



ĐẠI HỌC
BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY
OF SCIENCE AND TECHNOLOGY



KHOA TOÁN - TIN
FACULTY OF MATHEMATICS
AND INFORMATICS

THIẾT KẾ & MÔ PHỎNG CPU PIPELINE 5 TẦNG VỚI HAZARD DETECTION UNIT

Người thực hiện:

Nguyễn Khánh Anh - 20237290

Bùi Trần Hà Bình - 20237304

Nghiêm Phú Quang Hưng - 20237341

Dương Quang Minh - 20237362

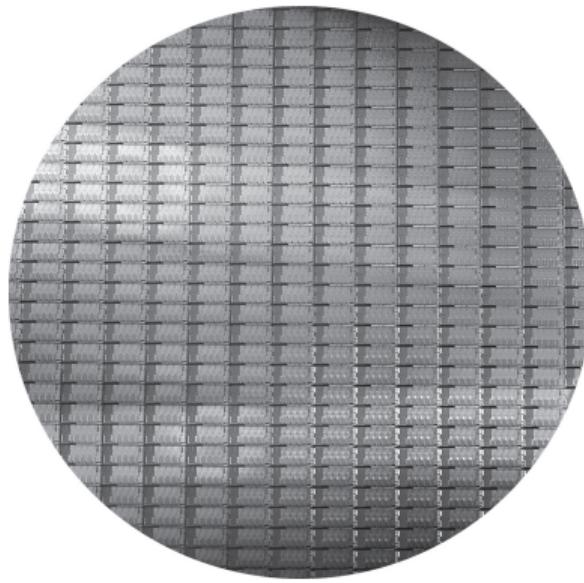
Nguyễn Anh Văn - 20237404

ONE LOVE. ONE FUTURE.

Nội dung trình bày I

1. Tổng quan dự án & Cơ sở lý thuyết
2. Thiết kế Datapath 5 tầng cơ bản
3. Xử lý Xung đột dữ liệu (Data Hazards)
4. Khối Điều khiển & Xung đột Rẽ nhánh (Control Unit & Control Hazards)
5. Kiểm thử & Demo

1. Mở đầu: Trái tim của Kỷ nguyên số



[Hình:](#) Tấm Wafer 300mm chứa các vi xử lý Intel Core i7

"Làm thế nào để thiết kế và tổ chức hàng tỷ transistor trong các ô vuông này?"

2. Giới thiệu Đề tài

Thiết kế Bộ vi xử lý MIPS 32-bit Pipelined

Dự án tập trung tái hiện kiến trúc phần cứng của vi xử lý MIPS (Microprocessor without Interlocked Pipelined Stages) - một kiến trúc RISC kinh điển.

Mục tiêu cốt lõi:

- ▶ Hiểu sâu về luồng dữ liệu (Datapath) và tín hiệu điều khiển (Control).
- ▶ Chuyển đổi từ lý thuyết sách giáo khoa sang phần cứng thực tế (Verilog).
- ▶ Giải quyết bài toán hiệu năng bằng kỹ thuật Đường ống (Pipelining).

3. Phương pháp xung nhịp (Clocking Methodology)

Tại sao cần phương pháp xung nhịp?

Để đảm bảo tính **dễ dự đoán**:

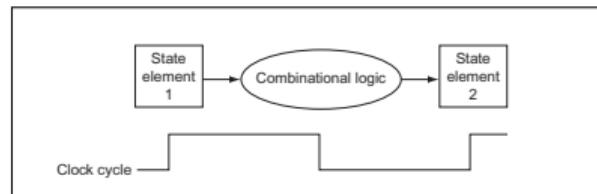
- ▶ Xác định rõ ràng thời điểm tín hiệu được Đọc và Ghi.
- ▶ Ngăn chặn hiện tượng "đua tín hiệu" gây ra giá trị không xác định.

Cơ chế kích hoạt theo cạnh:

- ▶ Mọi cập nhật trạng thái chỉ xảy ra tại **cạnh lên** của xung nhịp.

Chu trình hoạt động:

1. Đọc dữ liệu từ phần tử trạng thái 1.
2. Xử lý qua logic tổ hợp.
3. Ghi kết quả vào phần tử trạng thái 2 tại cạnh lên tiếp theo.

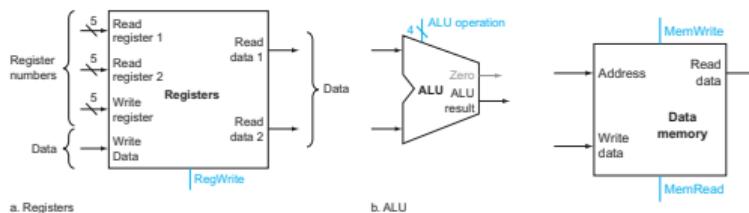
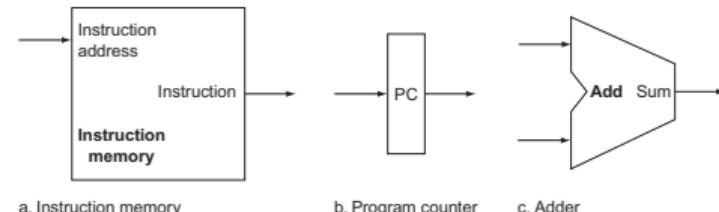


Hình: Mối quan hệ giữa trạng thái & logic

4. Các thành phần cơ bản của Datapath

Để thực thi một lệnh, CPU cần các khối chức năng chính:

- 1. Program Counter (PC):** Chứa địa chỉ lệnh.
- 2. Instruction Memory (IM):** Nơi chứa mã chương trình.
- 3. Register File (Reg):** Tập thanh ghi ($32 \times 32\text{-bit}$).
- 4. ALU:** Thực hiện tính toán (+, -, AND, OR).
- 5. Data Memory (DM):** Bộ nhớ lưu trữ dữ liệu.



Các phần tử trạng thái (State Elements) và logic tổ hợp chính

5. Thiết kế Đơn chu kỳ (Single-Cycle Implementation)

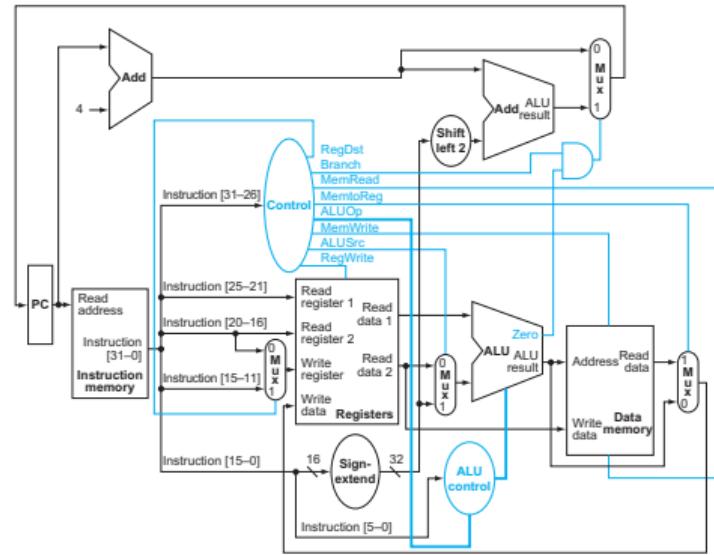
Định nghĩa

Là thiết kế mà mỗi lệnh được thực thi trọn vẹn trong **đúng một chu kỳ xung nhịp** ($CPI = 1$).

Nguyên tắc phần cứng

Do mọi việc phải xong trong 1 chu kỳ:

- ▶ **Không tái sử dụng tài nguyên:** Mỗi đơn vị chức năng chỉ được dùng 1 lần mỗi chu kỳ.
- ▶ **Nhân bản linh kiện:** Cần các bộ cộng (Adder) riêng để tính PC, không dùng chung với ALU chính được.
- ▶ **Tách biệt bộ nhớ:** Cần Instruction Memory và Data Memory riêng biệt để tránh xung đột.



Hình: Sơ đồ đơn chu kỳ: Cần nhiều bộ cộng và bộ nhớ tách biệt

6. Tại sao Single-Cycle không còn được sử dụng?

Các vấn đề cốt lõi

- ▶ **Chu kỳ xung nhịp dài:** Bị giới hạn bởi **đường đi dài nhất** của lệnh chậm nhất (LW đi qua 5 khối: IM → Reg → ALU → DM → Reg).
- ▶ **Lãng phí tài nguyên:** Các khối hoạt động tuần tự. Khi ALU đang tính toán, Bộ nhớ và các khối khác phải "ngồi chơi" chờ đợi.
- ▶ **Hiệu suất thấp:** Các lệnh thực thi nhanh (như ADD) buộc phải kéo dài thời gian chờ cho bằng lệnh chậm nhất.

Hệ quả thực tế:

- ▶ Vi phạm nguyên tắc thiết kế kiến trúc: "*Make the common case fast*".
- ▶ **Không phù hợp thực tế:** Không thể đạt tần số xung nhịp cao (GHz) để đáp ứng nhu cầu xử lý của các CPU hiện đại (như Core i7, Ryzen).

7. Tổng quan về Pipelining

"Never waste time."

— American Proverb —

Định nghĩa

"Pipelining is an implementation technique in which multiple instructions are overlapped in execution. Today, pipelining is nearly universal." (Patterson & Hennessy)

Bản chất kỹ thuật

Thay vì thực thi tuần tự (Sequential), kỹ thuật Đường ống cho phép các công đoạn của nhiều lệnh diễn ra **đồng thời**:

- ▶ **Cơ chế:** Khai thác xử lý song song ở cấp độ lệnh (Instruction-Level Parallelism - ILP).
- ▶ **Mục tiêu:** Tối đa hóa việc sử dụng tài nguyên phần cứng để tăng thông lượng (Throughput).

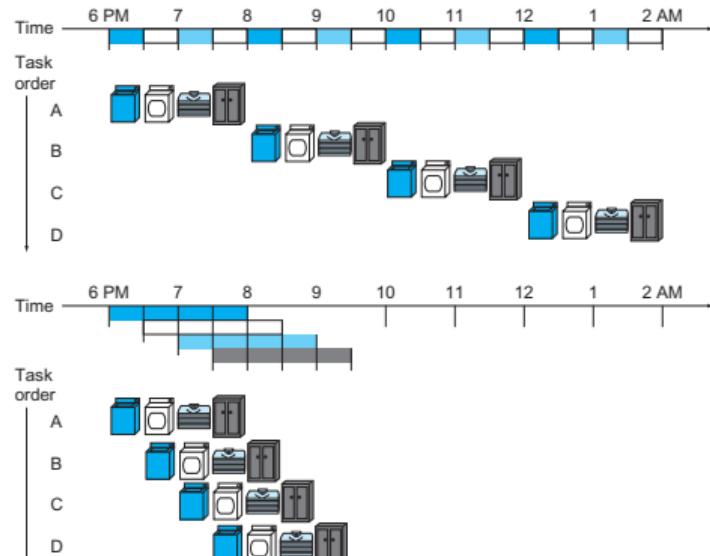
8. Ví dụ minh họa: Dây chuyền giặt Ủi

Quy trình 4 giai đoạn: Giặt → Sấy → Gấp
→ Cất.

- ▶ **Non-pipelined:** $4 \text{ mẻ đồ} \times 2\text{h} = 8 \text{ giờ}$.
- ▶ **Pipelined:** Chồng gối nhau = **3.5 giờ**.

Nghịch lý Pipeline

- ▶ **Độ trễ (Latency):** Không giảm (Vẫn mất 2h để giặt xong trọn vẹn 1 mẻ).
- ▶ **Thông lượng (Throughput):** Tăng mạnh (Hoàn thành số lượng mẻ giặt nhiều hơn trong cùng 1 giờ).



Hình: Sơ đồ thời gian giặt Ủi

9. So sánh Hiệu năng: Single-Cycle vs Pipeline

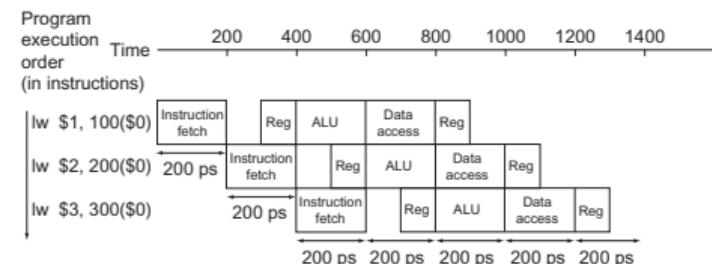
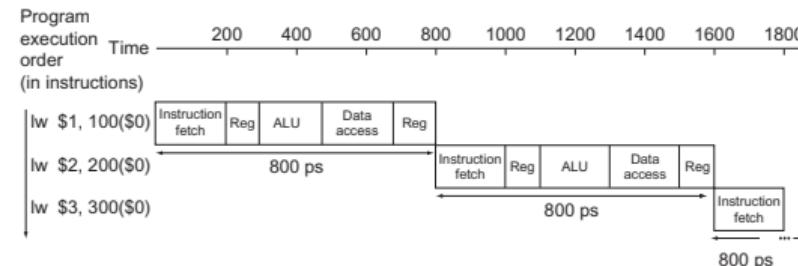
Nguyên lý Tăng tốc

- ▶ **Lý tưởng:** Tốc độ tăng gấp N lần (với N là số tầng Pipeline).
- ▶ **Ví dụ:** Pipeline 5 tầng → Nhanh gấp 5 lần.

Thực tế (Reality)

Tốc độ thực tế luôn **nhỏ hơn** lý tưởng do:

1. **Độ trễ bổ sung (Overhead):** Độ trễ thêm vào của các thanh ghi đường ống (Pipeline Registers).
2. **Mất cân bằng (Imbalance):** Các tầng không tồn thời gian bằng nhau hoàn toàn.



Hình: So sánh thời gian thực thi

9. Kiến trúc MIPS 5 Tầng của Nhóm

Áp dụng lý thuyết Pipeline vào đồ án: Hệ thống được thiết kế dựa trên kiến trúc MIPS 32-bit chuẩn, chia Datapath thành 5 tầng độc lập:

1. **IF (Instruction Fetch)**: Nạp lệnh từ bộ nhớ.
2. **ID (Instruction Decode)**: Giải mã lệnh & Đọc thanh ghi.
3. **EX (Execute)**: Thực thi phép tính (ALU).
4. **MEM (Memory Access)**: Truy cập bộ nhớ dữ liệu.
5. **WB (Write Back)**: Ghi kết quả về thanh ghi.

(Chi tiết thiết kế sẽ được trình bày ở phần sau)

Tập lệnh hỗ trợ (MIPS ISA)

Hỗ trợ các lệnh cơ bản: ADD, SUB, AND, OR (R-type), LW, SW, ADDI (I-type), BEQ, J (Control flow).



10. Công cụ thực hiện & Thách thức

Công cụ phát triển

- ▶ **Ngôn ngữ:** Verilog HDL (Mô tả phần cứng).
- ▶ **Mô phỏng:** ModelSim SE-64 2020.4 (Kiểm tra dạng sóng và gỡ lỗi logic).
- ▶ **Soạn thảo:** Visual Studio Code.

⇒ Đây là tiền đề để nhóm đi vào chi tiết thiết kế Datapath trong phần tiếp theo.

Thách thức & Phạm vi đề tài

Việc song song hóa sinh ra các **xung đột** phá vỡ tính đúng đắn:

1. **Data Hazard:** Dữ liệu chưa kịp ghi vào thanh ghi đã bị lệnh sau đọc.
2. **Control Hazard:** Lệnh rẽ nhánh (Branch/Jump) làm sai lệch dòng nạp lệnh.

→ **Mục tiêu chính:** Thiết kế Datapath 5 tầng
+ Các khối xử lý Hazard
(Forwarding/Stall/Flush).

Nội dung trình bày I

1. Tổng quan dự án & Cơ sở lý thuyết
2. Thiết kế Datapath 5 tầng cơ bản
3. Xử lý Xung đột dữ liệu (Data Hazards)
4. Khối Điều khiển & Xung đột Rẽ nhánh (Control Unit & Control Hazards)
5. Kiểm thử & Demo

1. Cấu trúc mã nguồn

Tổ chức Module

- ▶ **TOP_MODULE.v:** Kết nối Datapath với Control Unit và Hazard Unit.
- ▶ **Thiết kế phân tầng:** Mỗi giai đoạn (Stage) tương ứng với các file .v riêng biệt.
- ▶ **Testbench:** File tb_top_module.v kiểm thử toàn hệ thống.

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_MEMORY.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 ...
ALU_BIG_MODULE.v	✓ Verilog 1	12/28/2025 08:59:54 ...
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: Cây thư mục dự án trong ModelSim

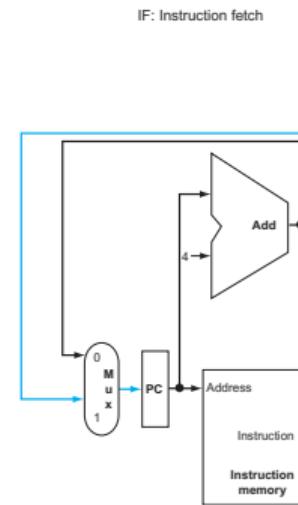
2. Chi tiết Tầng Nạp lệnh (IF - Instruction Fetch)

Tầng 1: Nạp lệnh (Instruction Fetch)

Mục đích: Lấy lệnh từ bộ nhớ dựa trên địa chỉ PC và chuẩn bị địa chỉ tiếp theo.

- ▶ **PC (Program Counter):** Thanh ghi chứa địa chỉ lệnh hiện tại.
- ▶ **Adder:** Tính toán địa chỉ tuần tự ($PC + 4$).
- ▶ **Instruction Memory:** Lưu trữ mã máy.
- ▶ **MUX PC:** Lựa chọn nguồn địa chỉ cho PC: $PC + 4$ (tín hiệu bình thường) hoặc địa chỉ đích (khi có rẽ nhánh/nhảy).

IF: Instruction fetch

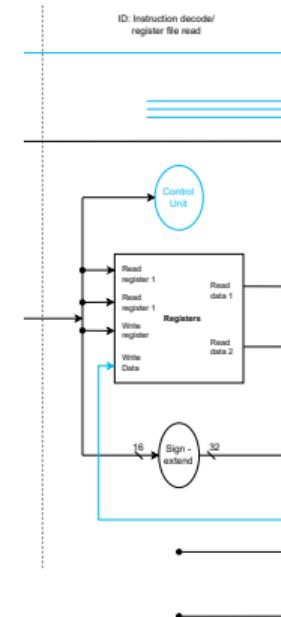


Hình: Sơ đồ khối chức năng tầng IF

3. Chi tiết Tầng Giải mã & Đọc thanh ghi (ID)

Mục đích: Giải mã lệnh để xác định thao tác và lấy dữ liệu từ tập thanh ghi.

- ▶ **Control Unit:** Bộ điều khiển trung tâm, giải mã Opcode để sinh tín hiệu điều khiển.
- ▶ **Register File:** Tập hợp 32 thanh ghi 32-bit (đọc 2 nguồn rs , rt , ghi 1 đích rd/rt tại pha WB).
- ▶ **Sign-Extend:** Khối mở rộng hằng số từ 16-bit lên 32-bit.



Hình: Sơ đồ khối chức năng tầng ID

4. Chi tiết Tầng Thực thi (EX - Execute)

Mục đích: Thực hiện các phép tính toán học, logic hoặc tính toán địa chỉ bộ nhớ.

► ALU (Arithmetic Logic Unit):

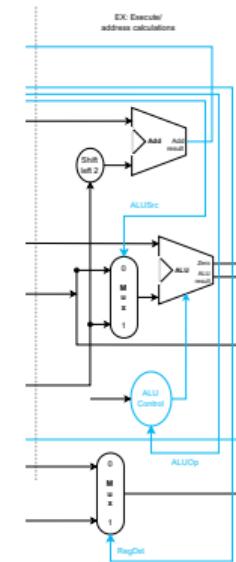
- Thực hiện các phép toán: ADD, SUB, AND, OR, XOR.
- Nhận tín hiệu điều khiển từ bộ **ALU Control** để xác định phép toán cụ thể.

► Bộ MUX ALUSrc:

- Lựa chọn toán hạng đầu vào thứ hai cho ALU.
- Chọn giữa giá trị thanh ghi (*rt*) hoặc giá trị tức thời đã mở rộng dấu (*Immediate*).

► Bộ MUX RegDst:

- Lựa chọn địa chỉ thanh ghi đích để ghi kết quả.
- Chọn giữa trường *rt* (cho lệnh I-type) hoặc *rd* (cho lệnh R-type).

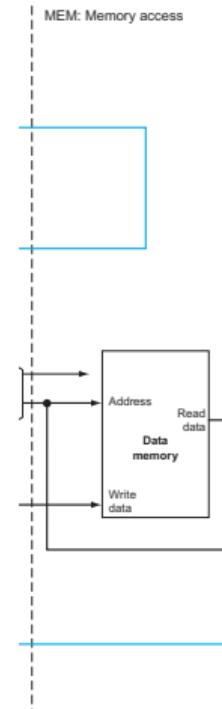


Hình: Sơ đồ khối chức năng tầng EX

5. Chi tiết Tầng Truy xuất bộ nhớ (MEM - Memory Access)

Mục tiêu: Đọc hoặc ghi dữ liệu vào bộ nhớ RAM dựa trên địa chỉ đã tính toán từ tầng EX.

- ▶ **Data Memory:** Bộ nhớ dữ liệu lưu trữ các biến, mảng và cấu trúc dữ liệu của chương trình.
- ▶ **Cơ chế Đọc (Load):** Truy xuất nội dung từ một ô nhớ trong RAM để đưa về đường dữ liệu.
- ▶ **Cơ chế Ghi (Store):** Lưu giá trị từ một thanh ghi vào vị trí xác định trong bộ nhớ RAM.

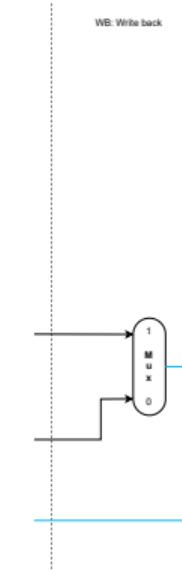


Hình: Sơ đồ khái niệm tầng MEM

6. Chi tiết Tầng Ghi ngược (WB - Write Back)

Mục tiêu: Cập nhật giá trị cuối cùng vào tập thanh ghi để hoàn tất thực thi lệnh.

- ▶ **MUX Write Back:** Bộ chọn nguồn dữ liệu để quyết định giá trị nào được ghi lại vào Register File.
- ▶ **Nguồn từ ALU:** Kết quả của các phép tính toán học hoặc logic (lệnh R-type, ADDI...).
- ▶ **Nguồn từ Memory:** Dữ liệu vừa được đọc lên từ RAM (lệnh LW).



Hình: Sơ đồ khối chức năng tầng WB

7. Luồng dữ liệu (Datapath Flow) & Các ngoại lệ

Quy tắc "Trái sang Phải" (Left-to-Right Flow)

Theo thiết kế chuẩn MIPS Pipeline, dữ liệu chảy tuần tự qua 5 tầng:

IF → **ID** → **EX** → **MEM** → **WB**

Hai ngoại lệ quan trọng

Để hệ thống hoạt động đúng, có 2 luồng dữ liệu đi "ngược":

- 1. Giai đoạn Write Back (WB):** Kết quả từ cuối đường ống (tầng 5) được vòng ngược về đầu đường ống để ghi vào **Register File** (tầng 2).
- 2. Cập nhật PC (PC Selection):** Địa chỉ rẽ nhánh/nhảy (Branch Target) được tính toán và đưa ngược về **PC** (tầng 1) để chọn lệnh tiếp theo.

→ Đây chính là nguyên nhân gốc rễ gây ra các vấn đề về Hazard.

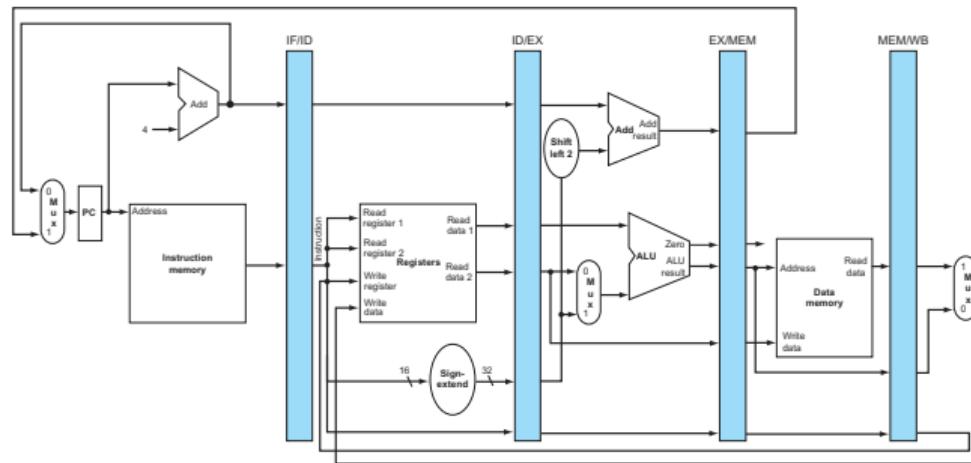
8. Thanh ghi Đường ống (Pipeline Registers)

Các thanh ghi này đóng vai trò là "vách ngăn" lưu trữ trạng thái giữa các chu kỳ xung nhịp:

- ▶ **IF/ID (64-bit):** Lưu PC+4 và Lệnh 32-bit.
- ▶ **ID/EX:** Lưu các giá trị thanh ghi (rs , rt), hằng số mở rộng (Imm) và các tín hiệu điều khiển.
- ▶ **EX/MEM:** Lưu kết quả ALU, cờ Zero và dữ liệu cần ghi ('Store Val').
- ▶ **MEM/WB:** Lưu dữ liệu đọc từ bộ nhớ ('Read Data') hoặc kết quả ALU để chuẩn bị ghi lại.

9. Sơ đồ Datapath với thanh ghi đường ống

- ▶ **Đặc điểm:** Datapath được chia thành các lát cắt dọc bởi các thanh ghi đường ống.
- ▶ **Cơ chế:** Mỗi tầng hoạt động độc lập trên một lệnh khác nhau trong cùng một chu kỳ xung nhịp.



⇒ Các thanh ghi đường ống lưu trữ trạng thái của lệnh để nhường chỗ cho lệnh kế tiếp nạp vào tầng trước đó.

10. Hệ thống điều khiển đường ống (Pipelined Control)

Làm thế nào để đảm bảo đúng tín hiệu điều khiển cho đúng lệnh ở đúng tầng?

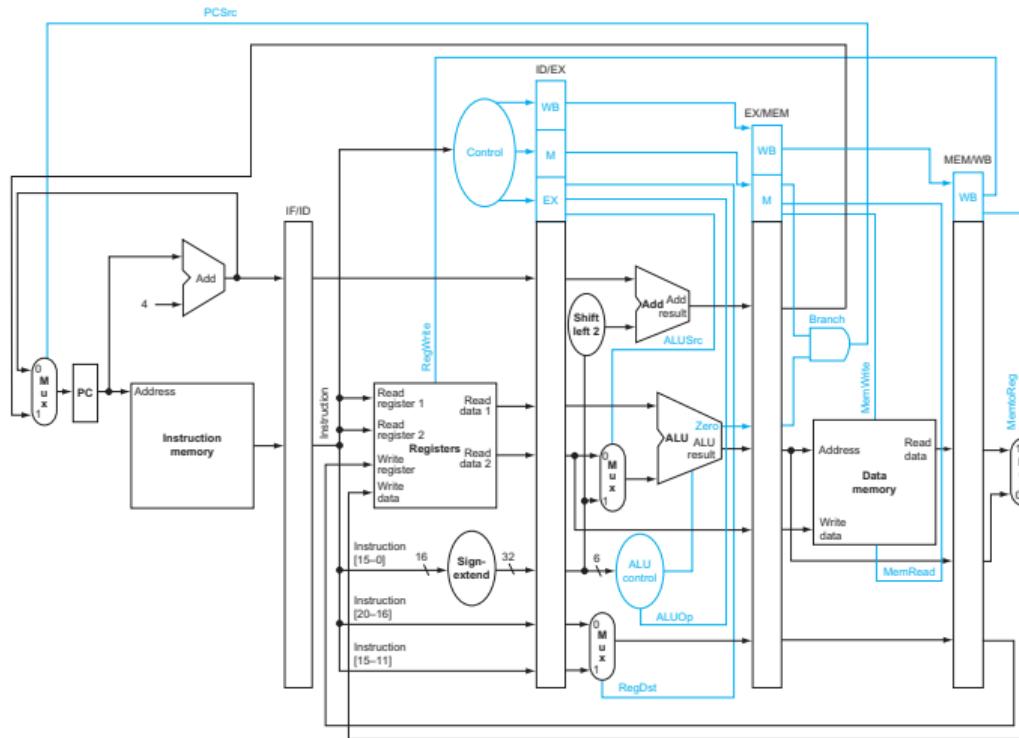
Chiến lược lan truyền

Tín hiệu điều khiển không được gửi trực tiếp toàn bộ ngay lập tức. Chúng được **nhóm lại và chuyển tiếp** qua các thanh ghi đường ống:

1. **Tại ID:** Control Unit sinh ra toàn bộ 9 tín hiệu (RegDst, ALUSrc, MemRead...).
2. **Tại ID/EX:** Giữ lại các tín hiệu cho EX (RegDst, ALUOp), đẩy tiếp nhóm MEM và WB.
3. **Tại EX/MEM:** Giữ lại các tín hiệu cho MEM (MemRead, MemWrite), đẩy tiếp nhóm WB.
4. **Tại MEM/WB:** Chỉ còn lại tín hiệu cho WB (RegWrite, MemtoReg).

⇒ **Đảm bảo tín hiệu điều khiển luôn đi song hành cùng dữ liệu của lệnh đó.**

11. Sơ đồ Datapath với tín hiệu điều khiển



Hình: Sơ đồ kết nối tín hiệu điều khiển vào các thanh ghi đường ống

Nội dung trình bày I

1. Tổng quan dự án & Cơ sở lý thuyết
2. Thiết kế Datapath 5 tầng cơ bản
3. Xử lý Xung đột dữ liệu (Data Hazards)
4. Khối Điều khiển & Xung đột Rẽ nhánh (Control Unit & Control Hazards)
5. Kiểm thử & Demo



1. Vấn đề: Xung đột dữ liệu (Data Hazard)

Định nghĩa RAW

Xảy ra khi lệnh sau đọc thanh ghi trước khi lệnh trước kịp ghi xong giá trị mới (Read-After-Write).

Ví dụ mã gây lỗi

ADD \$t1, \$t2, \$t3

(Ghi \$t1 tại WB - Chu kỳ 5)

SUB \$t4, \$t1, \$t5

(Đọc \$t1 tại ID - Chu kỳ 2)

→ **Kết quả:** SUB đọc sai giá trị.

⇒ Giải pháp: Phối hợp **Forwarding Unit** và **Hazard Detection Unit**.

Trường hợp khó: Load-Use

Lệnh sử dụng kết quả ngay sau lệnh LW.

- ▶ **Nguyên nhân:** Dữ liệu LW chỉ có sẵn sau tầng MEM (CK 4), lệnh sau cần dùng tại EX (CK 3).
- ▶ **Hệ quả:** Không thể xử lý chỉ bằng Forwarding đơn thuần.

*Ghi chú: Project sử dụng **Hazard Detection Unit** để chèn NOP cho Load-Use.*

2. Phân loại Xung đột dữ liệu (Classification)

Dựa trên khoảng cách giữa lệnh gây lỗi và lệnh bị ảnh hưởng:

Loại 1: Xung đột EX (Khoảng cách 1)

Lệnh ngay trước ghi vào thanh ghi mà lệnh hiện tại cần dùng.

- ▶ $EX/MEM.Rd = ID/EX.Rs$
- ▶ $EX/MEM.Rd = ID/EX.Rt$

Loại 2: Xung đột MEM (Khoảng cách 2)

Lệnh trước đó nữa ghi vào thanh ghi mà lệnh hiện tại cần dùng.

- ▶ $MEM/WB.Rd = ID/EX.Rs$
- ▶ $MEM/WB.Rd = ID/EX.Rt$

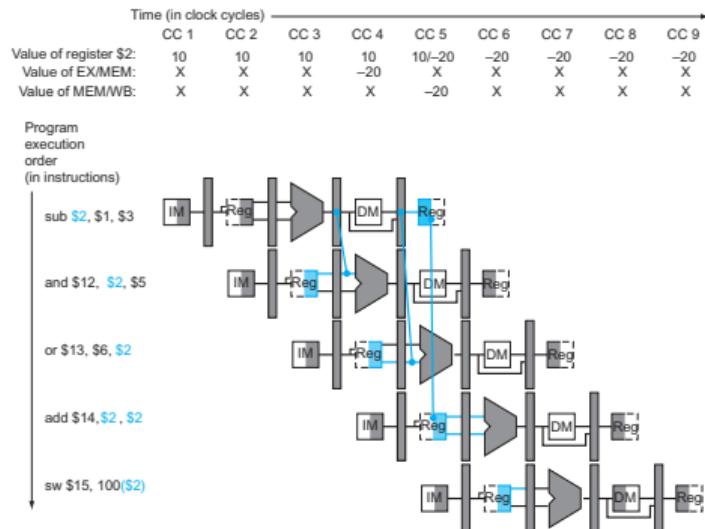
3. Cơ chế Forwarding (Chuyển tiếp dữ liệu)

Nguyên lý hoạt động

Lấy dữ liệu trực tiếp từ các tầng sau (EX/MEM hoặc MEM/WB) về đầu vào ALU mà không chờ ghi vào Register File.

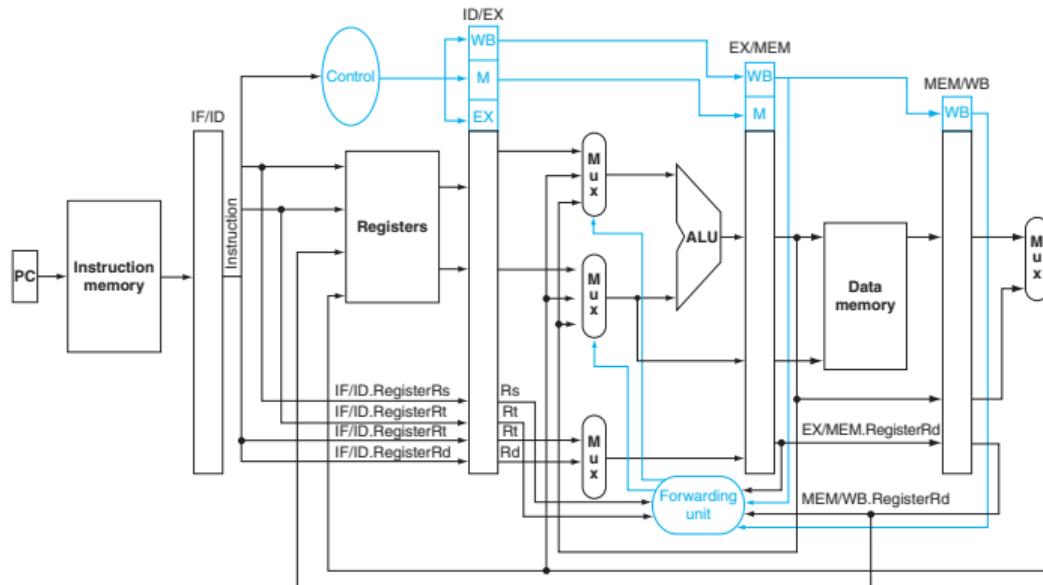
Điều kiện chuyển tiếp:

- Thanh ghi đích khác \$zero.
- Tín hiệu RegWrite của lệnh trước được bật.



Hình: Sơ đồ đa chu kỳ thể hiện Forwarding

4. Datapath tích hợp Forwarding Unit



Hình: Kiến trúc Datapath với bộ Forwarding Unit

- ▶ **Đơn vị chuyển tiếp:** So sánh các địa chỉ thanh ghi để điều khiển MUX đầu vào ALU.

5. Giải pháp Stall (Load-Use Hazard)

Trường hợp: Lệnh sử dụng kết quả ngay sau lệnh LW. Dữ liệu chỉ có sẵn sau tầng MEM nên kỹ thuật Forwarding không thể xử lý kịp thời.

1. Điều kiện phát hiện Xung đột (Hazard Detection Logic)

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

→ **Hành động:** Đinh trệ (Stall) đường ống 01 chu kỳ xung nhịp.

2. Cơ chế thực hiện NOP

Chèn "Bong bóng" (Bubble) vào ID/EX:

- ▶ Ép các tín hiệu điều khiển quan trọng về 0 (RegWrite, MemWrite...).
- ▶ Lệnh tại EX trở thành lệnh "không thao tác" (No Operation).

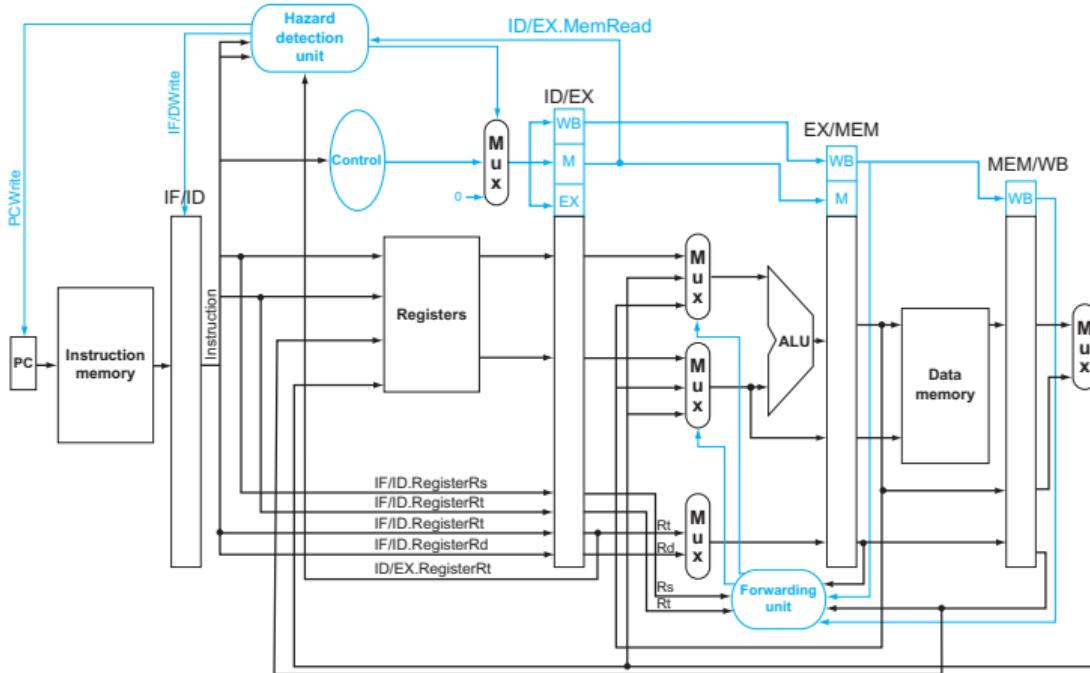
3. Đóng băng hệ thống

Giữ lệnh để xử lý lại:

- ▶ **PC stall:** Không nạp lệnh mới, giữ nguyên địa chỉ.
- ▶ **IF/ID stall:** Giữ lại lệnh hiện tại để giải mã lại ở chu kỳ sau.

Sau 1 chu kỳ Stall, dữ liệu sẽ được xử lý tiếp bằng cơ chế Forwarding.

6. Datapath hoàn chỉnh với Hazard Detection



Hình: Datapath tích hợp cả Hazard Detection và Forwarding

7. Triển khai thực tế trong Project

Nhóm sử dụng cơ chế **Stall + Bubble (NOP)** toàn diện:

Module MUX_HAZARD_CONTROL

Khi phát hiện Hazard, toàn bộ 7 tín hiệu điều khiển quan trọng được ép về 0:

- ▶ `reg_write = 0, mem_write = 0` (Ngăn ghi dữ liệu sai).
- ▶ `alu_op = 0, mem_read = 0...` (Tạo "bong bóng" trong Pipeline).
- ▶ **Tác động:** Đóng băng PC và IF/ID, cho phép lệnh nạp (LW) đi tới tầng MEM để lấy dữ liệu.

8. Ví dụ minh họa thực thi Stall

lw \$t0, 0(\$s0) → Đang ở ID/EX (cần đọc Mem)

add \$t1, \$t0, \$s1 → Đang ở IF/ID (cần dùng \$t0)

Chu kỳ	Trạng thái hệ thống
N	Hazard Detection phát hiện trùng khớp: $ID_EX_rt = IF_ID_rs = \$t0$. Bật tín hiệu Stall.
N+1	<ol style="list-style-type: none">PC giữ nguyên địa chỉ lệnh kế tiếp.IF/ID giữ lại lệnh ADD.ID/EX nhận BUBBLE (tất cả Control = 0).Lệnh LW tiến tới tầng MEM để lấy dữ liệu.
N+2	<ol style="list-style-type: none">Stall kết thúc. Lệnh ADD được giải mã bình thường.Dữ liệu từ tầng MEM/WB được Forwarding về cho ALU thực hiện phép cộng.

Nội dung trình bày I

1. Tổng quan dự án & Cơ sở lý thuyết
2. Thiết kế Datapath 5 tầng cơ bản
3. Xử lý Xung đột dữ liệu (Data Hazards)
4. Khối Điều khiển & Xung đột Rẽ nhánh (Control Unit & Control Hazards)
5. Kiểm thử & Demo

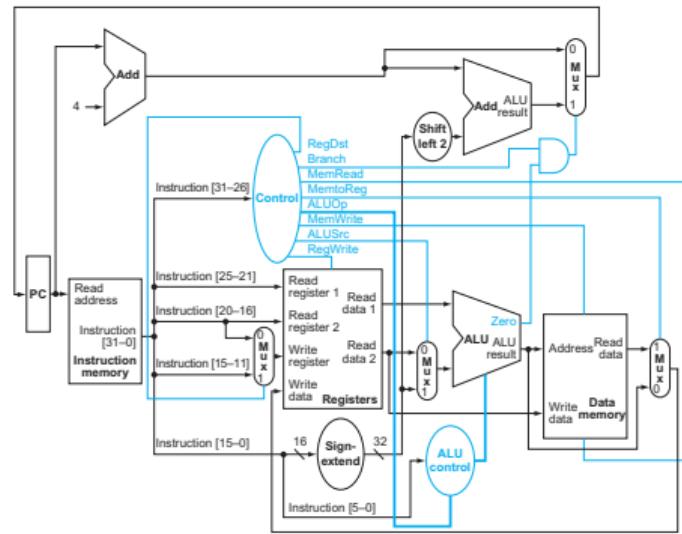
1. Phân tích Logic Khối Điều khiển (Control Unit)

Giải mã phân cấp

- ▶ **Main Control:** Giải mã Opcode [31:26] để sinh ra các tín hiệu điều khiển chính (Branch, Jump, MemRead...).
- ▶ **ALU Control:** Giải mã phụ thuộc, kết hợp ALUOp và trường funct (0-5) để điều khiển ALU.

Điểm đặc biệt trong Code

Sử dụng **ALUOp 3-bit** (thay vì 2-bit) giúp phân biệt rõ nhóm lệnh I-Type (addi, andi, ori...) ngay từ tầng Decode.



Hình: Sơ đồ phân cấp điều khiển thực tế

2. Chiến lược: Predict Not Taken (Dự đoán không nhảy)

Vấn đề

Việc dừng pipeline (Stall) chờ rẽ nhánh là quá chậm. Giải pháp tối ưu là dự đoán nhánh sẽ **không xảy ra**.

Cơ chế hoạt động

1. Luôn nạp lệnh tiếp theo theo tuần tự ($PC + 4$).
2. Nếu rẽ nhánh **không xảy ra**: Tiếp tục bình thường (Không mất phí).
3. Nếu rẽ nhánh **xảy ra (Taken)**: Các lệnh đã lỡ nạp (Fetch) và giải mã (Decode) bị coi là "rác" và phải bị loại bỏ.
→ Để loại bỏ lệnh, ta sử dụng kỹ thuật **Flush** (Xả đường ống).

3. Cơ chế thực hiện Flush

Để "hủy" một lệnh, ta chuyển đổi nó thành **NOP** (No Operation) bằng cách xóa các tín hiệu điều khiển về 0.

Logic phần cứng

Khác với Load-Use Stall (chỉ cần xóa control ở ID), Flush yêu cầu xóa cả lệnh đang nằm trong thanh ghi **IF/ID**.

Tác động

Các tầng sau (EX, MEM, WB) sẽ nhận được lệnh rỗng, không ghi vào thanh ghi hay bộ nhớ.

4. Tối ưu hóa: Rẽ nhánh sớm tại ID

Để giảm Branch Penalty từ 2-3 chu kỳ xuống còn **1 chu kỳ**, Project thực hiện quyết định rẽ nhánh ngay tại tầng ID:

- Chuyển bộ so sánh về ID:** Sử dụng cổng XOR để kiểm tra 'Equal' ngay sau khi đọc thanh ghi.
- Forwarding đặc biệt:** Cần thêm logic 'Forwarding' đến ID để so sánh dữ liệu mới nhất (từ EX hoặc MEM hồi tiếp về).
- Logic Flush:** Nếu quyết định nhảy được đưa ra tại ID, chỉ cần xả lệnh đang ở IF (lệnh $PC + 4$).

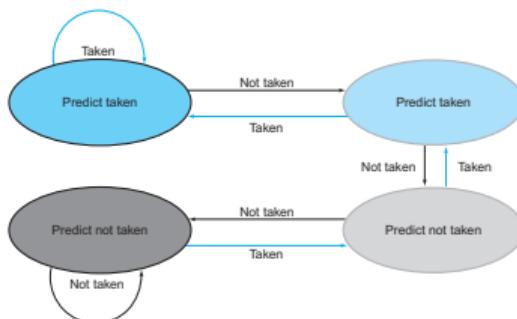
Kết quả: Tiết kiệm được 1 chu kỳ so với việc quyết định tại EX.

5. Mở rộng: Dự đoán Động & Delayed Branch

So sánh với các kỹ thuật nâng cao khác trong giáo trình:

Dynamic Branch Prediction

- ▶ Dùng **Branch History Table (BHT)** để lưu lịch sử (Taken/Not Taken).
- ▶ **Cơ chế 2-bit:** Chỉ thay đổi dự đoán nếu sai 2 lần liên tiếp.

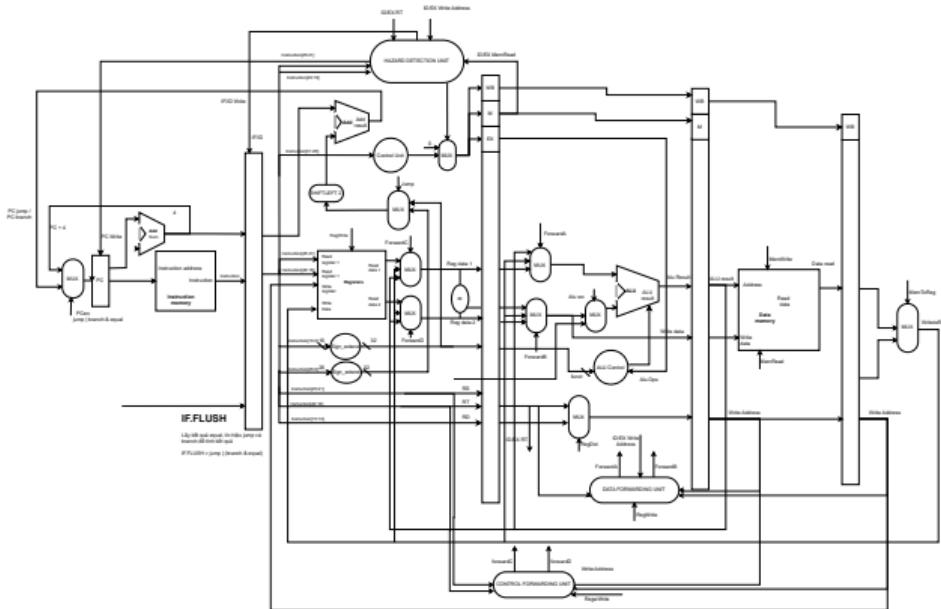


Hình: Sơ đồ dự đoán 2-bit

Delayed Branch

- ▶ **Branch Delay Slot:** Lệnh ngay sau rẽ nhánh luôn được thực thi (bất kể nhảy hay không).
- ▶ **Compiler:** Trình biên dịch phải sắp xếp lệnh hữu ích (hoặc NOP) vào slot này.
- ▶ *Project không dùng cách này do làm tăng độ phức tạp của trình biên dịch.*

6. Datapath hoàn chỉnh của Project



Hình: Datapath tổng quát: Forwarding, Hazard Detection và Early Branch

7. Implementation: Tính toán tại tầng ID

Chiến lược: Tính địa chỉ và điều kiện nhảy ngay tại module ID.v:

Code Verilog: Tính toán địa chỉ (Module PC_DECODE)

```
// Tính địa chỉ Branch (PC+4 + Offset*4)
assign branch_addr = pc_next + ({16{imm[15]}}, imm} << 2);

// Tính địa chỉ Jump ({PC[31:28], index, 2'b00})
assign jump_addr = { pc_next[31:28], instr[25:0], 2'b00 };

// Tín hiệu FLUSH kích hoạt khi: Jump OR (Branch AND Equal)
assign flush = jump | (reg_equal & branch);
```

→ Tín hiệu flush sẽ được gửi ngược về tầng IF để xóa lệnh kế tiếp.



8. Ngoại lệ: Cần Stall cho Control Hazard

Vấn đề: Branch after ALU Instruction

Nếu lệnh Branch cần dùng kết quả của lệnh ALU ngay trước đó (đang ở EX), Forwarding là không đủ (do tính toán Branch diễn ra ở ID).

Code Verilog: Hazard Detection Unit

```
// Nếu là Branch và lệnh trước đó ghi vào thanh ghi nguồn
if (branch && reg_write &&
((ID_EX_rd == IF_ID_rs) || (ID_EX_rd == IF_ID_rt)))
begin
    pc_stall = 1'b1;           // Giữ nguyên PC
    IF_ID_stall = 1'b1;        // Giữ nguyên IF/ID
    mux_control_hazard = 1'b1; // Chèn Bubble
end
```

9. Timeline so sánh Control Hazard

Ví dụ: beq \$t0, \$t1, LABEL

TRƯỜNG HỢP 1: TAKEN

($t0 == t1 \rightarrow$ Dự đoán sai, phải Flush)

CK	Trạng thái (IF & ID)
1	IF: Fetch beq
2	ID: Decode beq IF: Fetch Instr @ PC+4 \rightarrow Flush IF/ID, $PC \leftarrow Label$
3	ID: NOP (Lệnh @ PC+4 bị hủy) IF: Fetch Instr @ LABEL
4	ID: Decode Instr @ LABEL

Penalty = 1 Chu kỳ

TRƯỜNG HỢP 2: NOT TAKEN

($t0 \neq t1 \rightarrow$ Dự đoán đúng)

CK	Trạng thái (IF & ID)
1	IF: Fetch beq
2	ID: Decode beq IF: Fetch Instr @ PC+4 \rightarrow Giữ nguyên luồng lệnh
3	ID: Decode Instr @ PC+4 IF: Fetch Instr @ PC+8
4	EX: Execute Instr @ PC+4

Penalty = 0 Chu kỳ

Lưu ý: "Instr @ X" nghĩa là lệnh nằm tại địa chỉ bộ nhớ X.

10. Tổng kết Cơ chế Control Hazard

Đặc điểm	Trạng thái	Ghi chú kỹ thuật
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 Cycle	Khi dự đoán sai (Taken)
Dynamic Prediction	Không	Không dùng BHT/BTB
Delayed Branch	Không	Dùng cơ chế Flush hiện đại hơn
Xử lý đặc biệt	Stall	Khi gặp Data Hazard tại lệnh Branch

⇒ **Kết luận:** Hệ thống cân bằng giữa hiệu năng (Early Branch) và độ phức tạp phần cứng.

Nội dung trình bày I

1. Tổng quan dự án & Cơ sở lý thuyết
2. Thiết kế Datapath 5 tầng cơ bản
3. Xử lý Xung đột dữ liệu (Data Hazards)
4. Khối Điều khiển & Xung đột Rẽ nhánh (Control Unit & Control Hazards)
5. Kiểm thử & Demo

1. Chiến lược kiểm thử

Để kiểm chứng độ tin cậy của CPU, nhóm xây dựng kịch bản kiểm thử tự động tính toán số học phức tạp:

Bài toán: Tính tổng chuỗi số

Mục tiêu: Tính tổng $S = 1 + 2 + \dots + 10$.

- ▶ **Input:** $N = 10$.
- ▶ **Expected Output:** $S = 55$.
- ▶ **Thanh ghi sử dụng:**
 - ▶ \$s0: Biến chạy i .
 - ▶ \$s1: Biến tích lũy sum .
 - ▶ \$s2: Giới hạn $N = 10$.

Mục đích kiểm thử

Bài test này buộc CPU phải xử lý liên tục:

1. **Data Hazard:** Cộng dồn vào $\$s1$ liên tục.
2. **Control Hazard:** Lệnh nhảy J (Loop) và rẽ nhánh BEQ (Thoát loop).

2. Nạp chương trình vào Instruction Memory

Testbench tb_top_module.v nạp trực tiếp mã máy vào bộ nhớ:

Đoạn mã Assembly tương ứng

Loop:

```
ADD $s1, $s1, $s0 # sum += i  
BEQ $s0, $s2, DONE # if i==10 exit  
ADDI $s0, $s0, 1 # i++  
J LOOP # Quay lại
```

Done:

```
BEQ $s1, $s3, PASS # Check result
```

Testbench Setup

```
// Nạp lệnh vào I-MEM  
write_imem(4*4, 32'h02308820);  
// ADD $s1, $s1, $s0  
write_imem(7*4, 32'h08000004);  
// J LOOP (Target=4)
```

3. Kết quả Mô phỏng (Terminal Output)

Log trích xuất từ quá trình chạy thực tế trên ModelSim:

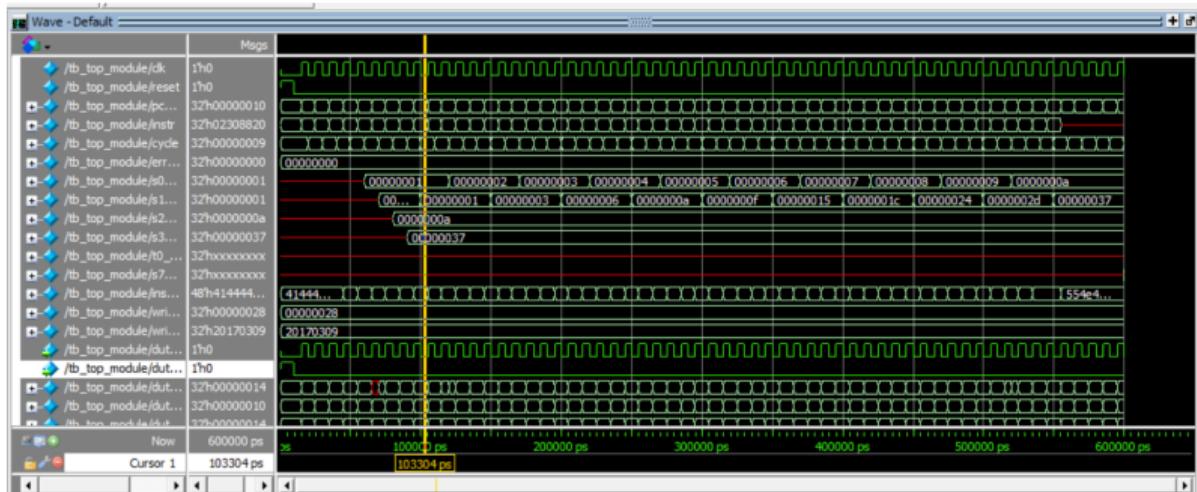
Console Log (Trích đoạn)

Cyc	PC	Inst	i(\$s0)	sum(\$s1)	Status
...					
50	00000014	BEQ	9	45	Checking Branch Condition...
51	00000018	ADDI	10	45	Executing...
52	00000020	BEQ	10	45	Checking Branch Condition...
53	00000024	ADDI	10	45	Executing...
54	00000028	ADDI	10	55	Executing...
55	0000002c	UNK	10	55	Executing...
...					

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

→ Chương trình chạy đúng 59 chu kỳ và ra kết quả chính xác $Sum = 55$.

4. Phân tích Dạng sóng (Waveform Analysis)



Phân tích dữ liệu (Data Flow):

- ▶ **Dòng s0 (Biến i):** Tăng tuần tự $1 \rightarrow 10$ ($0xA$).
- ▶ **Dòng s1 (Biến Sum):** Cộng dồn chính xác: $0 \rightarrow 1 \rightarrow 3 \dots \rightarrow 37_{hex} (55_{dec})$.

Hoạt động Pipeline:

- ▶ **Instr:** ADDI, BEQ, J thực thi gối đầu nhau, không bị ngắt quãng.
- ▶ **Hiệu năng:** 60 chu kỳ cho 10 vòng lặp \rightarrow Đạt CPI lý tưởng xấp xỉ 1.

5. Đánh giá Kết quả Thực thi

Dựa trên kết quả Waveform, nhóm xác nhận CPU hoạt động đúng thiết kế:

Kiểm chứng kết quả cuối cùng

- ▶ **Mong đợi (Expected):** $S = \frac{10 \times 11}{2} = 55$.
- ▶ **Thực tế (Actual):** Thanh ghi \$s1 đạt giá trị 32'h00000037.
- ▶ Quy đổi: $37_{16} = 3 \times 16 + 7 = 55_{10}$.

Chức năng	Trạng thái	Minh chứng trên Wave
Tính toán số học (ALU)	PASS	Dòng s1 cộng đúng
Rẽ nhánh (Branch/Jump)	PASS	Loop lặp lại đúng 10 lần
Xử lý Hazard	PASS	Không xuất hiện giá trị rác (X)

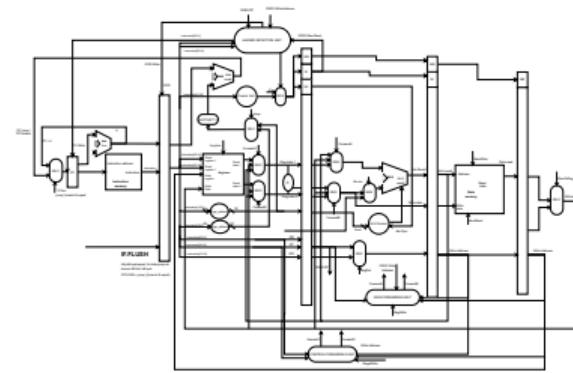
⇒ Hệ thống hoàn toàn đáp ứng yêu cầu của bài tập lớn.

6. Tổng kết sản phẩm

Dự án đã hoàn thành thiết kế trọn vẹn CPU MIPS 32-bit Pipelined:

Thành quả đạt được

1. **Kiến trúc:** 5 tầng chuẩn (IF, ID, EX, MEM, WB).
2. **Tối ưu hóa:**
 - ▶ *Early Branching:* Giảm Branch Penalty còn 1 chu kỳ.
 - ▶ *Full Forwarding:* Giải quyết triệt để Data Hazard.
3. **Tổ chức mã nguồn:**
 - ▶ Phân chia module rõ ràng (ALU_BIG_MODULE, HAZARD_DETECTION...).



Hình: Datapath hoàn thiện

7. Hướng phát triển

Dựa trên các phần nâng cao của Chapter 4 (Patterson & Hennessy):

1. Xử lý Ngoại lệ (Exceptions)

- ▶ Bổ sung thanh ghi **EPC** (Exception PC) và **Cause Register**.
- ▶ Cơ chế ngắt (Interrupt) để xử lý lệnh không hợp lệ hoặc tràn số học (Overflow).

2. Tính toán song song (Parallelism)

- ▶ **Deep Pipeline:** Chia nhỏ các tầng để tăng xung nhịp.
- ▶ **Multiple Issue:** Phát nhiều lệnh trong 1 chu kỳ (Superscalar).

XIN CẢM ƠN THẦY VÀ CÁC BẠN ĐÃ LẮNG NGHE!

Tài liệu tham khảo

- [1] David A. Patterson and John L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 5th Edition, Morgan Kaufmann, 2013.



HUST

THANK YOU !