

**ĐẠI HỌC BÁCH KHOA HÀ NỘI**  
**KHOA TOÁN - TIN**



**BÁO CÁO**  
**KIẾN TRÚC MÁY TÍNH**

**Chủ đề: Thiết kế & mô phỏng CPU pipeline 5 tầng  
với Hazard Detection Unit**

Sinh viên: **Nguyễn Khánh Anh** 20237290

**Bùi Trần Hà Bình** 20237304

**Nghiêm Phú Quang Hưng** 20237341

**Dương Quang Minh** 20237362

**Nguyễn Anh Văn** 20237404

Lớp: **163635**

Giảng viên giảng dạy: **PGS.TS.Nguyễn Đình Hân**

**HÀ NỘI, 1/2026**

# MỤC LỤC

LỜI GIỚI THIỆU . . . . .	
<b>1 GIỚI THIỆU</b>	<b>1</b>
1.1 Mô tả dự án và yêu cầu bài toán . . . . .	1
1.1.1 Tên đề tài và mục tiêu . . . . .	1
1.1.2 Yêu cầu hệ thống . . . . .	1
1.2 Cơ sở lý thuyết: Kiến trúc tập lệnh MIPS . . . . .	3
1.2.1 Tổng quan về MIPS ISA . . . . .	3
1.2.2 Tập thanh ghi . . . . .	4
1.2.3 Định dạng lệnh . . . . .	6
1.3 Cơ sở thiết kế . . . . .	9
1.3.1 Thiết kế CPU đơn chu kỳ . . . . .	9
1.3.2 Kỹ thuật đường ống (Pipelining) . . . . .	10
1.3.3 Các loại xung đột và kỹ thuật xử lý . . . . .	13
<b>2 PHƯƠNG PHÁP LUẬN</b>	<b>16</b>
2.1 Công cụ và môi trường phát triển . . . . .	16
2.1.1 Ngôn ngữ thiết kế . . . . .	16
2.1.2 Môi trường mô phỏng và kiểm thử . . . . .	16
2.1.3 Công cụ hỗ trợ . . . . .	16
2.1.4 Phạm vi thiết kế . . . . .	16
2.2 Cấu trúc mã nguồn . . . . .	16
2.2.1 Tổ chức thư mục và Module cấp cao . . . . .	17
2.2.2 Thiết kế phân tầng . . . . .	18
2.2.3 Môi trường kiểm thử (Testbench) . . . . .	18
2.3 Thiết kế kiến trúc hệ thống . . . . .	19
2.3.1 Đặc tả tập lệnh hỗ trợ . . . . .	19
2.3.2 Yêu cầu về tài nguyên và hiệu năng hệ thống . . . . .	19
2.3.3 Giao diện vào/ra . . . . .	20
2.4 Thiết kế Datapath Pipeline . . . . .	21
2.4.1 Sơ đồ kết nối tổng quát . . . . .	21

2.4.2	Chi tiết thiết kế phần cứng từng tầng . . . . .	22
2.4.3	Hệ thống thanh ghi đường ống . . . . .	27
2.5	Thiết kế khối điều khiển (Control Unit) . . . . .	27
2.5.1	Bộ điều khiển chính (Main Control) . . . . .	27
2.5.2	Bộ điều khiển ALU (ALU Control) . . . . .	28
2.5.3	Bảng tín hiệu điều khiển tổng quát . . . . .	28
2.6	Giải pháp xử lý xung đột . . . . .	28
2.6.1	Phân tích Xung đột dữ liệu (Data Hazards) . . . . .	29
2.6.2	Giải pháp Chuyển tiếp dữ liệu (Forwarding Unit) . . . . .	29
2.6.3	Giải pháp Dừng đường ống (Hazard Detection Unit) . . . . .	31
2.6.4	Xung đột điều khiển (Control Hazard) . . . . .	32
2.7	Hiện thực các module chính . . . . .	36
2.7.1	Module ALU (Khối thực thi) . . . . .	36
2.7.2	Module Register File & Decode Wrapper . . . . .	37
2.7.3	Module Control Unit & Address Calculation . . . . .	37
2.7.4	Module Hazard Handling (Xử lý xung đột) . . . . .	38
2.7.5	Module Pipeline Registers . . . . .	40
<b>3</b>	<b>KIỂM THỬ VÀ KẾT QUẢ</b>	<b>41</b>
3.1	Kiểm thử và Đánh giá kết quả . . . . .	41
3.1.1	Chiến lược kiểm thử . . . . .	41
3.1.2	Cấu hình Testbench . . . . .	41
3.1.3	Kết quả mô phỏng . . . . .	42
3.1.4	Đánh giá và Kết luận . . . . .	43
<b>4</b>	<b>KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN</b>	<b>45</b>
4.1	Kết quả đạt được . . . . .	45
4.2	Hạn chế . . . . .	45
4.3	Hướng phát triển . . . . .	46
4.3.1	Tích hợp cơ chế xử lý ngoại lệ (Exceptions) . . . . .	46
4.3.2	Nâng cao khả năng tính toán song song (Parallelism) . . . . .	46
<b>PHỤ LỤC</b>		<b>48</b>

<b>A PHÂN CÔNG CÔNG VIỆC</b>	<b>48</b>
<b>B KẾ HOẠCH LÀM VIỆC CỦA NHÓM</b>	<b>49</b>
<b>C HƯỚNG DẪN KHAI THÁC TÀI LIỆU ĐÍNH KÈM</b>	<b>50</b>
<b>TÀI LIỆU THAM KHẢO</b>	<b>51</b>



## **DANH MỤC HÌNH VẼ**

Hình 2.1	Cấu trúc cây thư mục dự án trong ModelSim . . . . .	17
Hình 2.2	Sơ đồ datapath Pipeline 5 tầng tích hợp xử lý Hazard . . . . .	21
Hình 2.3	Sơ đồ datapath tầng IF . . . . .	22
Hình 2.4	Sơ đồ datapath tầng ID . . . . .	23
Hình 2.5	Sơ đồ datapath tầng EX . . . . .	24
Hình 2.6	Sơ đồ datapath tầng MEM . . . . .	25
Hình 2.7	Sơ đồ datapath tầng WB . . . . .	26
Hình 2.8	Sơ đồ nguyên lý cơ chế chuyển tiếp dữ liệu . . . . .	30
Hình 2.9	Sơ đồ Datapath tích hợp bộ Forwarding Unit . . . . .	31
Hình 2.10	Datapath hoàn chỉnh tích hợp cả Hazard Detection và Forwarding . . . . .	32
Hình 2.11	Sơ đồ Datapath tích hợp Forwarding, Hazard Detection và Early Branch . . . . .	33
Hình 3.1	Dạng sóng mô phỏng quá trình tính tổng từ 1 đến 10 . . . . .	42

## **DANH MỤC BẢNG BIỂU**

Bảng 1.1	Chức năng cơ bản của các thanh ghi quan trọng . . . . .	5
Bảng 2.1	Danh sách chi tiết các lệnh MIPS được hiện thực hóa . . . . .	19
Bảng 2.2	Bảng chân lý tín hiệu điều khiển . . . . .	28
Bảng 2.3	Diễn biến xử lý Load-Use Hazard theo chu kỳ . . . . .	32
Bảng 2.4	So sánh hoạt động đường ống: Taken vs Not Taken . . . . .	34
Bảng 2.5	Tổng kết chiến lược xử lý Control Hazard . . . . .	36
Bảng 3.1	So sánh hiệu năng thực tế và lý thuyết . . . . .	43
Bảng A.1	Bảng phân công nhiệm vụ các thành viên . . . . .	48
Bảng B.1	Tiến độ thực hiện dự án . . . . .	49

# LỜI GIỚI THIỆU

Trong kỷ nguyên số, bộ vi xử lý (CPU) được xem là trái tim của mọi hệ thống máy tính, từ các thiết bị cá nhân cho đến những siêu máy tính mạnh mẽ nhất. Hiệu năng của CPU là yếu tố then chốt, quyết định trực tiếp đến tốc độ và khả năng xử lý của toàn bộ hệ thống. Tuy nhiên, các kiến trúc CPU đơn chu kỳ truyền thống ngày càng bộc lộ những hạn chế về mặt hiệu suất khi phải xử lý tuần tự từng lệnh một, gây lãng phí tài nguyên phần cứng. Bài toán đặt ra là làm thế nào để phá vỡ rào cản tuần tự này, cho phép CPU thực thi nhiều lệnh đồng thời nhằm tối đa hóa thông lượng (throughput) và hiệu năng.

Xuất phát từ yêu cầu đó, nhóm sinh viên chúng em đã thực hiện đồ án môn học Kiến trúc Máy tính với đề tài: **Thiết kế và mô phỏng CPU pipeline 5 tầng với Hazard Detection Unit sử dụng Verilog**. Mục tiêu cốt lõi của đồ án là xây dựng một mô hình CPU hoàn chỉnh, áp dụng kỹ thuật đường ống (pipelining) để chia quá trình thực thi lệnh thành các giai đoạn độc lập, đồng thời giải quyết triệt để các xung đột (hazards) phát sinh trong quá trình vận hành song song để đảm bảo tính chính xác của kết quả.

Trong quá trình thực hiện đồ án này, chúng em đã tập trung vận dụng các kiến thức nền tảng của học phần để thiết kế và hiện thực các thành phần cốt lõi, bao gồm Datapath 5 tầng (IF, ID, EX, MEM, WB), Control Unit, các thanh ghi pipeline, và đặc biệt là Hazard Unit để xử lý các xung đột dữ liệu và xung đột điều khiển. Toàn bộ hệ thống được mô tả bằng ngôn ngữ mô tả phần cứng Verilog và được kiểm tra, mô phỏng chi tiết bằng các công cụ chuyên dụng như ModelSim để chứng minh tính đúng đắn của thiết kế.

Đây không chỉ là một bài tập lớn nhằm áp dụng kiến thức từ học phần "Kiến trúc Máy tính" vào việc giải quyết một bài toán thiết kế cụ thể, mà còn là cơ hội để chúng em rèn luyện kỹ năng tư duy thiết kế hệ thống phần cứng, làm việc nhóm và phát triển một sản phẩm mô phỏng hoàn chỉnh từ lý thuyết đến thực tiễn.

Nhóm sinh viên thực hiện xin trân trọng gửi lời cảm ơn sâu sắc đến giảng viên phụ trách học phần PGS.TS. Nguyễn Đình Hân, đã tận tình giảng dạy, định hướng và cung cấp những góp ý vô cùng quý báu trong suốt quá trình chúng em thực hiện đồ án này.

Mặc dù nhóm đã rất nỗ lực để hoàn thiện sản phẩm một cách tốt nhất trong khuôn khổ thời gian và kiến thức cho phép, song chắc chắn không thể tránh khỏi những thiếu sót và hạn chế. Chúng em rất mong nhận được những ý kiến đóng góp quý báu từ thầy cô và các bạn để đồ án có thể được cải thiện và phát triển hơn nữa trong tương lai.

Hà Nội, ngày 14 tháng 01 năm 2025

Nhóm 3

# CHƯƠNG 1. GIỚI THIỆU

## 1.1 Mô tả dự án và yêu cầu bài toán

Trong kỷ nguyên số hiện nay, bộ vi xử lý (CPU) đóng vai trò trung tâm như “trái tim” của mọi hệ thống tính toán. Việc thiết kế và tổ chức hàng tỷ transistor để tạo nên sức mạnh xử lý là một thách thức lớn trong kỹ thuật máy tính. Đồ án này tập trung vào việc tái hiện và hiện thực hóa kiến trúc phần cứng nền tảng để giải quyết bài toán đó.

### 1.1.1 Tên đề tài và mục tiêu

**Tên đề tài:** Thiết kế và mô phỏng bộ vi xử lý MIPS Pipeline 5 tầng với khối phát hiện xung đột (Hazard Detection Unit).

#### Mục tiêu của đồ án:

Mục tiêu chính của đồ án này là thiết kế một bộ xử lý 32-bit dựa trên kiến trúc MIPS (Microprocessor without Interlocked Pipelined Stages) - một kiến trúc RISC (Reduced Instruction Set Computer) kinh điển, sử dụng ngôn ngữ mô tả phần cứng Verilog HDL. Cụ thể, đồ án tập trung vào các mục tiêu sau:

- Chuyển đổi các kiến thức lý thuyết về tổ chức máy tính và kiến trúc vi xử lý thành mô hình phần cứng thực tế chạy được trên Verilog.
- Hiểu và hiện thực hóa kiến trúc đường ống (Pipelining) để tăng thông lượng (throughput) xử lý lệnh so với kiến trúc đơn chu kỳ.
- Xây dựng đầy đủ 5 tầng xử lý: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM) và Write Back (WB).
- Phân tích và giải quyết các vấn đề xung đột (Hazards) trong pipeline, bao gồm xung đột dữ liệu (Data Hazard) và xung đột điều khiển (Control Hazard).
- Thiết kế khối Hazard Detection Unit để phát hiện xung đột và thực hiện kỹ thuật dừng đường ống (Stall) khi cần thiết.
- Thiết kế khối Forwarding Unit để chuyển tiếp dữ liệu, giảm thiểu số chu kỳ phải dừng pipeline.
- Kiểm chứng thiết kế thông qua mô phỏng dạng sóng trên ModelSim.

### 1.1.2 Yêu cầu hệ thống

Để đảm bảo bộ vi xử lý hoạt động chính xác theo nguyên lý Pipeline MIPS 32-bit, hệ thống cần đáp ứng các yêu cầu kỹ thuật sau:

### 1.1.2.1 Yêu cầu về kiến trúc tập lệnh

Hệ thống phải hỗ trợ các lệnh cơ bản 32-bit của MIPS thuộc ba định dạng:

- **R-Type (Register):** Các lệnh tính toán số học và logic như ADD, SUB, AND, OR, SLT.
- **I-Type (Immediate):** Các lệnh thao tác với hằng số và bộ nhớ như ADDI, ORI, XORI, LW (Load Word), SW (Store Word), BEQ (Branch on Equal).
- **J-Type (Jump):** Lệnh nhảy không điều kiện J.

### 1.1.2.2 Yêu cầu về Datapath (Đường dữ liệu)

- Datapath phải được chia tách rõ ràng thành 5 tầng độc lập.
- Giữa các tầng phải có các thanh ghi đệm (Pipeline Registers): IF / ID, ID / EX, EX / MEM, MEM / WB để lưu trữ trạng thái lệnh và dữ liệu điều khiển.
- Bộ nhớ lệnh (Instruction Memory) và bộ nhớ dữ liệu (Data Memory) phải được tách biệt theo kiến trúc Harvard nhằm tránh xung đột cấu trúc khi nạp lệnh và truy xuất dữ liệu đồng thời.

### 1.1.2.3 Yêu cầu về xử lý xung đột

Hệ thống phải có khả năng tự động phát hiện và xử lý các tình huống xung đột sau:

- **RAW (Read After Write):** Khi một lệnh đọc thanh ghi chưa được ghi lại bởi lệnh trước đó.
- **Load-Use Hazard:** Khi lệnh ngay sau phụ thuộc vào kết quả của lệnh LW. Khi đó hệ thống phải kích hoạt tín hiệu Stall (giữ nguyên PC và thanh ghi IF / ID) và chèn lệnh rỗng (NOP / Bubble) vào tầng EX.
- **Control Hazard:** Khi gặp lệnh rẽ nhánh (BEQ), hệ thống phải xác định địa chỉ nhảy đúng và xả (flush) các lệnh sai đang tồn tại trong pipeline nếu cần thiết.

### 1.1.2.4 Yêu cầu về công cụ và môi trường

- Ngôn ngữ thiết kế: Verilog HDL.
- Công cụ mô phỏng: ModelSim.
- Đầu ra: Kết quả mô phỏng dạng sóng chứng minh dữ liệu đi qua đúng 5 tầng pipeline và hệ thống dừng (stall) chính xác khi xảy ra xung đột.

## 1.2 Cơ sở lý thuyết: Kiến trúc tập lệnh MIPS

### 1.2.1 Tổng quan về MIPS ISA

Theo giáo trình *Computer Organization and Design* của Patterson và Hennessy, kiến trúc tập lệnh được định nghĩa là một giao diện trừu tượng nằm giữa phần cứng thực thi và phần mềm cấp thấp nhất [1].

ISA bao gồm tất cả các thông tin cần thiết để một chương trình ngôn ngữ máy hoạt động chính xác, bao gồm tập lệnh, hệ thống thanh ghi, cách truy cập bộ nhớ và các cơ chế vào/ra (I/O). Vai trò cốt lõi của ISA là tạo ra sự nhất quán, cho phép các phiên bản phần cứng khác nhau (về giá thành hoặc hiệu năng) có thể chạy cùng một phần mềm [1].

#### 1.2.1.1 Các thành phần chính của MIPS ISA

Dựa trên các nguyên lý thiết kế được trình bày trong Chương 2 của tài liệu tham khảo [1], kiến trúc MIPS bao gồm các khía cạnh chính sau:

- Lớp ISA và Thanh ghi:** MIPS sử dụng kiến trúc Load-Store với tập thanh ghi đa dụng (GPR) gồm 32 thanh ghi 32-bit.
- Kiểu dữ liệu và Bộ nhớ:** Hỗ trợ các kiểu dữ liệu byte, halfword, word (32-bit) với mô hình địa chỉ byte.
- Các phép toán:** Tập lệnh bao gồm các nhóm lệnh số học, logic, chuyển dữ liệu và điều khiển luồng cơ bản.
- Mã hóa lệnh:** MIPS sử dụng độ dài lệnh cố định 32-bit với 3 định dạng chính (R-type, I-type, J-type) để đơn giản hóa quá trình giải mã phần cứng.

#### 1.2.1.2 Mối liên hệ giữa MIPS, RISC và Pipelining

MIPS là một ví dụ điển hình của kiến trúc RISC, được thiết kế với triết lý đơn giản hóa tập lệnh để tối ưu hóa hiệu năng. Sự đơn giản và đồng nhất trong định dạng lệnh của MIPS là yếu tố then chốt giúp việc triển khai kỹ thuật đường ống trở nên hiệu quả, cho phép tăng tốc độ thực thi lệnh như được phân tích chi tiết trong Chương 4 của tài liệu [1].

#### 1.2.1.3 Nguyên lý hoạt động Load-Store và thao tác dữ liệu

**Thao tác với bộ nhớ** Trong kiến trúc MIPS, chỉ có duy nhất các lệnh chuyển đổi dữ liệu mới được phép truy cập trực tiếp vào bộ nhớ [1]. Cụ thể, các lệnh này bao gồm:

- Lệnh Tải (Load):** Chuyển dữ liệu từ bộ nhớ vào thanh ghi.

– Ví dụ: `lw` (load word), `lb` (load byte), `lh` (load halfword).

- **Lệnh Lưu trữ (Store):** Chuyển dữ liệu từ thanh ghi ra bộ nhớ.

– Ví dụ: `sw` (store word), `sb` (store byte), `sh` (store halfword).

Bộ nhớ trong MIPS được tổ chức theo mô hình **bộ nhớ phẳng**, là một mảng lớn một chiều được đánh địa chỉ theo từng byte. Dù bộ nhớ được đánh địa chỉ theo byte, các lệnh `lw` và `sw` thường chỉ hoạt động trên dữ liệu phải được **căn chỉnh** theo ranh giới từ, tức là địa chỉ phải là bội số của 4 [1].

**Thao tác với thanh ghi** Ngược lại với lệnh Load/Store, các lệnh tính toán và logic trong MIPS chỉ được phép thao tác trên các giá trị nằm trong các **thanh ghi**. Điều này tạo nên đặc trưng của kiến trúc **Load-Store**.

Để thực hiện một phép toán cơ bản như cộng hai số nguyên nằm trong bộ nhớ, CPU MIPS phải thực hiện một chuỗi các bước tuần tự:

1. Sử dụng lệnh Load (`lw`) để tải toán hạng thứ nhất từ bộ nhớ vào một thanh ghi.
2. Sử dụng lệnh Load (`lw`) để tải toán hạng thứ hai từ bộ nhớ vào một thanh ghi khác.
3. Sử dụng lệnh số học (`add`) để thực hiện phép cộng trên hai thanh ghi đó và lưu kết quả vào thanh ghi đích.
4. (Tùy chọn) Sử dụng lệnh Store (`sw`) để lưu kết quả từ thanh ghi đích trở lại bộ nhớ.

MIPS là một kiến trúc **3-địa chỉ**. Hầu hết các lệnh số học và logic đều có định dạng quy định ba thanh ghi: hai thanh ghi nguồn ( $R_s, R_t$ ) và một thanh ghi đích ( $R_d$ ). MIPS cung cấp **32 thanh ghi đa dụng 32-bit**, giúp tăng tốc độ truy cập dữ liệu và đơn giản hóa việc triển khai so với việc truy cập bộ nhớ liên tục [1].

**Kết luận** Sự phân tách rõ ràng giữa các lệnh Load/Store và các lệnh thao tác trên thanh ghi là một nguyên tắc thiết kế RISC cơ bản. Nó giữ cho các lệnh đơn giản, cố định độ dài (**32 bit**), và dễ dàng tối ưu hóa cho kỹ thuật đường ống được trình bày ở các chương sau.

### 1.2.2 Tập thanh ghi

Kiến trúc Tập lệnh MIPS (MIPS ISA) tuân thủ nguyên tắc thiết kế **RISC** (Reduced Instruction Set Computer) bằng cách tập trung các thao tác xử lý dữ liệu vào một tập hợp thanh ghi hữu hạn, nhằm tối ưu hóa tốc độ truy cập và đơn giản hóa thiết kế đường ống (*pipelining*).

### 1.2.2.1 Số lượng và kích thước

Kiến trúc MIPS-32 sử dụng **32 thanh ghi đa dụng**. Đây là các vị trí lưu trữ dữ liệu tốc độ cao được tích hợp trực tiếp trên chip CPU, và là thành phần bắt buộc phải sử dụng để thực hiện các phép toán số học.

- **Kích thước Thanh ghi:** Mỗi thanh ghi trong kiến trúc MIPS có kích thước **32 bit**.
- **Word (Tù)**: Trong kiến trúc MIPS, một nhóm 32 bit được gọi là một **word** (tù), tương ứng với kích thước tự nhiên để truy cập dữ liệu và kích thước của một thanh ghi.

Mặc dù MIPS-32 là phiên bản 32-bit, kiến trúc này cũng có phiên bản MIPS-64 hỗ trợ 32 thanh ghi 64-bit.

### 1.2.2.2 Quy ước sử dụng thanh ghi

Do số lượng thanh ghi giới hạn (32 thanh ghi), việc quản lý và sử dụng chúng được quy định chặt chẽ thông qua các quy ước gọi thủ tục. Các quy ước này giúp cho việc biên dịch và thực thi các chương trình phức tạp diễn ra hiệu quả. Dưới đây là chức năng cơ bản của các thanh ghi quan trọng:

**Bảng 1.1:** Chức năng cơ bản của các thanh ghi quan trọng

Thanh ghi	Ký hiệu	Vai trò cơ bản
Thanh ghi Zero	zero (0)	Luôn giữ giá trị <b>hằng số 0</b> .
Thanh ghi tạm thời	t0–t9 (8–15, 24–25)	Dùng cho các giá trị tạm thời trong quá trình tính toán của chương trình.
Thanh ghi lưu trữ	s0–s7 (16–23)	Dùng để lưu trữ các biến tương ứng với các biến trong chương trình cấp cao (ví dụ: C/Java).
Thanh ghi giá trị	v0–v1 (2–3)	Dùng để trả về các giá trị (kết quả) của hàm/thủ tục.
Thanh ghi đối số	a0–a3 (4–7)	Dùng để truyền bốn tham số đầu tiên cho hàm/thủ tục.
Thanh ghi Trả về	ra (31)	Lưu địa chỉ quay trở lại khi gọi một thủ tục.
Thanh ghi Assembler	at (1)	Dành riêng cho trình hợp dịch ( <i>assembler</i> ) để xử lý các hằng số lớn hoặc các lệnh giả.

**Thanh ghi đặc biệt: Program Counter (PC)** Bên cạnh 32 thanh ghi đa dụng, CPU MIPS còn chứa các thanh ghi đặc biệt quan trọng, trong đó nổi bật là **Program Counter (PC)**.

- **Định nghĩa và Vai trò:** PC là một thanh ghi đặc biệt, không nằm trong 32 thanh ghi GPR mà lập trình viên có thể truy cập trực tiếp. Nó chứa **địa chỉ bộ nhớ của lệnh tiếp theo** sẽ được nạp và thực thi. Đây là một phần cốt yếu của ý tưởng chương trình được lưu trữ.

#### • **Hoạt động trong Chu kỳ Lệnh:**

- Trong chu kỳ nạp lệnh (*Fetch cycle*), PC cung cấp địa chỉ cho bộ nhớ lệnh.
- Sau khi lệnh được nạp, PC thường được tăng thêm **4 byte** (vì mỗi lệnh MIPS là 32 bit, hay 4 byte) để trả đến lệnh tiếp theo theo trình tự tuần tự.
- Khi có lệnh rẽ nhánh (*branch*) hoặc nhảy (*jump*), giá trị của PC sẽ được cập nhật để chuyển quyền điều khiển chương trình đến một địa chỉ lệnh khác. Ví dụ, lệnh `jal` (*jump and link*) lưu địa chỉ lệnh tiếp theo (PC + 4) vào thanh ghi \$ra trước khi chuyển hướng thực thi.

Sự phân chia rõ ràng giữa dữ liệu được thao tác trong thanh ghi và dữ liệu được lưu trữ trong bộ nhớ (chỉ truy cập thông qua các lệnh Load và Store) là nguyên tắc nền tảng của MIPS ISA.

### 1.2.3 Định dạng lệnh

Việc thiết kế kiến trúc tập lệnh của MIPS dựa trên nguyên tắc cơ bản là **Tính đơn giản ưu tiên Tính đồng nhất** (*Simplicity favors regularity*). Điều này được thể hiện rõ nhất qua việc tất cả các lệnh MIPS đều có độ dài cố định là **32 bit**, và chỉ sử dụng ba định dạng lệnh cơ bản (R, I, J). Đối với một chuyên gia Kiến trúc máy tính, việc hiểu các định dạng này là trọng tâm, bởi vì chúng xác định cách thức tầng Giải mã lệnh (Instruction Decode – ID) trong CPU “cắt bit” và xác định các toán hạng cần thiết cho việc thực thi.

Dưới đây là chi tiết về ba định dạng lệnh cốt lõi trong MIPS ISA:

#### 1.2.3.1 Định dạng R-type

Định dạng R-type (*Register*) được thiết kế cho các lệnh thao tác hoàn toàn trên các thanh ghi đa dụng, chủ yếu là các phép toán số học và logic.

#### Cấu trúc (32 bit):

Trường	Opcode	rs	rt	rd	shamt	funct
Kích thước (bits)	6	5	5	5	5	6
Vị trí bit	31:26	25:21	20:16	15:11	10:6	5:0

### Vai trò của các trường:

- **Opcode (6 bits):** Mã hóa hoạt động cơ bản. Đối với hầu hết các lệnh R-format, trường Opcode này được đặt bằng 0.
- **rs (First Register Source):** Thanh ghi nguồn thứ nhất (5 bit, cho phép địa chỉ hóa 32 thanh ghi).
- **rt (Second Register Source/Target):** Thanh ghi nguồn thứ hai (5 bit). Trong định dạng R-type, nó thường là toán hạng nguồn thứ hai.
- **rd (Register Destination):** Thanh ghi đích (5 bit), nơi chứa kết quả của phép toán.
- **shamt (Shift Amount):** Lượng dịch chuyển (5 bit), chỉ được sử dụng trong các lệnh dịch bit (ví dụ: sll, srl). Nếu không phải lệnh dịch, giá trị này bằng 0.
- **funct (Function Code - 6 bits):** Đây là trường quan trọng nhất trong R-type. Khi Opcode bằng 0, trường funct sẽ xác định chính xác hoạt động mà Đơn vị Số học/Logic (ALU) cần thực hiện (ví dụ: 32<sub>ten</sub> cho add, 34<sub>ten</sub> cho sub).

#### 1.2.3.2 Định dạng I-type

Định dạng I-type (*Immediate*) được sử dụng rộng rãi cho các lệnh có sự tham gia của một **giá trị hàng số** hoặc các lệnh truy cập bộ nhớ và rẽ nhánh điều kiện.

### Cấu trúc (32 bit):

Trường	Opcode	rs	rt	Immediate / Address
Kích thước (bits)	6	5	5	16
Vị trí bit	31:26	25:21	20:16	15:0

### Vai trò của các trường:

- **Opcode (6 bits):** Xác định loại lệnh (ví dụ: 8<sub>ten</sub> cho addi, 35<sub>ten</sub> cho lw, 43<sub>ten</sub> cho sw, 4<sub>ten</sub> cho beq).
- **rs (Source / Base Register):** Thường là thanh ghi nguồn thứ nhất. Trong các lệnh Load/Store/Branch, nó là **thanh ghi cơ sở** chứa địa chỉ nền.
- **rt (Destination / Source / Comparison Register):** Có vai trò kép:
  - Là thanh ghi đích trong lệnh Immediate (ví dụ: addi) hoặc lệnh Load (lw).

- Là thanh ghi nguồn (hoặc thanh ghi so sánh thứ hai) trong lệnh Store (`sw`) hoặc lệnh Branch (`beq`).
- **Immediate / Address (16 bits):** Đây là trường 16-bit quan trọng nhất. Tùy thuộc vào Opcode, nó được giải thích khác nhau:
  - Hằng số tức thời:** Là giá trị hằng số 16 bit có dấu, được **mở rộng dấu** (*sign-extended*) lên 32 bit để sử dụng trong các phép toán như `addi`.
  - Độ lệch địa chỉ:** Là độ lệch 16 bit có dấu được cộng vào giá trị trong thanh ghi cơ sở (`rs`) để tạo ra địa chỉ bộ nhớ hiệu dụng cho các lệnh truy cập bộ nhớ (`lw, sw`).
  - Địa chỉ mục tiêu tương đối (PC-relative Offset):** Là độ lệch 16 bit có dấu cho các lệnh rẽ nhánh điều kiện (`beq, bne`). Giá trị này được dịch trái 2 bit (nhân 4) và cộng vào PC+4 để tính địa chỉ mục tiêu.

Việc giới hạn Immediate/Address là 16 bit khiến các hằng số hoặc địa chỉ truy cập bị giới hạn, nhưng điều này giúp giữ độ dài lệnh là 32 bit.

#### 1.2.3.3 Định dạng J-type

Định dạng J-type (*Jump*) được sử dụng cho các lệnh điều khiển luồng không điều kiện, như `j` (*jump*) và `jal` (*jump and link*).

#### Cấu trúc (32 bit):

Trường	Opcode	Target Address
<b>Kích thước (bits)</b>	6	26
<b>Vị trí bit</b>	31:26	25:0

#### Vai trò của các trường:

- **Opcode (6 bits):** Xác định hoạt động nhảy (`j` có Opcode  $2_{ten}$ ).
- **Target Address (26 bits):** Trường này xác định phần lớn địa chỉ mục tiêu. Để tạo ra địa chỉ 32-bit đầy đủ, địa chỉ 26-bit này được xử lý như sau:
  1. Dịch trái 2 bit (thêm 00 vào cuối), biến nó thành địa chỉ 28-bit.
  2. Nối với 4 bit cao nhất của địa chỉ lệnh tiếp theo ( $PC + 4$ ).

Cơ chế này cho phép lệnh nhảy có thể chuyển đến bất kỳ địa chỉ nào trong khối 256 MB mà chương trình hiện tại đang được đặt. Lệnh `jal` sử dụng định dạng này, đồng thời lưu địa chỉ quay lại ( $PC + 4$ ) vào thanh ghi `$ra` (*register 31*).

## 1.3 Cơ sở thiết kế

### 1.3.1 Thiết kế CPU đơn chu kỳ

#### 1.3.1.1 Nguyên lý thiết kế Datapath

Thiết kế đơn chu kỳ (Single-Cycle) là mô hình kiến trúc cơ bản nhất, trong đó mỗi lệnh được thực thi trọn vẹn trong **đúng một chu kỳ xung nhịp** ( $CPI = 1$ ). Để đạt được điều này, thiết kế Datapath phải tuân thủ nghiêm ngặt các nguyên tắc phần cứng sau:

- **Phương pháp xung nhịp (Clocking Methodology):** Hệ thống sử dụng cơ chế kích hoạt theo cạnh lên (*edge-triggered*). Mọi cập nhật trạng thái (ghi vào thanh ghi, ghi bộ nhớ) chỉ xảy ra tại sườn dương của xung nhịp. Điều này giúp ngăn chặn hiện tượng "đua tín hiệu" và đảm bảo chu trình hoạt động ổn định: Đọc dữ liệu → Xử lý qua logic tổ hợp → Ghi kết quả [1].
- **Không tái sử dụng tài nguyên:** Do chu kỳ lệnh chỉ kéo dài trong 1 clock, một thành phần phần cứng không thể được sử dụng hai lần trong cùng một lệnh. Do đó, phần cứng phải được nhân bản: cần có các bộ cộng (Adder) riêng để tính toán địa chỉ PC thay vì dùng chung ALU chính.
- **Tách biệt bộ nhớ:** Để tránh xung đột cấu trúc khi vừa nạp lệnh vừa truy cập dữ liệu trong cùng một chu kỳ, hệ thống cần tách hoàn toàn **Bộ nhớ lệnh** (Instruction Memory) và **Bộ nhớ dữ liệu** (Data Memory).

Các thành phần cốt lõi của Datapath bao gồm:

1. **Program Counter (PC):** Thanh ghi 32-bit chứa địa chỉ của lệnh hiện tại.
2. **Instruction Memory:** Nơi lưu trữ mã máy của chương trình, đầu ra là mã lệnh 32-bit tương ứng với địa chỉ PC.
3. **Register File:** Tập hợp 32 thanh ghi đa dụng ( $32 \times 32\text{-bit}$ ), cho phép đọc 2 cổng và ghi 1 cổng đồng thời.
4. **ALU (Arithmetic Logic Unit):** Thực hiện các phép toán số học (+, -) và logic (AND, OR, SLT).
5. **Data Memory:** Bộ nhớ chỉ được truy cập bởi các lệnh Load/Store.

#### 1.3.1.2 Thiết kế Control Unit

Khối điều khiển (Control Unit) đóng vai trò là "bộ não", chịu trách nhiệm giải mã lệnh và sinh ra các tín hiệu điều khiển cho Datapath. Trong thiết kế này, nhóm sử dụng chiến lược **giải mã phân cấp**:

- **Main Control:** Giải mã trường Opcode (6 bit cao nhất [31:26]) để sinh ra các tín hiệu điều khiển chính như: RegDst, Branch, MemRead, MemToReg, ALUOp, MemWrite, ALUSrc, RegWrite.
- **ALU Control:** Đây là bộ giải mã cục bộ, chịu trách nhiệm điều khiển trực tiếp ALU. Nó nhận tín hiệu ALUOp từ Main Control và kết hợp với trường funct (6 bit thấp nhất [5:0]) của lệnh R-type để quyết định phép toán cụ thể (ADD, SUB, AND, OR, SLT).

**Điểm cải tiến trong thiết kế:** Thay vì sử dụng ALUOp 2-bit như lý thuyết cơ bản, nhóm thiết kế sử dụng **ALUOp 3-bit**. Việc mở rộng này giúp bộ điều khiển phân biệt rõ ràng hơn các nhóm lệnh I-Type (như addi, andi, ori) ngay từ giai đoạn giải mã, tăng tính linh hoạt và chính xác cho tín hiệu điều khiển ALU.

#### 1.3.1.3 Đánh giá và Hạn chế

Mặc dù thiết kế đơn chu kỳ đơn giản và dễ hiểu, nó bộc lộ những hạn chế nghiêm trọng khiến nó không còn được sử dụng trong các bộ vi xử lý hiện đại:

- **Chu kỳ xung nhịp dài:** Thời gian của một chu kỳ bị giới hạn bởi lệnh chậm nhất. Trong MIPS, đó là lệnh LW (Load Word) khi phải đi qua tuần tự 5 khối: IM → Reg → ALU → DM → Reg. Điều này kéo lùi hiệu năng của các lệnh nhanh hơn (như ADD).
- **Hiệu suất thấp:** Không thể đạt tần số hoạt động cao (GHz), vi phạm nguyên tắc "Make the common case fast".
- **Lãng phí tài nguyên:** Tại bất kỳ thời điểm nào trong chu kỳ, phần lớn các đơn vị chức năng đều ở trạng thái nhàn rỗi.

Đây chính là động lực để nhóm chuyển sang thiết kế kiến trúc **Pipeline 5 tầng** ở chương tiếp theo.

### 1.3.2 Kỹ thuật đường ống (Pipelining)

#### 1.3.2.1 Tổng quan và Nguyên lý hoạt động

Kỹ thuật đường ống (Pipelining) là một kỹ thuật hiện thực hóa phần cứng trong đó nhiều câu lệnh được thực thi chồng gối lên nhau theo thời gian. Theo giáo trình *Computer Organization and Design*, đây là kỹ thuật gần như phổ quát trong các bộ vi xử lý hiện đại nhằm khai thác khả năng xử lý song song ở cấp độ lệnh (Instruction-Level Parallelism - ILP) [1].

Khác với phương pháp thực thi tuần tự - nơi mỗi lệnh phải hoàn thành trọn vẹn trước khi lệnh tiếp theo bắt đầu, kỹ thuật đường ống cho phép các công đoạn khác

nhau của nhiều lệnh diễn ra đồng thời. Mục tiêu cốt lõi của kỹ thuật này là tối đa hóa việc sử dụng tài nguyên phần cứng, từ đó tăng thông lượng tổng thể của hệ thống.

**Nghịch lý Pipeline** Một điểm quan trọng cần lưu ý trong kỹ thuật này là sự khác biệt giữa *Độ trễ* và *Thông lượng*:

- **Độ trễ (Latency):** Thời gian để xử lý trọn vẹn một lệnh riêng lẻ **không giảm** (thậm chí có thể tăng nhẹ do các chi phí quản lý đường ống).
- **Thông lượng (Throughput):** Số lượng lệnh hoàn thành trong một đơn vị thời gian **tăng lên đáng kể**.

Về mặt lý thuyết, tốc độ tăng tốc của một đường ống lý tưởng sẽ tỉ lệ thuận với số tầng của đường ống đó. Tuy nhiên, trong thực tế, tốc độ này luôn thấp hơn lý tưởng do hai nguyên nhân chính:

1. **Độ trễ bổ sung:** Thời gian trễ gây ra bởi quá trình thiết lập và lưu trữ dữ liệu tại các thanh ghi đường ống giữa các tầng.
2. **Sự mất cân bằng:** Các công đoạn xử lý (như truy cập bộ nhớ, tính toán ALU) có thời gian thực thi không hoàn toàn bằng nhau, khiến toàn bộ đường ống phải chạy theo tốc độ của công đoạn chậm nhất.

### 1.3.2.2 Kiến trúc Pipeline 5 tầng trong MIPS

Áp dụng lý thuyết trên vào đồ án, nhóm thiết kế xây dựng bộ vi xử lý MIPS 32-bit dựa trên mô hình đường ống chuẩn bao gồm 5 tầng độc lập. Mỗi tầng đảm nhiệm một chức năng chuyên biệt trong chu trình xử lý lệnh:

1. **Nạp lệnh (Instruction Fetch - IF):** CPU truy cập bộ nhớ lệnh (Instruction Memory) để đọc lệnh tại địa chỉ PC hiện tại, đồng thời tăng giá trị PC (PC + 4) để chuẩn bị cho lệnh tiếp theo.
2. **Giải mã và Đọc thanh ghi (Instruction Decode - ID):** Lệnh 32-bit được giải mã để xác định khối điều khiển (Control Unit) cần làm gì. Đồng thời, các giá trị từ tập thanh ghi (Register File) được đọc ra dựa trên các trường địa chỉ trong lệnh.
3. **Thực thi (Execute - EX):** ALU thực hiện các phép toán số học/logic (đối với lệnh R-type), tính toán địa chỉ bộ nhớ (đối với lệnh Load/Store) hoặc so sánh điều kiện rẽ nhánh (đối với lệnh Branch).
4. **Truy cập bộ nhớ (Memory Access - MEM):** Giai đoạn này chỉ dành cho các lệnh Load/Store để đọc hoặc ghi dữ liệu vào Bộ nhớ dữ liệu (Data Memory). Các lệnh khác sẽ bỏ qua bước này.

5. **Ghi ngược (Write Back - WB):** Kết quả tính toán từ ALU hoặc dữ liệu đọc được từ bộ nhớ sẽ được ghi ngược trở lại vào thanh ghi đích trong Register File.

### 1.3.2.3 Thanh ghi đường ống

Để chia tách Datapath thành 5 tầng hoạt động độc lập trong cùng một chu kỳ xung nhịp, hệ thống sử dụng các **thanh ghi đường ống** đặt giữa các tầng. Các thanh ghi này đóng vai trò như các "vách ngăn" lưu trữ trạng thái và dữ liệu của lệnh, cho phép lệnh trước di chuyển sang tầng kế tiếp và nhường chỗ cho lệnh sau nạp vào tầng trước đó [1].

Hệ thống bao gồm 4 thanh ghi đường ống chính:

1. **IF/ID (64-bit):** Ngăn cách tầng IF và ID. Nó lưu trữ địa chỉ lệnh kế tiếp ( $PC + 4$ ) và mã máy 32-bit của lệnh hiện tại.
2. **ID/EX:** Ngăn cách tầng ID và EX. Nó lưu trữ các giá trị đọc được từ thanh ghi ( $R_s, R_t$ ), hằng số mở rộng dấu và các tín hiệu điều khiển cần thiết cho các tầng sau.
3. **EX/MEM:** Ngăn cách tầng EX và MEM. Nó lưu kết quả tính toán của ALU, cờ Zero (dùng cho rẽ nhánh) và dữ liệu cần ghi vào bộ nhớ (đối với lệnh SW).
4. **MEM/WB:** Ngăn cách tầng MEM và WB. Nó lưu dữ liệu đọc được từ bộ nhớ hoặc kết quả từ ALU để chuẩn bị ghi ngược lại vào Register File.

### 1.3.2.4 Hệ thống điều khiển đường ống

Một thách thức lớn trong thiết kế pipeline là làm sao đảm bảo đúng tín hiệu điều khiển được gửi đến đúng tầng tại đúng thời điểm, khi mà có tối đa 5 lệnh đang cùng hoạt động.

Để giải quyết vấn đề này, nhóm sử dụng chiến lược "**Lan truyền tín hiệu**". Tương tự như dữ liệu, các tín hiệu điều khiển được sinh ra toàn bộ tại tầng ID (bởi Control Unit), sau đó được **nhóm lại** và chuyển tiếp qua các thanh ghi đường ống song hành cùng với lệnh đó [1].

Cụ thể, quá trình lan truyền diễn ra như sau:

- **Tại ID/EX:** Giữ lại các tín hiệu cho tầng EX (như RegDst, ALUOp, ALUSrc) và đẩy tiếp các nhóm tín hiệu còn lại.
- **Tại EX/MEM:** Giữ lại tín hiệu cho tầng MEM (như MemRead, MemWrite) và đẩy tiếp nhóm WB.
- **Tại MEM/WB:** Chỉ còn lại tín hiệu cho tầng WB (như RegWrite, MemToReg) để điều khiển việc ghi dữ liệu.

Chiến lược này đảm bảo rằng mỗi tầng chỉ nhận được tín hiệu điều khiển tương ứng với lệnh mà nó đang xử lý tại chu kỳ đó.

### 1.3.2.5 Luồng dữ liệu và nguyên nhân gây xung đột

Theo thiết kế chuẩn, dữ liệu trong pipeline chảy tuần tự từ trái sang phải: **IF** → **ID** → **EX** → **MEM** → **WB**. Tuy nhiên, để hệ thống hoạt động đúng chức năng, có hai ngoại lệ quan trọng mà luồng dữ liệu phải đi ngược chiều ("từ phải sang trái"):

1. **Cập nhật Register File:** Kết quả từ tầng WB (tầng 5) phải vòng ngược về tầng ID (tầng 2) để ghi vào thanh ghi.
2. **Cập nhật PC:** Địa chỉ rẽ nhánh tính toán tại tầng EX hoặc MEM phải được đưa ngược về tầng IF (tầng 1) để nạp lệnh mới.

Chính sự phụ thuộc giữa các lệnh đi trước và đi sau thông qua các luồng dữ liệu ngược chiều này là nguyên nhân gốc rễ sinh ra các **xung đột (Hazards)** được trình bày trong chương tiếp theo.

### 1.3.3 Các loại xung đột và kỹ thuật xử lý

Trong kiến trúc đường ống, hiệu năng lý tưởng ( $CPI = 1$ ) thường không đạt được do sự xuất hiện của các **xung đột** - tình huống ngăn cản lệnh tiếp theo thực thi đúng trong chu kỳ xung nhịp kế tiếp. Có ba loại xung đột chính: Xung đột cấu trúc, Xung đột dữ liệu và Xung đột điều khiển.

#### 1.3.3.1 Xung đột cấu trúc (Structural Hazards)

**Bản chất vấn đề** Xung đột cấu trúc xảy ra khi phần cứng không đủ tài nguyên để hỗ trợ tất cả các sự kết hợp lệnh cùng thực thi trong một chu kỳ.

Trong thiết kế đường ống MIPS 5 tầng, tình huống xung đột điển hình nhất là sự tranh chấp tài nguyên bộ nhớ:

- Tại tầng **IF (Instruction Fetch)**: CPU cần truy cập bộ nhớ để nạp lệnh.
- Tại tầng **MEM (Memory Access)**: Lệnh Load/Store (như `LW`, `SW`) cũng cần truy cập bộ nhớ để đọc/ghi dữ liệu.

Nếu hệ thống chỉ sử dụng một khối bộ nhớ duy nhất (kiến trúc Von Neumann), hai yêu cầu này sẽ xảy ra đồng thời trong cùng một chu kỳ, dẫn đến xung đột.

**Giải pháp** Khác với các xung đột dữ liệu hay điều khiển (vốn phụ thuộc vào code), xung đột cấu trúc trong MIPS đã được giải quyết triệt để ngay từ **nguyên lý thiết kế hệ thống**. Đồ án này hiện thực hóa chính xác các nguyên tắc đó để đảm bảo đường ống hoạt động không bị tắc nghẽn:

1. **Tuân thủ kiến trúc Harvard:** Để nạp lệnh (tầng IF) và truy xuất dữ liệu (tầng MEM) có thể diễn ra song song trong cùng một chu kỳ, kiến trúc MIPS yêu cầu tách biệt không gian nhớ. Trong mô hình phần cứng của đồ án, điều này được thực hiện bằng cách khởi tạo hai khối nhớ riêng biệt: **Instruction Memory** và **Data Memory**. Điều này giúp loại bỏ hoàn toàn tranh chấp tài nguyên bộ nhớ.
2. **Thiết kế Register File đa cổng:** Để giải quyết xung đột khi lệnh nạp (WB) và lệnh đọc (ID) cùng truy cập tập thanh ghi trong một chu kỳ, module *Register File* được thiết kế với các cổng Đọc và Ghi độc lập. Theo chuẩn MIPS, thao tác ghi được thực hiện ở nửa đầu chu kỳ và thao tác đọc ở nửa sau chu kỳ (hoặc ngược lại tùy thiết kế xung nhịp), đảm bảo dữ liệu luôn nhất quán.

**Kết luận** Do xung đột cấu trúc đã được loại bỏ nhờ việc tuân thủ thiết kế phần cứng chuẩn của MIPS, báo cáo này không cần xây dựng logic phát hiện lỗi cho nhóm xung đột này. Trọng tâm của việc xử lý xung đột sẽ nằm ở hai loại phụ thuộc vào chương trình là xung đột dữ liệu và xung đột điều khiển.

#### 1.3.3.2 Xung đột dữ liệu (Data Hazards)

**Bản chất vấn đề** Xung đột dữ liệu xảy ra khi một lệnh phụ thuộc vào kết quả của một lệnh trước đó nhưng kết quả này chưa kịp ghi vào thanh ghi. Phổ biến nhất là loại xung đột **RAW (Read-After-Write)**.

- **Ví dụ:** Lệnh SUB theo sau lệnh ADD cần đọc thanh ghi \$t1, nhưng lệnh ADD chỉ ghi kết quả vào \$t1 ở giai đoạn cuối (WB), trong khi lệnh SUB cần đọc ngay ở giai đoạn đầu (ID).

**Giải pháp 1: Kỹ thuật Chuyển tiếp (Forwarding)** Đây là giải pháp tối ưu phần cứng cho phép dữ liệu từ các tầng sau (EX/MEM hoặc MEM/WB) được đưa ngược về đầu vào của ALU cho lệnh đang chờ, bỏ qua việc phải chờ ghi vào Register File.

- **Cơ chế:** Một khối điều khiển Forwarding Unit sẽ so sánh địa chỉ thanh ghi nguồn và đích. Nếu phát hiện trùng khớp, nó điều khiển bộ dồn kênh (MUX) để chọn dữ liệu "nóng" từ đường ống thay vì từ tập thanh ghi.

**Giải pháp 2: Kỹ thuật Dừng đường ống (Stalling)** Đối với trường hợp đặc biệt như **Load-Use Hazard** (lệnh sử dụng dữ liệu ngay sau lệnh nạp bộ nhớ LW), dữ liệu chưa có sẵn ngay cả khi dùng Forwarding.

- **Cơ chế:** Hệ thống buộc phải tạm dừng (Stall) lệnh phụ thuộc lại 1 chu kỳ bằng cách giữ nguyên giá trị PC và chèn một lệnh rỗng (**NOP - No Operation**) vào

đường ống, tạo ra một "bong bóng" (bubble) để dữ liệu kịp thời được nạp từ bộ nhớ.

### 1.3.3.3 Xung đột điều khiển (Control Hazards)

**Bản chất vấn đề** Xung đột điều khiển (hay rẽ nhánh) xảy ra khi hệ thống gặp các lệnh làm thay đổi dòng thực thi chương trình (như BEQ, J). Do địa chỉ lệnh tiếp theo chưa được xác định ngay lập tức, pipeline có thể nạp nhầm các lệnh không mong muốn.

**Chiến lược xử lý: Dự đoán không nhảy (Predict-Not-Taken)** Để giảm thiểu độ trễ, hệ thống sử dụng chiến lược dự đoán tĩnh: luôn giả định rẽ nhánh không xảy ra và tiếp tục nạp lệnh tuần tự ( $PC + 4$ ).

- **Nếu dự đoán đúng:** Pipeline tiếp tục hoạt động bình thường, không mất phí tổn.
- **Nếu dự đoán sai (Taken):** Hệ thống phải loại bỏ các lệnh đã lỡ nạp sai bằng cơ chế **xả đường ống (Flushing)** - biến các lệnh sai thành NOP và cập nhật lại PC về địa chỉ đích đúng.

**Tối ưu hóa: Rẽ nhánh sớm** Để giảm thiểu số lượng lệnh bị hủy khi dự đoán sai, kiến trúc MIPS cải tiến chuyển việc so sánh điều kiện và tính địa chỉ đích từ tầng EX về sớm ngay tại tầng ID. Kỹ thuật này giúp giảm độ trễ rẽ nhánh xuống chỉ còn 1 chu kỳ xung nhịp.

## CHƯƠNG 2. PHƯƠNG PHÁP LUẬN

### 2.1 Công cụ và môi trường phát triển

Để đảm bảo quá trình thiết kế, mô phỏng và kiểm thử bộ vi xử lý MIPS 32-bit diễn ra chính xác và hiệu quả, nhóm thực hiện đã sử dụng bộ công cụ phần mềm tiêu chuẩn trong thiết kế vi mạch số.

#### 2.1.1 Ngôn ngữ thiết kế

- **Verilog HDL:** Toàn bộ mã nguồn của dự án được viết bằng ngôn ngữ Verilog. Đây là ngôn ngữ mô tả phần cứng tiêu chuẩn công nghiệp, cho phép mô tả hành vi của các mạch số ở mức độ thanh ghi và mức cổng. Việc sử dụng Verilog giúp nhóm dễ dàng định nghĩa cấu trúc phần cứng, các khối chức năng và các kết nối tín hiệu trong CPU.

#### 2.1.2 Môi trường mô phỏng và kiểm thử

- **ModelSim SE-64 2020.4:** Đây là công cụ chủ lực được sử dụng để biên dịch và mô phỏng hệ thống. ModelSim cung cấp môi trường mạnh mẽ để:
  - Kiểm tra tính đúng đắn của logic thiết kế thông qua biểu đồ dạng sóng.
  - Gỡ lỗi chi tiết từng tín hiệu trong các chu kỳ xung nhịp.
  - Quan sát sự thay đổi giá trị trong các thanh ghi và bộ nhớ trong quá trình thực thi lệnh.

#### 2.1.3 Công cụ hỗ trợ

- **Visual Studio Code (VS Code):** Được sử dụng làm trình soạn thảo mã nguồn chính. VS Code giúp việc viết code, định dạng cú pháp và quản lý cấu trúc dự án trở nên trực quan và thuận tiện hơn.

#### 2.1.4 Phạm vi thiết kế

Do giới hạn về thời gian và thiết bị, đồ án tập trung hoàn toàn vào việc **mô phỏng chức năng** trên máy tính (PC). Các thiết kế được kiểm chứng thông qua Testbench và Waveform trên ModelSim, chưa thực hiện tổng hợp và nạp lên các kít phần cứng thực tế (như FPGA).

### 2.2 Cấu trúc mã nguồn

Để đảm bảo khả năng quản lý, mở rộng và gỡ lỗi hiệu quả cho một hệ thống phức tạp như CPU, mã nguồn của dự án được tổ chức theo mô hình thiết kế phân

cấp. Toàn bộ hệ thống được chia nhỏ thành các module chức năng độc lập, tương ứng với các tầng của đường ống và các khối xử lý phụ trợ.

### 2.2.1 Tổ chức thư mục và Module cấp cao

Cấu trúc dự án được chia thành ba thành phần cốt lõi: Module chính, các module con và Testbench.

MUX_REG_DST.v	✓ Verilog 11	12/28/2025 08:59:54 ...
MUX_WB.v	✓ Verilog 12	12/28/2025 08:59:54 ...
PC.v	✓ Verilog 13	12/28/2025 08:59:54 ...
EX_MEM.v	✓ Verilog 4	12/30/2025 07:40:56 ...
MUX_HAZARD_CONTROL.v	✓ Verilog 10	12/29/2025 04:35:28 ...
ADDR_CALCULATE.v	✓ Verilog 0	12/28/2025 08:59:54 ...
<b>ALU_BIG_MODULE.v</b>	<b>✓ Verilog 1</b>	<b>12/28/2025 08:59:54 ...</b>
CONTROL_UNIT.v	✓ Verilog 2	12/28/2025 08:59:54 ...
HAZARD_DETECTION.v	✓ Verilog 6	12/30/2025 07:40:56 ...
ID_EX.v	✓ Verilog 7	12/30/2025 07:40:56 ...
DATA_MEMORY.v	✓ Verilog 3	12/28/2025 08:59:54 ...
IF_ID.v	✓ Verilog 8	12/30/2025 07:40:56 ...
REGISTER_FILE.v	✓ Verilog 14	12/28/2025 08:59:54 ...
FORWARD_UNIT.v	✓ Verilog 5	12/30/2025 07:40:56 ...
tb_top_module.v	✓ Verilog 15	12/30/2025 07:40:56 ...
MEM_WB.v	✓ Verilog 9	12/30/2025 07:40:56 ...
TOP_MODULE.v	✓ Verilog 16	12/30/2025 07:40:56 ...

**Hình 2.1:** Cấu trúc cây thư mục dự án trong ModelSim

#### 2.2.1.1 Module chính

File `TOP_MODULE.v` đóng vai trò là xương sống kết nối toàn bộ hệ thống. Thiết kế này thực hiện **khởi tạo trực tiếp** toàn bộ các thành phần phần cứng tại đây.

Cụ thể, module này quản lý và kết nối ba nhóm thành phần chính:

- **Hệ thống thanh ghi đường ống:** Khởi tạo 4 thanh ghi phân tách các tầng: `IF_ID_REGISTER`, `ID_EX_REGISTER`, `EX_MEM_REGISTER` và `MEM_WB_REGISTER`.
- **Các khối chức năng Datapath:** Bao gồm các module xử lý và lưu trữ như `PC`, `INSTRUCTION_MEMORY`, `BIG_REGISTER`, `ALU_BIG_MODULE` và `DATA_MEMORY`.
- **Hệ thống Điều khiển và Xử lý xung đột:** Bao gồm `CONTROL_UNIT` (giải mã tín hiệu), `HAZARD_DETECTION_UNIT` (phát hiện xung đột), cùng các khối xử lý chuyên biệt như `FORWARDING_UNIT` và `FLUSHCONTROL`.

Toàn bộ các module trên được liên kết với nhau thông qua mạng lưới dây dẫn nội bộ được định nghĩa ngay trong `TOP_MODULE`, tạo thành luồng dữ liệu khép kín từ tầng IF đến WB.

## 2.2.2 Thiết kế phân tầng

Tuân thủ kiến trúc đường ống 5 tầng, mã nguồn được chia tách và đóng gói thành các file .v riêng biệt cho từng giai đoạn xử lý. Cách tổ chức này giúp quản lý code gọn gàng, cô lập lỗi và dễ dàng kiểm soát luồng dữ liệu giữa các tầng:

- **Tầng IF (Instruction Fetch):** File PC.v đóng vai trò là module chứa toàn bộ tầng nạp lệnh, bao gồm: bộ đếm chương trình (PC), bộ dồn kênh chọn địa chỉ (MUX\_PC), bộ nhớ lệnh và bộ cộng địa chỉ (PC\_Add\_4).
- **Tầng ID (Instruction Decode):** Các thành phần giải mã được chia thành:
  - REGISTER\_FILE.v: Chứa tập thanh ghi 32-bit và tích hợp sẵn logic mở rộng dấu.
  - CONTROL\_UNIT.v: Module giải mã tín hiệu điều khiển chính.
  - HAZARD\_DETECTION.v: Module phát hiện xung đột dữ liệu Load-Use.
- **Tầng EX (Execute):** File ALU\_BIG\_MODULE.v là khối thực thi trung tâm, bao gồm ALU, bộ điều khiển ALU (ALU Control) và các MUX chọn toán hạng. Bên cạnh đó, file FORWARD\_UNIT.v chịu trách nhiệm điều khiển logic chuyển tiếp dữ liệu (Forwarding) để giải quyết xung đột.
- **Tầng MEM (Memory):** File DATA\_MEMORY.v mô tả bộ nhớ dữ liệu (RAM), hỗ trợ đọc/ghi đồng bộ.
- **Tầng WB (Write Back):** File MUX\_WB.v chứa bộ dồn kênh chọn nguồn dữ liệu (từ ALU hoặc Bộ nhớ) để ghi ngược về thanh ghi.
- **Hệ thống Thanh ghi đường ống:** Để đồng bộ hóa dữ liệu giữa các tầng, hệ thống sử dụng 4 file thanh ghi riêng biệt: IF\_ID.v, ID\_EX.v, EX\_MEM.v và MEM\_WB.v.

## 2.2.3 Môi trường kiểm thử (Testbench)

Để kiểm chứng hoạt động của toàn bộ CPU, nhóm sử dụng file tb\_top\_module.v. Đây là module không có cổng vào/ra thực tế, đóng vai trò tạo môi trường mô phỏng:

- Tạo xung nhịp hệ thống với chu kỳ 10ns.
- Tạo tín hiệu Reset để đưa CPU về trạng thái đầu.
- Nạp chương trình mã máy vào Instruction Memory thông qua task write\_imem để CPU thực thi và quan sát dạng sóng đầu ra.

## 2.3 Thiết kế kiến trúc hệ thống

Phần này trình bày các đặc tả kỹ thuật chi tiết của bộ vi xử lý MIPS 32-bit được thiết kế trong đồ án. Hệ thống được xây dựng dựa trên một tập con của kiến trúc tập lệnh MIPS, được tối ưu hóa cho mô hình đường ống 5 tầng.

### 2.3.1 Đặc tả tập lệnh hỗ trợ

Thay vì hiện thực hóa toàn bộ tập lệnh MIPS khổng lồ, đồ án tập trung vào một tập con các lệnh cốt lõi đại diện cho các nhóm chức năng quan trọng nhất: tính toán số học, logic, truy cập bộ nhớ và điều khiển dòng chảy. Điều này đảm bảo CPU đủ khả năng thực thi các thuật toán cơ bản mà vẫn giữ cho thiết kế phần cứng nằm trong phạm vi quản lý của đồ án.

Hệ thống hỗ trợ tập lệnh gồm **13 lệnh cơ bản**, bao gồm:

**Bảng 2.1:** Danh sách chi tiết các lệnh MIPS được hiện thực hóa

Định dạng	Lệnh	Opcode / Funct	Chức năng mô tả
<b>R-Type</b>	ADD	0x00 / 0x20	Cộng thanh ghi có dấu
	SUB	0x00 / 0x22	Trừ thanh ghi có dấu
	AND	0x00 / 0x24	Phép VÀ logic
	OR	0x00 / 0x25	Phép HOẶC logic
	SLT	0x00 / 0x2A	So sánh nhỏ hơn
<b>I-Type</b>	ADDI	0x08	Cộng hằng số tức thời
	ANDI	0x0C	Phép VÀ với hằng số
	ORI	0x0D	Phép HOẶC với hằng số
	XORI	0x0E	Phép XOR với hằng số
	SLTI	0x0A	So sánh nhỏ hơn với hằng số
	LW	0x23	Nạp từ (Load Word)
	SW	0x2B	Lưu từ (Store Word)
	BEQ	0x04	Rẽ nhánh nếu bằng
<b>J-Type</b>	J	0x02	Nhảy không điều kiện

### 2.3.2 Yêu cầu về tài nguyên và hiệu năng hệ thống

Để đảm bảo bộ vi xử lý hoạt động ổn định và xử lý triệt để các xung đột, hệ thống được thiết kế với các thông số kỹ thuật và ràng buộc tài nguyên như sau:

#### 2.3.2.1 Thông số kiến trúc

- Độ rộng dữ liệu:** 32-bit. Toàn bộ các thanh ghi, bộ ALU và bus dữ liệu đều hoạt động trên độ rộng 32-bit.

- **Không gian địa chỉ:** 32-bit, cho phép quản lý lý thuyết lên đến  $2^{32}$  bytes (4GB). Tuy nhiên, trong mô phỏng, dung lượng bộ nhớ được giới hạn ở mức 256 Word (cho Instruction Memory) và 256 Word (cho Data Memory) để tối ưu hóa tài nguyên mô phỏng.
- **Tập thanh ghi:** 32 thanh ghi đa dụng ( $32 \times 32$ -bit). Hỗ trợ cơ chế "Dual Read, Single Write" (2 cổng đọc, 1 cổng ghi).

### 2.3.2.2 Cấu hình bộ nhớ

Tuân thủ kiến trúc Harvard, hệ thống khởi tạo hai module bộ nhớ tách biệt:

- **Instruction Memory (IM):** Bộ nhớ chỉ đọc trong quá trình thực thi. Mã máy được nạp trực tiếp từ Testbench (qua task `write_imem`) tại thời điểm khởi tạo.
- **Data Memory (DM):** Bộ nhớ Đọc/Ghi (Read/Write), hỗ trợ truy xuất theo từ thông qua tín hiệu `mem_read` và `mem_write`.

### 2.3.2.3 Chiến lược tối ưu hiệu năng

Để tối đa hóa thông lượng và giảm thiểu số chu kỳ lãng phí, kiến trúc hệ thống áp dụng các chiến lược sau:

- **Xử lý Data Hazard:** Ưu tiên sử dụng kỹ thuật **Full Forwarding** để loại bỏ hoàn toàn việc dừng đường ống đối với các lệnh R-Type liên tiếp. Kỹ thuật Stall chỉ được sử dụng trong trường hợp bất khả kháng là Load-Use Hazard.
- **Xử lý Control Hazard:** Sử dụng chiến lược **Early Branching** kết hợp với dự đoán tĩnh “predict not taken”. Điều này giúp giảm chi phí phạt từ 3 chu kỳ xuống còn 1 chu kỳ khi dự đoán sai.

### 2.3.3 Giao diện vào/ra

Module chính của hệ thống (TOP\_MODULE) được đóng gói với giao diện tín hiệu tối giản:

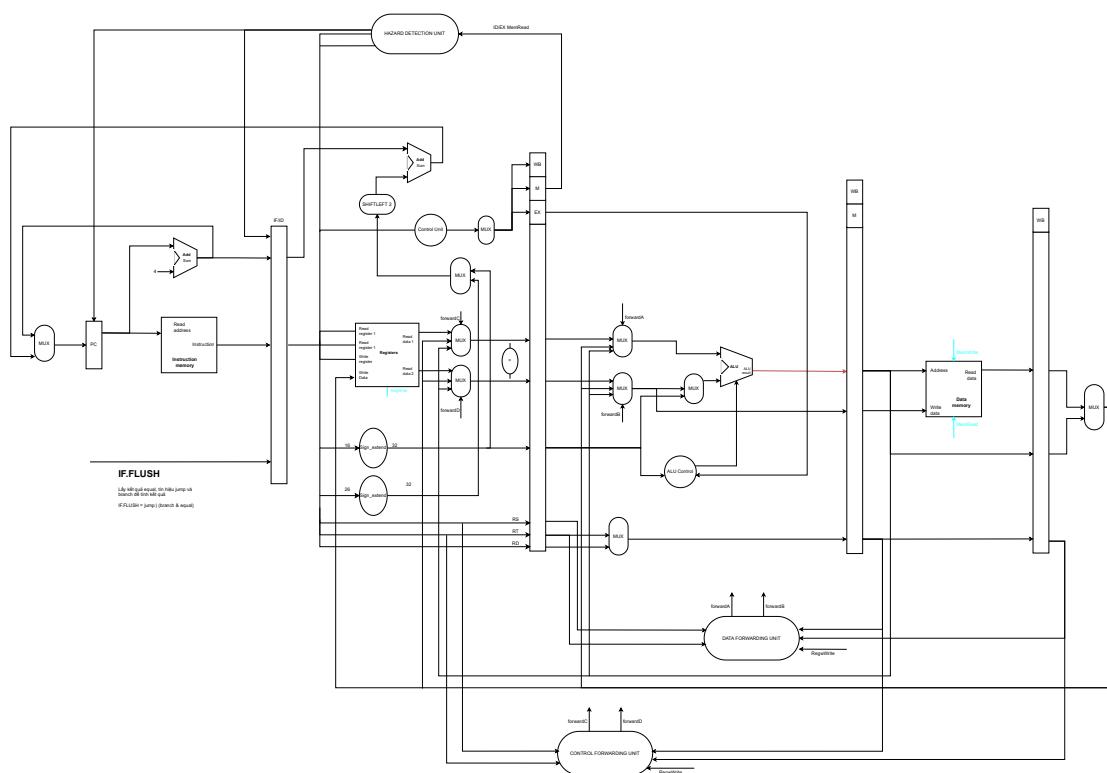
- **Đầu vào:**
  - `clk`: Tín hiệu xung nhịp hệ thống (kích hoạt cạnh lên).
  - `reset`: Tín hiệu khởi động lại đồng bộ, đưa PC về 0 và xóa sạch các thanh ghi đường ống.
- **Đầu ra:** Module không thiết kế các cổng đầu ra vật lý. Việc kiểm thử được thực hiện thông qua Testbench bằng cách truy cập trực tiếp các tín hiệu nội bộ.

## 2.4 Thiết kế Datapath Pipeline

Dựa trên kiến trúc tổng thể đã đề xuất, phần này đi sâu vào chi tiết thiết kế phần cứng của đường dữ liệu (Datapath). Sơ đồ dưới đây thể hiện sự kết nối cụ thể giữa các module chức năng, các bộ dồn kênh (MUX) và hệ thống đường dây tín hiệu trong mô hình 5 tầng pipeline của nhóm.

### 2.4.1 Sơ đồ kết nối tổng quát

Hình 2.2 mô tả Datapath hoàn chỉnh được tích hợp đầy đủ các khối chức năng, các bộ dồn kênh (MUX) và hệ thống đường dây tín hiệu:



**Hình 2.2:** Sơ đồ datapath Pipeline 5 tầng tích hợp xử lý Hazard

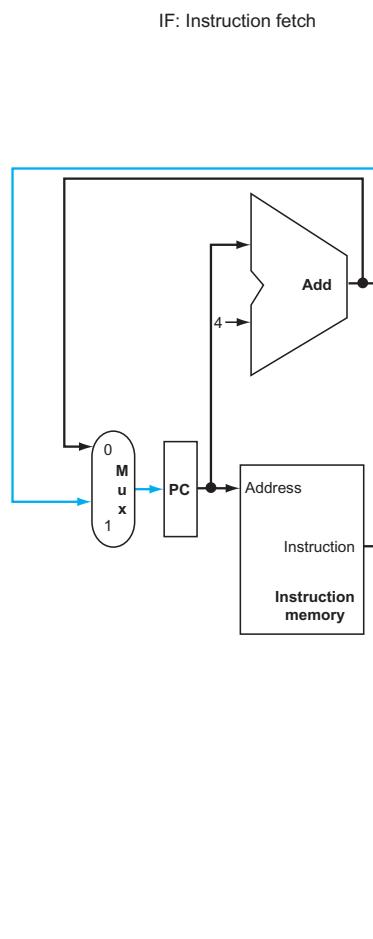
Để hệ thống hoạt động chính xác, bên cạnh luồng dữ liệu xuôi chiều từ IF đến WB, thiết kế bao gồm hai luồng dữ liệu hồi tiếp quan trọng:

- Cập nhật Register File:** Kết quả từ tầng cuối (WB) được vòng ngược về tầng 2 (ID) để ghi vào tập thanh ghi.
- Cập nhật PC:** Địa chỉ nhánh hoặc nhảy được tính toán tại tầng ID và đưa ngược về tầng 1 (IF) để điều khiển việc nạp lệnh tiếp theo.

## 2.4.2 Chi tiết thiết kế phần cứng từng tầng

### 2.4.2.1 Tầng 1: Nạp lệnh (Instruction Fetch - IF)

Mục đích của tầng này là truy xuất lệnh từ bộ nhớ và chuẩn bị địa chỉ cho chu kỳ kế tiếp.



Hình 2.3: Sơ đồ datapath tầng IF

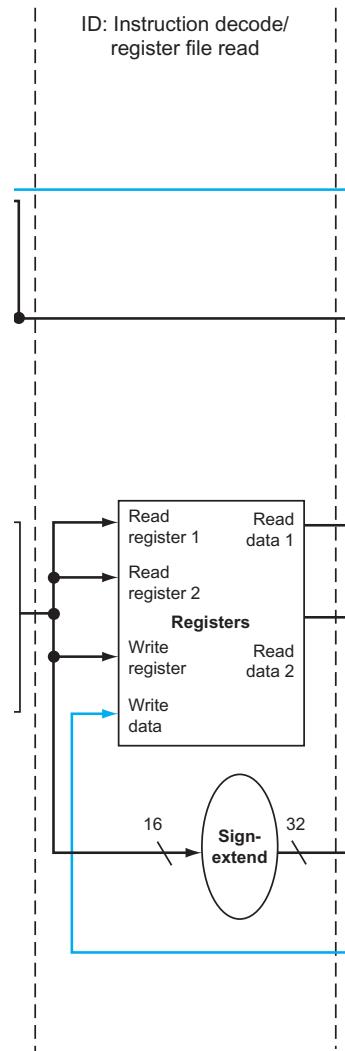
Thiết kế phần cứng của tầng này bao gồm ba thành phần chính được kết nối đồng bộ theo sườn dương xung nhịp:

- **Thanh ghi PC:** Lưu trữ địa chỉ lệnh hiện tại (32-bit). Đầu vào của PC được điều khiển bởi bộ đòn khen MUX\_PC để lựa chọn giữa địa chỉ tuần tự ( $PC + 4$ ) hoặc địa chỉ rẽ nhánh/nhảy (được tính toán từ tầng ID).
- **Instruction Memory:** Nhận địa chỉ từ PC và xuất ra mã lệnh 32-bit.
- **Bộ cộng (Adder +4):** Thực hiện phép tính  $PC_{next} = PC + 4$  để chuẩn bị cho chu kỳ tiếp theo.

Kết quả của tầng này (Lệnh và  $PC + 4$ ) được đẩy vào thanh ghi đường ống IF / ID.

#### 2.4.2.2 Tầng 2: Giải mã & Đọc thanh ghi (Instruction Decode - ID)

Đây là tầng trung tâm xử lý việc giải mã và điều khiển luồng. Thiết kế của nhóm tối ưu hóa việc xử lý rẽ nhánh tại đây.



**Hình 2.4:** Sơ đồ datapath tầng ID

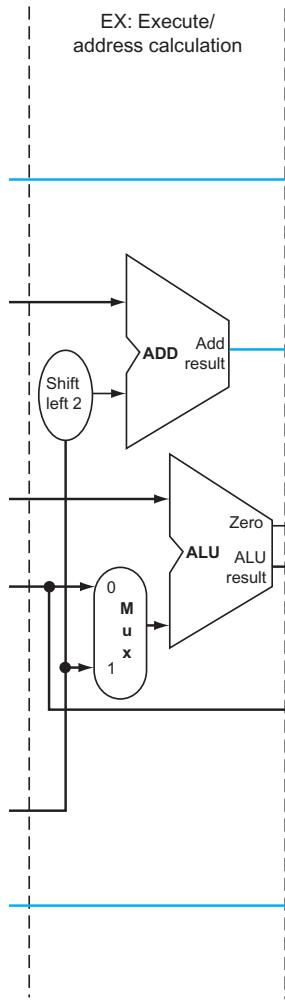
Các kết nối phần cứng bao gồm:

- **Register File:** Nhận địa chỉ đọc từ trường rs [25:21] và rt [20:16] của lệnh. Hai luồng dữ liệu đọc ra (ReadData1, ReadData2) được đưa đến bộ so sánh và thanh ghi ID/EX.
- **Bộ mở rộng dấu:** Chuyển đổi hằng số 16-bit [15:0] thành 32-bit để phục vụ tính toán địa chỉ và số học.
- **Logic so sánh:** Nhận dữ liệu từ Register File (hoặc từ Forwarding Unit nếu có xung đột) để so sánh điều kiện rẽ nhánh (ví dụ:  $Rs == Rt$ ). Kết quả so sánh điều khiển trực tiếp tín hiệu chọn PC.

- **Control Unit:** Giải mã Opcode [31:26] và phát sinh toàn bộ vector tín hiệu điều khiển cho các tầng sau.

#### 2.4.2.3 Tầng 3: Thực thi (Execute - EX)

Tầng này chịu trách nhiệm thực hiện các phép toán số học, logic và tính toán địa chỉ bộ nhớ.



Hình 2.5: Sơ đồ datapath tầng EX

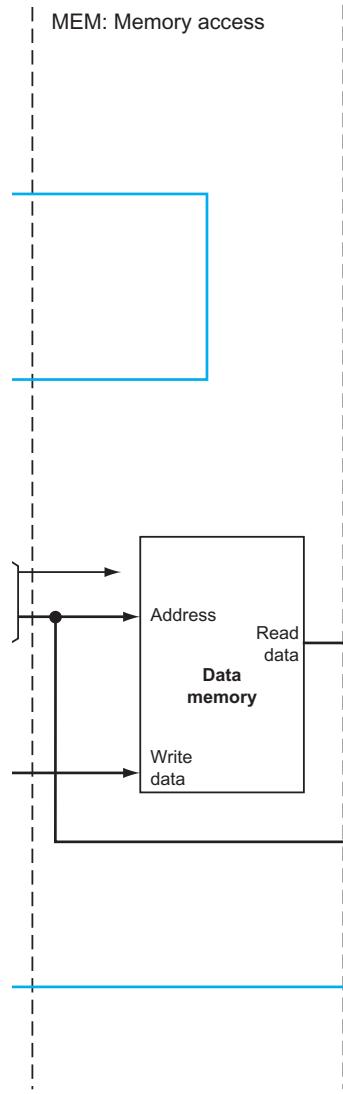
Trọng tâm của tầng này là bộ ALU và các khối xử lý dữ liệu:

- **Forwarding MUX:** Hai bộ MUX 3-to-1 được đặt trước ALU để lựa chọn dữ liệu nguồn: từ thanh ghi ID/EX, từ thanh ghi ID/EX (mặc định), từ kết quả tầng EX/MEM (khi ForwardA/B = 10), hoặc từ kết quả tầng MEM/WB (khi ForwardA/B = 01). Đây là thiết kế phần cứng cho kỹ thuật **Full Forwarding**.
- **ALU:** Thực hiện phép toán dựa trên tín hiệu ALU\_Control. Kết quả tính toán được dùng làm địa chỉ bộ nhớ (cho lệnh Load/Store) hoặc kết quả tính toán (cho lệnh R-Type).

- **ALUSrc:** Lựa chọn giữa thanh ghi  $Rt$  hoặc hằng số tức thời làm đầu vào thứ hai cho ALU.

#### 2.4.2.4 Tầng 4: Truy cập bộ nhớ (Memory - MEM)

Tầng này giao tiếp trực tiếp với bộ nhớ dữ liệu (RAM).



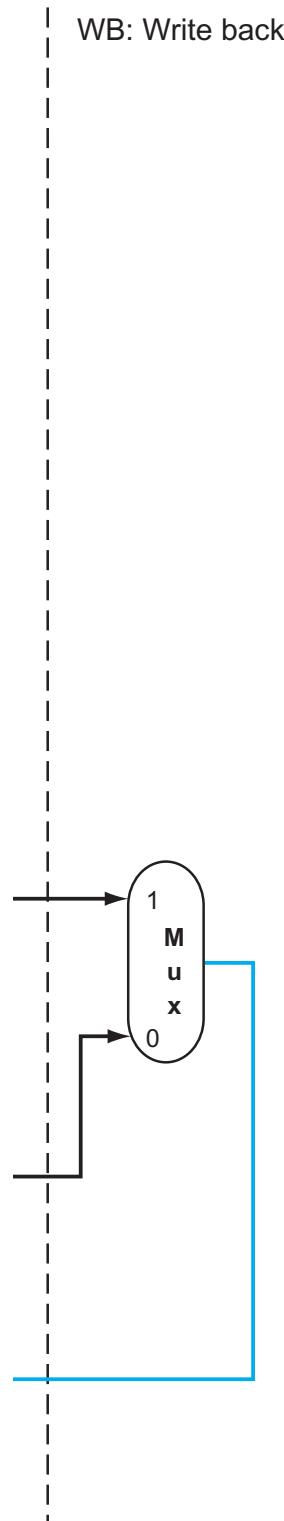
**Hình 2.6:** Sơ đồ datapath tầng MEM

Phần cứng tầng này kết nối trực tiếp với module Data\_Memory:

- **Địa chỉ:** Lấy từ ALU\_Result của tầng EX.
- **Dữ liệu ghi:** Lấy từ thanh ghi  $Rt$  (đã được đẩy qua các tầng pipeline).
- **Tín hiệu điều khiển:** MemRead và MemWrite kích hoạt việc đọc/ghi RAM.

## 2.4.2.5 Tầng 5: Ghi ngược (Write Back - WB)

Tầng cuối cùng hoàn tất chu trình lệnh bằng việc cập nhật kết quả vào Register File.



**Hình 2.7:** Sơ đồ datapath tầng WB

Tầng cuối cùng sử dụng một bộ MUX 2-to-1 (MemtoReg) để định tuyến dữ

liệu về Register File:

- Nếu tín hiệu MemtoReg = 1: Chọn dữ liệu từ bộ nhớ.
- Nếu tín hiệu MemtoReg = 0: Chọn kết quả từ ALU.

Dữ liệu được chọn sẽ được ghi vào thanh ghi đích (*Rd* hoặc *Rt*) tại cổng Write-Data của Register File khi tín hiệu RegWrite được bật.

### 2.4.3 Hệ thống thanh ghi đường ống

Để đảm bảo tính độc lập giữa các tầng, hệ thống sử dụng 4 thanh ghi đường ống đóng vai trò như các "vách ngăn" lưu trữ trạng thái giữa các chu kỳ xung nhịp.

- **IF/ID Register (64-bit):** Lưu trữ địa chỉ *PC* + 4 và mã lệnh 32-bit.
- **ID/EX Register:** Lưu trữ dữ liệu đọc từ thanh ghi (*Rs*, *Rt*), hằng số mở rộng và các nhóm tín hiệu điều khiển cho tầng EX, MEM, WB.
- **EX/MEM Register:** Lưu trữ kết quả tính toán của ALU, dữ liệu cần ghi vào bộ nhớ và nhóm tín hiệu điều khiển cho tầng MEM, WB.
- **MEM/WB Register:** Lưu trữ dữ liệu đọc từ bộ nhớ, kết quả ALU và nhóm tín hiệu điều khiển cho tầng WB.

## 2.5 Thiết kế khối điều khiển (Control Unit)

Khối điều khiển đóng vai trò trung tâm trong việc điều phối hoạt động của toàn bộ hệ thống. Nhiệm vụ của nó là giải mã các bit từ thanh ghi lệnh và sinh ra các tín hiệu kích hoạt chính xác cho Datapath.

Trong thiết kế này, logic điều khiển được phân chia thành hai cấp bậc và áp dụng chiến lược lan truyền tín hiệu đặc thù của kiến trúc đường ống.

### 2.5.1 Bộ điều khiển chính (Main Control)

Module này (file CONTROL\_UNIT.v) nhận đầu vào là 6 bit Opcode (bit [31:26]). Logic giải mã được thực hiện bằng cấu trúc case để định nghĩa tín hiệu ra.

Khác với kiến trúc MIPS chuẩn (xử lý rẽ nhánh tại tầng MEM), báo cáo này tối ưu hóa bằng cách phân chia tín hiệu thành 4 nhóm dựa trên thời điểm kích hoạt:

- **Nhóm tín hiệu tầng ID:** Bao gồm Branch, Jump. Do áp dụng kỹ thuật **Early Branching**, các tín hiệu này được tiêu thụ ngay lập tức tại tầng ID để điều khiển bộ MUX\_PC và khối Flush, không cần truyền qua các thanh ghi đường ống phía sau.

- **Nhóm tín hiệu tầng EX:** Bao gồm RegDst, ALUOp (3-bit), ALUSrc. Các tín hiệu này được đẩy vào thanh ghi ID/EX để sử dụng cho chu kỳ tính toán kế tiếp.
- **Nhóm tín hiệu tầng MEM:** Bao gồm MemRead, MemWrite. Các tín hiệu này đi qua hai tầng thanh ghi đệm (ID/EX → EX/MEM) để điều khiển việc đọc/ghi dữ liệu.
- **Nhóm tín hiệu tầng WB:** Bao gồm MemtoReg, RegWrite. Đây là nhóm tín hiệu đi quãng đường xa nhất (ID/EX → EX/MEM → MEM/WB) để điều khiển việc ghi ngược kết quả.

### 2.5.2 Bộ điều khiển ALU (ALU Control)

Module này kết hợp tín hiệu ALUOp (3-bit) và trường funct để sinh ra tín hiệu chọn phép toán cho ALU. Do thiết kế sử dụng **ALUOp 3-bit**, hệ thống có khả năng mã hóa linh hoạt hơn so với chuẩn 2-bit lý thuyết:

- ALUOp = 010: Yêu cầu ALU kiểm tra trường funct (dùng cho R-Type).
- ALUOp = 000: Yêu cầu ALU thực hiện phép cộng (dùng cho LW, SW, ADDI).
- ALUOp = 001, 011, . . . : Các phép toán logic bổ sung (nếu có).

### 2.5.3 Bảng tín hiệu điều khiển tổng quát

Dưới đây là bảng chân lý tóm tắt trạng thái các tín hiệu điều khiển.

**Lưu ý quan trọng:** Đối với lệnh BEQ, do hệ thống áp dụng kỹ thuật **Early Branching**, bộ ALU tại tầng EX không tham gia vào quyết định rẽ nhánh. Do đó, các tín hiệu liên quan đến ALU (ALUOp, ALUSrc) được đánh dấu là X (Don't Care).

**Bảng 2.2:** Bảng chân lý tín hiệu điều khiển

Lệnh	Opcode	RegDst	ALUSrc	MemtoReg	RegWr	MemRd	MemWr	Branch	ALUOp
R-Type	000000	1	0	0	1	0	0	0	010
lw	100011	0	1	1	1	1	0	0	000
sw	101011	X	1	X	0	0	1	0	000
beq	000100	X	X	X	0	0	0	1	XXX
addi	001000	0	1	0	1	0	0	0	000
j	000010	X	X	X	0	0	0	X	XXX

## 2.6 Giải pháp xử lý xung đột

Trong kiến trúc đường ống, việc đảm bảo tính toàn vẹn dữ liệu khi các lệnh thực thi song song là thách thức lớn nhất. Nhóm thiết kế đã tích hợp các khối phần cứng chuyên biệt để giải quyết triệt để vấn đề này.

## 2.6.1 Phân tích Xung đột dữ liệu (Data Hazards)

### 2.6.1.1 Định nghĩa và Ví dụ

Xung đột dữ liệu xảy ra khi một lệnh cần sử dụng kết quả của lệnh trước đó nhưng dữ liệu chưa kịp được ghi vào tập thanh ghi (lỗi RAW - Read After Write).

**Ví dụ minh họa:**

```
ADD $t1, $t2, $t3 # Ghi $t1 tại tầng WB (Chu kỳ 5)
SUB $t4, $t1, $t5 # Đọc $t1 tại tầng ID (Chu kỳ 2)
```

Trong ví dụ trên, lệnh SUB sẽ đọc được giá trị cũ của \$t1 nếu không có cơ chế xử lý, dẫn đến sai lệch kết quả tính toán.

### 2.6.1.2 Phân loại theo khoảng cách

Dựa trên khoảng cách giữa lệnh sinh dữ liệu (lệnh gây lỗi) và lệnh tiêu thụ (lệnh bị ảnh hưởng), nhóm phân loại xung đột dữ liệu thành hai trường hợp chính:

- **Loại 1 (Khoảng cách 1 lệnh – Xung đột EX):**

Xảy ra khi lệnh ngay liền trước ghi dữ liệu vào thanh ghi nguồn mà lệnh hiện tại cần sử dụng. Xung đột này được phát hiện tại tầng thực thi (EX).

- *Điều kiện:* Thanh ghi đích của lệnh trước (*EX/MEM.Rd*) trùng với thanh ghi nguồn của lệnh hiện tại (*ID/EX.Rs* hoặc *ID/EX.Rt*).
- *Biểu thức:*

$$EX/MEM.Rd = ID/EX.Rs \text{ hoặc } EX/MEM.Rd = ID/EX.Rt$$

- **Loại 2 (Khoảng cách 2 lệnh – Xung đột MEM):**

Xảy ra khi đứng trước lệnh hiện tại hai chu kỳ ghi dữ liệu vào thanh ghi nguồn của lệnh hiện tại. Xung đột này liên quan đến dữ liệu ở tầng truy cập bộ nhớ (MEM) và tầng giải mã (ID).

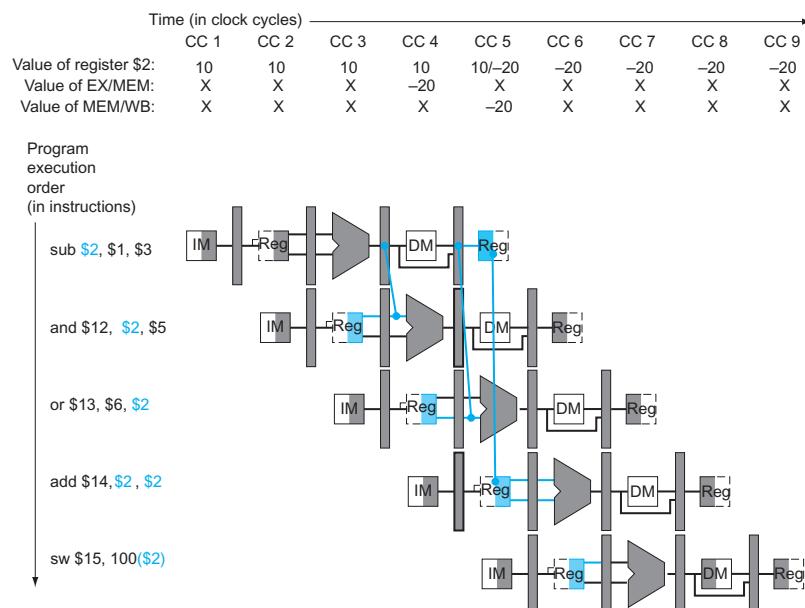
- *Điều kiện:* Thanh ghi đích của lệnh trước đó nữa (*MEM/WB.Rd*) trùng với thanh ghi nguồn của lệnh hiện tại.
- *Biểu thức:*

$$MEM/WB.Rd = ID/EX.Rs \text{ hoặc } MEM/WB.Rd = ID/EX.Rt$$

## 2.6.2 Giải pháp Chuyển tiếp dữ liệu (Forwarding Unit)

Để giải quyết các xung đột tính toán (R-Type) mà không làm giảm hiệu năng, nhóm sử dụng kỹ thuật Forwarding: lấy dữ liệu trực tiếp từ các tầng sau đưa ngược

về đầu vào ALU.

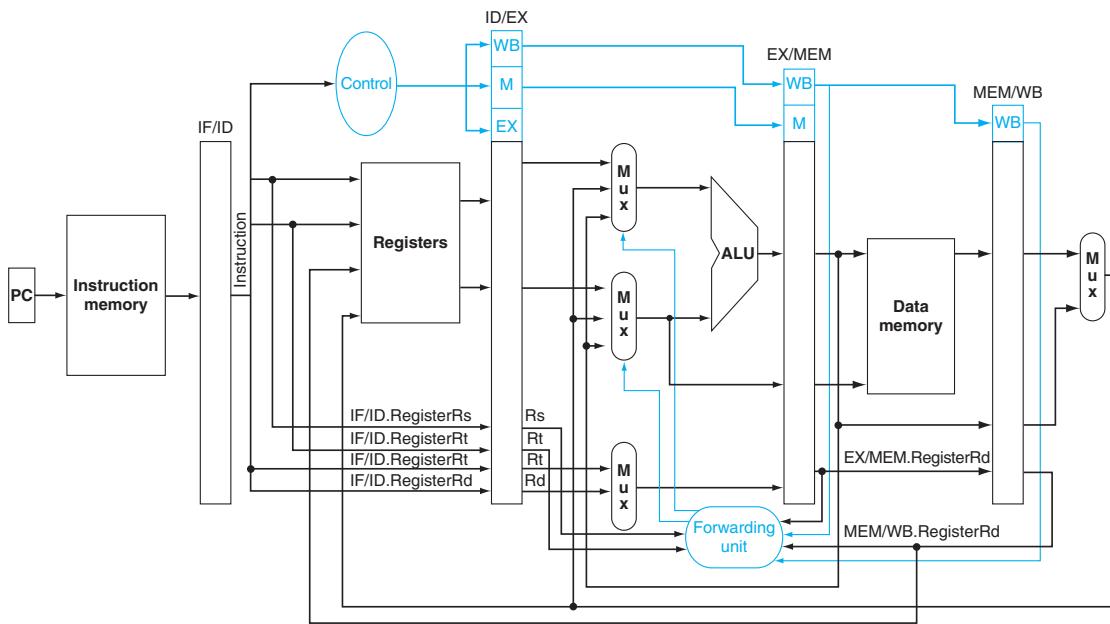


**Hình 2.8:** Sơ đồ nguyên lý cơ chế chuyển tiếp dữ liệu

Khối FORWARD\_UNIT. v hoạt động dựa trên nguyên tắc ưu tiên để chọn dữ liệu mới nhất:

- Ưu tiên 1 (Forwarding từ EX/MEM):** Nếu phát hiện trùng khớp thanh ghi đích của lệnh trước ( $Rd$ ) với nguồn của lệnh hiện tại ( $Rs/Rt$ ), dữ liệu được lấy ngay từ kết quả ALU tại tầng EX.
- Ưu tiên 2 (Forwarding từ MEM/WB):** Nếu không thỏa mãn ưu tiên 1 nhưng phát hiện trùng khớp với lệnh cách đó 1 chu kỳ, dữ liệu được lấy từ kết quả tại tầng WB.

**Điều kiện kích hoạt:** Logic chuyển tiếp chỉ hoạt động khi thanh ghi đích khác \$zero và tín hiệu cho phép ghi (RegWrite) đang được bật.



**Hình 2.9:** Sơ đồ Datapath tích hợp bộ Forwarding Unit

### 2.6.3 Giải pháp Dừng đường ống (Hazard Detection Unit)

Đối với trường hợp đặc biệt là **Load-Use Hazard** (lệnh sử dụng dữ liệu ngay sau lệnh nạp LW), dữ liệu chỉ có sẵn sau tầng MEM nên kỹ thuật Forwarding không thể đáp ứng kịp thời tại tầng EX.

**Logic phát hiện:** Module HAZARD\_DETECTION.v tại tầng ID kiểm tra điều kiện:

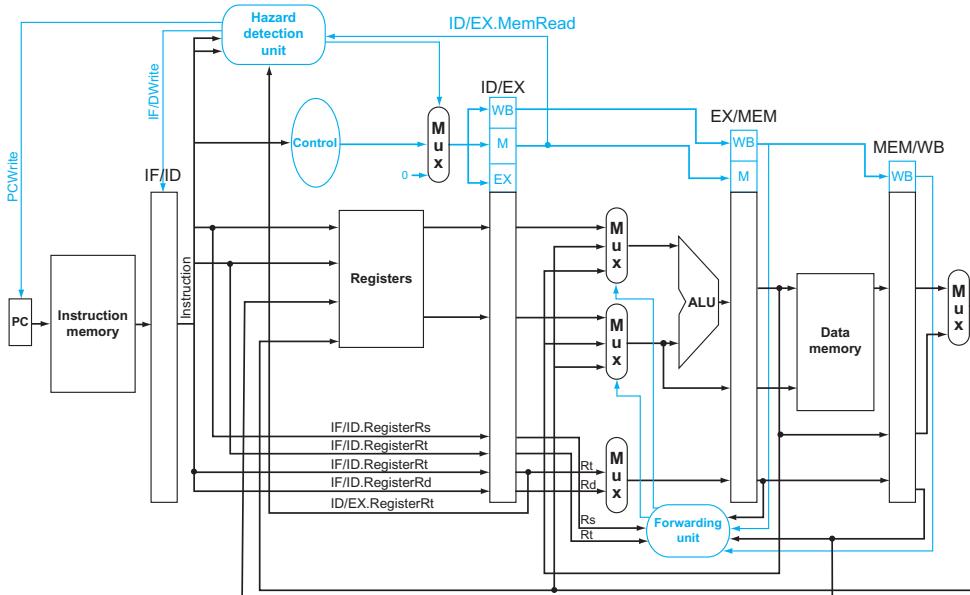
```
if (ID/EX.MemRead) and ((ID/EX.Rt == IF/ID.Rs) or (ID/EX.Rt == IF/ID.Rt))
```

**Hệ quả:** Định trệ (Stall) đường ống 01 chu kỳ xung nhịp.

#### 2.6.3.1 Cơ chế thực thi Stall và Bubble

Khi phát hiện xung đột Load-Use, hệ thống thực hiện đóng băng trạng thái:

- **PC\_Write = 0:** Khóa thanh ghi PC để không nạp lệnh mới.
- **IF\_ID\_Write = 0:** Giữ nguyên lệnh hiện tại ở thanh ghi IF/ID để giải mã lại vào chu kỳ sau.
- **Chèn bóng (Bubble):** Module MUX\_HAZARD\_CONTROL ép toàn bộ các tín hiệu điều khiển (RegWrite, MemWrite, ALUOp...) về 0. Điều này biến lệnh đang ở tầng EX thành lệnh rỗng (NOP), ngăn chặn các thao tác ghi sai.



**Hình 2.10:** Datapath hoàn chỉnh tích hợp cả Hazard Detection và Forwarding

### 2.6.3.2 Ví dụ diễn biến thực thi (Execution Timeline)

Dưới đây là bảng mô tả trạng thái hệ thống khi xử lý chuỗi lệnh: `LW $t0 ...` theo sau bởi `ADD $t1, $t0 ...`

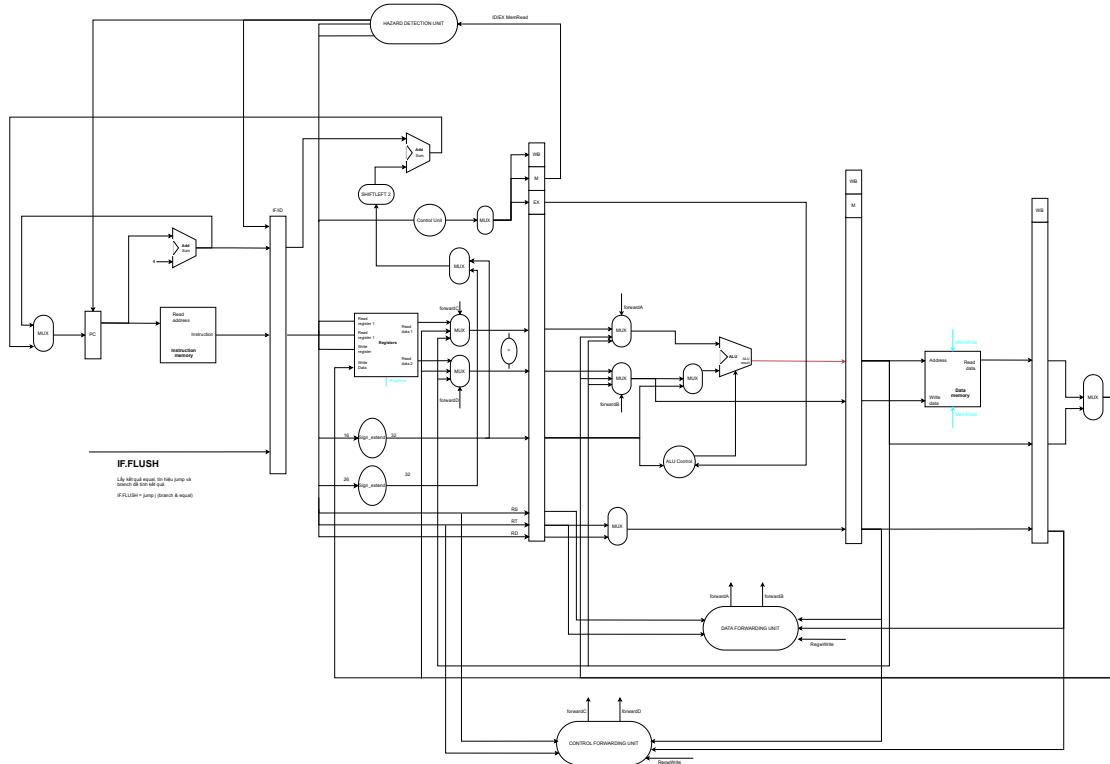
**Bảng 2.3:** Diễn biến xử lý Load-Use Hazard theo chu kỳ

Chu kỳ	Trạng thái hệ thống
N	Hazard Detection phát hiện trùng khớp: $ID/EX.Rt = IF/ID.Rs = \$t0$ . Tín hiệu Stall được kích hoạt.
N+1	1. PC và IF/ID bị khóa, giữ nguyên giá trị. 2. Tầng ID/EX nhận BUBBLE (mọi tín hiệu điều khiển = 0). 3. Lệnh LW tiến tới tầng MEM để lấy dữ liệu.
N+2	1. Stall kết thúc. Lệnh ADD được giải mã lại bình thường. 2. Dữ liệu từ tầng MEM/WB (của lệnh LW) được Forwarding về cho ALU để thực hiện phép cộng.

### 2.6.4 Xung đột điều khiển (Control Hazard)

Xung đột điều khiển xảy ra khi đường ống nạp các lệnh tiếp theo vào tầng Fetch trong khi chưa xác định được chính xác dòng thực thi của chương trình do gặp lệnh rẽ nhánh (Branch) hoặc lệnh nhảy (Jump).

Để giải quyết vấn đề này, nhóm thiết kế áp dụng chiến lược **Predict Not Taken** kết hợp với kỹ thuật **Early Branch** (Rẽ nhánh sớm) tại tầng ID. Sơ đồ đường dữ liệu tổng quát hỗ trợ cơ chế này được thể hiện trong Hình 2.11.



**Hình 2.11:** Sơ đồ Datapath tích hợp Forwarding, Hazard Detection và Early Branch

#### 2.6.4.1 Chiến lược Dự đoán: Predict Not Taken

Thay vì dừng đường ống (Stall) mỗi khi gặp lệnh rẽ nhánh – gây lãng phí tài nguyên xử lý nghiêm trọng, hệ thống sử dụng giả định rằng rẽ nhánh sẽ **không xảy ra**.

- **Cơ chế:** Bộ đếm chương trình luôn tự động tăng ( $PC + 4$ ) và nạp lệnh kế tiếp vào đường ống.
- **Trường hợp đúng (Not Taken):** Nếu điều kiện rẽ nhánh sai, luồng lệnh diễm ra tuân tự bình thường. Chi phí phạt là 0 chu kỳ.
- **Trường hợp sai (Taken):** Nếu điều kiện rẽ nhánh đúng, các lệnh đã nạp vào đường ống bị coi là lệnh sai và buộc phải bị loại bỏ.

#### 2.6.4.2 Tối ưu hóa: Rẽ nhánh sớm

Theo kiến trúc MIPS chuẩn, quyết định rẽ nhánh thường diễn ra ở tầng MEM hoặc EX, gây lãng phí 2-3 chu kỳ nếu dự đoán sai. Nhóm đã tối ưu hóa bằng cách chuyển logic so sánh và tính toán địa chỉ về tầng ID.

##### Cải tiến phần cứng:

- **Bộ so sánh:** Sử dụng cỗng XOR để kiểm tra điều kiện bằng nhau ('Equal') ngay sau khi đọc thanh ghi tại ID.

- **Tính toán địa chỉ:** Module TOP\_DECODE\_ADDR (bao gồm BRANCH\_ADDR và JUMP\_ADDR) thực hiện cộng địa chỉ đích ngay tại ID.

$$\begin{cases} \text{Branch\_Addr} = PC_{\text{next}} + (\text{SignExtend}(imm) \ll 2) \\ \text{Jump\_Addr} = \{PC_{\text{next}}[31 : 28], \text{instr}[25 : 0], 2'b00\} \end{cases}$$

- **Forwarding đặc biệt:** Bổ sung đường hồi tiếp dữ liệu từ các tầng sau về ID để đảm bảo bộ so sánh nhận được giá trị mới nhất.

**Kết quả:** Giảm chi phí phạt xuống chỉ còn **1 chu kỳ** xung nhịp trong trường hợp dự đoán sai.

#### 2.6.4.3 Cơ chế Xả đường ống (Flushing)

Khi phát hiện dự đoán sai, hệ thống kích hoạt tín hiệu Flush để hủy lệnh đang nằm trong tầng IF.

**Logic điều khiển:**

$$\text{Flush} = \text{Jump} \vee (\text{Branch} \wedge \text{Reg\_Equal})$$

Khi tín hiệu này được kích hoạt:

1. **Cập nhật PC:** PC được gán giá trị địa chỉ đích mới.
2. **Xóa lệnh sai:** Thanh ghi đường ống IF/ID bị xóa hoặc các tín hiệu điều khiển bị ép về 0. Lệnh sai trở thành lệnh rỗng (**NOP**) và không gây ảnh hưởng đến trạng thái CPU ở các tầng sau (EX, MEM, WB).

Bảng dưới đây minh họa sự khác biệt về trạng thái đường ống giữa hai trường hợp:

**Bảng 2.4:** So sánh hoạt động đường ống: Taken vs Not Taken

CK	Trường hợp 1: TAKEN (Dự đoán Sai)	Trường hợp 2: NOT TAKEN (Dự đoán Đúng)
1	IF: Fetch beq	IF: Fetch beq
2	ID: Decode beq (Phát hiện Taken) IF: Fetch Lệnh @ PC+4 → <b>Hành động: Flush IF/ID</b>	ID: Decode beq (Phát hiện Not Taken) IF: Fetch Lệnh @ PC+4 → <b>Hành động: Tiếp tục</b>
3	ID: NOP (Lệnh cũ bị hủy)	ID: Decode Lệnh @ PC+4
KL	<b>Mất 1 chu kỳ</b>	<b>Không mất chu kỳ nào</b>

#### 2.6.4.4 Ngoại lệ: Xung đột dữ liệu tại lệnh Rẽ nhánh

Mặc dù đã có Forwarding về ID, tồn tại một trường hợp đặc biệt cần phải dừng đường ống thay vì chỉ Flush. Đó là khi lệnh Branch phụ thuộc vào kết quả của lệnh ALU ngay liền trước nó (đang ở tầng EX).

Do việc tính toán rẽ nhánh diễn ra tại ID, dữ liệu từ lệnh ALU trước đó chưa kịp tính xong để Forward về.

- **Điều kiện phát hiện:**

$\text{Branch} \wedge \text{RegWrite}_{\text{pre}} \wedge (ID/EX.Rd = IF/ID.Rs \vee ID/EX.Rd = IF/ID.Rt)$

- **Hành động:** Hệ thống kích hoạt **Stall** (giữ nguyên PC và IF/ID) trong 1 chu kỳ, đồng thời chèn Bubble vào tầng EX.

```
// Logic Stall trong Hazard Unit
if (branch && reg_write &&
    ((ID_EX_rd == IF_ID_rs) || (ID_EX_rd == IF_ID_rt)))
begin
    pc_stall = 1'b1;           // Giữ PC
    if_id_stall = 1'b1;        // Giữ IF/ID
    mux_control_hazard = 1;   // Chèn NOP
end
```

#### 2.6.4.5 So sánh với các kỹ thuật nâng cao

Trong kiến trúc máy tính, bên cạnh chiến lược dự đoán tĩnh (Predict Not Taken) mà nhóm đang áp dụng, có hai kỹ thuật xử lý xung đột điều khiển phổ biến khác:

- **Dự đoán động (Dynamic Branch Prediction):** Kỹ thuật này sử dụng phần cứng chuyên biệt là *Bảng lịch sử rẽ nhánh* (*Branch History Table*) để lưu trạng thái "Taken" hoặc "Not Taken" trong quá khứ. Cơ chế dự đoán 2-bit thường được dùng để tăng độ ổn định, chỉ thay đổi dự đoán nếu sai hai lần liên tiếp.
- **Rẽ nhánh trễ (Delayed Branch):** Kỹ thuật này thay đổi hành vi của kiến trúc tập lệnh, quy định rằng lệnh nằm ngay sau lệnh rẽ nhánh sẽ luôn được thực thi bất kể kết quả rẽ nhánh. Điều này chuyển gánh nặng xử lý sang cho trình biên dịch, buộc phải sắp xếp lại code để chèn lệnh hữu ích hoặc lệnh NOP vào khe trễ này.

**Lý do lựa chọn thiết kế:** Nhóm quyết định không sử dụng *Delayed Branch* vì kỹ thuật này làm tăng đáng kể độ phức tạp của trình biên dịch và khó tương thích với code assembly viết tay. Đối với *Dynamic Prediction*, việc thêm bộ nhớ BHT sẽ làm tăng chi phí phần cứng không cần thiết cho một thiết kế đơn chu kỳ. Thay vào đó, sự kết hợp giữa **Predict Not Taken** và **Early Branch** (tại tầng ID) được lựa chọn để cân bằng tốt nhất giữa hiệu năng xử lý và tài nguyên phần cứng.

**Bảng 2.5:** Tổng kết chiến lược xử lý Control Hazard

<b>Đặc điểm</b>	<b>Trạng thái</b>	<b>Ghi chú kỹ thuật</b>
Chiến lược	Predict Not Taken	Tối ưu cho các đoạn mã tuần tự
Vị trí giải quyết	Tầng ID	Giảm Penalty xuống tối thiểu
Branch Penalty	1 chu kỳ	Khi dự đoán sai (Taken)
Xử lý đặc biệt	<b>Stall</b>	Khi gặp Data Hazard tại lệnh Branch

## 2.7 Hiện thực các module chính

Hệ thống được tổ chức thành các module phân cấp, trong đó các thành phần logic nhỏ được đóng gói vào các module lớn hơn theo từng tầng chức năng. Dưới đây là chi tiết hiện thực các khối quan trọng nhất bằng mã nguồn Verilog:

### 2.7.1 Module ALU (Khối thực thi)

Module `ALU_BIG_MODULE.v` đóng vai trò trung tâm trong tầng Execution. Thay vì chỉ chứa bộ ALU thuần túy, module này tích hợp sẵn:

- **ALU Core:** Thực hiện các phép toán số học (ADD, SUB), logic (AND, OR, XOR) và so sánh (SLT).
- **ALU Control:** Giải mã tín hiệu điều khiển ALUOp thành tín hiệu chọn phép toán cụ thể.

Việc đóng gói này giúp giảm số lượng dây nối ở module trên cùng (`TOP_MODULE`), làm cho thiết kế gọn gàng hơn.

```
// ALU Core: Khối thực hiện phép toán
always @(*) begin
    case (ALU_Sel)
        3'b000: ALU_Out = ALU_In_0 + ALU_In_1; // ADD
        3'b001: ALU_Out = ALU_In_0 - ALU_In_1; // SUB
        3'b010: ALU_Out = ALU_In_0 & ALU_In_1; // AND
        3'b011: ALU_Out = ALU_In_0 | ALU_In_1; // OR
        3'b100: ALU_Out = ALU_In_0 ^ ALU_In_1; // XOR
        3'b101: ALU_Out = (ALU_In_0 < ALU_In_1) ? 1 : 0; // SLT
        default: ALU_Out = 0;
    endcase
end
```

*Đoạn mã 2.1: Logic của khối ALU Core trong module ALU\_BIG\_MODULE*

```
// Forwarding MUX: Chọn nguồn dữ liệu đầu vào ALU
assign alu_in_a =
(ForwardA == 2'b10) ? EX_MEM_alu_result : // Từ tầng MEM
(ForwardA == 2'b01) ? MEM_WB_read_data : // Từ tầng WB
read_data_1; // Từ thanh ghi (mặc định)
```

*Đoạn mã 2.2: MUX Forwarding cho đầu vào A của ALU*

### 2.7.2 Module Register File & Decode Wrapper

Module REGISTER\_FILE.v trong thiết kế này không chỉ là bộ nhớ thanh ghi đơn thuần mà hoạt động như một bộ đóng gói cho toàn bộ tầng ID. Bên trong module này bao gồm:

- **Register Array:** Mảng thanh ghi  $32 \times 32$ -bit, hỗ trợ đọc 2 cổng và ghi 1 cổng.
- **Sign Extension:** Bộ mở rộng dấu cho hằng số tức thời.
- **Branch Comparator:** Logic so sánh (trong BIG\_REGISTER) để phục vụ cơ chế Early Branch ngay tại tầng ID.

```
// Ghi vào Register File tại cạnh lên clock
always @ (posedge clk) begin
if (reset) begin
for (i = 0; i < 32; i = i + 1) regs[i] <= 32'b0;
end
else if (reg_write_in && (write_addr != 5'd0)) begin
regs[write_addr] <= write_data; // Ghi dữ liệu
end
regs[0] <= 32'b0; // Thanh ghi $zero luôn = 0
end
```

*Đoạn mã 2.3: Logic ghi thanh ghi với bảo vệ thanh ghi \$zero*

```
// Forwarding cho Branch Comparator tại tầng ID
assign id_op_a = (forwardC == 2'b10) ? EX_MEM_value :
(forwardC == 2'b01) ? MEM_WB_value :
rf_rdl; // Mặc định từ RF
assign equal_after_forward = (id_op_a == id_op_b);
```

*Đoạn mã 2.4: Forwarding nội bộ và so sánh điều kiện rẽ nhánh tại ID*

### 2.7.3 Module Control Unit & Address Calculation

Khối điều khiển được chia thành hai phần nhiệm vụ riêng biệt:

- **Main Control (CONTROL\_UNIT.v):** Giải mã mã lệnh (Opcode) để sinh ra các tín hiệu điều khiển chính như RegDst, MemRead, MemWrite, ALUOp...
- **Address Calculation (ADDR\_CALCULATE.v):** Chuyên trách tính toán địa chỉ nhảy cho lệnh JUMP và địa chỉ rẽ nhánh cho lệnh BRANCH. Việc tách module này giúp logic tính toán địa chỉ độc lập với logic giải mã lệnh.

```
// Giải mã Opcode trong Control Unit
case (opcode)
  6'h00: begin // R-Type
    reg_dst = 1; reg_write = 1;
    alu_op = 3'b010; // Yêu cầu ALU kiểm tra funct
  end
  6'h23: begin // lw
    alu_src = 1; mem_to_reg = 1; reg_write = 1; mem_read = 1;
    alu_op = 3'b000; // ADD (tính địa chỉ)
  end
  6'h2B: begin // sw
    alu_src = 1; mem_write = 1;
    alu_op = 3'b000;
  end
  6'h04: begin // beq
    branch = 1; // Tín hiệu rẽ nhánh (xử lý tại ID)
  end
  6'h02: begin jump = 1; end // j
endcase
```

*Đoạn mã 2.5: Logic giải mã Opcode trong CONTROL\_UNIT*

#### 2.7.4 Module Hazard Handling (Xử lý xung đột)

Đây là thành phần phức tạp nhất đảm bảo tính đúng đắn của đường ống:

- **Hazard Detection (HAZARD\_DETECTION.v):** Phát hiện xung đột Load-Use và Control Hazard. Khi phát hiện lỗi, nó gửi tín hiệu đến bộ điều khiển MUX.
- **Stall Control (MUX\_HAZARD\_CONTROL.v):** Module này nhận tín hiệu từ bộ phát hiện xung đột và thực hiện chèn bong bóng bằng cách ép các tín hiệu điều khiển về 0, đồng thời giữ trạng thái PC.
- **Forwarding Unit (FORWARD\_UNIT.v):** Điều khiển các bộ MUX để nạp dữ liệu từ tầng MEM hoặc WB ngược trở lại tầng EX, giải quyết xung đột dữ liệu mà không cần dừng đường ống.

## CHƯƠNG 2. PHƯƠNG PHÁP LUÂN

---

```
// Forwarding từ tầng MEM về EX (Ưu tiên cao nhất)
if (EX_MEM_reg_write && (EX_MEM_rd != 0)
    && (EX_MEM_rd == ID_EX_rs))
    forwardA = 2'b10;

// Forwarding từ tầng WB về EX
else if (MEM_WB_reg_write && (MEM_WB_rd != 0)
    && (MEM_WB_rd == ID_EX_rs))
    forwardA = 2'b01;

// Không Forwarding (Lấy từ thanh ghi)
else
    forwardA = 2'b00;
```

*Đoạn mã 2.6: Logic ưu tiên trong khối Forwarding Unit (cho Rs)*

```
// Phát hiện Load-Use Hazard
if (ID_EX_mem_read &&
    ((ID_EX_rt == IF_ID_rs) || (ID_EX_rt == IF_ID_rt))) begin
    pc_stall = 1'b1;                      // Dừng PC
    IF_ID_stall = 1'b1;                    // Dừng thanh ghi IF/ID
    mux_control_hazard = 1'b1;            // Chèn Bubble (NOP)
end

// Phát hiện Control Hazard (Branch sau ALU)
if (branch && reg_write &&
    ((ID_EX_rd == IF_ID_rs) || (ID_EX_rd == IF_ID_rt))) begin
    pc_stall = 1'b1;
    IF_ID_stall = 1'b1;
    mux_control_hazard = 1'b1;
end
```

*Đoạn mã 2.7: Logic phát hiện xung đột trong HAZARD\_DETECTION\_UNIT*

```
// Chèn Bubble: Ép tín hiệu điều khiển về 0 khi có xung đột
assign reg_dst      = (stall) ? 1'b0 : reg_dst_in;
assign alu_src      = (stall) ? 1'b0 : alu_src_in;
assign alu_op       = (stall) ? 3'b000 : alu_op_in;
assign mem_read     = (stall) ? 1'b0 : mem_read_in;
assign mem_write    = (stall) ? 1'b0 : mem_write_in;
assign reg_write    = (stall) ? 1'b0 : reg_write_in;
assign mem_to_reg   = (stall) ? 1'b0 : mem_to_reg_in;
```

*Đoạn mã 2.8: Logic chèn Bubble trong MUX\_HAZARD\_CONTROL*

### 2.7.5 Module Pipeline Registers

Bốn module thanh ghi đường ống (IF\_ID, ID\_EX, EX\_MEM, MEM\_WB) đóng vai trò lưu trữ trạng thái giữa các chu kỳ xung nhịp.

- Các thanh ghi này được thiết kế tích hợp cơ chế Reset đồng bộ **Synchronous Reset** để phục vụ Flush và tín hiệu **Write Enable** để phục vụ Stall khi có xung đột Load-Use.

```
// Thanh ghi IF/ID với hỗ trợ Stall và Flush
always @ (posedge clk or posedge reset) begin
    if (reset) begin
        pc_next <= 32'b0;
        instruction <= 32'b0;
    end
    else if (stall) begin
        // Giữ nguyên trạng thái (cho Load-Use Hazard)
        pc_next <= pc_next;
        instruction <= instruction;
    end
    else if (flush) begin
        // Xóa lệnh sai (cho Branch/Jump misprediction)
        pc_next <= 32'b0;
        instruction <= 32'b0;
    end
    else begin
        pc_next <= pc_next_in;
        instruction <= instruction_in;
    end
end
```

*Đoạn mã 2.9: Thanh ghi IF/ID tích hợp Stall và Flush*

```
// Logic điều khiển Flush (xử lý Control Hazard)
assign flush = jump | (reg_equal_flag & branch_flag);
```

*Đoạn mã 2.10: Logic tạo tín hiệu Flush trong FLUSHCONTROL*

## CHƯƠNG 3. KIỂM THỬ VÀ KẾT QUẢ

### 3.1 Kiểm thử và Đánh giá kết quả

Để đảm bảo bộ vi xử lý hoạt động chính xác và xử lý triệt để các trường hợp xung đột, nhóm thực hiện kiểm thử dựa trên kịch bản tính toán số học có vòng lặp.

#### 3.1.1 Chiến lược kiểm thử

Nhóm xây dựng một kịch bản với bài toán tính tổng chuỗi số nguyên liên tiếp.

**Bài toán:** Tính tổng  $S = 1 + 2 + \dots + N$  với  $N = 10$ .

- **Đầu vào (Input):**  $N = 10$  (lưu trong thanh ghi \$s2).
- **Kỳ vọng (Output):**  $S = 55$  (lưu trong thanh ghi \$s1).
- **Tài nguyên sử dụng:**
  - \$s0: Biến đếm  $i$  (tăng từ 0 đến 10).
  - \$s1: Biến tích lũy tổng  $sum$ .
  - \$s2: Giới hạn lặp.

**Mục đích của kịch bản:** Bài test này được thiết kế để ép CPU phải đổi mặt và xử lý liên tục các loại xung đột:

1. **Data Hazard:** Việc cộng dồn liên tục vào thanh ghi \$s1 ( $sum = sum + i$ ) tạo ra sự phụ thuộc dữ liệu giữa các lệnh liền kề.
2. **Control Hazard:** Sự xuất hiện của lệnh nhảy không điều kiện J (quay lại đầu vòng lặp) và lệnh rẽ nhánh BEQ (kiểm tra điều kiện thoát) buộc bộ xử lý phải dự đoán hoặc xả đường ống (Flush).

#### 3.1.2 Cấu hình Testbench

Chương trình kiểm thử được viết dưới dạng mã máy và nạp trực tiếp vào bộ nhớ lệnh thông qua testbench tb\_top\_module.v.

Dưới đây là đoạn mã Assembly tương ứng và mã máy được nạp:

```
// Khởi tạo: $s0 = 0, $s1 = 0, $s2 = 10
Loop:
ADD  $s1, $s1, $s0      // sum = sum + i
BEQ  $s0, $s2, DONE     // Nếu i == 10 thì thoát
ADDI $s0, $s0, 1         // i++
J    LOOP                 // Quay lại vòng lặp
```

Done:

```
BEQ $s1, $s3, PASS // Kiểm tra kết quả cuối cùng
```

### 3.1.3 Kết quả mô phỏng

#### 3.1.3.1 Nhật ký thực thi

Quá trình chạy mô phỏng trên ModelSim cho thấy chương trình thực thi ổn định qua khoảng 60 chu kỳ xung nhịp. Dưới đây là trích đoạn log tại các thời điểm cuối cùng của quá trình tính toán:

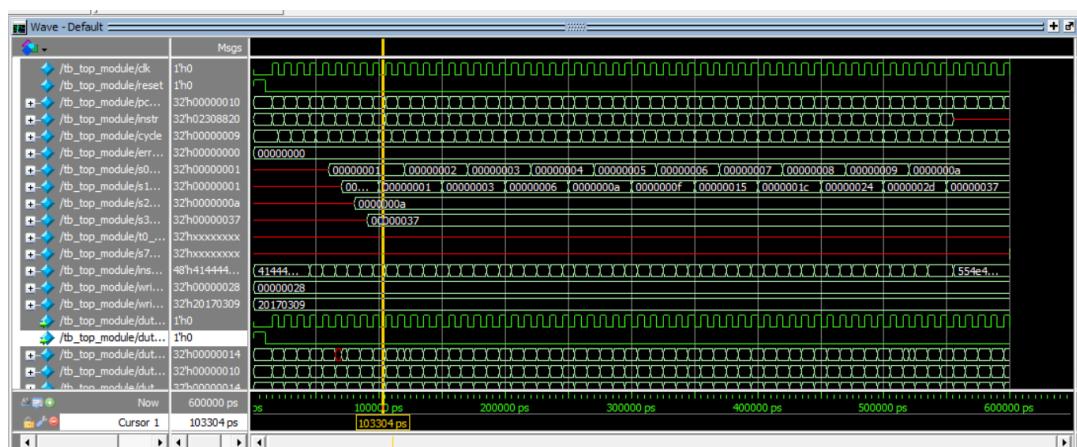
Cyc	PC	Inst	i (\$s0)	sum (\$s1)
...				
47	00000010	ADD	9	36
48	00000014	BEQ	9	36
49	00000018	ADDI	9	36
50	0000001c	J	9	36
51	00000010	ADD	10	36
52	00000014	BEQ	10	45
53	00000020	BEQ	10	45
54	00000028	ADDI	10	55

[SUCCESS] Test Passed! Sum = 55. PC reached target.

**Kết luận từ Log:** Tại chu kỳ thứ 54, thanh ghi \$s1 đã đạt giá trị 55. Chương trình kết thúc thành công.

#### 3.1.3.2 Phân tích dạng sóng

Hình 3.1 dưới đây minh họa tín hiệu chi tiết của các thanh ghi và đường ống trong quá trình chạy.



**Hình 3.1:** Dạng sóng mô phỏng quá trình tính tổng từ 1 đến 10

### Phân tích:

- **Dòng s0 (Biến i):** Giá trị tăng tuần tự từ 1 đến 10 (hiển thị dưới dạng Hex là  $1, 2, \dots, a$ ).
- **Dòng s1 (Biến Sum):** Giá trị được cộng dồn chính xác qua từng vòng lặp:  $0 \rightarrow 1 \rightarrow 3 \rightarrow 6 \dots \rightarrow 37_{16}$ .

**Phân tích hoạt động Pipeline:** Các lệnh ADDI, BEQ, và J được xử lý song song trong đường ống. Mặc dù có lệnh rẽ nhánh và phụ thuộc dữ liệu, dạng sóng cho thấy không có khoảng trống lớn hay tín hiệu "X" (Unknown), chứng tỏ cơ chế Forwarding và Hazard Unit đã hoạt động hiệu quả.

#### 3.1.4 Đánh giá và Kết luận

Dựa trên kết quả thực nghiệm, nhóm tiến hành đánh giá trên hai phương diện: Độ chính xác chức năng và hiệu năng xử lý.

##### 3.1.4.1 Kiểm chứng chức năng

- **Lý thuyết:** Tổng  $S = \frac{10 \times 11}{2} = 55$ .
- **Thực tế mô phỏng:** Thanh ghi \$s1 đạt giá trị  $32' h00000037$ .
- **Quy đổi:**  $37_{16} = (3 \times 16) + 7 = 55_{10}$ .
- **Kết luận:** Hệ thống tính toán chính xác tuyệt đối.

##### 3.1.4.2 Đánh giá hiệu năng (CPI Analysis)

Bảng dưới đây so sánh số lượng chu kỳ xung nhịp lý thuyết (nếu Pipeline lý tưởng) và thực tế đo đạc được:

**Bảng 3.1:** So sánh hiệu năng thực tế và lý thuyết

Thông số	Lý thuyết	Thực tế	Giải thích
Số lệnh thực thi	41	41	$10 \text{ vòng lặp} \times 4 \text{ lệnh} + 1 \text{ lệnh cuối}$
Số chu kỳ lý tưởng	45	-	$Cycles = Instrucs + 4$
Số chu kỳ thực tế	-	$\approx 55$	Phát sinh độ trễ do Flush đường ống
<b>CPI trung bình</b>	<b>1.0</b>	<b>1.34</b>	Chênh lệch do Branch Penalty

**Nhận xét:** Chỉ số CPI thực tế là 1.34 (lớn hơn 1.0) là điều hoàn toàn hợp lý đối với kiến trúc có rẽ nhánh.

- Mỗi lần lặp (lệnh J và BEQ sai dự đoán) gây ra tổn thất 1 chu kỳ (như đã phân tích ở Chương 2).

- Với 10 lần lặp, tổng tổn thất khoảng 10 chu kỳ, khớp với chênh lệch giữa lý thuyết (45) và thực tế (55).
- Điều này chứng minh cơ chế **Early Branching** đã hoạt động đúng thiết kế (nếu không có Early Branch, tổn thất sẽ là 2-3 chu kỳ/lần và CPI sẽ cao hơn nhiều).

#### *3.1.4.3 Kết luận chung*

Bảng tổng hợp trạng thái hoạt động của hệ thống:

Chức năng	Trạng thái	Minh chứng
Tính toán số học	<b>ĐẠT</b>	Kết quả thanh ghi \$s1 đúng
Điều khiển rẽ nhánh	<b>ĐẠT</b>	Thoát vòng lặp đúng thời điểm
Xử lý xung đột	<b>ĐẠT</b>	Pipeline chạy liên tục, không bị treo

Hệ thống vi xử lý đã hoạt động ổn định, đáp ứng đầy đủ các yêu cầu thiết kế đề ra.

## CHƯƠNG 4. KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

### 4.1 Kết quả đạt được

Sau quá trình nghiên cứu và thiết kế, nhóm đã hoàn thành trọn vẹn mục tiêu xây dựng một bộ vi xử lý MIPS 32-bit theo kiến trúc đường ống. Các kết quả cụ thể bao gồm:

1. **Kiến trúc đường ống hoàn chỉnh:** CPU hoạt động ổn định với 5 tầng chuẩn: IF, ID, EX, MEM, WB.
2. **Tối ưu hóa hiệu năng xử lý xung đột:**
  - **Kỹ thuật Early Branching:** Nhóm đã chuyển bộ so sánh về tầng ID, giúp giảm chi phí phạt xuống chỉ còn 1 chu kỳ xung nhịp so với 3 chu kỳ ở kiến trúc gốc.
  - **Kỹ thuật Full Forwarding:** Xử lý triệt để các xung đột dữ liệu, hạn chế tối đa việc phải dừng đường ống, giúp duy trì chỉ số CPI tiệm cận 1.
3. **Tổ chức mã nguồn mạch lạc:** Hệ thống được thiết kế theo tư duy mô-đun hóa cao. Các khối chức năng như ALU\_BIG\_MODULE, HAZARD\_DETECTION, FORWARD\_UNIT được tách biệt rõ ràng, giúp dễ dàng quản lý, gỡ lỗi và mở rộng sau này.
4. **Độ tin cậy:** Hệ thống đã vượt qua các bài kiểm thử tự động với các kịch bản tính toán phức tạp (vòng lặp, rẽ nhánh, phụ thuộc dữ liệu), cho kết quả chính xác trên mô phỏng ModelSim.

### 4.2 Hạn chế

Mặc dù đã đáp ứng được các yêu cầu cơ bản của đồ án, thiết kế hiện tại vẫn tồn tại một số hạn chế so với các vi xử lý thương mại thực tế:

- **Chưa hỗ trợ xử lý ngoại lệ (Exceptions):** CPU hiện tại chưa có cơ chế xử lý các tình huống bất thường như tràn số học (Overflow), lệnh không hợp lệ hoặc ngắt phần cứng (Interrupt). Khi gặp các lỗi này, hệ thống có thể hoạt động sai lệch hoặc không xác định.
- **Giới hạn về khả năng xử lý song song:** Thiết kế hiện tại là *Single-issue processor* (chỉ phát một lệnh mỗi chu kỳ). Tốc độ xử lý bị giới hạn bởi độ trễ của tầng dài nhất trong đường ống.
- **Hệ thống bộ nhớ đơn giản:** Dự án mới chỉ mô phỏng bộ nhớ lệnh và bộ nhớ dữ liệu lý tưởng, chưa tích hợp hệ thống bộ nhớ phân cấp (Cache L1, L2) để xử lý độ trễ truy cập thực tế.

### 4.3 Hướng phát triển

Dựa trên các hạn chế đã phân tích và các kiến thức nâng cao trong giáo trình (Chapter 4 - Patterson & Hennessy) [1], nhóm đề xuất các hướng phát triển trong tương lai:

#### 4.3.1 Tích hợp cơ chế xử lý ngoại lệ (Exceptions)

Để CPU hoạt động an toàn và tin cậy hơn, cần bổ sung hệ thống kiểm soát ngắn:

- **Thêm thanh ghi chuyên dụng:** Bổ sung thanh ghi EPC (Exception Program Counter) để lưu địa chỉ lệnh gây lỗi và thanh ghi Cause để lưu nguyên nhân lỗi.
- **Bộ điều khiển ngắn:** Thiết kế logic để tự động chuyển luồng thực thi sang trình xử lý ngoại lệ tại địa chỉ quy ước khi có sự cố.

#### 4.3.2 Nâng cao khả năng tính toán song song (Parallelism)

Để tăng cường hiệu năng xử lý, có thể áp dụng các kỹ thuật tiên tiến nhằm khai thác tính toán song song cấp lệnh (Instruction-Level Parallelism - ILP):

- **Đường ống siêu sâu (Deep Pipeline):** Chia nhỏ các tầng hiện tại để giảm thời gian trễ của từng tầng, từ đó cho phép tăng tần số xung nhịp lên cao hơn.
- **Phát lệnh đa luồng (Multiple Issue / Superscalar):** Nâng cấp phần cứng (nhân đôi PC, ALU, Register Ports) để cho phép nạp và thực thi nhiều lệnh trong cùng một chu kỳ xung nhịp, giảm CPI xuống dưới 1 (tức IPC > 1).

# **PHỤ LỤC**

## CHƯƠNG A. PHÂN CÔNG CÔNG VIỆC

**Bảng A.1:** Bảng phân công nhiệm vụ các thành viên

STT	Họ và tên	MSSV	Nhiệm vụ được phân công	Đóng góp
1	<b>Nguyễn Khánh Anh</b>	20237290	<ul style="list-style-type: none"> <li>Nghiên cứu tập lệnh MIPS</li> <li>Thiết kế Control Unit</li> <li>Vẽ sơ đồ Datapath cơ bản</li> </ul>	4
2	<b>Bùi Trần Hà Bình</b>	20237304	<ul style="list-style-type: none"> <li><b>Nhóm trưởng:</b> Quản lý tiến độ</li> <li>Viết Testbench</li> <li>Viết báo cáo cuối kỳ</li> <li>Soạn slide thuyết trình</li> </ul>	1
3	<b>Nghiêm Phú Quang Hưng</b>	20237341	<ul style="list-style-type: none"> <li>Code module ALU</li> <li>Code Register File</li> <li>Tích hợp Top Module</li> </ul>	3
4	<b>Dương Quang Minh</b>	20237362	<ul style="list-style-type: none"> <li>Code module Memory</li> <li>Code các thanh ghi Pipeline</li> <li>Chạy mô phỏng, chụp Waveform</li> </ul>	5
5	<b>Nguyễn Anh Văn</b>	20237404	<ul style="list-style-type: none"> <li>Code Hazard Detection</li> <li>Code Forwarding Unit</li> <li>Vẽ Datapath tổng quát</li> </ul>	2

**Đánh giá chung:** Tất cả các thành viên trong nhóm đều tích cực trao đổi, phối hợp chặt chẽ và hoàn thành tốt các nhiệm vụ được giao, đảm bảo báo cáo được hoàn thành đúng tiến độ và yêu cầu đề ra.

## CHƯƠNG B. KẾ HOẠCH LÀM VIỆC CỦA NHÓM

Nhóm đã thực hiện dự án trong thời gian 15 tuần theo lộ trình dưới đây.

**Bảng B.1:** Tiến độ thực hiện dự án

Tuần	Giai đoạn	Nội dung công việc chính	Trạng thái
1-3	Nghiên cứu	Tìm hiểu kiến trúc MIPS, nghiên cứu tài liệu tham khảo.	Hoàn thành
4-5	Thiết kế	Vẽ sơ đồ Datapath, xác định tín hiệu điều khiển.	Hoàn thành
6-9	Hiện thực	Viết code Verilog cho các module: ALU, RegFile, Pipeline Regs.	Hoàn thành
10-12	Xử lý lỗi	Tích hợp Hazard Unit, Forwarding Unit, gỡ lỗi xung đột.	Hoàn thành
13-14	Kiểm thử	Viết Testbench, chạy mô phỏng, kiểm tra kết quả Waveform.	Hoàn thành
15	Báo cáo	Hoàn thành báo cáo tổng kết, soạn slide và chuẩn bị demo.	Hoàn thành

## CHƯƠNG C. HƯỚNG DẪN KHAI THÁC TÀI LIỆU ĐÍNH KÈM

Toàn bộ mã nguồn, tài liệu tham khảo và nhật ký làm việc được nén trong file: BaoCaoCK\_NhomX\_SourceCode.zip.

### 1. Cấu trúc thư mục

```
Project_MIPS/
|--- src/                      // Chứa mã nguồn Verilog
|   |--- ALU_BIG_MODULE.v
|   |--- CONTROL_UNIT.v
|   |--- HAZARD_DETECTION.v
|   `--- ...
|--- testbench/                // Chứa file kiểm thử
|   |--- tb_top_module.v
|   `--- code_test.asm          // Mã nguồn Assembly mẫu
|--- docs/                      // Tài liệu bổ trợ
|   |--- Slide_BaoCao.pdf
|   `--- Datapath_TQ.pdf
|--- ref/                       // Chứa tài liệu tham khảo
`--- README.txt                 // Hướng dẫn sử dụng
```

### 2. Hướng dẫn chạy mô phỏng

Để tái hiện kết quả kiểm thử, vui lòng thực hiện các bước sau trên phần mềm ModelSim/Questasim:

1. Mở ModelSim, tạo Project mới và trỏ đường dẫn đến thư mục Project\_MIPS.
2. Thêm toàn bộ file trong thư mục src/ và testbench/.
3. Thực hiện **Compile All** để biên dịch mã nguồn.
4. Tại tab Library, chọn work > tb\_top\_module, chuột phải chọn **Simulate**.
5. Trong cửa sổ Instance, chuột phải vào tb\_top\_module, chọn **Add to > Wave > All items in region** để thêm các tín hiệu vào Waveform.
6. Chạy lệnh run 2000ns hoặc nhấn nút **Run All**.

## TÀI LIỆU THAM KHẢO

- [1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 5th. Waltham, MA: Morgan Kaufmann, 2014, ISBN: 978-0-12-407726-3.