|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI  **VIỆN ĐIỆN TỬ - VIỄN THÔNG**  logo_128  **BÁO CÁO BÀI TẬP LỚN**  **KIẾN TRÚC MÁY TÍNH**  **Đề tài:**  **Thiết kế bộ xử lý RISC-V Pipeline bằng ngôn ngữ Verilog và mô phỏng chạy chương trình bằng thuật toán sắp xếp nổi bọt**   |  |  | | --- | --- | | **Giảng viên hướng dẫn:** | **PGS.TS Nguyễn Đức Minh** | | **Nhóm sinh viên thực hiện:** | **Nhóm 10** |  |  |  | | --- | --- | | Phạm Quang Anh | 20182359 | | Trần Hồng Nhung | 20182713 | | Nguyễn Đức Quang | 20182736 |   **Hà Nội, 08-2022** |

**LỜI NÓI ĐẦU**

Kiến trúc RISC-V được phát triển tại Đại học California ở Berkeley bởi Yunsup Lee, Krste Asanović, David A. Patterson và Andrew Waterman. Nỗ lực này tiếp nối bốn dự án thiết kế kiến ​​trúc RISC lớn trước đây tại UC Berkeley, dẫn đến cái tên RISC-V, trong đó V đại diện cho số năm La Mã. RISC-V là một bộ kiến trúc bộ xử lý, là một mã nguồn mở cho một bộ xử lý tập lệnh rút gọn. Làm việc với RISC-V có thể biết được việc xử lý một tập lệnh của bộ xử lý diễn ra như thế nào với cấu trúc cụ thể. Và để có thể hiểu và nắm rõ hơn nữa, đề tài “Thiết kế bộ xử lý RISC-V pipeline bằng ngôn ngữ Verilog và mô phỏng chạy chương trình bằng thuật toán sắp xếp nổi bọt” đã được chọn để nghiên cứu.

Do thời gian thực hiện và kiến thức còn hạn chế nên còn nhiều sai sót trong quá trình thực hiện đề tài, nhóm rất mong được sự bổ sung đóng góp của thầy và các bạn để đề tài có thể hoàn thiện hơn. Nhóm xin chân thành cảm ơn PGS.TS Nguyễn Đức Minh đã tận tình hướng dẫn và giúp đỡ hoàn thành đề tài này cũng như trong quá trình học tập.

Nhóm sinh viên thực hiện

Nhóm 10

**MỤC LỤC**

[DANH MỤC HÌNH ẢNH i](#_Toc112202722)

[DANH MỤC BẢNG BIỂU iii](#_Toc112202723)

[CHƯƠNG 1. GIỚI THIỆU RISC-V PIPELINE 1](#_Toc112202724)

[1.1 Bộ xử lý RISC-V 1](#_Toc112202725)

[1.1.1 Định nghĩa 1](#_Toc112202726)

[1.1.2 Cấu trúc RISC-V 1](#_Toc112202727)

[1.1.3 Tập lệnh RISC-V 2](#_Toc112202728)

[1.1.4 Địa chỉ RISC-V 2](#_Toc112202729)

[1.2 RISC-V Pipeline 3](#_Toc112202730)

[1.2.1 Khái niệm 3](#_Toc112202731)

[1.2.2 Pipeline Datapath 4](#_Toc112202732)

[1.2.3 Pipeline Control 8](#_Toc112202733)

[1.3 Hazard trong pipeline 10](#_Toc112202734)

[1.3.1 Structural Hazard 10](#_Toc112202735)

[1.3.2 Data Hazard 10](#_Toc112202736)

[1.3.3 Control Hazard 12](#_Toc112202737)

[1.4 Kết luận 14](#_Toc112202738)

[CHƯƠNG 2. MÔ TẢ KIẾN TRÚC RISC-V PIPELINE 15](#_Toc112202739)

[2.1 Kiến trúc RISC-V pipeline 15](#_Toc112202740)

[2.1.1 Kiến trúc tổng quát 15](#_Toc112202741)

[2.1.2 Interface signals 15](#_Toc112202742)

[2.2 Khối Intruction Fetch (IF) 16](#_Toc112202743)

[2.2.1 Chức năng 17](#_Toc112202744)

[2.2.2 Kiến trúc khối IF 17](#_Toc112202745)

[2.3 Khối Instruction Decode (ID) 18](#_Toc112202746)

[2.3.1 Reg\_file block 19](#_Toc112202747)

[2.3.2 Control\_logic block 20](#_Toc112202748)

[2.3.3 Branch\_comp block 21](#_Toc112202749)

[2.3.4 Imm\_gen block 22](#_Toc112202750)

[2.3.5 Chức năng 22](#_Toc112202751)

[2.3.6 Kiến trúc khối ID 23](#_Toc112202752)

[2.4 Khối Excute (EX) 24](#_Toc112202753)

[2.4.1 Chức năng 25](#_Toc112202754)

[2.4.2 Kiến trúc khối EX 25](#_Toc112202755)

[2.5 Khối Memory Access (MEM) 26](#_Toc112202756)

[2.5.1 Chức năng 27](#_Toc112202757)

[2.5.2 Kiến trúc khối MEM 27](#_Toc112202758)

[2.6 Khối Register Write (WB) 28](#_Toc112202759)

[2.6.1 Chức năng 28](#_Toc112202760)

[2.6.2 Kiến trúc khối Register Write 28](#_Toc112202761)

[2.7 Khối Hazard Detection Unit 29](#_Toc112202762)

[2.8 Khối Forwarding Unit 29](#_Toc112202763)

[2.8.1 Chức năng 30](#_Toc112202764)

[2.9 Kết luận 31](#_Toc112202765)

[CHƯƠNG 3. KẾT QUẢ VÀ ĐÁNH GIÁ 32](#_Toc112202766)

[3.1 Kiểm thử RISC-V 32](#_Toc112202767)

[3.1.1 Các trường hợp kiểm tra kiểm thử 32](#_Toc112202768)

[3.1.2 Kết quả và đánh giá 34](#_Toc112202769)

[3.2 Thuật toán sắp xếp nổi bọt 36](#_Toc112202770)

[3.3 Đánh giá kết quả 38](#_Toc112202771)

[KẾT LUẬN 40](#_Toc112202772)

[TÀI LIỆU THAM KHẢO 41](#_Toc112202773)

DANH MỤC HÌNH ẢNH

[Hình 1.1 Single cycle versus pipelined 3](#_Toc112202663)

[Hình 1.2 Single-cycle datapath 5](#_Toc112202664)

[Hình 1.3 Pipeline Datapath 6](#_Toc112202665)

[Hình 1.4 Giai đoạn IF của lệnh lw 7](#_Toc112202666)

[Hình 1.5 Giai đoạn ID của lệnh lw 7](#_Toc112202667)

[Hình 1.6 Giai đoạn EX của lệnh lw 8](#_Toc112202668)

[Hình 1.7 Giai đoạn MEM của lw 8](#_Toc112202669)

[Hình 1.8 Pipeline datapath với các đường tín hiệu điều khiển 9](#_Toc112202670)

[Hình 1.9 Sự phụ thuộc dữ liệu giữa các câu lệnh trong pipeline 11](#_Toc112202671)

[Hình 1.10 Stall và Forwarding trong pipeline 12](#_Toc112202672)

[Hình 1.11 Control hazard trong pipeline 13](#_Toc112202673)

[Hình 2.1 Kiến trúc RISC-V pipeline 15](#_Toc112202674)

[Hình 2.2 stage\_IF block diagram 16](#_Toc112202675)

[Hình 2.3 IMEM block diagram 16](#_Toc112202676)

[Hình 2.4 Instruction fetch architecture 17](#_Toc112202677)

[Hình 2.5 stage\_ID block diagram 18](#_Toc112202678)

[Hình 2.6 reg\_file block diagram 19](#_Toc112202679)

[Hình 2.7 control\_logic block diagram 20](#_Toc112202680)

[Hình 2.8 branch\_comp block diagram 21](#_Toc112202681)

[Hình 2.9 imm\_gen block diagram 22](#_Toc112202682)

[Hình 2.10 Instruction Decode architecture 23](#_Toc112202683)

[Hình 2.11 stage\_EX block diagram 24](#_Toc112202684)

[Hình 2.12 alu block diagram 24](#_Toc112202685)

[Hình 2.13 Execute architecture 25](#_Toc112202686)

[Hình 2.14 stage\_MEM block diagram 26](#_Toc112202687)

[Hình 2.15 DMEM block diagram 26](#_Toc112202688)

[Hình 2.16 DMEM block diagram 27](#_Toc112202689)

[Hình 2.17 stage\_WB block diagram 28](#_Toc112202690)

[Hình 2.18 Register write architecture 28](#_Toc112202691)

[Hình 2.19 Hazard detection block diagram 29](#_Toc112202692)

[Hình 2.20 Forwarding unit block diagram 29](#_Toc112202693)

[Hình 3.1 Mô phỏng timing diagram khối Instruction Fetch 34](#_Toc112202694)

[Hình 3.2 Mô phỏng timing diagram khối Instruction Decode 34](#_Toc112202695)

[Hình 3.3 Mô phỏng timing diagram khối Execute 34](#_Toc112202696)

[Hình 3.4 Mô phỏng timing diagram khối Memory Access 35](#_Toc112202697)

[Hình 3.5 Mô phỏng timing diagram khối Register Write 35](#_Toc112202698)

[Hình 3.6 Mô phỏng timing diagram khối Forwarding Unit 35](#_Toc112202699)

[Hình 3.7 Mô phỏng timing diagram khối Hazard Detection Unit 35](#_Toc112202700)

[Hình 3.8 Kết quả khi chạy thuật toán sắp xếp nổi bọt 38](#_Toc112202701)

DANH MỤC BẢNG BIỂU

[Bảng 1.1 Single cycle versus pipelined 4](#_Toc112202702)

[Bảng 2.1 RISC-V pipeline port description 15](#_Toc112202703)

[Bảng 2.2 stage\_IF port description 16](#_Toc112202704)

[Bảng 2.3 Instruction Fetch port description 16](#_Toc112202705)

[Bảng 2.4 stage\_ID port description 18](#_Toc112202706)

[Bảng 2.5 reg\_file port description 20](#_Toc112202707)

[Bảng 2.6 control\_logic port description 20](#_Toc112202708)

[Bảng 2.7 Bảng điều khiển tín hiệu đầu ra (control logic) 21](#_Toc112202709)

[Bảng 2.8 branch\_comp port description 22](#_Toc112202710)

[Bảng 2.9 imm\_gen port description 22](#_Toc112202711)

[Bảng 2.10 stage\_EX port description 24](#_Toc112202712)

[Bảng 2.11 alu port diagram 25](#_Toc112202713)

[Bảng 2.12 stage\_MEM port description 26](#_Toc112202714)

[Bảng 2.13 DMEM port description 27](#_Toc112202715)

[Bảng 2.14 stage\_WB port description 28](#_Toc112202716)

[Bảng 2.15 Hazard detection port description 29](#_Toc112202717)

[Bảng 2.16 Forwarding Unit port description 30](#_Toc112202718)

[Bảng 2.17 Forwarding output ports 31](#_Toc112202719)

[Bảng 3.1 Các lệnh kiểm thử 32](#_Toc112202720)

[Bảng 3.2 Thuật toán sắp xếp nổi bọt 36](#_Toc112202721)

# GIỚI THIỆU RISC-V PIPELINE

Chương này đưa ra các khái niệm về RISC-V, cấu trúc, địa chỉ và tập lệnh trong RISC-V cùng với cái nhìn chung về RISC-V Pipeline.

## Bộ xử lý RISC-V

### Định nghĩa

**RISC** (*Reduced Instructions Set Computer*) là một phương pháp thiết kế các bộ vi xử lý theo hướng đơn giản hóa tập lệnh, trong đó thời gian thực thi tất cả các lệnh đều như nhau. Hiện nay các bộ vi xử lý RISC phổ biến là ARM, MIPS, DEC Alpha, SPARC, SuperH, PA-RISC, PIC, IBM và PowerPC.

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.

### Cấu trúc RISC-V

RISC-V cơ sở là một kiến ​​trúc bộ xử lý 32-bit với 31 thanh ghi có mục đích chung. Tất cả các câu lệnh đều dài 32 bit. ISA cơ sở hỗ trợ phép cộng và trừ số nguyên, nhưng không bao gồm phép nhân và phép chia số nguyên.

Các thanh ghi có chiều rộng 32 bit. Các thanh ghi mục đích chung **x1** đến **x31** có sẵn để sử dụng mà không có bất kỳ hạn chế hoặc các chức năng đặc biệt được phần cứng bộ xử lý chỉ định. Thanh ghi **x0** được thiết kế cố định với giá trị trả về 0 khi nó được đọc và sẽ loại bỏ bất kỳ giá trị nào được ghi vào nó.

1. **a0-a7** (x10-x17): 8 thanh ghi đối số để truyền tham số và hai giá trị trả về (a0-a1).
2. **ra**: một thanh ghi địa chỉ để trả về địa chỉ ban đầu (x1).
3. **s0-s1** (x8-x9) và **s2-s11** (x18-x27): các thanh ghi lưu trữ.

Thanh ghi **PC** (Program Counter) chứa bộ đếm chương trình 32-bit, giữ địa chỉ của lệnh hiện tại.

### Tập lệnh RISC-V

Tập lệnh RISC-V cơ bản chỉ bao gồm 47 lệnh. Chia làm 3 kiểu lệnh: Số học và logic (Arithmetic and Logical); Dịch chuyển dữ liệu (Data transfer and Shift); và Điều khiển rẽ nhánh (Branch).

Có 6 loại lệnh tương ứng:

* R-format: cho các phép toán số học với thanh ghi.
* I-format: cho các phép toán số học trực tiếp và tải.
* S-format: cho các lệnh lưu trữ.
* B-format: cho các lệnh rẽ nhánh.
* U-format: cho các lệnh trực tiếp với 20 bit cao.
* J-format: cho các lệnh nhảy.

Với lệnh loại **R,** 3 toán hạng là 3 thanh ghi; lệnh loại **I**, **S**, **B** là 2 thanh ghi và lệnh loại **U**, **J** là 1 thanh ghi. Các trường **funct7**, **funct3** và **imm** sẽ tương ứng với từng câu lệnh và loại lệnh khác nhau sẽ có giá trị khác nhau.

### Địa chỉ RISC-V

Có 5 loại địa chỉ:

1. Immediate addressing: địa chỉ trực tiếp, trong đó toán hạng là một hằng số.
2. Register addressing: địa chỉ thanh ghi, trong đó toán hạng là thanh ghi.
3. Base or displacement addressing: địa chỉ cơ sở, trong đó toán hạng ở bộ nhớ có địa chỉ là tổng của một thanh ghi và một hằng số trong lệnh.
4. PC-relative addressing: trong đó địa chỉ nhánh là tổng của PC và một hằng số trong lệnh.

Địa chỉ trong RISC-V được biểu diễn theo kiểu Little Endian, tức là địa chỉ được đánh số bit giảm dần từ trái sang phải. Byte bên phải nhất làm địa chỉ của từ. Đây là một trong những điểm khác biệt so với bộ xử lý MIPS. Địa chỉ của 2 ô nhớ liên tiếp cách nhau 1 từ (4 bytes).

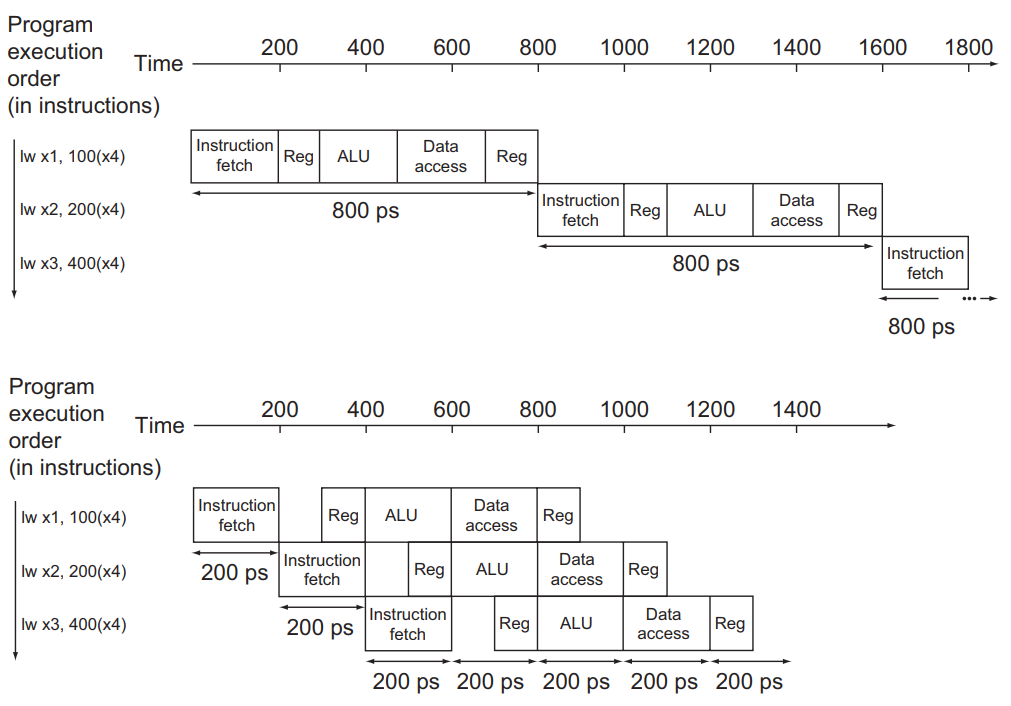
## RISC-V Pipeline

### Khái niệm

***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 giai đoạn:

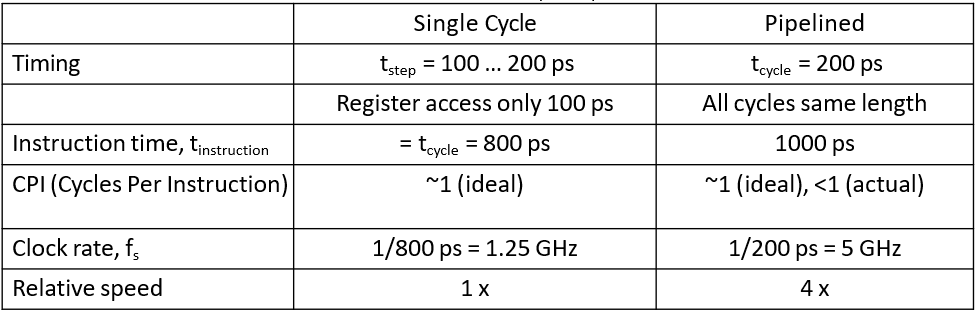
1. Instruction Fetch (IF): Fetch the instruction from memory
2. Instruction Decode (ID): Read register and decode the instruction
3. Execute (EX) - ALU (Arithmetic Logic Unit): Excute the operation or calculate an address
4. Memory Access (MEM): Access an operand in data memory (If necessary)
5. Write result into a register (WB) (if necessary)

Hình sau mô tả so sánh giữa single-cycle (non-pipelined) và pipelined:



Hình 1.1 Single cycle versus pipelined

Bảng 1.1 Single cycle versus pipelined



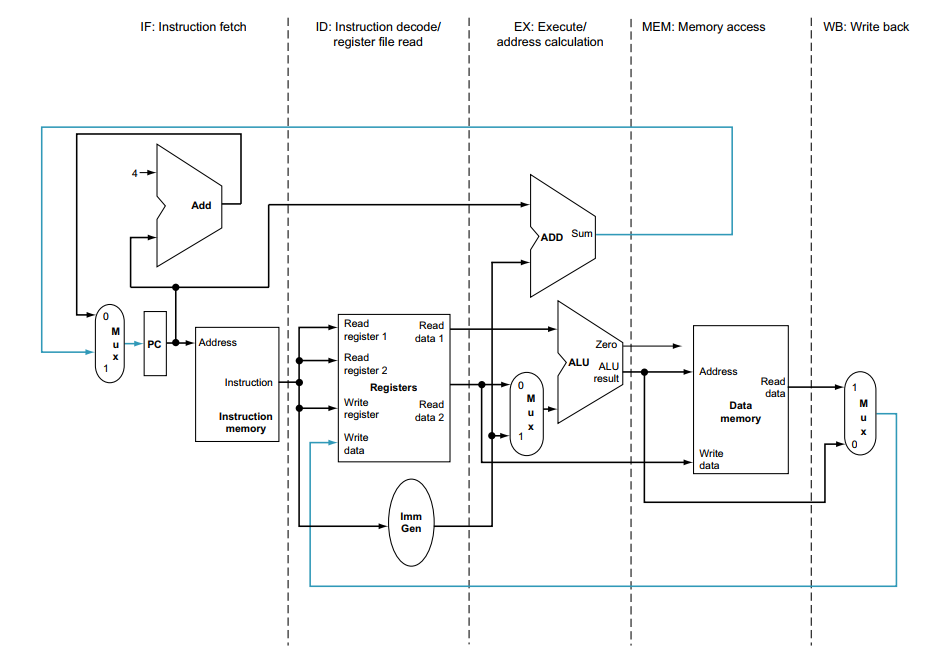
Dựa vào Hình 1.1, sử dụng chung một kiến trúc phần cứng, nhận thấy thời gian trung bình giữa các lệnh nhanh gấp 4 lần từ 800 ps xuống 200ps. Tuy nhiên, thời gian của giai đoạn pipeline cũng bị giới hạn bởi tài nguyên chậm nhất như hoạt động ALU hoặc truy cập vào bộ nhớ. Kết quả cụ thể được mô tả trong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

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.

### Pipeline Datapath

Hình 1.2 dưới đây mô tả single-cycle datapath với pipeline được định nghĩa. Việc chia lệnh thành 5 giai đoạn, nghĩa là một pipeline 5 giai đoạn, do đó có nghĩa là tối đa 5 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 5 phần, với mỗi phần được đặt tên tương ứng với một giai đoạn thực thi lệnh:

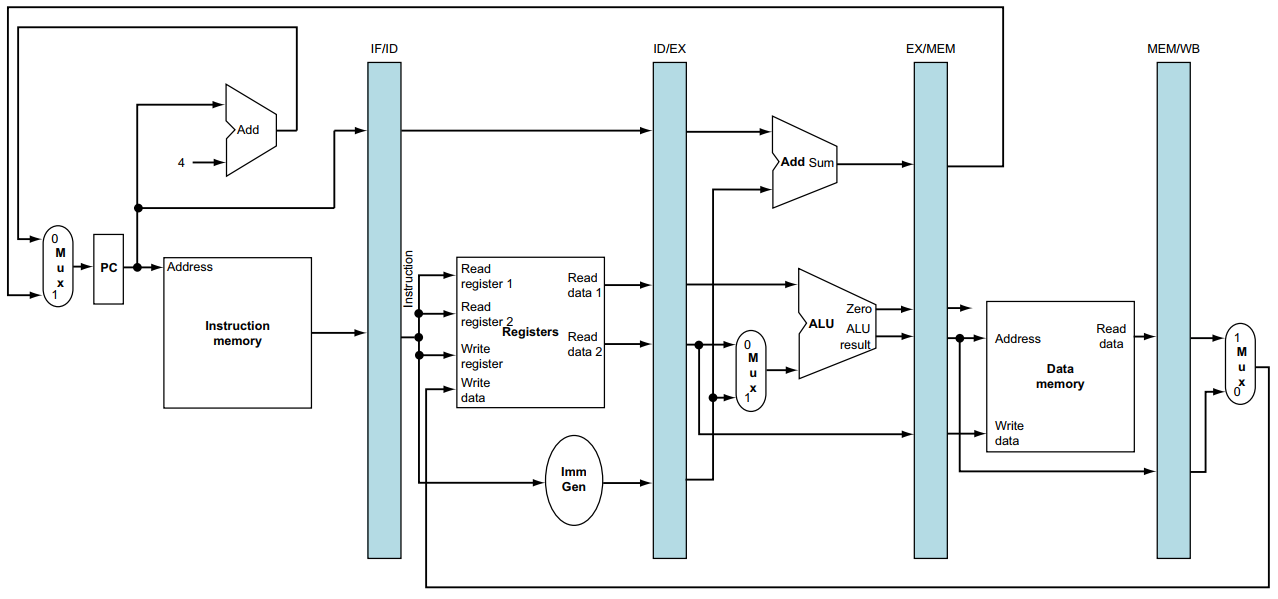
1. **IF**: *Instruction Fetch (nạp câu lệnh)*
2. **ID**: *Instruction Decode/register file read (giải mã lệnh)*
3. **EX**: *Excute/address calculation (thực thi/tính toán địa chỉ)*
4. **MEM**: *Memory access (truy cập bộ nhớ)*
5. **WB**: *Write back (ghi lại)*



Hình 1.2 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.

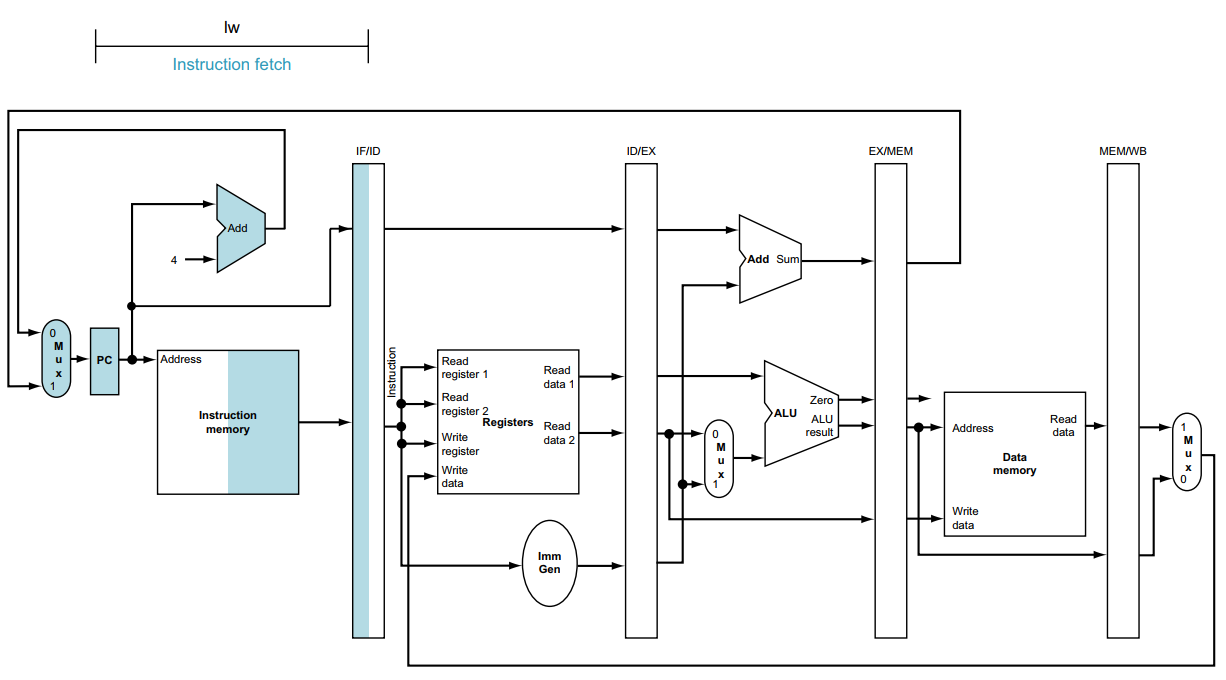
Hình 1.3 mô tả pipeline datapath với thanh ghi pipeline. Các thanh ghi pipeline được đặt tên theo hai giai đoạn bị ngăn cách bởi thanh ghi đó. Ví dụ, thanh ghi pipeline IF/ID ngăn cách hai giai đoạn IF và IF của datapath. Và không có thanh ghi tại giai đoạn cuối cùng WB. Các thanh ghi pipeline phải 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.



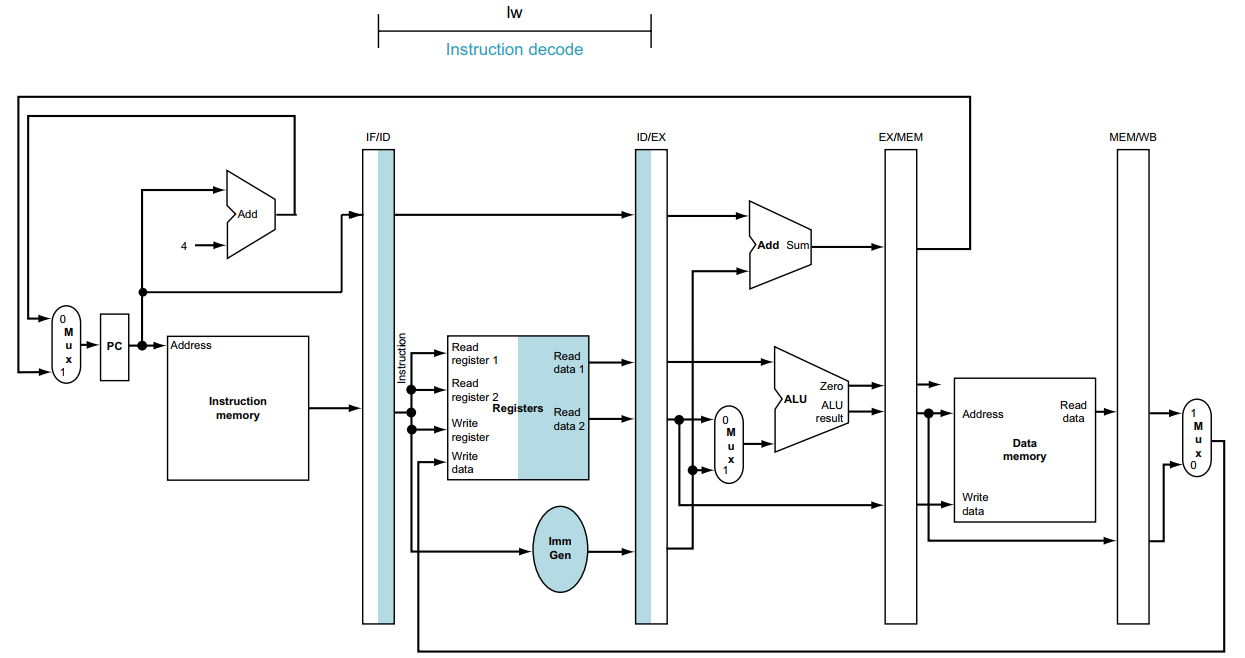
Hình 1.3 Pipeline Datapath

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

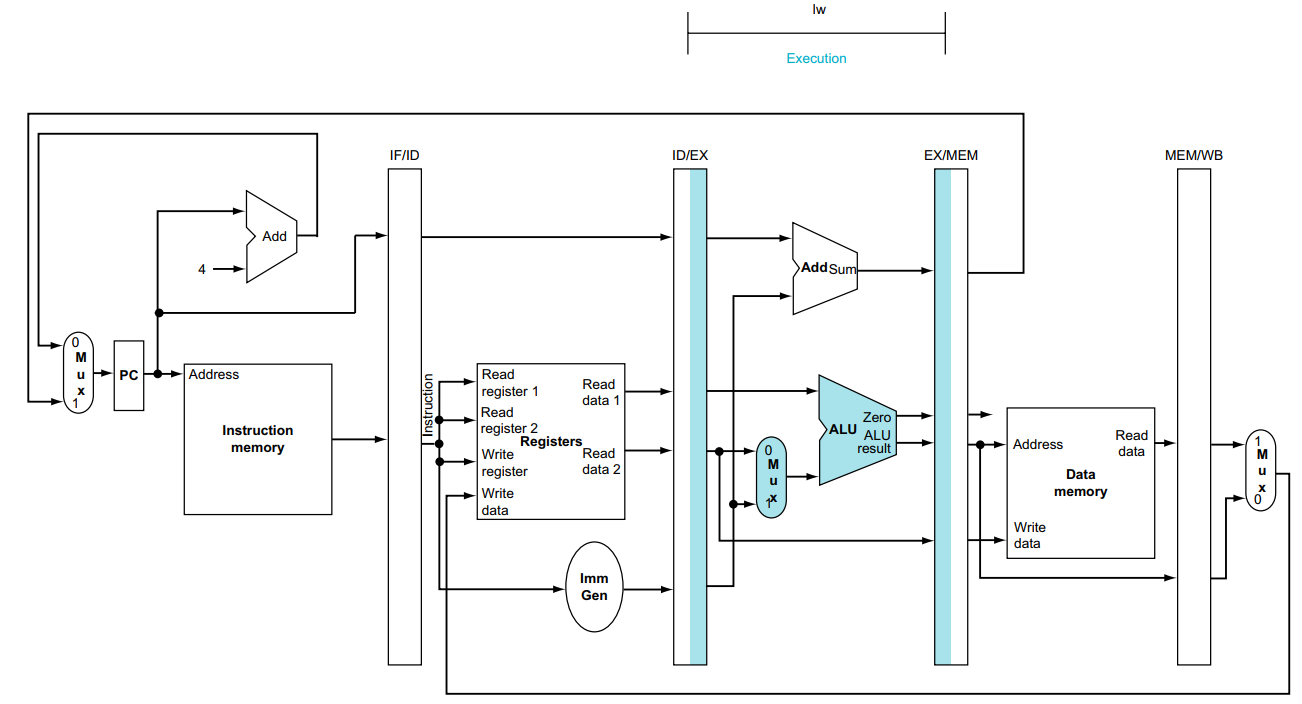
1. *Instruction Fetch (IF)*: Hình 1.4 cho thấy lệnh được đọc từ bộ nhớ bằng địa chỉ trong PC và sau đó được đặt trong thanh ghi pipeline IF/ID. Địa chỉ PC được tăng thêm 4 và sau đó được ghi lại vào PC để sẵn sàng cho chu kỳ xung nhịp tiếp theo. PC này cũng được lưu trong thanh ghi pipeline IF/ID trong 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 (ID) và register file read*: Hình 1.5 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 (EX) hoặc Address calculation*: Hình 1.6 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 (MEM)*: Hình 1.7 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 (WB)*: Đọc dữ liệu từ thanh ghi pipeline MEM/WB và ghi dữ liệu thanh ghi.



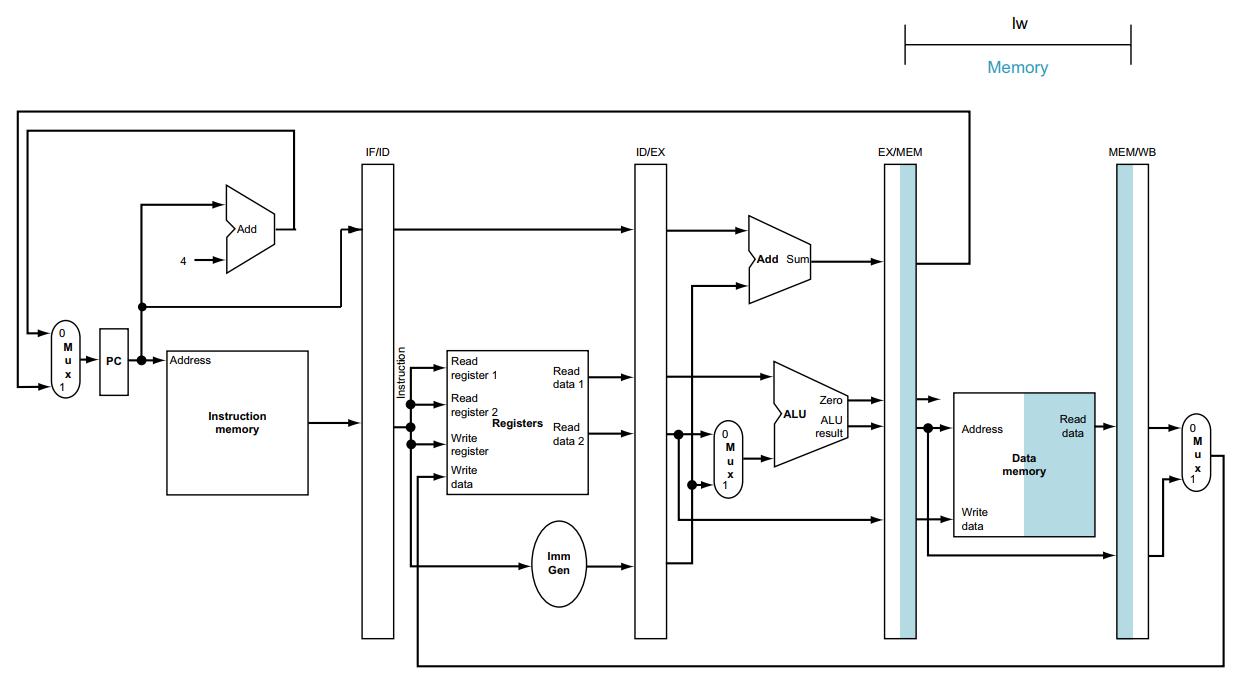
Hình 1.4 Giai đoạn IF của lệnh lw



Hình 1.5 Giai đoạn ID của lệnh lw



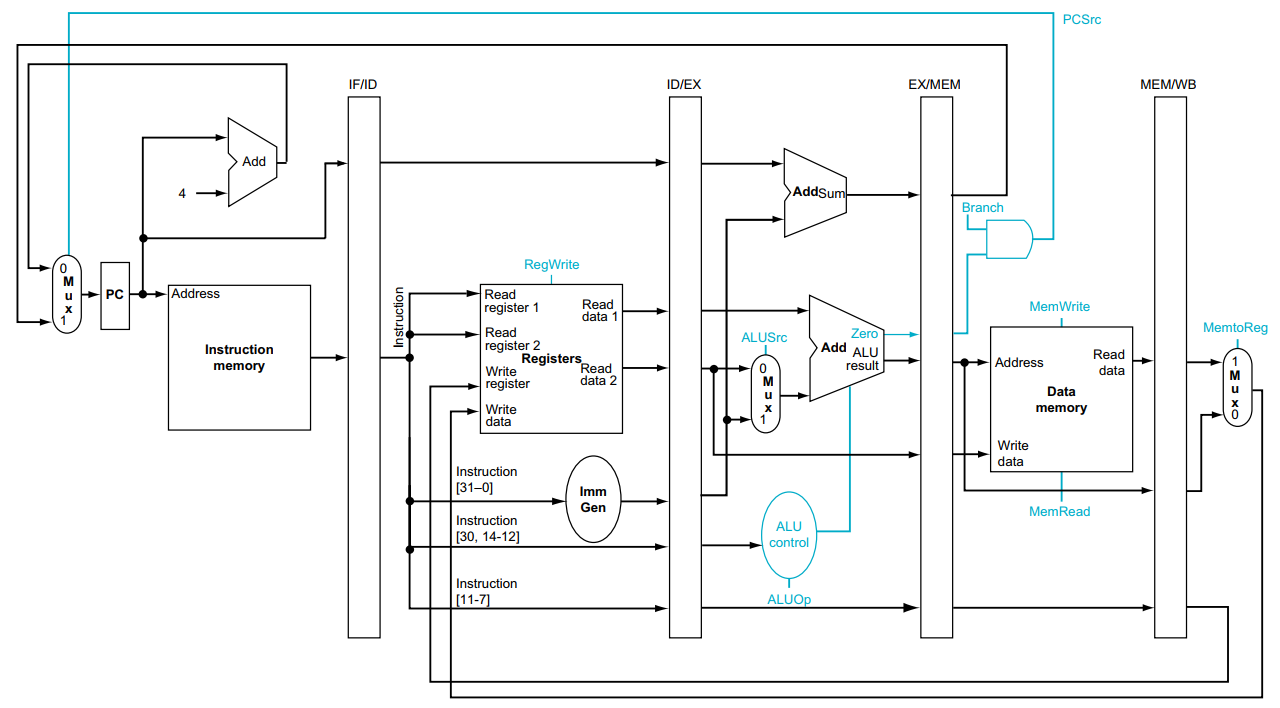
Hình 1.6 Giai đoạn EX của lệnh lw



Hình 1.7 Giai đoạn MEM của lw

### Pipeline 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 1.8 Pipeline 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 5 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. 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.

## 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:

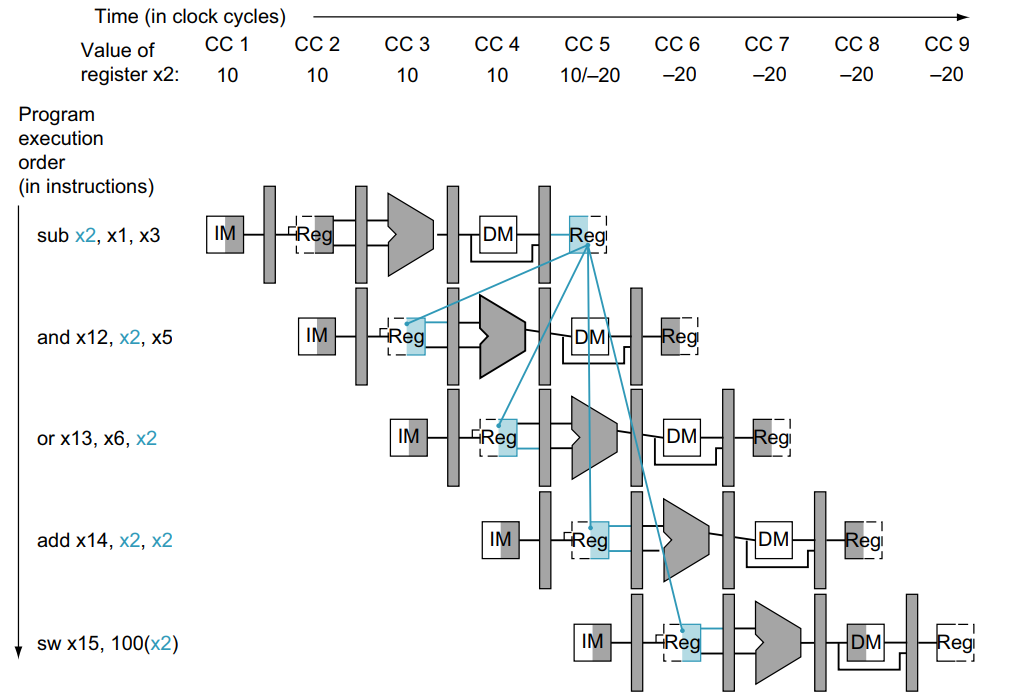
1. ***Structural hazard:*** Tài nguyên yêu cầu đang bận.
2. ***Data hazard:*** Sự phụ thuộc dữ liệu giữa các câu lệnh, cần đợi lệnh trước đó hoàn thành việc đọc ghi dữ liệu của nó.
3. ***Control hazard:*** Luồng thực hiện phụ thuộc vào lệnh trước đó.

### Structural Hazard

***Structural Hazard*** 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

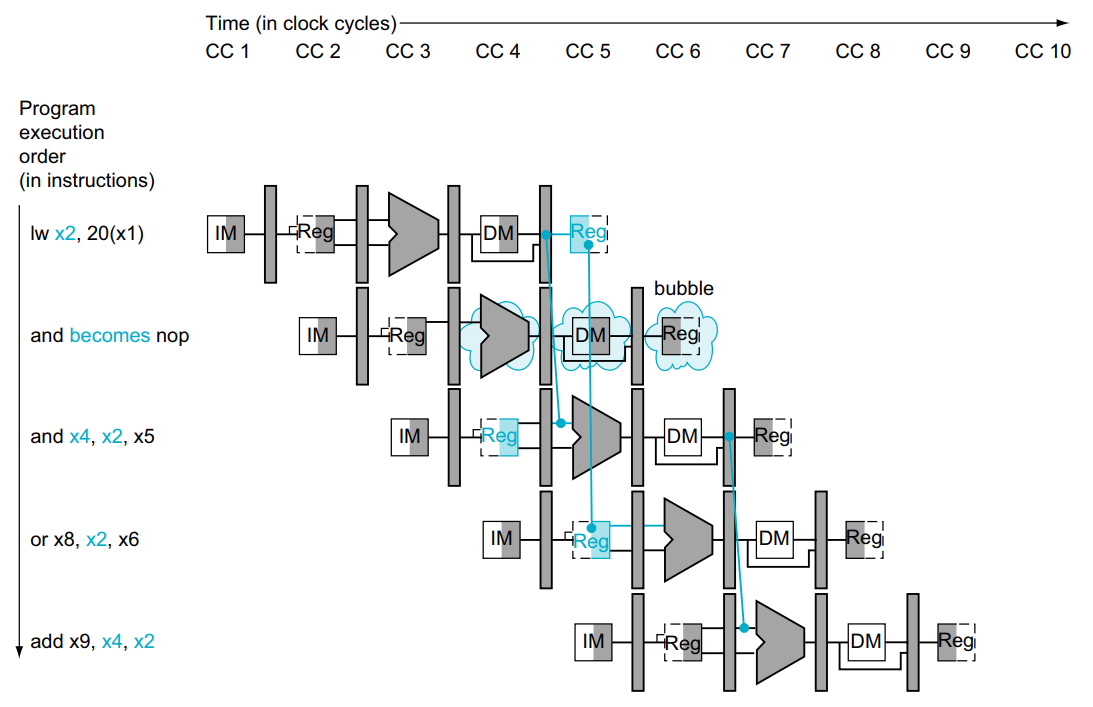
***Data Hazard*** thể hiện sự phụ thuộc dữ liệu giữa các câu lệnh trong pipeline, câu lệnh trước cần hoàn thành việc đọc ghi dữ liệu của nó. Hình 1.9 dưới đây thể hiện s phụ thuộc giữa các câu lệnh trong pipeline.



Hình 1.9 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 1.10 dưới đây.

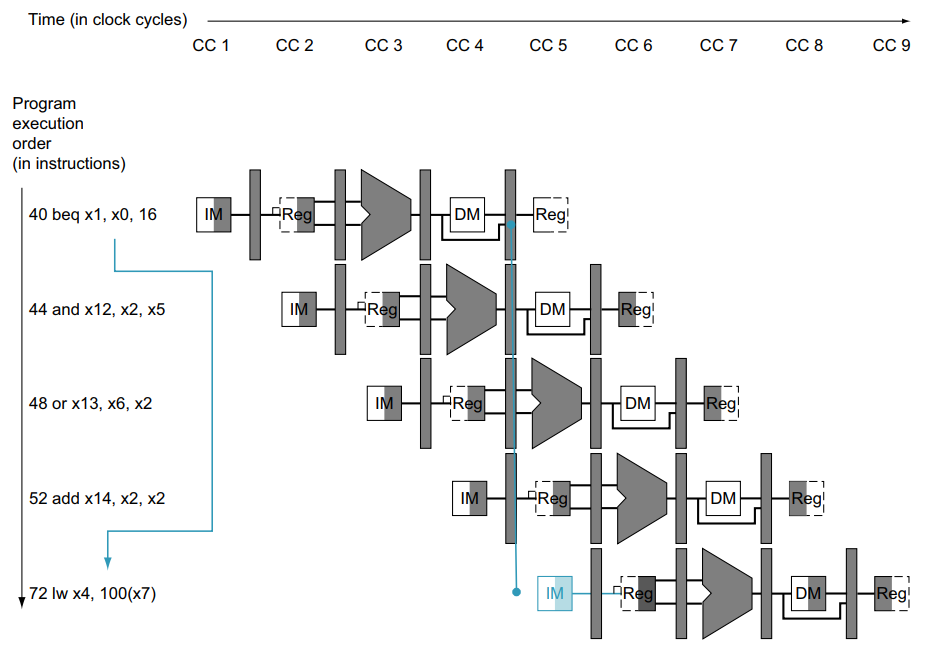


Hình 1.10 Stall và Forwarding trong pipeline

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 1.11 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.

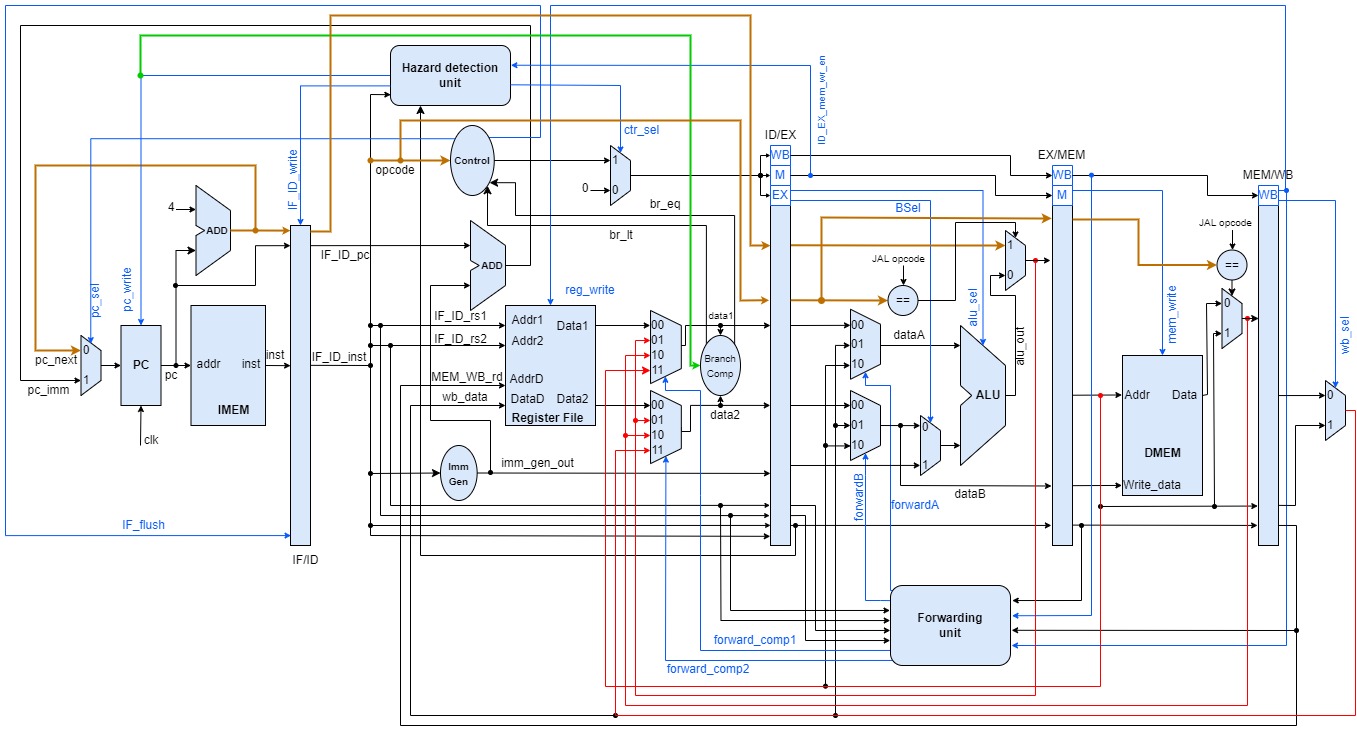
## Kết luận

# MÔ TẢ KIẾN TRÚC RISC-V PIPELINE

Chương này sẽ mô tả kiến trúc tổng quát của RISC-V Pipeline và chi tiết cấu trúc của từng phần có trong kiến trúc RISC-V Pipeline.

## Kiến trúc RISC-V pipeline

### Kiến trúc tổng quát



Hình 2.1 Kiến trúc RISC-V pipeline

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**.

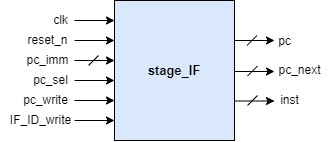
### Interface signals

Bảng 2.1 RISC-V pipeline port description

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal name** | **Width** | **Input/Output** | **Description** |
| **clk** | 1 | Input | Tín hiệu xung đồng hồ |
| **reset\_n** | 1 | Input | Tín hiệu reset không đồng bộ 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**.

## Khối Intruction Fetch (IF)

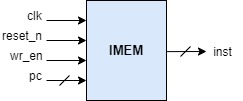


Hình 2.2 stage\_IF block diagram

Bảng 2.2 stage\_IF 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\_imm** |  | Input | pc = pc + immediate |
| **pc\_sel** | 1 | Input | Tín hiệu lựa chọn đầu vào PC |
| **pc\_write** | 1 | Input | Đầu ra khối Hazard detection unit, xác định giá trị PC thực hoặc tất cả bằng 0 |
| **IF\_ID\_write** | 1 | Input | Cho phép thanh ghi IF\_ID ghi dữ liệu mới |
| **pc** |  | Output | Giá trị PC làm đầu vào cho khối IMEM |
| **pc\_next** |  | Output | pc = pc +4 |
| **inst** |  | Output | Instruction của lệnh cần thực hiện |

Khối IMEM là submodule của khối stage\_IF.



Hình 2.3 IMEM block diagram

Bảng 2.3 Instruction Fetch port description

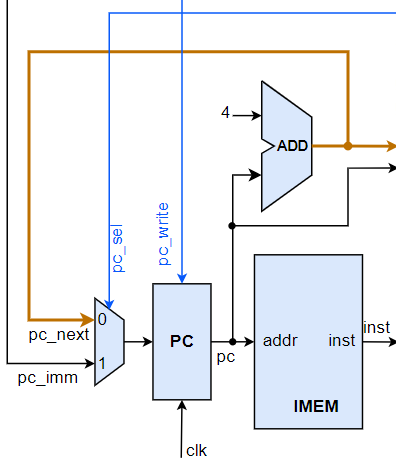
|  |  |  |  |
| --- | --- | --- | --- |
| **Signal name** | **Width** | **Input/Output** | **Description** |
| **clk** | 1 | Input | Tín hiệu xung đồng hồ |
| **reset\_n** | 1 | Input | Tín hiệu reset tích cực mức thấp |
| **wr\_en** | 1 | Input | Write enable |
| **pc** | 32 | Input | Giá trị PC làm đầu vào cho khối IMEM |
| **inst** | 32 | Output | Instruction của lệnh cần thực hiện |

### Chức năng

Khối 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.

### Kiến trúc khối IF

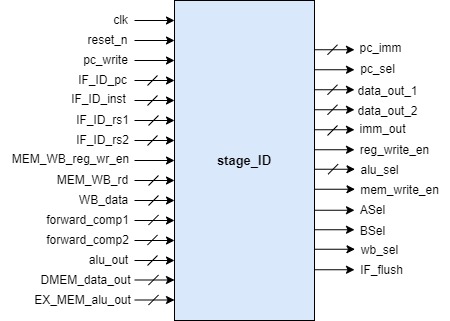


Hình 2.4 Instruction fetch architecture

Hình 2.4 mô tả các phần tử có trong kiến trúc của khối IF gồm:

* **Mux2to1** nhận tín hiệu **pc\_sel** 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** 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)

## Khối Instruction Decode (ID)



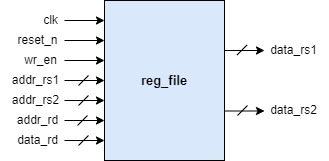
Hình 2.5 stage\_ID block diagram

Bảng 2.4 stage\_ID 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\_write** | 1 | Input | Đầu ra khối Hazard detection unit, xác định giá trị PC thực hoặc tất cả bằng 0 |
| **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 |
| **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 |
| **MEM\_WB\_reg\_wr\_en** | 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 |
| **forward\_comp1** | 2 | Input | Forwarding khi có hazard tại thanh ghi rs1 |
| **forward\_comp2** | 2 | Input | Forwarding khi có hazard tại thanh ghi rs2 |
| **alu\_out** | 32 | Input | Kết quả alu được forward về |
| **DMEM\_data\_out** | 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 |
| **pc\_imm** | 32 | Output | pc = pc + immediate |
| **pc\_sel** | 1 | Output | Tín hiệu lựa chọn đầu vào PC |
| **data\_out\_1** | 32 | Output | Data out rs1 |
| **data\_out\_2** | 32 | Output | Data out rs2 |
| **imm\_out** | 32 | Output | Immediate out |
| **reg\_write\_en** | 1 | Output | Cho phép ghi dữ liệu vào thanh ghi |
| **alu\_sel** | 3 | Output | ALU select |
| **mem\_write\_en** | 1 | Output | Cho phép memory ghi dữ liệu |
| **ASel** | 1 | Output | A select |
| **BSel** | 1 | Output | B select |
| **wb\_sel** | 1 | Output | WB select |
| **IF\_flush** | 1 | Output | Cho phép xóa thanh ghi IF\_ID |

Sub module của khối stage\_ID gồm có khối reg\_file, khối control\_logic, khối branch\_comp và khối imm\_gen.

### Reg\_file block

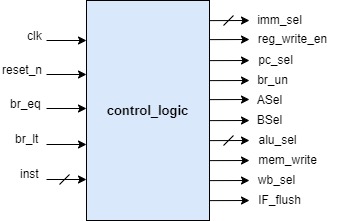


Hình 2.6 reg\_file block diagram

Bảng 2.5 reg\_file 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 |
| **wr\_en** | 1 | Input | Write enable |
| **addr\_rs1** | 5 | Input | Giá trị PC lưu trong thanh ghi IF\_ID |
| **addr\_rs2** | 5 | Input | Mã lệnh lưu trong thanh ghi IF\_ID |
| **addr\_rd** | 5 | Input | Address rd |
| **data\_rd** | 32 | Input | Data rd |
| **data\_rs1** | 32 | Output | Data rs1 |
| **data\_rs2** | 32 | Output | Data rs2 |

### Control\_logic block



Hình 2.7 control\_logic block diagram

Khối **control unit** nhận 32 bit instruction làm đầu vào và dựa vào phần opcode của instruction rồi đưa ra các tín hiệu điều khiển tương ứng như trong Bảng 2.6.

Bảng 2.6 control\_logic 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 |
| **br\_eq** | 32 | Input | Kết quả so sánh bằng giữa data1 và data2 |
| **br\_lt** | 32 | Input | Kết quả so sánh nhỏ hơn giữa data1 và data2 |
| **inst** | 1 | Input | Instruction của lệnh cần thực hiện |
| **imm\_sel** | 5 | Ouput | Immediate select |
| **reg\_write\_en** | 32 | Ouput | Cho phép ghi dữ liệu vào thanh ghi |
| **pc\_sel** | 3 | Ouput | Tín hiệu lựa chọn đầu vào PC |
| **br\_un** | 1 | Ouput | Kết quả so sánh giữa data1 và data2 |
| **ASel** | 1 | Ouput | A select |
| **BSel** | 1 | Ouput | B select |
| **alu\_sel** | 1 | Ouput | ALU select |
| **mem\_write** | 1 | Ouput | Cho phép memory ghi dữ liệu |
| **wb\_sel** | 2 | Ouput | WB select |
| **IF\_flush** | 1 | Ouput | Cho phép xóa thanh ghi IF\_ID |

Bảng 2.7 Bảng điều khiển tín hiệu đầu ra (control logic)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Inst[31:0]** | **BrEq** | **BrLT** | **PCSel** | **ImmSel BrUn ASel** | **BSel** | **ALUSel MemRW RegWEn WBSel** |
| **add** | \* | \* | +4 | \* \* Reg | Reg | Add Read 1 ALU |
| **sub** | \* | \* | +4 | \* \* Reg | Reg | Sub Read 1 ALU |
| **(R-R Op)** | \* | \* | +4 | \* \* Reg | Reg | (Op) Read 1 ALU |
| **addi** | \* | \* | +4 | I \* Reg | Imm | Add Read 1 ALU |
| **lw** | \* | \* | +4 | I \* Reg | Imm | Add Read 1 Mem |
| **sw** | \* | \* | +4 | S \* Reg | Imm | Add Write 0 \* |
| **beq** | 0 | \* | +4 | B \* PC | Imm | Add Read 0 \* |
| **beq** | 1 | \* | ALU | B \* PC | Imm | Add Read 0 \* |
| **bne** | 0 | \* | ALU | B \* PC | Imm | Add Read 0 \* |
| **bne** | 1 | \* | +4 | B \* PC | Imm | Add Read 0 \* |
| **blt** | \* | 1 | ALU | B 0 PC | Imm | Add Read 0 \* |
| **bltu** | \* | 1 | ALU | B 1 PC | Imm | Add Read 0 \* |
| **jalr** | \* | \* | ALU | I \* Reg | Imm | Add Read 1 PC+4 |
| **jal** | \* | \* | ALU | J \* PC | Imm | Add Read 1 PC+4 |
| **auipc** | \* | \* | +4 | U \* PC | Imm | Add Read 1 ALU |

### Branch\_comp block

Ảnh có chứa văn bản, đồng hồ, ăng-ten

Mô tả được tạo tự động

Hình 2.8 branch\_comp block diagram

Bảng 2.8 branch\_comp port description

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal name** | **Width** | **Input/Output** | **Description** |
| **pc\_write** | 1 | Input | Đầu ra khối Hazard detection unit, xác định giá trị PC thực hoặc tất cả bằng 0 |
| **br\_un** | 1 | Input | Kết quả so sánh giữa data1 và data2 |
| **data\_rs1** | 32 | Input |  |
| **data\_rs2** | 32 | Input |  |
| **br\_eq** | 1 | Output | Kết quả so sánh bằng giữa data1 và data2 |
| **br\_lt** | 1 | Output | Kết quả so sánh nhỏ hơn giữa data1 và data2 |

### Imm\_gen block

Ảnh có chứa văn bản, ăng-ten

Mô tả được tạo tự động

Hình 2.9 imm\_gen block diagram

Bảng 2.9 imm\_gen port description

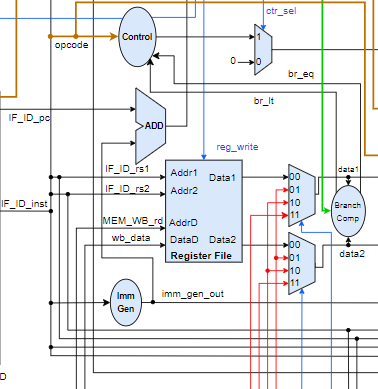
|  |  |  |  |
| --- | --- | --- | --- |
| **Signal name** | **Width** | **Input/Output** | **Description** |
| **inst** | 32 | Input | Instruction của lệnh cần thực hiện |
| **imm\_sel** | 3 | Input | Immediate select |
| **imm\_out** | 32 | Output | Immediate out |

### Chức năng

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.

### Kiến trúc khối ID

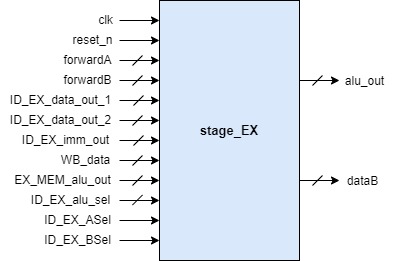


Hình 2.10 Instruction Decode architecture

Hình 2.10 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 **MEM\_WB\_rd**, **wb\_data** từ việc 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 branch
* 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

## Khối Excute (EX)



Hình 2.11 stage\_EX block diagram

Bảng 2.10 stage\_EX 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 |
| **forwardA** | 2 | Input | Forwarding khi có hazard tại thanh ghi rs1 |
| **forwardB** | 2 | Input | Forwarding khi có hazard tại thanh ghi rs2 |
| **ID\_EX\_data\_out\_1** | 32 | Input | ID/EX Data rs1 |
| **ID\_EX\_data\_out\_2** | 32 | Input | ID/EX Data rs2 |
| **ID\_EX\_imm\_out** | 32 | Input | ID/EX Immediate out |
| **WB\_data** | 32 | Input | Dữ liệu được write back |
| **EX\_MEM\_alu\_out** | 32 | Input | Kết quả của alu trong thanh ghi EX/MEM được forward về |
| **ID\_EX\_alu\_sel** | 3 | Input | ID/EX ALU select |
| **ID\_EX\_ASel** | 1 | Input | A select |
| **ID\_EX\_BSel** | 1 | Input | B select |
| **alu\_out** | 32 | Input | Kết quả alu được forward về |
| **dataB** | 32 | Input | Data B |

Khối alu là submodule của khối stage\_EX.

Ảnh có chứa văn bản, ăng-ten

Mô tả được tạo tự động

Hình 2.12 alu block diagram

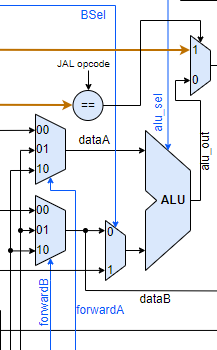
Bảng 2.11 alu port diagram

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal name** | **Width** | **Input/Output** | **Description** |
| **alu\_sel** | 3 | Input | ALU select |
| **dataA** | 32 | Input | Data A |
| **dataB** | 32 | Input | Data B |
| **alu\_out** | 32 | Output | Kết quả alu được forward về |

### Chức năng

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.

### Kiến trúc khối EX

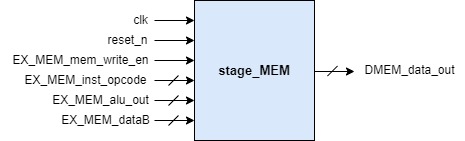


Hình 2.13 Execute architecture

Hình 2.13 mô tả kiến trúc của khối **Execute** gồm:

* 2 bộ **mux4to1** nhận tín hiệu từ khối **Forwarding unit** nhằm forward đầu vào cho khối **ALU** trong trường hợp hazard
* **Mux2to1** nhận tín hiệu **BSel** 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** nhận tín hiệu **alu\_sel** từ khối **control** và đưa ra tín hiệu **alu­\_out** đến khối **ALU** để điều khiển hoạt động của **ALU** theo bảng sự thật
* **ALU** sẽ dựa vào tín hiệu **alu­\_out** nhận được từ khối **ALU** 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

## Khối Memory Access (MEM)

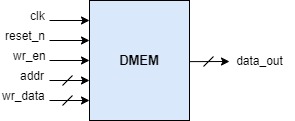


Hình 2.14 stage\_MEM block diagram

Bảng 2.12 stage\_MEM 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\_write\_en** | 1 | Input | Cho phép memory ghi dữ liệu lưu trong thanh ghi EX/MEM |
| **EX\_MEM\_inst\_opcode** | 7 | Input | EX/MEM opcode |
| **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 |
| **DMEM\_data\_out** | 32 | Output | ALU out |

Khối DMEM là submodule của khối stage\_MEM.



Hình 2.15 DMEM block diagram

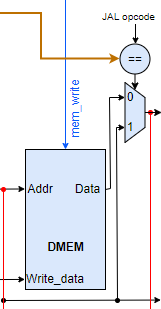
Bảng 2.13 DMEM port description

|  |  |  |  |
| --- | --- | --- | --- |
| **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 |
| **wr\_en** | 1 | Input | Write enable |
| **addr** | 32 | Input | Write address or read address |
| **wr\_data** | 32 | Input | Data written |
| **data\_out** | 32 | Output | Read data |

### Chức năng

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.

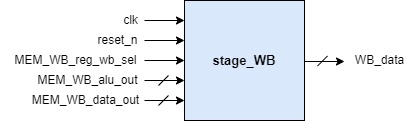
### Kiến trúc khối MEM



Hình 2.16 DMEM block diagram

Hình 2.16 mô tả kiến trúc khối **Memory access**, gồm một bộ nhớ **Data Memory** nhận tín hiệu điều khiển **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**.

## Khối Register Write (WB)



Hình 2.17 stage\_WB block diagram

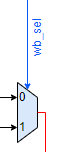
Bảng 2.14 stage\_WB port description

|  |  |  |  |
| --- | --- | --- | --- |
| **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 |
| **MEM\_WB\_reg\_wb\_sel** | 1 | Input |  |
| **MEM\_WB\_alu\_out** | 32 | Input | Kết quả alu được lưu trong thanh ghi MEM/WB |
| **MEM\_WB\_data\_out** | 32 | Input | DMEM data out |
| **WB\_data** | 32 | Output | Dữ liệu write back |

### Chức năng

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.

### Kiến trúc khối Register Write



Hình 2.18 Register write architecture

Hình 2.18 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**.

## Khối Hazard Detection Unit



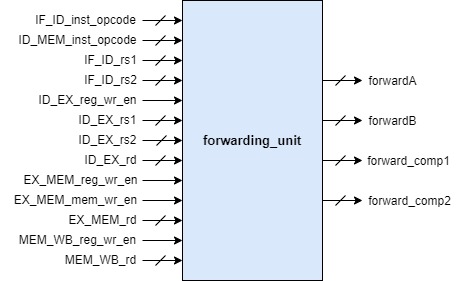
Hình 2.19 Hazard detection block diagram

Bảng 2.15 Hazard detection port description

| **Signal name** | **Width** | **Input/Output** | **Description** |
| --- | --- | --- | --- |
| **ID\_EX\_inst\_opcode** | 7 | 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 |
| **ID\_EX\_mem\_wr\_en** | 1 | Input | ID/EX Mem write enable |
| **pc\_write** | 1 | Output | Đầu ra khối Hazard detection unit, xác định giá trị PC thực hoặc tất cả bằng 0 |
| **IF\_ID\_write** | 1 | Output | Cho phép thanh ghi IF\_ID ghi dữ liệu mới |
| **ctrl\_sel** | 1 | Output | Chọn đầu ra control |

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 Forwarding Unit



Hình 2.20 Forwarding unit block diagram

Bảng 2.16 Forwarding Unit port description

| **Signal name** | **Width** | **Input/Output** | **Description** |
| --- | --- | --- | --- |
| **IF\_ID\_inst\_opcode** | 7 | Input | Địa chỉ của thanh ghi 1 lưu trong thanh ghi ID/EX |
| **EX\_MEM\_inst\_opcode** | 7 | Input | EX/MEM opcode |
| **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 |
| **ID\_EX\_reg\_wr\_en** | 1 | Input | Cho phép ghi dữ liệu vào thanh ghi 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 được lưu trong thanh ghi ID/EX |
| **EX\_MEM\_reg\_wr\_en** | 1 | Input | Cho phép ghi dữ liệu vào thanh ghi lưu trong thanh ghi EX/MEM |
| **EX\_MEM\_mem\_wr\_en** | 1 | Input | Cho phép ghi dữ liệu vào bộ nhớ 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\_wr\_en** | 1 | Input | Cho phép ghi dữ liệu vào thanh ghi lưu trong thanh ghi MEM/WB |
| **MEM\_WB\_rd** | 5 | Input | Cho phép ghi dữ liệu vào thanh ghi lưu trong thanh ghi ID/EX |
| **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\_comp1** | 2 | Output | Forwarding cho bộ branch compare khi có hazard tại thanh ghi rs1 ID |
| **forward\_comp2** | 2 | Output | Forwarding cho bộ branch compare khi có hazard tại thanh ghi rs2 ID |

### Chức năng

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.

Bảng 2.17 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 forward 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\_comp1 = 00** | **Register** | Đường input thứ nhất của **branch compare** đến từ tệp thanh ghi. |
| **forward\_comp1 = 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\_comp1 = 01** | **ALU** | Đường input thứ nhất của **branch compare** đến từ kết quả tính toán **ALU** của lệnh trước đó. |
| **forward\_comp1 = 11** | **EX/MEM** | Đường input thứ nhất của **branch compare** được foward từ kết quả tính toán của **ALU** cho lệnh ngay trước đó. |
| **forward\_comp2 = 00** | **Register** | Đường input thứ hai của **branch compare** đến từ tệp thanh ghi. |
| **forward\_comp2 = 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\_comp2 = 01** | **ALU** | Đường input thứ hai của **branch compare** đến từ kết quả tính toán **ALU** của lệnh trước đó. |
| **forward\_comp2 = 11** | **MEM/WB** | Đường input thứ hai của **branch compare** 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. |

## Kết luận

# KẾT QUẢ VÀ ĐÁNH GIÁ

Chương này trình bày các kết quả mô phỏng cho từng khối và toàn bộ thiết kế được triển khai bằng ngôn ngữ System Verilog trên phần mềm Questa Sim*.*

## Kiểm thử RISC-V

### Các trường hợp kiểm tra kiểm thử

Kết quả chạy trên đoạn code Assembly trong Bảng 3.1, trong đó bao gồm các trường hợp:

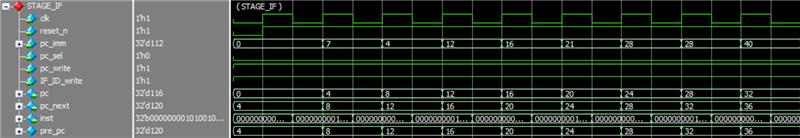
* Kiểm tra các trường hợp instruction không có hazard đối với các loại lệnh I, R, S, J, B.
* Kiểm tra các trường hợp instruction có data hazard (EX hazard, MEM hazard), trong đó có 1 số trường hợp đặc biệt cần chú ý như từ lệnh load sang branch hoặc từ jal sang branch.
* Kiểm tra các trường hợp instruction có control hazard.

Bảng 3.1 Các lệnh kiểm thử

|  |
| --- |
| add x8 ,x12,x14  sub x10,x12,x8  addi x15,x10,-50  lw x14, 8(x2)  add x5 ,x19, x14  sw x14, 4(x2)  jal x10, L2  beq x1 , x10 , L1  lw x7 ,20(x5)  addi x7 ,x11 ,2  sw x7 ,12(x5)  sub x2 ,x11 ,x7  and x14, x5 ,x3  L1:  sw x14,16(x5)  beq x1, x1 , L2  add x8 ,x12 ,x14  sub x10,x12 ,x8  L2:  addi x15, x10 ,-50  jal x15, L3  lw x14, 8(x2)  add x5 ,x14 , x19  lw x14, 20(x2)  beq x14, x1 , L3  add x15 ,x12 ,x14  L3:  addi x10, x0, 10  addi x14, x0, -10  bltu x10, x14, L4  beq x2, x15, L1  add x5, x6, x7  L4:  and x7, x5, x6 |

### Kết quả và đánh giá

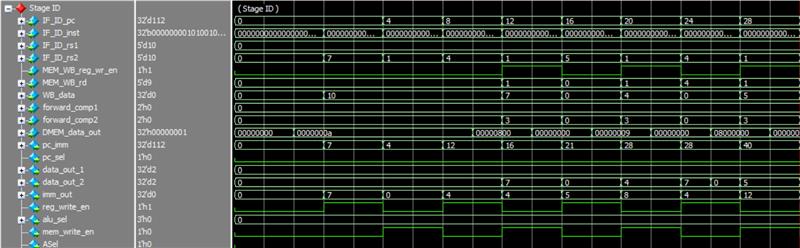
#### Khối Instruction Fetch



Hình 3.1 Mô phỏng timing diagram khối Instruction Fetch

Kết quả đầu ra hoạt động đúng logic khi thực hiện các lệnh được nạp sẵn vào **Instruction Memory**.

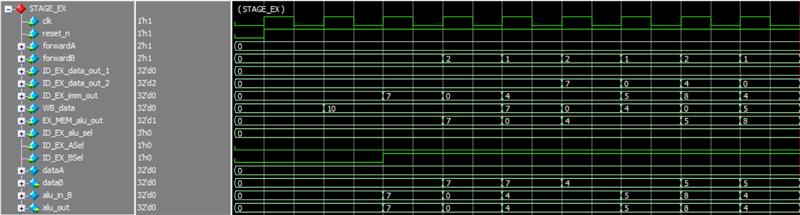
#### Khối Instruction Decode



Hình 3.2 Mô phỏng timing diagram khối Instruction Decode

Kết quả đầu ra của khối **Instruction Decode** hoạt động đúng với logic của thiết kế khi thực hiện các lệnh được nạp sẵn vào **Instruction Memory**.

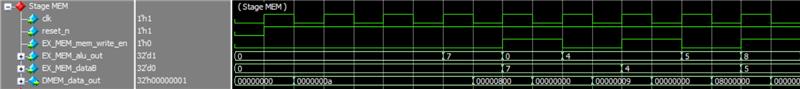
#### Khối Execute



Hình 3.3 Mô phỏng timing diagram khối Execute

Kết quả đầu ra của khối **Execute** hoạt động đúng với logic của thiết kế khi thực hiện các lệnh được nạp sẵn.

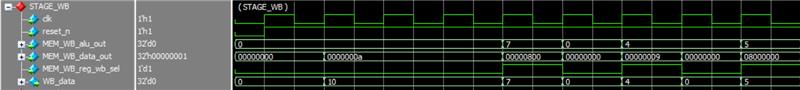
#### Khối Memory Access



Hình 3.4 Mô phỏng timing diagram khối Memory Access

Kết quả đầu ra của khối **Memory Access** hoạt động đúng với logic của thiết kế khi thực hiện các lệnh được nạp sẵn.

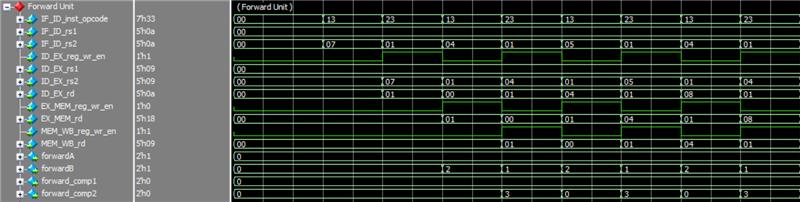
#### Khối Register Write



Hình 3.5 Mô phỏng timing diagram khối Register Write

Kết quả đầu ra của khối **Register Write** hoạt động đúng với logic của thiết kế.

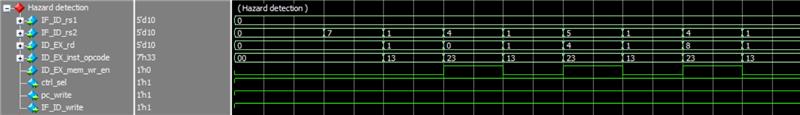
#### Khối Forwarding Unit



Hình 3.6 Mô phỏng timing diagram khối Forwarding Unit

Tín hiệu đầu ra của module khối **forwarding unit** theo các kích thích đầu vào tương ứng, từ Hình 3.6 cho thấy khối **forwarding unit** hoạt động đúng logic đã định nghĩa.

#### Khối Hazard Detection Unit



Hình 3.7 Mô phỏng timing diagram khối Hazard Detection Unit

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

## Thuật toán sắp xếp nổi bọt

Code Assembly cho thuật toán sắp xếp nổi bọt được trình bày trong Bảng 3.2.

Bảng 3.2 Thuật toán sắp xếp nổi bọt

|  |
| --- |
| # Store array: {7, 4, 5, 4, 3, 2, 1, 10, 8, 7}  addi x1, x0, 7  sw x1, 0(x0)  addi x1, x0, 4  sw x1, 4(x0)  addi x1, x0, 5  sw x1, 8(x0)  addi x1, x0, 4  sw x1, 12(x0)  addi x1, x0, 3  sw x1, 16(x0)  addi x1, x0, 2  sw x1, 20(x0)  addi x1, x0, 1  sw x1, 24(x0)  addi x1, x0, 10  sw x1, 28(x0)  addi x1, x0, 8  sw x1, 32(x0)  addi x1, x0, 7  sw x1, 36(x0)  addi x4, x0, 0 # x4 is base address  addi x6, x0, 0 # x6 = i = 0  addi x7, x0, 9 # n - 1 = 9  loop1:  beq x6, x7, exit # if (i == n - 1) exit  sub x8, x7, x6 # x8 = x7 - x6 = n - 1 - i  addi x9, x0, 0 # x9 = j = 0  loop2:  beq x8, x9, next # if (j == n - 1 - i) i = i + 1; else continue;  add x10, x9 , x9 # x10 = 2 \* x9  add x10, x10, x10 # x10 = 4 \* x9 = 4 \* j  add x10, x10, x4 # x10 = &(base address + 4 \* i) = &arr[j]  lw x11, 0(x10) # x11 = arr[j]  addi x12, x10, 4 # x12 = x10 + 4 = &arr[j + 1]  lw x13, 0(x12) # x13 = arr[j+1]  blt x13, x11, swap # arr[j+1] < arr[j] -> swap  addi x9, x9, 1 # else j += 1  jal x0, loop2 # continue loop2  swap:  sw x11, 0(x12) # arr[j + 1] = x11 = old arr[j]  sw x13, 0(x10) # arr[j] = old arr[j + 1]  addi x9, x9, 1 # j = j + 1  jal x0, loop2 # continue loop2  next:  addi x6, x6, 1 # i = i + 1  jal x0, loop1 # continue loop1  exit:  addi x1, x0, 1 |

Ảnh có chứa bàn

Mô tả được tạo tự động

Hình 3.8 Kết quả khi chạy thuật toán sắp xếp nổi bọt

Kết quả mô phỏng cho thuật toán sắp xếp nổi bọt trong Hình 3.8. Các phần tử được sắp xếp theo thứ tự tăng dần.

## Đánh giá kết quả

Như vậy, sau quá trình tìm hiểu và thực hiện đề tài, nhóm đã đạt được các kết quả sau:

* Chạy được các lệnh loại I, R, S, J, B.
* Xử lý được data hazard, control hazard, structural hazard.
* Thêm được nhánh xử lý và forward cho lệnh jal chỉ bị stall 1 chu kỳ.
* Thêm được nhánh forward cho lệnh branch.
* Chạy đúng thuật toán sắp xếp nổi bọt.

KẾT LUẬN

Qua quá trình tìm hiểu cùng với sự hướng dẫn của PGS.TS Nguyễn Đức Minh, nhóm đã hoàn thành báo cáo của bài tập lớn. Trong quá trình làm báo cáo nhóm đã hiểu và nắm rõ hơn cơ chế làm việc xử lý mã lệnh của RISC-V, cũng như cơ chế RISC-V pipeline, những ưu và nhược điểm của cơ chế pipeline và những phương án khắc phục. Nhóm đã thực hiện được chương trình sử dụng ngôn ngữ System Verilog được mô phỏng trên phần mềm Questa Sim, và cho ra kết quả hoạt động đúng với yêu cầu đầu vào đầu ra, xử lý được vấn đề Hazard. Bên cạnh đó, thuật toán sắp xếp nổi bọt được mô phỏng đúng với yêu cầu.

Báo cáo của em không thể tránh khỏi những sai sót không mong muốn, mong thầy đưa ra những nhận xét để nhóm hoàn thiện tốt hơn và rút ra kinh nghiệm cho những bài tập lớn tiếp theo.

Em xin cảm ơn sự giảng dạy và hướng dẫn tận tình của thầy!

TÀI LIỆU THAM KHẢO