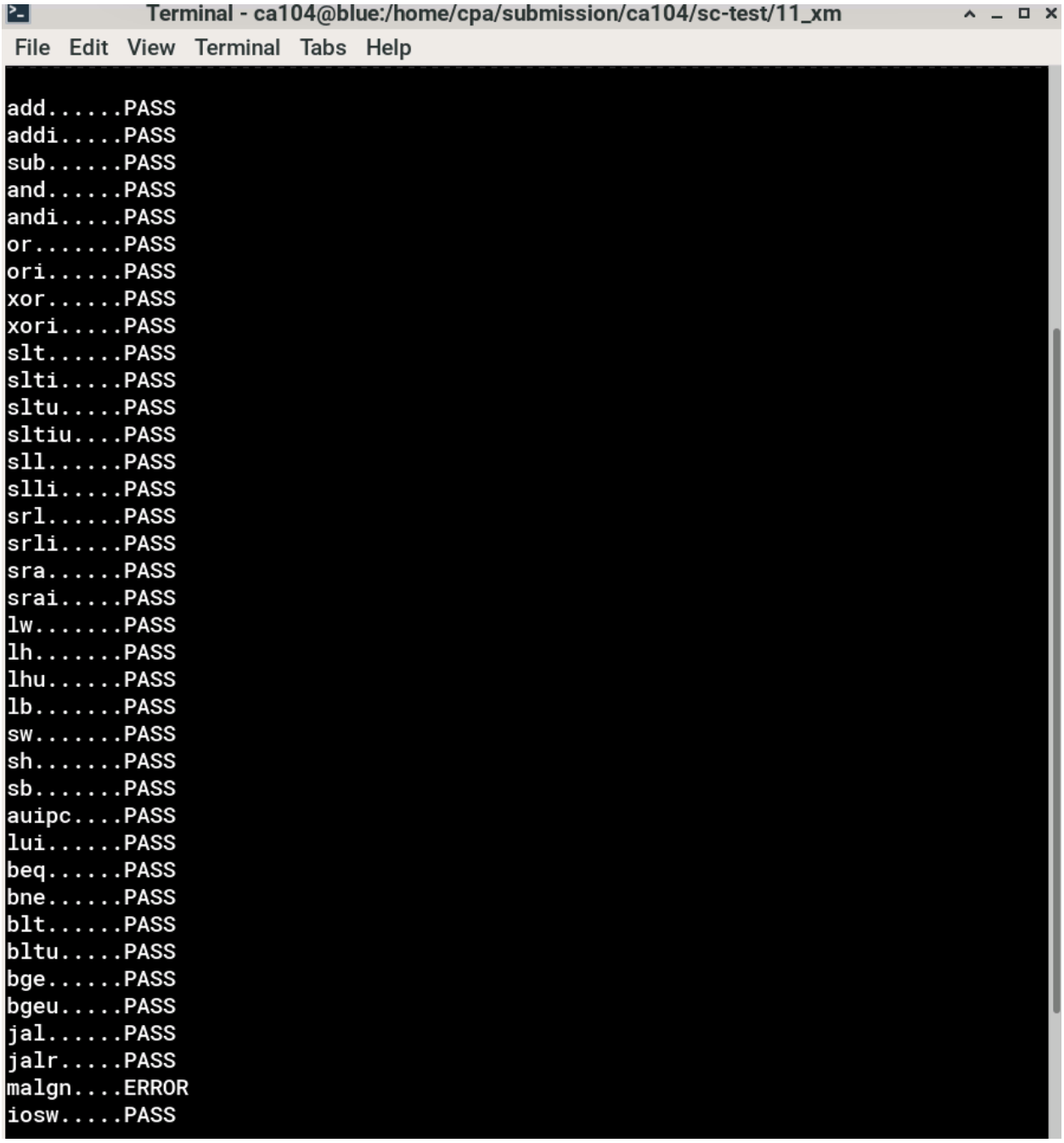
	Unsigned byte R/W	Đọc/ghi byte không dấu	PASS	
Switch Inputs	Switch input	Đọc trạng thái công tắc	PASS	
	Signed byte load	Đọc byte có dấu từ công tắc	PASS	
	Unsigned byte load	Đọc byte không dấu từ công tắc	PASS	



3.5. Singlecycle RiscV

Nhóm Lệnh	Lệnh	Kết Quả	Ghi chú

Số học	add	PASS	Kiểm tra cộng 2 số nguyên.
	addi	PASS	Kiểm tra cộng với hằng số.
	sub	PASS	Kiểm tra trừ 2 số nguyên.
	and/andi	PASS	Kiểm tra phép AND.
	or/ori	PASS	Kiểm tra phép OR.
	xor/xori	PASS	Kiểm tra phép XOR.
So sánh	slt/slti	PASS	Kiểm tra so sánh nhỏ hơn (có dấu).
	sltu/sltiu	PASS	Kiểm tra so sánh nhỏ hơn (không dấu).
Dịch bit	sll/slli	PASS	Kiểm tra dịch trái logic.
	srl/srli	PASS	Kiểm tra dịch phải logic.
	sra/srai	PASS	Kiểm tra dịch phải số học.
Truy cập bộ nhớ	lw	PASS	Kiểm tra đọc từ bộ nhớ (word).
	lh/lb	PASS	Kiểm tra đọc half-word/byte.
	sw	PASS	Kiểm tra ghi vào bộ nhớ.
Nhảy/Branch	beq/bne	PASS	Kiểm tra nhảy có điều kiện.
	blt/bltu	PASS	Kiểm tra nhảy nếu nhỏ hơn.
	bge/bgeu	PASS	Kiểm tra nhảy nếu lớn hơn hoặc bằng.
	jal/jalr	PASS	Kiểm tra nhảy không điều kiện và liên kết.
Khác	auipc	PASS	Kiểm tra tính địa chỉ tương đối PC.
	lui	PASS	Kiểm tra tải hằng số vào thanh ghi.
	malgn	**ERROR**	Lỗi căn chỉnh bộ nhớ (Optional)



4. Evaluation

Ưu điểm

- **Đơn giản và dễ hiểu**: Thiết kế single-cycle sử dụng một chu kỳ đồng hồ duy nhất để thực thi mọi lệnh, giúp logic điều khiển đơn giản, không cần xử lý

pipeline hay phát hiện hazard phức tạp. Điều này phù hợp cho mục đích giáo dục, giúp người học nắm bắt nguyên lý hoạt động cơ bản của bộ xử lý.

- **Tính nguyên bản của RISC-V**: Kiến trúc RISC-V với độ dài lệnh cố định (32-bit) và tập lệnh đơn giản (load-store architecture) tương thích tốt với mô hình single-cycle, giảm thiểu độ phức tạp khi triển khai.
- **Phù hợp cho ứng dụng tốc độ thấp**: Thiết kế này có thể ứng dụng trong các hệ thống yêu cầu tiết kiệm năng lượng hoặc xử lý tác vụ đơn giản, không đòi hỏi hiệu năng cao.

Nhược điểm

- **Hiệu năng thấp**: Chu kỳ đồng hồ bị giới hạn bởi lệnh có thời gian thực thi lâu nhất (ví dụ: lệnh load phải trải qua fetch, decode, execute, memory access, write-back). Điều này dẫn đến tốc độ xung nhịp thấp, không tận dụng được hiệu quả của các lệnh đơn giản (như add, sub).
- **Lãng phí tài nguyên**: Các thành phần như ALU, bộ nhớ chỉ được dùng một lần mỗi chu kỳ, trong khi thiết kế pipeline có thể tái sử dụng chúng qua các giai đoạn.
- **Khó mở rộng**: Việc thêm các lệnh phức tạp (nhân/chia, xử lý số thực) làm tăng độ trễ tổng, khiến thiết kế kém linh hoạt và không phù hợp với ứng dụng hiện đại.

Kết luận

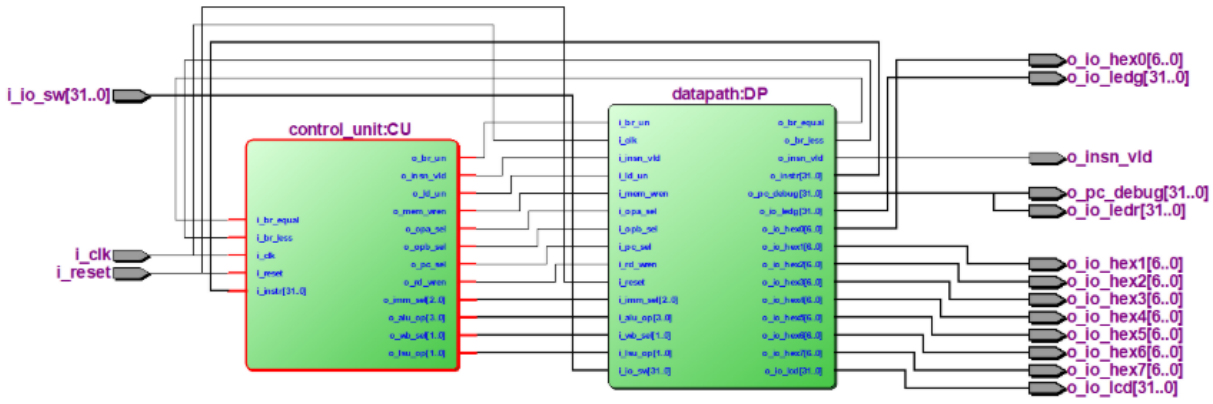
Thiết kế single-cycle RISC-V phù hợp làm công cụ giảng dạy hoặc triển khai trong hệ thống nhúng đơn giản nhờ ưu điểm về độ đơn giản. Tuy nhiên, hạn chế về hiệu năng

và khả năng mở rộng khiến nó không thực tế trong các bài toán yêu cầu xử lý tốc độ cao. Đây là nền tảng quan trọng để chuyển tiếp sang thiết kế pipeline hoặc kiến trúc đa chu kỳ tối ưu hơn.

5. Result

5.1. Quartus

Logic Utilization/Total logic elements	19073	xxx%
Total combinational functions	15490	
Total registers	9861	
Total pins	219	



References

Patterson, D. A., & Hennessy, J. L. (2017). *Computer Organization and Design: The Hardware/Software Interface, RISC-V Edition*. Morgan Kaufmann.

Waterman, A., & Asanović, K. (Eds.). (2019). *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon LLC.

Harris, S. L., & Harris, D. M. (2021). *Digital Design and Computer Architecture: RISC-V Edition*. Morgan Kaufmann.

RISC-V Foundation. (2023). *RISC-V Specifications: Volume I – User-Level ISA*. Truy cập từ https://riscv.org/technical/specifications/

Hennessy, J. L., & Patterson, D. A. (2019). *A New Golden Age for Computer Architecture: Domain-Specific Hardware/Software Co-Design*. Communications of the ACM, 62(2), 48–60.

MIT OpenCourseWare. (2016). *6.004: Computation Structures – Lecture Notes*. Truy cập từ https://ocw.mit.edu/courses/6-004-computation-structures-spring-2017/

Weste, N. H. E., & Harris, D. M. (2015). *CMOS VLSI Design: A Circuits and Systems Perspective*. Pearson.