ĐẠI HỌC BÁCH KHOA HÀ NỘI

**TRƯỜNG ĐIỆN – ĐIỆN TỬ**



**BÁO CÁO KIẾN TRÚC MÁY TÍNH NÂNG CAO**

**Đề tài:**

**THIẾT KẾ BỘ XỬ LÝ RISC V PIPELINED PROCESSOR.**

|  |  |
| --- | --- |
| Sinh viên thực hiện: | **Lê Tiến Đạt** – 20240002E |
|  | dat.lt240002E@sis.hust.edu.vn |
|  |  |
| Giảng viên hướng dẫn: | **TS Tạ Thị Kim Huệ** |

Hà Nội, 04 – 2025

ĐẠI HỌC BÁCH KHOA HÀ NỘI

**TRƯỜNG ĐIỆN – ĐIỆN TỬ**



**BÁO CÁO KIẾN TRÚC MÁY TÍNH NÂNG CAO**

**Đề tài:**

**THIẾT KẾ BỘ XỬ LÝ RISC V PIPELINED PROCESSOR.**

|  |  |
| --- | --- |
| Sinh viên thực hiện: | **Lê Tiến Đạt** – 20240002E |
|  | dat.lt240002E@sis.hust.edu.vn |
|  |  |
| Giảng viên hướng dẫn: | **TS Tạ Thị Kim Huệ** |

Hà Nội, 04 – 2025

DANH MỤC KÝ HIỆU VÀ CHỮ VIẾT TẮT

|  |  |
| --- | --- |
| Cobot | Collaborative robot (người máy cộng tác) |
| API | Application programming interface (giao diện lập trình ứng dụng) |
| SOAP | Simple object access protocol (giao thức truy cập đối tượng đơn giản) |
| REST | Representational state transfer (chuyển trạng thái đại diện) |
| RPC | Remote procedure call (gọi thủ tục từ xa) |

DANH MỤC HÌNH VẼ

DANH MỤC BẢNG BIỂU

CHƯƠNG 1: GIỚI THIỆU

* 1. Giới thiệu đề tài

Trong thời đại công nghệ số phát triển mạnh mẽ, các bộ xử lý (CPU) đóng vai trò trung tâm trong hầu hết các hệ thống nhúng và máy tính. Gần đây, kiến trúc RISC-V – một tập lệnh mở và miễn phí bản quyền – đã trở thành xu hướng mới trong lĩnh vực thiết kế vi xử lý. RISC-V cho phép sinh viên, nhà nghiên cứu, và các kỹ sư có thể tự thiết kế và mở rộng bộ xử lý theo nhu cầu thực tế.

Trong khuôn khổ đồ án này, nhóm thực hiện đề tài “**Thiết kế và mô phỏng bộ xử lý RISC-V Pipeline sử dụng Verilog**”. Bộ xử lý được mô phỏng theo kiến trúc RISC-V 32-bit (RV32I), hỗ trợ các lệnh cơ bản như: ADD, SUB, AND, OR, LW, SW, BEQ, JAL,... với mô hình xử lý đướng ống (pipeline processor). Tất cả các khối chức năng được xây dựng bằng ngôn ngữ mô tả phần cứng Verilog và được mô phỏng bằng phần mềm ModelSim.

* 1. Mục tiêu đề tài
* Thiết kế một CPU RISC-V đường ống 32-bit bằng ngôn ngữ Verilog.
* Hỗ trợ tập lệnh cơ bản RV32I: add, sub, addi, lw, sw,...
* Thực hiện mô phỏng và kiểm chứng hoạt động của CPU bằng phần mềm ModelSim.
* Kiểm tra kết quả thực thi thông qua waveform và testbench.

CHƯƠNG 2: CƠ SỞ LÝ THUYẾT

2.1 Kiến trúc tập lệnh

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.

A close-up of a sign

AI-generated content may be incorrect.

Hình 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.

2.2 Cấu trúc CPU đường ống (5-stage pipeline)

Với kiến trúc dữ liệu đơn chu kỳ truyền thống, mỗi lệnh được thực hiện trong một chu kỳ xung nhịp. Lệnh tiếp theo phải chờ cho đến khi lệnh trước hoàn tất mới có thể bắt đầu thực hiện. Hơn nữa, thời gian thực hiện mỗi lệnh có thể khác nhau. Có những lệnh mất nhiều thời gian hơn các lệnh khác để thực hiện. Điều này có thể dẫn đến lãng phí chu kỳ xung nhịp và làm giảm hiệu suất của bộ xử lý. Tuy nhiên, vấn đề này có thể được giải quyết bằng cách sử dụng kỹ thuật đường ống (pipelining).

Đường ống là một kỹ thuật trong kiến trúc máy tính được sử dụng để tăng hiệu suất tổng thể của bộ xử lý. Kỹ thuật này phân chia quá trình thực hiện một tác vụ thành nhiều giai đoạn nhỏ và cho phép các giai đoạn này chồng lấp nhau về mặt thời gian. Nhờ đó, nhiều lệnh có thể được thực thi đồng thời, từ đó cải thiện thông lượng của hệ thống.

Kỹ thuật đường ống phổ biến nhất trong thiết kế bộ xử lý hiện đại là đường ống 5 giai đoạn. Đúng như tên gọi, chu trình thực hiện lệnh được chia thành 5 giai đoạn riêng biệt. Mỗi giai đoạn sẽ thực hiện một thao tác cụ thể trên dữ liệu đầu vào và chuyển kết quả cho giai đoạn tiếp theo. Kết quả của giai đoạn đầu tiên sẽ trở thành đầu vào của giai đoạn thứ hai, và cứ như vậy. Ngoài ra, mỗi giai đoạn xử lý một phần khác nhau của dữ liệu, cho phép nhiều lệnh được xử lý đồng thời ở các giai đoạn khác nhau. Năm giai đoạn của một đường ống tiêu chuẩn bao gồm: nạp lệnh (fetch), giải mã (decode), thực thi (execute), bộ nhớ (memory) và ghi kết quả (writeback). Chi tiết của từng giai đoạn được trình bày dưới đây:

**Fetch – Nạp lệnh**

Ở giai đoạn này, bộ xử lý sẽ lấy lệnh từ bộ nhớ. Sau đó, lệnh được nạp vào bộ đệm lệnh (instruction cache), nơi lưu trữ các lệnh đã sử dụng gần đây.

**Decode – Giải mã**

Trong giai đoạn này, bộ xử lý sẽ giải mã lệnh đã nạp để xác định thao tác cần thực hiện. Lệnh được phân tích để xác định loại thao tác, các thanh ghi và vị trí của các toán hạng (operand) liên quan.

**Execute – Thực thi**

Tại giai đoạn này, bộ xử lý thực hiện thao tác được chỉ định trong lệnh. Điều này bao gồm các phép toán số học hoặc logic, chẳng hạn như cộng và trừ. Ngoài ra, có thể bao gồm việc truy cập bộ nhớ hoặc nhảy đến một phần khác của chương trình.

**Memory – Truy cập bộ nhớ**

Trong giai đoạn này, bộ xử lý truy cập bộ nhớ để đọc hoặc ghi dữ liệu được tạo ra từ giai đoạn trước. Tuy nhiên, đây là một giai đoạn tùy chọn – một số lệnh như lệnh "add" không yêu cầu truy cập bộ nhớ.

**Writeback – Ghi kết quả**

Ở giai đoạn cuối cùng này, kết quả của thao tác được ghi trở lại vào thanh ghi phù hợp trong tập thanh ghi (register file).



Hình 2: 5-stage pipeline

2.3 Phần mềm mô phỏng Model

ModelSim là một công cụ mô phỏng và xác minh phổ biến dành cho các mạch 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 thiết kế trước khi đưa vào triển khai trên phần cứng thực tế.

ModelSim cung cấp một tập hợp các tính năng mạnh mẽ để thiết kế, mô phỏng và xác minh các mạch và hệ thống số. Về mặt mô phỏng, ModelSim hỗ trợ cả hai ngôn ngữ Verilog và VHDL. Nó có khả năng mô phỏng ở tất cả các mức trừu tượng, từ mức cổng logic (gate-level) cho đến mức hành vi (behavioral-level). Về xác minh, ModelSim hỗ trợ mô phỏng chức năng (functional simulation) và mô phỏng thời gian (timing simulation), cũng như xác minh dựa trên mệnh đề (assertion-based verification). Ngoài ra, nó còn có thể tích hợp với các công cụ xác minh khác như Questa và UVM để thực hiện quá trình kiểm thử toàn diện hơn. Về thiết kế, ModelSim hỗ trợ việc tạo cấu trúc phân cấp thiết kế (design hierarchy), giúp các nhà thiết kế tổ chức các phần trong thiết kế thành các khối logic rõ ràng. Bên cạnh đó, nó cũng cung cấp các tính năng kiểm tra thiết kế như kiểm tra cú pháp (syntax checking) và kiểm tra quy tắc mã (linting).

A blue cover with text and graphics

AI-generated content may be incorrect.

Hình 3: ModelSim

Có nhiều lợi ích khi sử dụng ModelSim. Trước hết, ModelSim giúp nâng cao chất lượng thiết kế. Nó giúp phát hiện và khắc phục các lỗi thiết kế ngay từ sớm trong vòng đời phát triển, từ đó nâng cao chất lượng của sản phẩm cuối cùng. Tiếp theo, ModelSim giảm nhu cầu phải tạo mẫu phần cứng đắt tiền bằng cách cung cấp một môi trường mô phỏng ảo để kiểm thử và xác minh thiết kế. Điều này giúp tiết kiệm chi phí trong quá trình thiết kế. Hơn nữa, ModelSim còn hỗ trợ các ngôn ngữ và giao diện theo tiêu chuẩn công nghiệp, góp phần thúc đẩy sự chuẩn hóa và khả năng tương thích giữa các công cụ trong ngành điện tử.

Trong dự án này, ModelSim sẽ là phần mềm chính được sử dụng để thiết kế và mô phỏng bộ xử lý RISC-V. Nhiều mô-đun như bộ nhớ lệnh (instruction memory), bộ cộng (adder), thanh ghi (register), bộ nhớ dữ liệu (data memory), bộ logic số học (ALU) và điều khiển ALU (ALU control) sẽ được lập trình bằng Verilog trong ModelSim. Chức năng của từng phần tử sẽ được kiểm thử và xác minh thông qua mô phỏng bằng ModelSim. Cuối cùng, tất cả các phần tử này sẽ được tích hợp vào một mô-đun chính để hình thành thiết kế tổng thể của bộ xử lý RISC-V 32-bit sử dụng đường ống 5 giai đoạn.

A diagram of a flowchart

AI-generated content may be incorrect.

Hình 4: Quy trình thực hiện của dự án

CHƯƠNG 3: THIẾT KẾ HỆ THỐNG VÀ MÔ TẢ THIẾT KẾ

3.1 Tổng quan kiến trúc phần cứng

Bộ xử lý bao gồm các thành phần: ALU (bộ logic số học), ALU decoder (aludec), bộ giải mã chính (maindec), bộ nhớ lệnh (imem), bộ nhớ dữ liệu (dmem), các thanh ghi đường ống IF\_ID, ID\_IEx, IEx\_IMem, Imem\_IW, bộ chọn bộ đếm chương trình (pc\_mux), tập thanh ghi (reg\_file), bộ chọn chuyển tiếp A (forwardMuxA) và bộ chọn chuyển tiếp B (forwardMuxB). Mỗi thành phần trong đường dữ liệu (datapath) sẽ được trình bày chi tiết trong phần này.

3.2 Triển khai thiết kế

3.2.1 Arithmetic Logic Unit (alu)

Bộ logic số học (ALU) là một thành phần cơ bản của bộ xử lý trung tâm (CPU). Nó có nhiệm vụ thực hiện các phép toán số học và logic trên dữ liệu nhị phân. ALU nhận dữ liệu đầu vào từ thanh ghi đường ống ID\_IEx và thực hiện các phép toán số học khác nhau dựa trên tín hiệu điều khiển từ ALU decoder (aludec). Kết quả của các phép toán này được lưu vào biến ALUResults. Trong thiết kế bộ xử lý RISC-V này, có 9 lệnh được hiện thực trong ALU.

module *alu* (

    input [31:0] SrcA,

    input [31:0] SrcB,

    input [3:0] ALUControl,

    output reg [31:0] ALUResult,

    output Zero,

    output Sign

);

    wire [31:0] Sum;

    wire Overflow;

    assign Sum = SrcA + (ALUControl[0] ? ~SrcB : SrcB) + ALUControl[0];

    assign Overflow = ~(ALUControl[0] ^ SrcB[31] ^ SrcA[31]) & (SrcA[31] ^ Sum[31]) & (~ALUControl[0]);

    assign Zero = ~(|ALUResult);

    assign Sign = ALUResult[31];

    always @(\*) begin

        case (ALUControl)

            4'b000x: ALUResult = Sum; // sum or difference

            4'b0010: ALUResult = SrcA & SrcB; // and

            4'b0011: ALUResult = SrcA | SrcB; // or

            4'b0100: ALUResult = SrcA << SrcB; // sll, slli

            4'b0101: ALUResult = {{30{1'b0}}, Overflow ^ Sum[31]}; // slt, slti

            4'b0110: ALUResult = SrcA ^ SrcB; // xor

            4'b0111: ALUResult = SrcA >> SrcB; // srl, srli

            4'b1000: ALUResult = ($unsigned(SrcA) < $unsigned(SrcB));

            4'b1111: ALUResult = SrcA >>> SrcB; // shift arithmetic

            default: ALUResult = 32'bx;

        endcase

    end

endmodule

Bảng sau trình bày các lệnh số nguyên cơ bản cho kiến trúc RV321.

|  |  |  |
| --- | --- | --- |
| Category | Mnemonic | Description |
| **Arithmetic** | | |
| ADD | rd, rs1, rs2 | rd = rs1 + rs2 |
| SUB | rd, rs1, rs2 | rd = rs1 - rs2 |
| **Logical** | | |
| XOR | rd, rs1, rs2 | rd = rs1 ^ rs2 |
| AND | rd, rs1, rs2 | rd = rs1 & rs2 |
| OR | rd, rs1, rs2 | rd = rs1 | rs2 |
| **Shifts** | | |
| SHL | rd, rs1, rs2 | rd = rs1 << rs2 |
| SHR | rd, rs1, rs2 | rd = rs1 >> rs2 |
| **Compare** | | |
| SLT | rd, rs1, rs2 | rd = rs1 < rs2 ? 1 : 0 |
| SLTU | rd, rs1, rs2 | rd = rs1 < rs2 ? 1 : 0 |

3.2.2 ALU decode (aludec)

Bộ giải mã logic số học (ALU decoder – aludec) được sử dụng để giải mã các lệnh. Nó nhận tín hiệu từ bộ giải mã chính (maindec) và xác định loại phép toán mà ALU cần thực hiện. Việc giải mã được thực hiện bằng cách kết hợp 4 đầu vào: ALUOp, funct3, funct7b5 và opb5.

* **funct7b5** tương ứng với các bit từ 31 đến 25 trong lệnh (instruction[31:25])
* **funct3** tương ứng với các bit từ 14 đến 12 (instruction[14:12])
* **opb5** tương ứng với các bit từ 0 đến 6 (instruction[6:0]) trong định dạng lệnh của RISC-V.

Biến RTypeSub được tạo ra bằng cách thực hiện phép AND giữa funct7b5 và opb5, nhằm phân biệt giữa hai lệnh ADD và SUB thuộc loại lệnh R-type.

module *aludec* (

    input opd5,

    input [2:0] funct3,

    input funct7b5,

    input [1:0] ALUOP,

    output reg [3:0] ALUControl

);

    wire RtypeSub;

    assign RtypeSub = funct7b5 & opd5;

    always @(\*) begin

        case (ALUOP)

            2'b00: ALUControl = 4'b0000; // addi, lw, sw, beq, jalr

            2'b01: ALUControl = 4'b0001; // slti, sltiu

            default: case (funct3)

                3'b000: if (RtypeSub) ALUControl = 4'b0001; // sub

                        else ALUControl = 4'b0000; // add, addi

                3'b001: ALUControl = 4'b0100; // sll, slli

                3'b010: ALUControl = 4'b0101; // slt, slti

                3'b011: ALUControl = 4'b1000; // sltu, sltiu

                3'b100: ALUControl = 4'b0100; // xor, xori

                3'b101: if (~funct7b5) ALUControl = 4'b0111; // srl, srli

                        else ALUControl = 4'b1111; // sra, srai

                3'b110: ALUControl = 4'b0011; // or, ori

                3'b111: ALUControl = 4'b0010; // and, andi

                default: ALUControl = 4'bxxxx; // undefined

            endcase

        endcase

    end

endmodule

Đầu ra aludec tương ứng với trường hợp được chọn như thể hiện trong Bảng dưới.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *ALUOp* | *RTypeSub* | *funct3* | Alu decode | Operation |
| 2’b10 | 0 | 3’b000 | 4’b0000 | AND |
| 2’b10 | 1 | 3’b000 | 4’b0001 | SUB |
| 2’b10 | 0 | 3’b111 | 4’b0010 | AND |
| 2’b10 | 0 | 3’b110 | 4’b0011 | OR |
| 2’b10 | 0 | 3’b001 | 4’b0100 | SLLI |
| 2’b10 | 0 | 3’b011 | 4’b0101 | SLTI |
| 2’b10 | 0 | 3’b100 | 4’b0110 | XOR |
| 2’b10 | 0 | 3’b101 | 4’b0111 | SHR |
| 2’b10 | 0 | 3’b101 | 4’b1000 | SLTU |
| 2’b10 | 0 | 3’b101 | 4’b1111 | SHL |

3.2.3 Main decoder (maindec)

Bộ giải mã chính (maindec) được sử dụng để tạo ra các tín hiệu điều khiển từ mã lệnh 7 bit (opcode) nằm tại instruction[6:0], nhằm xác định loại lệnh đang được xử lý. Các tín hiệu điều khiển bao gồm: RegWrite, ImmSrc, ALUSrcA, ALUSrcB, MemWrite, ResultSrc, Branch, ALUOp và Jump. Mỗi tín hiệu điều khiển này điều khiển hoạt động của các bộ chọn (multiplexer) trong đường dữ liệu (datapath), giúp định hướng luồng dữ liệu phù hợp theo từng loại lệnh được thực thi.

module *maindec* (

    input [6:0] op,

    output [1:0] ResultSrc,

    output MemWrite,

    output Branch, ALUSrcA,

    output [1:0] ALUSrcB,

    output RegWrite, Jump,

    output [2:0] ImmSrc,

    output [1:0] ALUOp

);

    reg [13:0] controls;

    assign {RegWrite, ImmSrc, ALUSrcA, ALUSrcB, MemWrite, ResultSrc, Branch, ALUOP, Jump} = controls;

    always @(\*) begin

        case (op)

            7'b0000011: controls = 14'b1\_000\_0\_01\_0\_01\_0\_00\_0; // lw

            7'b0100011: controls = 14'b0\_001\_0\_01\_1\_00\_0\_00\_0; // sw

            7'b0110011: controls = 14'b1\_xxx\_0\_00\_0\_00\_0\_10\_0; // R–type

            7'b1100011: controls = 14'b0\_010\_0\_00\_0\_00\_1\_01\_0; // B-type

            7'b0010011: controls = 14'b1\_000\_0\_01\_0\_00\_0\_10\_0; // I–type

            7'b1101111: controls = 14'b1\_011\_0\_00\_0\_10\_0\_00\_1; // jal

            7'b0010111: controls = 14'b1\_100\_1\_10\_0\_00\_0\_00\_0; // auipc

            7'b0110111: controls = 14'b1\_100\_1\_01\_0\_00\_0\_00\_0; // lui

            7'b1100111: controls = 14'b1\_000\_0\_01\_0\_10\_0\_00\_1; // jalr

            7'b0000000: controls = 14'b0\_000\_0\_00\_0\_00\_0\_00\_0; // for default values on reset

            default:    controls = 14'bx\_xxx\_x\_xx\_x\_xx\_x\_xx\_x; // instruction not implemented

        endcase

    end

endmodule

Việc bật và tắt các tín hiệu này dựa trên các loại hướng dẫn được lập bảng trong Bảng sau:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Control Signal | Instruction | | | | | |
| lw | sw | R-Type | B-Type | I-Type | jal |
| *RegWrite* | 1 | 0 | 1 | 0 | 1 | 1 |
| *ImmSrc* | 000 | 001 | xxx | 010 | 000 | 011 |
| *ALUSrcA* | 0 | 0 | 0 | 0 | 0 | 0 |
| *ALUSrcB* | 01 | 01 | 00 | 00 | 01 | 00 |
| *MemWrite* | 0 | 1 | 0 | 0 | 00 | 0 |
| *ResultSrc* | 01 | 00 | 00 | 00 | 000 | 10 |
| *Branch* | 0 | 0 | 0 | 1 | 0 | 0 |
| *ALUOp* | 00 | 00 | 10 | 01 | 10 | 00 |
| *Jump* | 0 | 0 | 0 | 0 | 0 | 1 |

3.2.4 Data memory (dmem)

Trong kiến ​​trúc máy tính, bộ nhớ dữ liệu là một thành phần của hệ thống máy tính có trách nhiệm lưu trữ và truy xuất dữ liệu. Bộ nhớ dữ liệu thường được sử dụng để lưu trữ dữ liệu đang được alu xử lý tích cực. Thông thường có 3 đầu vào trong mô-đun này, đó là write\_enable, data\_address và write\_data. Mô-đun lấy địa chỉ bộ nhớ từ kết quả của alu (data\_address) và dữ liệu từ tệp đăng ký (write\_data). Write enable (write\_enable) được sử dụng để kiểm soát quyền ghi dữ liệu vào bộ nhớ dữ liệu.

module *dmem*(

    input clk, we,

    input [31:0] a, wd,

    output [31:0] rd

);

    reg [31:0] RAM[63:0];

    assign rd = RAM[a[31:2]];

    always @(posedge clk)

        if (we) RAM[a[31:2]] <= wd;

endmodule

3.2.5 Instruction memory (imem)

Trong kiến ​​trúc máy tính, bộ nhớ lệnh là một thành phần của hệ thống máy tính lưu trữ lệnh của một chương trình. Bộ nhớ này có nhiệm vụ lưu trữ trình tự lệnh mà CPU lấy, giải mã và thực thi trong quá trình thực thi chương trình. Trong mô-đun này, bộ lệnh 32 bit được tạo ra và lưu trữ trong mảng ram. Lệnh cần lấy dựa trên đầu vào bộ đếm chương trình lấy (PCF). Vì mỗi lệnh là 4 byte, giá trị của PCF sẽ tăng thêm 4 để lấy lệnh tiếp theo.

module *imem*(

    input [31:0] a,

    output [31:0] rd

);

reg [7:0] RAM[128:0]; // 128 x 8 = byte addressable memory with 128 locations

assign rd = {RAM[a + 3], RAM[a + 2], RAM[a + 1], RAM[a + 0]};

initial

    begin

    end

**//đoạn này sẽ thêm các lệnh RISC-V được chuyển về 8 bit hex**

**// ví dụ RAM[i] = 8’hxx;**

endmodule

3.2.6 Thanh ghi đường ống

Trong kiến ​​trúc máy tính, pipeline register là một thành phần lưu trữ tạm thời được sử dụng trong pipeline của bộ xử lý để lưu trữ dữ liệu giữa các giai đoạn khác nhau của quá trình thực thi lệnh. Nó đóng vai trò là điểm đồng bộ hóa giữa các giai đoạn liền kề, cho phép lệnh chạy qua pipeline theo cách được kiểm soát. Trong pipeline 5 giai đoạn, có 4 pipeline register là IF\_ID, ID\_IEx, IEx\_IMem và IMem\_IW. Các register được đặt tên theo hai giai đoạn được phân tách bởi register đó. Ví dụ, pipeline register đầu tiên là IF\_ID vì nó phân tách các giai đoạn lấy lệnh và giải mã lệnh.

3.2.6.1 IF\_ID

Thanh ghi IF\_ID như tên gọi của nó, nó tách biệt các giai đoạn lấy lệnh và giải mã lệnh. Nó được sử dụng để lưu trữ dữ liệu như lệnh lấy lệnh từ bộ nhớ lệnh và sẵn sàng để giải phóng cho giai đoạn giải mã ở chu kỳ xung nhịp tiếp theo. Bên cạnh đó, PC hiện tại và địa chỉ PC tăng tiếp theo (PCPlus4F) cũng được lưu trong thanh ghi IF\_ID trong trường hợp cần thiết sau này cho một lệnh, chẳng hạn như beq.

module *IF\_ID* (

    input clk, reset, clear, enable,

    input [31:0] InstrF, PCF, PCPlus4F,

    output reg [31:0] InstrD, PCD, PCPlus4D

);

always @( posedge clk or posedge reset ) begin

    if (reset) begin

        InstrD <= 0;

        PCD <= 0;

        PCPlus4D <= 0;

    end else if (enable) begin

         if (clear) begin

              InstrD <= 0;

              PCD <= 0;

              PCPlus4D <= 0;

         end

         else begin

              InstrD <= InstrF;

              PCD <= PCF;

              PCPlus4D <= PCPlus4F;

         end

     end

end

endmodule

3.2.6.2 ID\_IEx

Thanh ghi ID\_IEx như tên gọi của nó, nó tách biệt các giai đoạn giải mã lệnh và thực thi. Nó được sử dụng để lưu trữ thông tin như dữ liệu đọc (RD1, RD2) từ tệp thanh ghi và giá trị tức thời mở rộng (ImmExt). Bên cạnh đó, nó chuyển tiếp dữ liệu của PC và PCPlus4F từ thanh ghi IF\_ID. Instruction[11:7] (rd), Instruction[19:15] (rs1) và Instruction[24:20] (rs2) cũng sẽ được lưu trữ vào thanh ghi ID\_IEx và gửi đến Hazard Unit trong giai đoạn thực thi để xử lý nguy hiểm.

module *ID\_IEx*  (

    input clk, reset, clear,

    input [31:0] RD1D, RD2D, PCD,

    input [4:0] Rs1D, Rs2D, RdD,

    input [31:0] ImmExtD, PCPlus4D,

    output reg [31:0] RD1E, RD2E, PCE,

    output reg [4:0] Rs1E, Rs2E, RdE,

    output reg [31:0] ImmExtE, PCPlus4E

);

    always @( posedge clk or posedge reset ) begin

        if (reset) begin

            RD1E <= 0;

            RD2E <= 0;

            PCE <= 0;

            Rs1E <= 0;

            Rs2E <= 0;

            RdE <= 0;

            ImmExtE <= 0;

            PCPlus4E <= 0;

        end else if (clear) begin

            RD1E <= 0;

            RD2E <= 0;

            PCE <= 0;

            Rs1E <= 0;

            Rs2E <= 0;

            RdE <= 0;

            ImmExtE <= 0;

            PCPlus4E <= 0;

        end else begin

            RD1E <= RD1D;

            RD2E <= RD2D;

            PCE <= PCD;

            Rs1E <= Rs1D;

            Rs2E <= Rs2D;

            RdE <= RdD;

            ImmExtE <= ImmExtD;

            PCPlus4E <= PCPlus4D;

        end

    end

endmodule

3.2.6.3 IEx\_IMem

Thanh ghi IEx\_IMem như tên gọi của nó, nó tách biệt các giai đoạn thực thi và bộ nhớ. Nó được sử dụng để lưu trữ kết quả ALU (ALUResult) và ghi dữ liệu (Writedata). Đồng thời, Instruction[11:7] (rd) và PCPlus4F cũng được chuyển tiếp từ các thanh ghi đường ống trước đó và được lưu trữ trong thanh ghi IEx\_IMem.

module *IEx\_IMem*(

    input clk, reset,

    input [31:0] ALUResultE, WriteDataE,

    input [4:0] RdE,

    input [31:0] PCPlus4E,

    output reg [31:0] ALUResultM, WriteDataM,

    output reg [4:0] RdM,

    output reg [31:0] PCPlus4M

);

always @( posedge clk or posedge reset ) begin

    if (reset) begin

        ALUResultM <= 0;

        WriteDataM <= 0;

        RdM <= 0;

        PCPlus4M <= 0;

    end else begin

        ALUResultM <= ALUResultE;

        WriteDataM <= WriteDataE;

        RdM <= RdE;

        PCPlus4M <= PCPlus4E;

    end

end

endmodule

3.2.6.4 IMem\_IW

Thanh ghi IMem\_IW như tên gọi của nó, nó tách biệt bộ nhớ và giai đoạn ghi lại. Nó được sử dụng để lưu trữ kết quả ALU (ALUResult) và đọc dữ liệu (ReadData) từ bộ nhớ dữ liệu. Instruction[11:7] (rd) và PCPlus4F cũng được chuyển tiếp từ các thanh ghi đường ống trước đó và lưu trữ trong thanh ghi IMem\_IW.

module *IMem\_IW* (

    input clk, reset,

    input [31:0] ALUResultM, ReadDataM,

    input [4:0] RdM,

    input [31:0] PCPlus4M,

    output reg [31:0] ALUResultW, ReadDataW,

    output reg [4:0] RdW,

    output reg [31:0] PCPlus4W

);

always @( posedge clk or posedge reset ) begin

    if (reset) begin

        ALUResultW <= 0;

        ReadDataW <= 0;

        RdW <= 0;

        PCPlus4W <= 0;

    end else begin

        ALUResultW <= ALUResultM;

        ReadDataW <= ReadDataM;

        RdW <= RdM;

        PCPlus4W <= PCPlus4M;

    end

end

endmodule

3.2.7 Lựa chọn ghi dữ liệu MUX (result\_mux)

ALU có khả năng thực hiện các phép toán số học như phép cộng (A+B) hoặc phép toán logic như phép so sánh bằng (A=B). Tùy thuộc vào các lệnh cụ thể đang được thực thi, đầu ra của ALU có thể là địa chỉ bộ nhớ hoặc kết quả thu được từ phép toán ALU. Để xử lý tình huống này, cần có MUX để đưa ra quyết định giữa việc chọn địa chỉ dữ liệu hoặc giá trị đầu ra của ALU để ghi lại vào tệp thanh ghi. MUX hoạt động như một công tắc chọn một trong hai đầu vào dựa trên tín hiệu điều khiển ResultSrc. Điều này cho phép xử lý dữ liệu linh hoạt và hiệu quả trong đường ống của bộ xử lý.

module *result\_mux* (

    input [31:0] ALUResultW, ReadDataW, PCPlusW,

    input [1:0] ResultSrcW,

    output [31:0] ResultW

);

    assign ResultW = ResultSrcW[1] ? PCPlusW : (ResultSrcW[0] ? ReadDataW : ALUResultW);

endmodule

Đầu ra tương ứng của result\_mux được lập bảng trong Bảng bên dưới:

|  |  |
| --- | --- |
| ResultSrc | output |
| 00 | ALUResultW |
| 01 | ReadDataW |
| 10 | PCPlusW |

3.2.8 Thanh ghi chương trình pc MUX (pc\_mux)

Bộ đếm chương trình (PC) là một thành phần quan trọng được CPU sử dụng để duy trì lệnh hiện tại đang được thực thi. Trong những trường hợp bình thường, bộ đếm chương trình tăng theo một giá trị cố định, thường là 4 (tương ứng với lệnh 32 bit) cho mỗi chu kỳ xung nhịp. Điều này đảm bảo rằng bộ đếm chương trình luôn trỏ đến địa chỉ bộ nhớ của lệnh tiếp theo sẽ được thực thi. Tuy nhiên, bộ đếm chương trình có thể bị ngắt hoặc sửa đổi bằng tín hiệu nhảy (jump) từ bộ điều khiển. Khi các điều kiện được xác định trước được đáp ứng, bộ điều khiển sẽ hướng dẫn bộ đếm chương trình chệch khỏi mức tăng thông thường của nó và thay vào đó cập nhật giá trị của nó thành địa chỉ nhảy. Do đó, pc\_mux được sử dụng để chọn địa chỉ lệnh tăng (PCPlus4F) hoặc địa chỉ nhảy (JumpTargetE). pc\_mux được điều khiển bởi tín hiệu PCSrcE. Nếu tín hiệu PCSrcE cao, pc\_mux sẽ JumpTargetE ở chu kỳ xung nhịp tiếp theo, nếu không PCPlus4F sẽ được chọn.

module *pc\_mux* (

    input [31:0] PCPlus4F, JumpTargetE,

    input PCSrcE,

    output [31:0] PCNextF

);

    assign PCNextF = PCSrcE ? JumpTargetE : PCPlus4F;

endmodule

Đầu ra tương ứng của pc\_mux được trình bày trong Bảng bên dưới.

|  |  |
| --- | --- |
| PCSrcE | Output |
| 0 | PCPlus4F |
| 1 | JumpTargetE |

3.2.9 Thanh ghi file (regfile)

Tệp thanh ghi (regfile) trong CPU đóng vai trò quan trọng trong việc lưu trữ và thao tác dữ liệu trong quá trình thực thi chương trình. Tệp này đóng vai trò là thành phần lưu trữ tốc độ cao chứa một tập hợp các thanh ghi, mỗi thanh ghi có khả năng lưu trữ một chiều rộng cố định dữ liệu. Thông thường, tệp này có 4 đầu vào là RegWrite, Instruction[19:15], Instruction[24:20] và WriteData. Tín hiệu điều khiển RegWrite được sử dụng để điều khiển hoạt động ghi trên regfile. Khi RegWrite ở mức cao, dữ liệu từ WriteData sẽ được ghi vào regfile. Sau đó, Instruction[19:15] và Instruction[24:20] đóng vai trò là đầu vào từ thanh ghi đường ống (IF/ID) và đầu ra cho thanh ghi đường ống (ID/IEx) cho ALU trong giai đoạn thực thi.

module *regfile*(

    input clk,

    input we3,

    input [4:0] a1, a2, a3,

    input [31:0] wd3,

    output [31:0] rd1, rd2

);

    reg [31:0] rf[31:0];

    assign rd1 = (a1 != 0) ? rf[a1] : 0;

    assign rd2 = (a2 != 0) ? rf[a2] : 0;

    always @(negedge clk) if (we3) rf[a3] <= wd3;

endmodule

3.2.10 Nhân phát hiện xung dột (hazardunit)

Hazard Unit là một thành phần trong kiến ​​trúc của CPU có trách nhiệm phát hiện và xử lý các mối nguy hiểm có thể xảy ra trong quá trình thực hiện các lệnh. Các mối nguy hiểm đề cập đến các tình huống mà việc thực hiện tuần tự các lệnh có thể dẫn đến hành vi không chính xác hoặc không mong muốn do sự phụ thuộc hoặc xung đột giữa các lệnh. Hazard Unit phát hiện ra các mối nguy hiểm này và thực hiện các hành động thích hợp để giảm thiểu tác động của chúng. Với điều này, các mối phụ thuộc của dữ liệu ghi bắt đầu từ một thanh ghi đường ống, thay vì chờ giai đoạn WB ghi tệp thanh ghi. Do đó, dữ liệu cần thiết tồn tại kịp thời cho các lệnh sau, với các thanh ghi đường ống giữ dữ liệu để chuyển tiếp.

Có 2 giải pháp để xử lý nguy hiểm là *forwarding* và *stalling.*

3.2.10.1 Forwarding

Forward là một kỹ thuật được sử dụng bởi đơn vị nguy hiểm để xử lý các phụ thuộc dữ liệu giữa các lệnh và giảm thiểu các mối nguy hiểm. Nó cho phép CPU chuyển tiếp dữ liệu từ lệnh sản xuất đến lệnh tiêu dùng, bỏ qua các giai đoạn đường ống trung gian và tránh nhu cầu dừng lại hoặc bong bóng.

Có 2 cặp điều kiện nguy hiểm.

1. (a). EX/MEM.RegisterRd = ID/EX.RegisterRs

(b). EX/MEM.RegisterRd = ID/EX.RegisterRt

1. (a). MEM/WB.RegisterRd = ID/EX.RegisterRs

(b). MEM/WB.RegisterRd = ID/EX.RegisterRt

Những điều kiện nguy hiểm này có thể được xử lý bằng 2 tín hiệu điều khiển chuyển tiếp là ForwardA và ForwardB. ForwardA được sử dụng để điều khiển forwardMuxA và ForwardB được sử dụng để điều khiển forwardMuxB.

3.2.10.1.1 Forward Multiplexer A (ForwardMuxA)

Forward Multiplexer A lấy đầu vào của RD1, ALUResultsM và ResultW. RD1 được gọi là giá trị thanh ghi. ALUResultsM được gọi là kết quả alu trong giai đoạn bộ nhớ trong khi ResultW được gọi là kết quả trong giai đoạn ghi lại. Cả ALUResultsM và ResultW đều là giá trị được chuyển tiếp. Tín hiệu điều khiển ForwardA được sử dụng để chọn giá trị tệp thanh ghi hoặc giá trị được chuyển tiếp.

CHƯƠNG 4: MÔ PHỎNG VÀ KÊ