TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN ĐIỆN TỬ - VIỄN THÔNG

BÁO CÁO KIẾN TRÚC MÁY TÍNH

Đề tài: Thiết kế bộ xử lý RISC-V Pipeline sử dụng ngôn ngữ System Verilog

Sinh viên thực hiện:

Tên sinh viên	**MSSV**	**Mã lớp**
Lê Quang Hưng	20182562	129277

Giảng viên hướng dẫn: Tạ Thị Kim Huệ

Hà Nội - 2021

MỤC LỤC

DANH MỤC HÌNH ẢNH

DANH MỤC BẢNG

LỜI NÓI ĐẦU

Mã nguồn mở đang dần trở thành một phần quan trọng của thế giới IT khi mà nó góp mặt trong khoảng 96% phần mềm thương mại. Tương tự với phần cứng, vi xử lý mã nguồn mở RISC-V đang dần được quan tâm và hứa hẹn sẽ mang tới thay đổi lớn về bối cảnh của ngành điện toán. Chương trình học môn Kiến trúc máy tính (ET4041), RISC-V Processor Design là một phần quan trọng giúp hiểu rõ về quá trình hoạt động khi thực hiện các lệnh. Trong báo cáo này, chúng em triển khai kiến trúc RISC-V áp dụng kĩ thuật pipeline nhằm tăng tốc độ xử lí, bên cạnh đó, thiết kế đã xữ lí được toàn bộ các hazard có thể xảy ra trong quá trình xử lí thực hiện lệnh của CPU bao gồm data hazard, mem hazard và control hazard. Thiết kế được tiến hành triển khai bằng ngôn ngữ mô tả phần cứng SystemVerilog và mô phỏng kiểm thử trên phần mềm ModelSim. Cho ra kết quả hoạt động đúng với yêu cầu vào ra. Chúng em sẽ đi trình bày cụ thể những gì chúng em đã làm được thông qua 4 chương sau:

Chương 1: Giới thiệu

Chương 2: Đặc tả thông số kĩ thuật

Chương 3: Kiểm thử

Chương 4: Kết luận

Chúng em xin chân thành cảm ơn cô Tạ Thị Kim Huệ đã tận tâm hướng dẫn chúng em trong quá trình thực hiện đồ án cũng như hoàn hiện báo cáo này !

CHƯƠNG 1. GIỚI THIỆU (INTRODUCTION)

Chương này giới thiệu khái quát và các kiến thức cơ bản về RISC-V 32I.

1.1 Giới thiệu chung

RISC-V là một kiến trúc tập lệnh tiêu chuẩn mở (ISA) (*Instruction Set Architecture*) dựa trên nguyên tắc *Reduced Instruction Set Computer* (RISC) đã được thiết lập. RISC – V ISA được cung cấp theo *Open Source Licenses* không yêu cầu phí sử dụng.

Các tính năng đáng chú ý của RISC – V:

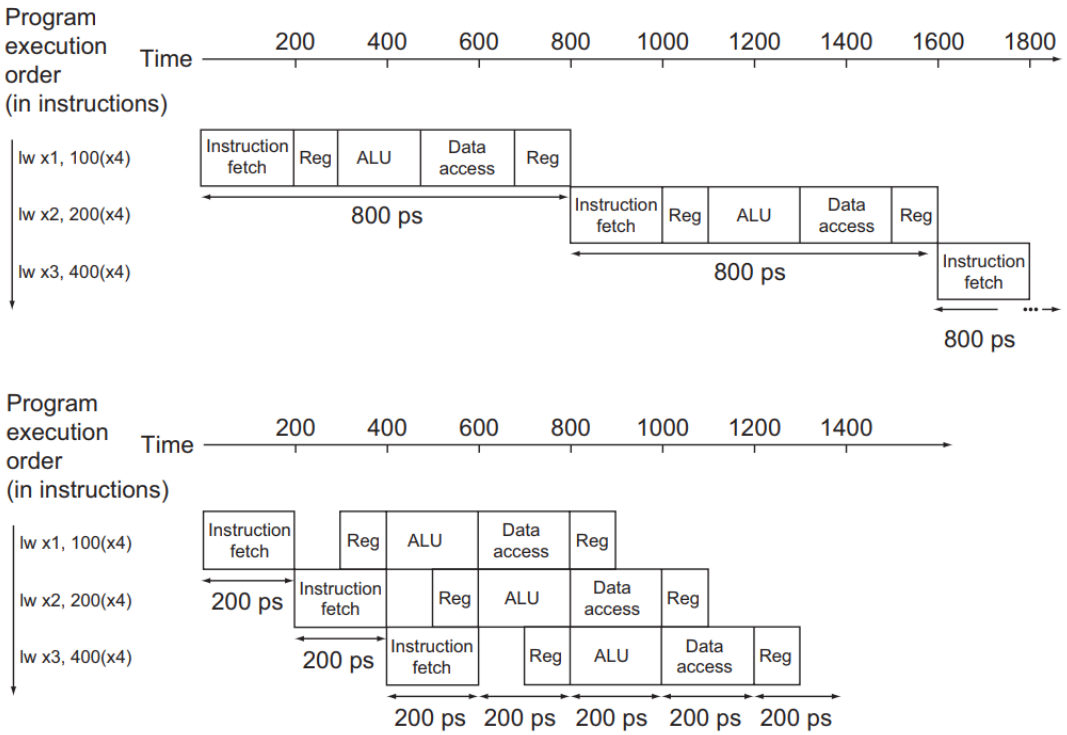
- Kiến trúc *load – store*
- Các mẫu bit để đơn giản hóa bộ MUX trong CPU
- Dấu phẩy động IEEE754
- Thiết kế trung lập về mặt kiến trúc và đặt *most-significant* bits tại một vị trí cố định để tăng tốc độ *sign extension*
- Tập lệnh được thiết kế cho nhiều mục đích sử dụng. Tập lệnh cơ sở có độ dài cố định gồm các lệnh 32 bit được căn chỉnh tự nhiên

1.2 Pipeline

Pipelining là một kĩ thuật triển khai trong đó nhiều lệnh được chồng lên nhau trong quá trình thực thi. Ngày nay, pipeline gần như phổ biến và có nhiều ứng dụng. Quá trình thực hiện một lệnh trong RISC – V cổ điển bao gồm 5 bước:

- Fetch the instruction from memory
- Read register and decode the instruction
- Excute the operation or calculate an address
- Access an operand in data memory (If necessary)
- Write result into a register (if necessary)

Hình 1.1 mô tả so sánh giữa single – cycle (nonpipelined) và pipeline.



Hình 1.1 Single cycle versus pipelined (1)

	Single Cycle	Pipelined
Timing	t_{step} = 100 ... 200 ps	t_{cycle} = 200 ps
	Register access only 100 ps	All cycles same length
Instruction time, $t_{instruction}$	= t_{cycle} = 800 ps	1000 ps
CPI (Cycles Per Instruction)	~1 (ideal)	~1 (ideal), <1 (actual)
Clock rate, f_s	1/800 ps = 1.25 GHz	1/200 ps = 5 GHz
Relative speed	1 x	4 x

Hình 1.2 Single cycle versus pipelined (2)

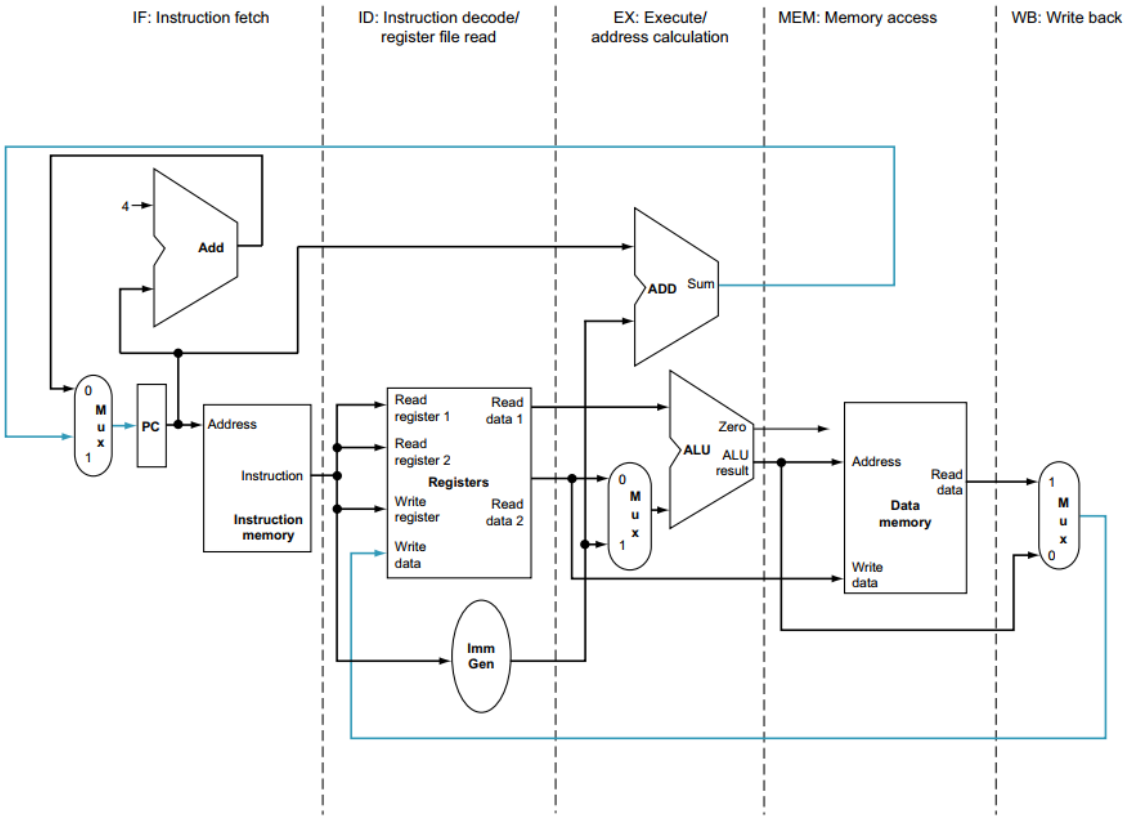
Dựa vào Hình 1.1, ta sử dụng chung một kiến trúc phần cứng, nhận thấy thời gian trung bình giữa các lệnh nhanh gấp 4 lần từ 800 ps xuống 200ps. Tuy nhiên, thời gian của giai đoạn pipeline cũng bị giới hạn bởi tài nguyên chậm nhất như hoạt động ALU hoặc truy cập vào bộ nhớ. Kết quả cụ thể được mô tả trên Hình 1.2.

Pipelining cải thiện hiệu suất bằng cách tăng thông lượng lệnh, ngược lại với việc giảm thời gian thực thi của một lệnh riêng lẻ, nhưng thông lượng lệnh là thước đo quan trọng vì các chương trình thực thực hiện hàng tỷ lệnh.

1.3 Pipelined Datapath

Hình 1.3 dưới đây mô tả single – cycle datapath với pipeline được định nghĩa. Việc chia lệnh thành năm giai đoạn có nghĩa là một pipeline năm giai đoạn, do đó có nghĩa là tối đa năm lệnh sẽ được thực thi trong bất kỳ chu kỳ đồng hồ đơn nào (single clock cycle). Do đó, phải tách đường dữ liệu thành năm phần, với mỗi phần được đặt tên tương ứng với một giai đoạn thực thi lệnh:

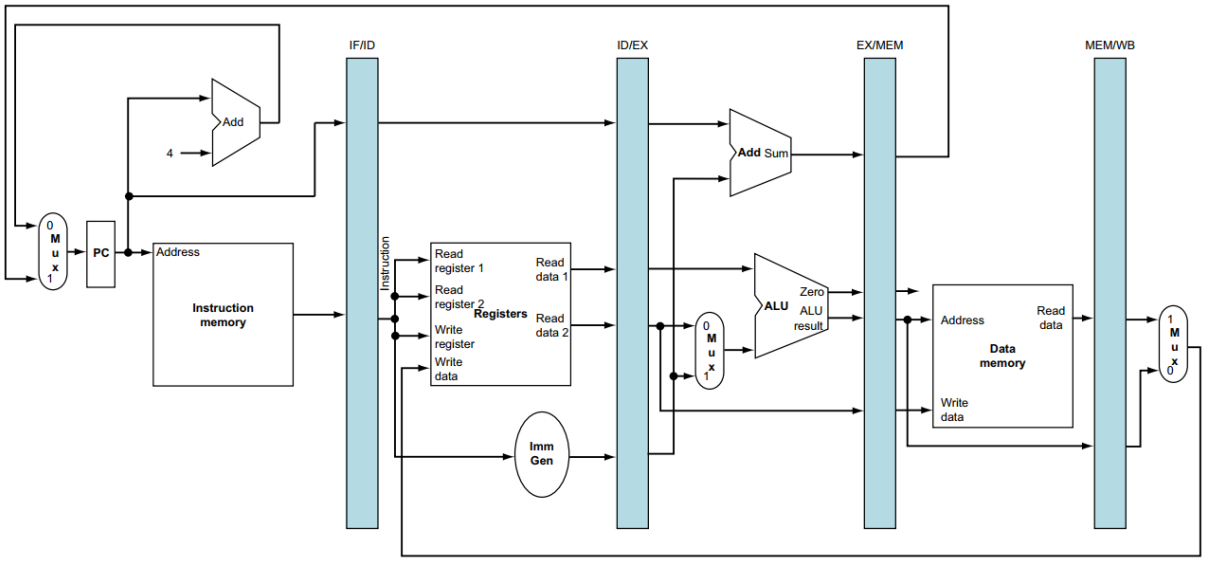
1. IF: Instruction Fetch (nạp câu lệnh)
2. ID: Instruction Decode/register file read (giải mã lệnh)
3. EX: Excute/address calculation (thực thi/tính toán địa chỉ)
4. MEM: Memory access (truy cập bộ nhớ)
5. WB: Write back (ghi lại)



Hình 1.3 Single-cycle datapath

Tùy thuộc vào từng loại lệnh mà các giai đoạn có được thực thi hay không. Tại giai đoạn Write back, kết quả được lưu trữ lại thanh ghi ở phần giữa của datapath. Và việc lựa chọn giá trị tiếp theo của PC, chọn giữa giá trị PC được tăng lên 4 và địa chỉ rẽ nhánh ở giai đoạn MEM.

Hình 1.4 mô tả pipeline datapath với thanh ghi pipeline. Các thanh ghi pipeline được đặt tên theo hai giai đoạn bị ngăn cách bởi thanh ghi đó. Ví dụ, thanh ghi pipeline IF/ID ngăn cách hai giai đoạn IF và IF của datapath. Và không có thanh ghi tại giai đoạn cuối cùng WB. Các thanh ghi pipeline phải đủ lớn để lưu trữ các giá trị tương ứng đi qua chúng. Ví dụ, thanh ghi IF/ID phải có độ dài 96 bit vì chúng chứa 32 bit địa chỉ từ Instruction memory và 64 bit địa chỉ PC.



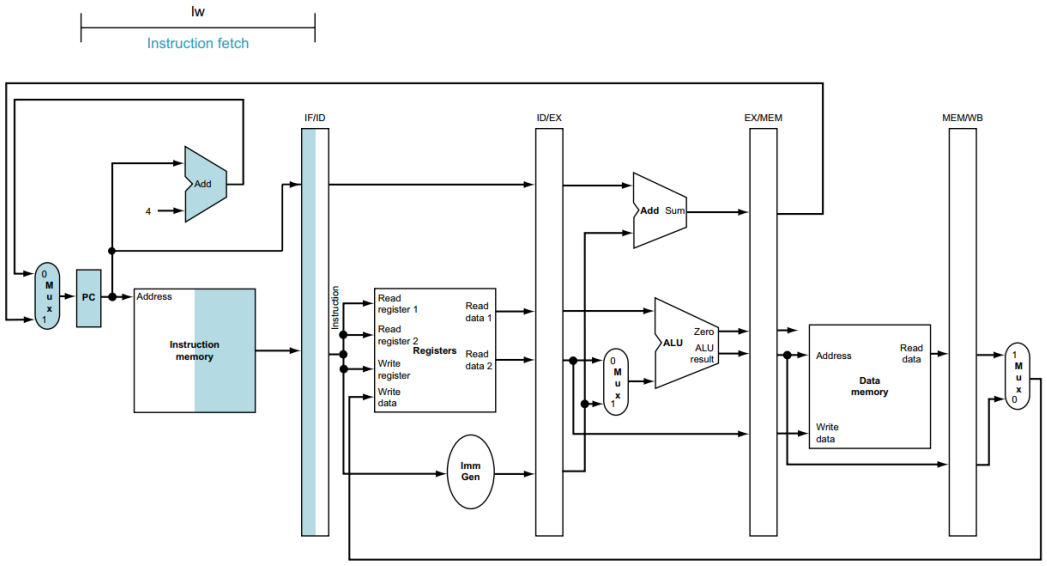
Hình 1.4 Pipeline Datapath

Dưới đây là năm giai đoạn của lệnh *lw:*

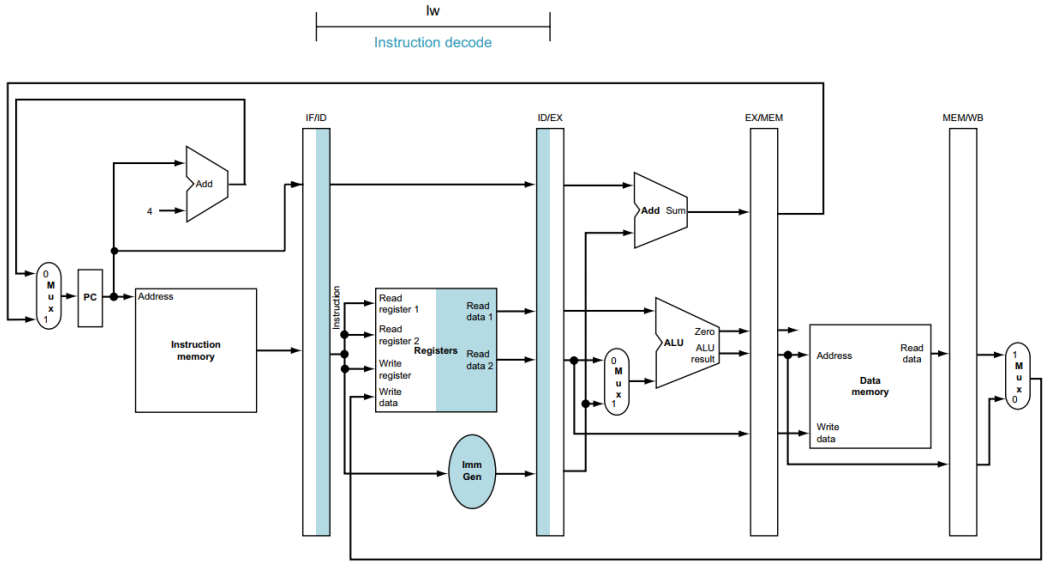
1. *Instruction Fetch*: Hình 1.5 cho thấy lệnh được đọc từ bộ nhớ bằng địa chỉ trong PC và sau đó được đặt trong thanh ghi pipeline IF/ID. Địa chỉ PC được tăng thêm 4 và sau đó được ghi lại vào PC để sẵn sàng cho chu kỳ xung nhịp tiếp theo. PC này cũng được lưu trong thanh ghi pipeline IF/ID trong trường hợp sau này cần thiết để hướng dẫn, chẳng hạn như beq. Máy tính không thể biết loại lệnh nào đang được tìm nạp, vì vậy nó phải chuẩn bị cho bất kỳ lệnh nào, chuyển thông tin có thể cần thiết xuống pipeline.
2. *Instruction Decode và register file read*: Hình 1.6 cho thấy phần lệnh của thanh ghi pipeline IF/ID cung cấp trường imediate được sign – extend đến 64 bit, và các register numbers để đọc hai thanh ghi. Tất cả ba giá trị được lưu trữ trong thanh ghi pipeline ID/EX, cùng với địa chỉ PC. Và chuyển mọi thứ có thể cần thiết bằng bất kỳ lệnh nào trong chu kỳ đồng hồ sau đó.
3. *Execute hoặc Address calculation*: Hình 1.7 cho thấy rằng lệnh *load* đọc nội dung của một thanh ghi và sign - extend imediate từ thanh ghi pipeline ID/EX

và thêm chúng bằng cách sử dụng ALU. Tổng đó được đặt trong thanh ghi pipeline EX/MEM.

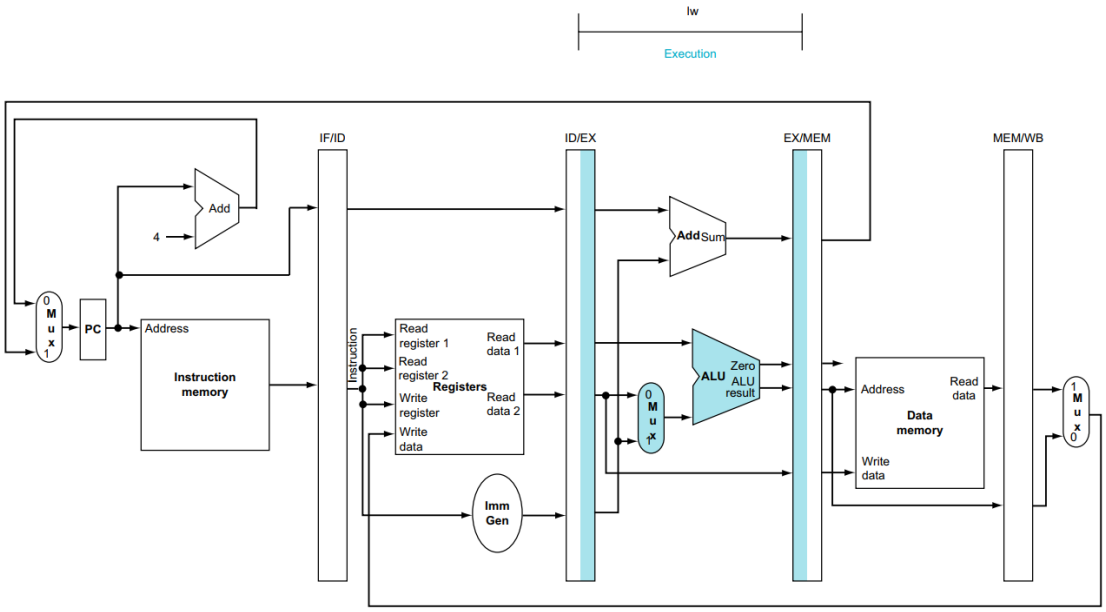
4. *Memory access*: Hình 1.8 cho thấy lệnh *load* đọc nội dung bộ nhớ sử dụng địa chỉ từ thanh ghi pipeline EX/MEM và tải dữ liệu vào thanh ghi MEM/WB.
5. *Write back*: Đọc dữ liệu từ thanh ghi pipeline MEM/WB và ghi dữ liệu thanh ghi.



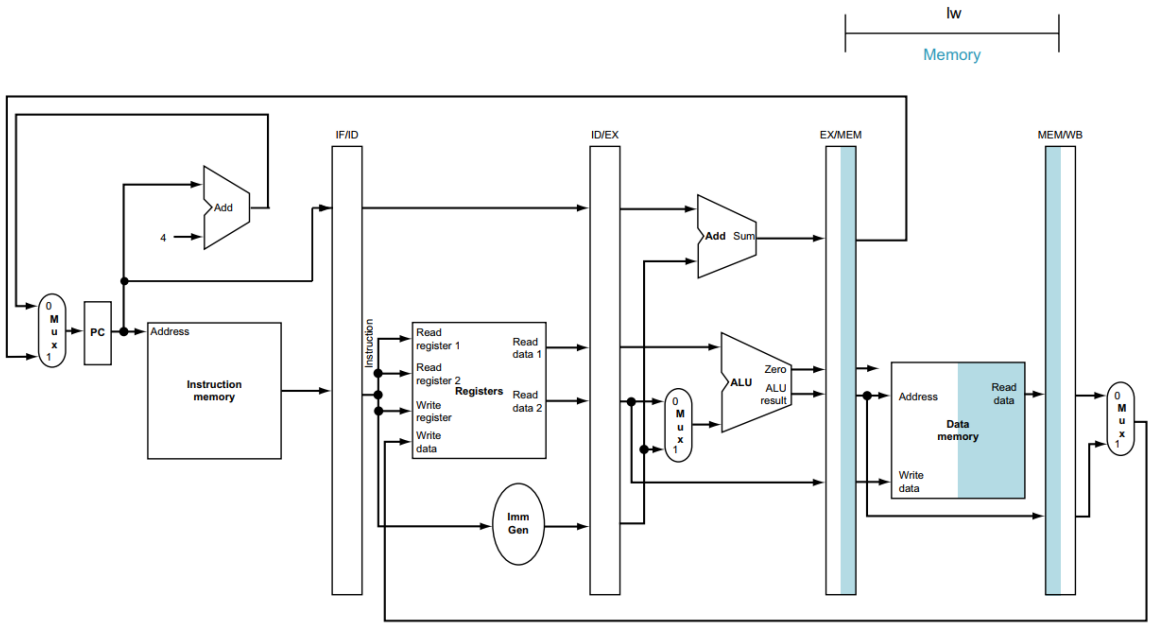
Hình 1.5 Giai đoạn đầu tiên IF của lệnh lw



Hình 1.6 Giai đoạn ID của lệnh lw



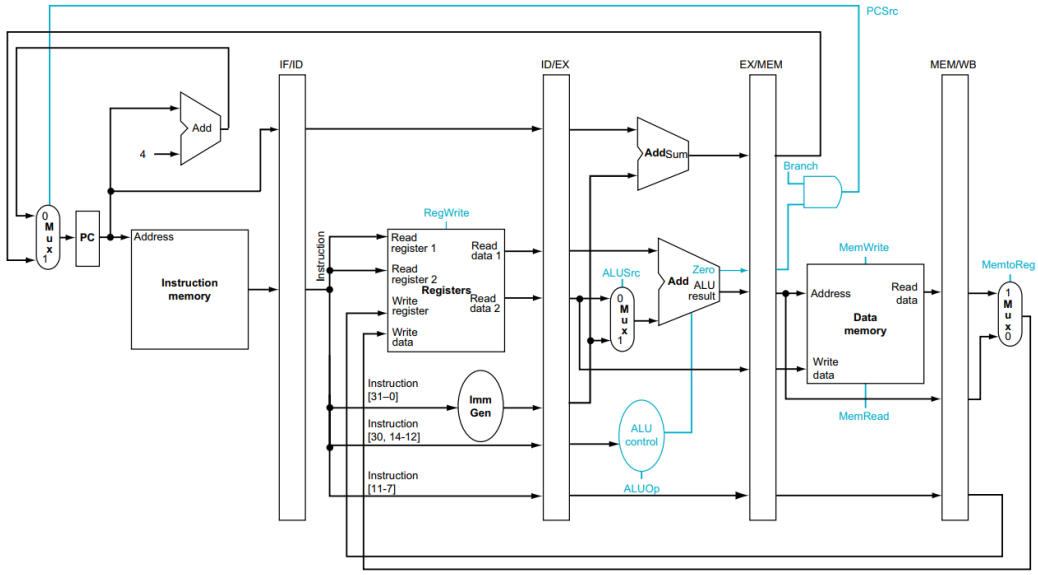
Hình 1.7 Giai đoạn EX của lệnh lw



Hình 1.8 Giai đoạn MEM của lw

1.4 Pipelined Control

Như trường hợp thực hiện chu kỳ đơn, giả định rằng PC được ghi trên mỗi chu kỳ đồng hồ, do đó không có tín hiệu ghi riêng cho PC. Theo cùng một đối số, không có tín hiệu ghi riêng biệt cho các thanh ghi pipeline (IF/ID, ID/EX, EX/MEM và MEM/WB) vì các thanh ghi pipeline cũng được ghi trong mỗi chu kỳ đồng hồ.



Hình 1.9 Pipelined datapath với các đường tín hiệu điều khiển

Để chỉ định điều khiển cho đường ống, chúng ta chỉ cần đặt các giá trị điều khiển trong mỗi giai đoạn pipeline. Bởi vì mỗi đường điều khiển được liên kết với một thành phần chỉ hoạt động trong một giai đoạn pipeline duy nhất, chúng ta có thể chia các đường điều khiển thành năm nhóm theo giai đoạn đường ống:

1. *Instruction fetch:* Các tín hiệu điều khiển để đọc bộ nhớ lệnh và ghi PC luôn được xác nhận, vì vậy không có gì đặc biệt để điều khiển trong giai đoạn đường ống này.
2. *Instruction decode/register file read:* Hai thanh ghi nguồn luôn ở cùng một vị trí trong các định dạng lệnh RISC-V, vì vậy không có gì đặc biệt để kiểm soát trong giai đoạn đường ống này.
3. *Execute/address calculation:* Các tín hiệu được đặt là ALUOp và ALUSrc (xem Hình 4.49 và 4.50). Các tín hiệu chọn hoạt động ALU và Read data 2 hoặc sign – extended imediate làm đầu vào cho ALU.
4. *Memory access:* Các dòng điều khiển được thiết lập trong giai đoạn này là Branch, MemRead và MemWrite. Các lệnh rẽ nhánh nếu bằng nhau, tải và lưu trữ sẽ đặt các tín hiệu này tương ứng.
5. *Write back*: Hai dòng điều khiển là MemtoReg, quyết định giữa việc gửi kết quả ALU hoặc giá trị bộ nhớ vào tệp thanh ghi và RegWrite, ghi giá trị đã chọn.

Instruction	ALUOp	operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
lw	00	load word	XXXXXXX	XXX	add	0010
sw	00	store word	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

Hình 1.10 ALU control bits

Instruction	Execution/address calculation stage control lines		Memory access stage control lines			Write-back stage control lines	
	ALUOp	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	10	0	0	0	0	1	0
lw	00	1	0	1	0	1	1
sw	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

Hình 1.11 Đường điều khiển

1.5 Hazard trong pipeline

Một vấn đề xảy ra với pipeline là hiện tượng hazard. Hazard là một tình huống ngăn cản việc bắt đầu lệnh tiếp theo trong chu kì tiếp theo. Có ba loại hazard:

- Structural hazard: Tài nguyên yêu cầu đang bận (ví dụ: cần trong nhiều giai đoạn)
- Data hazard: Sự phụ thuộc dữ liệu giữa các câu lệnh, cần đợi lệnh trước đó hoàn thành việc đọc ghi dữ liệu của nó
- Control hazard: Luồng thực hiện phụ thuộc vào lệnh trước đó

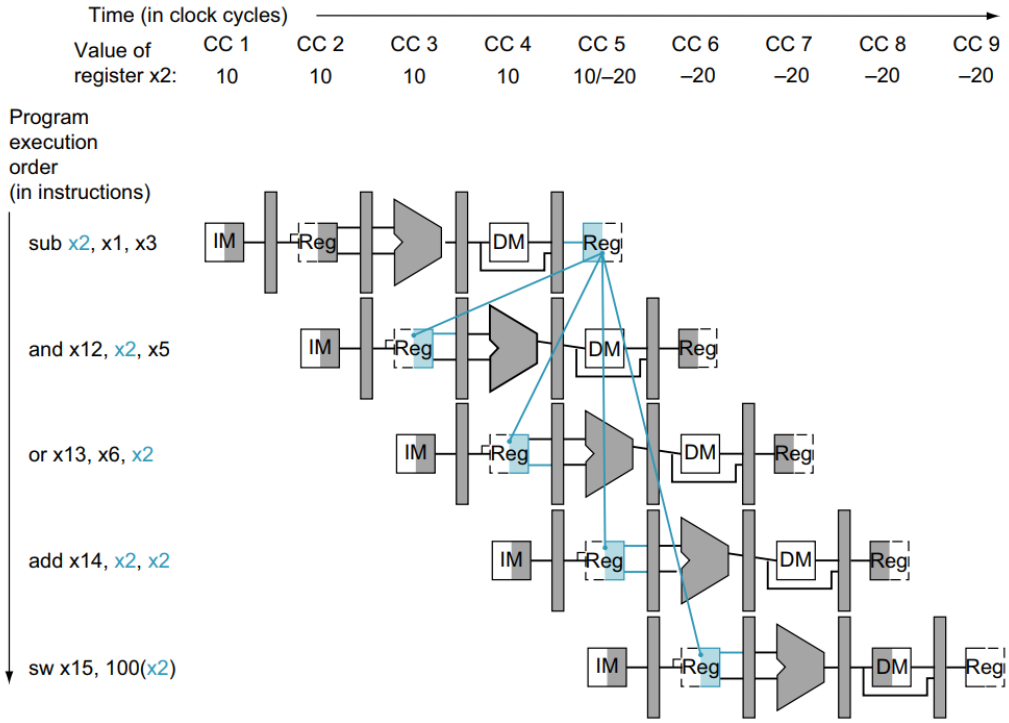
1.5.1 Structural Hazard

Đầu tiên là structural hazard, nó có nghĩa là hai hay nhiều lệnh trong pipeline cùng truy cập, yêu cầu một phần tài nguyên vật lý. Khi một lệnh được lập kế hoạch không thể thực thi trong chu kỳ đồng hồ thích hợp vì phần cứng không hỗ trợ sự kết hợp của các lệnh được thiết lập để thực thi. Structural hazard là xung đột trong việc sử dụng tài nguyên. Trong pipeline RISC-V sử dụng một bộ nhớ duy nhất, lệnh load/store yêu cầu truy cập dữ liệu, và nếu không có phân chia bộ nhớ, việc tìm nạp lệnh sẽ phải dừng lại

trong chu kì đó, các hoạt động khác trong pipeline phải dừng lại chờ đợi. Do đó, pipelined datapath yêu cầu bộ nhớ lệnh/dữ liệu riêng biệt.

1.5.2 Data Hazard

Data hazard thể hiện sự phụ thuộc dữ liệu giữa các câu lệnh trong pipeline, câu lệnh trước cần hoàn thành việc đọc ghi dữ liệu của nó. Hình 1.12 dưới đây thể hiện sử phụ thuộc giữa các câu lệnh trong pipeline.

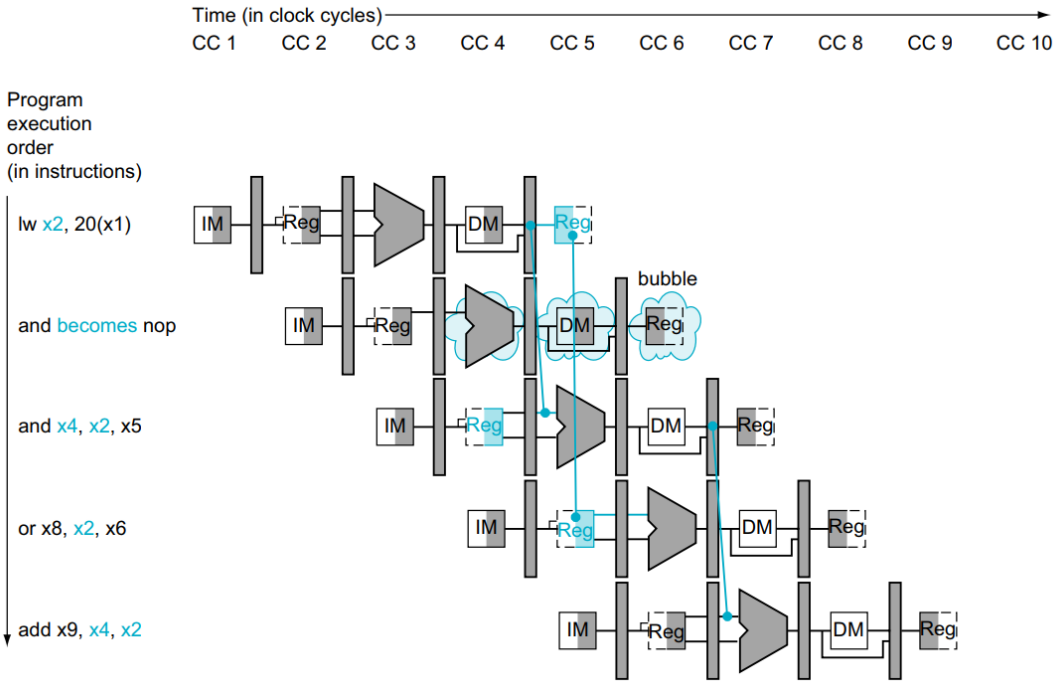


Hình 1.12 Sự phụ thuộc dữ liệu giữa các câu lệnh trong pipeline

Lệnh đầu tiên ghi dữ liệu vào x2, các lệnh sau nó đều đọc dữ liệu từ x2 để thực hiện lệnh. Tuy nhiên, việc ghi dữ liệu vào x2 được thực hiện ở chu kì 5 (CC5), các chu kì trước đó hoàn toàn chưa có giá trị của x2 sau khi được tính toàn và lưu vào trong x2.

Hướng giải quyết cho vấn đề data hazard là stall và forwarding. Stall là chuyển lệnh thành "nops", các giai đoạn pipeline không thực thi gì và các giai đoạn tiếp theo thực thi bình thường. Forwarding dựa vào việc khi thực hiện lệnh, giá trị của kết quả có từ các giai đoạn trước của pipeline trước khi được ghi vào memory hoặc register. Nếu lệnh trong giai đoạn ID bị stall, thì lệnh trong giai đoạn IF cũng phải bị stall; nếu không, sẽ mất instruction đã được nạp. Việc ngăn cản hai lệnh này thực hiện tiến trình được thực hiện đơn giản bằng cách ngăn thanh ghi PC và thanh ghi pipeline IF/ID thay đổi. Miễn là các thanh ghi này được giữ nguyên, lệnh trong giai đoạn IF sẽ tiếp tục được đọc bằng

cách sử dụng cùng một PC và các thanh ghi trong giai đoạn ID sẽ tiếp tục được đọc bằng cách sử dụng các trường lệnh tương tự trong thanh ghi pipeline IF/ID. Nửa sau của pipeline bắt đầu với giai đoạn EX phải đang làm gì đó; những gì nó đang làm là thực hiện các lệnh không có tác dụng thực thi: nops. Cụ thể phương pháp stall và forwarding được mô tả trên Hình 1.13 dưới đây.

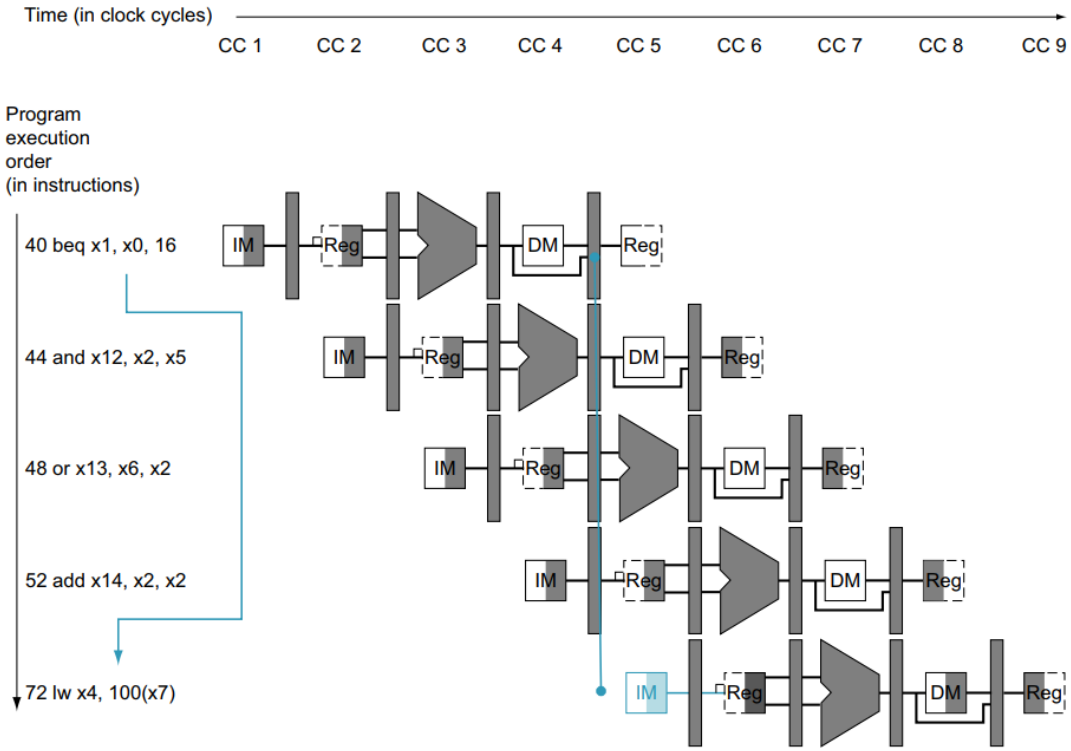


Hình 1.13 Stall và forwarding trong pipeline

Một bubble được chèn vào bắt đầu từ chu kỳ đồng hồ 4, bằng cách thay đổi lệnh *and* thành *nop*. Lưu ý rằng lệnh và thực sự được tìm nạp và giải mã trong chu kỳ đồng hồ 2 và 3, nhưng giai đoạn EX của nó bị trì hoãn cho đến chu kỳ đồng hồ 5 (so với vị trí không được chặn trong chu kỳ đồng hồ 4). Tương tự như vậy, lệnh hoặc được tìm nạp trong chu kỳ đồng hồ 3, nhưng giai đoạn ID của nó bị trì hoãn cho đến chu kỳ đồng hồ 5 (so với vị trí của chu kỳ đồng hồ 4 không được chặn). Sau khi chèn bubble, tất cả các phần phụ thuộc sẽ tiếp tục theo thời gian và không có thêm hazard nào xảy ra.

1.5.3 Control Hazard

Control hazard do lệnh rẽ nhánh gây ra, luồng thực hiện phụ thuộc vào lệnh trước đó. Nếu lệnh rẽ nhánh không được thực hiện, các instruction fetch sau nó được thực hiện chính xác. Nếu lệnh rẽ nhánh được thực hiện, phải loại bỏ các hướng dẫn không chính xác ra khỏi pipeline bằng cách chuyển chúng về NOPs. Hình 1.14 dưới đây mô tả vấn đề control hazard.



Hình 1.14 Control hazard trong pipeline

Một cách để cải thiện hiệu suất của nhánh có điều kiện là giảm chu kì clk phải đánh đổi khi lệnh rẽ nhánh được thực hiện. Giả sử PC tiếp theo cho một nhánh được chọn trong giai đoạn MEM, nhưng nếu chúng ta di chuyển việc thực thi nhánh có điều kiện sớm hơn trong pipeline, thì cần phải xóa ít lệnh hơn do vậy số chu kì clk phải đánh đổi khi lệnh rẽ nhánh được thực hiện. Chuyển việc quyết định có rẽ nhánh lên giai đoạn sớm hơn (thay vi trong giai đoạn MEM) đòi hỏi hai hành động xảy ra trước đó: tính toán địa chỉ mục tiêu nhánh và đánh giá quyết định nhánh. Việc tính toán địa chỉ nhánh sớm hơn khá dễ dàng triển khai do đã có giá trị PC và trường ngay lập tức trong thanh ghi pipeline IF/ID, vì vậy chúng ta chỉ cần di chuyển bộ cộng nhánh từ giai đoạn EX sang giai đoạn ID; tất nhiên, tính toán địa chỉ cho các mục tiêu nhánh sẽ được thực hiện cho tất cả các lệnh, nhưng chỉ được sử dụng khi cần thiết.

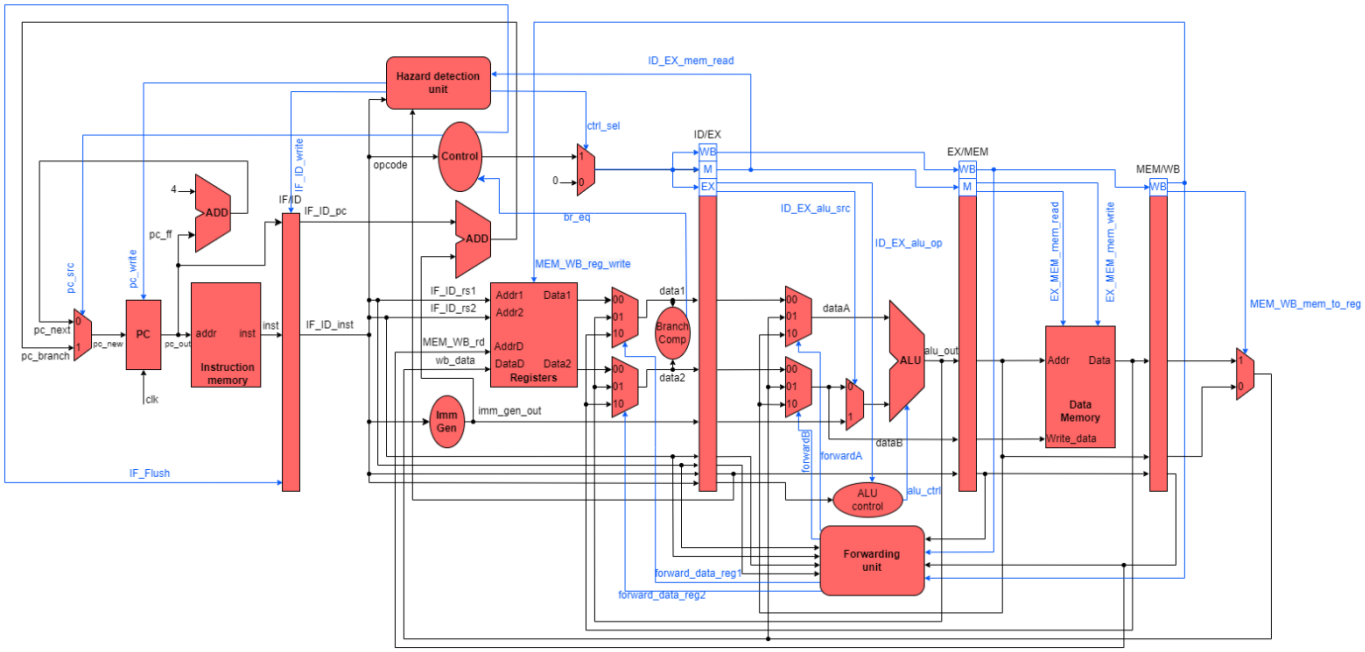
Phần khó hơn là quyết định nhánh nào được chọn. Đối với nhánh nếu bằng nhau, chúng ta sẽ so sánh hai lần đọc thanh ghi trong giai đoạn ID để xem chúng có bằng nhau hay không. Sự bằng nhau có thể được kiểm tra bằng cách XOR các vị trí bit riêng lẻ của hai thanh ghi và OR kết quả XOR (một đầu ra bằng không của cổng OR có nghĩa là hai thanh ghi bằng nhau), và có thể sử dụng "dự đoán nhánh" để đoán nhánh nào sẽ đi sớm hơn trong pipeline và chỉ flush pipeline nếu dự đoán nhánh không chính xác.

CHƯƠNG 2. ĐẶC TẢ THÔNG SỐ KĨ THUẬT (SPECIFICATION)

Chương này mô tả thông số kĩ thuật và kiến trúc chi tiết của từng phần có trong kiến trúc RISC-V được triển khai.

2.1 RISC-V pipeline architecture

2.1.1 Architecture



Hình 2.1 riscv_pipeline architecture

Hình 2.1 mô tả kiến trúc hoàn chỉnh cho cpu **riscv32i** hỗ trợ tập lệnh đơn giản gồm đầy đủ 4 loại lệnh R-type, I-type, S-type, B-type, được áp dụng kĩ thuật pipeline và xử lí các loại hazard (data, mem, control) bằng sự kết hợp của **Hazard detection unit** và **Forwarding unit**.

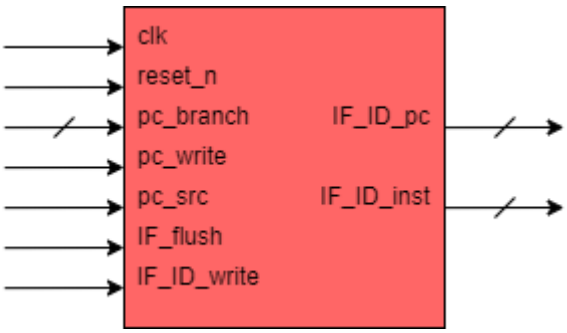
2.1.2 Interface signals

Bảng 2.1 riscv_pipeline port description

Signal name	Width	Input/Output	Description
clk	1	Input	Tín hiệu xung đồng hồ
reset_n	1	Input	Tín hiệu reset tích cực mức thấp

CPU sẽ nhận xung **clk** và tín hiệu **reset** để hoạt động, chương trình cần CPU thực hiện sẽ được nạp sẵn vào bộ nhớ **Instruction memory**.

2.2 Module Intruction Fetch (instruction_fetch)



Hình 2.2 instruction_fetch block diagram

2.2.1 Interface signals

Bảng 2.2 instruction_decode port description

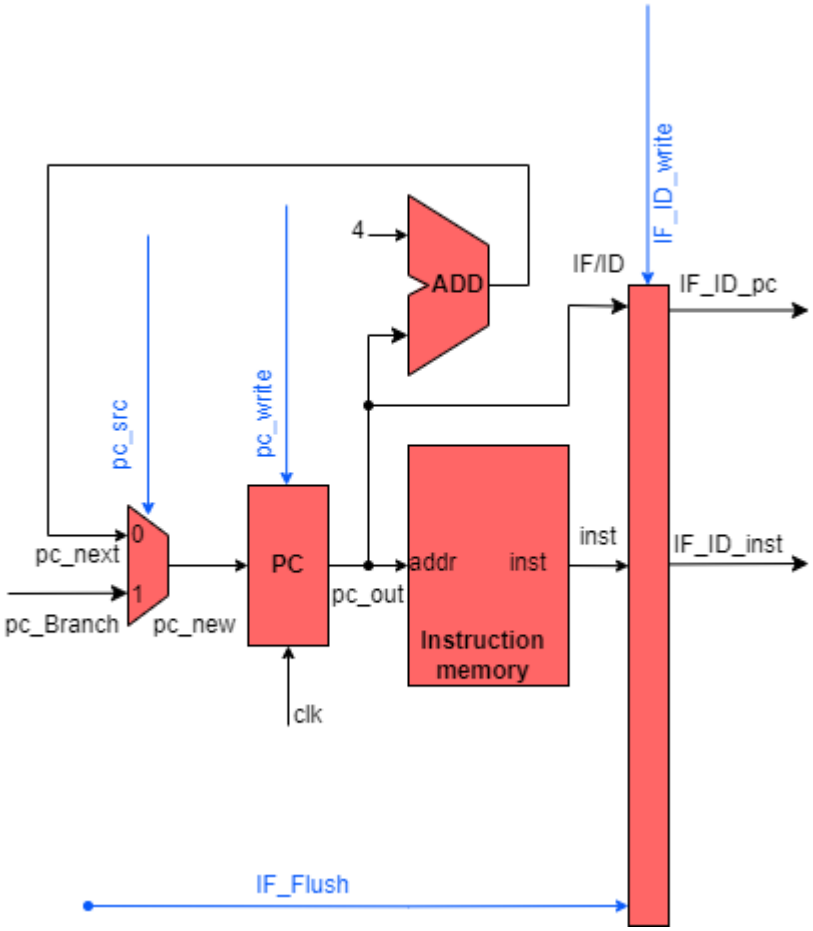
Signal name	Width	Input/Output	Description
clk	1	Input	Tín hiệu xung đồng hồ
reset_n	1	Input	Tín hiệu reset tích cực mức thấp
pc_branch	32	Input	Giá trị PC cần nhảy đến khi có lệnh rẽ nhánh
pc_write	1	Input	Cho phép thay đổi đầu ra PC
pc_src	1	Input	Cho phép chọn PC+4 hoặc PC_branch
IF_flush	1	Input	Cho phép xóa thanh ghi IF_ID
IF_ID_write	1	Input	Cho phép thanh ghi IF_ID ghi dữ liệu mới
IF_ID_pc	32	Output	Đầu ra thanh ghi cho biết giá trị PC
IF_ID_inst	32	Output	Đầu ra thanh ghi cho biết mã lệnh tương ứng với PC

2.2.2 Function description

Module instruction_fetch tìm nạp lệnh từ bộ *instruction memory*. Mã lệnh được đọc từ bộ nhớ bằng địa chỉ PC và được đặt trong thanh ghi pipeline IF/ID. Địa chỉ PC được tăng thêm 4 và sau đó được đưa vào bộ mux để chọn giữa PC+4 và PC cần nhảy đến. Tiếp theo PC được ghi lại để sẵn sàng cho chu kỳ xung nhịp tiếp theo.

Giá trị PC được lưu trong thanh ghi IF/ID trong trường hợp cần cho lệnh rẽ nhánh vì vậy nó phải chuẩn bị cho bất kỳ lệnh nào, chuyển thông tin có thể cần thiết xuống pipeline.

2.2.3 Instruction fetch architecture



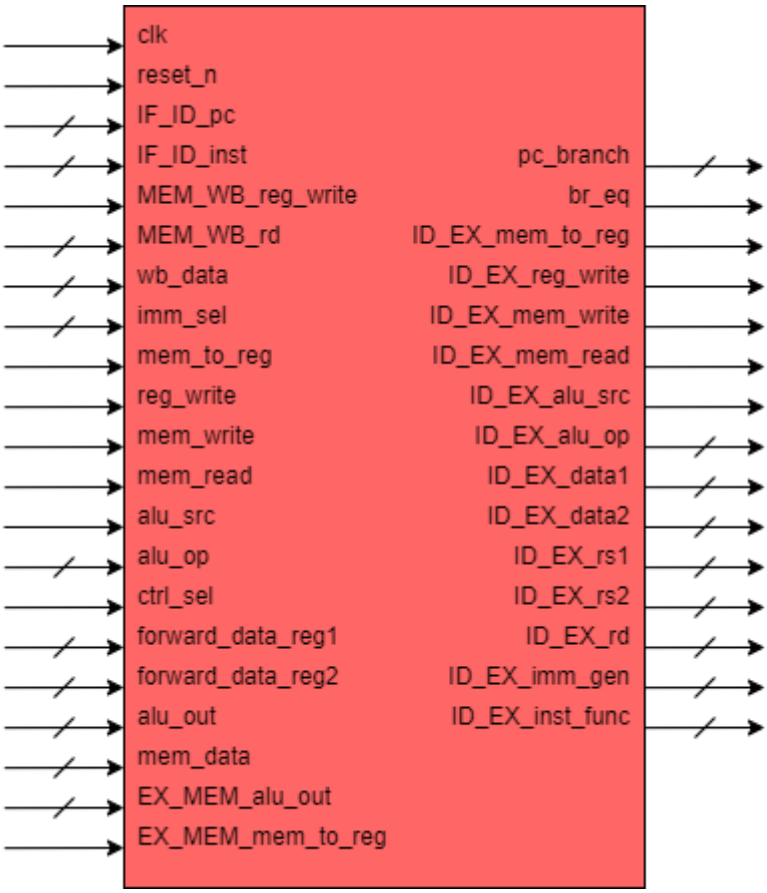
Hình 2.3 Instruction fetch architecture

Hình 2.3 mô tả các phần tử có trong kiến trúc của khối IF gồm:

- **Mux2to1** nhận tín hiệu **pc_src** từ khối **control** và quyết định xem giá trị **pc** tiếp theo đưa vào thanh ghi lưu **PC**
- Thanh ghi **PC** nhận tín hiệu **pc_write** từ khối **hazard detection unit** để nhằm mục đích stall cho CPU, khi **pc_write = 1** thì **pc** tiếp theo từ **mux2to1** sẽ được đưa vào thanh ghi **PC** nếu **pc_write = 0** thì thanh ghi **PC** sẽ giữ nguyên giá trị khí đó giá trị **pc** sẽ được giữ nguyên (CPU được stall)
- Bộ cộng nhằm mục đích cộng thêm 4 vào giá trị **pc** để chương trình nạp lệnh tiếp theo có địa chỉ **pc + 4**
- **Instruction memory** là bộ nhớ lưu chương trình gồm các lệnh assembly cần **CPU** thực hiện. **Instruction memory** nhận đầu vào là địa chỉ **pc** của lệnh và đưa ra lệnh (**inst**) tại địa chỉ **pc** tương ứng cần nạp vào **CPU** để thực hiện

- **IF/ID** là thanh ghi pipeline lưu kết quả **pc** và **inst** tương ứng đã được nạp trong giai đoạn nạp lệnh IF (instruction fetch). Tín hiệu điều khiển **IF_flush** nhận từ khối **control** để xoá flush thanh ghi khi có **control hazard** (khi lệnh được nạp ngay sau lệnh **beq** sai). Tín hiệu điều khiển **IF_ID_write** nhằm mục đích stall pipeline nhằm xử lí hazard, khi **IF_ID_write = 1** thì thanh ghi **IF/ID** sẽ nạp kết quả **pc** và **inst** của lệnh tiếp theo như bình thường, nếu **IF_ID_write = 0** thì thanh ghi sẽ giữ nguyên giá trị của lệnh đã được nạp trước đó (stall pipeline)

2.3 Module Instruction Decode (instruction_decode)



Hình 2.4 instruction_decode block diagram

2.3.1 Interface signals

Bảng 2.3 instruction_decode port description

Signal name	Width	Input/Output	Description
clk	1	Input	Tín hiệu xung đồng hồ
reset_n	1	Input	Tín hiệu reset tích cực mức thấp
IF_ID_pc	32	Input	Giá trị PC lưu trong thanh ghi IF_ID
IF_ID_inst	32	Input	Mã lệnh lưu trong thanh ghi IF_ID

MEM_WB_reg_write	1	Input	Tín hiệu cho phép ghi dữ liệu vào thanh ghi lưu trong thanh ghi MEM/WB
MEM_WB_rd	5	Input	Địa chỉ thanh ghi đích lưu trong thanh ghi MEM/WB
wb_data	32	Input	Dữ liệu được write back
imm_sel	3	Input	Chọn kiểu cho Immidiate Generate
mem_to_reg	1	Input	Tín hiệu cho phép write back
reg_write	1	Input	Cho phép ghi dữ liệu vào thanh ghi
mem_write	1	Input	Cho phép memory ghi dữ liệu
mem_read	1	Input	Cho phép memory đọc dữ liệu
alu_src	1	Input	Chọn chế độ địa chỉ trực tiếp
alu_op	2	Input	Opcode chọn phép toán
ctrl_sel	1	Input	Chọn đầu ra control
forward_data_reg1	2	Input	Forwarding khi có hazard tại thanh ghi rs1
forward_data_reg2	2	Input	Forwarding khi có hazard tại thanh ghi rs2
alu_out	32	Input	Kết quả alu được forward về
mem_data	32	Input	Dữ liệu memory được forward về
EX_MEM_alu_out	32	Input	Kết quả của alu trong thanh ghi EX/MEM được forward về
EX_MEM_mem_to_reg	1	Input	Tín hiệu điều khiển chọn dữ liệu từ memory về thanh ghi của thanh ghi EX/MEM
pc_branch	32	Output	Giá trị pc cần nhảy đến
br_eq	1	Output	Kết quả so sánh data1 và data2
ID_EX_mem_to_reg	1	Output	Tín hiệu điều khiển chọn dữ liệu từ memory về thanh ghi của thanh ghi ID/EX
ID_EX_reg_write	1	Output	Cho phép ghi dữ liệu vào thanh ghi lưu trong thanh ghi ID/EX
ID_EX_mem_write	1	Output	Cho phép memory ghi dữ liệu lưu trong thanh ghi ID/EX
ID_EX_mem_read	1	Output	Cho phép memory đọc dữ liệu lưu trong thanh ghi ID/EX
ID_EX_alu_src	1	Output	Chọn chế độ địa chỉ trực tiếp lưu trong thanh ghi ID/EX
ID_EX_alu_op	2	Output	Opcode chọn phép toán lưu trong thanh ghi ID/EX
ID_EX_data1	32	Output	Dữ liệu từ thanh ghi 1 lưu trong thanh ghi ID/EX
ID_EX_data2	32	Output	Dữ liệu từ thanh ghi 2 lưu trong thanh ghi ID/EX
ID_EX_rs1	5	Output	Địa chỉ của thanh ghi 1 lưu trong thanh ghi ID/EX
ID_EX_rs2	5	Output	Địa chỉ của thanh ghi 2 lưu trong thanh ghi ID/EX
ID_EX_rd	5	Output	Địa chỉ của thanh ghi đích lưu trong thanh ghi ID/EX
ID_EX_imm_gen	32	Output	Kết quả immidiate lưu trong thanh ghi ID/EX
ID_EX_inst_func	4	Output	Function chọn phép toán lưu trong thanh ghi ID/EX

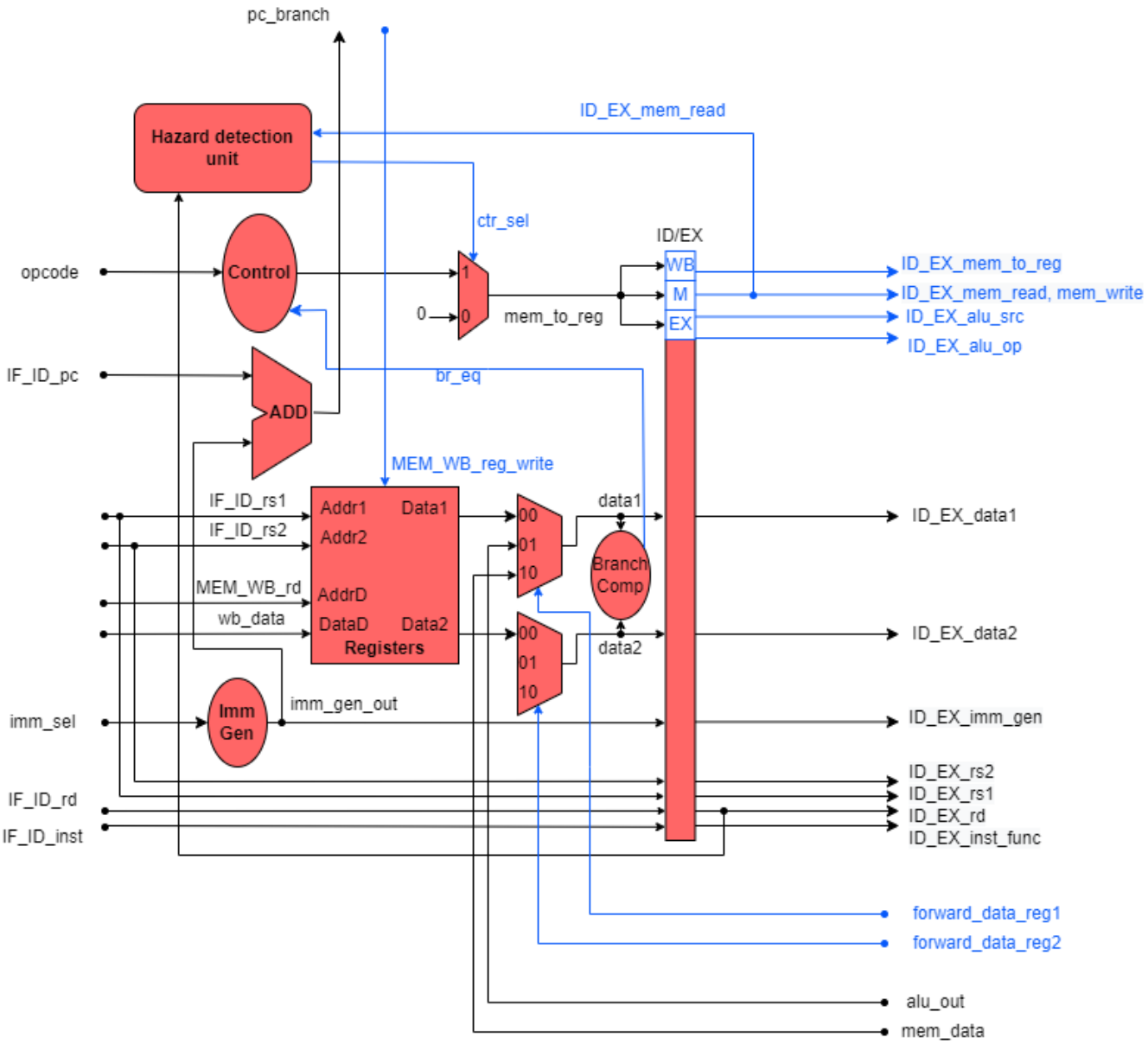
2.3.2 Function description

Khối Instruction Decode lấy đầu vào là instruction đã được nạp vào thanh ghi IF/ID và thực hiện:

- Đọc dữ liệu từ tệp thanh ghi dựa vào địa chỉ rs1, rs2, rd lấy từ instruction
- Gen ra 32bit immediate từ trường imm trong instruction

- Gửi trường opcode của instruction sang khối control để đưa ra các tín hiệu điều khiển đến các khối khác
- So sánh giá trị data của của 2 thanh ghi rs1, rs2 trong lệnh branch và gửi kết quả so sanh sang khối control. Nhận tín hiệu forward từ khối forwarding unit để quyết định xem sẽ so sánh giá trị thanh ghi hay giá trị được forward về trong trường hợp có hazard

2.3.3 Instruction Decode architecture



Hình 2.5 Instruction Decode architecture

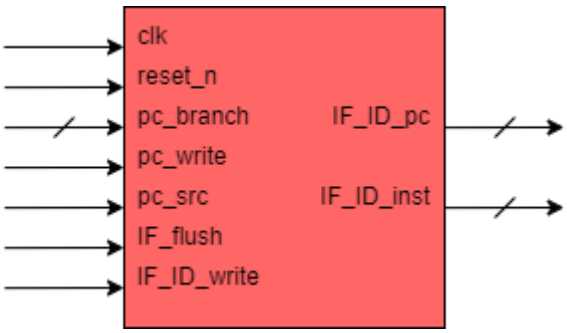
Hình 2.5 mô tả kiến trúc của khối **Instruction decode** gồm:

- Tệp thanh ghi nhận các giá trị địa chỉ thanh ghi **rs1**, **rs2** từ **inst** để đưa ra giá trị **data1, data2** tương ứng cho 2 thanh ghi. Nhận địa chỉ **rd**, **wb_data** từ khối **Register write**

(thực hiện giai đoạn write back) và tín hiệu **reg_write** từ **control** để ghi kết quả tính toán vào thanh ghi **rd**

- Bộ cộng nhằm tính toán địa chỉ **pc** cho lệnh brach
- Imm Gen nhằm signed extend cho giá trị **immediate**
- 2 bộ **mux4to1** nhận tín hiệu điều khiển từ khối **Forwarding unit** nhằm forward các kết quả tính toán của giai đoạn EX và MEM về bộ so sánh (branch compare) nhằm khắc phục data hazard khi thực hiện lệnh branch
- Branch comp là bộ so sánh so sánh giá trị của 2 thanh ghi **rs1**, **rs2** từ lệnh branch và đưa ra kết quả đến khối **control** để xác định xem chương trình sẽ nhảy hay sẽ thực hiện lệnh tiếp theo có địa chỉ **pc + 4**
- Thanh ghi **ID/EX** sẽ nhận kết quả từ khối **Instruction decode** đồng thời nhận các tín hiệu điều khiển được đưa ra từ khối **control** để gửi đến các khối **Execute, Memory access** và **Register write** để thực hiện tính toán cho các giai đoạn pipeline sau của lệnh

2.4 Module Excute (execute)



Hình 2.6 execute block diagram

2.4.1 Interface signals

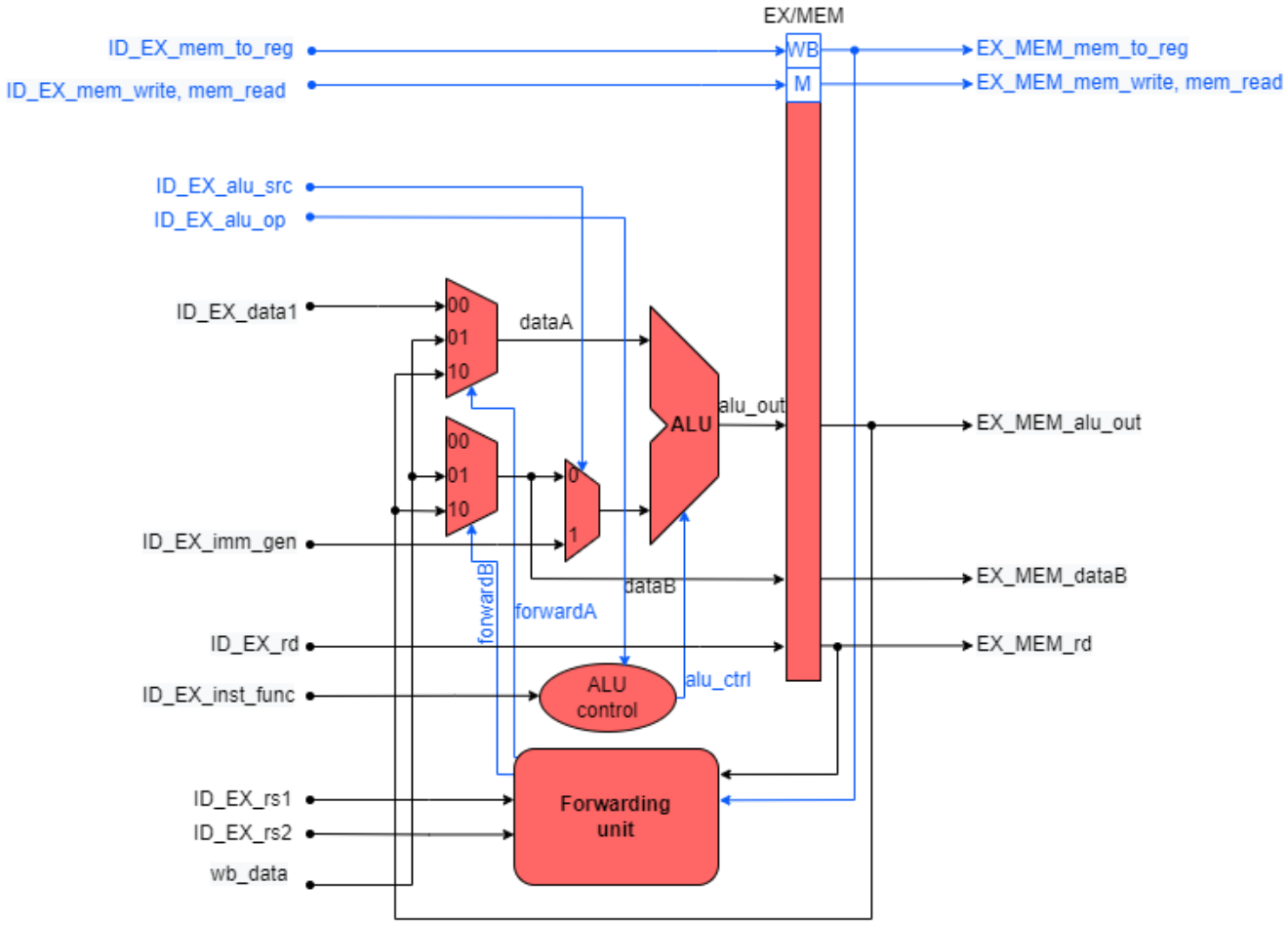
Bảng 2.4 execute port description

Signal name	Width	Input/Output	Description
clk	1	Input	Tín hiệu xung đồng hồ
reset_n	1	Input	Tín hiệu reset tích cực mức thấp
ID_EX_mem_to_reg	1	Input	Tín hiệu điều khiển chọn dữ liệu từ memory về thanh ghi của thanh ghi ID/EX
ID_EX_reg_write	1	Input	Cho phép ghi dữ liệu vào thanh ghi lưu trong thanh ghi ID/EX
ID_EX_mem_write	1	Input	Cho phép memory ghi dữ liệu lưu trong thanh ghi ID/EX
ID_EX_mem_read	1	Input	Cho phép memory đọc dữ liệu lưu trong thanh ghi ID/EX
ID_EX_alu_src	1	Input	Chọn chế độ địa chỉ trực tiếp lưu trong thanh ghi ID/EX
ID_EX_alu_op	2	Input	Opcode chọn phép toán lưu trong thanh ghi ID/EX
ID_EX_data1	32	Input	Dữ liệu từ thanh ghi 1 lưu trong thanh ghi ID/EX
ID_EX_data2	32	Input	Dữ liệu từ thanh ghi 2 lưu trong thanh ghi ID/EX
ID_EX_imm_gen	32	Input	Kết quả immidiate lưu trong thanh ghi ID/EX
ID_EX_rs1	5	Input	Địa chỉ của thanh ghi 1 lưu trong thanh ghi ID/EX
ID_EX_rs2	5	Input	Địa chỉ của thanh ghi 2 lưu trong thanh ghi ID/EX
ID_EX_rd	5	Input	Địa chỉ của thanh ghi đích lưu trong thanh ghi ID/EX
ID_EX_inst_func	4	Input	Function chọn phép toán lưu trong thanh ghi ID/EX
forwardA	2	Input	Forwarding khi có hazard tại thanh ghi rs1
forwardB	2	Input	Forwarding khi có hazard tại thanh ghi rs2
wb_data	32	Input	Dữ liệu được write back
EX_MEM_alu_out	32	Output	Kết quả alu được lưu trong thanh ghi EX/MEM
EX_MEM_mem_to_reg	1	Output	Tín hiệu điều khiển chọn dữ liệu từ memory về thanh ghi của thanh ghi EX/MEM
EX_MEM_reg_write	1	Output	Cho phép ghi dữ liệu vào thanh ghi lưu trong thanh ghi EX/MEM
EX_MEM_mem_write	1	Output	Cho phép memory ghi dữ liệu lưu trong thanh ghi EX/MEM
EX_MEM_mem_read	1	Output	Cho phép memory đọc dữ liệu lưu trong thanh ghi EX/MEM
EX_MEM_dataB	32	Output	Dữ liệu thanh ghi 2 được lưu trong thanh ghi EX/MEM
EX_MEM_rd	5	Output	Địa chỉ thanh ghi đích được lưu trong thanh ghi EX/MEM
alu_out	32	Output	Kết quả alu

2.4.2 Function description

Khối Execute sẽ nhận các tín hiệu điều khiển cho giai đoạn EX đã được lưu trong thanh ghi ID/EX và thực hiện tính toán dữ liệu hoặc địa chỉ tương ứng với yêu cầu của các instruction bằng khối ALU và ALU control.

2.4.3 Execute architecture



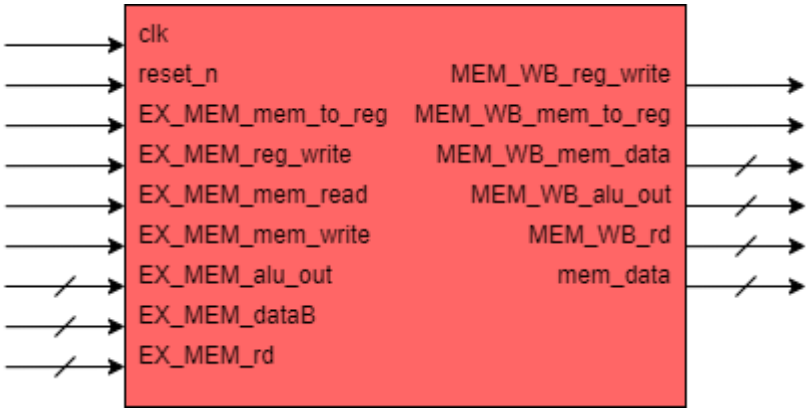
Hình 2.7 Execute architecture

Hình 2.7 mô tả kiến trúc của khối **Execute** gồm:

- 2 mộ **mux4to1** nhận tín hiệu từ khối **Forwarding unit** nhằm forward đầu vào cho khối **ALU** trong trường hợp hazard
- **Mux2to1** nhận tín hiệu **alu_src** từ khối **control** để chọn giữa dữ liệu thanh ghi **rs2** hoặc giá trị đầu ra của khối **Imm Gen** đã được lưu trong thanh ghi **ID/EX**
- **ALU control** nhận tín hiệu **alu_op** từ khối **control** và đưa ra tín hiệu **alu_ctrl** đến khối **ALU** để điều khiển hoạt động của **ALU** theo bảng sự thật Bảng 2.9
- **ALU** sẽ dựa vào tín hiệu **alu_ctrl** nhận được từ khối **ALU control** để thực hiện các phép toán tương ứng (add, sub, AND, OR, XOR,…)

- **ID/EX** là thanh ghi pipeline nhận các kết quả tính toán từ **ALU, dataB** và các giá trị của tín hiệu điều khiển để phục vụ cho các giai đoạn pipeline sau của lệnh. Bên cạnh đó còn nhận giá trị địa chỉ thanh ghi **rd** phục vụ cho việc nhận diện các trường hợp có hazard

2.5 Module Memory Access (memory_access)



Hình 2.8 memory_access block diagram

2.5.1 Interface signals

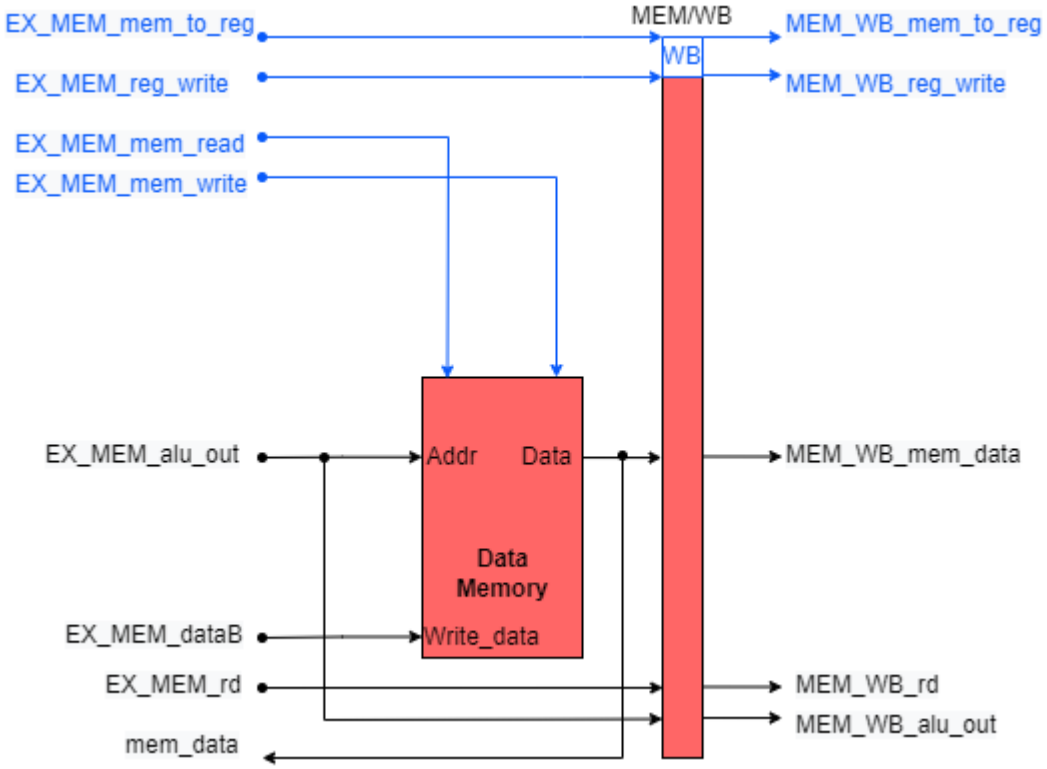
Bảng 2.5 memory_access port description

Signal name	Width	Input/Output	Description
clk	1	Input	Tín hiệu xung đồng hồ
reset_n	1	Input	Tín hiệu reset tích cực mức thấp
EX_MEM_mem_to_reg	1	Input	Tín hiệu điều khiển chọn dữ liệu từ memory về thanh ghi của thanh ghi EX/MEM
EX_MEM_reg_write	1	Input	Cho phép ghi dữ liệu vào thanh ghi lưu trong thanh ghi EX/MEM
EX_MEM_mem_read	1	Input	Cho phép memory đọc dữ liệu lưu trong thanh ghi EX/MEM
EX_MEM_mem_write	1	Input	Cho phép memory ghi dữ liệu lưu trong thanh ghi EX/MEM
EX_MEM_alu_out	32	Input	Kết quả alu được lưu trong thanh ghi EX/MEM
EX_MEM_dataB	32	Input	Dữ liệu thanh ghi 2 được lưu trong thanh ghi EX/MEM
EX_MEM_rd	5	Input	Địa chỉ thanh ghi đích được lưu trong thanh ghi EX/MEM
MEM_WB_reg_write	1	Output	Cho phép ghi dữ liệu vào thanh ghi lưu trong thanh ghi MEM/WB
MEM_WB_mem_to_reg	1	Output	Tín hiệu điều khiển chọn dữ liệu từ memory về thanh ghi của thanh ghi MEM/WB
MEM_WB_mem_data	32	Output	Dữ liệu từ memory được lưu trong thanh ghi MEM/WB
MEM_WB_alu_out	32	Output	Kết quả alu được lưu trong thanh ghi MEM/WB
MEM_WB_rd	5	Output	Địa chỉ thanh ghi đích được lưu trong thanh ghi MEM/WB
mem_data	32	Output	Dữ liệu từ memory

2.5.2 Function description

Khối Memory access nhận các tín hiệu điều khiển đọc ghi cho giai đoạn MEM từ thanh ghi EX/MEM để đọc hoặc ghi dữ liệu từ Data memory.

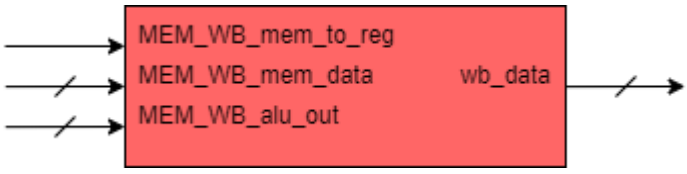
2.5.3 Memory access architecture



Hình 2.9 Memory access architecture

Hình 2.9 mô tả kiến trúc khối **Memory access**, gồm một bộ nhớ **Data Memory** nhận tín hiệu điều khiển **mem_read**, **mem_write** điều khiển việc đọc ghi memory. **Data Memory** nhận địa chỉ đã được tính toán từ **ALU** trong giai đoạn pipeline trước đó được lưu vào thanh ghi **EX/MEM** và đưa ra data tại địa chỉ đó nếu có tín hiệu đọc, hoặc sẽ ghi data được chuyển từ **rs2** (**DataB**) sang thanh ghi **EX/MEM** khi có tín hiệu ghi. Thanh ghi **MEM/WB** lưu lại dữ liệu được đọc ra từ memory và kết quả tính toán từ **ALU** được lấy ra ở thanh ghi **EX/MEM** để làm dữ liệu thực hiện giai đoạn pipeline tiếp theo là **write back**.

2.6 Module Register Write (register_write)



Hình 2.10 register_write block diagram

2.6.1 Interface signals

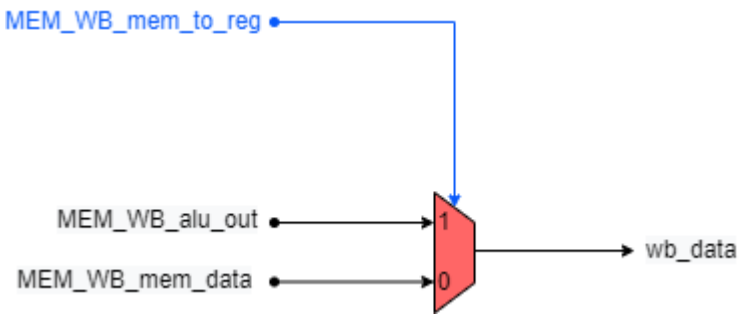
Bảng 2.6 register_write port description

Signal name	Width	Input/Output	Description
MEM_WB_mem_to_reg	1	Input	Tín hiệu điều khiển chọn dữ liệu từ memory về thanh ghi của thanh ghi MEM/WB
MEM_WB_mem_data	32	Input	Dữ liệu từ memory được lưu trong thanh ghi MEM/WB
MEM_WB_alu_out	32	Input	Kết quả alu được lưu trong thanh ghi MEM/WB
wb_data	32	Output	Dữ liệu write back

2.6.2 Function description

Khối Register Write thực hiện việc quyết định xem ghi kết quả vào thanh ghi hay không dựa vào tín hiệu **reg_write** nhận từ MEM/WB và quyết định xem ghi dữ liệu từ Data memory về thanh ghi hay giá trị được tính toán từ ALU chuyển sang.

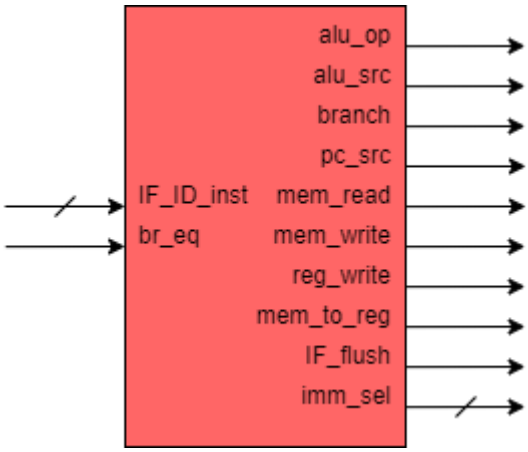
2.6.3 Register Write architecture



Hình 2.11 Register write architecture

Hình 2.11 mô mô tả kiến trúc khối **Register Write** thực hiện giai đoạn pipeline write back. Bộ **mux2to1** nhận tín hiệu điều khiển **mem_to_reg** để thực hiện việc quyết định giữa đưa giá trị dữ liệu từ **Data memory** về thanh ghi hay kết quả tính toán của **ALU**.

2.7 Module Control (control)



Hình 2.12 control block diagram

2.7.1 Interface signals

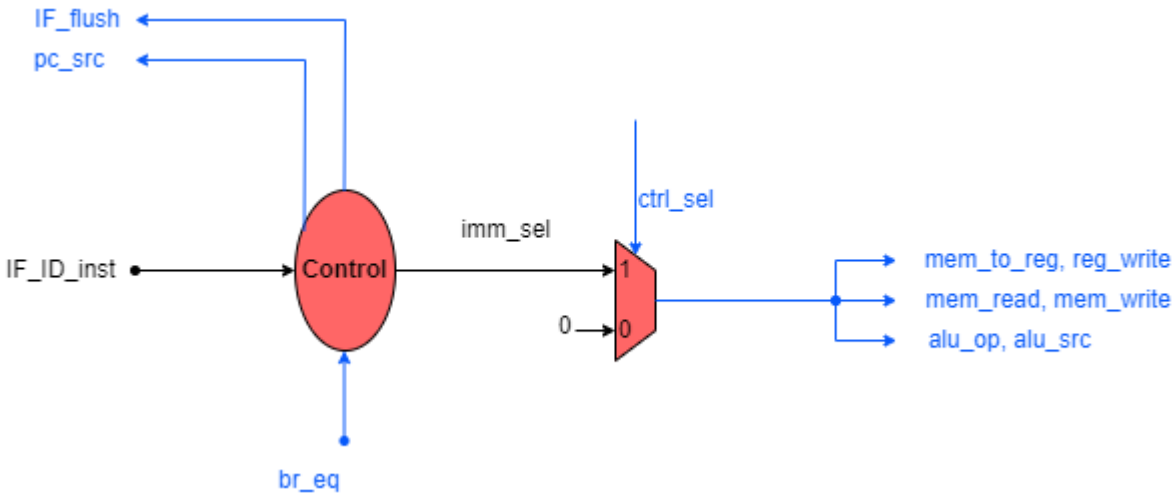
Bảng 2.7 control port description

Signal name	Width	Input/Output	Description
IF_ID_inst	32	Input	Mã lệnh được lưu trong thanh ghi IF/ID
br_eq	1	Input	Kết quả so sánh data1 và data2
alu_op	2	Output	Opcode chọn phép toán
alu_src	1	Output	Chọn chế độ địa chỉ trực tiếp
branch	1	Output	Cho phép nhảy đến địa chỉ offset
pc_src	1	Output	Cho phép chọn PC+4 hoặc PC_branch
mem_read	1	Output	Cho phép memory đọc dữ liệu
mem_write	1	Output	Cho phép memory ghi dữ liệu
reg_write	1	Output	Cho phép ghi dữ liệu vào thanh ghi
mem_to_reg	1	Output	Tín hiệu cho phép write back
IF_flush	1	Output	Cho phép xóa thanh ghi IF_ID
imm_sel	3	Output	Chọn kiểu cho Immidiate Generate

2.7.2 Function description

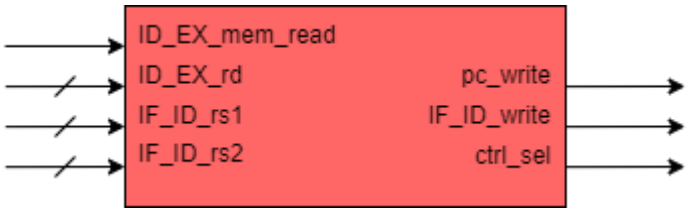
Khối **control** nhận 32 bit instruction làm đầu vào và dựa vào phần opcode của instruction rồi đưa ra các tín hiệu điều khiển tương ứng theo bảng sự thật Bảng 2.9

2.7.3 Control architecture



Hình 2.13 Control architecture

2.8 Module Hazard Detection Unit (hazard_detection_unit)



Hình 2.14 hazard_detection_unit block diagram

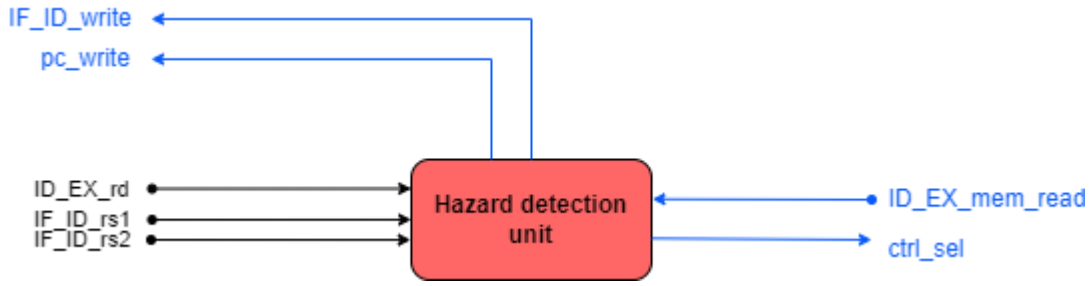
2.8.1 Interface signals

Signal name	Width	Input/Output	Description
ID_EX_mem_read	1	Input	Cho phép memory đọc dữ liệu lưu trong thanh ghi ID/EX
ID_EX_rd	5	Input	Địa chỉ của thanh ghi đích lưu trong thanh ghi ID/EX
IF_ID_rs1	5	Input	Địa chỉ của thanh ghi 1 lưu trong thanh ghi ID/EX
IF_ID_rs2	5	Input	Địa chỉ của thanh ghi 2 lưu trong thanh ghi ID/EX
pc_write	1	Output	Cho phép thay đổi đầu ra PC
IF_ID_write	1	Output	Cho phép thanh ghi IF_ID ghi dữ liệu mới
ctrl_sel	1	Output	Chọn đầu ra control

2.8.2 Function description

Khối **Hazard detection unit** có nhiệm vụ xác định có hazard mà không thể sử dụng kĩ thuật forwarding để xử lí mà cần stall pipeline 1 chu kì clk. Khối **Hazard detection unit** xác định trường hợp cần stall như sau:

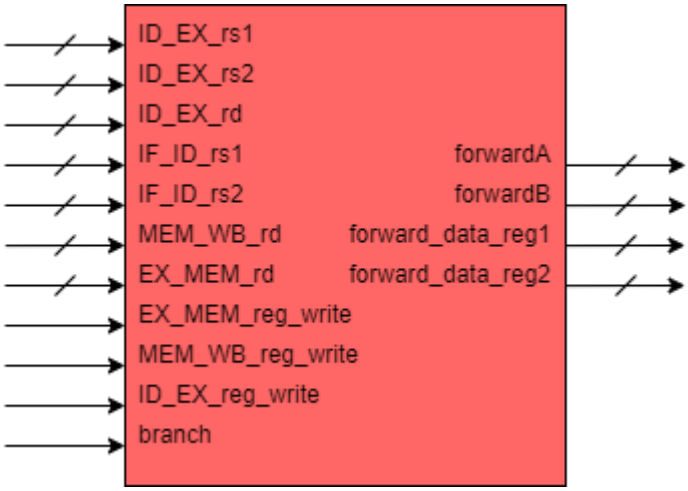
2.8.3 Hazard detection architecture



Hình 2.15 Hazard detection architecture

```
 if (ID_EX_mem_read) and (ID_EX_rd == IF_ID_rs1 or ID_EX_rd == IF_ID_rs2) then stall
the pipline.
```

2.9 Module Forwarding Unit (forwarding_unit)



Hình 2.16 forwarding_unit block diagram

2.9.1 Interface signals

Bảng 2.8 forwarding_unit port description

Signal name	Width	Input/Output	Description
ID_EX_rs1	5	Input	Địa chỉ của thanh ghi 1 lưu trong thanh ghi ID/EX
ID_EX_rs2	5	Input	Địa chỉ của thanh ghi 2 lưu trong thanh ghi ID/EX
ID_EX_rd	5	Input	Địa chỉ của thanh ghi đích lưu trong thanh ghi ID/EX
IF_ID_rs1	5	Input	Địa chỉ của thanh ghi 1 lưu trong thanh ghi IF/ID
IF_ID_rs2	5	Input	Địa chỉ của thanh ghi 2 lưu trong thanh ghi IF/ID
MEM_WB_rd	5	Input	Địa chỉ thanh ghi đích được lưu trong thanh ghi MEM/WB
EX_MEM_rd	5	Input	Địa chỉ thanh ghi đích được lưu trong thanh ghi EX/MEM
EX_MEM_reg_write	1	Input	Cho phép ghi dữ liệu vào thanh ghi lưu trong thanh ghi EX/MEM
MEM_WB_reg_write	1	Input	Cho phép ghi dữ liệu vào thanh ghi lưu trong thanh ghi MEM/WB
ID_EX_reg_write	1	Input	Cho phép ghi dữ liệu vào thanh ghi lưu trong thanh ghi ID/EX
branch	1	Input	Cho phép nhảy đến địa chỉ offset
forwardA	2	Output	Forwarding cho ALU khi có hazard tại thanh ghi rs1 EX
forwardB	2	Output	Forwarding cho ALU khi có hazard tại thanh ghi rs2 EX
forward_data_reg1	2	Output	Forwarding cho bộ branch compare khi có hazard tại thanh ghi rs1 ID
forward_data_reg2	2	Output	Forwarding cho bộ branch compare khi có hazard tại thanh ghi rs2 ID

2.9.2 Function description

Khối **Forwarding unit** sẽ nhận các giá trị địa chỉ **rs1**, **rs2**, **rd** và các tín hiệu điều khiển từ các thanh ghi pipeline nhằm xác định các trường hợp các lệnh liên tiếp nhau có xảy ra hazard (data hazard, mem hazard, control hazard) để đưa ra các tín hiệu điều khiển cho các bộ **mux** nhằm forward các dữ liệu bị hazard tương ứng để có được kết quả tính toán chính xác nhất mà không cần phải stall pipeline quá nhiều chu kì clk.

Khối **Forwarding unit** sẽ xác định các trường hợp cần forward như sau:

```
if (MEM_WB_reg_write
   and (MEM_WB_reg_write ≠ 0)
   and not(EX_MEM_reg_write and (EX_MEM_rd ≠ 0)
   and (EX_MEM_rd = ID_EX_rs1))
   and (MEM_WB_reg_write = ID_EX_rs1)) forwardA = 01
```

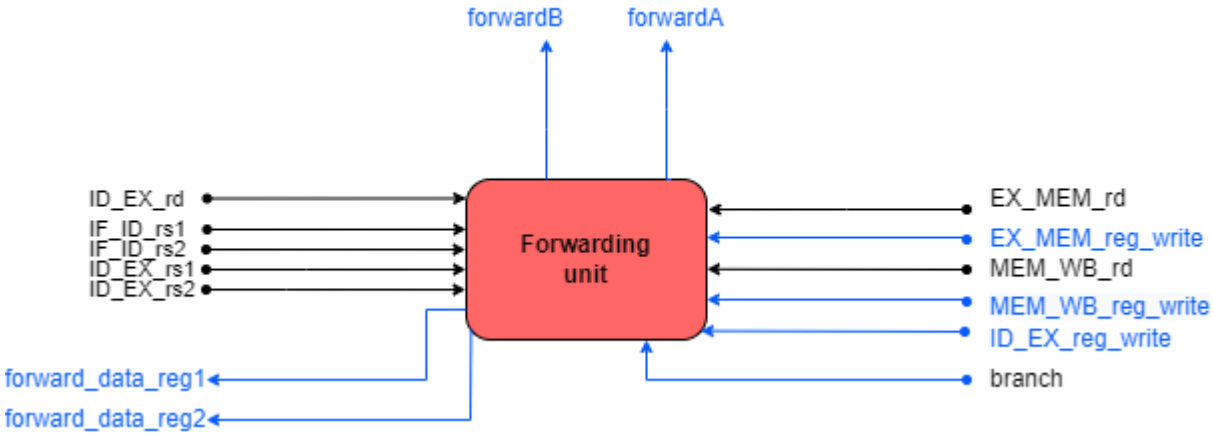
```
if (MEM_WB_reg_write
   and (MEM_WB_reg_write ≠ 0)
   and not(EX_MEM_reg_write and (EX_MEM_rd ≠ 0)
   and (EX_MEM_rd = ID_EX_rs2))
   and (MEM_WB_reg_write = ID_EX_rs2)) forwardB = 01


if (branch && (ID_EX_rd != 0) && ID_EX_reg_write && (IF_ID_rs1 == ID_EX_rd))
   forward_data_reg1 = 01;
else if (branch && (EX_MEM_rd != 0) && ~(ID_EX_reg_write && (IF_ID_rs1 == ID_EX_rd))
&& EX_MEM_reg_write && (EX_MEM_rd == IF_ID_rs1))
   forward_data_reg1 = 10;
else forward_data_reg1 = 00;


if (branch && (ID_EX_rd != 0) && ID_EX_reg_write && (IF_ID_rs2 == ID_EX_rd))
   forward_data_reg2 = 01;
else if (branch && (EX_MEM_rd != 0) && ~(ID_EX_reg_write && (IF_ID_rs2 == ID_EX_rd))
&& EX_MEM_reg_write && (EX_MEM_rd == IF_ID_rs2))
   forward_data_reg2 = 10;
else forward_data_reg2 = 00;
```

Bảng 2.9 Forwarding output ports

Mux control	**Source**	**Explaination**
forwardA = 00	**ID/EX**	Đường input thứ nhất của **ALU** đến từ tệp thanh ghi.
forwardA = 10	**EX/MEM**	Đường input thứ nhất của **ALU** được foward từ kết quả tính toán của **ALU** cho lệnh ngay trước đó.
forwardA = 01	**MEM/WB**	Đường input thứ nhất của **ALU** được forward từ kết quả đọc data từ **memory** hoặc kết quả tính toán của **ALU** cho lệnh trước đó cách lệnh đang thực hiện 1 lệnh.
forwardB = 00	**ID/EX**	Đường input thứ hai của **ALU** đến từ tệp thanh ghi.
forwardB = 10	**EX/MEM**	Đường input thứ hai của **ALU** được foward từ kết quả tính toán của **ALU** cho lệnh ngay trước đó.
forwardB = 01	**MEM/WB**	Đường input thứ hai của **ALU** được forward từ kết quả đọc data từ memory hoặc kết quả tính toán của **ALU** cho lệnh trước đó cách lệnh đang thực hiện 1 lệnh.
forward_ data_reg1 = 00	**Register**	Đường input thứ nhất của **branch compare** đến từ tệp thanh ghi.

forward_ data_reg1 = 10	**mem** or **EX/MEM**	Đường input thứ nhất của **branch compare** đến từ kết quả đọc data từ **memory** hoặc kết quả **ALU** của lệnh trước đó cách lệnh đang thực hiện 1 lệnh.
forward_ data_reg1 = 01	**ALU**	Đường input thứ nhất của **branch compare** đến từ kết quả tính toán **ALU** của lệnh trước đó.
forward_ data_reg2 = 00	**Register**	Đường input thứ hai của **branch compare** đến từ tệp thanh ghi.
forward_ data_reg2 = 10	**mem** or **EX/MEM**	Đường input thứ hai của **branch compare** đến từ kết quả đọc data từ **memory** hoặc kết quả **ALU** của lệnh trước đó cách lệnh đang thực hiện 1 lệnh.
forward_data_reg2 = 01	**ALU**	Đường input thứ hai của **branch compare** đến từ kết quả tính toán **ALU** của lệnh trước đó.

2.9.3 Forwarding unit architecture



Hình 2.17 Forwarding unit architecture

2.10 Module Top (riscv_pipeline_top)

Hình 2.18 riscv_pipeline_top block diagram

Hình 2.1818 mô tả toàn bộ các module con có trong khối top và các interface tương ứng được kết nối với nhau.

2.10.1 Interface signals

Bảng 2.10 riscv_pipeline_top port description

Signal name	Width	Input/Output	Description
clk	5	Input	Tín hiệu xung đồng hồ
reset_n	5	Input	Tín hiệu reset không đồng bộ, tích cực mức thấp

2.10.2 Function description

Khối top là khối kết nối tất cả các module con của CPU, nhận xung **clk** và tín hiệu **reset** để hoạt động, chương trình cần CPU thực hiện sẽ được nạp sẵn vào bộ nhớ **Instruction memory**.

CHƯƠNG 3. KIỂM THỬ (VERIFICATION)

Chương này trình bày các kết quả mô phỏng cho từng khối và toàn bộ thiết kế được triển khai bằng ngôn ngữ SystemVerilog trên phần mềm ModelSim.

3.1 Kế hoạch kiểm thử

3.1.1 Kịch bản – Test case

- Kiểm tra quá trình reset, đảm bảo các đầu ra được thiết lập về đúng giá trị
- Kiểm tra trường hợp reset bất thường khi mạch đang hoạt động
- Kiểm tra trường hợp tín hiệu reset được đặt tích cực trong nhiều chu kỳ
- Kiểm tra các trường hợp instruction không có hazard
- Kiểm tra các trường hợp instruction có data hazard
- Kiểm tra các trường hợp instruction có mem hazard
- Kiểm tra các trường hợp instruction có control hazard

3.1.2 Kích thích đầu vào – Stimulus

- Tạo sẵn các mã lệnh trong instruction memory như bảng

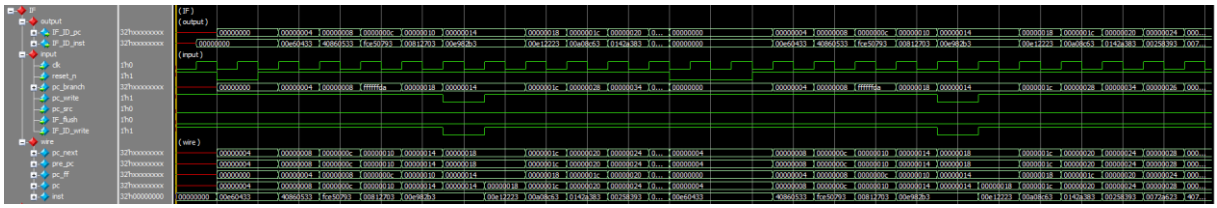
Localparam	**Address**	**Assembly code**	**Instruction**
NONE	32'h0000	Nothing	Nothing
INST1	32'h0004	add x8, x12, x14	0000000_01110_01100_000_01000_0110011
INST2	32'h0008	sub x10, x12, x8	0100000_01000_01100_000_01010_0110011
INST3	32'h000C	addi x15, x10, -50	111111001110_01010_000_01111_0010011
INST4	32'h0010	lw x14, 8(x2)	000000001000_00010_010_01110_0000011
INST5	32'h0014	add x5, x19, x14	0000000_01110_10011_000_00101_0110011
INST6	32'h0018	sw x14, 4(x2)	0000000_01110_00010_010_00100_0100011
INST7	32'h001C	beq x1, x10, offset(12)	0_000000_01010_00001_000_1100_0_1100011
INST8	32'h0020	lw x7, 20(x5)	000000010100_00101_010_00111_0000011
INST9	32'h0024	addi x7, x11, 2	000000000010_01011_000_00111_0010011
INST10	32'h0028	sw x7, 12(x5)	000000000111_00101_010_01100_0100011
INST11	32'h002C	sub x2, x11, x7	0100000_00111_01011_000_00010_0110011
INST12	32'h0030	and x14, x5, x3	0000000_00011_00101_111_01110_0110011
INST13	32'h0034	sw x14, 16(x5)	000000001110_00101_010_10000_0100011
INST14	32'h0038	beq x1, x14, offset(12)	0_000000_01110_00001_000_1100_0_1100011
INST15	32'h003C	add x8, x12, x14	0000000_01110_01100_000_01000_0110011
INST16	32'h0040	sub x10, x12, x8	0100000_01000_01100_000_01010_0110011

INST17	32'h0044	addi x15, x10, -50	111111001110_01010_000_01111_0010011
INST18	32'h0048	lw x14, 8(x2)	000000001000_00010_010_01110_0000011
INST19	32'h004C	add x5, x19, x14	0000000_01110_10011_000_00101_0110011
INST20	32'h0050	lw x14, 10(x2)	000000001010_00010_010_01110_0000011
INST21	32'h0054	beq x1, x14 , offset(12)	0_000000_01110_00001_000_1100_0_1100011
INST22	32'h0058	add x15, x12, x14	0000000_01110_01100_000_01111_0110011

Các lệnh trên được tạo ra để test các kịch bản có thể xảy ra khi thực hiện chương trình bằng kiến trúc vi xử lí RISCV sử dụng kĩ thuật pipeline. Kịch bản test trên đã bao gồm data hazard, mem hazard và cuối cùng là control hazard. Các thanh ghi được highlight thể hiện có hazard trong chuỗi các instruction.

3.2 Kết quả và đánh giá

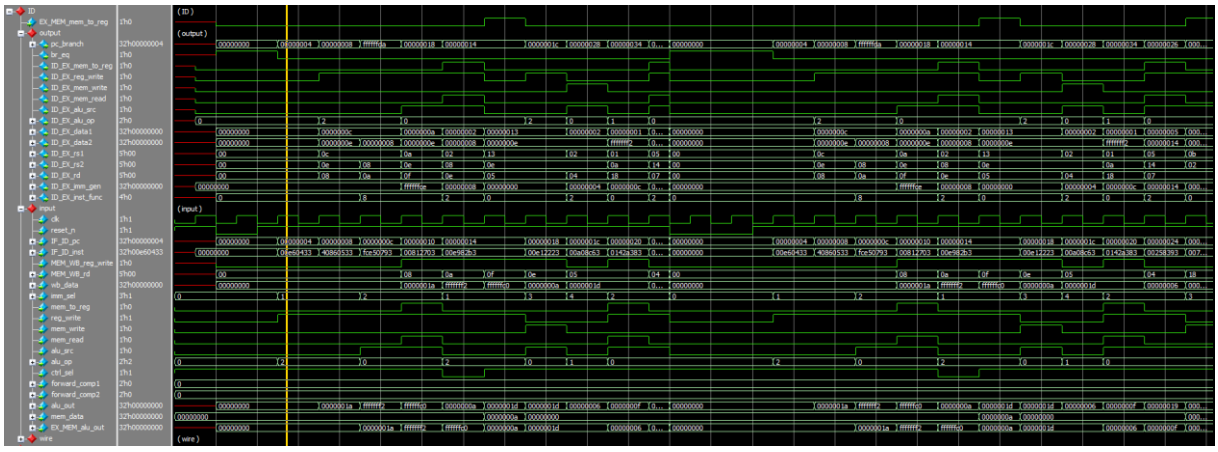
3.2.1 Module instruction_fetch



Hình 3.1 Mô phỏng timing diagram khối instruction_fetch

Hình 3.1 cho thấy kết quả đầu ra hoạt động đúng logic khi thực hiện các lệnh trong kịch bản được nạp sẵn vào **Instruction Memory**.

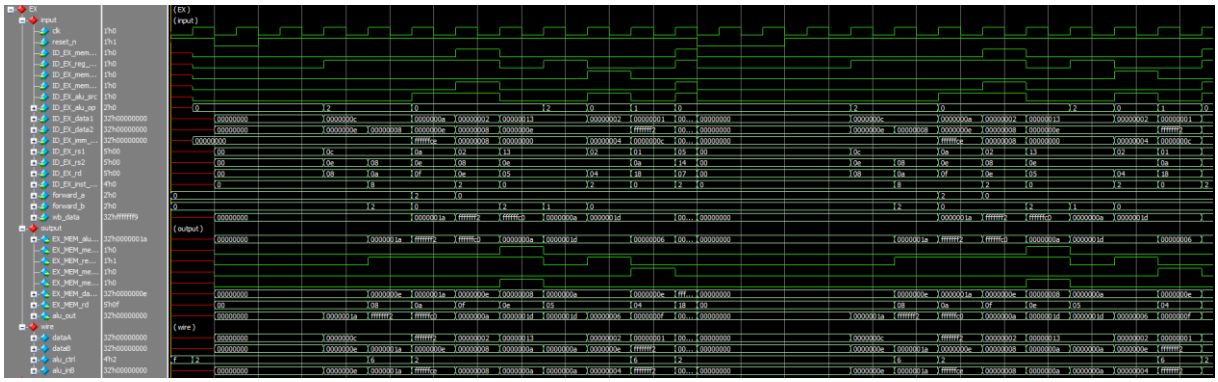
3.2.2 Module instruction_decode



Hình 3.2 Mô phỏng timing diagram khối instruction_decode

Hình 3.2 cho thấy kết quả đầu ra của khối **Instruction Decode** hoạt động đúng với logic của thiết kế khi thực hiện các lệnh trong kịch bản được nạp sẵn vào **Instruction Memory**.

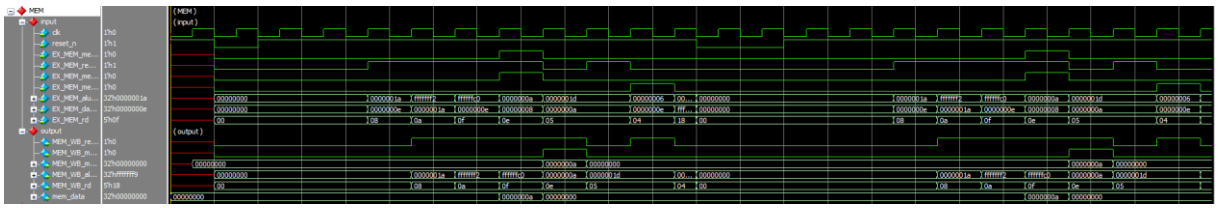
3.2.3 Module execute



Hình 3.3 Mô phỏng timing diagram khối execute

Hình 3.3 cho thấy kết quả đầu ra của khối **Execute** hoạt động đúng với logic của thiết kế khi thực hiện các lệnh trong kịch bản được nạp sẵn.

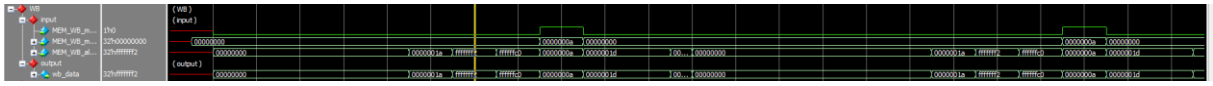
3.2.4 Module memory_access



Hình 3.4 Mô phỏng timing diagram khối memory_access

Hình 3.4 cho thấy kết quả đầu ra của khối **Memory Access** hoạt động đúng với logic của thiết kế khi thực hiện các lệnh trong kịch bản được nạp sẵn.

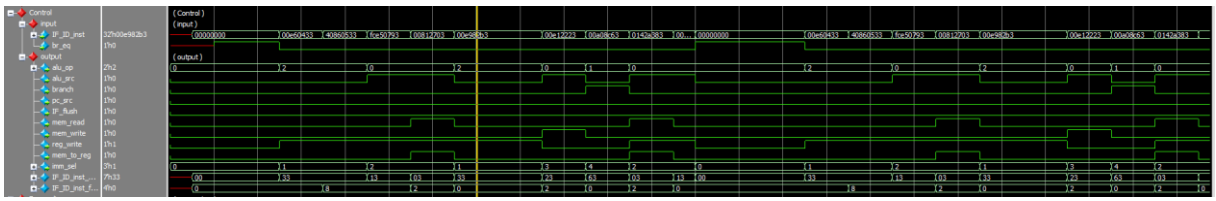
3.2.5 Module register_write



Hình 3.5 Mô phỏng timing diagram khối register_write

Hình 3.5 cho thấy kết quả đầu ra của khối **Register Write** hoạt động đúng với logic của thiết kế.

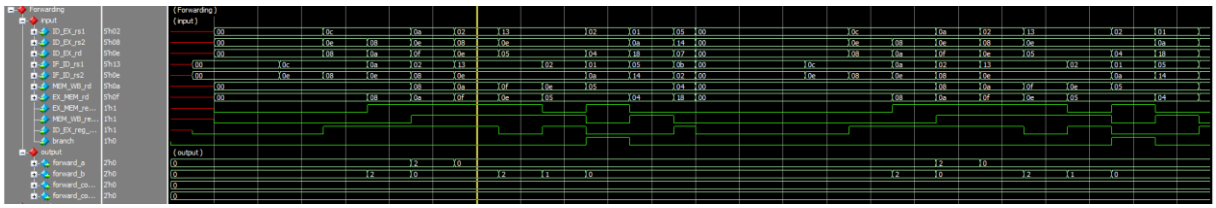
3.2.6 Module control



Hình 3.6 Mô phỏng timing diagram khối control

Hình 3.6 cho thấy các tín hiệu điều khiển **control** gửi đến **datapath** đúng với logic thiết kế khi thực hiện các lệnh được nạp sẵn trong **Instruction Memory**.

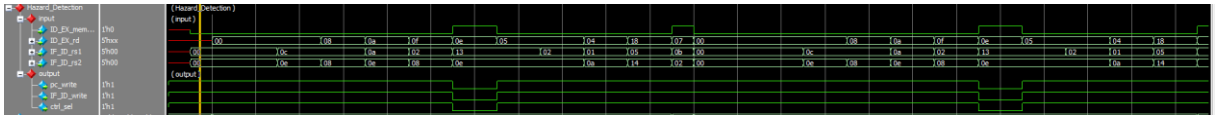
3.2.7 Module forwarding_unit



Hình 3.7 Mô phỏng timing diagram khối forwarding_unit

Hình 3.7 mô tả tín hiệu đầu ra của module khối **forwarding unit** theo các kích thích đầu vào tương ứng, wave form cho thấy module hoạt động đúng logic đã định nghĩa.

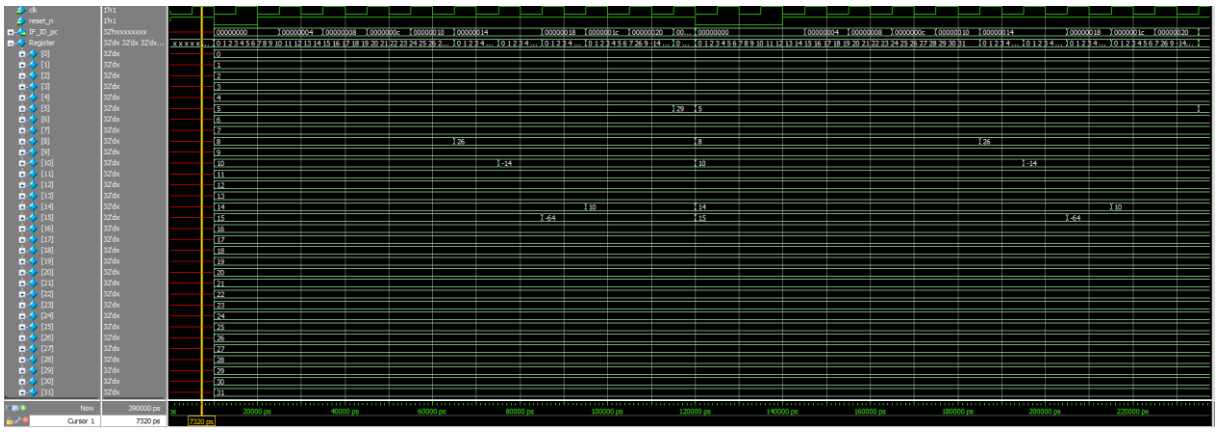
3.2.8 Module hazard_detection_unit



Hình 3.8 Mô phỏng timing diagram khối hazard_detection_unit

Hình 3.8 mô tả tín hiệu đầu ra của module khối **hazard detection unit** theo các kích thích đầu vào tương ứng, wave form cho thấy module hoạt động đúng logic đã định nghĩa.

3.2.9 Register

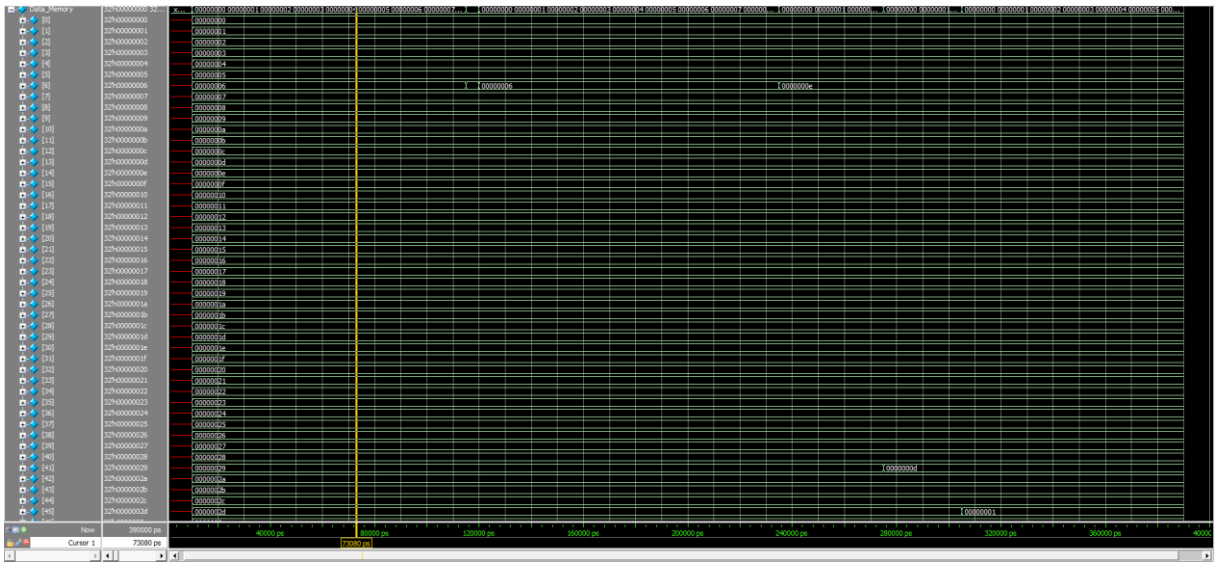


Hình 3.9 Mô phỏng 32 Registers

Hình 3.9 thể hiện giá trị của tệp thanh ghi đã được thay đổi sau khi thực hiện chương trình. Wave tín hiệu cho thấy kể từ khi lệnh được nạp vào CPU (tương ứng với giá trị **pc** thay đổi) thì sau 5 chu kì tiếp theo giá trị thanh ghi được thay đổi đúng như kết quả tính toán từ trước cho các lệnh nạp sẵn theo kịch bản cho trước.

Khi có tín hiệu **reset** bất thường, CPU được reset về trạng thái ban đầu, giá trị thanh ghi được khởi tạo tương ứng với địa chỉ và **pc** được reset về 0, chương trình trong **Instruction memory** được chạy lại từ đầu và vẫn đũng so với kết quả đã tính toán trước theo kịch bản lệnh được nạp sẵn.

3.2.10 Data_Memory



Hình 3.10 Mô phỏng Data_Memory

Hình 3.10 mô tả sự thay đổi của dữ liệu bên trong **Data Memory**, dữ liệu thay đổi đúng như kịch bản tính toán của các câu lệnh được nạp sẵn trong **Instruction Memory**.

CHƯƠNG 4. KẾT LUẬN

Báo cáo này đã trình bày và triển khai kiến trúc của một RISCV32I processor áp dụng kĩ thuật pipeline nhằm tăng tốc độ xử lí, bên cạnh đó, thiết kế đã xữ lí được toàn bộ các hazard có thể xảy ra trong quá trình xử lí thực hiện lệnh của CPU bao gồm data hazard, mem hazard và control hazard. Thiết kế được tiến hành triển khai bằng ngôn ngữ mô tả phần cứng SystemVerilog và mô phỏng kiểm thử trên phần mềm ModelSim. Cho ra kết quả hoạt động đúng với yêu cầu vào ra. Kiến trúc có thể thực hiện 4 kiểu lệnh assembly trong tập lệnh của RISCV gồm: R-type, I-type, S-type, B-type, tuy nhiên chưa thể thực hiện được tất cả các lệnh có trong tập lệnh của RISCV (xấp xỉ 40 lệnh). Trong tương lai sẽ tiến hành hoàn thiện và triển khai kiến trúc của một RISCV32I processor hoàn chỉnh với đầy đủ các chức năng, thực hiện được đầy đủ các lệnh trong kiến trúc tập lệnh của RISCV một cách tối ưu nhất.

TÀI LIỆU THAM KHẢO

[1] https://en.wikipedia.org/wiki/RISC-V Ngày truy cập cuối cùng: 3/1/2021

[2] Slide: Kiến trúc máy tính của cô Tạ Thị Kim Huệ

[3] Computer Organization and Design Risc – V The Hardware Software Interface by David A. Patterson and John L. Hennessy

[4] Link project: https://github.com/LeQuangHung00/Riscv_pineline