|  |
| --- |
| **TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI**  **VIỆN ĐIỆN TỬ - VIỄN THÔNG**  Ảnh có chứa văn bản, ký hiệu, thực phẩm  Mô tả được tạo tự động  **BÁO CÁO KIẾN TRÚC MÁY TÍNH**  **Đề tài: Thiết kế bộ xử lý RISC-V Pipeline bằng ngôn ngữ Verilog**  **Nhóm sinh viên thực hiện:**  **Tên sinh viên MSSV Mã lớp**  Phạm Ngọc Lâm 20182628 124876  Dương Văn Biên 20172426 124876  Nguyễn Văn Chiến 20172431 124876  **Giảng viên hướng dẫn:** PGS - TS. Nguyễn Đức Minh  **Hà Nội - 2021** |

MỤC LỤC

[MỤC LỤC 1](#_Toc72089230)

[DANH MỤC HÌNH ẢNH i](#_Toc72089231)

[DANH MỤC BẢNG i](#_Toc72089232)

[LỜI NÓI ĐẦU ii](#_Toc72089233)

[CHƯƠNG 1. GIỚI THIỆU (INTRODUCTION) 1](#_Toc72089234)

[1.1 Giới thiệu chung 1](#_Toc72089235)

[CHƯƠNG 2. ĐẶC TẢ THÔNG SỐ KĨ THUẬT (SPECIFICATION) 2](#_Toc72089236)

[2.1 Tổng quan hệ thống 2](#_Toc72089237)

[2.1.1 Sơ đồ khối 2](#_Toc72089238)

[2.1.2 Mô tả chức năng 3](#_Toc72089239)

[2.2 Khối Controller 3](#_Toc72089240)

[2.2.1 Sơ đồ khối 3](#_Toc72089241)

[2.2.2 Mô tả chức năng khối Controller 4](#_Toc72089242)

[2.2.3 Triển khai khối controller 6](#_Toc72089243)

[2.3 Khối Datapath 7](#_Toc72089244)

[2.3.1 Khối Instruction Fetch 7](#_Toc72089245)

[2.3.2 Khối Instruction Decode 8](#_Toc72089246)

[2.3.3 Khối Execute 9](#_Toc72089247)

[2.3.4 Khối Memory Access 11](#_Toc72089248)

[2.3.5 Mô tả chức năng của khối Datapath 12](#_Toc72089249)

[CHƯƠNG 3. KIỂM THỬ (VERTIFICATION) 13](#_Toc72089250)

[3.1 Kế hoạch kiểm thử - khối Control Logic 13](#_Toc72089251)

[3.1.1 Kịch bản – Test case 13](#_Toc72089252)

[3.1.2 Kích thích đầu vào – Stimulus 13](#_Toc72089253)

[3.1.3 Kiểm tra đầu ra 13](#_Toc72089254)

[3.1.4 Mô phỏng kiểm thử thiết kế 14](#_Toc72089255)

[3.2 Kế hoạch kiểm thử - khối Datapath 14](#_Toc72089256)

[3.2.1 Kịch bản – Test case 14](#_Toc72089257)

[3.2.2 Kích thích đầu vào – Stimulus 14](#_Toc72089258)

[3.2.3 Mô phỏng kiểm thử thiết kế 15](#_Toc72089259)

[CHƯƠNG 4. KẾT LUẬN 16](#_Toc72089260)

[TÀI LIỆU THAM KHẢO 17](#_Toc72089261)

DANH MỤC HÌNH ẢNH

[Hình 2.1 Sơ đồ khối hệ thống 2](file:///D:\EDABK\Lab%20Exercise\MegaBonus\Datapath\BÁO-CÁO-KTMT-20202%20(1).docx#_Toc72089190)

[Hình 2.2 Sơ đồ khối Controller 3](file:///D:\EDABK\Lab%20Exercise\MegaBonus\Datapath\BÁO-CÁO-KTMT-20202%20(1).docx#_Toc72089191)

[Hình 2.3 Control Logic Truth Table 5](file:///D:\EDABK\Lab%20Exercise\MegaBonus\Datapath\BÁO-CÁO-KTMT-20202%20(1).docx#_Toc72089192)

[Hình 2.4 Timing của khối control logic 5](file:///D:\EDABK\Lab%20Exercise\MegaBonus\Datapath\BÁO-CÁO-KTMT-20202%20(1).docx#_Toc72089193)

[Hình 2.5 ROM-based control 6](file:///D:\EDABK\Lab%20Exercise\MegaBonus\Datapath\BÁO-CÁO-KTMT-20202%20(1).docx#_Toc72089194)

[Hình 2.6 Kiến trúc ROM-based control 6](file:///D:\EDABK\Lab%20Exercise\MegaBonus\Datapath\BÁO-CÁO-KTMT-20202%20(1).docx#_Toc72089195)

[Hình 2.7 Sơ đồ khối Instruction Fetch 7](file:///D:\EDABK\Lab%20Exercise\MegaBonus\Datapath\BÁO-CÁO-KTMT-20202%20(1).docx#_Toc72089196)

[Hình 2.8 Sơ đồ khối Instruction Decode 8](file:///D:\EDABK\Lab%20Exercise\MegaBonus\Datapath\BÁO-CÁO-KTMT-20202%20(1).docx#_Toc72089197)

[Hình 2.9 Sơ đồ khối Execute 9](#_Toc72089198)

[Hình 2.10 Sơ đồ khối Memory Access 11](#_Toc72089199)

[Hình 2.11 Timing diagram quá trình hoạt động khối Datapath 12](file:///D:\EDABK\Lab%20Exercise\MegaBonus\Datapath\BÁO-CÁO-KTMT-20202%20(1).docx#_Toc72089200)

[Hình 3.1 Đầu ra mô phỏng trên ModelSim 14](file:///D:\EDABK\Lab%20Exercise\MegaBonus\Datapath\BÁO-CÁO-KTMT-20202%20(1).docx#_Toc72089201)

[Hình 3.2 Kết quả mô phỏng kiểm thử khối top\_datapath 15](file:///D:\EDABK\Lab%20Exercise\MegaBonus\Datapath\BÁO-CÁO-KTMT-20202%20(1).docx#_Toc72089202)

DANH MỤC BẢNG

[Bảng 2‑1 Tín hiệu vào ra hệ thống 2](#_Toc72077705)

[Bảng 2‑2 Tín hiệu vào/ra khối Controller 4](#_Toc72077706)

[Bảng 2‑3 Bảng tín hiệu vào/ra khối Instruction Fetch 7](#_Toc72077707)

[Bảng 2‑4 Bảng tín hiệu vào/ra khối Instruction Decode 9](#_Toc72077708)

[Bảng 2‑5 Tín hiệu vào/ra khối Execute 10](#_Toc72077709)

[Bảng 2‑6 Tín hiệu vào/ra khối Memory Access 11](#_Toc72077710)

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 trong RISC-V 32I nhằm hiểu rõ về quá trình hoạt động khi thực hiện các lệnh. Trong báo cáo này, chúng em triển khai kiến trúc RISC-V đơn giản sử dụng ngôn ngữ mô tả phần cứng Verilog và mô phỏng kiểm thử trên phần mêm ModelSim. 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**

Nhóm chúng em xin chân thành cảm ơn PGS.TS Nguyễn Đức Minh đã tận tâm hướng dẫn chúng em trong quá trình thực hiện đồ án cũng như hoàn hiện báo cáo này !

# GIỚI THIỆU (INTRODUCTION)

*Chương này giới thiệu khái quát về Risc-V 32I.*

## 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:

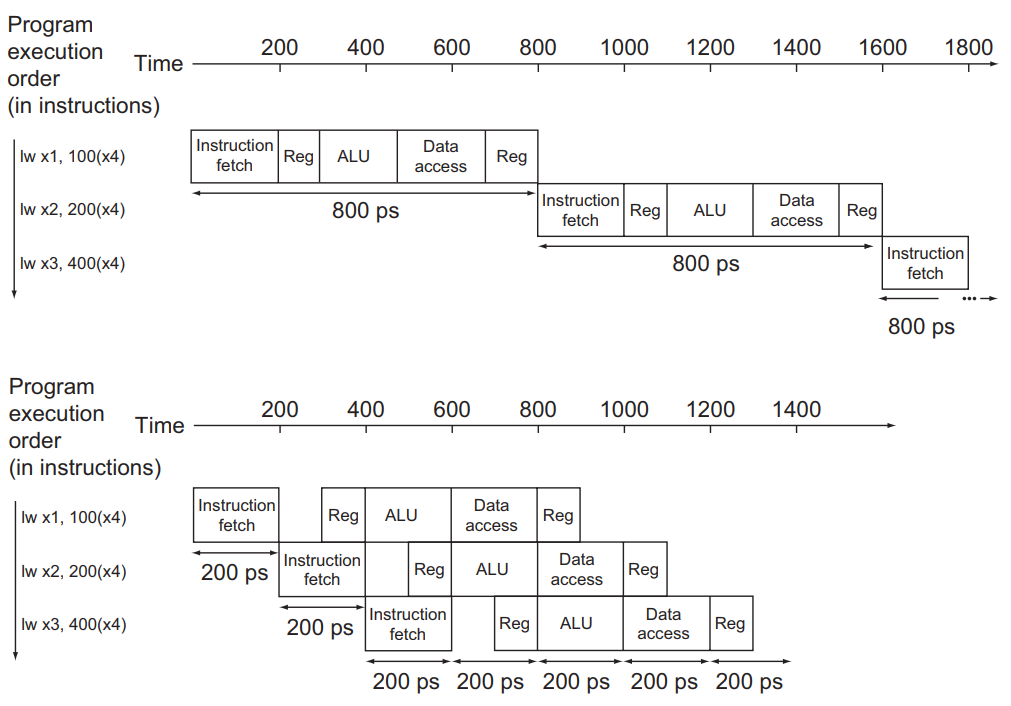
* Kiến trúc *load – store*
* Các mẫu bit để đơn giản hóa bộ MUX trong CPU
* Dấu phẩy động IEEE754
* Thiết kế trung lập về mặt kiến trúc và đặt *most-significant* bits tại một vị trí cố định để tăng tốc độ *sign extension*
* Tập lệnh được thiết kế cho nhiều mục đích sử dụng. Tập lệnh cơ sở có độ dài cố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

## Pipeline

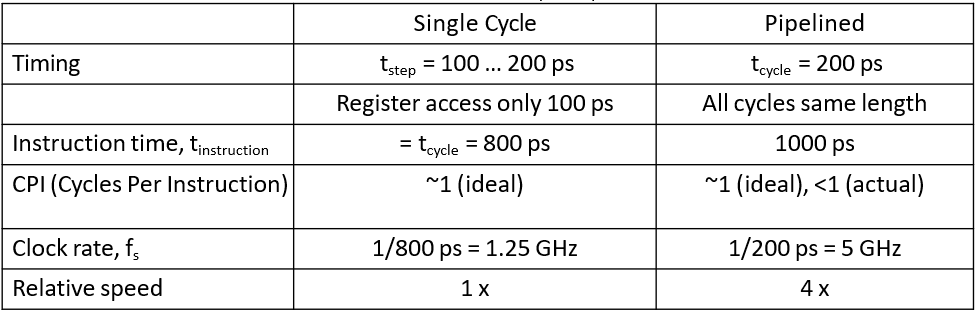
Pipelining là một kĩ thuật triển khai trong đó nhiều lệnh được chồng lên nhau trong quá trình thực thi. Ngày nay, pipeline gần như phổ biến và có nhiều ứng dụng. Quá trình thực hiện một lệnh trong RISC – V cổ điển bao gồm 5 bước:

* 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)

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



Hình 1.1 Single cycle versus pipelined (1)



Hình 1.2 Single cycle versus pipelined (2)

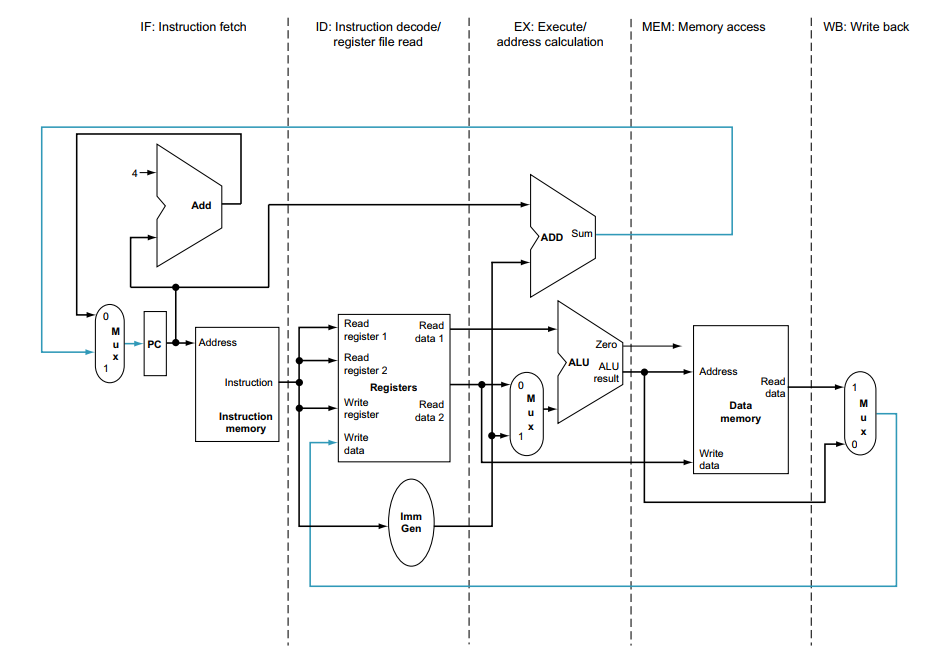
Dựa vào Hình 1.1, ta sử dụng chung một kiến trúc phần cứng, nhận thấy thời gian trung bình giữa các lệnh tăng gấp 4 lần từ 800 ps xuống 200ps. Tuy nhiên, thời gian của giai đoạn pipeline cũng bị giới hạn bởi tài nguyên chậm nhất như hoạt động ALU hoặc truy cập vào bộ nhớ. Kết quả cụ thể được mô tả trên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

## 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:

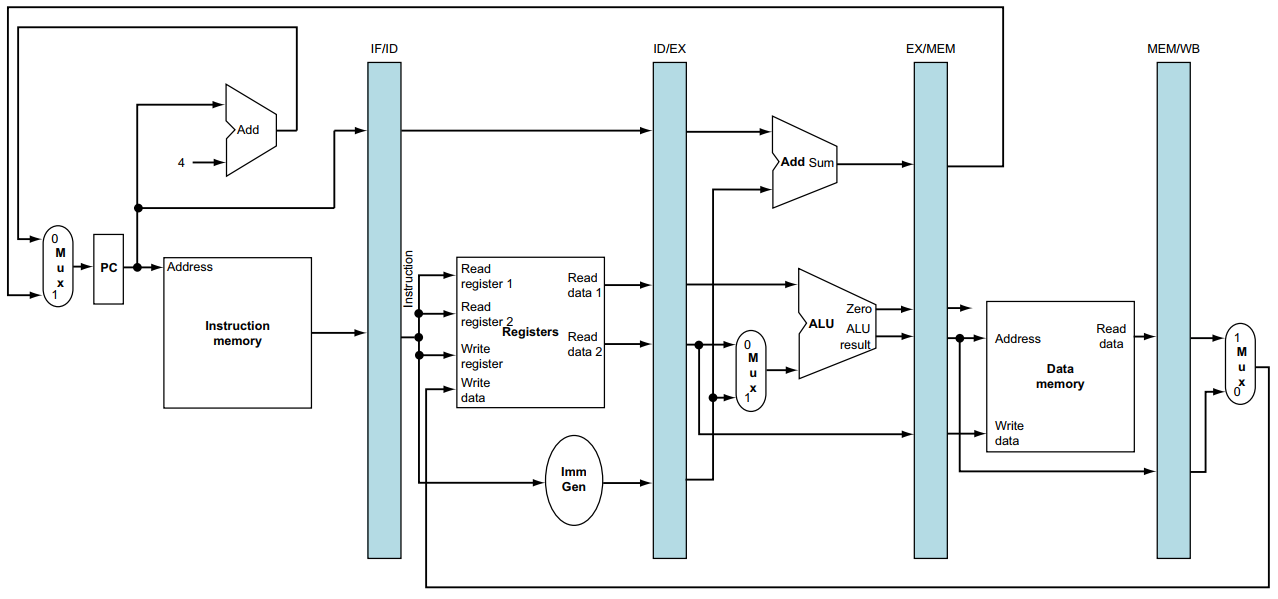
1. IF: Instruction Fetch
2. ID: Instruction Decode/register file read
3. EX: Excute/address calculation
4. MEM: Memory access
5. WB: Write back



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.

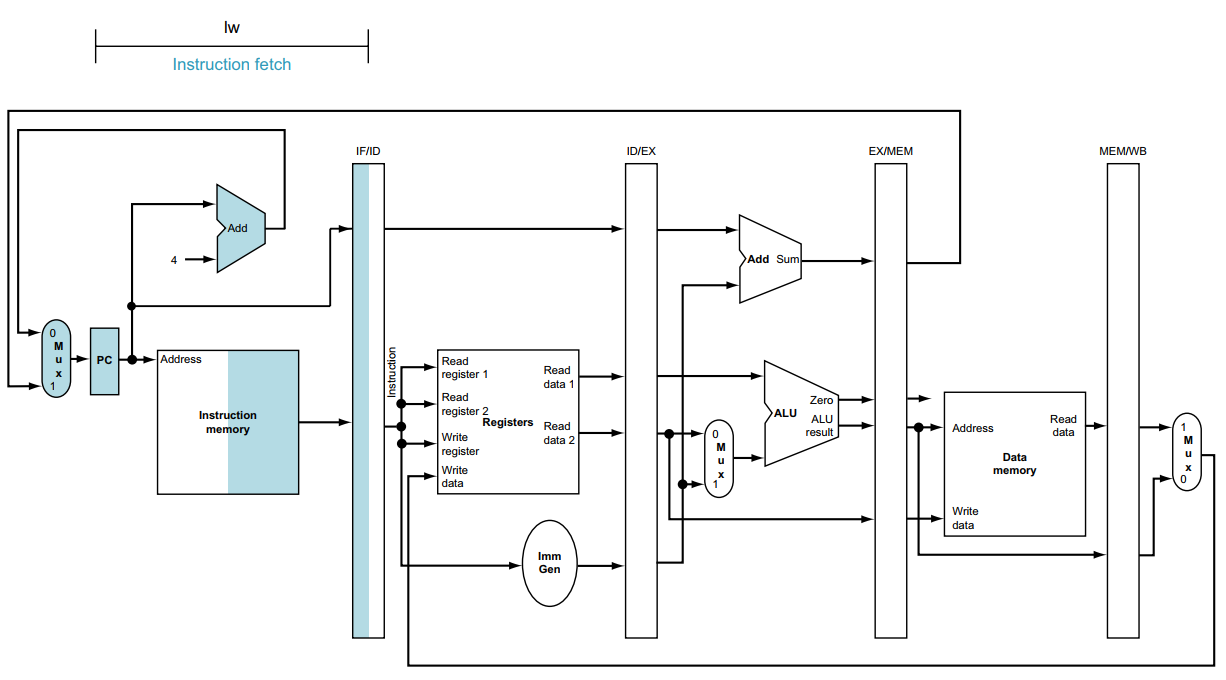
Hình 1.4 mô tả pipeline datapath với thanh ghi pipeline. Các thanh ghi pipeline được đặt tên theo hai giai đoạn bị ngăn cách bởi thanh ghi đó. Ví dụ thanh ghi pipeline IF/ID ngăn cách hai giai đoạn IF và IF của datapath. Và không có thanh ghi tại giai đoạn cuối cùng WB. Các thanh ghi pipeline phải đủ 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.



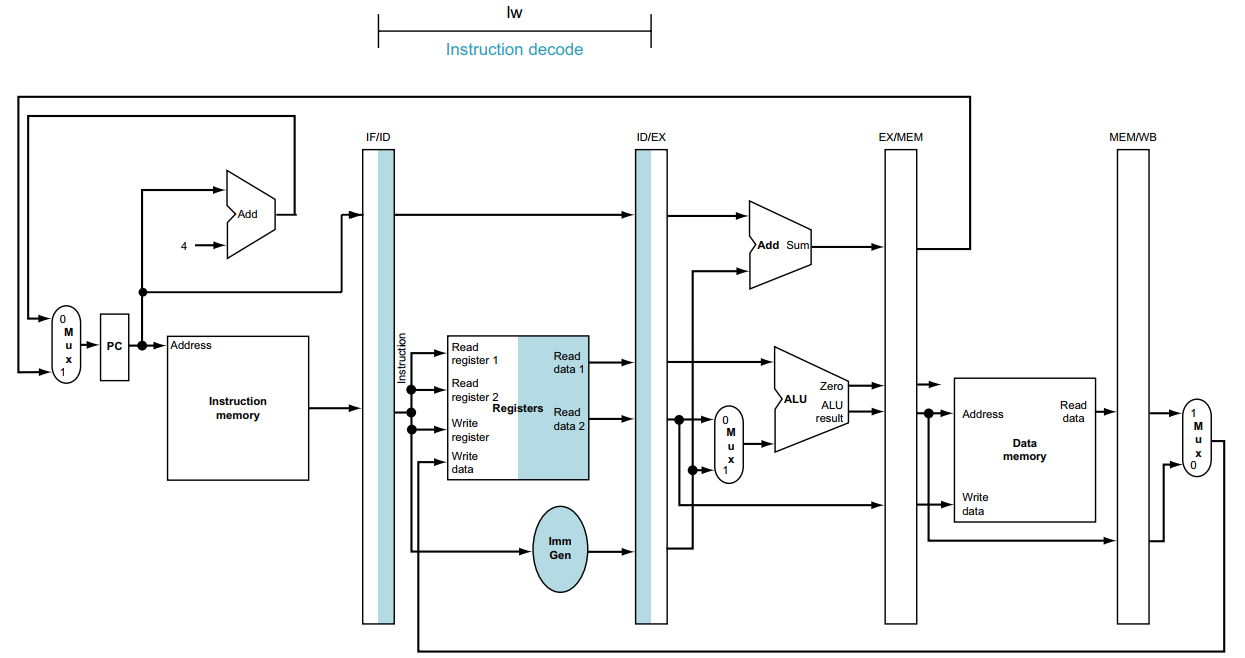
Hình 1.4 Pipeline Datapath

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

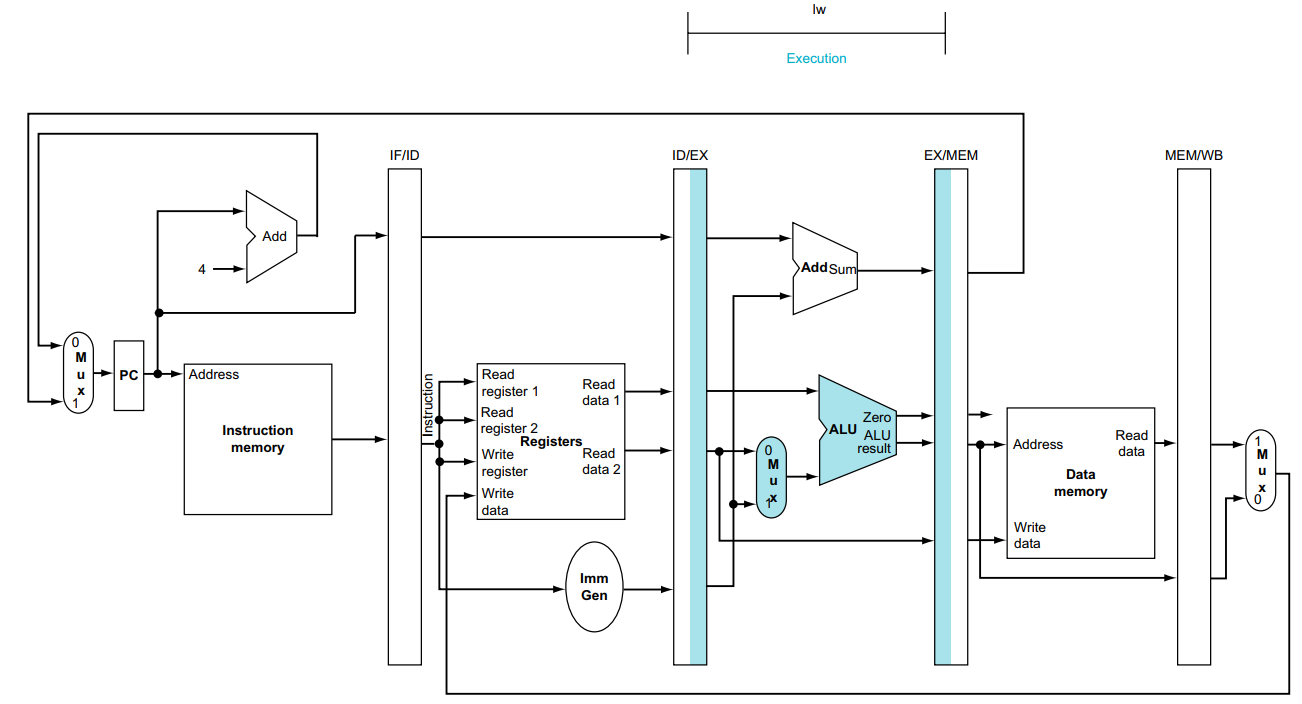
1. *Instruction Fetch*: Hình 1.5 cho thấy lệnh được đọc từ bộ nhớ bằng địa chỉ trong PC và sau đó được đặt trong thanh ghi pipeline IF/ID. Địa chỉ PC được tăng thêm 4 và sau đó được ghi lại vào PC để sẵn sàng cho chu kỳ xung nhịp tiếp theo. PC này cũng được lưu trong thanh ghi pipeline IF/ID trong trường 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.



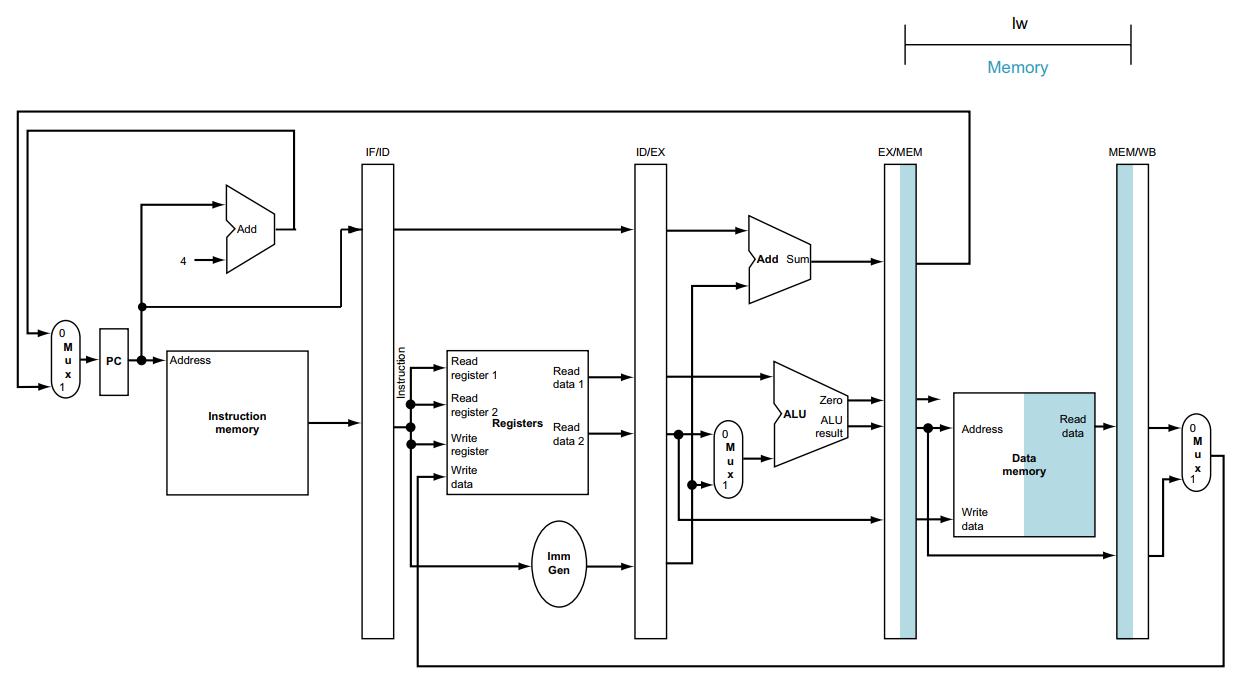
Hình 1.5 Giai đoạn đầu tiên IF của lệnh lw



Hình 1.6 Giai đoạn ID của lệnh lw



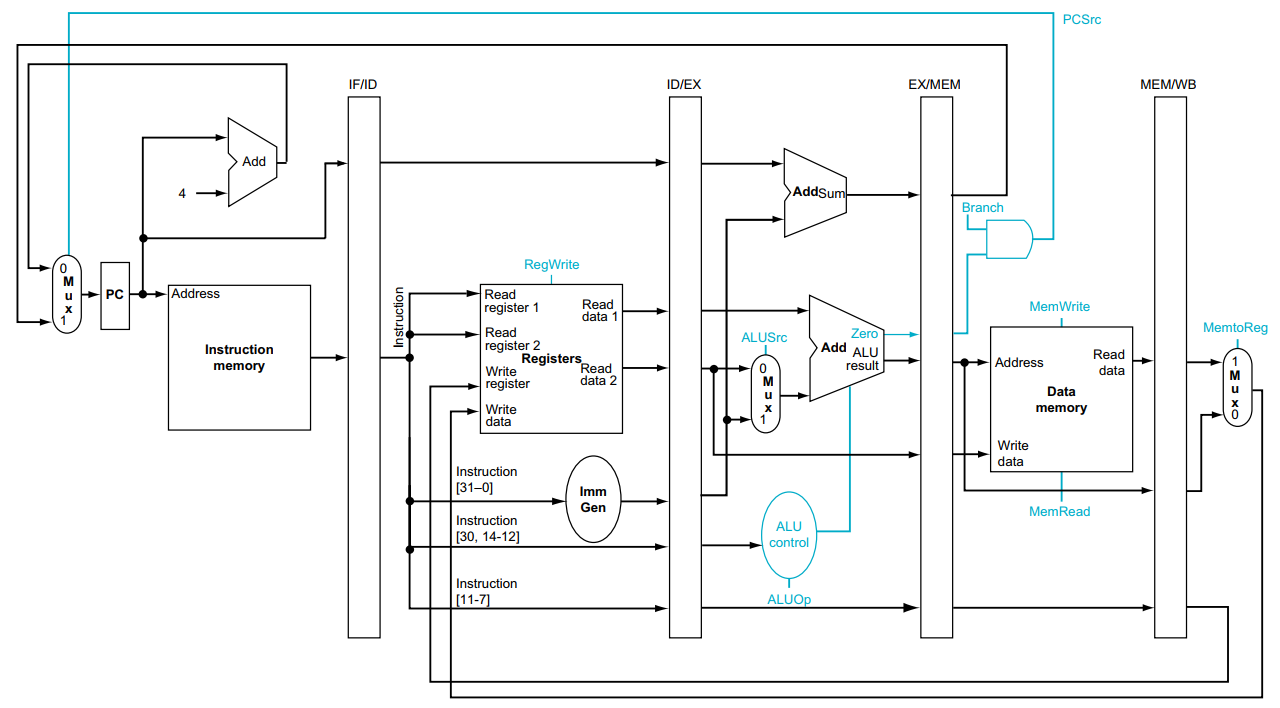
Hình 1.7 Giai đoạn EX của lệnh lw



Hình 1.8 Giai đoạn MEM của lw

## Pipelined Control

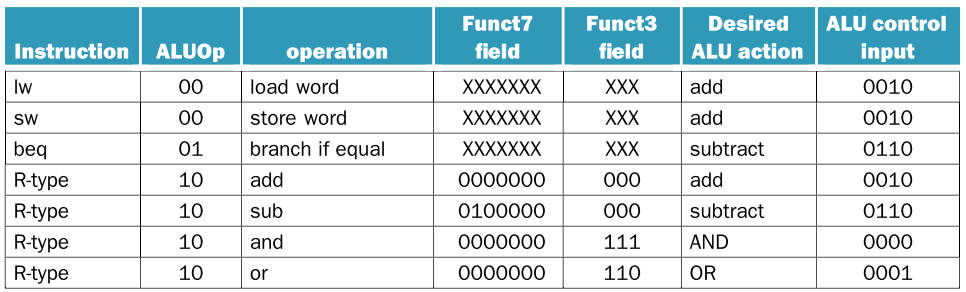
Như trường hợp thực hiện chu kỳ đơn, giả định rằng PC được ghi trên mỗi chu kỳ đồng hồ, do đó không có tín hiệu ghi riêng cho PC. Theo cùng một đối số, không có tín hiệu ghi riêng biệt cho các thanh ghi pipeline (IF/ID, ID/EX, EX/MEM và MEM/WB) vì các thanh ghi pipeline cũng được ghi trong mỗi chu kỳ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



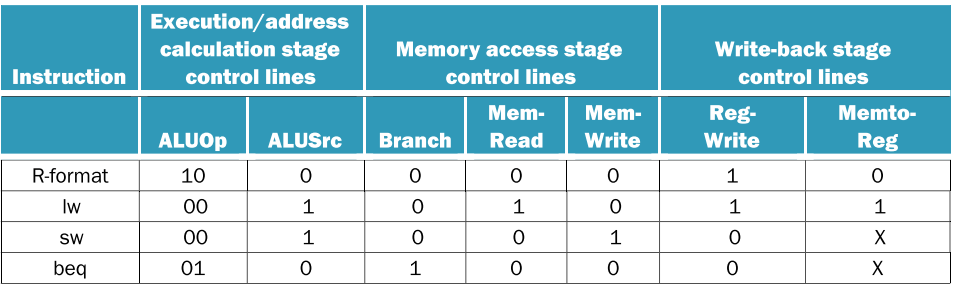
Hình 1.9 Pipelined datapath với các đường tín hiệu điều khiển

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.



Hình 1.10 ALU control bits



Hình 1.11 Đường điều khiển

## 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:

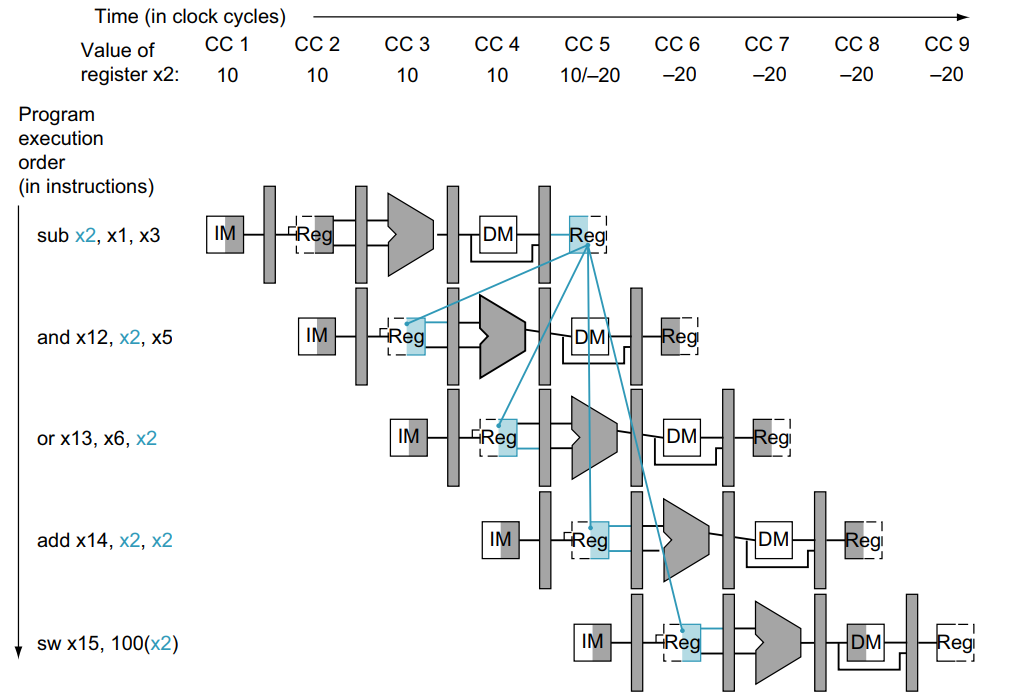
* Structural hazard: Tài nguyên yêu cầu đang bận (ví dụ: cần trong nhiều giai đoạn)
* Data hazard: Sự phụ thuộc dữ liệu giữa các câu lệnh, cần đợi lệnh trước đó hoàn thành việc đọc ghi dữ liệu của nó.
* Control hazard: Luồng thực hiện phụ thuộc vào lệnh trước đó.

### 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.

### Data Hazard

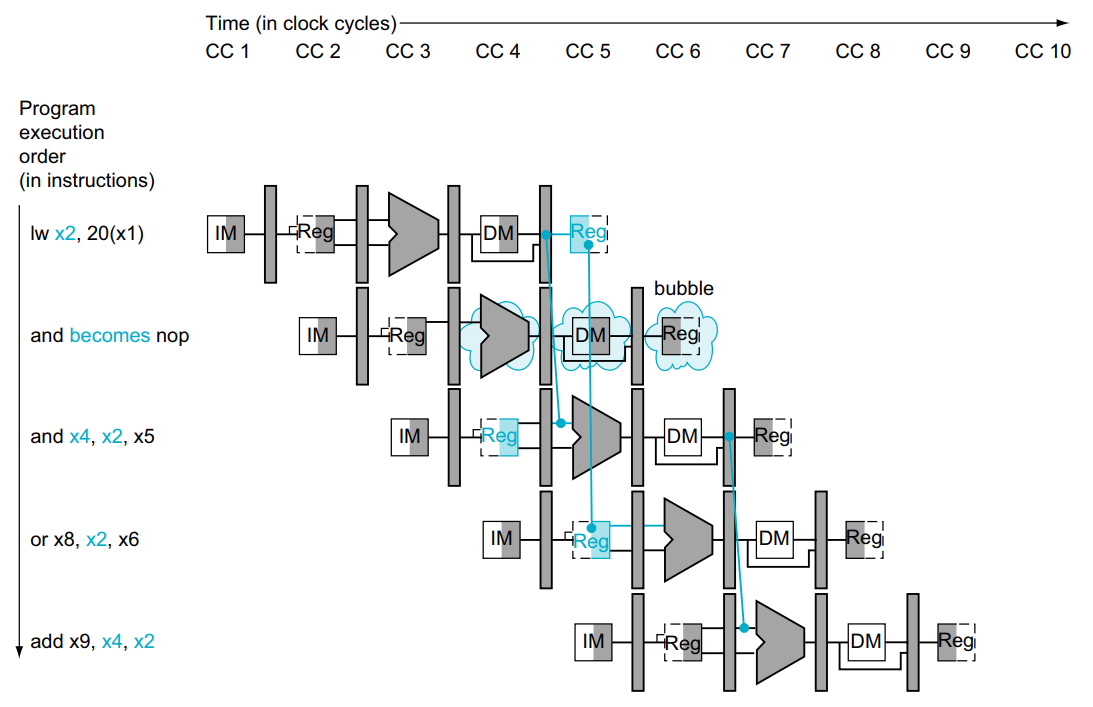
Data hazard thể hiện sự phụ thuộc dữ liệu giữa các câu lệnh trong pipeline, câu lệnh trước cần hoàn thành việc đọc ghi dữ liệu của nó. Hình 1.12 dưới đây thể hiện sử phụ thuộc giữa các câu lệnh trong pipeline.



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 ảnh hưởng không làm gì và các giai đoạn tiếp theo thực hiện 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 hướng dẫn đã tìm 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: nops. Cụ thể phương pháp stall và forwarding được mô tả trên Hình 1.13 dưới đây.

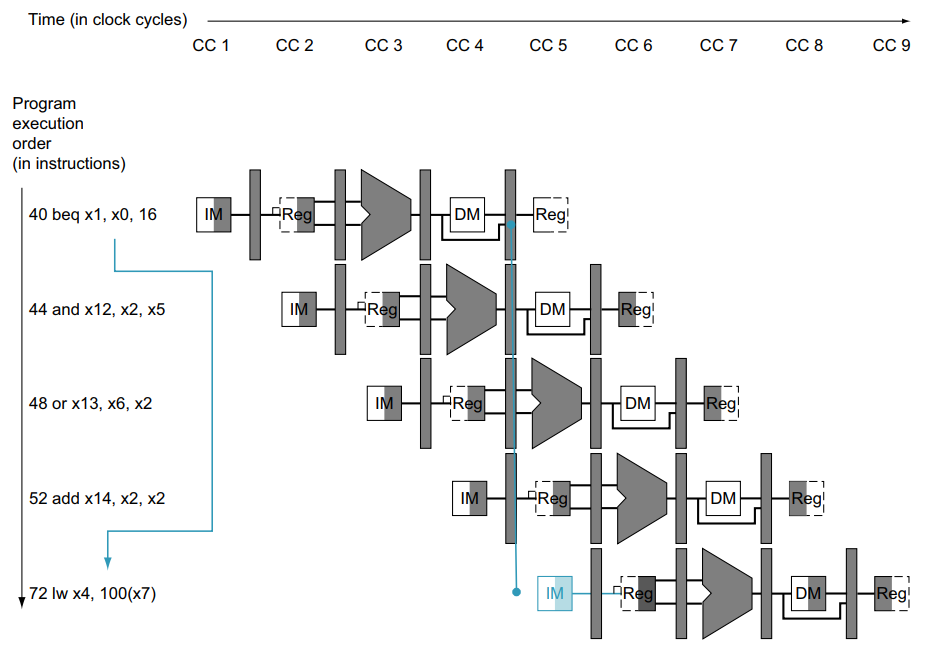


Hình 1.13 Stall và forwarding trong pipeline

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.

### 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

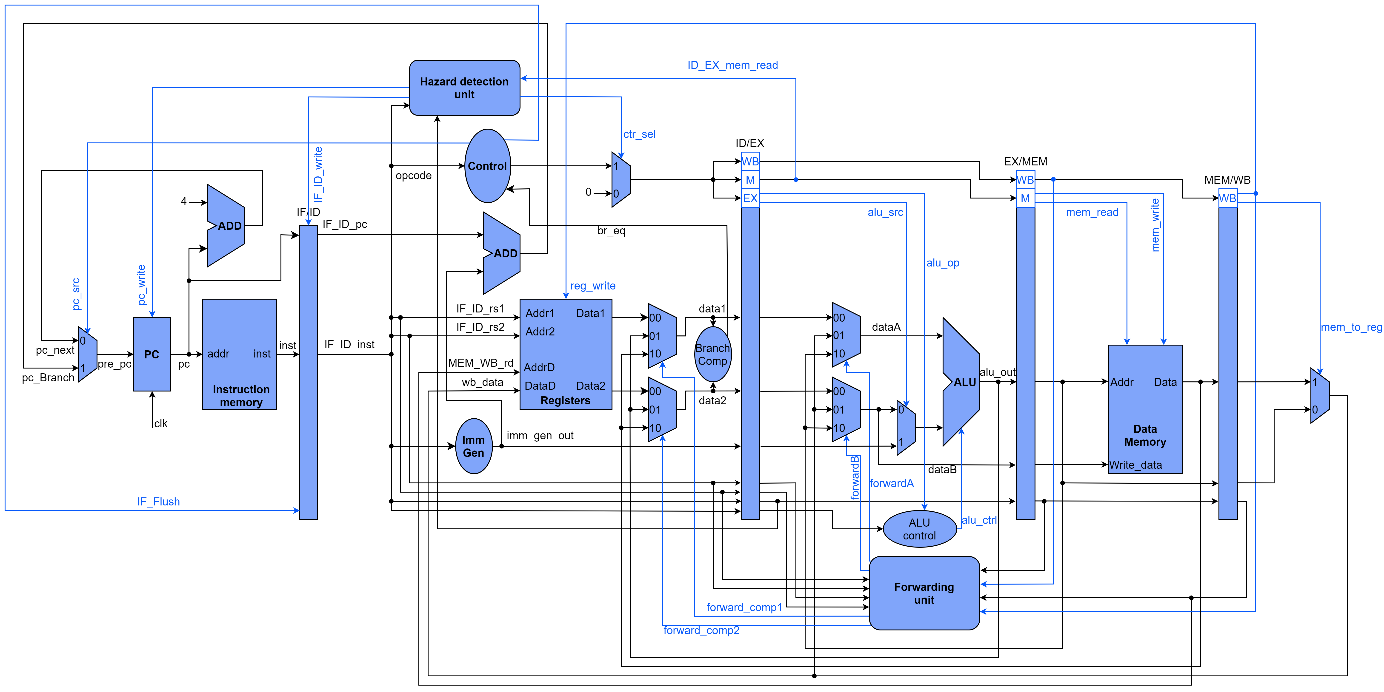
Một cách để cải thiện hiệu suất của nhánh có điều kiện là giảm chi phí của nhánh được lấy. Chúng ta đã giả định rằng PC tiếp theo cho một nhánh được chọn trong giai đoạn MEM, nhưng nếu chúng ta di chuyển việc thực thi nhánh có điều kiện sớm hơn trong pipeline, thì cần phải xóa ít lệnh hơn. Chuyển quyết định nhánh lên đòi hỏi hai hành động xảy ra trước đó: tính toán địa chỉ mục tiêu nhánh và đánh giá quyết định nhánh. Phần dễ dàng của sự thay đổi này là tăng tính toán địa chỉ nhánh. 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 bản thân 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.

# ĐẶC TẢ THÔNG SỐ KĨ THUẬT (SPECIFICATION)

## RISC-V pipeline architecture

### Architecture



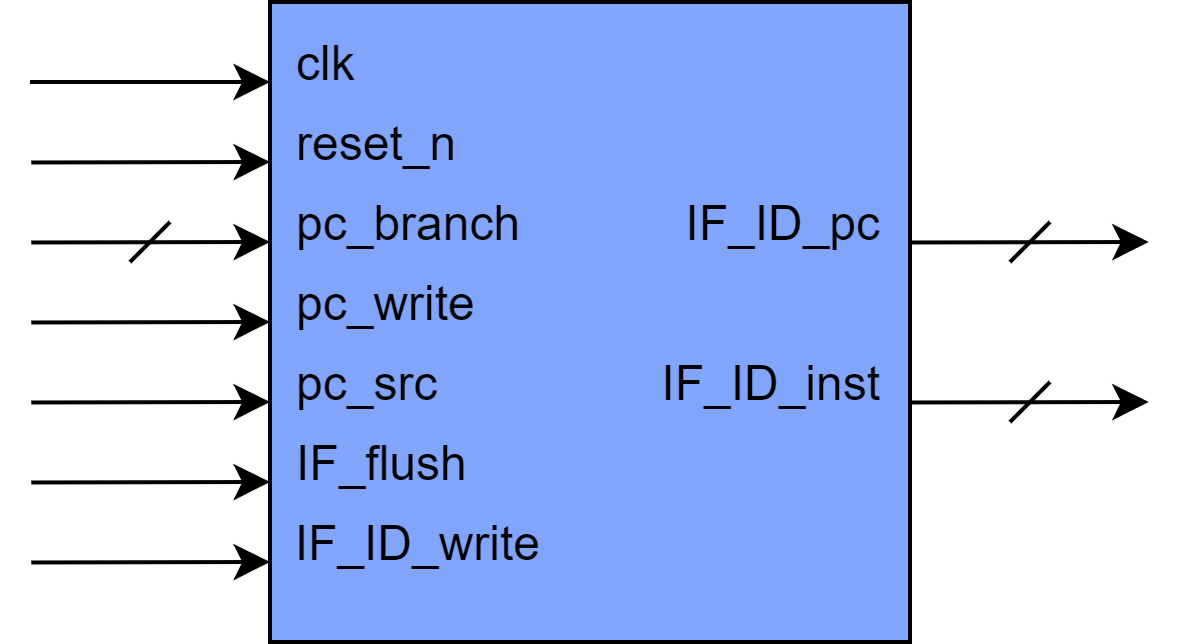
Hình 2.1 riscv\_pipeline architecture

### 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 |

## Module Intruction Fetch (instruction\_fetch)



Hình 2.2 instruction\_fetch block diagram

### Interface signals

Bảng 2.2 instruction\_decode port description

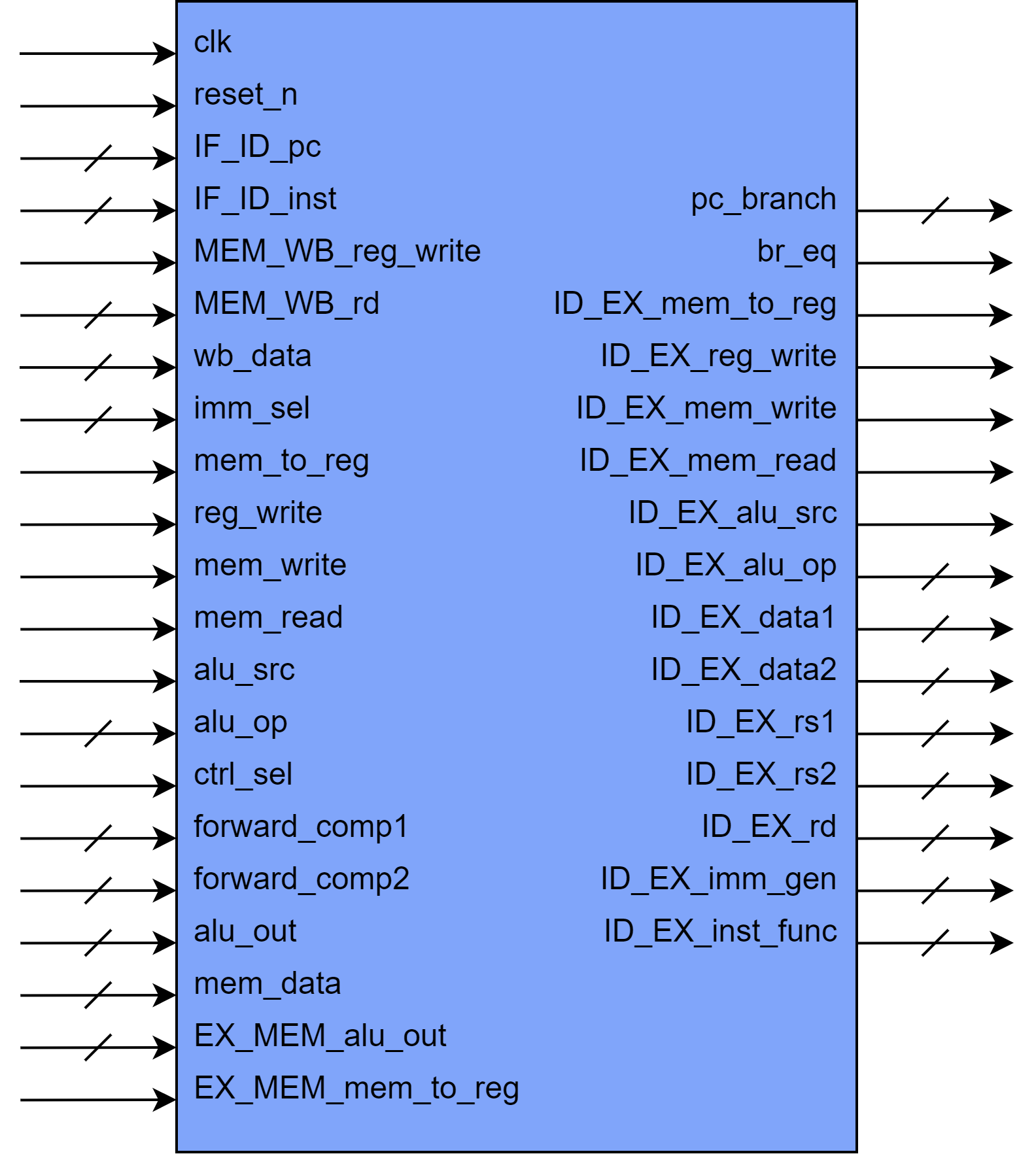
|  |  |  |  |
| --- | --- | --- | --- |
| **Signal name** | **Width** | **Input/Output** | **Description** |
| **clk** | 1 | Input | Tín hiệu xung đồng hồ |
| **reset\_n** | 1 | Input | Tín hiệu reset tích cực mức thấp |
| **pc\_branch** | 32 | Input | Giá trị PC cần nhảy đến khi có lệnh rẽ nhánh |
| **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 |

### 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.

## Module Instruction Decode (instruction\_decode)



Hình 2.3 instruction\_decode block diagram

### 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 cần nhảy đến khi có lệnh rẽ nhánh |
| **IF\_ID\_inst** | 32 | Input | Cho phép thay đổi đầu ra PC |
| **MEM\_WB\_reg\_write** | 1 | Input | Cho phép chọn PC+4 hoặc PC\_branch |
| **MEM\_WB\_rd** | 5 | Input | Cho phép xóa thanh ghi IF\_ID |
| **wb\_data** | 32 | Input | Cho phép thanh ghi IF\_ID ghi dữ liệu mới |
| **imm\_sel** | 3 | Input | Cho biết giá trị PC |
| **mem\_to\_reg** | 1 | Input | Cho biết mã lệnh tương ứng với PC |
| **reg\_write** | 1 | Input |  |
| **mem\_write** | 1 | Input |  |
| **mem\_read** | 1 | Input |  |
| **alu\_src** | 1 | Input |  |
| **alu\_op** | 2 | Input |  |
| **ctrl\_sel** | 1 | Input |  |
| **forward\_comp1** | 2 | Input |  |
| **forward\_comp2** | 2 | Input |  |
| **alu\_out** | 32 | Input |  |
| **mem\_data** | 32 | Input |  |
| **EX\_MEM\_alu\_out** | 32 | Input |  |
| **EM\_MEM\_mem\_to\_reg** | 1 | Input |  |
| **pc\_branch** | 32 | Output |  |
| **br\_eq** | 1 | Output |  |
| **ID\_EX\_mem\_to\_reg** | 1 | Output |  |
| **ID\_EX\_reg\_write** | 1 | Output |  |
| **ID\_EX\_mem\_write** | 1 | Output |  |
| **ID\_EX\_mem\_read** | 1 | Output |  |
| **ID\_EX\_alu\_src** | 1 | Output |  |
| **ID\_EX\_alu\_op** | 2 | Output |  |
| **ID\_EX\_data1** | 32 | Output |  |
| **ID\_EX\_data2** | 32 | Output |  |
| **ID\_EX\_rs1** | 5 | Output |  |
| **ID\_EX\_rs2** | 5 | Output |  |
| **ID\_EX\_rd** | 5 | Output |  |
| **ID\_EX\_imm\_gen** | 32 | Output |  |
| **ID\_EX\_inst\_func** | 4 | Output |  |

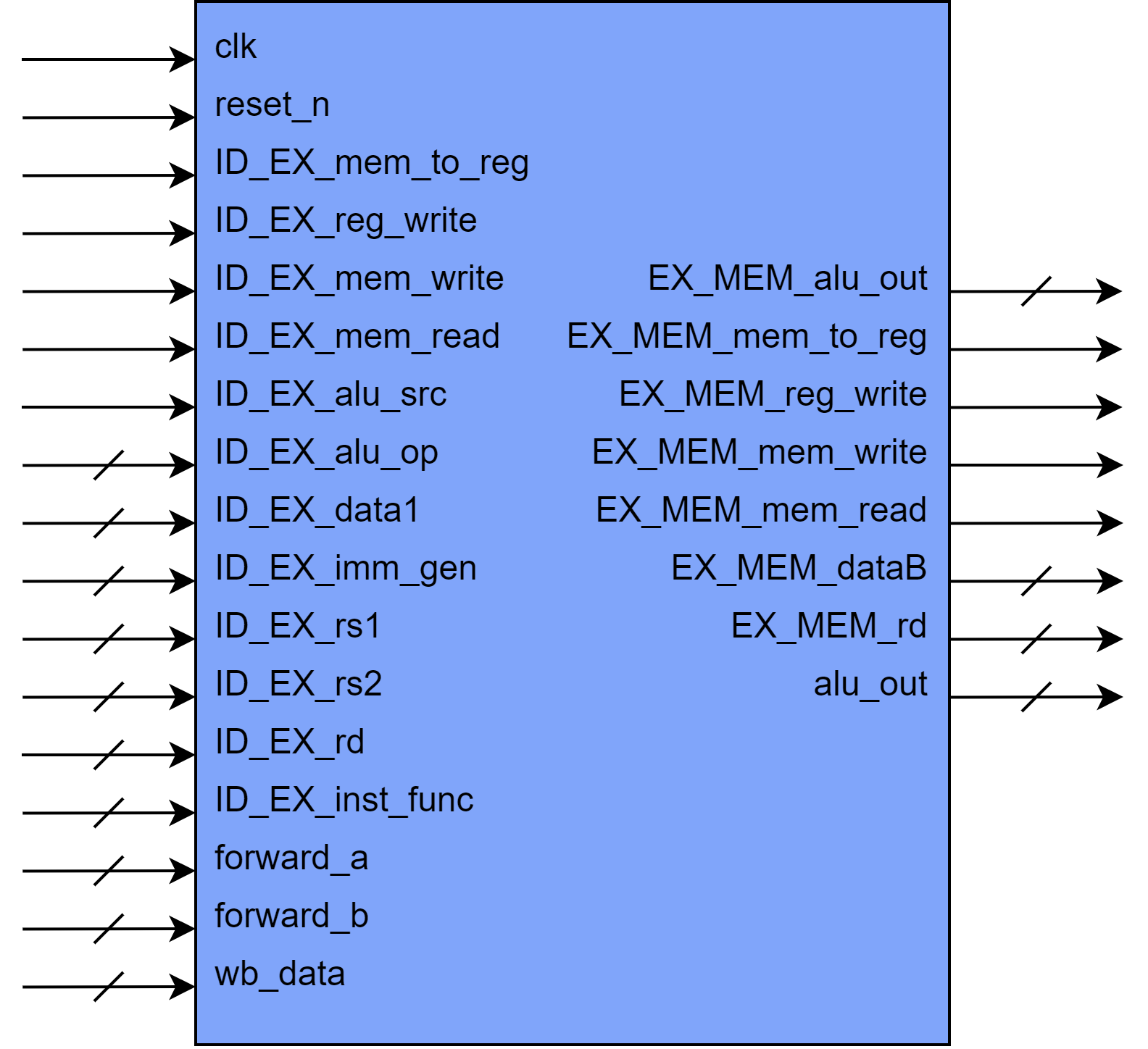
### Function description

Khối controller thực hiện nhiệm vụ điều khiển quá trình thực hiện lệnh trong Risc-V dựa vào các tín hiều đầu vào và đưa ra đầu ra tương ứng.

Dưới đây là bảng sự thật của khối Controller để điều khiển quá trình thực hiện lệnh.

Dưới đây là hình mô tả timming tổng quát của khối Control logic, do hoạt động khá đơn giản bằng kiến trúc ROM-based nên khối hoạt động duy nhất trong một chu kì clk.

## Module Excute (execute)



Hình 2.4 execute block diagram

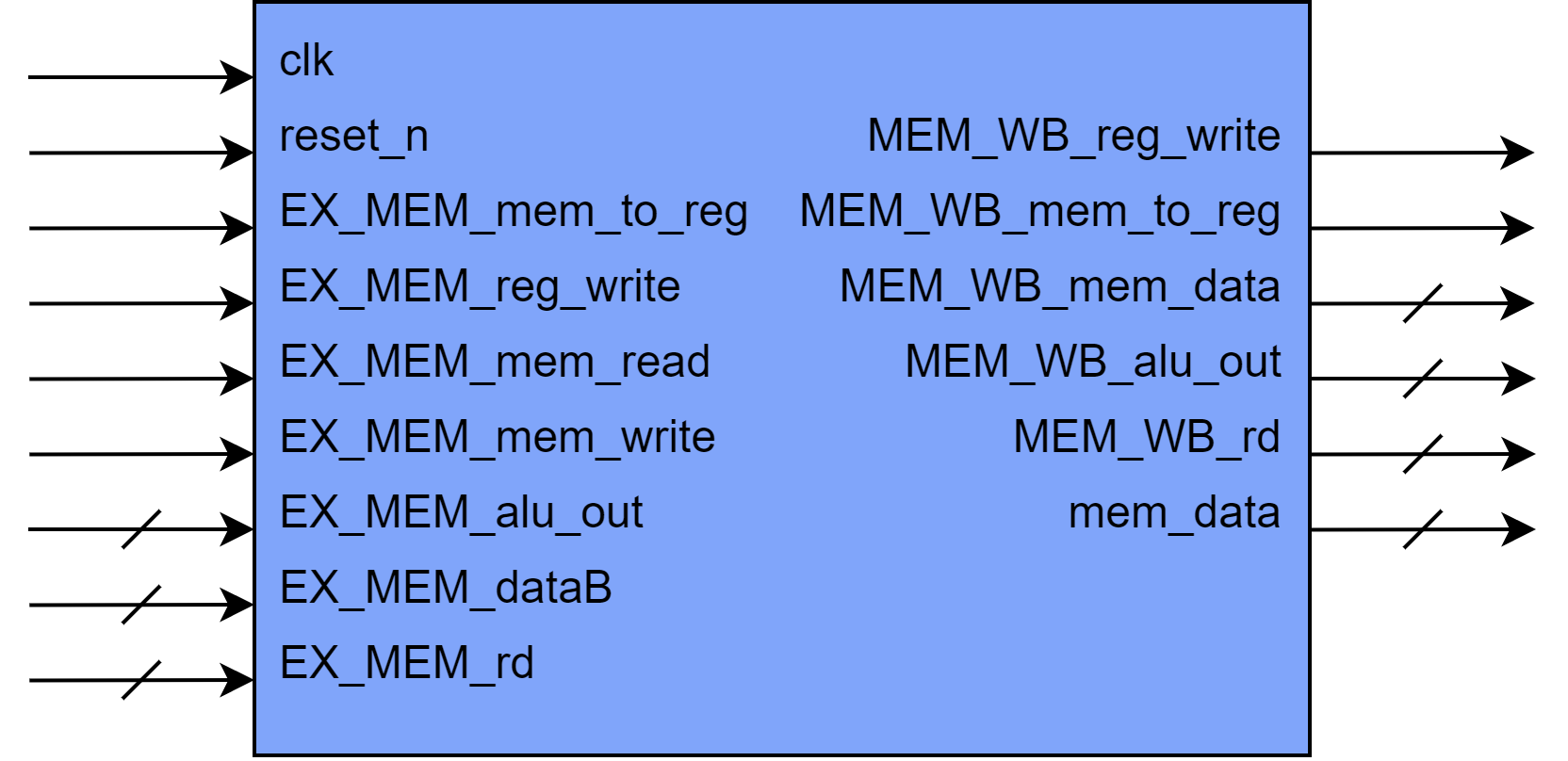
### Interface signals

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 |  |
| **ID\_EX\_reg\_write** | 1 | Input |  |
| **ID\_EX\_mem\_write** | 1 | Input |  |
| **ID\_EX\_mem\_read** | 1 | Input |  |
| **ID\_EX\_alu\_src** | 1 | Input |  |
| **ID\_EX\_alu\_op** | 2 | Input |  |
| **ID\_EX\_data1** | 32 | Input |  |
| **ID\_EX\_data2** | 32 | Input |  |
| **ID\_EX\_imm\_gen** | 32 | Input |  |
| **ID\_EX\_rs1** | 5 | Input |  |
| **ID\_EX\_rs2** | 5 | Input |  |
| **ID\_EX\_rd** | 5 | Input |  |
| **ID\_EX\_inst\_func** | 4 | Input |  |
| **forward\_a** | 2 | Input |  |
| **forward\_b** | 2 | Input |  |
| **wb\_data** | 32 | Input |  |
| **EX\_MEM\_alu\_out** | 32 | Output |  |
| **EX\_MEM\_mem\_to\_reg** | 1 | Output |  |
| **EX\_MEM\_reg\_write** | 1 | Output |  |
| **EX\_MEM\_mem\_write** | 1 | Output |  |
| **EX\_MEM\_mem\_read** | 1 | Output |  |
| **EX\_MEM\_dataB** | 32 | Output |  |
| **EX\_MEM\_rd** | 5 | Output |  |
| **alu\_out** | 32 | Output |  |

### Function description

## Module Memory Access (memory\_access)



Hình 2.5 memory\_access block diagram

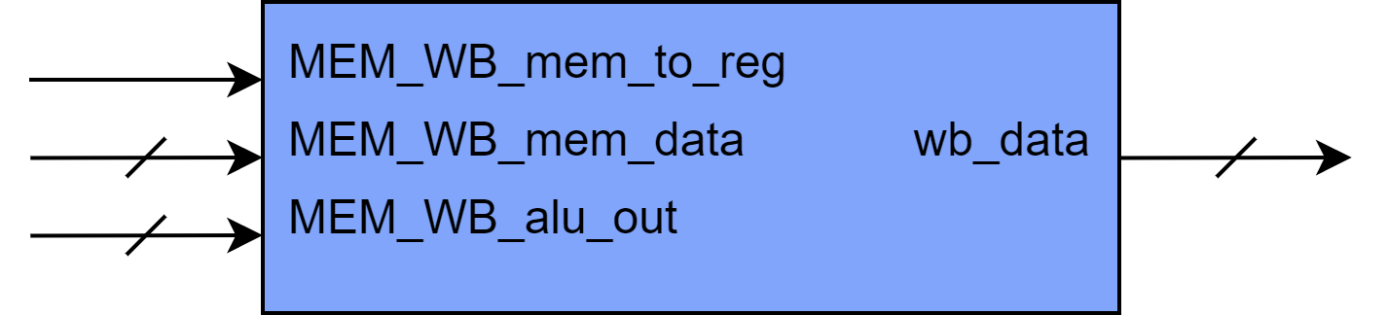
### Interface signals

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 |  |
| **EX\_MEM\_reg\_write** | 1 | Input |  |
| **EX\_MEM\_mem\_read** | 1 | Input |  |
| **EX\_MEM\_mem\_write** | 1 | Input |  |
| **EX\_MEM\_alu\_out** | 32 | Input |  |
| **EX\_MEM\_dataB** | 32 | Input |  |
| **EX\_MEM\_rd** | 5 | Input |  |
| **MEM\_WB\_reg\_write** | 1 | Output |  |
| **MEM\_WB\_mem\_to\_reg** | 1 | Output |  |
| **MEM\_WB\_mem\_data** | 32 | Output |  |
| **MEM\_WB\_alu\_out** | 32 | Output |  |
| **MEM\_WB\_rd** | 5 | Output |  |
| **mem\_data** | 32 | Output |  |

### Function description

## Module Register Write (register\_write)



Hình 2.6 register\_write block diagram

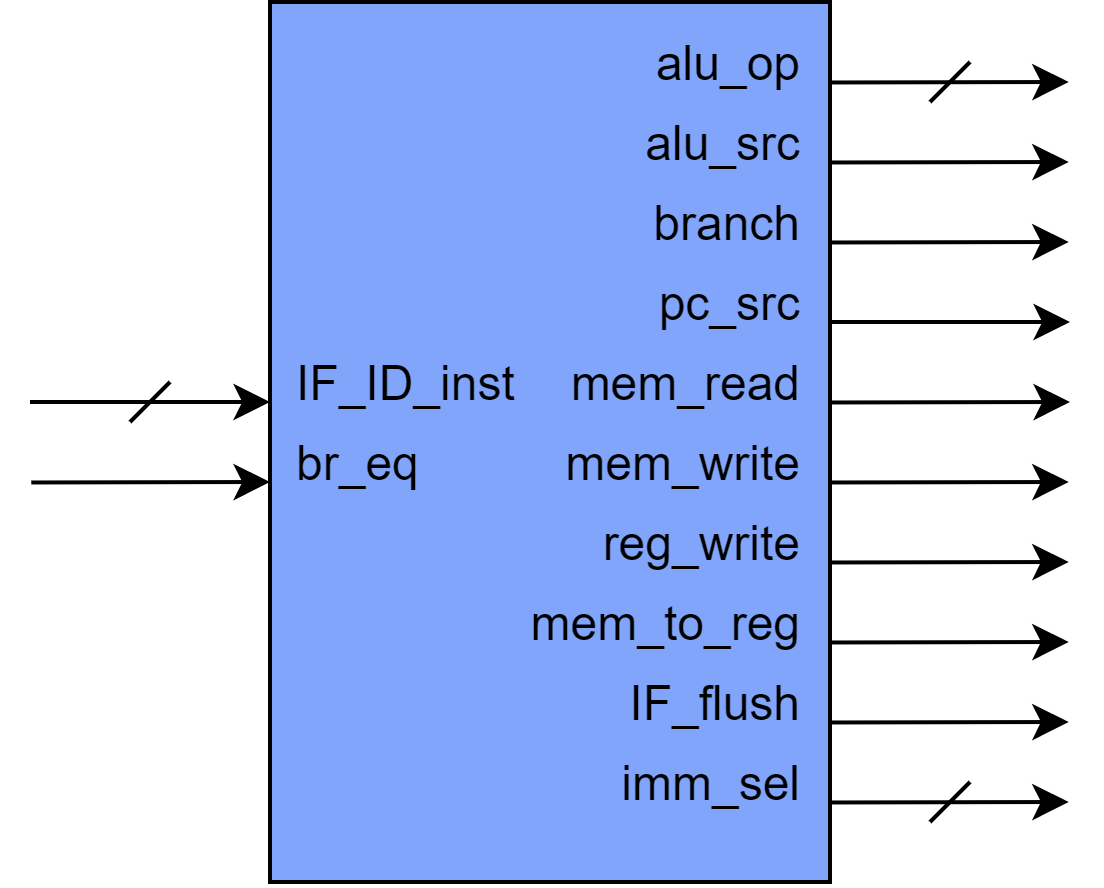
### Interface signals

Bảng 2.6 register\_write port description

| **Signal name** | **Width** | **Input/Output** | **Description** |
| --- | --- | --- | --- |
| **MEM\_WB\_mem\_to\_reg** | 1 | Input |  |
| **MEM\_WB\_mem\_data** | 32 | Input |  |
| **MEM\_WB\_alu\_out** | 32 | Input |  |
| **wb\_data** | 32 | Output |  |

### Function description

## Module Control (control)



Hình 2.7 control block diagram

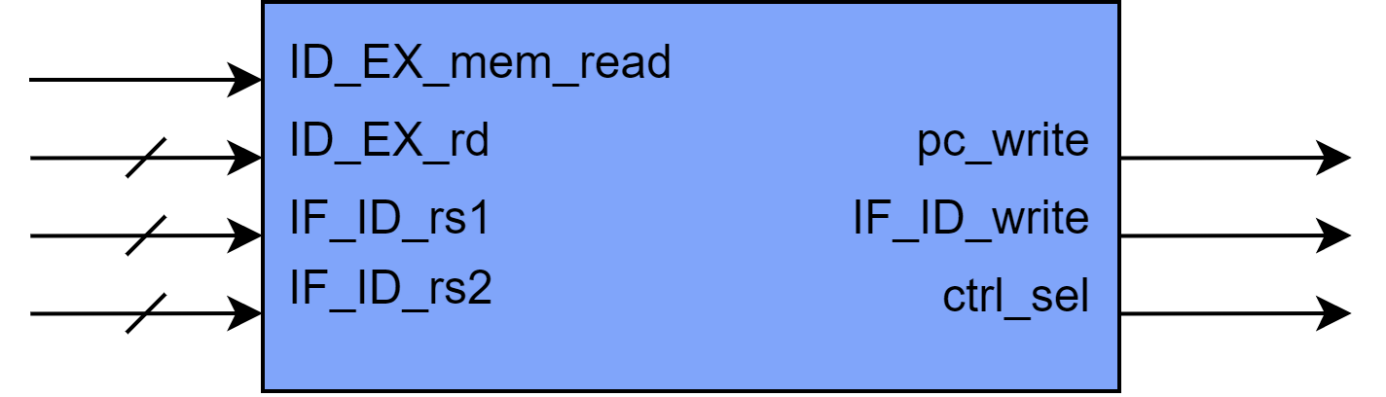
### Interface signals

Bảng 2.7 control port description

| **Signal name** | **Width** | **Input/Output** | **Description** |
| --- | --- | --- | --- |
| **IF\_ID\_inst** | 32 | Input |  |
| **br\_eq** | 1 | Input |  |
| **alu\_op** | 2 | Output |  |
| **alu\_src** | 1 | Output |  |
| **branch** | 1 | Output |  |
| **pc\_src** | 1 | Output |  |
| **mem\_read** | 1 | Output |  |
| **mem\_write** | 1 | Output |  |
| **reg\_write** | 1 | Output |  |
| **mem\_to\_reg** | 1 | Output |  |
| **IF\_flush** | 1 | Output |  |
| **imm\_sel** | 3 | Output |  |

### Function description

## Module Hazard Detection Unit (hazard\_detection\_unit)



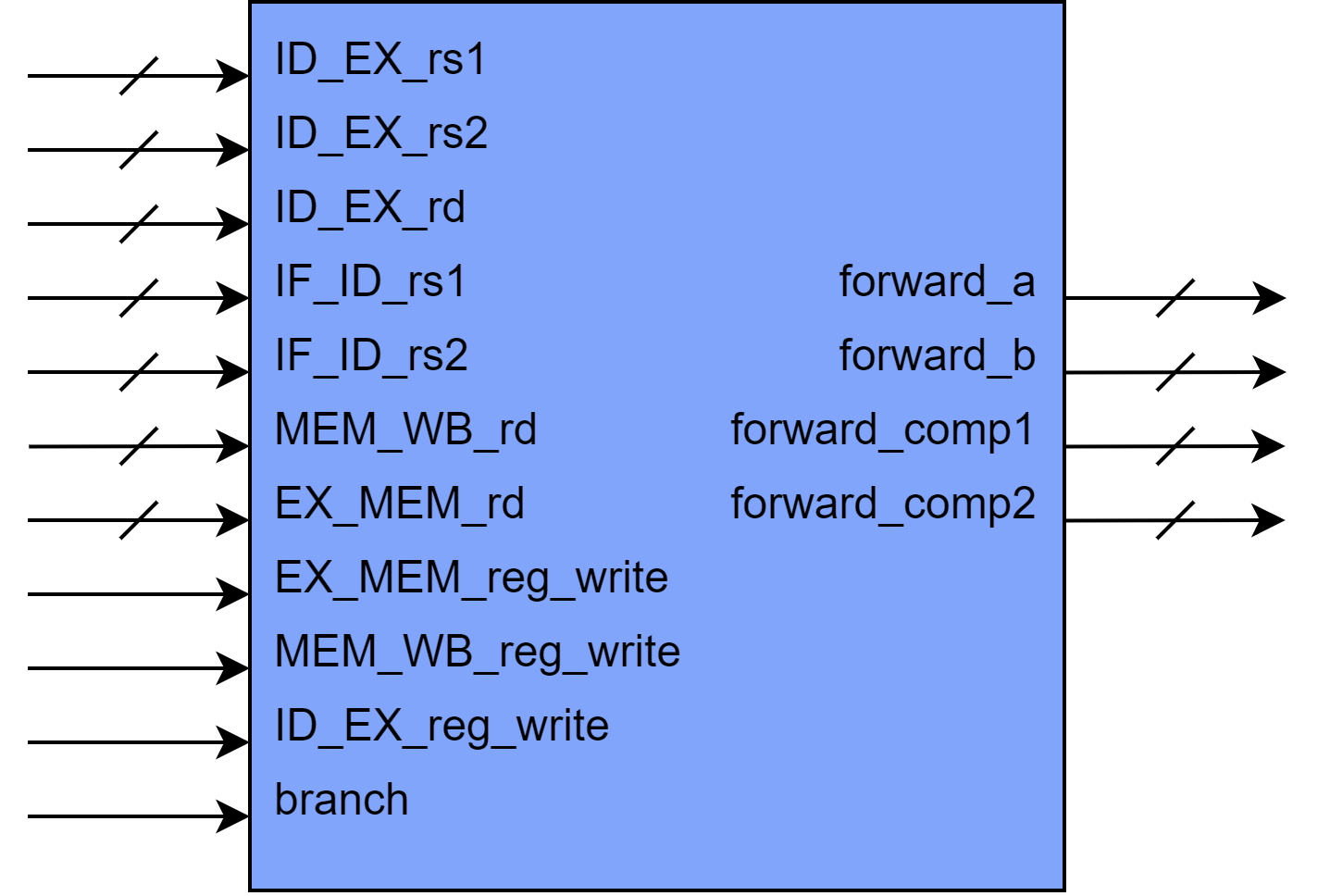
Hình 2.8 hazard\_detection\_unit block diagram

### Interface signals

| **Signal name** | **Width** | **Input/Output** | **Description** |
| --- | --- | --- | --- |
| **ID\_EX\_mem\_read** | 1 | Input |  |
| **ID\_EX\_rd** | 5 | Input |  |
| **IF\_ID\_rs1** | 5 | Input |  |
| **IF\_ID\_rs2** | 5 | Input |  |
| **pc\_write** | 1 | Output |  |
| **IF\_ID\_write** | 1 | Output |  |
| **ctrl\_sel** | 1 | Output |  |

### Function description

## Module Forwarding Unit (forwarding\_unit)



Hình 2.9 forwarding\_unit block diagram

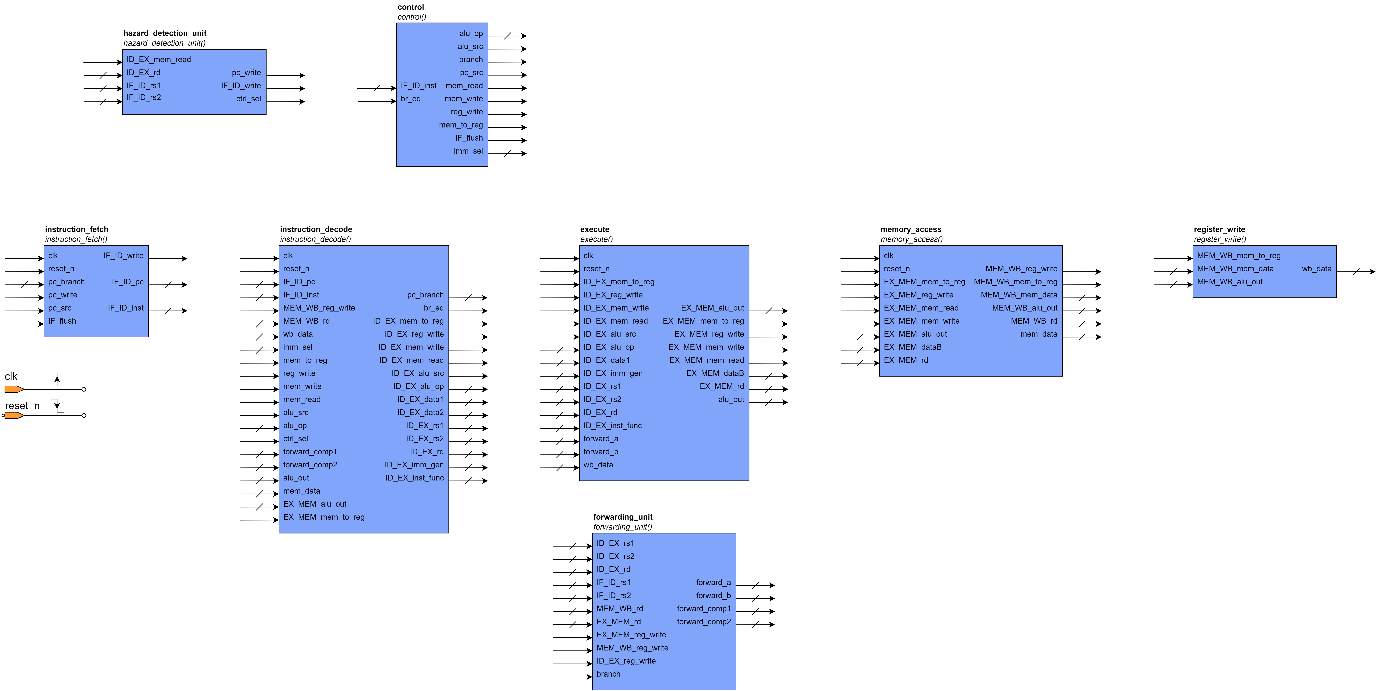
### Interface signals

Bảng 2.8 forwarding\_unit port description

| **Signal name** | **Width** | **Input/Output** | **Description** |
| --- | --- | --- | --- |
| **ID\_EX\_rs1** | 5 | Input |  |
| **ID\_EX\_rs2** | 5 | Input |  |
| **ID\_EX\_rd** | 5 | Input |  |
| **IF\_ID\_rs1** | 5 | Input |  |
| **IF\_ID\_rs2** | 5 | Input |  |
| **MEM\_WB\_rd** | 5 | Input |  |
| **EX\_MEM\_rd** | 5 | Input |  |
| **EX\_MEM\_reg\_write** | 1 | Input |  |
| **MEM\_WB\_reg\_write** | 1 | Input |  |
| **ID\_EX\_reg\_write** | 1 | Input |  |
| **branch** | 1 | Input |  |
| **forward\_a** | 2 | Output |  |
| **forward\_b** | 2 | Output |  |
| **forward\_comp1** | 2 | Output |  |
| **forward\_comp2** | 2 | Output |  |

### Function description

## Module Top (riscv\_pipeline\_top)



Hình 2.10 riscv\_pipeline\_top block diagram

### Interface signals

Bảng 2.9 riscv\_pipeline\_top port description

| **Signal name** | **Width** | **Input/Output** | **Description** |
| --- | --- | --- | --- |
| **clk** | 5 | Input |  |
| **reset\_n** | 5 | Input |  |

### Function description

# KIỂM THỬ (VERIFICATION)

## Kế hoạch kiểm thử - khối Control Logic

### 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

### Kích thích đầu vào – Stimulus

* Tạo sẵn các mã lệnh test bao gồm
  + add x8, x12, x14 🡪 0000000 01110 01100 000 01000 0110011
  + sub x10, x12, x11 🡪 0100000 01011 01100 000 01010 0110011
  + addi x15, x1, -50 🡪 111111001110 00001 000 01111 0010011
  + lw x14, 8(x2) 🡪 000000001000 00010 010 01110 0000011
  + sw x14, 8(x2) 🡪 000000001110 00010 010 01000 0100011
  + beq x19, x10, offset 🡪 0 000000 01010 10011 000 1000 0 1100011
* Sinh các đầu vào bằng cách gán giá trị trong testbench

### Kiểm tra đầu ra

Đầu ra tín hiệu điều khiển có giá trị đúng với giá trị trong Hình 2.1 Control logic Truth Table.

### Mô phỏng kiểm thử thiết kế

Hình 3.1 Đầu ra mô phỏng trên ModelSim

Đầu ra tín hiệu điều khiển của khối Control logic mô phỏng cho lệnh add đúng với giá trị trong Hình 2.1.

## Kế hoạch kiểm thử - khối Datapath

### 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

### Kích thích đầu vào – Stimulus

* Tạo sẵn các mã lệnh test bao gồm
  + add x8, x12, x14 🡪 0000000 01110 01100 000 01000 0110011
  + sub x10, x12, x11 🡪 0100000 01011 01100 000 01010 0110011
  + addi x15, x1, -50 🡪 111111001110 00001 000 01111 0010011
  + lw x14, 8(x2) 🡪 000000001000 00010 010 01110 0000011
  + sw x14, 8(x2) 🡪 000000001110 00010 010 01000 0100011
  + beq x19, x10, offset 🡪 0 000000 01010 10011 000 1000 0 1100011
* Sinh các đầu vào bằng cách gán giá trị trong testbench

### Mô phỏng kiểm thử thiết kế

Hình 3.2 Kết quả mô phỏng kiểm thử khối top\_datapath

**Nhận xét:**

* Các tín hiệu **PC**, **inst\_tmp** lên ngay cùng với sườn dương xung **clk** (trừ **clk** đầu tiên)
* **DataA**, **DataB** nhận dữ liệu ngay sau 1 chu kì **clk**
* ALU trả kết quả và đưa ra **alu** sau khi tính toán xong
* **DataD** nhận kết quả và ghi và thanh ghi sau nửa chu kì **clk** (ở sườn âm clk tiếp theo)

# KẾT LUẬN

Báo cáo này đã trình bày kiến trúc Datapath và Control logic của một RISCV32I processor đơn giản và tiến hành triển khai mô phỏng kiểm thử trên phần mềm ModelSim. Cho ra kết quả hoạt động đúng với yêu cầu vào ra. Tuy nhiên kiến trúc được trình bày trong báo cáo mới chỉ mô phỏng và chạy được các lệnh đơn giản của kiến trúc tập lệnh RISCV gồm: add, sub, lw, sw và các phép toán logic giữa các thanh ghi với nhau. Hai khối Datapath và Control logic được triển khai độc lập, chưa có sự ghép nối hoàn thiện thành một RISCV processor. Hoạt động của kiến trúc mới chỉ dừng lạnh ở single-cycle based, tuy nhiên vẫn còn có thể tối ưu hoá hơn nữa về mặt thời gian, tăng tốc độ và hiệu quả xử lí của processor, công việc này đòi hỏi phải cần một kiến trúc phức tạp hơn mà báo cáo này chưa trình bày. Trong tương lai nhóm 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: 16/5/2021
2. Slide: RISC – V 2021. PGS TS. Nguyễn Đức Minh
3. Computer Organization and Design Risc – V The Hardware Software Interface by David A. Patterson and John L. Hennessy
4. Code đính kèm: <https://github.com/lampn27022k/RISC-V---Datapath-and-Control-Logic.git>