ĐẠI HỌC BÁCH KHOA HÀ NỘI
**TRƯỜNG ĐIỆN – ĐIỆN TỬ**


# KIẾN TRÚC MÁY TÍNH NÂNG CAO

**Đề tài:**

# THIẾT KẾ MÔ PHỎNG BỘ XỬ LÝ RISC V – 32I BẰNG NGÔN NGỮ LẬP TRÌNH PHẦN CỨNG VERILOG

**Giảng viên hướng dẫn:** TS. Tạ Thị Kim Huệ

**Mã lớp:** 158218

**Sinh viên thực hiện:**

Phạm Anh Tú 20240353E

**Hà Nội, 4-2025**

# MỤC LỤC

DANH MỤC HÌNH VẼ ..... i
DANH MỤC KÝ HIỆU VÀ CHỮ VIẾT TẮT ..... ii
CHƯƠNG 1. GIỚI THIỆU CHUNG ..... 1
1.1. Tổng quan về dự án ..... 1
1.2. Giới thiệu vấn đề ..... 2
1.3. Mục tiêu đề tài ..... 3
CHƯƠNG 2: TRIỂN KHAI HỆ THỐNG ..... 4
2.1 Nền tảng lý thuyết ..... 4
2.1.1 Kiến trúc tập lệnh (ISA) ..... 4
2.1.2 Bộ xử lý RISC-V đơn chu kỳ (Single-Cycle Processor) ..... 5
2.1.3 ModelSim ..... 7
2.2 Thực thi thiết kế hệ thống ..... 8
2.2.1 Program Counter (PC) ..... 8
2.2.2 Instruction Memory (Inst_Mem) ..... 9
2.2.3 Arithmetic Logic Unit (ALU) ..... 10
2.2.4 Register File (Reg_File) ..... 11
2.2.5 Data_Mem (Data Memory) ..... 13
2.2.6 ALU_Control (ALU Control) ..... 14
2.2.7 Control_Unit (Control Unit) ..... 15
CHƯƠNG 3: KẾT QUẢ MÔ PHỎNG ..... 17
3.1 Môi trường mô phỏng ..... 17
3.2 Quy trình mô phỏng ..... 17
3.3 Các tín hiệu được kiểm tra ..... 17
3.4 Kết quả mô phỏng ..... 18
TÀI LIỆU THAM KHẢO ..... 19

# DANH MỤC HÌNH VẼ

Hình 2. 1 Opcode và toán hạng của lệnh MOV ..... 4
Hình 2. 2 Bộ xử lý đơn chu kỳ ..... 6
Hình 2. 3 Quy trình của dự án ..... 8
Hình 2. 4 Code Verilog của khối Program Counter ..... 9
Hình 2. 5 Code Verilog của khối Instruction Memory ..... 10
Hình 2. 6 Code Verilog của khối ALU ..... 11
Hình 2. 7 Code Verilog khối RegisterFile ..... 13
Hình 2. 8 Code Verilog của khối Data Memory ..... 14
Hình 2. 9 Code Verilog của ALU_Control ..... 15
Hình 2. 10 Code Verilog của Control_Unit ..... 16
Hình 3. 1 Kết quả mô phỏng trên ModelSim 18

# DANH MỤC KÝ HIỆU VÀ CHỮ VIẾT TẮT

| STT | Kí hiệu | Thuật ngữ đầy đủ |
|---|---|---|
| 1 | ISA | Instruction Set Architecture |
| 2 | HDL | Hardware Description Language |
| 3 | CISC | Complex Instruction Set Computing |
| 4 | RISC | Reduced Instruction Set Computing |
| 5 | ARM | Advanced RISC Machines |
| 6 | MIPS | Microprocessor without Interlocked Pipelined Stages |
| 7 | FPGA | Field Programmable Gate Arrays |
| 8 | VWF | Vector Waveform File |
| 9 | PC | Program Counter |

# CHƯƠNG 1. GIỚI THIỆU CHUNG

## 1.1. Tổng quan về dự án

Thiết kế VLSI (Very Large-Scale Integration) bắt đầu từ những năm 1970 khi công nghệ bán dẫn và truyền thông đang trong quá trình phát triển. Đây là yếu tố then chốt thúc đẩy cuộc cách mạng vi xử lý và mở ra nhiều đột phá trong các hệ thống điện tử và máy tính. Từ đó, nhiều khoản đầu tư nghiên cứu đã được rót vào nhằm khai thác tiềm năng ứng dụng của máy tính trong nhiều lĩnh vực khác nhau như hàng không, y tế, điện thoại di động và công nghiệp ô tô.

Theo dự báo, đến năm 2030 sẽ có hơn 22,7 tỷ thiết bị được kết nối thông qua các mạng IoT (W. Wang và cộng sự, 2021). Các thiết bị IoT này đóng vai trò là nền tảng cho việc hiện thực hóa các khái niệm trong tương lai như thành phố thông minh, xe tự hành, và các công nghệ khám phá không gian.

VLSI là quá trình tạo ra một vi mạch tích hợp (IC) bằng cách kết hợp hàng ngàn bóng bán dẫn trên một con chip duy nhất. Trước khi công nghệ VLSI ra đời, các IC chỉ thực hiện được một số chức năng hạn chế. VLSI cho phép các nhà thiết kế IC tích hợp toàn bộ các thành phần như bộ xử lý trung tâm (CPU), bộ nhớ chỉ đọc (ROM), bộ nhớ truy cập ngẫu nhiên (RAM) và các khối logic khác vào cùng một con chip. Các công nghệ hiện đại như truyền video độ phân giải cao với tốc độ bit thấp và truyền thông di động đang mang lại cho người dùng hàng loạt ứng dụng mạnh mẽ với khả năng xử lý cao và tính di động ưu việt. Xu hướng này được dự đoán sẽ phát triển nhanh chóng và có ảnh hưởng sâu rộng đến thiết kế VLSI và thiết kế hệ thống.

Bộ vi xử lý được xem là "bộ não" của các hệ thống máy tính điện tử hiện đại. Nó giúp kết nối và giao tiếp với các thiết bị ngoại vi như chuột, bàn phím, loa và xử lý thông tin nhận được từ các thiết bị này. Hầu hết các bộ xử lý ngày nay được xây dựng dựa trên hai kiến trúc chính là CISC (Complex Instruction Set Computing) và RISC (Reduced Instruction Set Computing). Hai kiến trúc này đưa ra những phong cách thiết kế và mạch điện khác nhau, khác biệt chủ yếu ở cách dữ liệu được xử lý và lưu trữ. Mỗi kiến trúc đều có ưu điểm và nhược điểm riêng, do đó việc triển khai, tối ưu và kế thừa những lợi thế sẵn có vẫn có ý nghĩa lớn đối với cả giới học thuật lẫn ngành công nghiệp.

RISC-V là một kiến trúc tập lệnh (ISA) mở và miễn phí được phát triển dựa trên nền tảng kiến trúc RISC truyền thống. Nó được giới thiệu vào năm 2010 tại Đại học California, Berkeley. RISC-V nhanh chóng thu hút sự chú ý từ ngành công nghiệp vì cho phép sử dụng mã nguồn mở mà không yêu cầu trả phí bản quyền – điều này đã phá vỡ nhiều rào cản trong ngành bán dẫn. RISC-V được thiết kế theo hướng mô-đun, với chỉ 47 lệnh cơ bản, và có thể được mở rộng linh hoạt tùy theo

yêu cầu thiết kế. Kiến trúc tập lệnh RISC-V không quy định cách thiết kế cụ thể hay các tập con bắt buộc phải có, vì vậy các máy tính sử dụng RISC-V có thể linh hoạt lựa chọn các phần mở rộng gọn nhẹ nhằm giảm tiêu thụ năng lượng, kích thước mã lệnh và bộ nhớ sử dụng.

### 1.2. Giới thiệu vấn đề

Kiến trúc tập lệnh (ISA – Instruction Set Architecture) là một phần của mô hình trừu tượng trong máy tính, định nghĩa cách bộ xử lý trung tâm (CPU) được điều khiển bởi phần mềm. Nó đóng vai trò như một giao diện giữa phần cứng và phần mềm. ISA xác định các kiểu dữ liệu được hỗ trợ, các thanh ghi, bộ nhớ, tập lệnh thực thi và các đặc điểm của bộ xử lý.

Các công ty như Intel, IBM và ARM đều sử dụng kiến trúc tập lệnh riêng của họ trong các sản phẩm thương mại. Tuy nhiên, những ISA này thường được thiết kế theo cách độc quyền, nhằm ngăn cản bên thứ ba sử dụng mà không có giấy phép. Việc đàm phán để có quyền sử dụng có thể kéo dài hàng tháng và chi phí rất cao, khiến nó trở thành rào cản đối với cộng đồng mã nguồn mở và các tổ chức nhỏ.

Để vượt qua vấn đề này, cần có một kiến trúc tập lệnh mở nhằm thúc đẩy đổi mới sáng tạo trên quy mô rộng. Việc chia sẻ thiết kế lõi mở sẽ tạo điều kiện cho các tổ chức nhỏ tham gia thị trường cạnh tranh, từ đó tạo ra một môi trường cạnh tranh lành mạnh. Người tiêu dùng cũng sẽ hưởng lợi từ các sản phẩm có hiệu năng phù hợp và giá thành hợp lý.

Như đã nêu trên trang web của tổ chức RISC-V, “Sự quan tâm trên toàn cầu đối với RISC-V không phải vì đây là một công nghệ chip mới tuyệt vời, mà bởi vì nó là một chuẩn mở, miễn phí và phổ biến, cho phép các phần mềm có thể được chuyển đổi và chạy trên bất kỳ phần cứng nào mà bất cứ ai đều có thể phát triển”. Chính những đặc điểm này khiến ISA RISC-V trở thành sự lựa chọn lý tưởng cho nhu cầu của chúng ta.

Hiệu suất của bộ xử lý bị ảnh hưởng rất lớn bởi cách thực hiện lệnh. Trong thiết kế vi xử lý đơn chu kỳ (single-cycle), lệnh kế tiếp chỉ được thực hiện sau khi lệnh hiện tại đã hoàn tất. Khi độ phức tạp của tập lệnh tăng lên, hiệu suất của bộ xử lý đơn chu kỳ sẽ giảm đáng kể. Hơn nữa, thời gian một chu kỳ xung nhịp cần được thiết kế đủ dài để đảm bảo cả lệnh chậm nhất có thể hoàn tất, dẫn đến độ trễ tổng thể của bộ xử lý bị tăng.

Để khắc phục nhược điểm này, thiết kế pipeline (đường ống lệnh) được sử dụng, cho phép thực hiện nhiều lệnh đồng thời theo cách xếp chồng (overlap) giữa các giai đoạn thực thi.

### 1.3. Mục tiêu đề tài

1. Hiểu các kiến thức cơ bản về kiến trúc RISC-V
2. Thiết kế bộ xử lý RISC V đơn xung nhịp thực thi được một chương trình đơn giản mô tả bằng ngôn ngữa ASSEMBLY RISC V
3. Xác minh tính đúng đắn của thiết kế thông qua testbench và mô phỏng

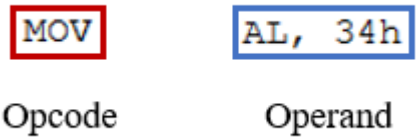
# CHƯƠNG 2: TRIỂN KHAI HỆ THỐNG

## 2.1 Nền tảng lý thuyết

### *2.1.1 Kiến trúc tập lệnh (ISA)*

Kiến trúc tập lệnh (ISA) là thuật ngữ dùng để chỉ tập hợp các lệnh mà một bộ xử lý máy tính có thể thực thi. Đây là một khía cạnh nền tảng trong kiến trúc máy tính, quyết định khả năng và giới hạn của một bộ xử lý.

ISA bao gồm tập hợp các lệnh xác định các thao tác mà bộ xử lý có thể thực hiện, cũng như định dạng và ý nghĩa của từng lệnh. Mỗi lệnh thường bao gồm một opcode (mã thao tác) – xác định hành động cần thực hiện, và một hoặc nhiều toán hạng (operands) – xác định dữ liệu mà thao tác sẽ được áp dụng lên. Hình 2.1 minh họa ví dụ về opcode và toán hạng của lệnh MOV.




*Hình 2. 1 Opcode và toán hạng của lệnh MOV*

Từ hình 2.1, ta thấy opcode là lệnh MOV, trong khi các phần còn lại được gọi là toán hạng. Toán hạng là các đối tượng dữ liệu được thao tác bởi opcode. Trong ví dụ này, toán hạng là thanh ghi AL và giá trị 34 (hệ thập lục phân).

ISA có thể được phân loại thành hai nhóm chính:

- RISC (Reduced Instruction Set Computing)
- CISC (Complex Instruction Set Computing)

ISA thuộc loại RISC gồm một tập hợp nhỏ các lệnh đơn giản có thể được thực thi nhanh chóng, trong khi đó ISA loại CISC chứa tập hợp lớn hơn các lệnh phức tạp có thể thực hiện nhiều thao tác chỉ trong một lệnh duy nhất.

Hiện nay, hướng tiếp cận RISC thường được ưu tiên trong thiết kế bộ xử lý hiện đại vì nó cho phép thời gian thực thi nhanh hơn và thiết kế bộ xử lý đơn giản hơn. Tuy nhiên, kiến trúc CISC vẫn được sử dụng trong một số ứng dụng chuyên biệt, nơi mà độ phức tạp bổ sung được biện minh bởi tính năng mở rộng.

Một ví dụ điển hình của kiến trúc CISC ISA là x86, được sử dụng rộng rãi trong máy tính cá nhân và máy chủ. Kiến trúc x86 bao gồm một tập lệnh lớn với độ phức tạp cao, trong đó mỗi lệnh có thể thực hiện nhiều thao tác cùng lúc. Ví dụ, lệnh MOV trong kiến trúc CISC có thể truyền dữ liệu giữa hai vị trí bộ nhớ, hoặc giữa thanh ghi và bộ nhớ chỉ với một lệnh duy nhất. Ngoài ra, x86 còn bao gồm nhiều lệnh chuyên biệt như xử lý chuỗi ký tự, nhập/xuất dữ liệu, và tính toán

dấu chấm động. ISA x86 nổi bật với độ phức tạp và khả năng tương thích ngược cao. Tuy nhiên, chính sự phức tạp này khiến việc tối ưu hiệu năng hoặc tiết kiệm năng lượng trở nên khó khăn hơn, và việc viết phần mềm chạy hiệu quả trên nhiều phiên bản x86 khác nhau cũng là một thách thức.

Ngược lại, một ví dụ điển hình của ISA RISC là kiến trúc ARM, được sử dụng phổ biến trong các thiết bị như điện thoại thông minh, máy tính bảng, và hệ thống nhúng. ISA ARM chỉ bao gồm một tập hợp nhỏ các lệnh đơn giản giúp thực thi nhanh hơn. Ví dụ, lệnh MOV trong kiến trúc ARM chỉ đơn thuần là sao chép dữ liệu từ một thanh ghi sang một thanh ghi khác. Bên cạnh đó, ARM còn cung cấp các lệnh chuyên biệt như:

- Nạp và lưu nhiều thanh ghi cùng lúc,
- Thực hiện lệnh có điều kiện,
- Thực hiện các phép toán trên nhiều kiểu dữ liệu khác nhau.

ISA ARM được biết đến với độ đơn giản và hiệu suất năng lượng cao. Chính sự đơn giản trong kiến trúc giúp dễ dàng tối ưu hóa hiệu năng và dễ phát triển phần mềm có thể hoạt động hiệu quả trên các bộ xử lý ARM khác nhau.

### *2.1.2 Bộ xử lý RISC-V đơn chu kỳ (Single-Cycle Processor)*

Trong phần này, chúng ta tiến hành thiết kế một bộ xử lý RISC-V đơn chu kỳ từ đầu, khám phá từng thành phần thiết yếu của nó. Bộ xử lý đơn chu kỳ thực thi mỗi lệnh trong một chu kỳ xung nhịp, giúp cấu trúc đơn giản nhưng hiệu suất không cao bằng các thiết kế đa chu kỳ hoặc pipeline phức tạp hơn. RISC-V là một kiến trúc tập lệnh mã nguồn mở (ISA) nổi bật nhờ sự đơn giản, tính mô-đun và khả năng mở rộng, rất phù hợp cho giảng dạy, nghiên cứu, cũng như các ứng dụng thương mại.

Các thành phần chính của bộ xử lý RISC-V đơn chu kỳ

1. Bộ đếm chương trình (Program Counter - PC)

PC giữ địa chỉ của lệnh hiện tại đang được thực thi. Sau mỗi chu kỳ, nó tự động tăng để trỏ đến lệnh tiếp theo, trừ khi bị thay đổi bởi các lệnh nhảy hoặc rẽ nhánh (branch/jump).

2. Bộ nhớ lệnh (Instruction Memory)

Mô-đun này chứa các lệnh máy của chương trình. PC sẽ dùng để truy xuất lệnh hiện tại từ bộ nhớ này. Sau đó, lệnh được giải mã để điều khiển hoạt động của bộ xử lý.

3. Tập thanh ghi (Register File)

Tập thanh ghi bao gồm 32 thanh ghi đa dụng (x0 đến x31). Cho phép đọc đồng thời hai thanh ghi và ghi một thanh ghi trong cùng một chu kỳ. Theo chuẩn RISC-V, thanh ghi x0 luôn mang giá trị 0 và không thể bị ghi đè.

4. Bộ ALU (Arithmetic Logic Unit)

ALU thực hiện các phép toán số học và logic, ví dụ: cộng, trừ, AND, OR, so sánh... Loại thao tác được điều khiển bằng các tín hiệu điều khiển dựa trên loại lệnh (ví dụ: R-type, I-type...).

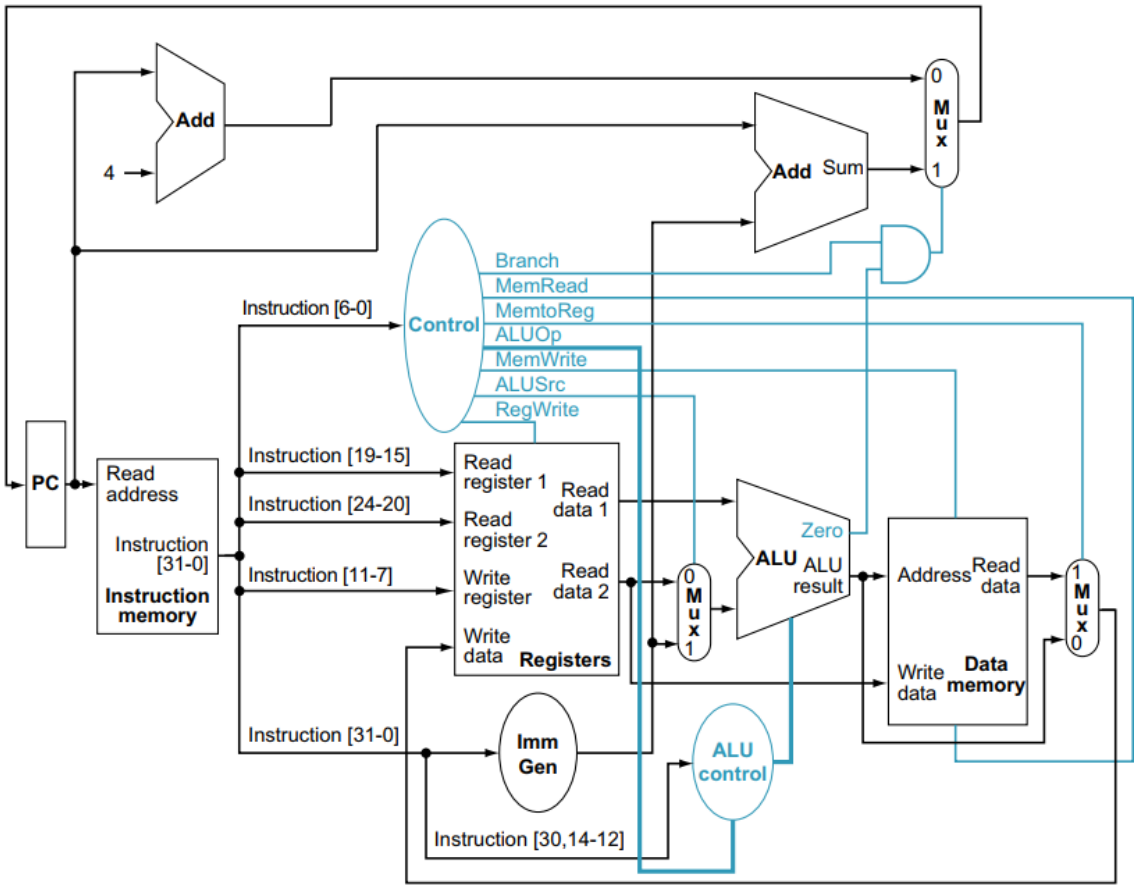
5. Bộ điều khiển (Control Unit)

Bộ điều khiển tạo ra các tín hiệu điều khiển cần thiết để điều phối dòng dữ liệu bên trong bộ xử lý. Nó giải mã opcode của lệnh vừa được truy xuất và thiết lập các tín hiệu điều khiển để dẫn hướng hoạt động của ALU, truy cập bộ nhớ, và điều khiển các bộ chọn (multiplexers).

6. Bộ nhớ dữ liệu (Data Memory)

Bộ nhớ dữ liệu được sử dụng cho các lệnh tải (load) và lưu (store). Nó đọc dữ liệu từ bộ nhớ khi thực hiện lệnh load, và ghi dữ liệu vào bộ nhớ khi thực hiện lệnh store, sử dụng địa chỉ được tính bởi ALU.

Hình 2.2 minh họa tổng quan về bộ xử lý RISC-V đơn chu kỳ




*Hình 2. 2 Bộ xử lý đơn chu kỳ*

### *2.1.3 ModelSim*

ModelSim là một công cụ mô phỏng và kiểm tra phổ biến dành cho các mạch số và hệ thống số. Nó được sử dụng rộng rãi bởi các kỹ sư và nhà thiết kế trong ngành công nghiệp điện tử để xác thực và gỡ lỗi (debug) các thiết kế của họ trước khi hiện thực hóa trên phần cứng.

ModelSim cung cấp một tập hợp tính năng mạnh mẽ để thiết kế, mô phỏng và kiểm chứng các mạch số và hệ thống. Về mặt mô phỏng, ModelSim hỗ trợ cả hai ngôn ngữ mô tả phần cứng phổ biến là Verilog và VHDL. Nó có thể mô phỏng ở mọi mức độ trừu tượng – từ mức cổng logic (gate level) đến mức hành vi (behavioral level).

Về mặt kiểm chứng, ModelSim hỗ trợ mô phỏng chức năng (functional simulation) và mô phỏng thời gian (timing simulation), cũng như kiểm chứng dựa trên mệnh đề (assertion-based verification). Ngoài ra, ModelSim còn có thể tích hợp với các công cụ kiểm chứng khác như Questa hoặc UVM nhằm thực hiện các chiến lược kiểm chứng toàn diện hơn.

Trong quá trình thiết kế, ModelSim hỗ trợ tổ chức thiết kế theo cấu trúc phân cấp (design hierarchy), giúp các nhà thiết kế sắp xếp và quản lý thiết kế theo từng khối logic riêng biệt. Ngoài ra, phần mềm cũng cung cấp các tính năng kiểm tra thiết kế như linting và kiểm tra cú pháp (syntax checking) để phát hiện lỗi sớm.

Việc sử dụng ModelSim mang lại nhiều lợi ích:

- Nâng cao chất lượng thiết kế: Phát hiện và sửa lỗi sớm trong vòng đời thiết kế, từ đó cải thiện chất lượng sản phẩm cuối cùng.
- Giảm nhu cầu tạo mẫu phần cứng đắt đỏ: Cung cấp môi trường ảo để xác minh và kiểm thử thiết kế, giúp tiết kiệm chi phí.
- Hỗ trợ các ngôn ngữ và chuẩn công nghiệp: Tăng tính tương thích và khả năng chuẩn hóa giữa các thiết kế trong ngành công nghiệp điện tử.

Ứng dụng ModelSim trong dự án

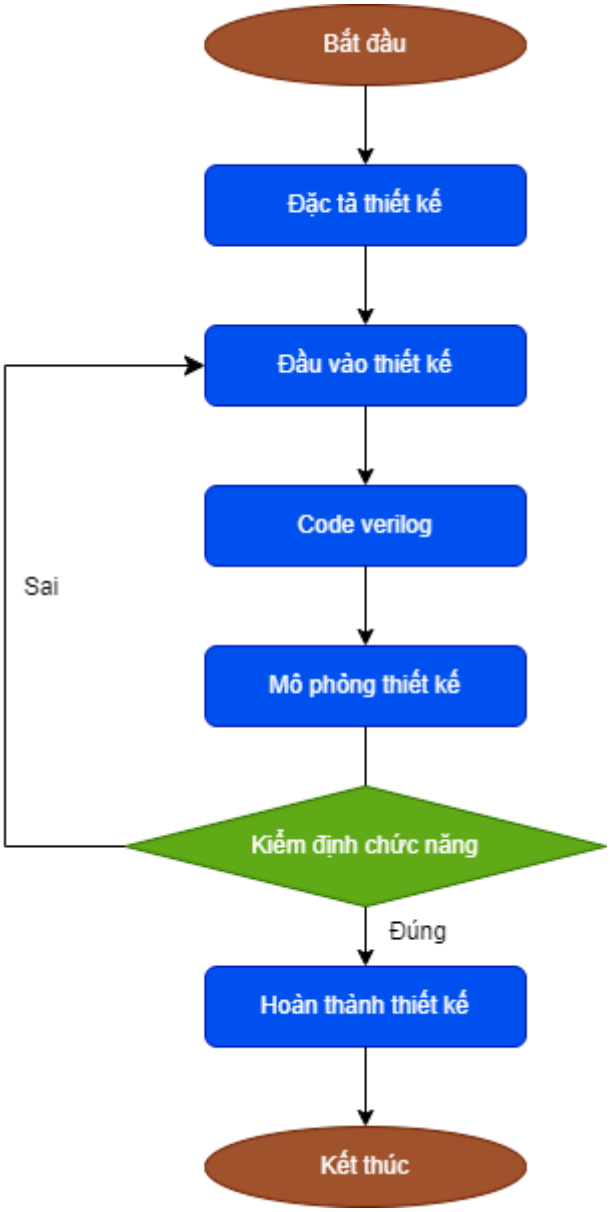
Trong đồ án này, ModelSim sẽ là phần mềm chính được sử dụng để thiết kế và mô phỏng bộ xử lý RISC-V. Các mô-đun riêng biệt như:

- Bộ nhớ lệnh (Instruction Memory)
- Bộ cộng (Adder)
- Thanh ghi (Register File)
- Bộ nhớ dữ liệu (Data Memory)
- Bộ ALU và điều khiển ALU (ALU + ALU Control)

…sẽ được lập trình bằng ngôn ngữ Verilog trong ModelSim.

Chức năng của từng phần tử sẽ được kiểm thử và xác minh thông qua mô phỏng. Cuối cùng, tất cả các mô-đun sẽ được tích hợp trong một mô-đun chính để hình thành thiết kế tổng thể của một bộ xử lý RISC-V 32-bit đơn chu kỳ.

Hình 2.3 minh họa biểu đồ quy trình của dự án.




*Hình 2. 3 Quy trình của dự án*

## 2.2 Thực thi thiết kế hệ thống

Bộ xử lý bao gồm ALU, ALU_Control, ALU_Mux, BranchAdder, ControlUnit, DataMem, Mux_DataMem, Imm_Gen, Inst_Mem, PC, PC_Adder, PC_Mux, Register_File,. Mỗi thành phần của Datapath sẽ được thảo luận trong phần này.

### *2.2.1 Program Counter (PC)*

Program Counter (PC) là một thanh ghi (register) trong CPU dùng để lưu trữ địa chỉ ô nhớ của lệnh tiếp theo sẽ được thực thi. PC đóng vai trò then chốt trong việc

duy trì trình tự thực hiện tuần tự của chương trình, khi nó tự động tăng sau mỗi chu kỳ xung nhịp để trỏ đến lệnh kế tiếp trong bộ nhớ.

Bên cạnh việc thực hiện tuần tự các lệnh, PC còn đảm nhiệm vai trò điều hướng dòng thực thi chương trình trong các trường hợp xảy ra rẽ nhánh (branch) hoặc nhảy (jump). Khi đó, giá trị của PC được cập nhật không theo thứ tự tăng thông thường mà được gán bằng địa chỉ đích của lệnh rẽ nhánh hoặc nhảy.

Ngoài ra, PC còn tham gia vào cơ chế xử lý ngoại lệ (exception) hoặc ngắt (interrupt) bằng cách chuyển hướng luồng điều khiển đến các routines (chương trình con) xử lý ngoại lệ được chỉ định sẵn trong hệ thống.

```
module PC( clk, rst, pc_in, pc_out );
    input clk;
    input rst;
    input [31:0] pc_in;
    output reg [31:0] pc_out;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            pc_out <= 32'b00;
        end else begin
            pc_out <= pc_in;
        end
    end
endmodule
```

*Hình 2. 4 Code Verilog của khối Program Counter*

### *2.2.2 Instruction Memory (Inst_Mem)*

Instruction Memory là một khối bộ nhớ chỉ đọc (read-only) trong bộ xử lý RISC-V đơn chu kỳ, dùng để lưu trữ các lệnh của chương trình. Khối bộ nhớ này được địa chỉ hóa bởi thanh ghi Program Counter (PC), cho phép truy xuất tuần tự các lệnh.

Trong mỗi chu kỳ xung nhịp, Instruction Memory thực hiện việc truy xuất lệnh tại địa chỉ được chỉ bởi PC. Sau đó, PC sẽ được tăng lên để trỏ tới lệnh kế tiếp trong trình tự thực thi.

Instruction Memory giữ vai trò then chốt trong giai đoạn lấy lệnh (instruction fetch stage) của bộ xử lý. Nó đảm bảo rằng các lệnh được truy xuất và chuẩn bị đầy đủ cho các giai đoạn xử lý tiếp theo như giải mã, thực thi, truy xuất bộ nhớ và ghi kết quả.

Nhờ vào cơ chế này, bộ xử lý có thể thực thi tuần tự từng lệnh một trong mỗi chu kỳ xung nhịp, đảm bảo sự đúng đắn và trật tự trong quá trình thực thi chương trình.

```
module Inst_Mem(rst, clk, read_address, instruction_out);

input rst, clk;
input [31:0] read_address;
output [31:0] instruction_out;
reg [31:0] I_Mem [127:0];
integer k;
assign instruction_out = I_Mem[read_address];

always @(posedge clk or posedge rst)
begin
    if (rst) begin
        for (k = 0; k < 64; k = k + 1) begin
            I_Mem[k] = 32'b00;
        end
    end else begin
// R-type
I_Mem[0]  = 32'b00000000000000000000000000000000;  // no operation
I_Mem[4]  = 32'b0000000_11001_11000_01101_000_0110011;  // add x13, x24, x25
I_Mem[8]  = 32'b0100000_00111_01000_00101_000_0110011;  // sub x5, x8, x7
I_Mem[12] = 32'b0000000_00010_00011_00001_111_0110011;  // and x1, x3, x2
I_Mem[16] = 32'b0000000_00101_00011_00100_110_0110011;  // or x4, x3, x5
```

```
// I-type
I_Mem[20] = 32'b000000000011_10011_10101_000_0010011;  // addi x21, x19, 3
I_Mem[24] = 32'b000000000001_01010_01011_110_0010011;  // ori x11, x10, 1

// L-type
I_Mem[28] = 32'b000000011111_00111_010_01010_0000011;  // lw x10, 15(x7)
I_Mem[32] = 32'b000000000011_00011_010_01101_0000011;  // lw x13, 3(x3)

// S-type
I_Mem[36] = 32'b0000000_01111_00101_010_01100_0100011;  // sw x15, 12(x5)
I_Mem[40] = 32'b0000000_01110_00010_010_01110_0100011;  // sw x14, 14(x2)

// SB-type
I_Mem[44] = 32'h0948663;  // beq x9, x9, 12



    end
end

endmodule
```

*Hình 2. 5 Code Verilog của khối Instruction Memory*

### *2.2.3 Arithmetic Logic Unit (ALU)*

Trong bộ xử lý RISC-V đơn chu kỳ, ALU (Arithmetic Logic Unit) là một thành phần cốt lõi, chịu trách nhiệm thực hiện các phép toán số học và logic trên dữ liệu. ALU nhận vào hai toán hạng đầu vào, thực hiện phép toán được chỉ định (ví dụ: cộng, trừ, AND, OR, v.v.), và trả về kết quả đầu ra.

Trong kiến trúc đơn chu kỳ, mỗi lệnh được thực thi trong một chu kỳ xung nhịp duy nhất. Do đó, ALU được thiết kế để hoàn tất các phép toán trong một chu kỳ xung nhịp, nhằm đảm bảo hiệu suất đồng bộ với các thành phần khác trong đường dữ liệu (datapath).

ALU đóng vai trò thiết yếu trong giai đoạn thực thi (Execute) của chu trình lệnh, nơi các phép toán được thực hiện dựa trên dữ liệu từ các thanh ghi và điều khiển bởi các tín hiệu điều khiển do bộ điều khiển (Control Unit) phát sinh. Chính vì vậy,

ALU là một thành phần không thể thiếu trong việc đảm bảo quá trình xử lý lệnh diễn ra chính xác và hiệu quả.

```
module ALU( A, B, ALUcontrol_In, Result, Zero );
    input [31:0] A;
    input [31:0] B;
    input [3:0] ALUcontrol_In;
    output reg [31:0] Result;
    output reg Zero;

    always @(A or B or ALUcontrol_In) begin
        case (ALUcontrol_In)
            4'b0000: Result = A + B;                // ADD
            4'b0001: Result = A - B;                // SUB
            4'b0010: Result = A & B;                // AND
            4'b0011: Result = A | B;                // OR
            4'b0100: Result = A ^ B;                // XOR
            4'b0101: Result = A << B[4:0];          // SLL (Shift Left Logical)
            4'b0110: Result = A >> B[4:0];          // SRL (Shift Right Logical)
            4'b0111: Result = $signed(A) >>> B[4:0];    // SRA (Shift Right Arithmetic)
            4'b1000: Result =($signed(A) < $signed(B)) ? 32'b1 : 32'b0;
            default: Result = 32'b0;
        endcase


        Zero = (Result == 32'b0) ? 1 : 0;
    end

endmodule
```

*Hình 2. 6 Code Verilog của khối ALU*

*Bảng 1. Các lệnh ALU cơ bản*

| | | Mô tả |
|---|---|---|
| **Số học** | | |
| ADD | rd, rs1, rs2 | rd ← rs1 + rs2 |
| SUB | rd, rs1, rs2 | rd ← rs1 - rs2 |
| **Logic** | | |
| XOR | rd, rs1, rs2 | rd ← rs1 ^ rs2 |
| AND | rd, rs1, rs2 | rd ← rs1 & rs2 |
| OR | rd, rs1, rs2 | rd ← rs1 \| rs2 |
| **Dịch** | | |
| SHL | rd, rs1, rs2 | rd ← rs1 << rs2 |
| SHR | rd, rs1, rs2 | rd ← rs1 >> rs2 |
| **So sánh** | | |
| SLT | rd, rs1, rs2 | rd ← rs1 < rs2 ? 1 : 0 |
| SLTU | rd, rs1, rs2 | rd ← rs1 < rs2 ? 1 : 0 |

### *2.2.4 Register File (Reg_File)*

Trong kiến trúc RISC-V, có 32 thanh ghi số nguyên đa dụng (từ x0 đến x31) được sử dụng cho các phép tính toán. Ngoài ra, các thanh ghi số thực (từ f0 đến f31) được sử dụng trong các phép toán dấu phẩy động. Đối với các thao tác song song theo chiều dữ liệu (SIMD), thanh ghi vectơ (từ v0 đến v31) được sử dụng để hỗ trợ tính toán hiệu quả trên tập dữ liệu lớn.

Bên cạnh các thanh ghi dữ liệu, RISC-V còn bao gồm các thanh ghi điều khiển và trạng thái (CSRs - Control and Status Registers), đóng vai trò trong việc quản lý hoạt động và trạng thái của bộ xử lý, như quyền truy cập, chế độ vận hành, hoặc thông tin ngắt. Bộ đếm chương trình (Program Counter – pc) theo dõi địa chỉ của lệnh tiếp theo sẽ được thực hiện, đảm bảo luồng thực thi lệnh liên tục và chính xác.

Một số biến thể của RISC-V còn có thanh ghi chuyên biệt như Link Register (lr), phục vụ cho các mục đích cụ thể như lưu địa chỉ trở về sau khi thực hiện lệnh gọi hàm.

Các thanh ghi là thành phần thiết yếu của bộ xử lý, cho phép lưu trữ dữ liệu tạm thời, thao tác nhanh chóng, và quản lý luồng điều khiển trong quá trình thực thi chương trình, từ đó góp phần quan trọng vào hiệu năng tổng thể và tính năng linh hoạt của hệ thống tính toán.

```
module Reg_File(clk, rst, RegWrite,Rs1,Rs2, Rd,Write_data,read_data1, read_data2);

input clk, rst, RegWrite;
input [4:0] Rs1,Rs2, Rd;
input [31:0] Write_data;
output [31:0] read_data1, read_data2;

reg [31:0] Registers [31:0];

 initial begin
Registers[0]  = 0;
Registers[1]  = 81;
Registers[2]  = 57;
Registers[3]  = 12;
Registers[4]  = 73;
Registers[5]  = 39;
Registers[6]  = 5;
Registers[7]  = 66;
Registers[8]  = 28;
Registers[9]  = 94;
Registers[10] = 18;
Registers[11] = 3;
Registers[12] = 91;
Registers[13] = 24;
Registers[14] = 49;
Registers[15] = 37;
Registers[16] = 63;
Registers[17] = 44;
Registers[18] = 72;
Registers[19] = 87;
```

```
  Registers[18] = 72;
  Registers[19] = 87;
  Registers[20] = 16;
  Registers[21] = 55;
  Registers[22] = 99;
  Registers[23] = 13;
  Registers[24] = 60;
  Registers[25] = 11;
  Registers[26] = 90;
  Registers[27] = 35;
  Registers[28] = 4;
  Registers[29] = 76;
  Registers[30] = 8;
  Registers[31] = 100;

  end

  integer k;
always @(posedge clk) begin
  if (rst)
begin
      for (k = 0; k < 32; k = k + 1) begin
        Registers[k] = 32'b00;
      end
    end

    else if (RegWrite ) begin
      Registers[Rd] = Write_data;
    end
  end

  assign read_data1 = Registers[Rs1];
  assign read_data2 = Registers[Rs2];

endmodule
```

*Hình 2. 7 Code Verilog khối RegisterFile*

### *2.2.5 Data_Mem (Data Memory)*

Trong bộ xử lý RISC-V đơn chu kỳ, bộ nhớ dữ liệu (Data Memory) là thành phần chịu trách nhiệm lưu trữ và truy xuất dữ liệu từ các vị trí bộ nhớ. Nó đóng vai trò như một cầu nối giữa bộ xử lý và bộ nhớ chính (RAM), nơi dữ liệu của chương trình được lưu trữ.

Trong thiết kế đơn chu kỳ, bộ nhớ dữ liệu thực hiện hai thao tác chính:

- Đọc dữ liệu (Data Read): Khi bộ xử lý cần truy xuất dữ liệu từ bộ nhớ, bộ nhớ dữ liệu nhận địa chỉ bộ nhớ từ bộ xử lý, sau đó truy xuất dữ liệu tại vị trí đó và cung cấp dữ liệu về cho bộ xử lý để tiếp tục xử lý.
- Ghi dữ liệu (Data Write): Khi bộ xử lý cần lưu trữ dữ liệu vào bộ nhớ, nó gửi dữ liệu kèm theo địa chỉ bộ nhớ tới bộ nhớ dữ liệu. Sau đó, bộ nhớ sẽ ghi dữ liệu đó vào vị trí bộ nhớ được chỉ định trong RAM.

Vì mỗi lệnh trong kiến trúc đơn chu kỳ chỉ thực thi trong một chu kỳ xung nhịp duy nhất, các thao tác truy xuất dữ liệu trong bộ nhớ cũng phải được hoàn tất trong một chu kỳ xung nhịp duy nhất. Do đó, bộ nhớ dữ liệu cần được thiết kế tối ưu về tốc độ và hiệu suất, nhằm đảm bảo các thao tác đọc/ghi dữ liệu diễn ra nhanh chóng và đồng bộ với hoạt động của toàn bộ hệ thống xử lý.

```
module Data_Mem( clk, rst, MemRead, MemWrite, address, write_data, read_data );
    input clk;
    input rst;
    input MemRead;
    input MemWrite;
    input [31:0] address;
    input [31:0] write_data;
    output[31:0] read_data;
  reg [31:0] D_Memory [63:0];
  integer k;


  assign read_data = (MemRead) ? D_Memory[address] : 32'b00;

  always @(posedge clk or posedge rst) begin
    if (rst ) begin
      for (k = 0; k < 64; k = k + 1) begin
        D_Memory[k] = 32'b00;
      end
    end else if (MemWrite) begin
      D_Memory[address] = write_data;
    end
  end

endmodule
```

*Hình 2. 8 Code Verilog của khối Data Memory*

### *2.2.6 ALU_Control (ALU Control)*

Bộ điều khiển ALU (ALU Control) trong bộ xử lý RISC-V đơn chu kỳ có nhiệm vụ xác định phép toán cụ thể mà Bộ số học và logic (ALU) cần thực hiện trên các toán hạng, dựa vào mã lệnh (opcode) của lệnh hiện tại.

ALU Control sử dụng các tín hiệu điều khiển (Control Signals) để lựa chọn đúng chức năng của ALU, từ đó cho phép thực hiện hiệu quả các phép toán số học và logic như cộng, trừ, AND, OR, so sánh,... trong một chu kỳ xung nhịp duy nhất.

Bộ điều khiển ALU là một phần cốt lõi trong luồng điều khiển của datapath, đảm bảo rằng việc giải mã lệnh và thực thi toán học được đồng bộ hóa, phù hợp với thiết kế đơn giản, hiệu quả và có tính tuần tự cao của kiến trúc RISC-V.

```
module ALU_Control(funct3, funct7, ALUOp, ALUcontrol_Out);
    input [2:0] funct3;
    input [6:0]funct7;
    input [1:0] ALUOp;
    output reg [3:0] ALUcontrol_Out;

always @(*) begin
  case ({ALUOp, funct7, funct3})
    12'b10_0000000_000 : ALUcontrol_Out <= 4'b0000;    // ADD
    12'b00_0000000_000 : ALUcontrol_Out <= 4'b0000;    // ADD
    12'b00_0000000_001 : ALUcontrol_Out <= 4'b0000;    // ADD
    12'b00_0000000_010 : ALUcontrol_Out <= 4'b0000;    // ADD
    12'b10_0100000_000 : ALUcontrol_Out <= 4'b0001;    // SUB
    12'b10_0000000_111 : ALUcontrol_Out <= 4'b0010;    // AND
    12'b10_0000000_110 : ALUcontrol_Out <= 4'b0011;    // OR
    12'b10_0000000_100 : ALUcontrol_Out <= 4'b0100;    // XOR
    12'b10_0000000_001 : ALUcontrol_Out <= 4'b0101;    // SLL
    12'b10_0000000_101 : ALUcontrol_Out <= 4'b0110;    // SRL
    12'b10_0100000_101 : ALUcontrol_Out <= 4'b0111;    // SRA
    12'b10_0000000_010 : ALUcontrol_Out <= 4'b1000;    // SLT
    default            : ALUcontrol_Out <= 4'b0000;
  endcase
end


endmodule
```

*Hình 2. 9 Code Verilog của ALU_Control*

### *2.2.7 Control_Unit (Control Unit)*

Bộ điều khiển (Control Unit) có chức năng giải mã lệnh và tạo ra các tín hiệu điều khiển để đồng bộ hoạt động của các thành phần khác nhau trong bộ xử lý RISC-V đơn chu kỳ.

Thông qua việc phân tích trường opcode trong lệnh, bộ điều khiển xác định loại lệnh đang được thực thi (R-type, I-type, S-type, v.v.), từ đó sinh ra các tín hiệu điều khiển thích hợp như: RegWrite, ALUSrc, MemRead, MemWrite, Branch, ALUOp,... Những tín hiệu này điều phối hoạt động của các khối chức năng như thanh ghi, ALU, bộ nhớ dữ liệu, và bộ chọn (multiplexer), đảm bảo rằng lệnh được thực thi chính xác trong một chu kỳ xung nhịp.

Bộ điều khiển đóng vai trò trung tâm trong kiến trúc bộ xử lý, quyết định đến dòng dữ liệu và trình tự hoạt động của từng thành phần trong datapath, từ đó đảm bảo hiệu quả và tính đúng đắn trong quá trình xử lý lệnh.

```
module Control_Unit( opcode, RegWrite, MemRead, MemWrite, MemToReg, ALUSrc, Branch, ALUOp );
    input [6:0] opcode;
    output reg RegWrite;
    output reg MemRead;
    output reg MemWrite;
    output reg MemToReg;
    output reg ALUSrc;
    output reg Branch;
    output reg [1:0] ALUOp;

    always @(*) begin
        case (opcode)
            7'b0110011:  // R-type
            begin  {ALUSrc, MemToReg, RegWrite, MemRead, MemWrite, Branch, ALUOp} <= {1'b0, 1'b0, 1'b1, 1'b0, 1'b0, 1'b0, 2'b10};end
            7'b0010011:   // I-type
            begin {ALUSrc, MemToReg, RegWrite, MemRead, MemWrite, Branch, ALUOp} <= {1'b1, 1'b0, 1'b1, 1'b0, 1'b0, 1'b0, 2'b10};end
            7'b0000011:   // Load
            begin {ALUSrc, MemToReg, RegWrite, MemRead, MemWrite, Branch, ALUOp} <= {1'b1, 1'b1, 1'b1, 1'b1, 1'b0, 1'b0, 2'b00};end
            7'b0100011:   // Store
            begin {ALUSrc, MemToReg, RegWrite, MemRead, MemWrite, Branch, ALUOp} <= {1'b1, 1'b0, 1'b0, 1'b0, 1'b1, 1'b0, 2'b00};end
            7'b1100011:   // Branch
            begin {ALUSrc, MemToReg, RegWrite, MemRead, MemWrite, Branch, ALUOp} <= {1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b1, 2'b11};end
            7'b1101111:   // Jump
            begin  {ALUSrc, MemToReg, RegWrite, MemRead, MemWrite, Branch, ALUOp} <= {1'b0, 1'b0, 1'b1, 1'b0, 1'b0, 1'b0, 2'b10};end
            7'b0110111:   // LUI
            begin {ALUSrc, MemToReg, RegWrite, MemRead, MemWrite, Branch, ALUOp} <= {1'b0, 1'b0, 1'b1, 1'b0, 1'b0, 1'b0, 2'b10};end
            default:
            begin {ALUSrc, MemToReg, RegWrite, MemRead, MemWrite, Branch, ALUOp} <= {1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 2'b00};end
        endcase
    end
```

*Hình 2. 10 Code Verilog của Control_Unit*

# CHƯƠNG 3: KẾT QUẢ MÔ PHỎNG

## 3.1 Môi trường mô phỏng

- **Phần mềm mô phỏng:** ModelSim
- **Ngôn ngữ sử dụng:** Verilog HDL
- **Testbench chính:** Tb.v
- **Module tích hợp:** Top.v
- **Hệ thống mô phỏng:** mô phỏng một phần mềm xử lý lệnh RISC-V cơ bản, bao gồm các thành phần: ALU, Register File, Control Unit, Data Memory, Instruction Memory, Immediate Generator, Program Counter, các bộ MUX, và hệ thống điều hướng nhảy (Branch).

## 3.2 Quy trình mô phỏng

Quá trình mô phỏng được thực hiện theo các bước sau:

1. **Tạo project trong ModelSim** và thêm toàn bộ các file .v vào.
2. **Set module testbench** là tb_top.v.
3. **Compile toàn bộ dự án** để đảm bảo không lỗi cú pháp.
4. **Chạy mô phỏng** với các lệnh trong ModelSim Console:
    - vsim work.tb_top
    - add wave *
    - run 1000ns
5. **Quan sát waveform** các tín hiệu đầu ra, kiểm tra hoạt động của hệ thống.

## 3.3 Các tín hiệu được kiểm tra

Một số tín hiệu quan trọng được theo dõi trong quá trình mô phỏng:

Bảng 2 *Các tín hiệu được kiểm tra*

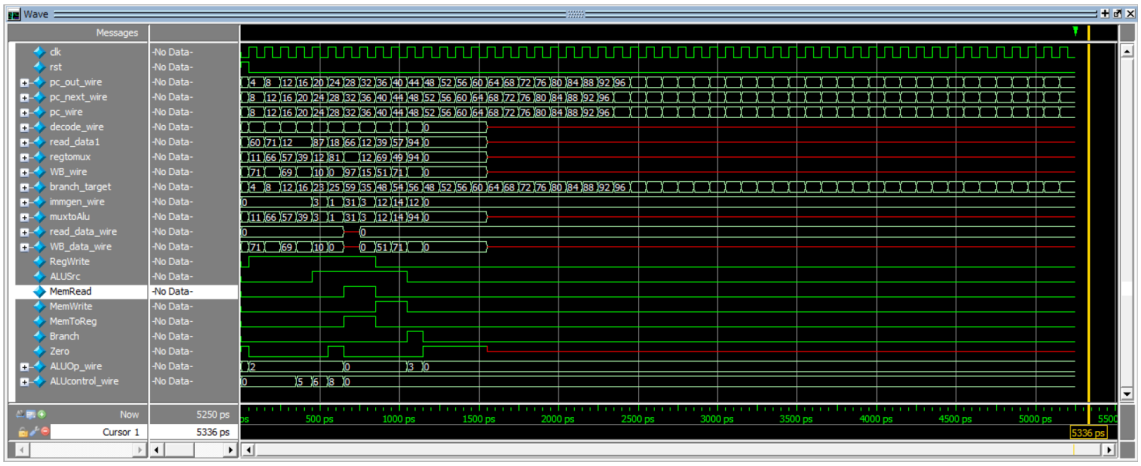
| **Tín hiệu** | **Chức năng** |
|---|---|
| PC_Out_debug | Giá trị bộ đếm chương trình hiện tại |
| Instruction_debug | Lệnh được nạp từ bộ nhớ lệnh |
| ALU_Result_debug | Kết quả từ khối ALU |
| MemRead, MemWrite | Tín hiệu điều khiển đọc/ghi dữ liệu từ RAM |
| RegWrite | Tín hiệu ghi dữ liệu vào thanh ghi |
| Branch, Zero | Tín hiệu điều hướng rẽ nhánh |

| Tín hiệu | Chức năng |
|---|---|
| WriteData | Dữ liệu được ghi vào thanh ghi hoặc bộ nhớ dữ liệu |

## 3.4 Kết quả mô phỏng

Khi thực hiện mô phỏng trên tb_top.v, hệ thống hoạt động như sau:

- **Bộ đếm chương trình (PC)** cập nhật chính xác theo từng chu kỳ.
- **Instruction Memory** cung cấp đúng lệnh tại mỗi địa chỉ PC.
- **ALU** thực hiện đúng phép toán với dữ liệu đọc từ thanh ghi.
- **Data Memory** thực hiện đúng thao tác đọc/ghi khi có tín hiệu điều khiển phù hợp.
- **Các tín hiệu điều khiển** (Control Signals) hoạt động đúng với loại lệnh (R-type, I-type, S-type, B-type...).




*Hình 3. 1 Kết quả mô phỏng trên ModelSim*

# TÀI LIỆU THAM KHẢO

[1] C. W. Chien, “Design and Implementation of a RISC-V Processor,” UTAR Institutional Repository, Oct. 2022. [Online]. Available: http://eprints.utar.edu.my/5966/

[2] RISC-V Foundation, “RISC-V Specifications.” [Online]. Available: https://riscv.org/specifications/

[3] emTutayto, “RISCV_SingleCycle,” GitHub. [Online]. Available: https://github.com/emTutayto/RISCV_SingleCycle