ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

Thiết kế Vi mạch (CO3097)

THIẾT KẾ RISC CPU ĐƠN GIẢN

Giảng viên hướng dẫn: Nguyễn Thành Lộc

Sinh viên thực hiện: Ngô Quang Tùng 2213869
Nguyễn Thanh Tùng 2112711

Thành phố Hồ Chí Minh, Tháng 05 Năm 2025

Danh sách thành viên và phân chia công việc

STT	Họ và tên	MSSV	Tỉ lệ hoàn thành
1	Ngô Quang Tùng	2213869	100%
2	Nguyễn Thanh Tùng	2112711	0%

Mục lục

1 Giới thiệu đề tài

Bộ xử lý trung tâm (CPU - Central Processing Unit) là thành phần cốt lõi của hầu hết các hệ thống tính toán hiện đại, đóng vai trò "bộ não"thực thi các chương trình và xử lý dữ liệu. Trong bối cảnh phát triển không ngừng của công nghệ vi mạch, việc hiểu rõ nguyên lý hoạt động và quy trình thiết kế CPU trở nên vô cùng quan trọng đối với kỹ sư ngành Kỹ thuật Máy tính.

Đề tài "Thiết kế RISC CPU đơn giản"tập trung vào việc nghiên cứu và xây dựng một bộ xử lý theo kiến trúc RISC (Reduced Instruction Set Computer). Kiến trúc RISC đặc trưng bởi việc sử dụng một tập lệnh đơn giản hóa, tối ưu hóa cho tốc độ thực thi. Dự án này là cơ hội để chúng em áp dụng các kiến thức đã học trong môn Thiết kế Vi mạch số để giải quyết một bài toán thiết kế thực tế.

Mục tiêu của đề tài là thiết kế một CPU RISC cơ bản với các đặc tả cụ thể: sử dụng mã lệnh (opcode) 3-bit, cho phép định nghĩa tối đa 8 lệnh khác nhau, và sử dụng toán hạng/địa chỉ 5-bit, cung cấp không gian địa chỉ 32 ô nhớ. Bộ xử lý sẽ hoạt động đồng bộ theo tín hiệu xung nhịp (clock), có cơ chế khởi động lại (reset) và dừng hoạt động khi nhận tín hiệu HALT. Thông qua việc hoàn thành đề tài này, nhóm mong muốn rèn luyện kỹ năng thiết kế mạch số phức tạp bằng ngôn ngữ mô tả phần cứng, kỹ năng phân tích, chia nhỏ bài toán và kiểm thử hệ thống.

1.1 Công cụ và Thiết bị sử dụng

Công cụ:

- Ngôn ngữ mô tả phần cứng: Verilog HDL là ngôn ngữ chính được sử dụng để mô tả hành vi và cấu trúc của các thành phần trong CPU.
- GitHub: Được sử dụng để quản lý mã nguồn (source code), lưu trữ phiên bản, và hỗ trợ làm việc nhóm hiệu quả. GitHub giúp theo dõi các thay đổi trong mã và dễ dàng phối hợp giữa các thành viên.

Thiết bị sử dụng:

- Kit Arty-Z7 20: Đây là board phát triển FPGA dựa trên chip Xilinx Zynq-7000, được sử dụng để triển khai và kiểm tra thực tế thiết kế RISC CPU. Kit này hỗ trợ giao diện lập trình linh hoạt và có nhiều tính năng hữu ích để thực hiện các dự án liên quan đến xử lý tín hiệu số.

1.2 Chức năng chính của Sản phẩm

Sản phẩm cuối cùng của dự án là một mô hình CPU RISC đơn giản, có khả năng thực hiện các chức năng cơ bản sau:

Thực thi chu trình lệnh: CPU có khả năng thực hiện chu trình nạp lệnh (fetch), giải mã lệnh (decode), thực thi lệnh (execute) và lưu kết quả (write-back) một cách tuần tự.

Tập lệnh cơ bản: Hỗ trợ thực thi 8 lệnh cơ bản đã được định nghĩa, bao gồm các lệnh xử lý số học/logic (ADD, AND, XOR), lệnh di chuyển dữ liệu (LDA, STO), lệnh điều khiển luồng (JMP, SKZ - Skip if Zero) và lệnh dừng chương trình (HLT).

Tương tác với bộ nhớ: Có khả năng đọc lệnh và đọc/ghi dữ liệu từ một khối bộ nhớ (Memory) với không gian địa chỉ 5-bit và dữ liệu 8-bit.

Các thành phần cốt lõi: Bao gồm các khối chức năng chính như Bộ đếm chương trình (Program Counter), Bộ chọn địa chỉ (Address Mux), Bộ nhớ (Memory), Thanh ghi lệnh (Instruction Register), Thanh ghi tích lũy (Accumulator Register), Đơn vị Số học và Logic (ALU), và Bộ điều khiển (Controller).

2 Cơ sở lý thuyết

2.1 Kiến trúc CPU RISC

Kiến trúc CPU RISC trong đề tài thiết kế một CPU đơn giản dựa trên triết lý Reduced Instruction Set Computer (RISC), với tập lệnh tinh gọn (8 lệnh, opcode 3-bit, toán hạng 5-bit), xử lý thanh ghi là chủ đạo, và điều khiển cứng. CPU gồm các khối chính: Program Counter (quản lý địa chỉ lệnh), ALU (thực hiện phép toán), Controller (điều phối qua 8 trạng thái), Memory (lưu trữ lệnh/dữ liệu), Register (lưu tạm), và Address Mux (chọn địa chỉ). Thiết kế đơn giản, đồng bộ với xung clock, phù hợp cho mô phỏng Verilog và dễ mở rộng (pipeline, xử lý hazard). So với CISC, RISC hiệu quả hơn, ít phức tạp, phù hợp cho hệ thống nhúng.

2.2 Các khối chức năng trong CPU

2.2.1 Program Counter

- Program Counter là bộ đếm quan trọng dùng để đếm địa chỉ lệnh của chương trình, ngoài ra có thể đếm trạng thái chương trình.
- Counter hoạt động khi có xung lên của clk.
- Reset kích hoạt mức cao, bộ đếm trở về 0.
- Counter có độ rộng số đếm là 5-bit.
- Counter có chức năng load một số bất kỳ vào bộ đếm; nếu không, bộ đếm sẽ hoạt động bình thường (tăng dần).

2.2.2 Address Mux

- Khối Address Mux với chức năng của Mux sẽ chọn giữa địa chỉ lệnh trong giai đoạn nạp lệnh và địa chỉ toán hạng trong giai đoạn thực thi lệnh.
- Mux có độ rộng mặc định là 5-bit.
- Độ rộng cần sử dụng parameter để vẫn thay đổi được nếu cần.

2.2.3 ALU

- ALU thực hiện các phép toán số học và logic dựa trên opcode 3-bit.
- ALU hỗ trợ 8 phép toán trên toán hạng 8-bit (inA và inB).
- Kết quả đầu ra gồm 8bit output và 1bit is_zero (bất đồng bộ) để kiểm tra inA có bằng 0 hay không.
- Opcode 3bit quyết định phép toán thực hiện: HLT, SKZ, ADD, AND, XOR, LDA, STO, JMP.

Opcode	Mã	Hoạt động
HLT	000	Dừng hoạt động chương trình
SKZ	001	Trước tiên sẽ kiểm tra kết quả của ALU có bằng 0 hay không, nếu bằng 0 thì sẽ bỏ qua câu lệnh tiếp theo, ngược lại sẽ tiếp tục thực thi như bình thường
ADD	010	Cộng giá trị trong Accumulator vào giá trị bộ nhớ địa chỉ trong câu lệnh và kết quả được trả về Accumulator.
AND	011	Thực hiện AND giá trị trong Accumulator và giá trị bộ nhớ địa chỉ trong câu lệnh và kết quả được trả về Accumulator.
XOR	100	Thực hiện XOR giá trị trong Accumulator và giá trị bộ nhớ địa chỉ trong câu lệnh và kết quả được trả về Accumulator.
LDA	101	Thực hiện đọc giá trị từ địa chỉ trong câu lệnh và đưa vào Accumulator.
STO	110	Thực hiện ghi dữ liệu của Accumulator vào địa chỉ trong câu lệnh.
JMP	111	Lệnh nhảy không điều kiện, nhảy đến địa chỉ đích trong câu lệnh và tiếp tục thực hiện chương trình.

2.2.4 Controller

- Controller quản lý những tín hiệu điều khiển của CPU. Bao gồm nạp và thực thi lệnh.
- Controller phải hoạt động khi có xung lên của clk.
- Tín hiệu rst đồng bộ và kích hoạt mức cao.
- Tín hiệu đầu vào opcode 3-bit tương ứng với ALU.
- Controller có 7 output như bảng sau::

Output	Function
sel	select
rd	memory read
ld_ir	load instruction register
halt	halt
inc_pc	increment program counter
ld_ac	load accumulator
ld_pc	load program counter
wr	memory write
data_e	data enable

- Controller có 8 trạng thái hoạt động liên tục trong 8 chu kỳ clk theo thứ tự: INST_ADDR, INST_FETCH, INST_LOAD, IDLE,OOP_ADDR,OP_FETCH, ALU_OP, STORE. Trạng thái reset là INST_ADDR.
- Output của Controller dựa theo trạng thái và opcode như bảng sau:

Outputs	Phase								Notes
	INST_ ADDR	INST_ FETCH	INST_ LOAD	IDLE	OP_ ADDR	OP_ FETCH	ALU_ OP	STORE	
sel	1	1	1	1	0	0	0	0	ALU OP = 1 if opcode is ADD, AND, XOR or LDA
rd	0	1	1	1	0	ALUOP	ALUOP	ALUOP	
ld_ir	0	0	1	1	0	0	0	0	
halt	0	0	0	0	HALT	0	0	0	
inc_pc	0	0	0	0	1	0	SKZ & & zero	0	
ld_ac	0	0	0	0	0	0	0	ALUOP	
ld_pc	0	0	0	0	0	0	JMP	JMP	
wr	0	0	0	0	0	0	0	STO	
data_e	0	0	0	0	0	0	STO	STO	

2.2.5 Register

- Tín hiệu đầu vào có độ rộng 8-bit.
- Tín hiệu reset đồng bộ và kích hoạt mức cao.

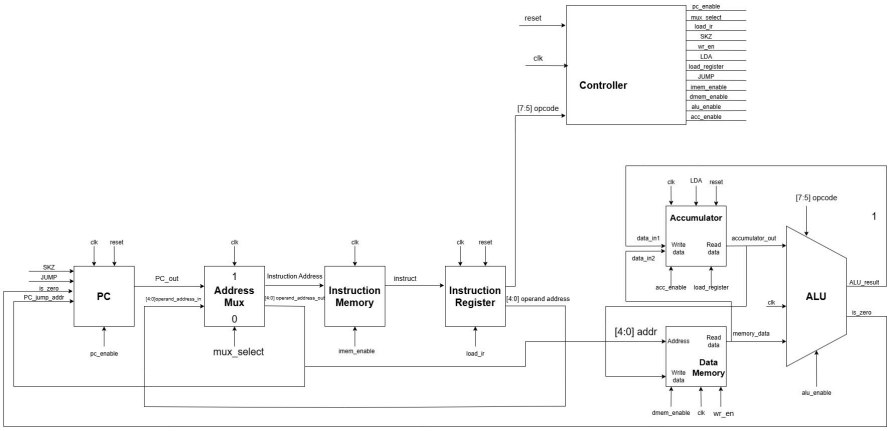
- Register hoạt động khi có xung lên của clk.
- Khi có tín hiệu load, giá trị đầu vào sẽ chuyển đến đầu ra.
- Ngược lại, giá trị đầu ra sẽ không đổi.

2.2.6 Memory

- Memory sẽ lưu trữ instruction và data.
- Memory cần được thiết kế tách riêng chức năng đọc/ghi bằng cách sử dụng Single bidirectional data port. Không được đọc và ghi cùng lúc.
- Địa chỉ 5-bit và dữ liệu 8-bit.
- Có 1-bit tín hiệu cho phép đọc/ghi. - Memory hoạt động khi có xung lên của clk.

3 Thiết kế hệ thống

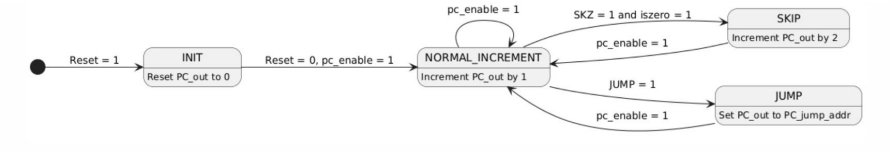
3.1 Sơ đồ khối tổng thể



Hình 1: *Sơ đồ khối của toàn bộ mạch*

3.2 Mô tả chi tiết các khối chức năng

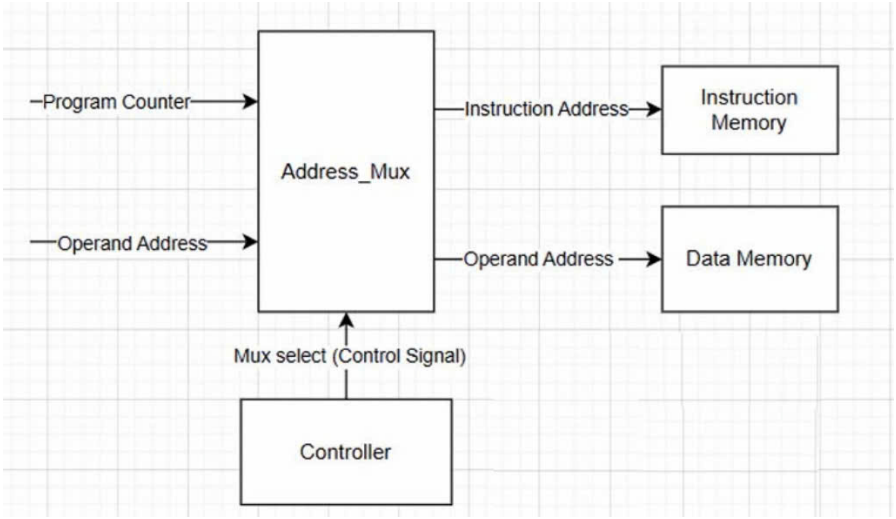
3.2.1 Program Counter



Hình 2: *Sơ đồ Program Counter (PC)*

- NORMAL INCREMENT: Thanh ghi PC tự động tăng khi kết thúc mỗi câu lệnh.
- SKIP AND JUMP: Khi có các lệnh đặc biệt như SKZ và JUMP, thanh ghi PC sẽ chờ tín hiệu điều khiển và tính toán địa chỉ trước khi thực hiện nhảy.

3.2.2 Address Mux



Hình 3: *Sơ đồ khối Address Mux*

Address Mux có vai trò là bộ chọn địa chỉ giữa địa chỉ lệnh (Instruction Address) và địa chỉ toán hạng (Operand Address).

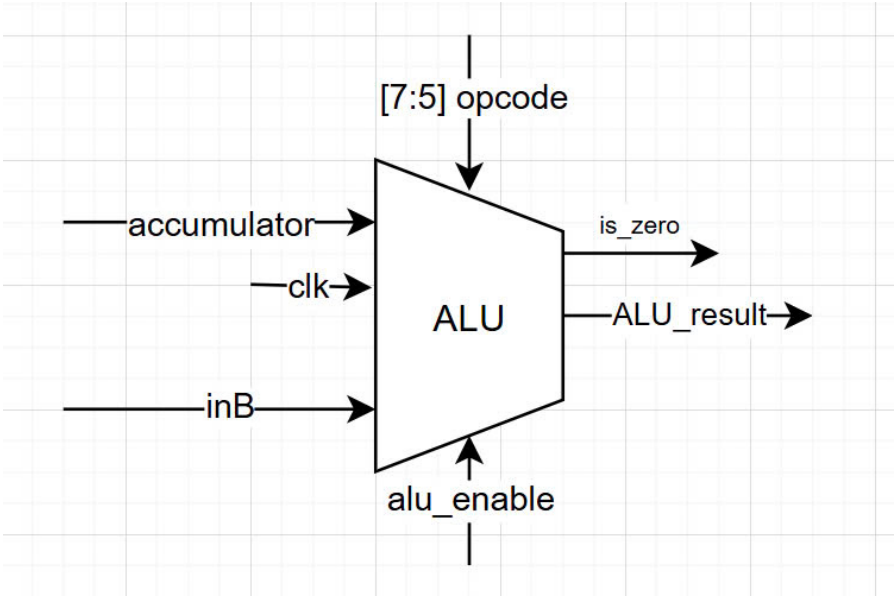
Đầu vào

- Program Counter (PC): cung cấp địa chỉ của lệnh cần lấy từ Instruction Memory.
- Operand Address: là địa chỉ của dữ liệu cần truy xuất từ Data Memory.
- Tín hiệu điều khiển Mux_select từ khối Controller.

Hoạt động

- Nếu CPU đang ở giai đoạn Fetch, Address Mux chọn đầu vào từ PC để truy cập Instruction Memory.
- Nếu CPU đang ở giai đoạn Execute, Address Mux chọn Operand Address để truy cập Data Memory.
- Cho phép CPU sử dụng chung một đường địa chỉ cho hai mục đích:
  - Truy xuất lệnh.
  - Đọc/ghi dữ liệu.

3.2.3 ALU



Hình 4: *Sơ đồ khối ALU*

- Thực hiện các phép toán số học và logic.
- Các phép toán phụ thuộc vào opcode 3-bit của lệnh, bao gồm:
  - **HLT (000)**: Dừng hoạt động chương trình.
  - **SKZ (001)**: Kiểm tra kết quả ALU có bằng 0 hay không. Nếu bằng 0, bỏ qua lệnh tiếp theo; nếu không, tiếp tục thực hiện.
  - **ADD (010)**: Cộng giá trị trong *Accumulator* với giá trị từ bộ nhớ (*Memory*) và lưu kết quả vào *Accumulator*.
  - **AND (011)**: Thực hiện phép toán *AND* giữa *Accumulator* và *Memory*, lưu kết quả vào *Accumulator*.
  - **XOR (100)**: Thực hiện phép toán *XOR* giữa *Accumulator* và *Memory*, lưu kết quả vào *Accumulator*.
  - **LDA (101)**: Đọc giá trị từ *Memory* vào *Accumulator*.
  - **STO (110)**: Ghi giá trị từ *Accumulator* vào *Memory*.
  - **JMP (111)**: Lệnh nhảy không điều kiện đến địa chỉ mục tiêu trong lệnh.

Đầu vào

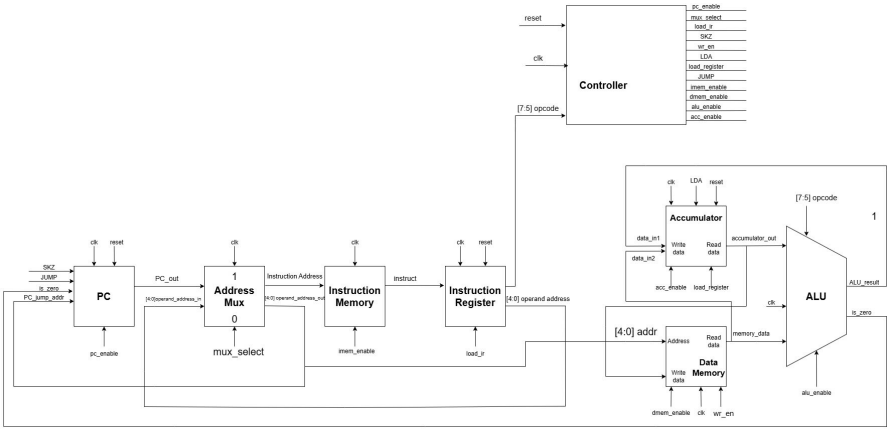
- Hai toán hạng 8-bit: `Accumulator` và `inB`.
- Mã `opcode` 3-bit để xác định loại phép toán.

- Tín hiệu `alu_enable` được sinh ra từ khối *Controller* để kích hoạt ALU hoạt động.
- `Controller` phải hoạt động khi có xung lên của `clk`.

Đầu ra

- Kết quả 8-bit.
- Cờ `is_zero` (1-bit) cho biết kết quả ALU có bằng 0 hay không.

3.2.4 Controller



Hình 5: *Sơ đồ Program Counter (PC)*

Khối Controller có nhiệm vụ chính là quản lý và điều phối hoạt động của các thành phần trong hệ thống dựa trên các tín hiệu điều khiển và `opcode` của lệnh hiện tại.

Đầu vào

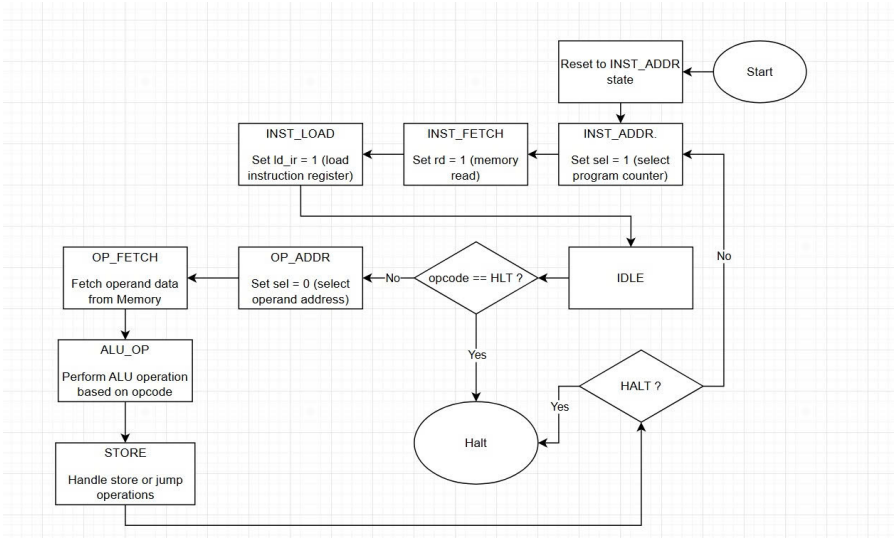
- Hoạt động khi có xung lên của tín hiệu `clk`.
- Tín hiệu reset đồng bộ và kích hoạt ở mức cao.
- Tín hiệu opcode 3-bit tương ứng với lệnh xử lý trong ALU.

Đầu ra

- `pc_enable`: Kích hoạt khối *Program Counter (PC)*.
- `mux_select`: Điều khiển khối *Address Mux* để chọn giữa địa chỉ lệnh hoặc địa chỉ toán hạng.

- `imem_enable`: Kích hoạt khối *Instruction Memory* để nạp lệnh.
- `load_ir`: Kích hoạt khối *Instruction Register* để cho phép giải mã lệnh.
- Các tín hiệu `SKZ` và `JUMP`: Kết hợp cùng với `pc_enable` để điều chỉnh địa chỉ của `PC` phù hợp với từng loại lệnh.
- `dmem_enable`: Kích hoạt khối *Data Memory*.
- `wr_en`: Cho phép khối *Data Memory* thực hiện thao tác đọc hoặc ghi.
- `acc_enable`: Kích hoạt khối *Accumulator Register*.
- `LDA` kết hợp với `acc_enable`: Cho phép lệnh `LDA` ghi vào khối *Accumulator*.
- `load_register` kết hợp với `acc_enable`: Cho phép ghi vào khối *Accumulator Register* (ngoại trừ lệnh `LDA`).
- `alu_enable`: Kích hoạt khối *ALU* để thực hiện các phép toán.

3.3 Flowchart



Hình 6: *Flowchart*

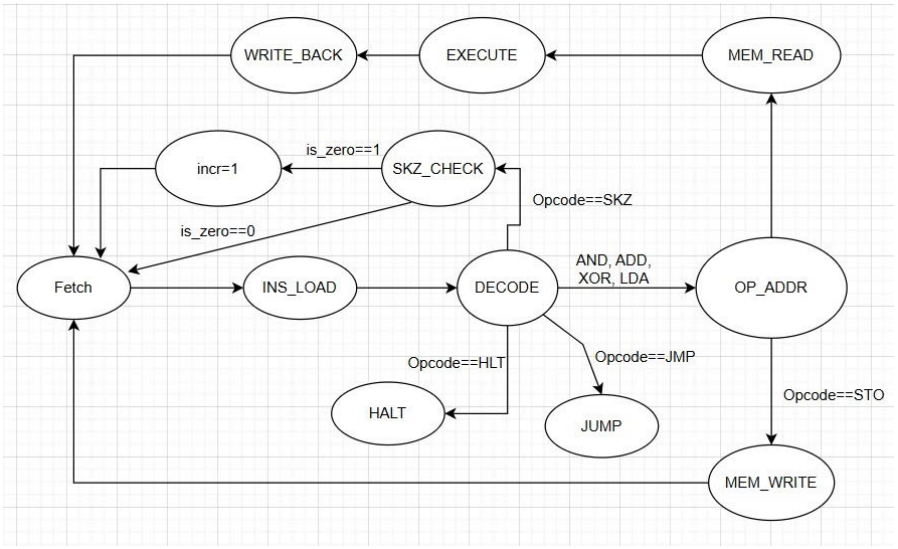
Sơ đồ khối biểu diễn luồng hoạt động của bộ xử lý RISC (Reduced Instruction Set Computer) đơn giản được thiết kế bằng ngôn ngữ mô tả phần cứng Verilog, như đã nêu trong tài liệu. Bộ xử lý này thực thi các lệnh với opcode 3 bit và địa chỉ toán hạng 5 bit, hỗ trợ 8 loại lệnh khác nhau và 32 địa chỉ bộ nhớ. CPU hoạt động đồng bộ với xung đồng hồ (`clk`) và sẽ dừng lại khi gặp lệnh `HALT`. Sơ đồ này minh họa cho bộ điều khiển (Controller) dưới dạng máy trạng thái, điều phối

chu trình *fetch–decode–execute*, đồng thời thể hiện sự tương tác giữa các khối chức năng chính: Bộ đếm chương trình (Program Counter), bộ chọn địa chỉ (Address MUX), bộ nhớ (Memory), thanh ghi lệnh (Instruction Register), thanh ghi tích lũy (Accumulator), ALU, và bộ điều khiển (Controller).

Cấu trúc và thành phần: Sơ đồ khối sử dụng các ký hiệu tiêu chuẩn để mô tả các bước xử lý, điểm quyết định và điểm kết thúc:

- **Hình oval:** Biểu thị điểm bắt đầu và kết thúc (ví dụ: *Start*, *Halt*).
- **Hình chữ nhật:** Đại diện cho các bước xử lý, tương ứng với các trạng thái và hành động của bộ điều khiển (ví dụ: thiết lập tín hiệu điều khiển như `sel`, `rd`, `ld ir`).
- **Hình thoi:** Biểu thị các điểm ra quyết định, chẳng hạn như kiểm tra giá trị opcode hoặc điều kiện dừng (`halt`).
- **Mũi tên:** Thể hiện luồng điều khiển giữa các trạng thái và quyết định, tạo thành một vòng lặp quay lại bước nạp địa chỉ lệnh (`INST ADDR`), trừ khi gặp lệnh dừng.

3.4 Máy trạng thái hữu hạn



Hình 7: *Finite State Machine Figure*

Từ Hình 7, biểu đồ máy trạng thái hữu hạn (FSM) thể hiện luồng điều khiển của một bộ xử lý đơn giản. Quá trình bắt đầu từ trạng thái `Fetch`, nơi lệnh được truy xuất từ bộ nhớ. Từ `Fetch`, FSM chuyển sang trạng thái `INS_LOAD` để tải lệnh

vào thanh ghi lệnh. Sau đó, luồng điều khiển tiếp tục đến trạng thái `DECODE`, nơi mã lệnh (opcode) được giải mã.

Dựa vào giá trị của opcode, FSM sẽ chuyển sang các trạng thái khác nhau. Nếu opcode là `HLT`, FSM sẽ đi đến trạng thái `HALT` và dừng thực thi. Nếu opcode là `JMP`, nó chuyển sang trạng thái `JUMP` để thực hiện lệnh nhảy. Đối với các opcode như `AND`, `ADD`, `XOR` hoặc `LDA`, FSM sẽ tiếp tục đến trạng thái `OP_ADDR` để xử lý địa chỉ toán hạng.

Từ trạng thái `OP_ADDR`, FSM có thể chuyển đến `MEM_READ` để đọc dữ liệu từ bộ nhớ, hoặc đến `MEM_WRITE` nếu opcode là `STO` nhằm ghi dữ liệu vào bộ nhớ. Sau khi hoàn tất việc đọc dữ liệu, FSM tiến vào trạng thái `EXECUTE` để thực hiện phép toán.

Nếu kết quả thực hiện bằng 0 (is_zero = 1), FSM sẽ chuyển sang trạng thái `SKZ_CHECK`; ngược lại (is_zero = 0), nó chuyển trực tiếp sang trạng thái với `incr = 1`. Từ `SKZ_CHECK`, nếu opcode là `SKZ`, FSM sẽ quay trở lại trạng thái `DECODE`; nếu không, nó tiếp tục đến `EXECUTE`. Sau khi thực thi xong, FSM có thể chuyển sang trạng thái `WRITE_BACK` để lưu kết quả, sau đó quay lại `Fetch` để tiếp tục chu trình xử lý lệnh.

4 Hiện thực hệ thống

4.1 Program Counter

- PC được thiết kế để cung cấp địa chỉ lệnh cho khối Instruction Memory trong mỗi chu kỳ lệnh.
- Chức năng chính: tự động tăng giá trị địa chỉ sau mỗi chu kỳ xung clock (trừ khi có lệnh nhảy hoặc bỏ qua).

4.2 Address Mux

- Chức năng chính: dùng để chọn địa chỉ lệnh trong giai đoạn nạp lệnh và địa chỉ toán hạng trong giai đoạn thực thi lệnh.

4.3 Instruction Memory

- Chức năng chính: lấy lệnh được cung cấp từ Instruction Memory để giải mã bao gồm địa chỉ toán hạng và opcode của lệnh.

4.4 Data Memory

- Đối với các lệnh số học, logic: địa chỉ toán hạng được cung cấp để truy xuất dữ liệu đưa vào ALU thực hiện tính toán.
- Đối với lệnh LDA: thực hiện đọc giá trị trong bộ nhớ để nạp vào Accumulator.
- Đối với lệnh STO: thực hiện ghi giá trị từ Accumulator vào bộ nhớ.

4.5 Accumulator

- Accumulator là thanh ghi tạm thời để lưu trữ kết quả.
- Nhận kết quả từ ALU hoặc dữ liệu từ Data Memory.

4.6 ALU

- ALU thực hiện 8 phép toán dựa trên opcode.
- Kết hợp hai dữ liệu (Input A và Input B) và trả kết quả.
- Tín hiệu is_zero báo kết quả có bằng 0 hay không.

4.7 Controller

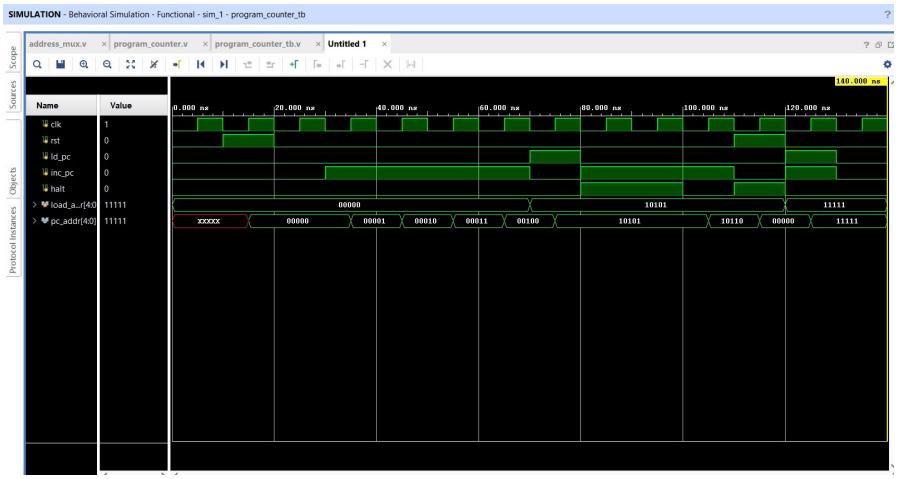
- Controller đảm bảo việc thực thi lệnh đúng thứ tự trong mỗi chu kỳ clock.

Trình tự hoạt động

- Nạp lệnh: PC tăng, địa chỉ được chọn vào Instruction Memory.
- Giải mã: Opcode chuyển tới Controller.
- Thực thi: ALU hoạt động và chuyển kết quả.
- Ghi kết quả: Kết quả lưu trong Accumulator hoặc ghi vào Data Memory.

5 Simulation

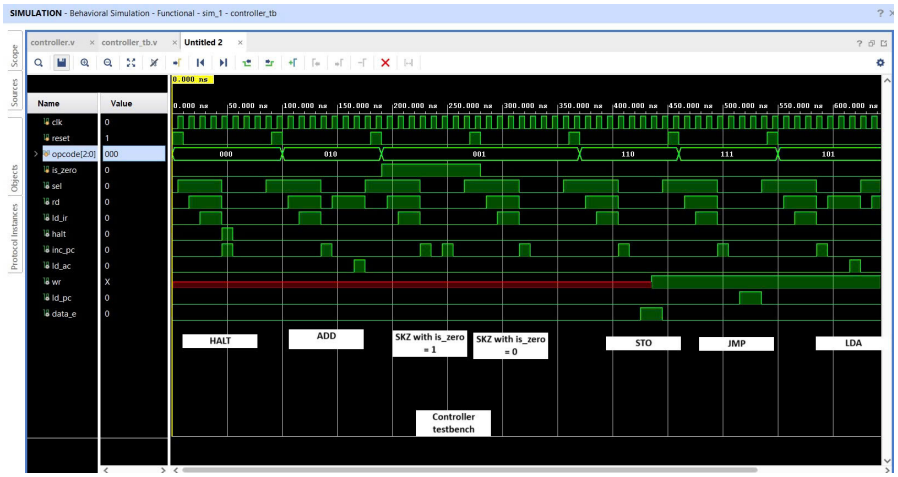
5.1 Program Counter



Hình 8: *Program Counter Testbench*

- **Kiểm thử Reset (0 ns – 20 ns):** Tại thời điểm 10 ns, tín hiệu `rst` được kích hoạt (mức cao), khiến địa chỉ `pc_addr` được đặt lại về `00000` vào lúc 20 ns, xác nhận hành vi khởi tạo đúng.
- **Kiểm thử Tăng (30 ns – 60 ns):** Bắt đầu từ 30 ns, `inc_pc` được kích hoạt, và `pc_addr` tăng từ `00000` lên `00001` tại 40 ns. Quá trình tăng tiếp tục, đạt đến `00100` vào 60 ns, xác minh chức năng tăng hoạt động đúng.
- **Kiểm thử Nạp địa chỉ (70 ns – 80 ns):** Tại 70 ns, tín hiệu `ld_pc` được kích hoạt cùng với `load_addr` được đặt là `10101`. Đến 80 ns, `pc_addr` cập nhật thành `10101`, chứng minh việc nạp địa chỉ thành công.
- **Kiểm thử Dừng (90 ns – 110 ns):** Tại 90 ns, tín hiệu `halt` được kích hoạt trong khi `inc_pc` vẫn hoạt động. `pc_addr` giữ nguyên tại `10101` cho đến 110 ns, khi `halt` được tắt, cho phép tăng lên `10110`, xác nhận cơ chế dừng đã ngăn chặn cập nhật.
- **Reset trong lúc Dừng (110 ns – 120 ns):** Tại 110 ns, `rst` được kích hoạt khi `halt` vẫn đang bật, dẫn đến `pc_addr` được đặt lại về `00000` vào 120 ns, xác nhận ưu tiên của tín hiệu reset.
- **Nạp và Tăng đồng thời (130 ns – 140 ns):** Tại 130 ns, cả `ld_pc` và `inc_pc` đều được bật cùng với `load_addr` là `11111`. `pc_addr` được cập nhật thành `11111` vào 140 ns, xác nhận rằng nạp địa chỉ có ưu tiên cao hơn so với tăng.

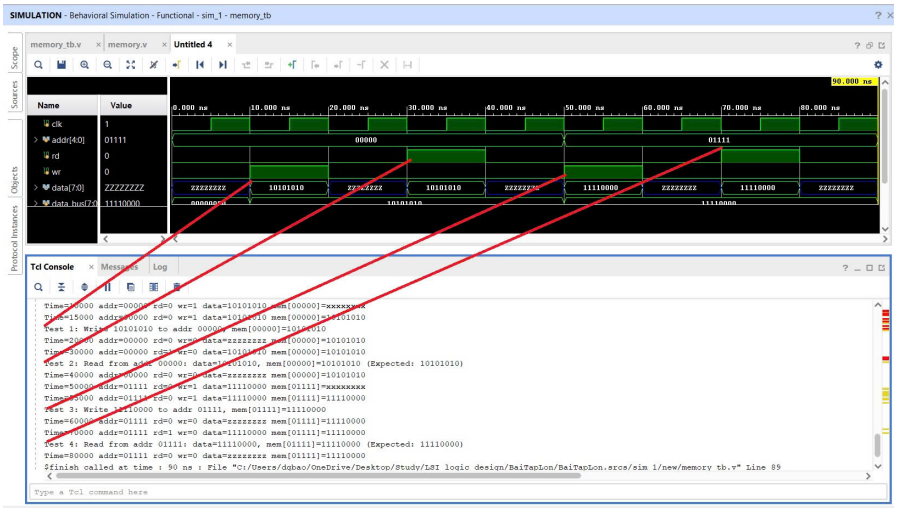
5.2 Controller



Hình 9: *Controller Testbench*

- **HALT (0–400 ns, opcode = 000):** Lệnh HALT tạm dừng hoạt động của CPU. Trong giai đoạn `ALU_OP`, tín hiệu `halt = 1` khiến Program Counter bị đóng băng. Tuy nhiên, theo đặc tả, giá trị mong đợi của `halt` là 0 tại thời điểm này.
- **ADD (400–800 ns, opcode = 010):** Lệnh ADD thực hiện lần lượt các bước: nạp lệnh, nạp toán hạng, thực hiện phép cộng, và ghi kết quả vào Accumulator (`ld_ac`).
- **SKZ với `is_zero = 1` (800–1200 ns, opcode = 001):** Lệnh SKZ kiểm tra cờ `is_zero`. Vì giá trị là 1 nên sẽ bỏ qua lệnh tiếp theo bằng cách tăng Program Counter thêm một bước.
- **SKZ với `is_zero = 0` (1200–1600 ns, opcode = 001):** Trường hợp `is_zero = 0`, lệnh SKZ không bỏ qua, chương trình tiếp tục thực hiện lệnh kế tiếp như bình thường.
- **STO (1600–2000 ns, opcode = 110):** Lệnh STO ghi giá trị từ Accumulator vào bộ nhớ, sử dụng tín hiệu `wr` và `data_e`.
- **JMP (2000–2400 ns, opcode = 111):** Lệnh JMP nạp một địa chỉ mới vào Program Counter bằng cách sử dụng tín hiệu `ld_pc`.
- **LDA (2400–2800 ns, opcode = 101):** Lệnh LDA đọc một giá trị từ bộ nhớ và nạp vào Accumulator thông qua tín hiệu `ld_ac`.

5.3 Memory



Hình 10: *Memory Testbench*

waveform và mô phỏng cho thấy hành vi của mô-đun bộ nhớ cho thiết kế CPU RISC, kiểm tra các hoạt động đọc và ghi của nó. Mô-đun sử dụng địa chỉ 5 bit (addr[4:0]), một bus dữ liệu 8 bit (data[7:0]), một xung nhịp (clk), một tín hiệu điều khiển đọc/ghi (rd wr) và một mảng bộ nhớ (mem). Dưới đây là lời giải thích ngắn gọn dựa trên dạng sóng và nhật ký được cung cấp.

- **Ghi vào địa chỉ `00000` (0–20 ns):**
  - **Log:**

    ```
    Time=10 addr=00000 rd=0 wr=1 data=10101010 mem[00000]=xxxxxxxx
    Test 1: Write 10101010 to addr 00000, mem[00000]=xxxxxxxx
    ```

  - **Mô tả:** Tín hiệu ghi xảy ra tại cạnh lên của xung clock ở thời điểm 10 ns. Bộ nhớ tại địa chỉ `00000` được cập nhật thành `10101010` vào 20 ns.
- **Đọc từ địa chỉ `00000` (30–40 ns):**
  - **Log:**

    ```
    Time=30 addr=00000 rd=1 wr=0 data=10101010 mem[00000]=10101010
    Test 2: Read from addr 00000: data=10101010, mem[00000]=10101010
    ```

  - **Mô tả:** Lệnh đọc tại 30 ns trả về giá trị `10101010`, đúng với kỳ vọng từ lần ghi trước đó.

- **Ghi vào địa chỉ `01111` (50–60 ns):**
  - **Log:**

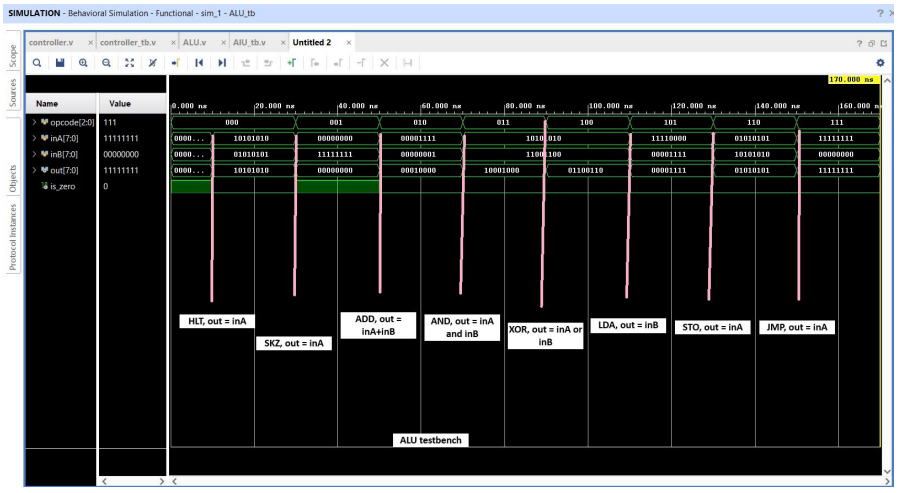
    ```
    Time=50 addr=01111 rd=0 wr=1 data=11110000 mem[01111]=xxxxxxxx
    Test 3: Write 11110000 to addr 01111, mem[01111]=xxxxxxxx
    ```

  - **Mô tả:** Lệnh ghi tại thời điểm 50 ns cập nhật giá trị `11110000` vào ô nhớ `01111` tại 60 ns.
- **Đọc từ địa chỉ `01111` (70–80 ns):**
  - **Log:**

    ```
    Time=70 addr=01111 rd=1 wr=0 data=11110000 mem[01111]=11110000
    Test 4: Read from addr 01111: data=11110000, mem[01111]=11110000
    Time=80 addr=01111 rd=0 wr=0 data=zzzzzzzz mem[01111]=11110000
    ```

  - **Mô tả:** Tại 70 ns, đọc đúng giá trị `11110000` từ địa chỉ `01111`, phù hợp với kết quả ghi trước đó. Ghi chú tại thời điểm 80 ns với kỳ vọng `10101010` là lỗi trong testbench, không phản ánh đúng trạng thái hệ thống.

5.4 ALU



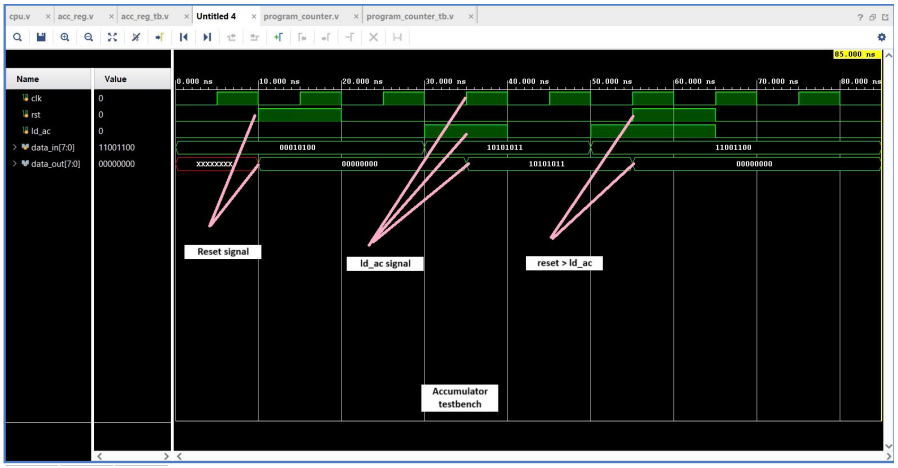
Hình 11: *ALU Testbench*

Waveform từ ALU tb.v testbench mô phỏng mô-đun ALU của thiết kế CPU RISC trong khoảng thời gian 0–170 ns, kiểm tra tất cả 8 mã lệnh (HLT, SKZ, ADD, AND, XOR, LDA, STO, JMP). Mô-đun ALU lấy một mã lệnh 3 bit, đầu vào 8 bit inA và inB, và tạo ra đầu ra 8 bit và 1 bit là số không. Tài liệu dự án chỉ

rõ rằng ALU thực hiện 8 phép toán trên đầu vào 8 bit, với số không cho biết nếu inA = 0 (không đồng bộ). Dạng sóng cho thấy hành vi kết hợp của ALU (không có xung nhịp).

- **0–20 ns: HLT (opcode = 000):** Lệnh HLT gán `out = inA` với giá trị `11111111`. Tín hiệu `is_zero` = 0 vì `inA` $\neq$ 0.
- **20–40 ns: SKZ (opcode = 001):** Lệnh SKZ gán `out = inA` với giá trị `00000000`. Tín hiệu `is_zero` = 1 vì `inA` = 0.
- **40–60 ns: ADD (opcode = 010):** ALU thực hiện phép cộng `inA + inB` với giá trị `00001111 + 11111111 = 100001110`. Bit thứ 9 (bit nhớ) bị loại bỏ nên `out = 00001110`. `is_zero` = 0 vì `inA` $\neq$ 0.
- **60–80 ns: AND (opcode = 011):** ALU thực hiện phép AND: `inA & inB = 00001111`. `out = 00001111`, `is_zero` = 0 vì `inA` $\neq$ 0.
- **80–100 ns: XOR (opcode = 100):** ALU thực hiện phép XOR: `inA ▮ inB = 10101010 ▮ 11001100 = 01100110`. `is_zero` = 0 vì `inA` $\neq$ 0.
- **100–120 ns: LDA (opcode = 101):** Lệnh LDA gán `out = inB = 00001111`. `is_zero` = 0 vì `inA` $\neq$ 0.
- **120–140 ns: STO (opcode = 110):** Lệnh STO gán `out = inA = 01010101`. `is_zero` = 0 vì `inA` $\neq$ 0.
- **140–160 ns: JMP (opcode = 111):** Lệnh JMP gán `out = inA = 11111111`. `is_zero` = 0 vì `inA` $\neq$ 0.

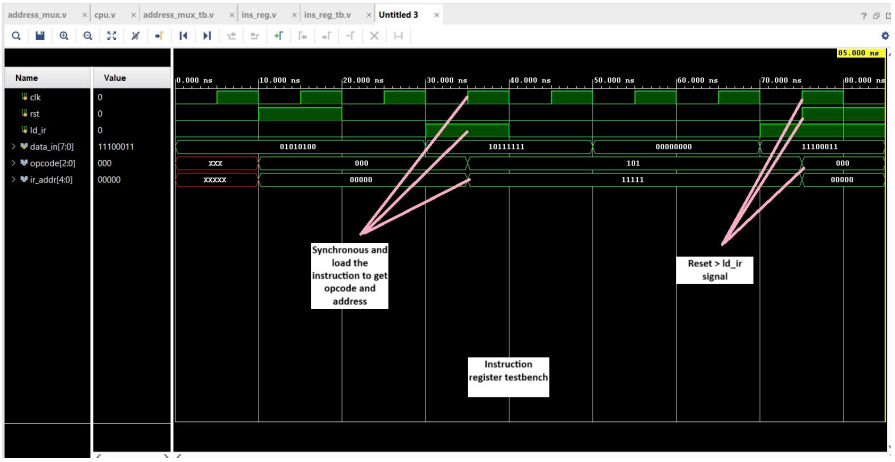
5.5 Accumulator Register



Hình 12: *Accumulator Register Testbench*

- **0–20 ns: Tín hiệu Reset:** Tín hiệu `rst = 1` được kích hoạt tại cạnh lên của `clk` ở thời điểm 0 ns, khiến `data_out` được đặt về `8'b00000000`, bất kể trạng thái của `ld_ac` hay `data_in`.
- **20–60 ns: Tín hiệu ld_ac:** Tại 20 ns, khi `rst = 0` và `ld_ac = 1`, giá trị `data_in = 10101011` được nạp vào `data_out`. Đến 40 ns, `ld_ac` vẫn giữ mức 1 nhưng `data_in` không thay đổi, nên `data_out` vẫn giữ nguyên giá trị `10101011`.
- **60–80 ns: Reset ưu tiên hơn ld_ac:** Khi `rst = 1`, tín hiệu reset có ưu tiên cao hơn, do đó `data_out` được đặt lại về `8'b00000000` ngay cả khi `ld_ac` đang được kích hoạt.
- **80–85 ns: Không thay đổi:** Với `rst = 0` và `ld_ac = 0`, `data_out` giữ nguyên giá trị trước đó là `00000000`.

5.6 Instruction Register

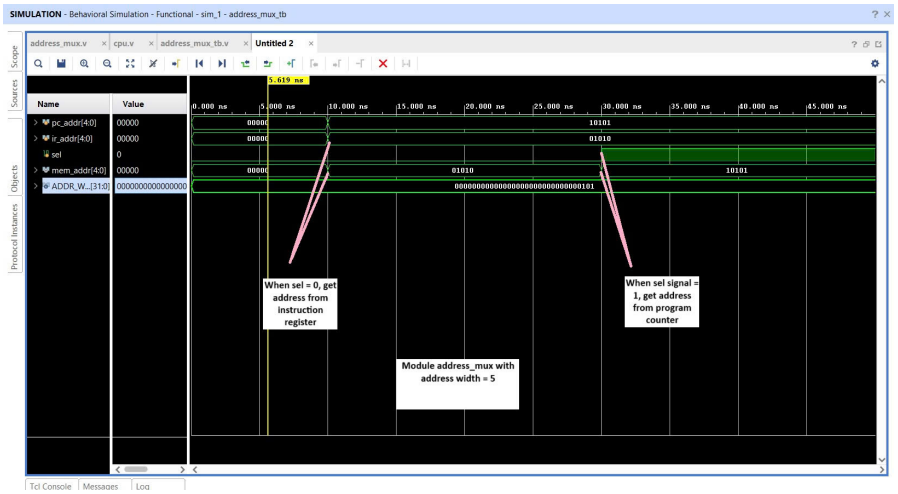


Hình 13: *Instruction Register Testbench*

- **0–20 ns: Tín hiệu Reset:** Khi `rst = 1` được kích hoạt tại cạnh lên của `clk` ở thời điểm 0 ns, thanh ghi lệnh `ir` được đặt lại về `8'b00000000`, tương ứng với `opcode = 000` và `ir_addr = 00000`.
- **20–60 ns: Nạp lệnh (Instruction Load):**
  - Tại 20 ns, `ld_ir = 1` cho phép nạp dữ liệu `data_in = 01010101` vào `ir`. Khi đó, `opcode = 010` (các bit 7–5) và `ir_addr = 01001` (các bit 4–0).
  - Tại 40 ns, tiếp tục với `ld_ir = 1`, dữ liệu `10111111` được nạp vào `ir`, tương ứng với `opcode = 101` và `ir_addr = 11111`.

- **60–80 ns: Ưu tiên tín hiệu Reset hơn ld_ir:** Khi `rst = 1`, tín hiệu reset có ưu tiên cao hơn, do đó `ir` được đặt lại về `8'b00000000` bất chấp trạng thái của `ld_ir`, kết quả là `opcode = 000` và `ir_addr = 00000`.
- **80–85 ns: Không thay đổi:** Với `rst = 0` và `ld_ir = 0`, thanh ghi lệnh `ir` giữ nguyên giá trị hiện tại là `00000000`.

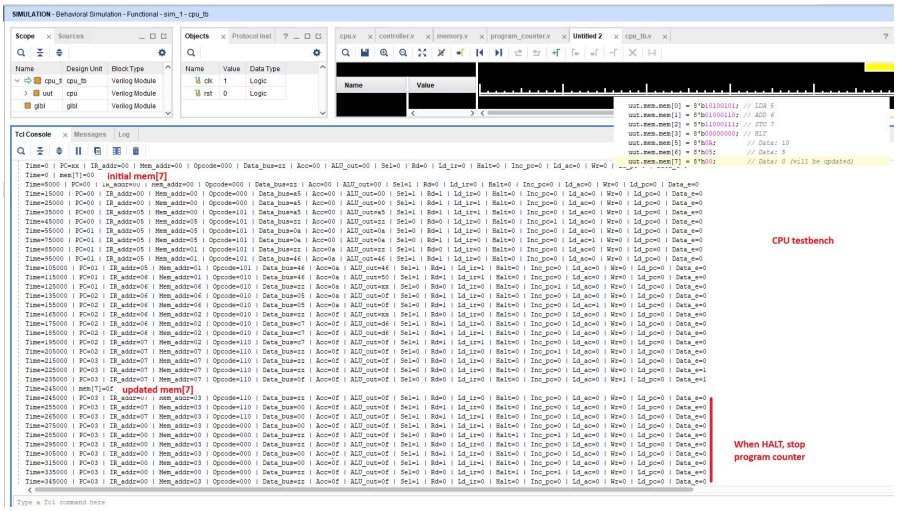
5.7 Address MUX



Hình 14: *Address MUX Testbench*

- **0–10 ns: Trạng thái khởi tạo:** Với `sel = 0`, `mem_addr` nhận giá trị mặc định là `ir_addr = 00000` theo logic mặc định trong bộ chọn địa chỉ.
- **10–20 ns: Trường hợp kiểm thử 1 (sel = 0):** `sel = 0` chọn địa chỉ từ `ir_addr = 01010`, do đó `mem_addr = 01010`.
- **20–30 ns: Trường hợp kiểm thử 2 (sel = 1):** `sel = 1` chọn địa chỉ từ `pc_addr = 10101`, nên `mem_addr = 10101`.
- **30–45 ns: Không thay đổi:** `sel = 1` giữ nguyên, do đó `mem_addr` vẫn là `10101`.

5.8 CPU



Hình 15: *CPU Testbench*

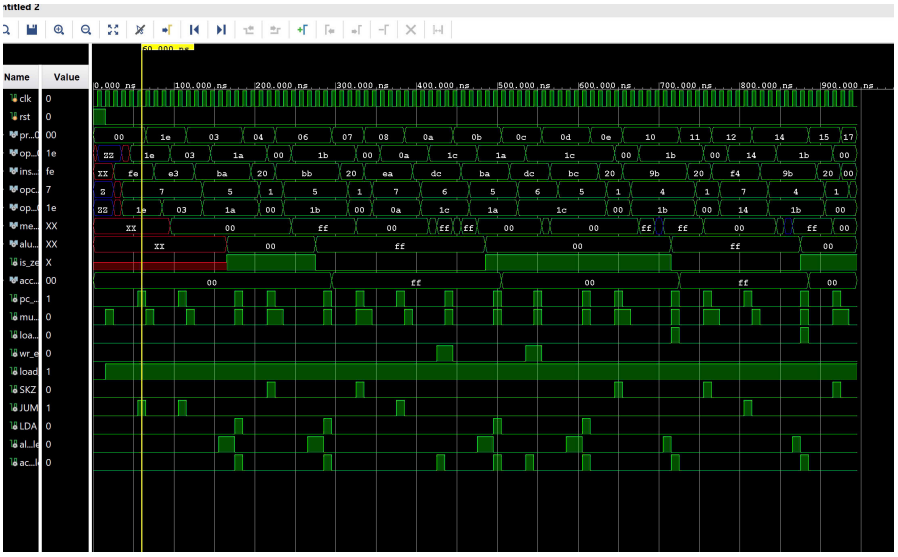
- **0–10 ns: Giai đoạn Reset:** CPU khởi động và thiết lập lại tất cả các thành phần về trạng thái ban đầu.
- **10–90 ns: Lệnh 1 – LDA 5 (Chu trình nạp – thực thi):** CPU thực hiện lệnh `LDA` với toán hạng tại địa chỉ 5. Giá trị được đọc từ bộ nhớ và nạp vào thanh ghi tích lũy (Accumulator).
- **90–170 ns: Lệnh 2 – ADD 6:** CPU thực hiện phép cộng giữa giá trị trong Accumulator và dữ liệu tại địa chỉ 6. Kết quả được ghi đè trở lại Accumulator.
- **170–250 ns: Lệnh 3 – STO 7:** CPU ghi giá trị hiện tại trong Accumulator vào bộ nhớ tại địa chỉ 7.
- **250–330 ns: Lệnh 4 – HLT:** CPU gặp lệnh `HALT` và bắt đầu tiến vào trạng thái dừng hoạt động.
- **330–510 ns: Trạng thái dừng (Halted State):** CPU không thực thi thêm lệnh nào và duy trì trạng thái ổn định sau lệnh HLT.

6 Kiểm thử và kết quả

6.1 Test case

```
/*****************************************************************************
 * Test program
 *
 * Kết quả cần có: Chương trình sau kết thúc (halt) ở lệnh địa chỉ 17(hex)
 *****************************************************************************/

//opcode_operand  // addr                  assembly code
//--------------  // ----  ------------------------------------------------
@00 111_11110     //  00   BEGIN:   JMP TST_JMP
    000_00000     //  01            HLT
    000_00000     //  02            HLT
    101_11010     //  03   JMP_OK:  LDA DATA_1
    001_00000     //  04            SKZ
    000_00000     //  05            HLT
    101_11011     //  06            LDA DATA_2
    001_00000     //  07            SKZ
    111_01010     //  08            JMP SKZ_OK
    000_00000     //  09            HLT
    110_11100     //  0A   SKZ_OK:  STO TEMP
    101_11010     //  0B            LDA DATA_1
    110_11100     //  0C            STO TEMP
    101_11100     //  0D            LDA TEMP
    001_00000     //  0E            SKZ
    000_00000     //  0F            HLT
    100_11011     //  10            XOR DATA_2
    001_00000     //  11            SKZ
    111_10100     //  12            JMP XOR_OK
    000_00000     //  13            HLT
    100_11011     //  14   XOR_OK:  XOR DATA_2
    001_00000     //  15            SKZ
    000_00000     //  16            HLT
    000_00000     //  17   END:     HLT
    111_00000     //  18            JMP BEGIN

@1A 00000000      //  1A   DATA_1:            (giá trị hằng 0x00)
    11111111      //  1B   DATA_2:            (giá trị hằng 0xFF)
    10101010      //  1C   TEMP:              (biến khởi tạo với giá trị 0xAA)

@1E 111_00011     //  1E   TST_JMP: JMP JMP_OK
    000_00000     //  1F            HLT
```

Hình 16: *Test Case*



Hình 17: *Testbench RTL*

- Đầu tiên, ta sẽ gặp lệnh **JUMP** (Opcode là `111`) tại địa chỉ `0x00`. Địa chỉ nhảy đến là `0x1E`.
- Tại địa chỉ `0x1E`, tiếp tục gặp lệnh **JUMP** và nhảy đến địa chỉ `0x03`.
- Tại địa chỉ `0x03`, gặp lệnh **LDA** (Opcode `101`) để load giá trị `0x00` vào Accumulator.
- Đến địa chỉ `0x04`, gặp lệnh **SKZ** (Opcode `001`). Do kết quả ALU lúc này là `0x00`, nên PC sẽ nhảy đến địa chỉ `0x06`.
- Tại địa chỉ `0x06`, gặp lệnh **LDA** (Opcode `101`) để load giá trị `0xFF` vào Accumulator.
- Tại địa chỉ `0x07`, gặp lệnh **SKZ** (Opcode `001`). Vì ALU khác `0`, nên tiếp tục đến địa chỉ `0x08`.
- Tại địa chỉ `0x08`, gặp lệnh **JUMP** (Opcode `111`) nhảy đến địa chỉ `0x0A`.
- Tại địa chỉ `0x0A`, gặp lệnh **STO** (Opcode `110`) để lưu giá trị `0xFF` của Accumulator vào địa chỉ `0x1C`.
- Tại địa chỉ `0x0B`, gặp lệnh **LDA** (Opcode `101`) load giá trị `0x00` vào Accumulator.
- Tại địa chỉ `0x0C`, gặp lệnh **STO** (Opcode `110`) lưu giá trị `0x00` vào địa chỉ `0x1C`.

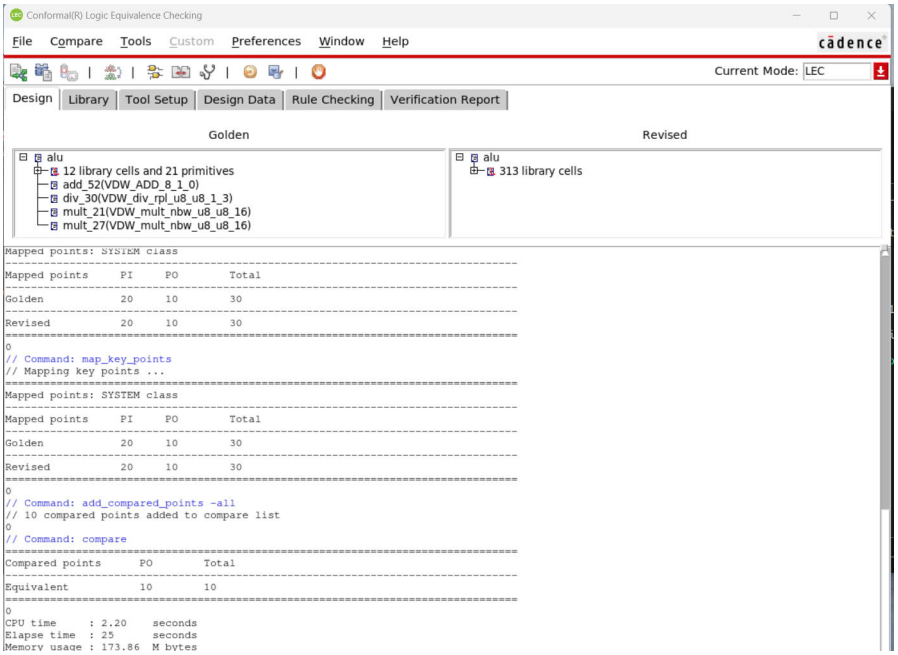
- Tại địa chỉ `0x0D`, gặp lệnh **LDA** (Opcode `101`) load giá trị `0x00` vào Accumulator.
- Đến địa chỉ `0x0E`, gặp lệnh **SKZ** (Opcode `001`). Do kết quả ALU lúc này là `0x00`, nên PC sẽ nhảy đến địa chỉ `0x10`.
- Tại địa chỉ `0x10`, gặp lệnh **XOR** (Opcode `100`) thực hiện XOR giữa `0x00` trong Accumulator và giá trị tại địa chỉ `0x1B` (`0xFF`), kết quả là `0xFF`.
- Tại địa chỉ `0x11`, gặp lệnh **SKZ** (Opcode `001`). Vì ALU khác `0`, nên tiếp tục đến địa chỉ `0x12`.
- Tại địa chỉ `0x12`, gặp lệnh **JUMP** và nhảy đến địa chỉ `0x14`.
- Tại địa chỉ `0x14`, gặp lệnh **XOR** (Opcode `100`) thực hiện XOR giữa `0xFF` trong Accumulator và `0xFF` tại địa chỉ `0x1B`, kết quả là `0x00`.
- Tại địa chỉ `0x15`, gặp lệnh **SKZ** (Opcode `001`). Do kết quả ALU lúc này là `0x00`, nên PC sẽ nhảy đến địa chỉ `0x17`.
- Tại địa chỉ `0x17`, gặp lệnh **HAL** (Opcode `000`) và chương trình kết thúc tại đây.

6.2 Kết quả Synthesis

```
============================================================
  Generated by:           Genus(TM) Synthesis Solution 22.17-s071_1
  Generated on:           Apr 29 2025  08:05:02 pm
  Module:                 alu
  Technology libraries:   slow_1v0
                          pll 0.0
                          CDK_S128x16 0.0
                          CDK_S256x16 0.0
                          CDK_R512x16 0.0
                          physical_cells
                          slow_1v0
                          pll 0.0
                          CDK_S128x16 0.0
                          CDK_S256x16 0.0
                          CDK_R512x16 0.0
                          physical_cells
  Operating conditions:   slow
  Interconnect mode:      global
  Area mode:              physical library
============================================================

Timing
-------
  Cost    Critical         Violating
 Group   Path Slack  TNS     Paths
--------------------------------------
default    No paths   0.0
--------------------------------------
Total                 0.0          0

Instance Count
--------------
Leaf Instance Count             313
Physical Instance count           0
Sequential Instance Count         0
Combinational Instance Count    313
Hierarchical Instance Count       0

Area
----
Cell Area                         770.526
Physical Cell Area                0.000
Total Cell Area (Cell+Physical)   770.526
Net Area                          455.951
Total Area (Cell+Physical+Net)    1226.477

Runtime                           0.0 seconds
Elapsed Runtime                   39 seconds
Genus peak memory usage           1625.02
Innovus peak memory usage         no_value
Hostname                          vlsiktmt
```

Hình 18: *Kết quả Synthesis*

6.3 Kết quả LEC



Hình 19: *Kết quả LEC*

7 Đánh giá và định hướng phát triển

7.1 Đánh giá thiết kế

- Thiết kế CPU theo kiến trúc RISC đơn giản đã được hiện thực thành công ở mức RTL và trải qua quá trình kiểm thử mô phỏng chức năng. Các khối chức năng chính bao gồm Program Counter, ALU, Instruction Register, Memory, Controller,... đều được cài đặt riêng biệt, kiểm tra độc lập và tích hợp thành hệ thống hoàn chỉnh.
- Thông qua waveform mô phỏng, CPU xử lý chính xác các lệnh trong tập lệnh 8-bit với 3-bit opcode và 5-bit toán hạng, bao gồm các lệnh như `LDA`, `STO`, `XOR`, `SKZ`, `JUMP`, và `HALT`. Luồng điều khiển được quản lý chính xác thông qua module Controller FSM nhiều trạng thái, và các tín hiệu điều khiển như `load_ir`, `alu_enable`, `acc_enable`, `wr_en` đều hoạt động đúng thời điểm.
- Kết quả tổng hợp (synthesis) từ Cadence Genus cho thấy thiết kế đạt hiệu quả cao với tổng số 313 cổng tổ hợp, không có phần tử tuần tự, và không vi phạm thời gian (Slack = 0, TNS = 0). LEC (Logic Equivalence Checking) giữa RTL và netlist cũng đạt kết quả tương đương tuyệt đối, đảm bảo tính toàn vẹn thiết kế trong quá trình tổng hợp.

7.2 Khó khăn gặp phải

- Chưa tối ưu về tài nguyên
- Thiếu phần tử tuần tự như thanh ghi đa năng (general-purpose registers), làm giới hạn khả năng mở rộng lệnh.
- Số lượng lệnh còn ít, chưa hỗ trợ nhánh có điều kiện phức tạp.

7.3 Hướng phát triển trong tương lai

- **Tăng số lượng và độ phức tạp của tập lệnh:** Bổ sung các lệnh mới như `SUB`, `MUL`, `DIV`, `BEQ`, `BNE`, hoặc hỗ trợ nhánh có điều kiện.
- **Mở rộng kiến trúc thanh ghi:** Thiết kế thêm các thanh ghi đa năng để nâng cao tính linh hoạt trong thao tác dữ liệu.
- **Tối ưu hoá tài nguyên phần cứng:** Áp dụng clock gating, logic sharing, hoặc áp dụng synthesis ở chế độ low-power để giảm diện tích và năng lượng tiêu thụ.
- **Thêm hỗ trợ bộ nhớ đệm (cache):** Thiết kế cache đơn giản để rút ngắn thời gian truy xuất dữ liệu từ bộ nhớ.

- **Tự động hoá kiểm thử:** Viết testbench tự sinh lệnh để kiểm thử toàn diện cho mọi trường hợp có thể xảy ra.

8 Tổng kết

Trong dự án này, nhóm đã thiết kế và hiện thực thành công một CPU đơn giản theo kiến trúc RISC với tập lệnh 8-bit bao gồm 3-bit opcode và 5-bit toán hạng. Hệ thống được xây dựng theo phương pháp thiết kế phân cấp, chia thành các khối chức năng độc lập như Program Counter, ALU, Instruction Register, Memory, Controller, Accumulator, và Address Mux.

Quá trình mô phỏng cho thấy hệ thống thực thi chính xác các lệnh như `LDA`, `STO`, `XOR`, `SKZ`, `JUMP`, và `HALT`. Tín hiệu điều khiển và các giá trị dữ liệu thay đổi đúng với kỳ vọng. Kết quả tổng hợp bằng Cadence Genus chứng minh rằng thiết kế không vi phạm thời gian (Slack = 0, TNS = 0), và đạt hiệu quả về tài nguyên với 313 cổng logic tổ hợp. Đồng thời, việc kiểm tra tương đương logic (LEC) giữa RTL và netlist cho kết quả hoàn toàn chính xác (Equivalent), đảm bảo độ tin cậy của thiết kế trong suốt quá trình tổng hợp.

Thông qua dự án, nhóm đã nâng cao được hiểu biết thực tế về quy trình thiết kế mạch số ở mức RTL, kỹ năng viết Verilog HDL, phương pháp kiểm thử chức năng và đánh giá hiệu năng hệ thống. Ngoài ra, nhóm cũng học được cách làm việc nhóm, phân chia công việc hiệu quả và xử lý các vấn đề phát sinh trong quá trình phát triển.

Dù vẫn còn một số giới hạn và điểm có thể cải thiện, dự án đã đạt được các mục tiêu đề ra và tạo nền tảng vững chắc cho các dự án nâng cao hơn trong tương lai.

Link github của dự án được cập nhật tại đây: https://github.com/tungngo2525/SIMPLE-RISC-CPU-DESIGN

9 Tài liệu tham khảo

[1] The simplest 8-bit RISC CPU Truy cập từ https://hackaday.io/project/201690-the-simplest-8-bit-risc-cpu

[2] A very simple 8-bit RISC processor for FPGA

[3] Design of a General Purpose 8-bit RISC Processor for Computer Architecture Learning