

DẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
DẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH



KIẾN TRÚC MÁY TÍNH (CO2007)

Bài tập lớn

Đề tài: Hiện thực máy tính MIPS PIPELINE đơn giản

THÀNH PHỐ HỒ CHÍ MINH, 2023



Mục lục

1	Tổng quan	2
1.1	Yêu cầu chung	2
1.2	Yêu cầu kỹ thuật	2
2	Quy trình thiết kế	3
2.1	Xác định tập lệnh của kiến trúc	3
2.2	Xác định các stage	4
2.2.1	IF stage	4
2.2.2	ID stage	4
2.2.3	EXE stage	4
2.2.4	MEM stage	4
2.2.5	WB stage	4
2.3	Xác định các khối cơ bản	4
2.3.1	Khối PC Register	4
2.3.2	Khối Instruction memory	4
2.3.3	Khối REG (Register file)	5
2.3.4	Khối control	5
2.3.5	Khối ALU	6
2.3.6	Khối ALU control	6
2.3.7	Khối ALU status	7
2.3.8	Khối Data memory	7
2.3.9	Các khối hiển thị kết quả	7
2.3.10	Các khối khác	8
2.4	Xác định datapath của các nhóm lệnh	8
2.4.1	Xây dựng kiến trúc ban đầu	8
2.4.2	Bổ sung khối forward unit	11
2.4.3	Hiện thực phép nhân, chia thông qua thanh ghi high/low	13
2.4.4	Bổ sung khối kiểm soát hazard	15
2.4.5	Bổ sung khối kiểm soát exception	17
2.5	Hoàn thiện kiến trúc	19
3	Kết quả	20
3.1	Chương trình 1	21
3.1.1	Kết quả mô phỏng	22
3.1.2	Kết quả hiện thực trên board	25
3.2	Chương trình 2	26
3.3	Vấn đề lưu dữ liệu vào thanh ghi	31
3.3.1	Nguyên nhân	31
3.3.2	Giải pháp	31
4	Kết luận	32
4.1	Những hạn chế của kiến trúc	32
4.2	Thuận lợi và khó khăn	32
4.2.1	Thuận lợi	32
4.2.2	Khó khăn	32
4.3	Phương hướng hoàn thiện đề tài	32

1 Tổng quan

1.1 Yêu cầu chung

- Tìm hiểu, phân tích, thiết kế bộ xử lý pipeline đơn giản.
- Làm quen board Arty Z7-20.
- Làm quen ngôn ngữ đặc tả phần cứng Verilog.

1.2 Yêu cầu kỹ thuật

Cùng với các yêu cầu của giảng viên, nhóm đặt ra một số yêu cầu bổ sung nhằm giới hạn phạm vi, đi sâu vào nghiên cứu các tính năng của kiến trúc.

- Hoàn thiện kiến trúc thỏa mãn các tính năng theo yêu cầu của đề bài, là kiến trúc pipeline 5-stage.
- Hiện thực tính năng kiểm soát hazard dựa trên kiến trúc được xây dựng ở textbook “Computer Organization and Design”[1].
- Điều chỉnh một số thành phần nhằm phù hợp với tập lệnh do nhóm đề xuất.

Quá trình thực hiện của nhóm được lưu trữ tại [Github](#).

2 Quy trình thiết kế

Trong phần này, nhóm mô tả quá trình thiết kế kiến trúc dựa trên kiến trúc được học từ textbook [1]. Song song với xây dựng datapath theo kiến trúc trên, nhóm sẽ chỉ rõ các điểm khác biệt trong thiết kế của nhóm.

2.1 Xác định tập lệnh của kiến trúc

Dựa theo yêu cầu của đề tài, tập lệnh của textbook và các lệnh thường xuyên được sử dụng xuyên suốt môn học, nhóm xây dựng tập lệnh cho kiến trúc như sau:

STT	Lệnh	Kiểu lệnh	STT	Lệnh	Kiểu lệnh
1	nop		15	addi	I
2	add	R	16	andi	I
3	sub	R	17	ori	I
4	mult	R	18	xori	I
5	div	R	19	lb	I
6	mfhi	R	20	lh	I
7	mflo	R	21	lw	I
8	and	R	22	sb	I
9	or	R	23	sh	I
10	xor	R	24	sw	I
11	nor	R	25	beq	I
12	slt	R	26	j	J
13	sll	R			
14	srl	R			

Bảng 1: Tập lệnh của kiến trúc

Một số điểm khác biệt quan trọng so với tập lệnh đã được học:

- Bộ ALU được mở rộng nhằm thực hiện các phép shift logical (sll, srl), phép bitwise xor (xor) và phép nhân (mult) chia (div) đại số.
- Việc hiện thực nhân, chia cần sự xuất hiện của cặp thanh ghi high/low để lưu kết quả, do đó cần bổ sung các lệnh mfhi, mflo.
- Bổ sung các lệnh đại số / logic thuộc kiểu lệnh I (addi, andi, ori, xori).
- Bổ sung các lệnh load / store cho byte và half-word (lb, lh, sb, sh).

Sự bổ sung trên sẽ làm kiến trúc có những thay đổi nhất định và sẽ được nhóm trình bày cụ thể ở các phần tiếp theo.

2.2 Xác định các stage

Các stage mà nhóm thiết kế được thể hiện qua sơ đồ sau:



Hình 1: Các stage trong kiến trúc

2.2.1 IF stage

Tại cạnh xuống của clock, đọc lệnh tại địa chỉ tương ứng với PC hiện tại, đồng thời cập nhật PC mới.

2.2.2 ID stage

Tại cạnh lên của clock, đổ dữ liệu qua thanh ghi IF/ID. Tại cạnh xuống, truy xuất dữ liệu từ **Register file**.

2.2.3 EXE stage

Tại cạnh lên của clock, đổ dữ liệu qua thanh ghi ID/EXE, thực hiện các phép tính toán số học/logic. Tại cạnh xuống, truy xuất dữ liệu từ thanh ghi **high, low** (các thanh ghi hỗ trợ hiện thực các phép nhân, chia), cập nhật tín hiệu exception (nếu có xảy ra exception).

2.2.4 MEM stage

Tại cạnh lên của clock, ghi dữ liệu vào thanh ghi **high, low**, đổ dữ liệu qua thanh ghi EXE/MEM. Tại cạnh xuống của clock, đọc dữ liệu từ data memory, dữ liệu này (hoặc kết quả của bộ ALU) sẽ được đưa thẳng về **Register file** mà không đợi qua thanh ghi MEM/WB mới đưa về như kiến trúc được học để đảm bảo các tác vụ của kiến trúc được thực hiện đầy đủ trong 5 stage.

2.2.5 WB stage

Tại cạnh lên của clock, ghi dữ liệu (nếu có) vào data memory, ghi dữ liệu (nếu có) vào **Register file** và đổ dữ liệu qua thanh ghi MEM/WB.

2.3 Xác định các khối cơ bản

2.3.1 Khối PC Register

Cập nhật giá trị PC mới (từ bộ chọn PC) tại cạnh xuống của clock.

2.3.2 Khối Instruction memory

Bộ nhớ lệnh chứa mã máy chương trình, nhóm hiện thực bằng thư viện IP Block Memory Generator (8.4).

2.3.3 Khối REG (Register file)

Diều khiển việc đọc, ghi của 32 thanh ghi trong kiến trúc MIPS.

2.3.4 Khối control

Dựa vào opcode của các lệnh để đưa ra các tín hiệu điều khiển như sau:

Tên	Bit	Ý nghĩa
Jump	13	Xác định lệnh hiện tại là lệnh Jump
Branch	12	Xác định lệnh hiện tại là lệnh Branch
MemRead	11:10	Xác định kiểu lệnh load
MemWrite	9:8	Xác định kiểu lệnh store
ALUOp	7:5	Xác định lệnh tính toán cụ thể, input của bộ ALU control
ALUSrc	4	Xác định toán hạng đưa vào ALU là từ thanh ghi Rt hay là số Immediate
RegDst	3	Xác định thanh ghi đích ghi vào Register file là Rt hay Rd
Exception	2	Tích cực khi không xác định được lệnh hiện tại
Mem2Reg	1	Xác định dữ liệu ghi vào Register file (nếu có).
RegWrite	0	Xác định lệnh hiện tại có ghi vào Register file không.

Bảng 2: Các tín hiệu control

Sự thay đổi của các tín hiệu control so với kiến trúc đã được học:

- Tín hiệu **MemRead**: tăng lên thành 2 bit nhằm kiểm soát các lệnh **lb**, **lh**.
- Tín hiệu **MemWrite**: tăng lên thành 2 bit nhằm kiểm soát các lệnh **sb**, **sh**.
- Tín hiệu **ALUOp**: tăng lên thành 3 bit nhằm phù hợp với tập lệnh đã được bổ sung.

Bên cạnh đó, các tín hiệu điều khiển được sử dụng trong cùng stage được nhóm lại gần nhau.

Từ các tín hiệu nêu trên, nhóm xác định cụ thể các tín hiệu được khối control sinh ra cho từng lệnh, cụ thể:

Lệnh	[13]	[12]	[11:10]	[9:8]	[7:5]	[4]	[3]	[2]	[1]	[0]
nop	0	0	00	00	000	0	0	0	0	0
R-type	0	0	00	00	011	0	1	0	1	1
addi	0	0	00	00	111	1	0	0	1	1
andi	0	0	00	00	100	1	0	0	1	1
ori	0	0	00	00	101	1	0	0	1	1
xori	0	0	00	00	110	1	0	0	1	1
lb	0	0	01	00	111	1	0	0	0	1
lh	0	0	10	00	111	1	0	0	0	1
lw	0	0	11	00	111	1	0	0	0	1
sb	0	0	00	01	111	1	0	0	0	0
sh	0	0	00	10	111	1	0	0	0	0
sw	0	0	00	11	111	1	0	0	0	0
beq	0	1	00	00	000	1	0	0	0	0
j	1	0	00	00	000	0	0	0	0	0

Bảng 3: Tín hiệu control cho từng lệnh

2.3.5 Khối ALU

Thực hiện các tính toán số học (cộng, nhân, chia) và logic (and, or, xor, nor, shift).

STT	Lệnh	Phép toán	STT	Lệnh	Phép toán
1	nop	Không thực hiện	15	addi	Cộng
2	add	Cộng	16	andi	Bitwise and
3	sub	Cộng	17	ori	Bitwise or
4	mult	Nhân	18	xori	Bitwise xor
5	div	Chia	19	lb	Cộng
6	mfhi	Không thực hiện	20	lh	Cộng
7	mflo	Không thực hiện	21	lw	Cộng
8	and	Bitwise and	22	sb	Cộng
9	or	Bitwise or	23	sh	Cộng
10	xor	Bitwise xor	24	sw	Cộng
11	nor	Bitwise nor	25	beq	Không thực hiện
12	slt	Compare	26	j	Không thực hiện
13	sll	Shift logical			
14	srl	Shift logical			

Bảng 4: Phép toán khối ALU thực hiện

Một số lệnh bộ ALU không thực hiện tính toán:

- **nop**: No-operation, bộ ALU không thực hiện tính toán.
- **mfhi**: Đọc kết quả từ thanh ghi high (thay cho việc tính toán bằng bộ ALU).
- **mflo**: Đọc kết quả từ thanh ghi low (thay cho việc tính toán bằng bộ ALU).
- **beq**: Việc tính toán địa chỉ rẽ nhánh tới được thực hiện ở ID stage nhằm phục vụ cho bộ so sánh sớm.
- **j**: Việc tính toán địa chỉ nhảy đến được thực hiện ở ID stage.

2.3.6 Khối ALU control

Điều khiển khối ALU thực hiện đúng phép toán cần thực thi của lệnh hiện tại.

Bit	Ý nghĩa
4	Operand_0_invert (nếu thực hiện cơ chế đảo) / Tín hiệu ALU_control
3	Operand_1_invert (nếu thực hiện cơ chế đảo) / Tín hiệu ALU_control
2	Xác định thực hiện / không thực hiện cơ chế đảo
1	Tín hiệu ALU_control
0	Tín hiệu ALU_control

Bảng 5: Các tín hiệu ALU_control

Từ các phép toán được thực hiện bởi khối ALU, có thể thấy một số lệnh sử dụng chung phép toán nhưng các toán hạng có thể bị đảo. Nhằm giảm số phép toán cần xét, nhóm sử dụng 1 bit trong **ALU_control** giúp xác định có cần thực hiện cơ chế đảo neu trên hay không.

2.3.7 Khối ALU status

Lưu các trạng thái của phép tính thực thi tại khối ALU.

Tên	Bit	Ý nghĩa	Exception
Zero	7	Kết quả của phép tính là số zero	
Overflow	6	Xảy ra tràn số	x
Carry	5	Kết quả có nhớ	
Negative	4	Kết quả là số âm	
Invalid_address	3	Địa chỉ word, halfword không aligned	x
Div_zero	2	Chia cho 0	x
Undefined	1:0	Không sử dụng	

Bảng 6: Các trạng thái của phép tính tại khối ALU

2.3.8 Khối Data memory

Vùng nhớ chứa 64 x 32 words dữ liệu, nhóm hiện thực bằng thư viện IP Block Memory Generator (8.4).

2.3.9 Các khối hiển thị kết quả

- Khối system_control: lựa chọn 32 bit dữ liệu đưa ra LCD dựa trên tín hiệu điều khiển SYS_output_sel.

SYS_output_sel	Output
0000	IMEM instruction
0001	Giá trị của thanh ghi Rs
0010	Giá trị của thanh ghi Rt
0011	Địa chỉ PC chương trình rẽ nhánh nếu xảy ra hazard (id_calculated_pc)
0100	Toán hạng thứ nhất của khối ALU
0101	Toán hạng thứ hai của khối ALU
0110	Kết quả của khối ALU
0111	Kết quả đọc từ thanh ghi high, low (nếu có)
1000	Kết quả của EXE stage
1001	Dữ liệu ghi vào data memory (nếu có)
1010	ALU status
1011	Dữ liệu đọc từ data memory (nếu có)
1100	Các tín hiệu control
1101	Các tín hiệu ALU control
1110	Địa chỉ của lệnh kế tiếp (PC + 4)
1111	EPC

Bảng 7: Dữ liệu được hiển thị ứng với các giá trị SYS_output_sel

- Khối lcd_control: điều khiển các chân của LCD để hiển thị kết quả.
- Khối debouncer: chống rung công tắc khi tạo xung clock.

2.3.10 Các khối khác

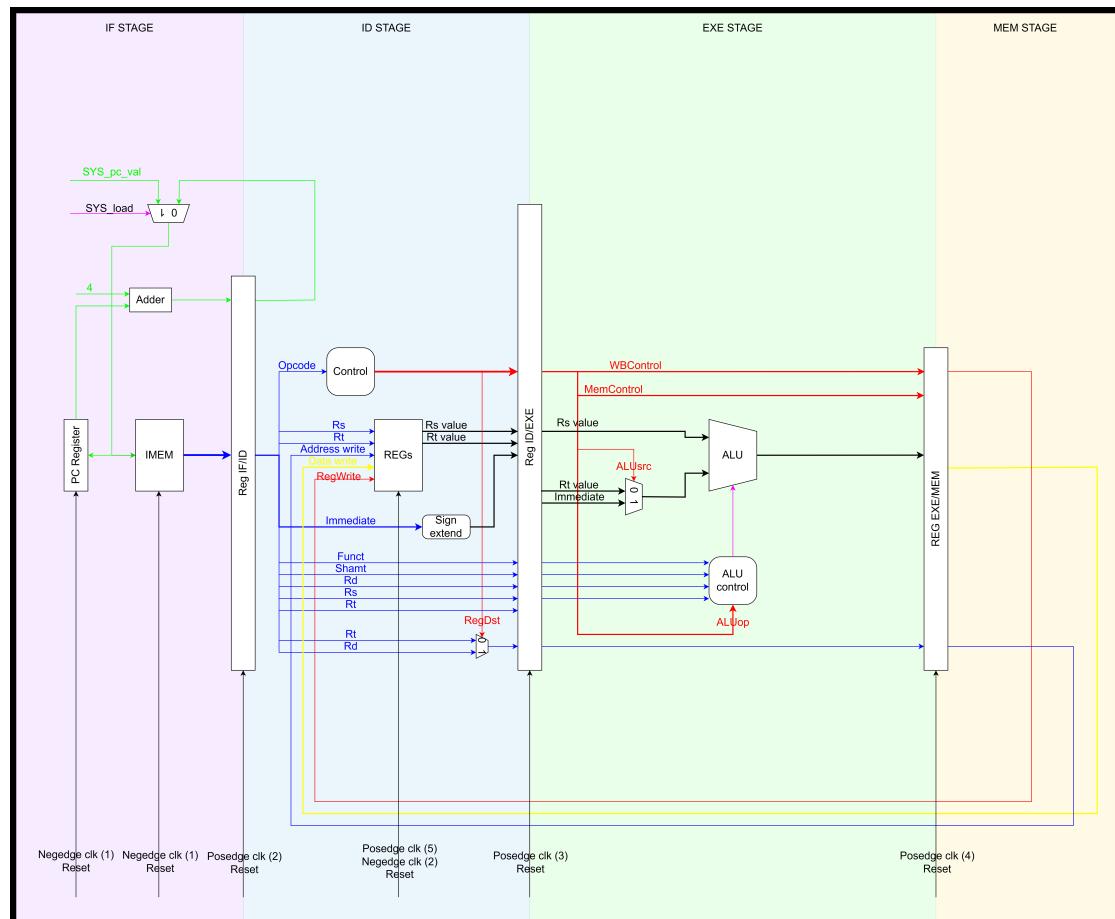
- Các thanh ghi trung gian, dùng để lưu kết quả của mỗi stage.
- Các khối forward unit, khối kiểm soát hazard, exception, thanh ghi **high**, **low**, sẽ được đặc tả rõ khi thiết kế datapath cho các lệnh có liên quan.

2.4 Xác định datapath của các nhóm lệnh

2.4.1 Xây dựng kiến trúc ban đầu

Trước tiên, nhóm thiết kế kiến trúc ban đầu đơn giản để thực hiện các lệnh cơ bản như tính toán số học/logic, load, store mà chưa xử lý những hazard có thể xảy ra.

- a) Các lệnh tính toán số học/logic

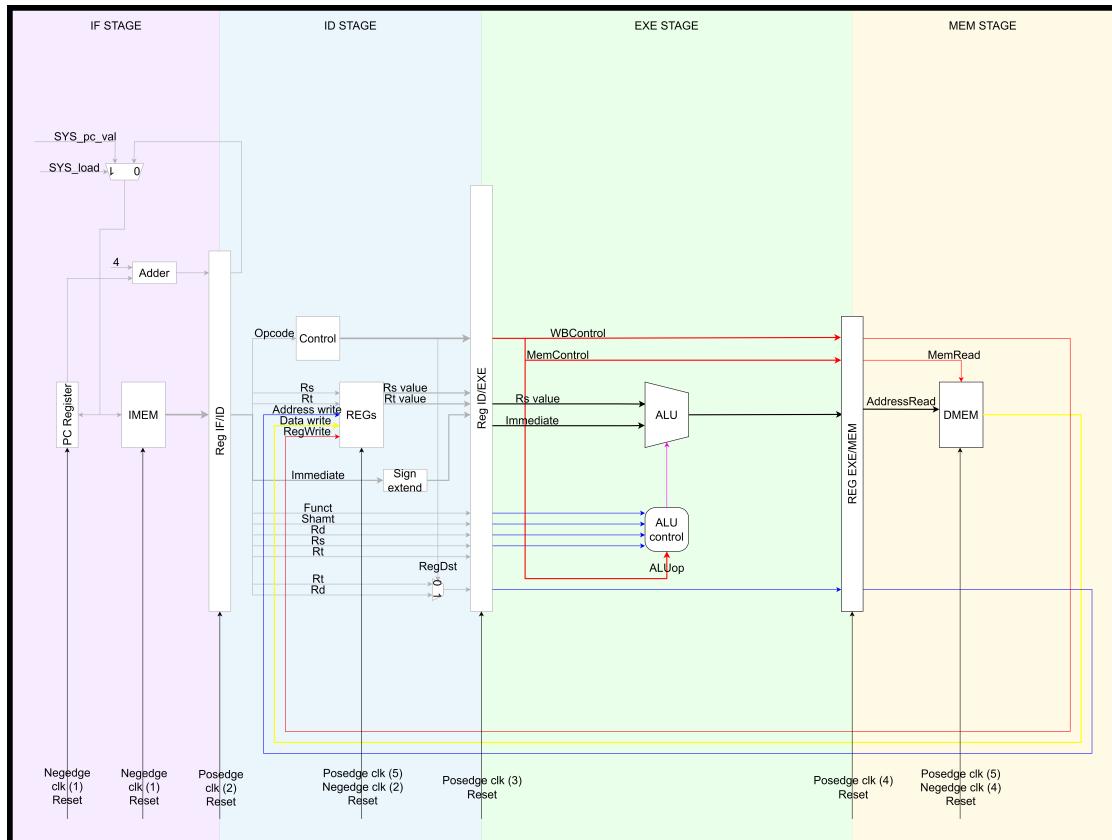


Hình 2: Datapath đơn giản cho các lệnh tính toán số học/logic

Nhóm thiết kế datapath của các lệnh này tương tự với datapath được học, ngoại trừ việc đọc lệnh mới tại cạnh xuống và truyền các tín hiệu hỗ trợ việc ghi vào **Register file** ngay tại MEM stage để đảm bảo kiến trúc có 5 stage như đã nêu. Bên cạnh đó, việc chọn thanh

ghi cần ghi cũng được thực hiện ở ID stage nhằm phù hợp với tập lệnh do nhóm đề xuất và việc kiểm soát hazard sau này.

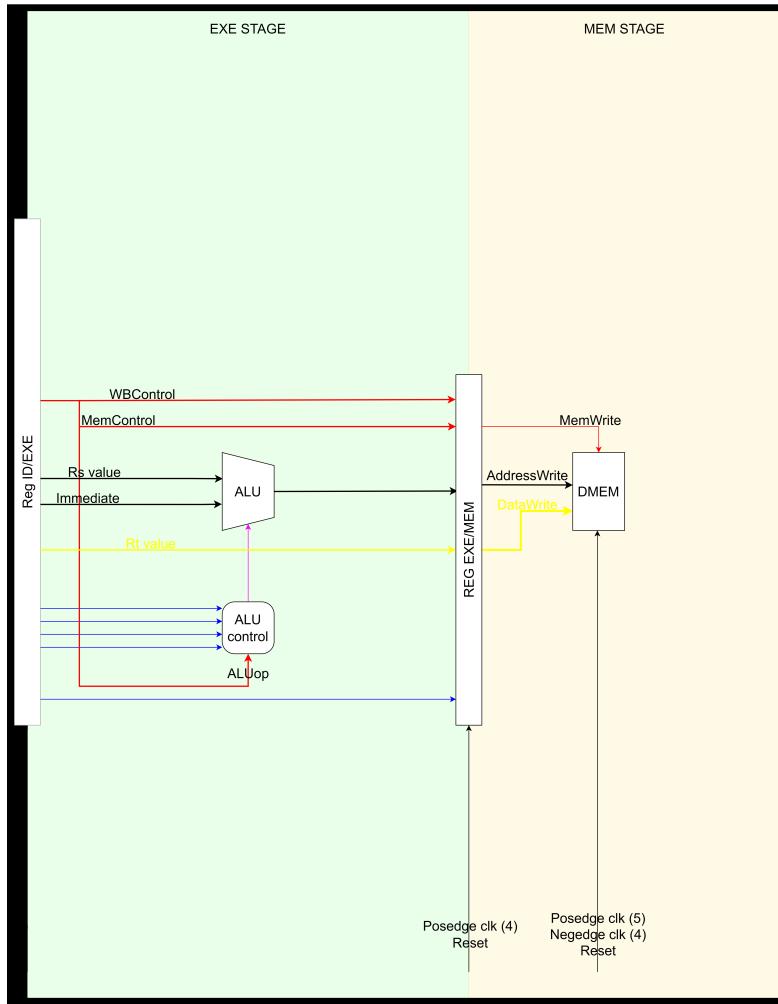
b) Các lệnh load



Hình 3: Datapath đơn giản cho các lệnh load

Nhóm thiết kế datapath của các lệnh này tương tự với datapath được học, ngoại trừ một số điểm khác đề cập ở phần a.

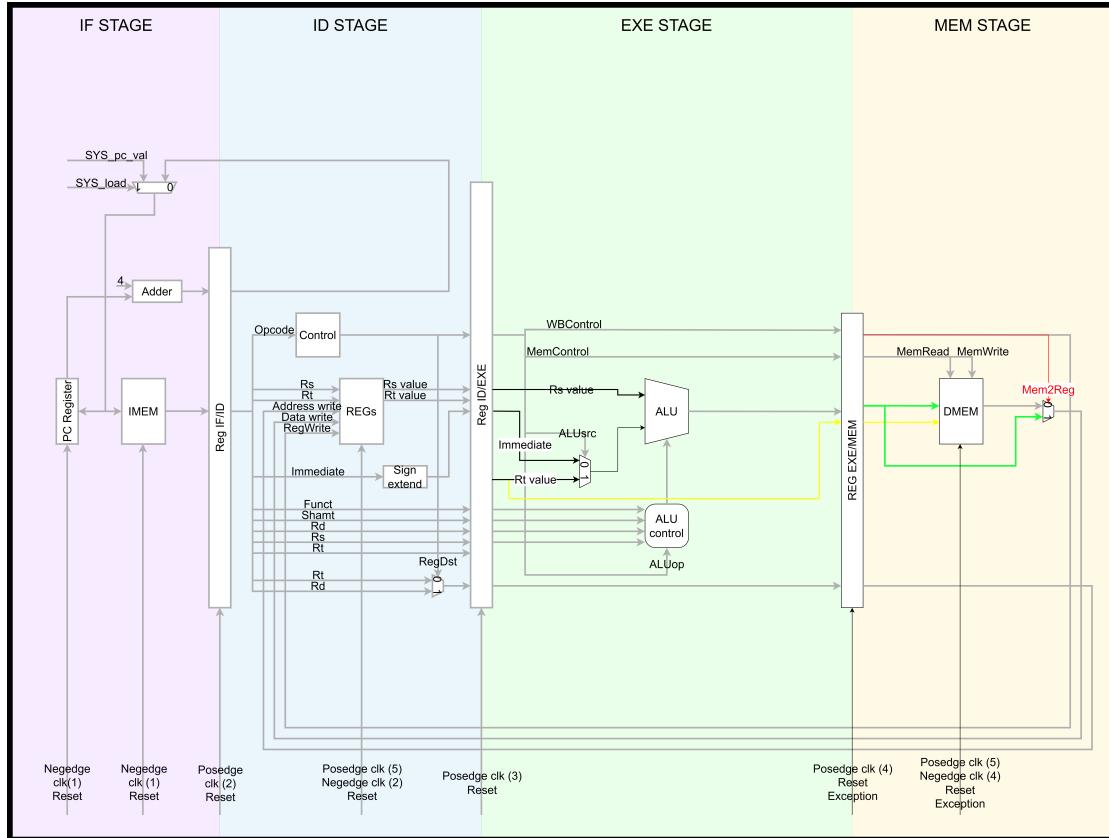
c) Các lệnh store



Hình 4: Datapath đơn giản cho các lệnh store

Nhóm thiết kế datapath của các lệnh này tương tự với datapath được học, ngoại trừ một số điểm khác đê cập ở phần a, ở đây nhóm chỉ thể hiện những phần datapath khác với datapath của 2 nhóm lệnh trước.

Tổng hợp 3 datapath trên, nhóm thiết kế được datapath tổng quát đơn giản như sau:



Hình 5: Datapath đơn giản cho các lệnh tính toán, lệnh load và store

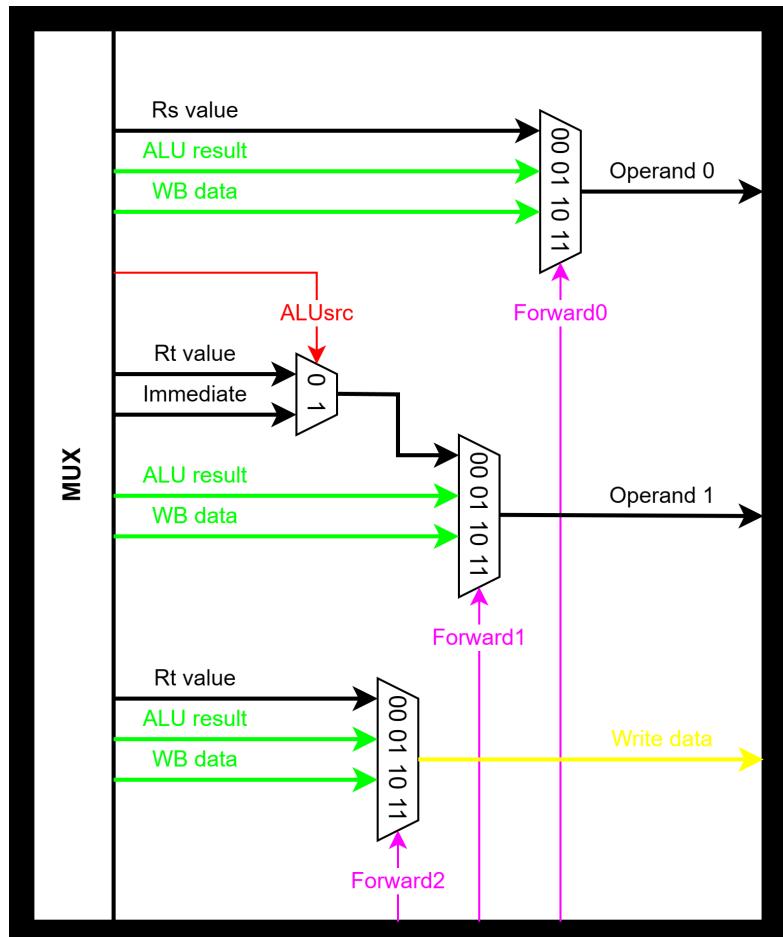
2.4.2 Bổ sung khối forward unit

Với khối forwarding unit, nhóm giải quyết một số data hazard bằng cách lấy dữ liệu cần thiết cho tính toán ngay khi dữ liệu đó sẵn sàng mà không cần đợi tới khi nó được lưu vào thanh ghi. Nếu thanh ghi lưu kết quả của 2 lệnh liên trước (nếu có) trùng với các thanh ghi cần truy xuất dữ liệu của lệnh hiện tại (trong báo cáo này, khi đề cập tới lệnh hiện tại, nhóm đề cập tới lệnh đang được thực thi tại EXE stage), nhóm thay thế dữ liệu đọc từ **Register file** bằng dữ liệu phù hợp. Tuy nhiên, do có nhiều format lệnh khác nhau, cần xác định các thanh ghi cần truy xuất dữ liệu, bên cạnh đó cũng cần có cơ chế để nhận biết 2 lệnh liền trước có ghi vào các thanh ghi không. Vì vậy, nhóm sử dụng thêm một số tín hiệu điều khiển:

- RegDst: Xác định lệnh hiện tại là lệnh kiểu R hay kiểu I. Nếu là lệnh kiểu R, cần truy xuất dữ liệu từ cả thanh ghi **Rs** và **Rt**, nếu là lệnh kiểu I, chỉ truy xuất dữ liệu từ thanh ghi **Rt**.
- MemWrite: Xác định lệnh hiện tại có phải là lệnh store hay không. Nếu có, ngoài việc phải truy xuất thanh ghi **Rs** để tính địa chỉ ghi vào data memory, còn cần phải truy xuất thanh ghi **Rt** để lấy dữ liệu ghi vào địa chỉ đó.

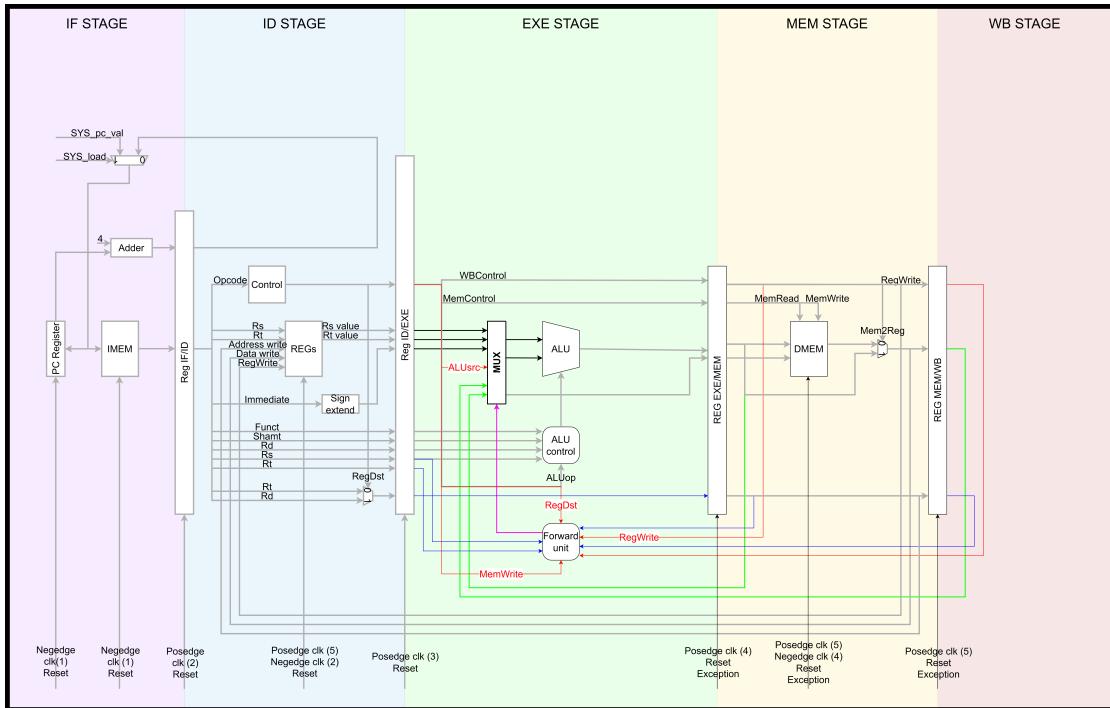
- Các tín hiệu RegWrite: Xác định 2 lệnh liền trước có ghi vào các thanh ghi hay không, nếu có mới thực hiện forward.

Từ các tín hiệu đó, các thanh ghi phù hợp được so sánh và cho ra các tín hiệu điều khiển bộ chọn sau để lựa chọn dữ liệu cho các toán hạng của khối ALU cũng như dữ liệu ghi vào data memory (đối với lệnh store).



Hình 6: Bộ chọn dữ liệu forward

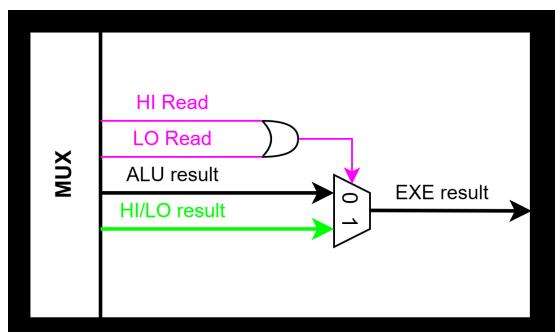
Như vậy, nhóm thiết kế được datapath tích hợp khối forward unit như sau:



Hình 7: Datapath tích hợp khối forward unit

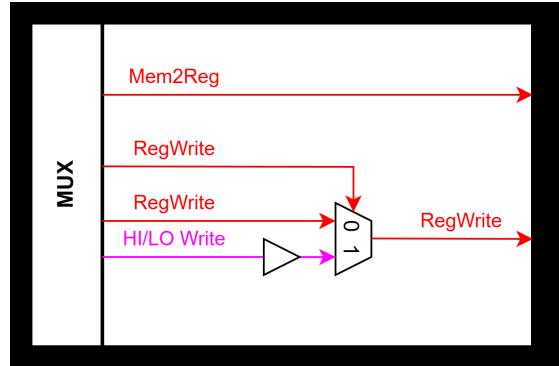
2.4.3 Hiện thực phép nhân, chia thông qua thanh ghi high/low

Khi hiện thực phép nhân 2 số 32 bit, kết quả là một số 64 bit nên cần phải có 1 thanh ghi lưu 32 bit cao và một thanh ghi lưu 32 bit thấp của tích. Tương tự, khi thực hiện phép chia 2 số 32 bit, cũng cần có một thanh ghi lưu 32 bit thương số và một thanh ghi lưu 32 bit số dư. Do vậy, nhóm bổ sung vào kiến trúc 2 thanh ghi **high** và **low** để hỗ trợ thực hiện phép nhân, chia. Ngoài ra, nhóm cũng hiện thực các lệnh **mfhi** và **mflo** để lưu giá trị của các thanh ghi **high** và **low** vào các thanh ghi trong **Register file**. Các tín hiệu điều khiển để đọc 2 thanh ghi này còn được sử dụng để xác định dữ liệu đưa về **Register file**:



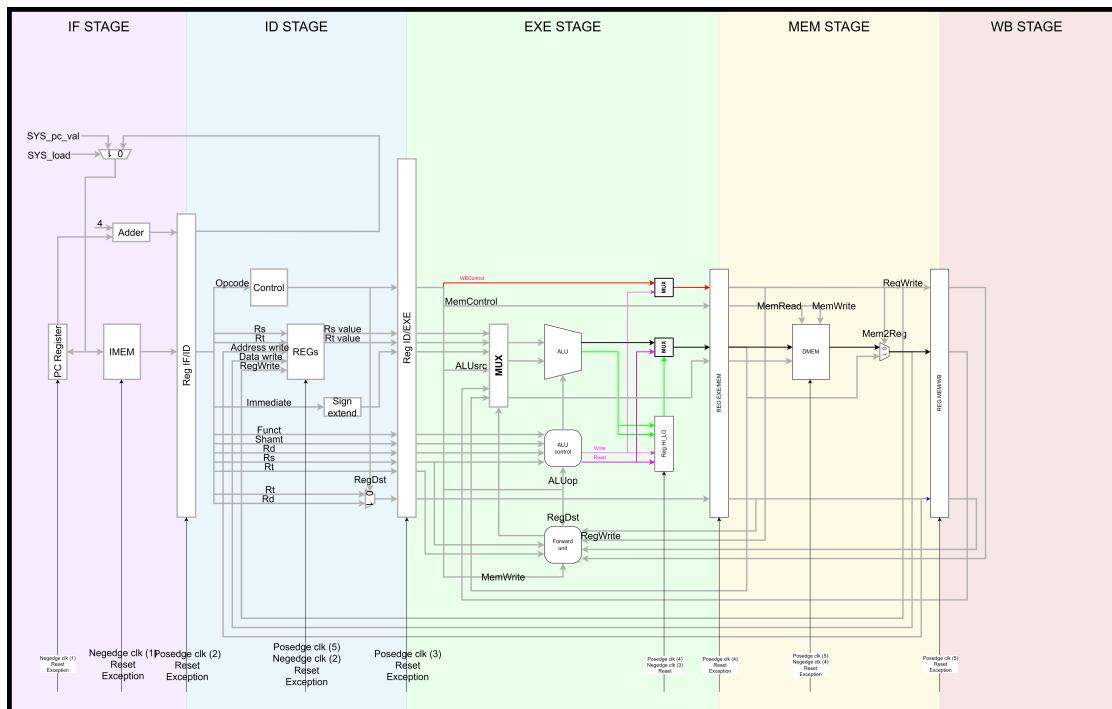
Hình 8: Bộ chọn kết quả của EXE stage

Mặt khác, các lệnh **div** và **mult** trong kiến trúc MIPS có opcode giống với các lệnh kiểu R, qua bộ control sẽ đưa tín hiệu điều khiển **RegWrite** lên 1, từ đó có thể ghi giá trị không mong muốn vào **Register File**. Vì vậy nhóm hiện thực thêm một bộ chọn để vô hiệu hóa tín hiệu điều khiển này như sau:



Hình 9: Bộ điều khiển tín hiệu RegWrite

Tích hợp 2 bộ điều khiển trên, nhóm thiết kế được datapath cho các lệnh nhân, chia:



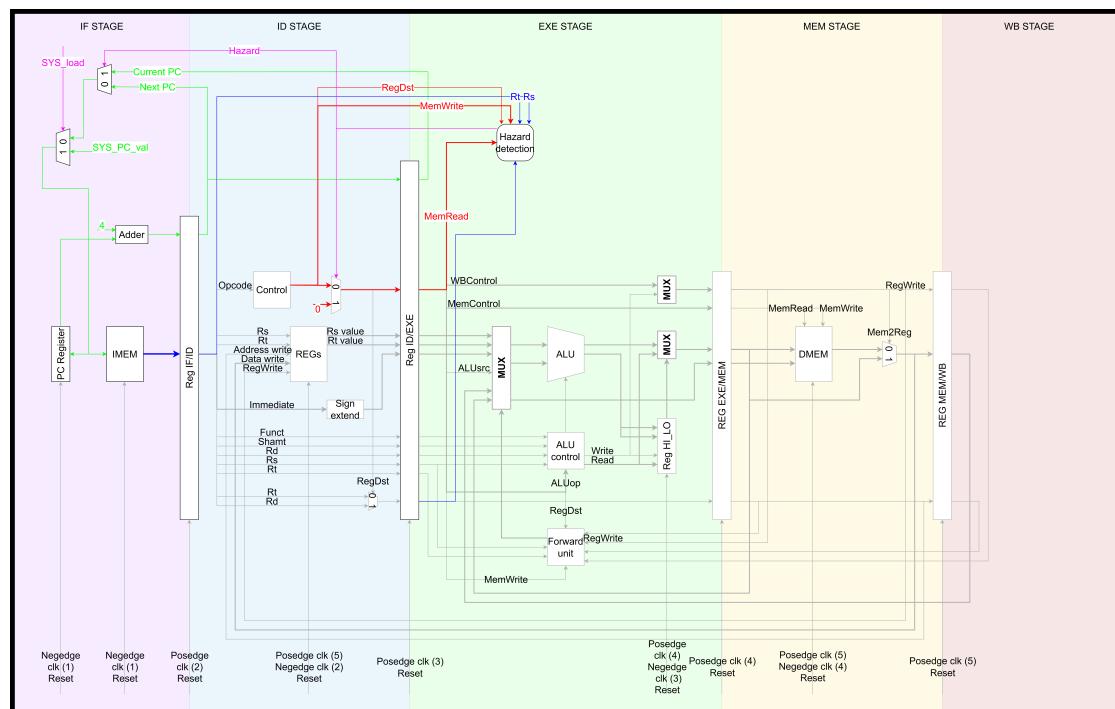
Hình 10: Datapath cho các lệnh nhân, chia

2.4.4 Bổ sung khối kiểm soát hazard

Với kiến trúc tập lệnh và sự điều khiển của khối forward unit, tất cả các lệnh số học, logic thông thường đều có thể thực hiện mà không xảy ra hazard. Tuy nhiên, data hazard vẫn có thể xảy ra khi một lệnh load lưu dữ liệu vào một thanh ghi mà lệnh liền sau nó cần truy xuất. Bên cạnh đó, hiện thực lệnh branch, jump cũng đòi hỏi phải xử lý các control hazard. Vì vậy, nhóm bổ sung khối kiểm soát hazard vào kiến trúc.

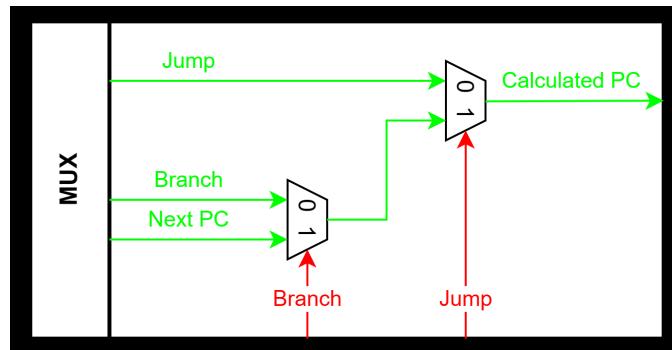
Để xử lý các hazard, nhóm sử dụng phương pháp chèn stall bằng cách đưa các tín hiệu control về 0 để vô hiệu hóa những lệnh chưa đủ điều kiện để thực thi.

Trước tiên, đối với data hazard từ các lệnh load, cần sử dụng tín hiệu **MemWrite** để xác định lệnh hiện tại có phải lệnh load hay không. Bên cạnh đó, nhóm đưa vào khối này thanh ghi **Rs**, **Rt** của lệnh tiếp theo so sánh với thanh ghi được load dữ liệu vào để xác định data hazard. Tương tự với khối forward unit, cần phải có thêm các tín hiệu điều khiển **RegDst**, **MemWrite** để xác định các thanh ghi cần truy xuất dữ liệu của lệnh tiếp theo. Nếu data hazard được phát hiện, khối này sinh ra một tín hiệu đặt các tín hiệu control về 0, vô hiệu hóa lệnh tiếp theo (đang ở ID stage), đồng thời điều khiển bộ chọn PC để đưa địa chỉ của lệnh đó về lại IF stage mà không tăng thêm 4. Datapath nêu trên được tổng hợp qua sơ đồ sau:



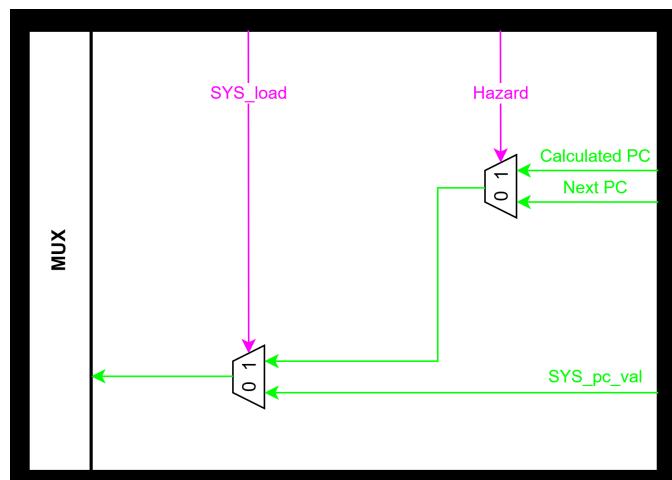
Hình 11: Datapath tích hợp kiểm soát data hazard

Bên cạnh data hazard, cần kiểm soát các control hazard để hiện thực các lệnh branch và jump. Trước tiên, nhóm sử dụng các tín hiệu control **Jump**, **Branch** để xác định địa chỉ của lệnh tiếp theo (sau lệnh stall nếu có) nếu chương trình rẽ nhánh:



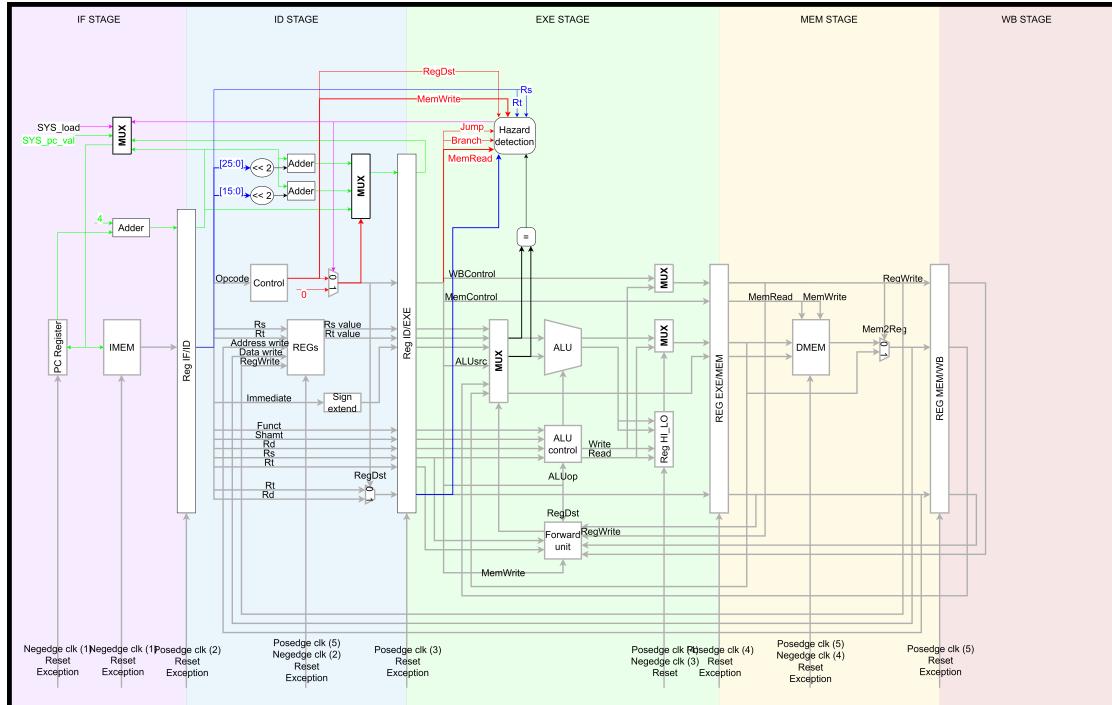
Hình 12: Bộ chọn xác định địa chỉ của lệnh tiếp theo

Đối với lệnh branch, nhóm áp dụng nguyên tắc tiên đoán tĩnh, dự đoán chương trình sẽ không rẽ nhánh, đồng thời sử dụng bộ so sánh sớm tại EXE stage để xác định tính chính xác của dự đoán. Nếu dự đoán sai (giá trị của 2 thanh ghi **Rs**, **Rt** bằng nhau), khối kiểm soát hazard sinh ra một tín hiệu dựa các tín hiệu control về 0 để vô hiệu hóa lệnh tiếp theo (đang ở ID stage) cũng như điều khiển bộ chọn PC để lựa chọn địa chỉ rẽ nhánh.



Hình 13: Bộ chọn PC khi tích hợp khối kiểm soát control hazard

Tương tự, khi gặp lệnh jump, lệnh tiếp theo cũng vẫn được đọc nhưng sẽ bị vô hiệu hóa bởi các tín hiệu control và sau một stall thì lệnh tại địa chỉ jump sẽ được đọc. Tổng kết lại, nhóm thiết kế được data path của khối kiểm soát hazard như sau:



Hình 14: Datapath khối kiểm soát hazard

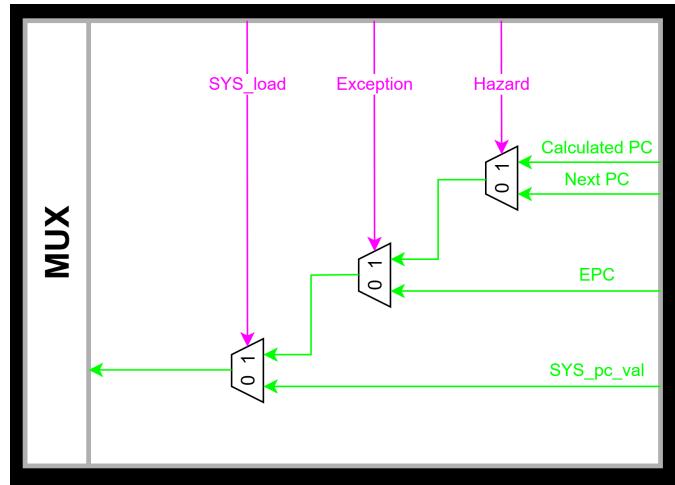
Ngoài ra, nếu bộ so sánh sớm được đặt ở ID stage, dữ liệu được so sánh sẽ là dữ liệu cũ đã được lưu trữ trong 2 thanh ghi Rs, Rt. Dữ liệu trên có thể đã bị thay đổi bởi các lệnh liền trước nó. Vì vậy, nhóm dịch chuyển sang EXE stage nhằm lấy dữ liệu mới nhất được bộ forward truyền về.

2.4.5 Bổ sung khối kiểm soát exception

Trong kiến trúc này, nhóm xử lý các nhóm ngoại lệ và sinh ra tín hiệu exception tương ứng tại các khối như sau:

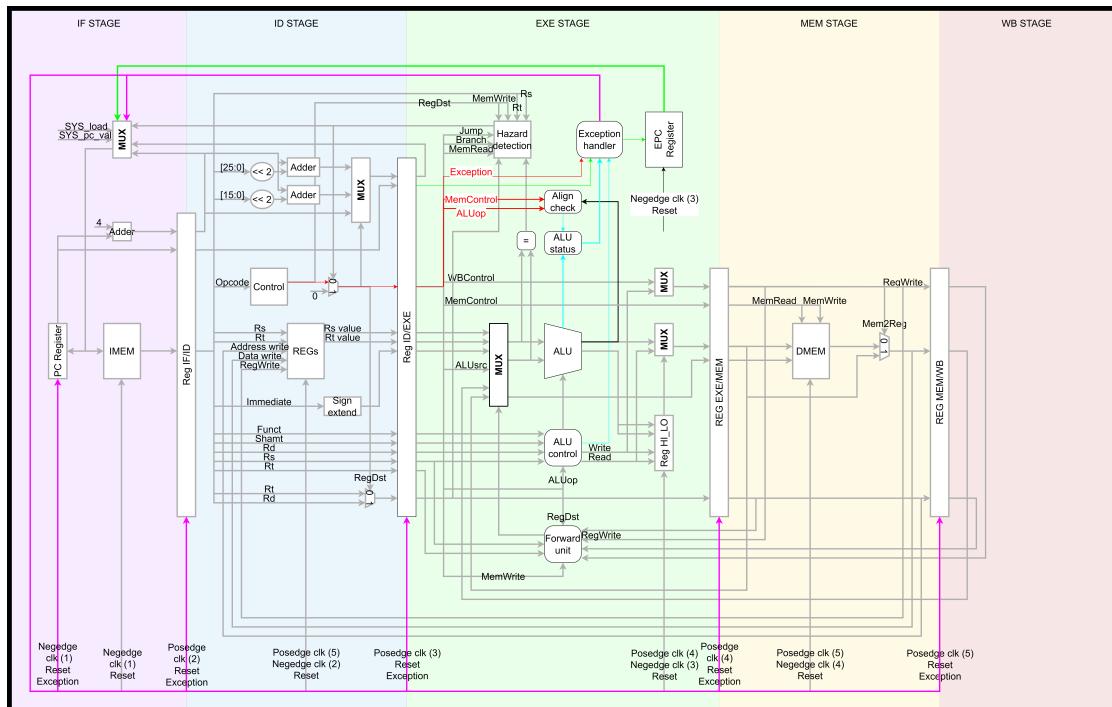
- Không xác định được lệnh: xử lý tại khối Control, ALU control.
- Canh lề địa chỉ word sai: xử lý tại khối Align check.
- Ngoại lệ về số học (tròn số, chia cho 0): xử lý tại khối ALU.

Tín hiệu exception sinh ra từ các khối trên được đưa vào khối exception handler để lưu giá trị PC gây ra exception vào thanh ghi **EPC** cũng như sinh ra tín hiệu vô hiệu hóa các thanh ghi trung gian (reset output của các thanh ghi đó về 0, xem như các thanh ghi đó đang thực hiện một lệnh **nop**) và điều khiển bộ chọn PC để đưa giá trị EPC vào thanh ghi PC.



Hình 15: Bộ chọn PC khi tích hợp khối kiểm soát exception

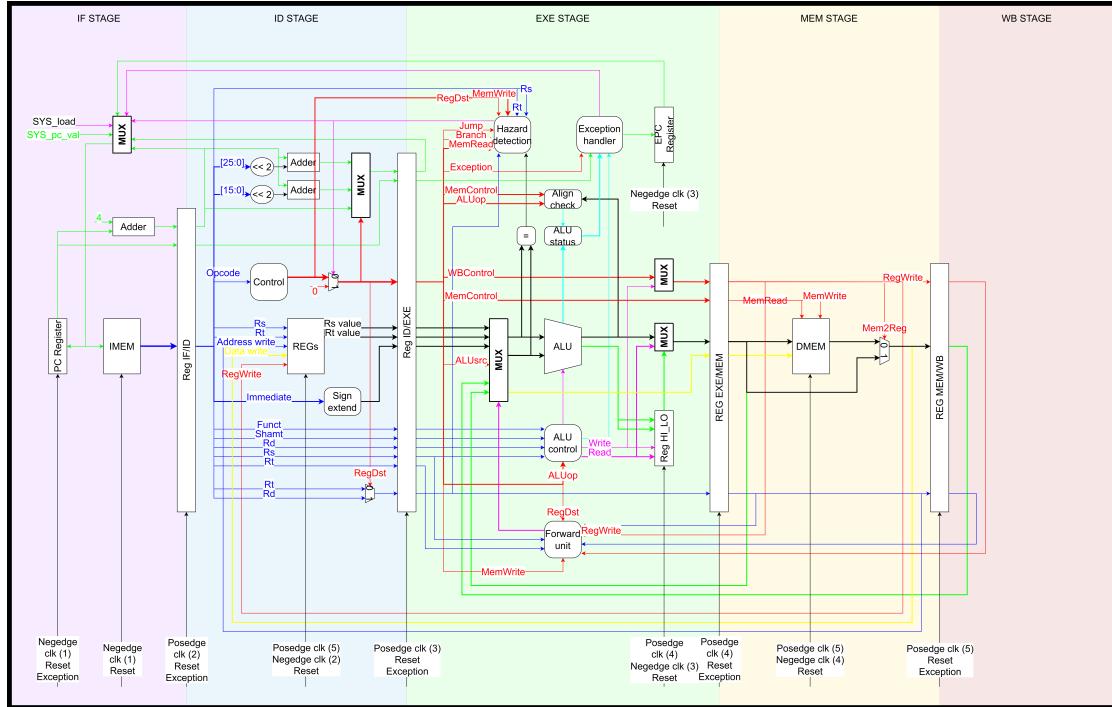
Riêng đối với **Register file**, bộ data memory, bộ instruction memory và thanh ghi high/low, nhóm không đưa tín hiệu exception vào các khối này để đảm bảo những lệnh liền trước lệnh gây ra exception vẫn được thực hiện qua hết 5 stages.



Hình 16: Data path khi tích hợp khối kiểm soát exception

2.5 Hoàn thiện kiến trúc

Tổng hợp các khối cơ bản, datapath của các lệnh, nhóm thu được kết quả cuối cùng sau đây:



Hình 17: Data path hoàn thiện

Datapath này được nhóm lưu trữ tại [link](#).

3 Kết quả

Trong phần này, nhóm mô phỏng và hiện thực lên mạch một số chương trình để kiểm tra tính đúng đắn của kiến trúc đã thiết kế. Bên cạnh đó, nhóm xác định được tài nguyên tổng hợp của hệ thống như sau:

Utilization		Post-Synthesis Post-Implementation	
Resource	Estimation	Available	Utilization %
LUT	4802	53200	9.03
FF	2637	106400	2.48
BRAM	1.50	140	1.07
DSP	4	220	1.82
IO	30	125	24.00
BUFG	12	32	37.50

Hình 18: Tài nguyên tổng hợp Post-Synthesis của hệ thống

Utilization		Post-Synthesis Post-Implementation	
Resource	Utilization	Available	Utilization %
LUT	4758	53200	8.94
FF	2637	106400	2.48
BRAM	1.50	140	1.07
DSP	4	220	1.82
IO	30	125	24.00
BUFG	12	32	37.50

Hình 19: Tài nguyên tổng hợp Post-Implementation của hệ thống

3.1 Chương trình 1

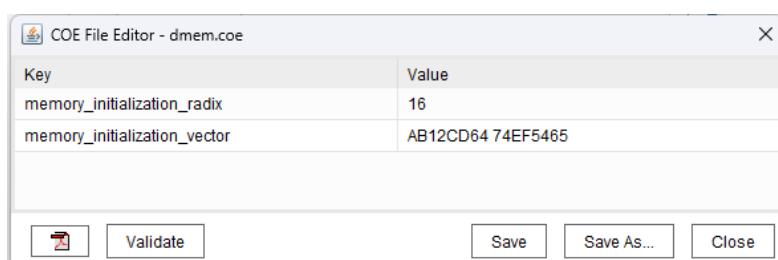
Khi hiện thực trên board, nhóm không ghi được dữ liệu vào các thanh ghi **high**, **low**, **Register file** và data memory. Do đó nhóm không ghi lại kết quả hiện thực khi thực thi những lệnh truy xuất từ các vùng nhớ này, mà chỉ ghi lại kết quả của các lệnh lấy dữ liệu thông qua khối forward unit, chương trình sau là chương trình chỉ bao gồm các lệnh như vậy:

```

1 addi $t0 , $0 , 5
2 addi $t1 , $0 , 3
3 addi $t4 , $0 , 3
4 beq $t4 , $t1 , label
5 sub $t2 , $t0 , $t1
6 xori $t3 , $t2 , 7
7 or $t2 , $t1 , $t2
8 j end
9 label:
10 addi $t3 , $0 , 3
11 addi $t4 , $t3 , -4
12 lw $s0 , 1($t4)
13 addi $s0 , $s0 , 2
14
15 xori $s1 , $0 , 27
16 andi $s2 , $s1 , 10
17 sub $s3 , $s1 , $s2
18 div $s3 , $s2
19
20 addi $s4 , $s3 , -20
21 mfhi $t8
22 mult $s4 , $t8
23
24 lh $t6 , 23($t9)
25 lw $t5 , 23($t9)
26
27 end:

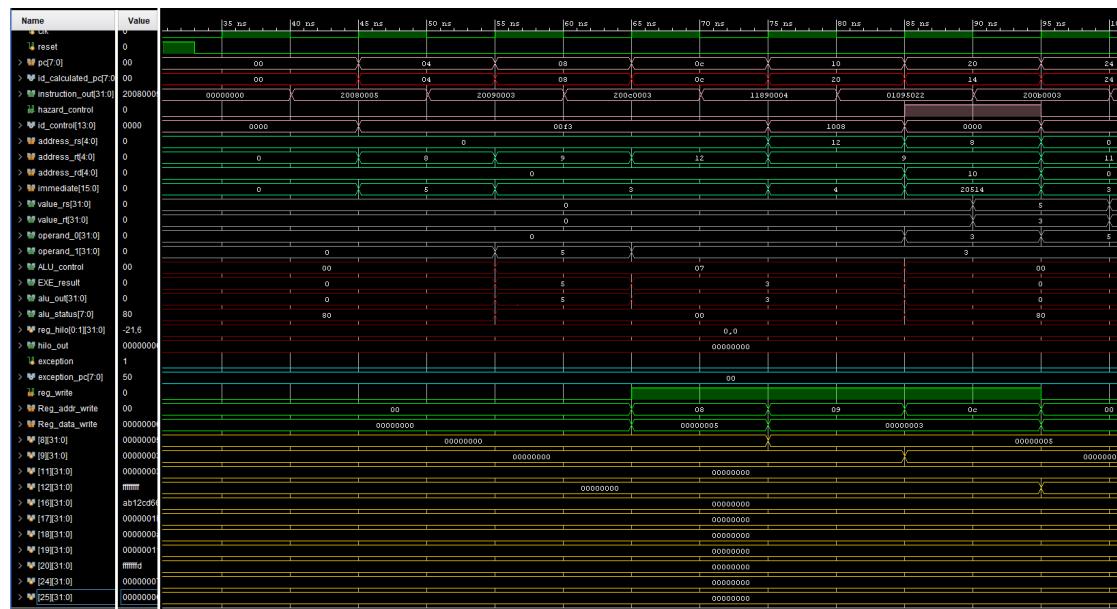
```

Ngoài ra, nhóm cài đặt trước 2 word đầu tiên trong vùng data memory lần lượt bằng 0xAB12CD64 và 0x74EF5465.



Hình 20: Cài đặt giá trị ban đầu cho data memory

3.1.1 Kết quả mô phỏng

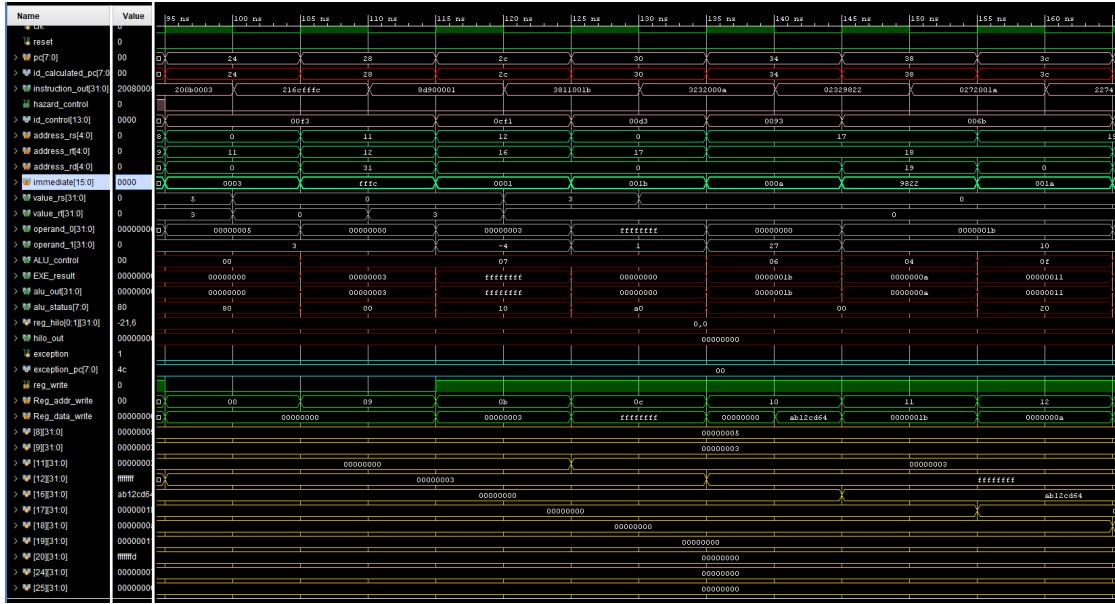


Hình 21: Kết quả mô phỏng từ 33ns - 95ns

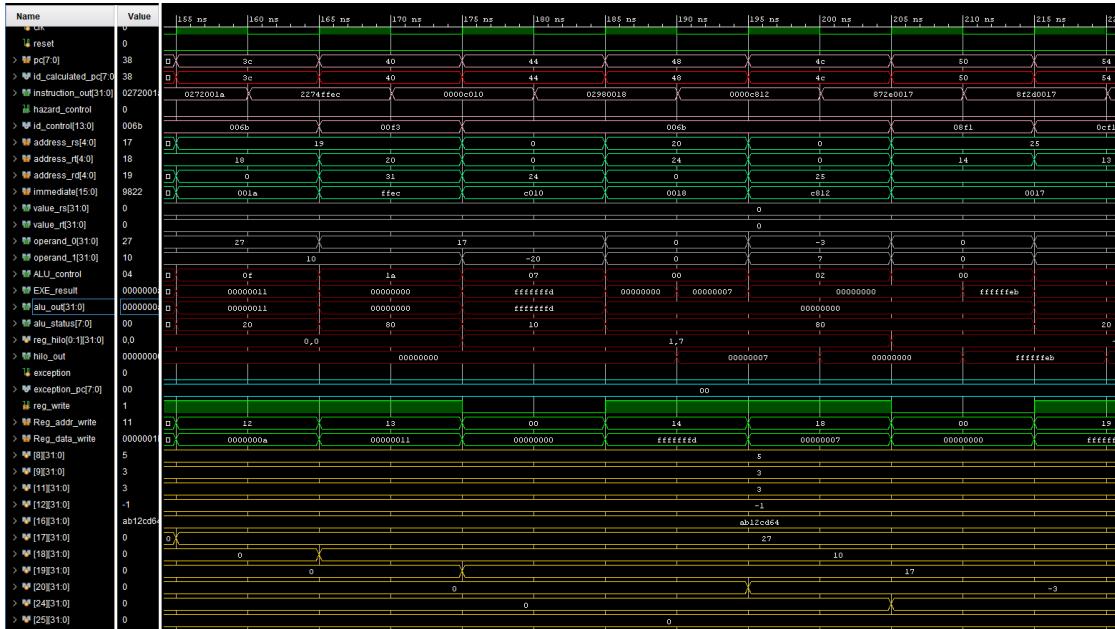
- Từ đầu thời điểm mô phỏng đến thời điểm 33ns, tín hiệu reset tích cực nên các khối được đặt về 0, riêng khối ALU status có giá trị 0x80 (1000 0000) vì kết quả của khối ALU bằng 0.
- Tại 40ns clock có cạnh xuống, lệnh thứ nhất được đọc từ instruction memory.
- Tại 45ns clock có cạnh lên, dữ liệu được đổ qua thanh ghi IF/ID, thanh ghi **Rs**, **Rt** và giá trị immediate được giải mã theo thứ tự bằng 0(\$zero), 8(\$t0) và 5. Lệnh đang thực thi là lệnh addi, cho tín hiệu control là 0x00f3 (00 0000 1111 0011), theo bảng 2 các tín hiệu tích cực là ALUOp(011) đưa vào bộ ALU control, RegDst chỉ định thanh ghi **Rt** là thanh ghi được ghi vào, Mem2Reg chỉ định dữ liệu ghi vào **Register file** là lấy từ bộ ALU và RegWrite để cho phép ghi vào **Register file**.
- Tại 50ns clock có cạnh xuống, giá trị tại thanh ghi **Rs**, **Rt** được truy xuất và đều bằng 0 đồng thời lệnh mới được đọc từ instruction memory.
- Tại 55ns clock có cạnh lên, các dữ liệu được đổ qua thanh ghi ID/EXE, giá trị của thanh ghi **Rs** và số immediate theo thứ tự được đưa vào toán hạng operand0 (0) và operand1(5) của khối ALU. Khối ALU control nhận tín hiệu ALUOp và phần funct trong mã lệnh, đưa ra tín hiệu điều khiển 0x70 (00111) xác định phép tính cần thực hiện là phép cộng, từ đó khối ALU cho kết quả bằng 5 và không có bit nào của ALU status được tích cực. Lệnh này không phải là lệnh đọc thanh ghi **high**, **low** nên kết quả của EXE stage cũng là 5. Tại thời điểm này, lệnh tiếp theo cũng được giải mã và cho kết quả mô phỏng đúng.
- Tại 65ns clock có cạnh lên, các dữ liệu được đổ qua thanh ghi EXE/MEM, tín hiệu RegWrite tích cực, xác định được thanh ghi 8(\$t0) được ghi vào giá trị 5. Tại thời điểm này, kết quả bộ ALU của lệnh kế tiếp cũng được xác định và lệnh tiếp sau đó được giải mã.



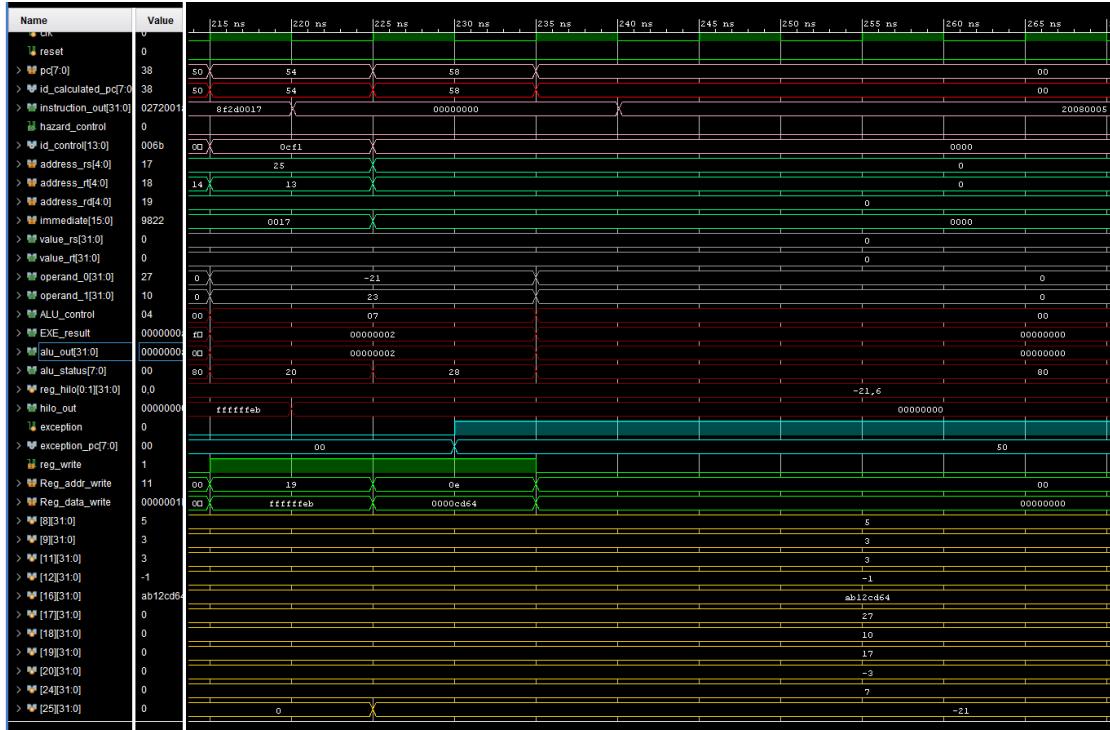
- Tại 75ns clock có cạnh lên, các dữ liệu được đổ qua thanh ghi MEM/WB, đồng thời **Register file** thực hiện ghi giá trị 5 vào thanh ghi 8(\$t0). Tại thời điểm này, tín hiệu RegWrite, thanh ghi cần được ghi và giá trị ghi vào thanh ghi đó của lệnh kế tiếp, kết quả bộ ALU của lệnh tiếp sau đó cũng được xác định và lệnh thứ 4 được giải mã. Như vậy, lệnh đầu tiên đã được hoàn thành hết 5 stage một cách chính xác.
- Tại các clock tiếp theo, các stage vẫn cho kết quả đúng, tới 85ns clock có cạnh lên, bộ so sánh sớm của lệnh thứ 4 (beq) xác định được 2 thanh ghi **Rs**, **Rt** bằng nhau và chương trình sẽ rẽ nhánh, từ đó tích cực tín hiệu hazard control để đưa tín hiệu control về 0, vô hiệu hóa lệnh kết tiếp đang được giải mã (sub), đồng thời lựa chọn được địa chỉ PC tiếp theo là 0x20. Tại thời điểm này, do địa chỉ PC lệnh liền trước trước là 0x10 và lệnh đang giải mã không phải lệnh branch nên theo bộ chọn tại hình 12 giá trị **id_calculated_pc** (địa chỉ rẽ nhánh tối nếu có hazard) là 0x14. Tuy nhiên, vì lệnh tại địa chỉ 0x20 (addi) không gây ra hazard nên địa chỉ của lệnh tiếp theo là $0x20 + 4 = 0x24$.



Hình 22: Kết quả mô phỏng từ 95ns - 155ns



Hình 23: Kết quả mô phỏng từ 155ns - 215ns



Hình 24: Kết quả mô phỏng từ 215ns

- Tại các clock tiếp theo, các stage vẫn cho kết quả đúng với đặc tả của datapath, tới 225ns clock có cạnh lên, bộ ALU của lệnh tại địa chỉ 0x50 (`lw $t5, 23($t9)`) được tính, cho kết quả bằng 2. Vì đây là lệnh load word nên địa chỉ 2 là không hợp lệ, do đó tại 230ns clock có cạnh xuống, tín hiệu exception tích cực, đưa output của tất cả thanh ghi trung gian về 0 và lưu địa chỉ 0x50 vào thanh ghi EPC.

Như vậy, kết quả chạy mô phỏng chương trình 1 là đúng với đặc tả của datapath mà nhóm đã thiết kế.

3.1.2 Kết quả hiện thực trên board

Khi hiện thực trên board, giá trị PC mà nhóm hiển thị là địa chỉ PC mà chương trình sẽ rẽ nhánh tới nếu xảy ra hazard (giá trị `id_calculated_pc` trong mô phỏng). Bên cạnh đó, vì dữ liệu không được ghi vào các thanh ghi `high`, `low` nên tại các lệnh `mfhi`, `mflo` (PC = 0x48, PC = 0x50) kết quả đọc được bằng 0 và địa chỉ cần truy xuất từ data memory của lệnh tại địa chỉ PC = 0x4C (`lh $t6, 23($t9)`) bằng 23, gây ra exception, do đó giá trị của thanh ghi EPC là 0x4C thay vì 0x50 như trong mô phỏng.

Đối với các lệnh còn lại, kết quả hiện thực trên board giống với kết quả mô phỏng và phù hợp với datapath mà nhóm đã thiết kế. Cụ thể, kết quả chạy chương trình khi hiện thực lên board được nhóm ghi lại qua [video](#).

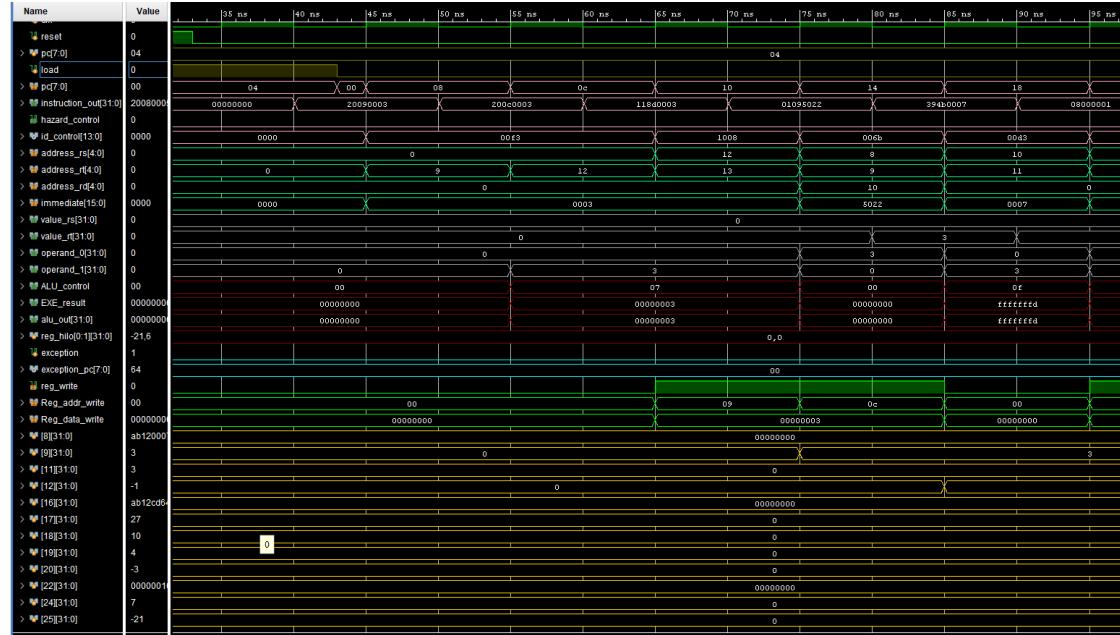


3.2 Chương trình 2

Tại chương trình này, nhóm thêm vào một số lệnh đọc và ghi vào **Register file**, thanh ghi **high, low** và data memory so với chương trình 1 để kiểm tra tính đúng đắn của kiến trúc riêng trong mô phỏng.

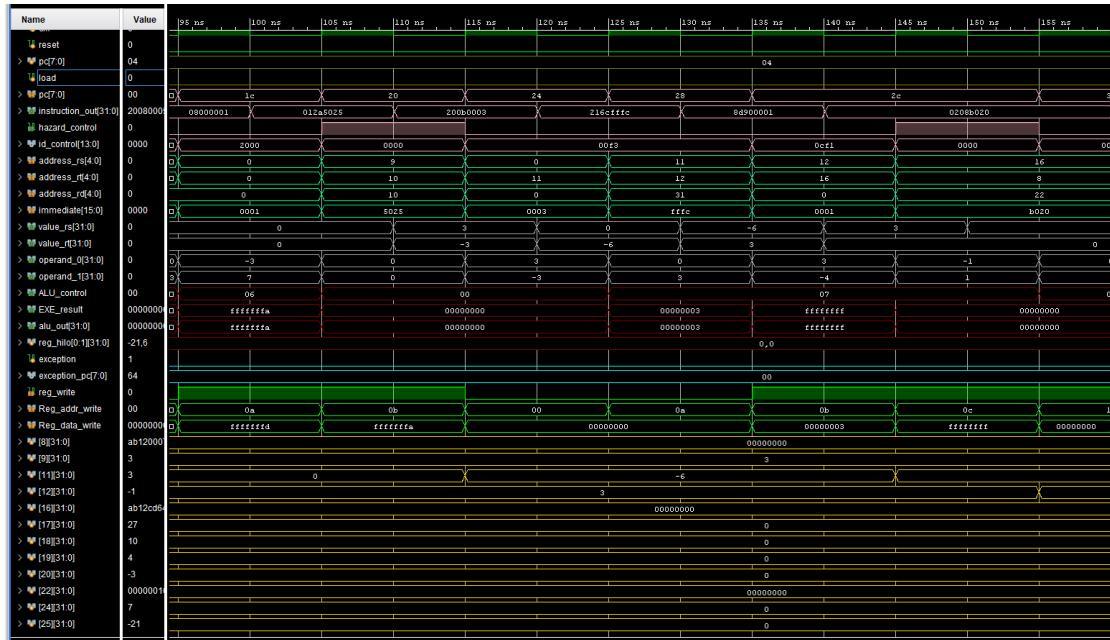
```
1 addi $t0, $0, 5
2 addi $t1, $0, 3
3 addi $t4, $0, 3
4 beq $t4, $t5, label
5 sub $t2, $t0, $t1
6 xori $t3, $t2, 7
7 j label
8 or $t2, $t1, $t2
9 label:
10 addi $t3, $0, 3
11 addi $t4, $t3, -4
12 lw $s0, 1($t4)
13 add $s6, $s0, $t0
14
15 xori $s1, $0, 27
16 andi $s2, $s1, 10
17 sub $s3, $s1, $s2
18 div $s3, $s2
19
20 addi $s4, $s3, -20
21 mfhi $t8
22 mult $s4, $t8
23 mflo $t9
24 slt $s3, $t8, $t9
25 sll $s3, $s3, 2
26 sh $t8, 23($t9)
27 lw $t0, 21($t9)
28 nor $s6, $t0, $t9
29 add $s1, $t0, $s0
```

Tương tự chương trình 1, nhóm cài đặt trước 2 word đầu tiên trong vùng data memory lần lượt bằng 0xAB12CD64 và 0x74EF5465. Chương trình 2 cho kết quả mô phỏng như sau:



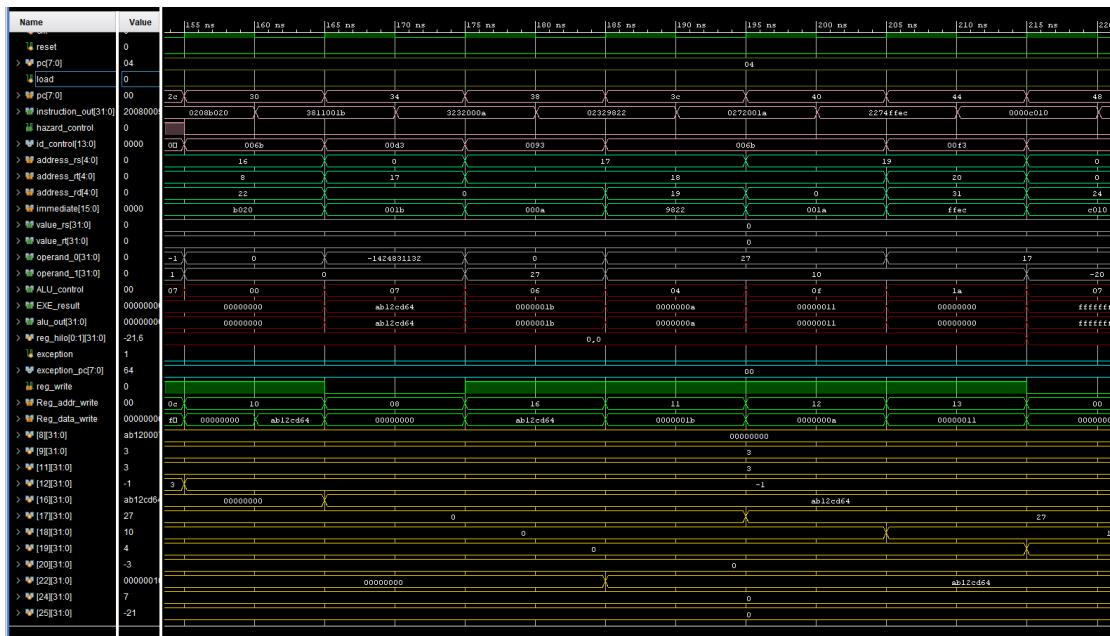
Hình 25: Kết quả mô phỏng từ 33ns - 95ns

- Từ đầu thời điểm mô phỏng tới 33ns, tín hiệu reset tích cực nên các khối được đặt về 0. Từ 33ns tới 43ns, tín hiệu load tích cực nên PC được đặt bằng 0x4 và lệnh tương ứng tại địa chỉ đó được thực thi (`addi $t1, $0 , 3`). Tới 45ns, clock có cạnh lên và tín hiệu load bằng 0 nên PC được cập nhật lên 0x8.
- Tại 60ns clock có cạnh xuống, lệnh `beq $t4, $t5`, label được đọc lên, tại 75ns, giá trị tại 2 thanh ghi của lệnh này được đưa vào bộ so sánh sớm, 2 giá trị này khác nhau nên tín hiệu **hazard_control** không tích cực như tại chương trình 1, lệnh tiếp theo vẫn được thực thi.
- Tại 90ns clock có cạnh xuống, lệnh `j` label được đọc lên, tại 105ns, tín hiệu jump sinh ra từ lệnh này được đưa vào bộ kiểm soát hazard, tích cực tín hiệu **hazard_control** vô hiệu hóa lệnh đang được giải mã tại ID stage (`or $t2, $t1, $t2`).

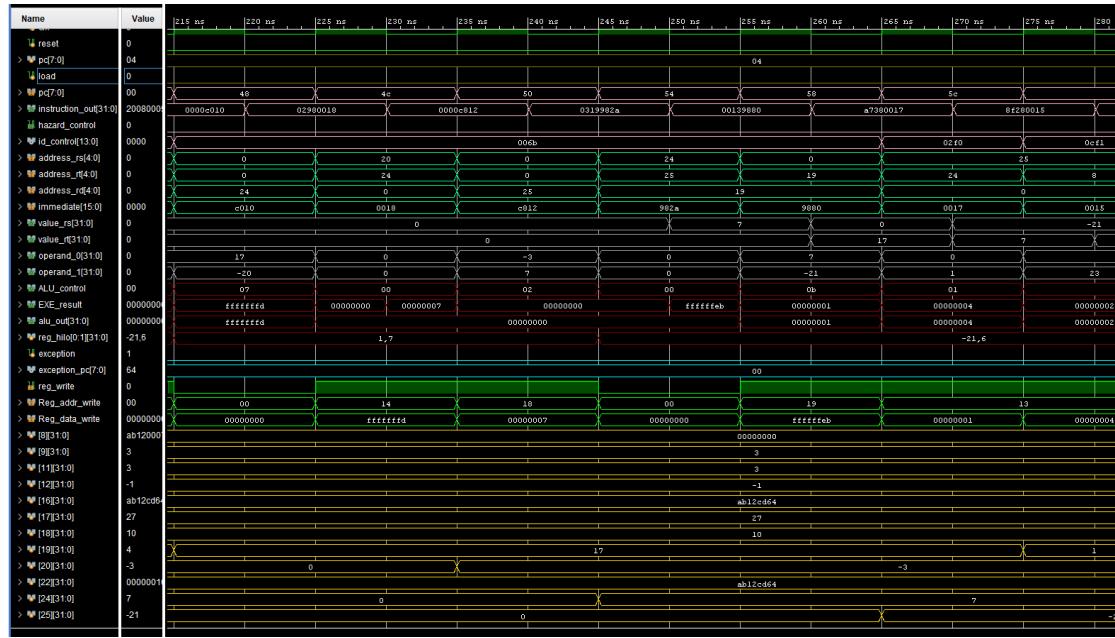


Hình 26: Kết quả mô phỏng từ 95ns - 155ns

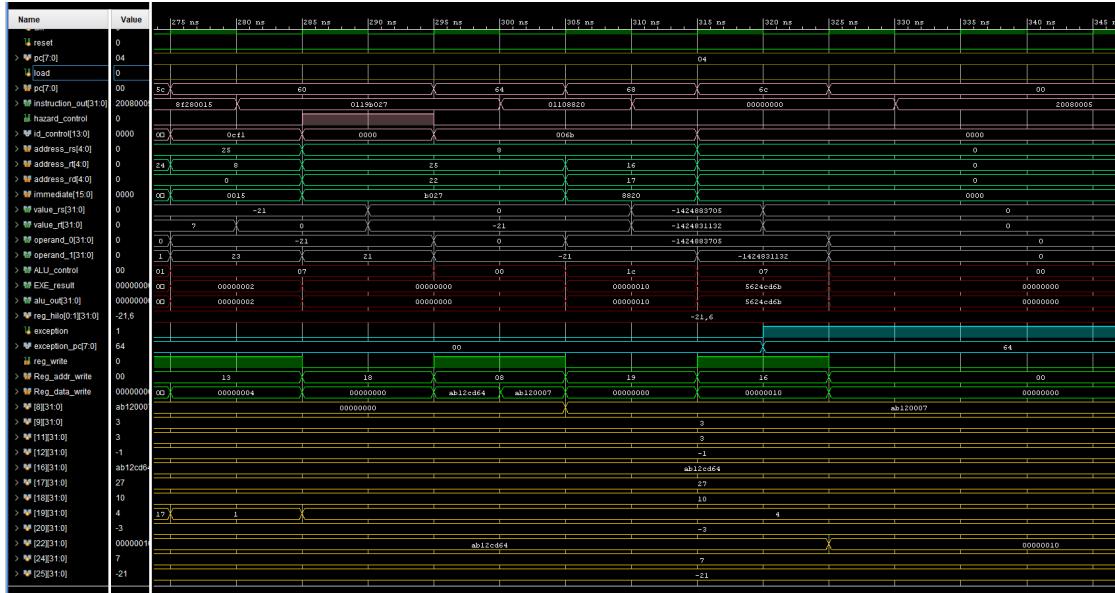
- Tương tự tại 145ns, lệnh `lw $s0, 1($t4)` được thực thi tại EXE stage, xác định được thanh ghi đích trùng với thanh ghi **Rs** của lệnh tiếp theo (`$s0`) nên tín hiệu hazard control được tích cực, một stall được chèn vào luồng thực thi của chương trình.



Hình 27: Kết quả mô phỏng từ 155ns - 215ns



Hình 28: Kết quả mô phỏng từ 215ns - 275ns



Hình 29: Kết quả mô phỏng từ 275ns

- Tại 315ns, khối ALU thực thi phép cộng từ lệnh `add $s1, $t0, $s0` với 2 toán hạng lần lượt là -1424883705 và -1424883112. Phép tính này gây ra tràn số và tích cực tín hiệu exception tại cạnh xuống tiếp theo, từ đó vô hiệu hóa các thanh ghi trung gian. Tuy nhiên, tín hiệu này không vô hiệu hóa **Register file**, kết quả của lệnh liền trước (`nor $s6, $t0, $t9`) vẫn được ghi vào thanh ghi thích hợp.

Tại các thời điểm không được đề cập, các tín hiệu mô phỏng cũng cho ra kết quả đúng. Như vậy, kết quả chạy mô phỏng chương trình 2 là đúng với đặc tả của datapath mà nhóm đã thiết kế.

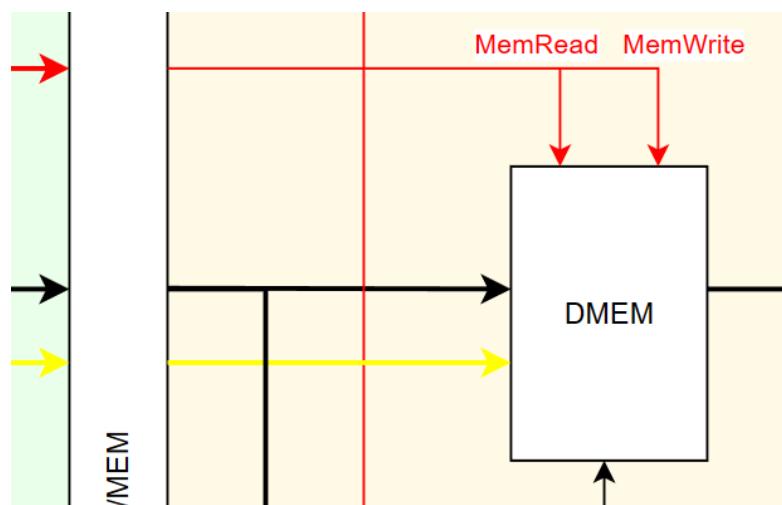
3.3 Vấn đề lưu dữ liệu vào thanh ghi

3.3.1 Nguyên nhân

Theo kết quả mô phỏng, việc lưu dữ liệu vào thanh ghi của kiến trúc là thành công và hoàn toàn ổn định. Tuy nhiên, khi hiện thực bằng phần cứng, việc lưu dữ liệu lại thất bại.

Giải thích cho vấn đề trên, nhóm cho rằng sự bất ổn định của dữ liệu và tín hiệu điều khiển việc ghi là nguyên nhân gây ra sự khác biệt.

Chứng minh cho giả định của mình, nhóm chọn ví dụ sau đây:



Hình 30: Sự bất ổn định của dữ liệu và tín hiệu điều khiển

Có thể thấy khi nhận cạnh lên của clock, dữ liệu ghi (màu vàng) và tín hiệu điều khiển (MemWrite) sẽ điều khiển việc ghi vào khối DMEM. Tuy nhiên, cùng lúc này, dữ liệu đó cũng đang được cập nhật lại bởi thanh ghi trung gian EXE/MEM khiến tín hiệu bị mất ổn định trên thực tế.

Vấn đề tương tự cũng xảy ra đối với việc ghi dữ liệu vào bộ thanh ghi REGs, cùp thanh ghi **high, low**.

3.3.2 Giải pháp

Để khắc phục vấn đề trên, nhóm đưa ra một số giải pháp sau đây:

- Sử dụng một clock riêng, thích hợp cho việc ghi dữ liệu nhằm đảm bảo sự ổn định của tín hiệu.
- Nâng cấp kiến trúc theo hướng super-pipeline nhằm tạo ra nhiều stage hơn, hướng tới khắc phục tình trạng ghi, cập nhật dữ liệu đồng thời.

4 Kết luận

4.1 Những hạn chế của kiến trúc

- Vì hạn chế về phần cứng, bộ nhớ lệnh và bộ nhớ dữ liệu chỉ lưu trữ tối đa 256 byte, chưa tương đồng với bộ nhớ của kiến trúc. Vẫn đề trên đã đến một số hạn chế như thanh ghi pc, epc chỉ có 8 bit; lệnh **j** phải thực hiện theo cơ chế riêng.
- Tập lệnh vẫn chưa hoàn thiện, vẫn còn một số lệnh được sử dụng thường xuyên nhưng chưa được hiện thực như **jal**, **jr**, **lui**, các lệnh unsigned,...
- Khối control chưa hoàn toàn kiểm soát được lệnh dẫn đến một số trường hợp cần các tín hiệu điều khiển bổ sung, gây ra những nhập nhằng không đáng có.
- **Chưa thể lưu vào thanh ghi khi hiện thực trên phần cứng.**

4.2 Thuận lợi và khó khăn

4.2.1 Thuận lợi

- Mọi thành viên đều có ý thức tự giác làm việc, hỗ trợ lẫn nhau hoàn thành đề tài.
- Việc giao tiếp, trao đổi trong nhóm diễn ra thuận lợi.
- Được sự quan tâm, hướng dẫn tận tình của giảng viên.
- Nguồn tài liệu tham khảo phong phú, đa dạng.

4.2.2 Khó khăn

- Thời gian thực hiện đề tài dồn về cuối kỳ khiến việc thực hiện bị ảnh hưởng khá nghiêm trọng.
- Việc hiện thực lên phần cứng gặp nhiều khó khăn do vẫn đề mượn phần cứng từ trường còn phức tạp.

4.3 Phương hướng hoàn thiện đề tài

- Xác định chính xác nguyên nhân của vấn đề lưu dữ liệu vào thanh ghi và lựa chọn giải pháp khắc phục thích hợp.
- Tiếp tục hoàn thiện tập lệnh của kiến trúc theo hướng mở rộng nhằm phù hợp với thực tiễn.
- Đề xuất và hiện thực hóa các tín hiệu control mới phù hợp hơn với kiến trúc.



Tài liệu

- [1] Patterson, David A., Hennessy, John L.. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. Netherlands: Elsevier Science, 2014.