ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC & KỸ THUẬT MÁY TÍNH

Báo cáo tổng kết
Thiết kế RISC CPU đơn giản

GV hướng dẫn: Huỳnh Phúc Nghị

Nhóm 3 - L06

Võ Hoàng Tân - 2213070
Trần Hoàng Phúc Đạt - 2210718
Ngô Trí Sơn - 2212941
Nguyễn Hoàng Sơn - 2212943
Lê Nhật Minh - 2212046

Tp. Hồ Chí Minh, Tháng 4/2025

Mssv	Họ và tên	Nhiệm vụ	Tỷ lệ tham gia
2210718	Trần Hoàng Phúc Đạt	Thiết kế các khối risc cpu, kiểm thử hệ thống, làm báo cáo phần 1,2	100%
2213070	Võ Hoàng Tân	Thiết kế khối alu, viết báo cáo phần 3,4	100%
2212941	Ngô Trí Sơn	Thiết kế khối register, viết báo cáo phần 5	100%
2212943	Nguyễn Hoàng Sơn	Thiết kế khối memory, viết báo cáo phần 6	100%
2212046	Lê Nhật Minh	Thiết kế khối address mux, viết báo cáo phần 1	100%

Mục lục

1 Giới thiệu

1.1 Giới thiệu đề tài

Trong lĩnh vực thiết kế hệ thống số, kiến trúc RISC (Reduced Instruction Set Computer) là một phương pháp thiết kế bộ xử lý hiệu quả với tập lệnh đơn giản, giúp tối ưu hóa hiệu suất và giảm độ phức tạp của phần cứng. Đề tài **Thiết kế RISC CPU đơn giản** tập trung vào việc xây dựng một CPU RISC với tập lệnh cơ bản, sử dụng ngôn ngữ mô tả phần cứng Verilog HDL. CPU được thiết kế có khả năng thực thi các lệnh đơn giản với 3-bit opcode (hỗ trợ 8 loại lệnh) và 5-bit toán hạng (hỗ trợ 32 không gian địa chỉ). Hệ thống hoạt động dựa trên tín hiệu xung nhịp (`clk`) và tín hiệu reset (`rst`), dừng hoạt động khi nhận tín hiệu `HALT`.

1.2 Công cụ sử dụng

Quá trình thiết kế và kiểm thử CPU RISC được thực hiện với các công cụ sau:

- **Cadence Design Systems**: Sử dụng các công cụ của Cadence như *Incisive Enterprise Simulator (IES)* để mô phỏng và kiểm tra mã Verilog, đảm bảo tính chính xác của thiết kế.
- **Verilog HDL**: Ngôn ngữ mô tả phần cứng được sử dụng để viết mã cho các khối chức năng của CPU và testbench.
- **MobaXterm**: Sử dụng MobaXterm để kết nối SSH tới máy chủ Linux có bộ công cụ phục vụ biên dịch và mô phỏng Verilog HDL.

1.3 Thiết bị sử dụng

Do đây là thiết kế dựa trên mô phỏng, không có thiết bị phần cứng vật lý cụ thể được sử dụng. Toàn bộ quá trình thiết kế, mô phỏng và kiểm thử được thực hiện trên máy tính cá nhân với cấu hình phù hợp, chạy các công cụ của Cadence. Các tệp chương trình (`program1.mem`, `program2.mem`, `program3.mem`) được sử dụng để nạp dữ liệu và lệnh vào bộ nhớ của CPU trong quá trình kiểm thử.

1.4 Chức năng của sản phẩm

CPU RISC đơn giản được thiết kế với các chức năng chính như sau:

- **Nạp lệnh**: Lấy lệnh từ bộ nhớ (`Memory`) dựa trên địa chỉ được cung cấp bởi `Program Counter`.
- **Giải mã lệnh**: Xử lý lệnh thông qua `Instruction Register` để trích xuất opcode (3-bit) và toán hạng (5-bit).
- **Lấy dữ liệu toán hạng**: Sử dụng `Address Mux` để chọn địa chỉ toán hạng từ bộ nhớ nếu cần.
- **Thực thi lệnh**: Thực hiện các phép toán (cộng, AND, XOR, tải, lưu, nhảy, v.v.) thông qua `ALU`, dựa trên opcode.
- **Lưu kết quả**: Ghi kết quả vào `Accumulator Register` hoặc bộ nhớ.
- **Quản lý luồng điều khiển**: `Controller` điều phối các trạng thái hoạt động (nạp lệnh, thực thi, lưu trữ) và xử lý tín hiệu `HALT` để dừng chương trình.

Hệ thống hỗ trợ 8 lệnh cơ bản, bao gồm dừng (`HLT`), nhảy có điều kiện (`SKZ`), cộng (`ADD`), AND (`AND`), XOR (`XOR`), tải (`LDA`), lưu (`STO`), và nhảy vô điều kiện (`JMP`).

2 Lý thuyết sơ lược

2.1 Kiến trúc RISC

RISC (Reduced Instruction Set Computer) là một kiến trúc bộ xử lý với tập lệnh đơn giản, được thiết kế để tối ưu hóa tốc độ thực thi và giảm độ phức tạp của phần cứng. So với kiến trúc CISC (Complex Instruction Set Computer), RISC sử dụng ít lệnh hơn, mỗi lệnh thực hiện một tác vụ đơn giản và hoàn thành trong một chu kỳ xung nhịp. Các đặc điểm chính của RISC bao gồm:

- **Tập lệnh đơn giản**: Mỗi lệnh có kích thước cố định và thực hiện một thao tác cơ bản.
- **Đăng ký đa năng**: Sử dụng nhiều thanh ghi để lưu trữ dữ liệu trung gian, giảm truy cập bộ nhớ.
- **Điều khiển cứng**: Logic điều khiển được hiện thực trực tiếp trong phần cứng, thay vì sử dụng vi mã.
- **Tối ưu hóa pipeline**: Các lệnh đơn giản cho phép dễ dàng triển khai pipeline, tăng hiệu suất.

Trong thiết kế này, CPU RISC sử dụng tập lệnh 8-bit (3-bit opcode, 5-bit toán hạng) và hoạt động theo mô hình tuần tự, với các trạng thái được điều khiển bởi `Controller`.

2.2 Các thành phần chính của CPU RISC

CPU RISC được thiết kế theo kiến trúc mô-đun, bao gồm các khối chức năng chính, mỗi khối đảm nhiệm một vai trò cụ thể trong chu trình xử lý lệnh (nạp, giải mã, thực thi, lưu trữ). Dưới đây là mô tả chi tiết về từng thành phần:

2.2.1 Program Counter (PC)

Chức năng: Program Counter (PC) là một thanh ghi lưu trữ địa chỉ của lệnh hiện tại trong bộ nhớ chương trình, đóng vai trò như một con trỏ để chỉ định vị trí của lệnh tiếp theo cần nạp.

Cách hoạt động:

- PC có độ rộng 5-bit, hỗ trợ không gian địa chỉ 32 vị trí (từ 0 đến 31).
- Hoạt động đồng bộ với xung nhịp `clk`. Tại mỗi xung nhịp, PC có thể:
  - **Tăng giá trị** (khi `inc_pc = 1`): PC tăng thêm 1 để trỏ đến lệnh tiếp theo.
  - **Nạp giá trị mới** (khi `ld_pc = 1`): PC nhận giá trị từ toán hạng 5-bit `load_val` trong trường hợp thực hiện lệnh nhảy `JMP`.
  - **Reset** (khi `rst = 1`): PC trở về giá trị 0.
- Trong mã Verilog `program_counter.v`, PC được hiện thực bằng một thanh ghi 5-bit, cập nhật giá trị dựa trên các tín hiệu điều khiển `inc_pc`, `ld_pc`, và `rst`.

Vai trò: PC cung cấp địa chỉ cho khối Address Mux trong giai đoạn nạp lệnh, đảm bảo CPU biết vị trí của lệnh cần thực thi tiếp theo. Trong kiến trúc RISC, PC hỗ trợ luồng điều khiển tuần tự và các lệnh nhảy với độ trễ thấp.

2.2.2 Address Mux

Chức năng: Khối Address Mux (bộ chọn địa chỉ) quyết định địa chỉ nào sẽ được gửi đến bộ nhớ: địa chỉ lệnh (từ PC) hay địa chỉ toán hạng (từ lệnh).

Cách hoạt động:

- Nhận hai đầu vào 5-bit:

  - `pc_addr`: Địa chỉ từ PC.
  - `op_addr`: Địa chỉ toán hạng từ Instruction Register.
- Tín hiệu điều khiển `sel` xác định đầu ra:
  - `sel = 1`: Chọn `pc_addr` (dùng trong giai đoạn nạp lệnh).
  - `sel = 0`: Chọn `op_addr` (dùng trong giai đoạn lấy toán hạng).
- Độ rộng địa chỉ là 5-bit, được định nghĩa bằng tham số `WIDTH = 5` trong mã Verilog `address_mux.v`, cho phép thay đổi nếu cần.
- Đầu ra `addr_out` được gửi đến khối Memory để truy cập dữ liệu hoặc lệnh.

Vai trò: Address Mux đảm bảo CPU chuyển đổi linh hoạt giữa việc nạp lệnh và lấy dữ liệu toán hạng, giảm xung đột trong truy cập bộ nhớ. Trong RISC, thành phần này hỗ trợ các lệnh truy cập bộ nhớ (`load/store`) với hiệu suất cao.

2.2.3 Memory

Chức năng: Bộ nhớ Memory lưu trữ cả lệnh và dữ liệu, cung cấp dữ liệu cho CPU trong quá trình nạp lệnh hoặc lấy toán hạng, và lưu trữ kết quả khi thực hiện lệnh ghi.

Cách hoạt động:

- Bộ nhớ có kích thước 32 vị trí (5-bit địa chỉ), mỗi vị trí lưu dữ liệu 8-bit.
- Sử dụng cổng dữ liệu hai chiều `data`:
  - **Đọc** (`rd = 1`, `wr = 0`): Dữ liệu tại địa chỉ `addr` được xuất ra `data`.
  - **Ghi** (`wr = 1`, `rd = 0`): Dữ liệu từ `data` được ghi vào địa chỉ `addr`.
- Hoạt động đồng bộ với `clk` khi ghi, nhưng đọc là bất đồng bộ để tăng tốc độ.
- Trong mã Verilog `memory.v`, bộ nhớ được khởi tạo bằng các tệp `program1.mem`, `program2.mem`, hoặc `program3.mem` thông qua tín hiệu `prog_sel`.

Vai trò: Bộ nhớ là trung tâm lưu trữ, cung cấp lệnh cho Instruction Register và dữ liệu cho ALU, đồng thời lưu trữ kết quả từ Accumulator. Trong RISC, bộ nhớ chỉ được truy cập qua các lệnh `load/store`, giúp đơn giản hóa điều khiển.

2.2.4 Instruction Register (IR)

Chức năng: Thanh ghi lệnh Instruction Register lưu trữ lệnh được nạp từ bộ nhớ, trích xuất `opcode` (3-bit) và toán hạng (5-bit) để điều khiển CPU.

Cách hoạt động:

- Nhận dữ liệu 8-bit từ bộ nhớ qua cổng `data` khi tín hiệu `ld_ir = 1`.
- Đầu ra `ir_out` cung cấp:
  - Bit [7:5]: `opcode` (gửi đến Controller và ALU).
  - Bit [4:0]: Toán hạng (gửi đến Address Mux hoặc Program Counter).
- Hoạt động đồng bộ với `clk`, reset về 0 khi `rst = 1`.
- Trong mã Verilog `register.v`, IR được hiện thực như một thanh ghi 8-bit với tín hiệu `load`.

Vai trò: IR là cầu nối giữa bộ nhớ và các khối điều khiển, cung cấp thông tin lệnh để giải mã và thực thi. Trong RISC, IR hỗ trợ tập lệnh cố định (8-bit), đơn giản hóa quá trình giải mã.

2.2.5 Accumulator Register (AC)

Chức năng: Thanh ghi tích lũy Accumulator Register lưu trữ toán hạng hoặc kết quả trung gian từ ALU, đóng vai trò như một bộ đệm dữ liệu.

Cách hoạt động:

- Nhận đầu vào 8-bit từ ALU khi `ld_ac = 1`.
- Đầu ra `ac_out` được gửi đến ALU (toán hạng `inA`) hoặc bộ nhớ (khi ghi dữ liệu).
- Hoạt động đồng bộ với `clk`, reset về 0 khi `rst = 1`.
- Trong mã Verilog `register.v`, AC được hiện thực tương tự IR, với tín hiệu `load` để cập nhật giá trị.

Vai trò: AC cung cấp dữ liệu cho các phép toán trong ALU và lưu trữ kết quả tạm thời, giảm số lần truy cập bộ nhớ. Trong RISC, AC là một thanh ghi đa năng, hỗ trợ tăng tốc độ xử lý.

2.2.6 ALU (Arithmetic Logic Unit)

Chức năng: Đơn vị số học và logic ALU thực hiện các phép toán số học (cộng) và logic (AND, XOR) trên hai toán hạng 8-bit, dựa trên `opcode` 3-bit.

Cách hoạt động:

- Nhận hai đầu vào:
  - `inA`: Từ Accumulator.
  - `inB`: Từ bộ nhớ (thông qua thanh ghi tạm `mem_data_reg`).
- Đầu ra:
  - `out` (8-bit): Kết quả phép toán.
  - `is_zero` (1-bit): Bằng 1 nếu kết quả bằng 0, dùng cho lệnh nhảy có điều kiện `SKZ`.
- Các phép toán theo `opcode` (xem `alu.v`):
  - `000`: `HLT` (kết quả = 0).
  - `001`: `SKZ` (kết quả = `inA`).
  - `010`: `ADD` (kết quả = `inA` + `inB`).
  - `011`: `AND` (kết quả = `inA` $\wedge$ `inB`).
  - `100`: `XOR` (kết quả = `inA` $\oplus$ `inB`).
  - `101`: `LDA` (kết quả = `inB`).
  - `110`: `STO` (kết quả = `inA`).
  - `111`: `JMP` (kết quả = 0).
- Hoạt động bất đồng bộ, đầu ra cập nhật ngay khi đầu vào thay đổi.

Vai trò: ALU là trung tâm xử lý, thực hiện các phép toán cần thiết và cung cấp kết quả cho Accumulator hoặc bộ nhớ. Trong RISC, ALU được thiết kế đơn giản, chỉ thực hiện các phép toán cơ bản để đảm bảo chu kỳ xử lý nhanh.

2.2.7 Controller

Chức năng: Bộ điều khiển Controller quản lý luồng hoạt động của CPU, điều phối các tín hiệu điều khiển dựa trên trạng thái hiện tại và `opcode`.

Cách hoạt động:

- Hoạt động như một máy trạng thái hữu hạn (FSM) với 8 trạng thái (xem `controller.v`):
  1. `INST_ADDR`: Chuẩn bị địa chỉ lệnh.
  2. `INST_FETCH`: Nạp lệnh từ bộ nhớ.
  3. `INST_LOAD`: Lưu lệnh vào Instruction Register.
  4. `IDLE`: Trạng thái chờ.
  5. `OP_ADDR`: Chuẩn bị địa chỉ toán hạng.
  6. `OP_FETCH`: Lấy toán hạng từ bộ nhớ.
  7. `ALU_OP`: Thực hiện phép toán trong ALU.
  8. `STORE`: Lưu kết quả hoặc cập nhật PC.
- Nhận đầu vào:
  - `opcode` (3-bit, từ Instruction Register).
  - `zero` (từ ALU, cho lệnh `SKZ`).
- Tạo các tín hiệu điều khiển: `sel`, `rd`, `ld_ir`, `halt`, `inc_pc`, `ld_ac`, `ld_pc`, `wr`, `data_e`.
- Hoạt động đồng bộ với `clk`, reset về trạng thái `INST_ADDR` khi `rst = 1`.
- Hỗ trợ xử lý lệnh nhảy có điều kiện `SKZ` bằng biến `skip_next`.

Vai trò: Controller là "bộ não" của CPU, điều phối tất cả các khối chức năng để thực hiện chu trình lệnh chính xác. Trong RISC, Controller sử dụng logic điều khiển cứng, giúp giảm độ trễ và tăng hiệu suất.

2.3 Tính năng và ứng dụng thực tế

CPU RISC đơn giản được thiết kế có các tính năng nổi bật:

- **Đơn giản và dễ mở rộng**: Kiến trúc mô-đun cho phép dễ dàng thêm các lệnh hoặc khối chức năng mới.
- **Hiệu quả tài nguyên**: Sử dụng ít tài nguyên phần cứng, phù hợp với các hệ thống nhúng.
- **Hỗ trợ kiểm thử**: Mỗi khối chức năng được thiết kế với testbench riêng, đảm bảo tính chính xác.

Trong thực tế, các CPU RISC được ứng dụng rộng rãi trong:

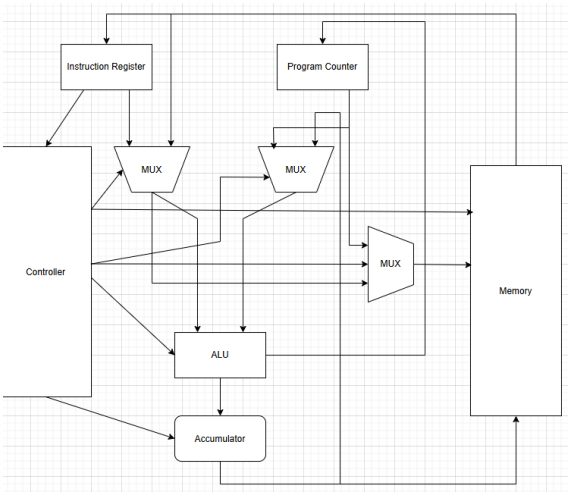
- **Hệ thống nhúng**: Ví dụ, vi điều khiển trong thiết bị IoT, ô tô, và thiết bị y tế.
- **Bộ xử lý tín hiệu số**: Xử lý âm thanh, hình ảnh với hiệu suất cao.
- **Máy tính hiệu năng cao**: Các kiến trúc như ARM, RISC-V sử dụng nguyên lý RISC để tối ưu hóa tốc độ và năng lượng.

Thiết kế này, mặc dù đơn giản, cung cấp nền tảng để hiểu và phát triển các CPU RISC phức tạp hơn trong các ứng dụng thực tế.

3 Thiết kế

3.1 Sơ đồ khối

Sơ đồ khối đơn giản của hệ thống RISC CPU:



Chức năng của các khối:

- Program Counter: chức năng lưu trữ thanh ghi địa chỉ của chương trình (program address)
- AddressMux: nhằm lựa chọn giữa địa chỉ chương trình hoặc địa chỉ của câu lệnh (instruction)
- Memory: lưu trữ và cung cấp dữ liệu cho chương trình
- Instruction Register: xử lý dữ liệu instruction
- Accumulator Register: xử lý dữ liệu từ ALU
- ALU: xử lý dữ liệu từ Memory, Accumulator và opcode của Instruction

4 Hiện thực hệ thống

4.1 Luồng hoạt động:

Luồng hoạt động có thể được mô tả như sau:

1. Xác định lệnh tiếp theo
   CPU bắt đầu bằng việc kiểm tra Program Counter (PC) để xác định địa chỉ của lệnh tiếp theo cần thực thi. PC giữ vai trò chỉ định vị trí trong bộ nhớ nơi lệnh được lưu.
2. Truy xuất lệnh từ bộ nhớ
   CPU gửi địa chỉ từ PC đến Memory để lấy lệnh. Bộ nhớ sẽ trả về dữ liệu tại địa chỉ đó, và lệnh được lưu vào Instruction Register (IR).
3. Giải mã lệnh
   Controller nhận lệnh từ IR và tiến hành giải mã. Controller hiểu lệnh yêu cầu gì, ví dụ: đọc dữ liệu, thực hiện phép toán, hoặc ghi dữ liệu.

4. Truy xuất dữ liệu đầu vào
   Nếu lệnh yêu cầu xử lý dữ liệu, CPU sẽ sử dụng MUX (Multiplexer) để chọn nguồn dữ liệu phù hợp. Dữ liệu có thể được lấy từ bộ nhớ hoặc các thanh ghi.
5. Xử lý dữ liệu
   Dữ liệu được gửi đến ALU (Arithmetic Logic Unit), nơi thực hiện các phép toán số học (như cộng, trừ) hoặc logic (như AND, OR). Kết quả tính toán được lưu vào Accumulator (ACC).
6. Cập nhật kết quả
   Kết quả từ ACC có thể được gửi trở lại bộ nhớ hoặc tiếp tục được xử lý trong các lệnh kế tiếp.
7. Cập nhật Program Counter
   Controller sẽ cập nhật PC để trỏ đến địa chỉ của lệnh tiếp theo trong bộ nhớ, chuẩn bị cho chu kỳ mới.
8. Lặp lại chu trình
   Chu trình tiếp tục cho đến khi toàn bộ chương trình được thực thi hoặc CPU nhận lệnh dừng.

4.2 Code hiện thực các module:

Address Mux:

```
`timescale 1ns/1ps
module address_mux #(
    parameter WIDTH = 5
) (
    input sel,
    input [WIDTH-1:0] pc_addr, op_addr,
    output [WIDTH-1:0] addr_out
);
    assign addr_out = sel ? pc_addr : op_addr;
endmodule
```

ALU:

```
`timescale 1ns/1ps
module alu (
    input [2:0] opcode,
    input [7:0] inA, inB,
    output reg [7:0] out,
    output reg is_zero
);
    always @(*) begin
        case (opcode)
            3'b000: out = 8'b0;          // HLT
            3'b001: out = inA;           // SKZ
            3'b010: out = inA + inB;     // ADD
            3'b011: out = inA & inB;     // AND
            3'b100: out = inA ^ inB;     // XOR
            3'b101: out = inB;           // LDA
            3'b110: out = inA;           // STO
            3'b111: out = 8'b0;          // JMP
            default: out = 8'b0;
        endcase
        is_zero = (out == 8'b0) ? 1'b1 : 1'b0;
    end

    // Debug
    always @(*) begin
        $display("ALU: opcode = %b, inA = %h, inB = %h, out = %h, is_zero = %b", opcode, inA, inB, out, is_zero);
    end
endmodule
```

Controller:

```
`timescale 1ns/1ps
module controller (
    input clk, rst,
    input [2:0] opcode,
    input zero,
    output reg sel, rd, ld_ir, halt, inc_pc, ld_ac, ld_pc, wr, data_e
);
    parameter INST_ADDR  = 3'd0,
              INST_FETCH = 3'd1,
              INST_LOAD  = 3'd2,
              IDLE       = 3'd3,
              OP_ADDR    = 3'd4,
              OP_FETCH   = 3'd5,
              ALU_OP     = 3'd6,
              STORE      = 3'd7;

    reg [2:0] state, next_state;
    reg skip_next;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            state <= INST_ADDR;
            skip_next <= 1'b0;
        end
        else if (!halt) begin
            state <= next_state;
            if (state == ALU_OP && opcode == 3'b001 && zero)
                skip_next <= 1'b1;
            else if (state == STORE)
                skip_next <= 1'b0;
        end
    end

    always @(*) begin
        case (state)
            INST_ADDR:  next_state = INST_FETCH;
            INST_FETCH: next_state = INST_LOAD;
            INST_LOAD:  next_state = IDLE;
            IDLE:       next_state = OP_ADDR;
            OP_ADDR:    next_state = OP_FETCH;
            OP_FETCH:   next_state = ALU_OP;
            ALU_OP:     next_state = STORE;
            STORE:      next_state = INST_ADDR;
            default:    next_state = INST_ADDR;
        endcase
    end

    always @(*) begin
        sel = 1'b0;
        rd = 1'b0;
        ld_ir = 1'b0;
        halt = 1'b0;
        inc_pc = 1'b0;
        ld_ac = 1'b0;
        ld_pc = 1'b0;
        wr = 1'b0;
        data_e = 1'b0;
        case (state)
            INST_ADDR: sel = 1'b1;
            INST_FETCH: begin
                sel = 1'b1;
                rd = 1'b1;
            end
            INST_LOAD: begin
                sel = 1'b1;
                rd = 1'b1;
                ld_ir = 1'b1;
            end
            IDLE: sel = 1'b1;
            OP_ADDR: begin
                if (opcode == 3'b010 || opcode == 3'b011 || opcode == 3'b100 || opcode == 3'b101 || opcode == 3'b110)
                    sel = 1'b0;
            end
            OP_FETCH: begin
                if (opcode == 3'b010 || opcode == 3'b011 || opcode == 3'b100 || opcode == 3'b101 || opcode == 3'b110)
                    rd = 1'b1;
            end
            ALU_OP: begin
                if (opcode == 3'b010 || opcode == 3'b011 || opcode == 3'b100 || opcode == 3'b101)
                    ld_ac = 1'b1;
                if (opcode == 3'b001 && zero)
                    inc_pc = 1'b1;
            end
            STORE: begin
                case (opcode)
                    3'b110: begin
                        wr = 1'b1;
                        data_e = 1'b1;
                        inc_pc = 1'b1;
                    end
                    3'b000: halt = 1'b1;
                    3'b111: ld_pc = 1'b1;
                    default: begin
                        inc_pc = 1'b1;
                        if (skip_next && opcode == 3'b001)
                            inc_pc = 1'b1;
                    end
                endcase
            end
        endcase
    end

    always @(posedge clk) begin
        $display("Controller: state = %d, opcode = %b, zero = %b, sel = %b, rd = %b, ld_ir = %b, halt = %b, inc_pc = %b, ld_ac = %b, ld_pc = %b, wr = %b, data_e = %b, skip_next = %b",
                 state, opcode, zero, sel, rd, ld_ir, halt, inc_pc, ld_ac, ld_pc, wr, data_e, skip_next);
        if (state == ALU_OP && opcode == 3'b001)
            $display("SKZ: zero = %b, skip_next = %b, inc_pc = %b", zero, skip_next, inc_pc);
        if (state == STORE && skip_next)
            $display("SKZ STORE: skip_next = %b, inc_pc = %b", skip_next, inc_pc);
    end
endmodule
```

Memory:

```
`timescale 1ns/1ps
module memory (
    input clk, rd, wr,
    input [4:0] addr,
    inout [7:0] data
);
    reg [7:0] mem[31:0];
    reg [7:0] data_out;

    initial $readmemb("../test/program2.mem", mem); // Tên tệp tạm thời, sẽ được ghi đè

    always @(posedge clk) begin
        if (wr && !rd) mem[addr] <= data;
    end

    always @(*) begin
        if (rd && !wr) data_out = mem[addr];
        else data_out = 8'bz;
    end

    assign data = (rd && !wr) ? data_out : 8'bz;

    // Debug
    always @(posedge clk) begin
        if (rd && !wr)
            $display("Memory read: addr = %h, data = %h", addr, data_out);
        if (wr && !rd)
            $display("Memory write: addr = %h, data = %h", addr, data);
    end
endmodule
```

Program Counter:

```
`timescale 1ns/1ps
module program_counter (
    input clk, rst, ld_pc, inc_pc,
    input [4:0] load_val,
    output reg [4:0] pc
);
    always @(posedge clk or posedge rst) begin
        if (rst) pc <= 5'b0;
        else if (ld_pc) pc <= load_val;
        else if (inc_pc) pc <= pc + 1;
    end
endmoduleS
```

Register:

```
`timescale 1ns/1ps
module register (
    input clk, rst, load,
    input [7:0] in,
    output reg [7:0] out
);
    always @(posedge clk or posedge rst) begin
        if (rst) out <= 8'b0;
        else if (load) out <= in;
    end
endmodule
```

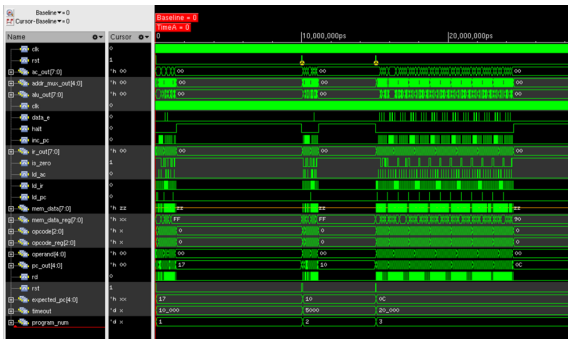
Risc CPU:

```
`timescale 1ns/1ps
module risc_cpu (
    input clk, rst
);
    wire [4:0] pc_out;
    wire [4:0] addr_mux_out;
    wire [7:0] mem_data;
    wire [7:0] ir_out;
    wire [7:0] ac_out;
    wire [7:0] alu_out;
    wire is_zero;
    wire [2:0] opcode;
    wire [4:0] operand;
    wire sel, rd, ld_ir, halt, inc_pc, ld_ac, ld_pc, wr, data_e;

    reg [7:0] mem_data_reg;
    always @(posedge clk) begin
        if (rst) mem_data_reg <= 8'h0;
        else if (ctrl.state == 5 && !ctrl.rd)
            mem_data_reg <= mem_data;
    end

    reg [2:0] opcode_reg;
    always @(posedge clk) begin
        if (rst) opcode_reg <= 3'b0;
        else opcode_reg <= ir_out[7:5];
    end

    // Kết nối các mô-đun
    program_counter pc (.clk(clk), .rst(rst), .ld_pc(ld_pc), .inc_pc(inc_pc), .load_val(operand), .pc(pc_out));
    address_mux addr_mux (.sel(sel), .pc_addr(pc_out), .op_addr(operand), .addr_out(addr_mux_out));
    memory mem (.clk(clk), .rd(rd), .wr(wr), .addr(addr_mux_out), .data(mem_data));
    register ir (.clk(clk), .rst(rst), .load(ld_ir), .in(mem_data), .out(ir_out));

    alu alu_inst (.opcode(opcode), .inA(ac_out), .inB(mem_data_reg), .out(alu_out), .is_zero(is_zero));
    register ac (.clk(clk), .rst(rst), .load(ld_ac), .in(alu_out), .out(ac_out));

    controller ctrl (.clk(clk), .rst(rst), .opcode(opcode), .zero(is_zero),
                     .sel(sel), .rd(rd), .ld_ir(ld_ir), .halt(halt), .inc_pc(inc_pc),
                     .ld_ac(ld_ac), .ld_pc(ld_pc), .wr(wr), .data_e(data_e));

    assign opcode = opcode_reg;
    assign operand = ir_out[4:0];

    assign mem_data = (data_e && wr && !rd) ? ac_out : 8'bz;

    always @(posedge clk) begin
        $display("CPU: PC = %h, AC = %h, IR = %h, opcode = %b, alu_out = %h, ld_ac = %b, is_zero = %b, mem_data = %h, mem_data_reg = %h",
                 pc_out, ac_out, ir_out, opcode, alu_out, ld_ac, is_zero, mem_data, mem_data_reg);
    end
endmodule
```

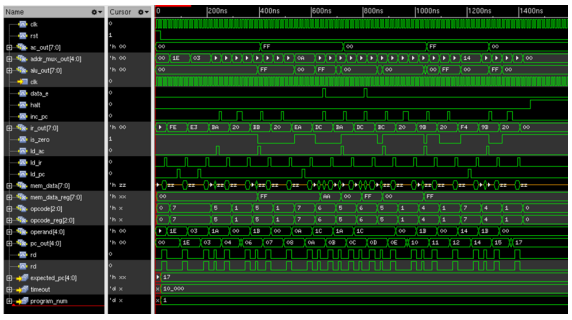
5 Kiểm thử và đánh giá

5.1 Kết quả mô phỏng các trường hợp kiểm thử

Tổng quan chương trình:



Program 1:



Mục đích: Chương trình `program1.mem` được thiết kế nhằm kiểm tra khả năng hoạt động của một bộ xử lý đơn giản theo kiến trúc RISC. Mục tiêu của chương trình là thực hiện các thao tác tải dữ liệu vào thanh ghi tích lũy (Accumulator), thực hiện một số phép toán logic (XOR), điều kiện rẽ nhánh (SKZ), nhảy không điều kiện (JMP), và cuối cùng là dừng hệ thống (HLT) tại địa chỉ dự kiến. Mục tiêu của simulation là xác định xem CPU có dừng đúng tại địa chỉ lệnh `0x17` với trạng thái `halt = 1` hay không. **Giải thích chương trình:**
Bộ nhớ chương trình bao gồm các opcode theo định dạng 8-bit. Trong đó, ba bit cao biểu diễn opcode, năm bit thấp biểu diễn địa chỉ hoặc dữ liệu. Các lệnh bao gồm:

- JMP: nhảy đến địa chỉ chỉ định.
- LDA: tải dữ liệu từ bộ nhớ vào thanh ghi tích lũy.
- SKZ: kiểm tra xem tích lũy có bằng 0 hay không, nếu có thì bỏ qua lệnh tiếp theo.
- XOR: thực hiện phép XOR giữa dữ liệu bộ nhớ và tích lũy.
- STO: ghi dữ liệu từ tích lũy ra bộ nhớ.
- HLT: dừng CPU.

Chương trình bắt đầu bằng hai lệnh `JMP`, lần lượt nhảy đến địa chỉ `0x1E` và tiếp theo là `0x03`. Mục đích là bỏ qua phần đầu bộ nhớ, chuyển đến đoạn chương trình chính bắt đầu từ địa chỉ `0x04`.

Tại địa chỉ `0x04`, chương trình thực hiện tải dữ liệu từ ô nhớ `0x1A` vào Accumulator. Do ô này chứa giá trị `0x00`, kết quả trong Accumulator là `0x00`. Lệnh tiếp theo là `SKZ`, kiểm tra Accumulator. Vì giá trị đang là 0, CPU bỏ qua lệnh kế tiếp (`HLT`) và tiếp tục thực hiện lệnh tại địa chỉ `0x07`.

Tại địa chỉ `0x07`, CPU tiếp tục tải giá trị từ ô nhớ `0x1B`, là `0xFF`, vào Accumulator. Do Accumulator khác 0, lệnh `SKZ` tại địa chỉ `0x08` không bỏ qua dòng tiếp theo. Lệnh `JMP` tại `0x09` thực hiện nhảy đến `0x0A`, nơi có một lệnh `HLT`. Tuy nhiên, lệnh này có thể là một điểm dừng tạm hoặc không được thực thi tùy thời điểm. CPU sau đó tiếp tục thực hiện các lệnh ghi và tải, như `STO` để ghi Accumulator ra địa chỉ `0x1C`, `LDA` để tải lại dữ liệu từ các địa chỉ `0x1A` và `0x1C`, tiếp tục thực hiện phép `XOR` với dữ liệu từ `0x1B`, và kiểm tra điều kiện `SKZ`.

Sau khi thực hiện phép `XOR 0x00` $\wedge$ `0xFF = 0xFF`, Accumulator mang giá trị `0xFF`. CPU tiếp tục thực hiện lệnh nhảy và cuối cùng thực hiện lại phép `XOR 0xFF` $\wedge$ `0xFF = 0x00`. Lúc này, điều kiện `SKZ` là đúng và CPU bỏ qua lệnh `HLT` kế tiếp. Tuy nhiên, địa chỉ kế tiếp sau lệnh bị bỏ qua chính là `0x17`, và CPU dừng tại đây như mong muốn.

Phân tích waveform:

Trong quá trình mô phỏng, ta sử dụng waveform để quan sát hoạt động của CPU. Các tín hiệu quan trọng được theo dõi gồm:

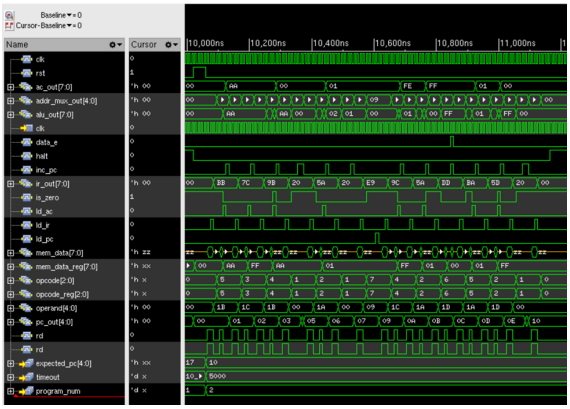
- clk: xung clock điều khiển nhịp hoạt động.
- rst: tín hiệu reset ban đầu.
- pc out: hiển thị địa chỉ lệnh hiện tại mà CPU đang thực thi.
- ac out: giá trị hiện tại của thanh ghi tích lũy.
- ir out: nội dung Instruction Register – lệnh đang xử lý.
- mem data: dữ liệu từ bộ nhớ chương trình.
- ld ir, ld pc: các tín hiệu điều khiển việc nạp lệnh và cập nhật PC.
- is zero: báo hiệu Accumulator có bằng 0 hay không.
- halt: tín hiệu báo dừng CPU.
- expected pc: giá trị PC kỳ vọng để test.

Waveform thể hiện rõ chuỗi hành vi tương ứng với từng lệnh. Tại mỗi cạnh lên của clock, CPU thực hiện một phần của chu kỳ lệnh. Từ khoảng 200ns đến hơn 1000ns, có thể quan sát được:

- PC thay đổi đúng theo logic của chương trình.
- Các lệnh như LDA, SKZ, XOR, JMP được thực thi chính xác.

- o Dữ liệu được nạp vào và ghi ra đúng địa chỉ nhớ, đặc biệt là ở các địa chỉ 0x1A, 0x1B, và 0x1C.
- ac out thể hiện rõ kết quả của các phép toán logic, đặc biệt là kết quả XOR chuyển từ 0xFF thành 0x00.
- Tín hiệu is zero phản ánh trạng thái của Accumulator và được sử dụng bởi lệnh SKZ.
- Đến khoảng thời điểm 520ns trở đi, halt = 1 và pc out = 0x17, xác nhận rằng chương trình đã thực thi thành công và dừng đúng vị trí yêu cầu.

Program 2:



Mục đích: Chương trình `program2.mem` được thiết kế nhằm kiểm tra sâu hơn các chức năng của CPU đơn giản kiến trúc RISC như: tải dữ liệu, các phép toán logic (`AND`, `XOR`, `ADD`), kiểm tra điều kiện (`SKZ`), ghi vào bộ nhớ (`STO`), và rẽ nhánh không điều kiện (`JMP`). Chương trình mô phỏng khả năng xử lý dữ liệu và thực hiện thao tác logic phức tạp hơn so với `program1.mem`.

Giải thích chương trình: Chương trình bắt đầu từ địa chỉ 0x00 với chuỗi các lệnh sau:

1. LDA DATA 2 (0x1B): Nạp giá trị 0xAA vào Accumulator.
2. AND DATA 3 (0x1C): Thực hiện phép 0xAA 0xFF = 0xAA. Accumulator vẫn giữ giá trị 0xAA.
3. XOR DATA 2 (0x1B): $0xAA \wedge 0xAA = 0x00$. Accumulator trở về 0.
4. SKZ: Vì Accumulator bằng 0, lệnh tiếp theo (HLT) bị bỏ qua.
5. ADD DATA 1 (0x1A): 0x00 + 0x01 = 0x01. Accumulator cập nhật thành 0x01.
6. SKZ: Accumulator khác 0 → thực hiện lệnh tiếp theo.
7. JMP 0x09: Nhảy đến địa chỉ 0x09.
8. XOR DATA 3 (0x1C): $0x01 \wedge 0xFF = 0xFE$. Accumulator = 0xFE.
9. ADD DATA 1 (0x1A): 0xFE + 0x01 = 0xFF.
10. STO TEMP (0x1D): Ghi 0xFF vào địa chỉ 0x1D.
11. LDA DATA 1 (0x1A): Nạp 0x01 vào Accumulator.
12. ADD TEMP (0x1D): 0x01 + 0xFF = 0x00 (do tràn bit, giả sử 8-bit).
13. SKZ: Accumulator bằng 0 → bỏ qua HLT tại 0x0F.
14. HLT tại 0x10: CPU dừng lại ở đây.

Giá trị của các biến:

- DATA 1 (0x1A): 0x01
- DATA 2 (0x1B): 0xAA
- DATA 3 (0x1C): 0xFF
- TEMP (0x1D): Kết thúc bằng 0xFF

Phân tích waveform:

Hình ảnh waveform cho thấy các tín hiệu chính vận hành theo đúng trình tự chương trình:

- `pc_out` tăng theo lệnh, ngoại trừ khi có `JMP`, thì nhảy về địa chỉ tương ứng.
- `ac_out` hiển thị kết quả chính xác sau mỗi phép toán (`AND`, `XOR`, `ADD`): `0xAA` → `0x00` → `0x01` → `0xFE` → `0xFF` → `0x01` → `0x00`.
- `is_zero` kích hoạt tại thời điểm `ac_out = 0x00`, điều khiển logic của lệnh `SKZ`.
- `halt` được kích hoạt khi CPU dừng tại địa chỉ `0x10`, trùng với `expected_pc = 0x10`, xác nhận chương trình kết thúc đúng vị trí.
- `mem_data` và `mem_data_reg` thể hiện dữ liệu được truy xuất chính xác từ các địa chỉ `0x1A`, `0x1B`, `0x1C`, `0x1D`.

Program 3:

Mục đích: Chương trình 3 là một chương trình mô phỏng thuật toán tính dãy Fibonacci, được triển khai trên vi xử lý đơn giản sử dụng tập lệnh cơ bản như `LDA`, `ADD`, `STO`, `JMP`, `SKZ`,... Dãy Fibonacci được lưu qua hai biến `FN1` và `FN2`, với kết quả mới được lưu tạm vào `TEMP`, và chương trình sẽ tiếp tục cộng dồn các số Fibonacci cho đến khi đạt một giới hạn xác định bởi biến `LIMIT`.

Giải thích chương trình: *Các dòng lệnh trong* ***`program3.mem`*** *hoạt động theo chu trình như sau:*

1. Khởi tạo (Boot):
   0x00: JMP 0x03; Nhảy tới LOOP chính (bỏ qua phần HLT rỗng)
2. LOOP chính (0x03–0x0B):
   0x03: LDA FN2 ; Tải giá trị Fibonacci thứ hai (FN2) vào AC
   0x04: STO TEMP ; Lưu FN2 vào TEMP
   0x05: ADD FN1 ; Cộng FN1 → AC = FN1 + FN2

0x06: STO FN2 ; Ghi kết quả mới vào FN2
0x07: LDA TEMP ; Lấy lại FN2 cũ từ TEMP
0x08: STO FN1 ; Ghi vào FN1 (dịch FN2 cũ thành FN1)
0x09: XOR LIMIT ; Kiểm tra nếu FN2 == LIMIT → kết quả XOR sẽ là 0
0x0A: SKZ ; Nếu bằng LIMIT → skip lệnh tiếp theo
0x0B: JMP 0x03 ; Nhảy lại vòng lặp LOOP

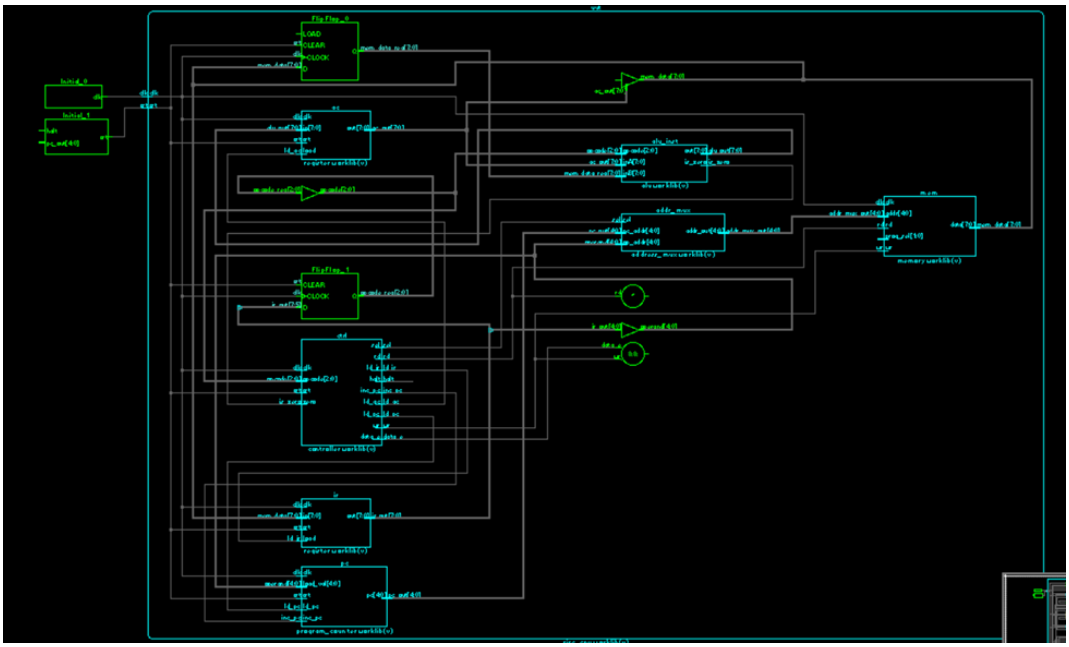
3. Kết thúc chương trình
   0x0C: HLT ; Nếu đã đạt LIMIT thì kết thúc chương trình

Phân tích waveform:

1. Biến pc out (Program Counter)
   Giá trị nhảy từ 0x00 → 0x03 ngay từ đầu, xác nhận lệnh JMP 0x03 hoạt động chính xác. Các giá trị PC tăng dần từ 0x03 đến 0x0B, rồi quay lại 0x03, thể hiện vòng lặp của chương trình. Đến cuối cùng, khi FN2 đạt LIMIT = 0x90, ta thấy PC nhảy qua JMP và thực thi HLT tại địa chỉ 0x0C.
2. Accumulator (ac out) và ALU (alu out)
   Accumulator lần lượt chứa các giá trị trong chuỗi Fibonacci: 0x00, 0x01, 0x01, 0x02, 0x03, 0x05, 0x08, ..., đến 0x90. ALU thực hiện phép cộng giữa FN1 và FN2, kết quả được lưu về FN2, phản ánh trong tín hiệu alu out.
3. Dữ liệu bộ nhớ (mem data, mem data reg)
   Cho thấy đúng giá trị đang được đọc và ghi tại các địa chỉ tương ứng: FN1 (0x1A), FN2 (0x1B), TEMP (0x1C), LIMIT (0x1D). Sau mỗi vòng lặp, giá trị tại 0x1A và 0x1B thay đổi để phản ánh dãy Fibonacci.
4. Kiểm tra điều kiện dừng (is zero, opcode, operand)
   Sau khi thực hiện XOR giữa FN2 và LIMIT, khi hai giá trị trùng nhau, XOR trả về 0 → is zero = 1. Tín hiệu SKZ được kích hoạt, khiến lệnh JMP LOOP bị bỏ qua và PC đi tiếp đến HLT.
5. Số vòng lặp
   Có thể quan sát chương trình thực hiện đúng số vòng lặp cần thiết cho đến khi FN2 = 0x90 (144 thập phân). Với FN1 ban đầu = 0x01 và FN2 = 0x00, các số Fibonacci tính được là: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 (0x90) → dừng.

5.2 Đánh giá thông số hiệu năng:

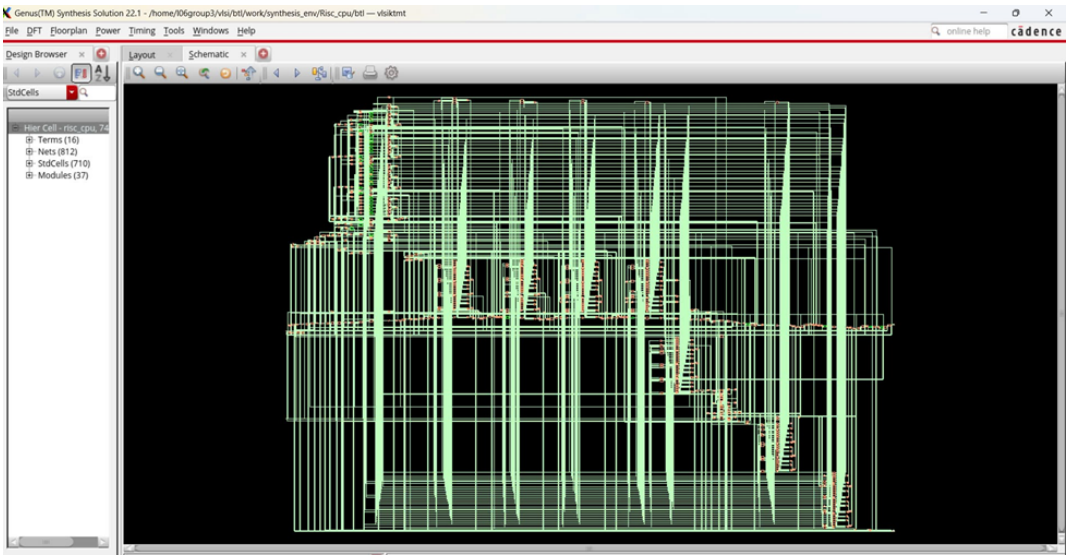
- Kết quả schematic:



Schematic trên bao gồm các khối `flip-flop`, `alu`, `mem`, `controller`, `IR`, `PC`, `addr_mux`, `initial`, cũng như cách kết nối giữa các module, và đường đi của tín hiệu trong thiết kế phần cứng.

Mục đích của schematic giúp người lập trình viên hiểu hơn về cấu trúc phần cứng của mình, giúp kiểm tra các kết nối, đảm bảo tín hiệu đi đúng hướng, các module kết nối logic. Ngoài ra nó còn giúp cho việc debug trở nên dễ dàng, thuận tiện hơn.

- Kết quả netlist:



Quá trình synthesis đã chuyển mã verilog thành phần cứng thực tế (gates, flip-flop, multiplexer...), gọi là netlist. Mục đích là để chuẩn bị cho việc triển khai thực tế lên FPGA hoặc ASIC, hoặc kiểm tra lỗi logic, thiếu kết nối trong mô tả HDL.

- Báo cáo area:
Báo cáo tổng hợp được tạo bởi phần mềm Cadence Genus Synthesis Solution, sử dụng chế

độ tổng hợp theo thư viện vật lý (physical library) với điều kiện hoạt động chậm (slow), tương ứng với các trường hợp xấu nhất (worst-case). Các thông số như sau:

  - Module tổng hợp: risc cpu
  - Tổng số cell: 747.Là tổng số phần tử logic (gồm các cổng logic, flip-flop, mux, ...) được sử dụng để hiện thực module.
  - Cell Area: 2759.598. Đây là diện tích chiếm bởi các cell logic theo đơn vị tính của thư viện vật lý (thường là micromet vuông, tùy theo công nghệ).
  - Net Area: 842.321. Biểu thị diện tích chiếm bởi các kết nối dây (wires) giữa các phần tử logic.
  - Tổng diện tích (Total Area): 3601.919. Tổng của Cell Area và Net Area. Thông số này rất quan trọng khi ước lượng khả năng triển khai trên chip hoặc FPGA.

- Báo cáo timing:
  Báo cáo timing (thời gian lan truyền tín hiệu) cho một đường truyền (path) trong module risc cpu có các thông số tổng quan như sau:
  - Path: từ ctrl state reg reg[2]/CK đến ir out reg[2]/D
  - Thời gian yêu cầu (Required Time): 4531 ps
  - Thời gian thực tế (Arrival Time): 4452 ps
  - Slack: +79 ps. Thỏa điều kiện Setup Timing (không bị vi phạm timing)

  Chi tiết timing:
  - `ctrl_state_reg_reg[2]`: thuộc cổng `DFFRX1` (flip-flop), có delay là 498ps, là điểm bắt đầu của path.
  - Các cổng logic: `MX2X1`, `AND`, `AOI221`, `NAND3`, `OR4`, `TBUFX4`,... được dùng để xử lý tín hiệu, có delay tăng dần lên đến 3377.
  - `TBUFX4`, `INVXL`, ...: là buffer cuối cùng (cổng Buffer & inverter) có delay từ 3566 → 4452.
  - `ir_out_reg[2]/D`: là cổng flip-flop, cũng là cổng kết thúc.

- Báo cáo qor:
  - Thông số thời gian (Timing):
  - Chu kỳ xung đồng hồ (clk): 5000 ps (tương đương 200 MHz)
  - Slack nhỏ nhất trên tất cả các nhóm: +79.2 ps
  - Tổng số lỗi thời gian (TNS): 0 ps
  - Số lượng đường truyền bị vi phạm: 0

Kết luận: Thiết kế không có lỗi timing. Tất cả các đường truyền đều thỏa mãn yêu cầu thời gian, đảm bảo hoạt động chính xác ở tần số thiết kế.
Thống kê cấu trúc:

- Tổng số instance (cell): 747
- Sequential (có xung): 328
- Combinational (logic tổ hợp): 419
- Số lượng khối phân cấp (hierarchical instance): 37

Diện tích chiếm dụng (Area):

- Diện tích cell logic (Cell Area): 2759.598 m^2
- Diện tích liên kết (Net Area): 842.321 m^2
- Tổng diện tích: 3601.919 m^2

Công suất tiêu thụ (Power):

- Công suất rò (Leakage Power): 91.182 nW
- Công suất động (Dynamic Power): 76.743 µW
- Tổng công suất tiêu thụ: 76.834 µW

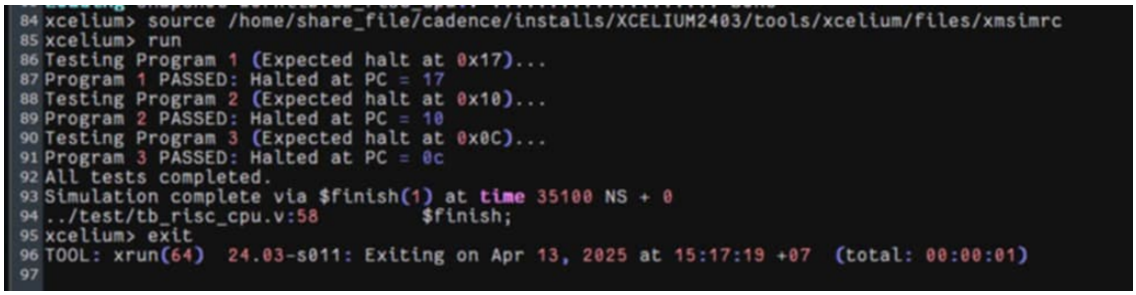
Công suất tương đối thấp, phù hợp với một CPU đơn giản có kích thước nhỏ. Thông tin fanout và clock gating:

- Max fanout: 40 (tại clk)
- Min fanout: 1
- Fanout trung bình: 2.6
- Số lượng clock gating logic: 37
- Tỷ lệ term-to-net: 3.67
- Tỷ lệ term-to-instance: 3.70

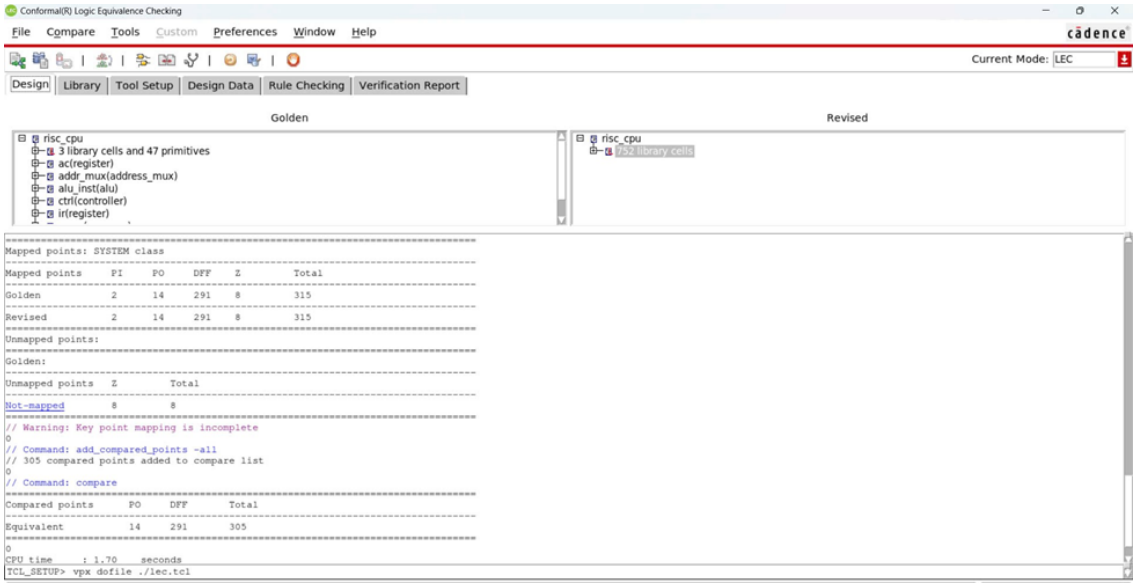
Hiệu suất tổng hợp:

- Thời gian tổng hợp thực tế: 34.2 giây
- Tổng thời gian trôi qua: 43 giây
- Đỉnh bộ nhớ sử dụng (Genus): 1640 MB

Kết quả khi chạy test:



5.3 Kiểm tra lỗi sau khi tổng hợp Logic Equivalence Checking (LEC)::



Mục đích: Đảm bảo rằng bản thiết kế RTL và bản netlist sau tổng hợp (sử dụng Genus Synthesis Solution) có cùng chức năng logic. Phát hiện các lỗi logic tiềm ẩn do quá trình tổng hợp gây ra (ví dụ: mất mạch, sai nối, hoặc lỗi timing optimization ảnh hưởng chức năng).
Cấu hình và dữ liệu:

- Module kiểm tra: risc cpu
- Golden: Thiết kế gốc ở mức RTL bao gồm các module con như alu, ctrl, ir, register, v.v.
- Revised: Netlist sau tổng hợp, bao gồm 152 thư viện cell đã mapping.
- Công cụ sử dụng: Cadence Conformal LEC
- Tập lệnh sử dụng: lec.tcl chứa các lệnh so sánh giữa Golden và Revised

Kết quả kiểm tra:
Dưới đây là các số liệu thu được từ quá trình kiểm tra tương đương logic:

Thông số	**Golden**	**Revised**
Primary input(PI)	14	14
Primary outputs(PO)	291	291
Flip-flop(FF)	8	8
Tổng số điểm logic	315	315

Nhận xét:

- Có tổng cộng 305 điểm logic đã được so sánh thành công.
- Không có báo cáo lỗi mismatch hoặc bất kỳ cảnh báo sai khác nghiêm trọng nào.
- Một cảnh báo nhỏ được ghi nhận: Warning: Key point mapping is incomplete, tuy nhiên điều này không ảnh hưởng đến tính toàn vẹn logic tổng thể vì các điểm thiết yếu đã được ánh xạ và kiểm tra đầy đủ.

6 Kết luận

6.1 Tổng kết

Quá trình thực hiện đề tài "Thiết kế RISC CPU đơn giản" đã tạo cơ hội cho nhóm nghiên cứu sâu về kiến trúc RISC cũng như ứng dụng ngôn ngữ mô tả phần cứng Verilog HDL vào hiện thực hóa một hệ thống CPU đơn giản. Các khái niệm và nguyên tắc thiết kế mạch số cơ bản đã được hiểu và vận dụng qua việc phân tích, triển khai và kiểm thử các khối chức năng chính:

- Program Counter (PC): Đóng vai trò chỉ định vị trí của lệnh tiếp theo cần thực thi, đảm bảo tính liên tục của quá trình xử lý.
- Address Mux, Memory và Instruction Register (IR): Hỗ trợ quá trình nạp và giải mã lệnh, góp phần vào việc truy xuất và lưu trữ thông tin hiệu quả.
- ALU và Accumulator (AC): Thực hiện các phép tính số học và logic cơ bản, đảm bảo xử lý dữ liệu đơn giản nhưng chính xác.
- Controller: Điều phối các tín hiệu điều khiển thông qua một máy trạng thái hữu hạn, giúp việc sắp xếp và thực thi chu trình lệnh được thực hiện một cách khoa học và nhất quán.

Thông qua việc thực hiện và mô phỏng các trường hợp kiểm thử đa dạng (bao gồm cả thuật toán tính dãy Fibonacci), đề tài đã chứng minh được tính ổn định cũng như khả năng đáp ứng các yêu cầu thiết kế ban đầu.

6.2 Hướng phát triển trong tương lai

Dựa trên những kết quả đạt được và kinh nghiệm thu được trong suốt quá trình thực hiện, nhóm nhận thấy có thể mở rộng và cải tiến thiết kế CPU RISC theo các hướng sau:

- Mở rộng tập lệnh:
  Nghiên cứu và tích hợp thêm các lệnh mới như lệnh dịch bit, so sánh hay các phép toán phức tạp, nhằm nâng cao khả năng xử lý và tính ứng dụng của CPU trong các hệ thống thực tiễn.
- Tối ưu hoá kiến trúc:
  Cải tiến các khối chức năng như ALU, Controller và bộ nhớ thông qua việc áp dụng các kỹ thuật tối ưu hóa mạch, triển khai cơ chế pipeline và xử lý song song để gia tăng tốc độ xử lý và hiệu năng tổng thể.
- Mở rộng ứng dụng và tích hợp:
  Khai thác tiềm năng tích hợp CPU RISC vào các hệ thống nhúng phức tạp, ứng dụng trong các thiết bị IoT, hệ thống tự động hóa hay xử lý tín hiệu số, qua đó mở rộng phạm vi ứng dụng và cải thiện hiệu năng hệ thống.

6.3 Các khó khăn và thách thức

Trong quá trình thực hiện dự án, nhóm có gặp phải một số vấn đề cần khắc phục cũng như những thách thức đáng lưu ý:

- Đồng bộ dữ liệu tín hiệu:
  Việc đồng bộ giữa các khối, đặc biệt là giữa tín hiệu clock (clk) và reset (rst), đòi hỏi sự cẩn trọng trong thiết kế để tránh các lỗi không mong muốn trong quá trình thực thi lệnh.
- Tối ưu hoá code Verilog:
  Các vấn đề về tối ưu hóa code HDL khi tích hợp các module độc lập đã gây ra một số khó khăn, đặc biệt trong việc xử lý các tình huống đặc biệt như tràn số hay giao tiếp giữa tín hiệu bất đồng bộ.

- Tích hợp và kiểm thử hện thống:
  Việc kết hợp các module độc lập và tiến hành kiểm thử qua các chương trình mô phỏng đòi hỏi công tác kiểm tra kỹ lưỡng nhằm đảm bảo hệ thống hoạt động ổn định và đáp ứng chính xác các yêu cầu đặt ra.