TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

TRƯỜNG ĐIỆN - ĐIỆN TỬ

BÁO CÁO BÀI TẬP LỚN

KIẾN TRÚC MÁY TÍNH

Đề tài:

THIẾT KẾ BỘ XỬ LÝ RISCV ĐƯỜNG ỐNG BẰNG VERILOG VÀ MÔ PHỎNG CHẠY CHƯƠNG TRÌNH SẮP XẾP NỔI BỌT

Họ tên sinh viên	MSSV	Lớp – Khóa
Nguyễn Việt Thi	20182798	Điện tử 10-K63
Vũ Tiến Thịnh	20182809	Điện tử 11-K63
Đỗ Thành Đạt	20182411	Điện tử 11-K63

Giảng viên hướng dẫn: PGS TS. NGUYỄN ĐỨC MINH

Hà Nội, 8-2022

LỜI NÓI ĐẦU

Hiện nay, công nghệ phát triển ngày càng nhanh, máy tính cũng yêu cầu phải làm được nhiều việc hơn với tốc độ xử lý nhanh và hiệu suất cao hơn. Hàng loạt các cấu trúc x86, x64 theo đà phát triển của công nghệ đang dần trở nên lỗi thời và kém hiệu quả khi sử dụng. Để theo kịp xu thế đó thì bộ xử lý RISC-V đã được ra đời nhằm đáp ứng yêu cầu ngày càng cao của con người. Kiến trúc này tuy ra đời chưa lâu nhưng nó đã giúp đáp ứng được những yêu cầu cao trong quá trình xử lý của máy tính , đồng thời khắc phục những hạn chế, yếu điểm của các kiến trúc cũ.

Chương trình học môn Kiến trúc máy tính (ET4041) RISC-V Processor Design là một phần quan trọng trong RISC-V nhằm giúp chúng em hiểu rõ về quá trình hoạt động khi thực hiện các lệnh. Trong bài báo cáo này, chúng em có thực hiện thiết kế bộ xử lý RISC-V đường ống bằng ngôn ngữ VERILOG và mô phỏng chạy chương trình sắp xếp nổi bọt trên phần mềm Questasim.

Nhóm chúng em xin chân thành cảm ơn PSG.TS Nguyễn Đức Minh đã tận tâm giảng dạy và cung cấp kiến thức trong quá trình học tập môn Kiến trúc máy tính. Qua đó giúp chúng em có thêm kiến thức để thực hiện đề tài một cách tốt nhất.

MỤC LỤC

DANH MỤC KÝ HIỆU VÀ CHỮ VIẾT TẮT

ISA	Instruction Set Architecture
RISC	Reduced Instruction Set Computer
ALU	Arithmetic logic unit

DANH MỤC HÌNH VẼ

DANH MỤC BẢNG BIỂU

CHƯƠNG 1. GIỚI THIỆU CHUNG

1.1 Giới thiệu chung

RISC-V là một kiến trúc tập lệnh tiêu chuẩn mở (ISA) (*Instruction Set Architecture*) phần cứng mã nguồn mở dựa trên kiến trúc tập lệnh máy tính với tập lệnh đơn giản hóa *Reduced Instruction Set Computer* (RISC).Không giống như hầu hết các ISA khác, thiết kế RISC – V ISA được cung cấp theo *Open Source Licenses* không yêu cầu phí sử dụng.

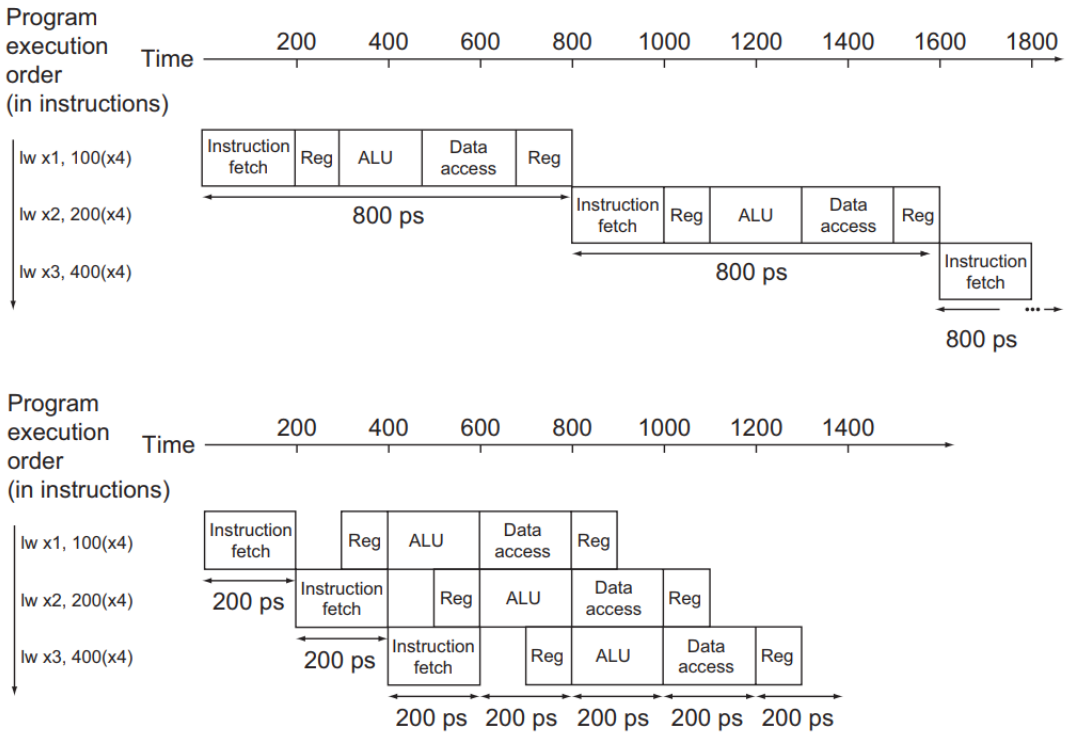
Các tính năng đáng chú ý của RISC – V bao gồm *load-store architecture*, các bit mẫu để đơn giản hóa ghép kênh trong CPU, dấu phẩy IEEE 754. Thiết kế trung lập về mặt kiến trúc và đặt *most-significant* bits tại một vị trí cố định để tăng tốc độ *sign extension.* Kiến trúc tập lệnh RISC – V cho nhiều mục đích sử dụng, nó có thể thay đổi kích thước bit mã hóa và có thể mở rộng để luôn có thể thêm nhiều bit mã hóa hơn. Nó hỗ trợ các thanh ghi có chiều dài 32, 64, 128 bits và nhiều tập hợp con. Với mỗi thanh ghi có kích thước khác nhau thì có cách đánh địa chỉ khác nhau khi sử dụng. Đặc biệt kiến trúc tập lệnh phù hợp cho tất cả hệ thống máy tính, hệ thống nhúng và siêu máy tính.

1.2 Pipeline

Pipelining là kỹ thuật mà nhiều lệnh được thực hiện theo dạng nạp chồng (overlap). Kỹ thuật này được sử dụng phổ biến trong các kiến trúc CPU hiện nay. Quá trình thực hiện một lệnh trong RISC – V cổ điển bao gồm 5 bước:

1. Nạp lệnh từ bộ nhớ
2. Giải mã lệnh và đọc các thanh ghi cần
3. Thực thi các phép tính hoặc tính toán địa chỉ
4. Truy xuất các toán hạng trong bộ nhớ
5. Ghi kết quả cuối vào thanh ghi

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



Hình 1.1 Single cycle versus pipelined (1)

	Single Cycle	Pipelined
Timing	t_{step} = 100 ... 200 ps	t_{cycle} = 200 ps
	Register access only 100 ps	All cycles same length
Instruction time, $t_{instruction}$	= t_{cycle} = 800 ps	1000 ps
CPI (Cycles Per Instruction)	~1 (ideal)	~1 (ideal), <1 (actual)
Clock rate, f_s	1/800 ps = 1.25 GHz	1/200 ps = 5 GHz
Relative speed	1 x	4 x

Hình 1.2 Single cycle versus pipelined (2)

Trong trường hợp lý tưởng : khi mà các công đoạn pipeline hoàn toàn bằng nhau thì thời gian giữa hai lệnh liên tiếp được thực thi trong pipeline bằng:

$$Thời\ gian\ giữa\ 2\ lệnh\ liên\ tiếp\ pipeline\ = \frac{thời\ gian\ giữa\ hai\ lệnh\ liên\ tiếp\ không\ pipeline}{số\ tầng\ pipeline}$$

Trong thực tế : Các công đoạn thực tế không bằng nhau, việc áp dụng pipeline phải chọn công đoạn dài nhất để làm một chu kỳ pipeline. Vì vậy trong ví dụ trên thời gian liên tiếp giữa hai lệnh pipeline là 200ps Dựa vào Hình 1.1, ta sử dụng chung một kiến trúc phần cứng, thời gian khi thực hiện các lệnh không sử dụng pipeline là 3 x 800 = 2400 ps, nhưng khi sử dụng pipeline là 3 x 200 = 600 ps. Như vậy khi áp dụng pipeline tăng tốc gấp 4 lần so với không sử dụng pipeline.

Tuy nhiên, thời gian của giai đoạn pipeline cũng bị giới hạn bởi tài nguyên chậm nhất như hoạt động ALU hoặc truy cập vào bộ nhớ. Kết quả cụ thể được mô tả trên Hình 1.2.

Kỹ thuật pipeline không giúp giảm thời gian thực thi của từng lệnh riêng lẻ mà giúp giảm tổng thời gian thực thi của từng đoạn lệnh/ chương trình chứa nhiều lệnh (từ đó giúp thời gian trung bình của mỗi lệnh giảm). Việc giúp giảm thời gian thực thi cho nhiều lệnh vô cùng quan trọng, vì chương trình chạy trong thực tế thông thường lên đến hàng tỉ lệnh.

1.3 Hazard trong pipeline

Một vấn đề xảy ra với pipeline là hiện tượng hazard. Hazard là một tình huống ngăn cản việc bắt đầu lệnh tiếp theo trong chu kì tiếp theo. Có ba loại hazard:

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

Việc loại bỏ hazard sẽ được trình bày chi tiết ở CHƯƠNG 2.

CHƯƠNG 2. THIẾT KẾ RISC-V 5-STAGE 32BIT PIPELINE

Chương này nêu tổng quan về các khối có trong RISC-V 5-stages pipeline 32-bit và ý nghĩa của chúng, cũng như sơ đồ kết nối giữa các khối để tạo ra một thiết kế hoàn chỉnh.

2.1 Tổng quan hệ thống

2.1.1 Mô tả hệ thống

Bảng 2.1 mô tả các chức năng được hỗ trợ của hệ thống RISC-V. Thiết kế có tham khảo ở [1].

Bảng 2.1. Tính năng hỗ trợ của hệ thông

Tính năng	RISC-V CPU
ISA	RISC-V
Pipelining	5 stages
Data forwarding	Có
Instruction	32 bit
Hazard detection	Data hazard (MEM-Hazard, EX-Hazard), Structural harzard, Control hazard

Các lệnh được hỗ trợ:

- Lệnh loại R: add sub xor or and sll srl sra slt sltu
- Lệnh loại I: addi xori ori andi slli srli srai slti sltiu lw
- Lệnh loại S: sw
- Lệnh Loại B: beq bne blt bge bltu bgeu
- Lệnh loại J: jal (jump and link)
- Lệnh U: lui (Load Upper Imm)

2.1.2 Sơ đồ khối

Hình 2.1 mô tả sơ đồ khối của hệ thống bao gồm datapath và control logic trong RISC-V 32 pipeline để thực hiện các chức năng đã nêu rõ ở mục 2.1.1.

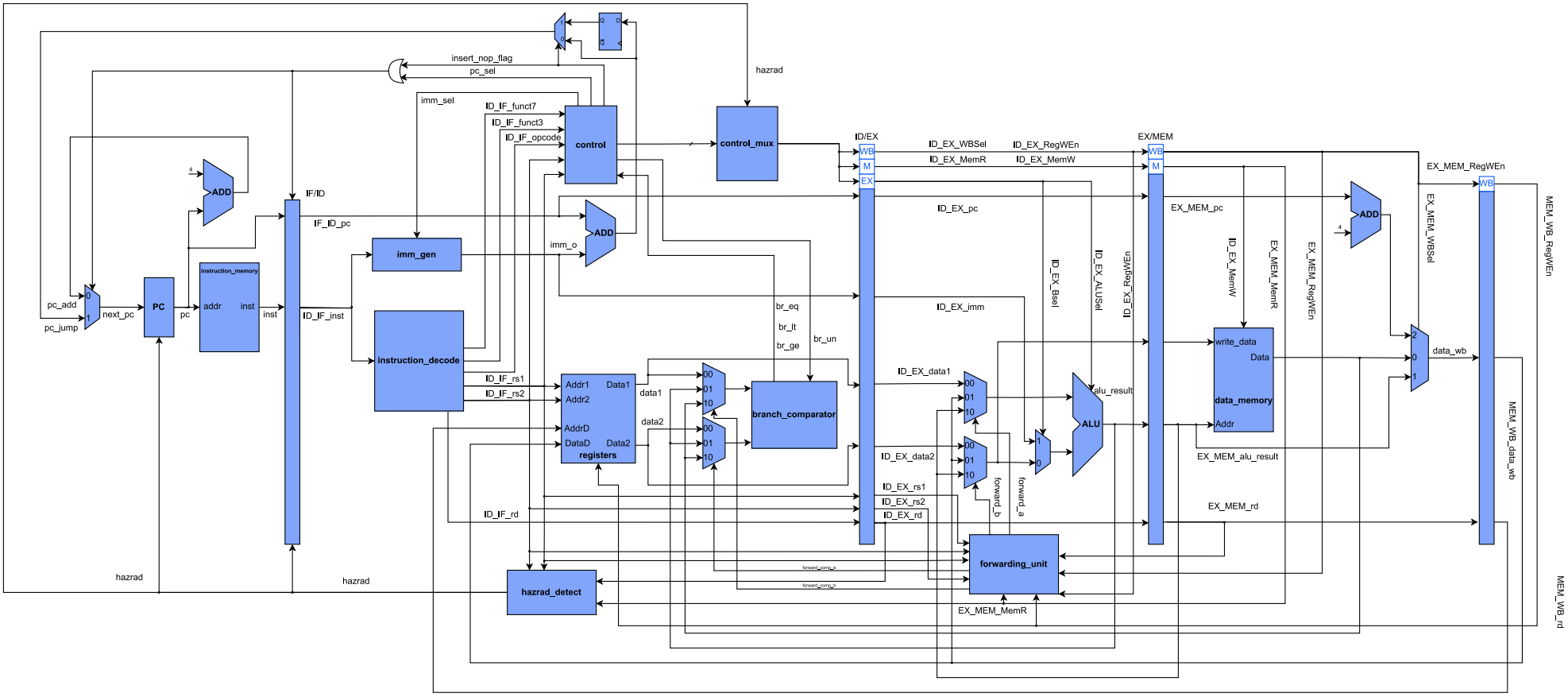
Bảng 2.2 liệt kê các tín hiệu vào ra của hệ thống RISC-V pipeline 32bit.

Bảng 2.2. Tín hiệu vào ra của hệ thống

Name	Width	Input/Output	Description
clk	1	Input	Tín hiệu xung đồng hồ
rst_n	1	Input	Tín hiệu reset tích cực mức thấp

Hệ thống đã triển khai được lệnh loại B với 1 chu kỳ stall (1 lệnh NOP) và lệnh JAL với 2 chu kỳ stall.

Khối top là khối kết nối tất cả các module con của CPU, nhận xung clk và tín hiệu reset (Bảng 2.2) để hoạt động, chương trình cần CPU thực hiện sẽ được nạp sẵn vào bộ nhớ Instruction memory.

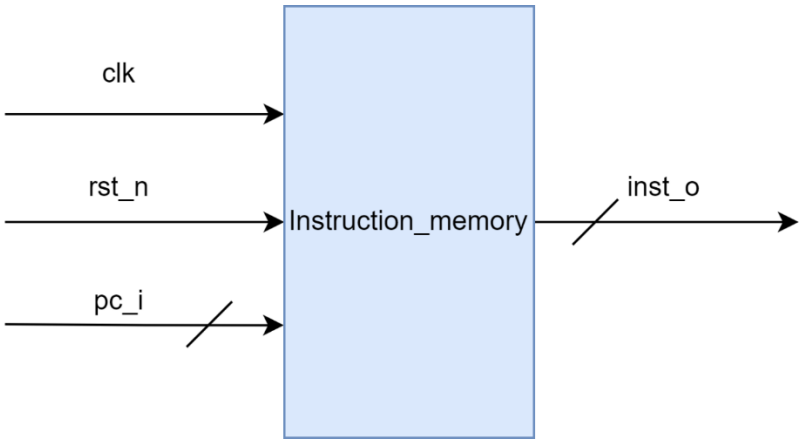


Hình 2.1. Sơ đồ khối tổng quát của hệ thống RISC-V 5-stage pipeline 32 bit

2.2 Mô tả chi tiết

2.2.1 Instruction memory

Instruction memory có nhiệm vụ lưu trữ các lệnh được nạp vào hệ thống. Từ đó, nó sẽ trả về lệnh có địa chỉ tương ứng (PC). Hình 2.2 mô tả sơ đồ tổng quát của instruction memory.



Hình 2.2. Sơ đồ khối của khối instruction memory

Bảng 2.3 chỉ rõ chức năng, độ rộng của các tín hiệu có trong instruction memory.

Để tránh trường hợp structural harzard, khối instruction memory sẽ được tách riêng biệt với khối data memory.

Bảng 2.3. Chân vào ra của instruction memory

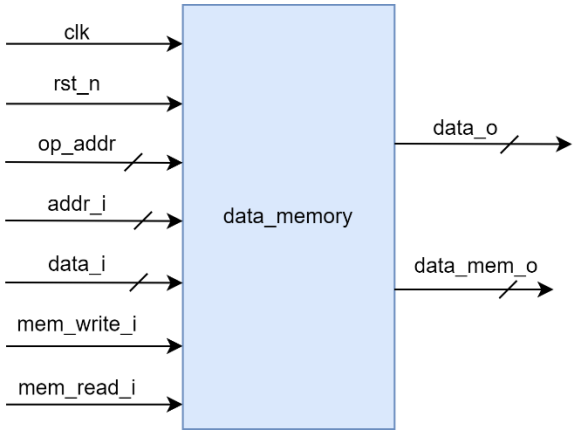
Name	Width	Input/Output	Description
clk	1	Input	Tín hiệu xung đồng hồ
rst_n	1	Input	Tín hiệu reset tích cực mức thấp
pc_i	32	Input	Tín hiệu PC đầu vào
inst_o	32	Output	Lệnh trả ra với PC tương ứng

Ở đây, để nạp lệnh vào hệ thống, nhóm đang sử dụng lệnh readmemh trong khối initial.

2.2.2 Data Memory

Khối data memory (Hình 2.3) thực hiện chức năng lưu trữ dữ liệu (RAM). Các tín hiệu cho phép đọc ghi (mem_read, mem_write) sẽ điều khiển dữ liệu vào ra.

Tương tự khối instruction memory, để tránh trường hợp structural harzard, khối data memory sẽ tách riêng biệt.



Hình 2.3. Sơ đồ khối của khối data memory

Bảng 2.4 mô tả chức năng của các tín hiệu vào ra của khối.

Bảng 2.4. Chân vào ra của khối data memory

Name	Width	Input/Output	Description
clk	1	Input	Tín hiệu xung đồng hồ
rst_n	1	Input	Tín hiệu reset tích cực mức thấp
op_addr	5	Input	Địa chỉ đọc dữ liệu từ bên ngoài
addr_i	32	Input	Địa chỉ đọc ghi
data_i	32	Input	Dữ liệu ghi vào RAM
mem_write_i	1	Input	Tín hiệu cho phép ghi
mem_read_i	1	Input	Tín hiệu cho phép đọc
data_o	32	Output	Dữ liệu đọc từ RAM

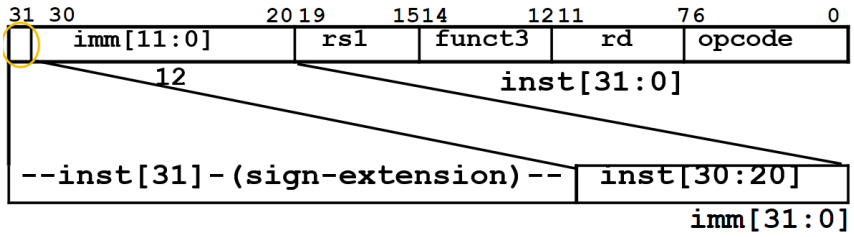
2.2.3 Imm_gen

Khối Immediate generation có chức năng nhận diện các loại lệnh động thời sắp xếp kết hợp các bit loại immediate của các loại lệnh đó theo cấu trúc như Hình 2.4, đồng thời mở rộng bit dấu để tạo ra đầu ra 32 bit.

Hình 2.5 là một ví dụ về tạo giá trị immediate của lệnh loại R.

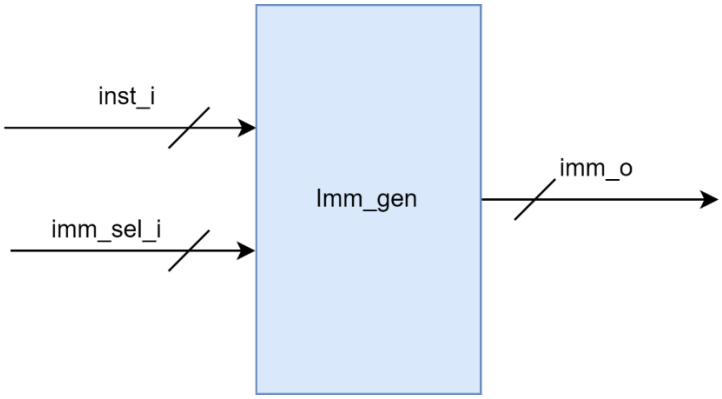
31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

Hình 2.4. Cấu trúc của instruction trong RISC-V



Hình 2.5. Ví dụ về tạo giá trị immediate của lệnh R

Hình 2.6 mô tả sơ đồ tổng quát của khối immediate generation



Hình 2.6. Sơ đồ khối của khối imm_gen

Bảng 2.5 mô tả chức năng, độ rộng các tín hiệu vào ra của khối immediate generation.

Bảng 2.5. Chân vào ra của immediate generation

Name	Width	Input/Output	Description
inst_i	32	Input	Lệnh được đưa vào
imm_sel_i	3	Input	Tín hiệu xác định loại lệnh
imm_o	32	Output	Giá trị immediate đã được tách và mở rộng dấu

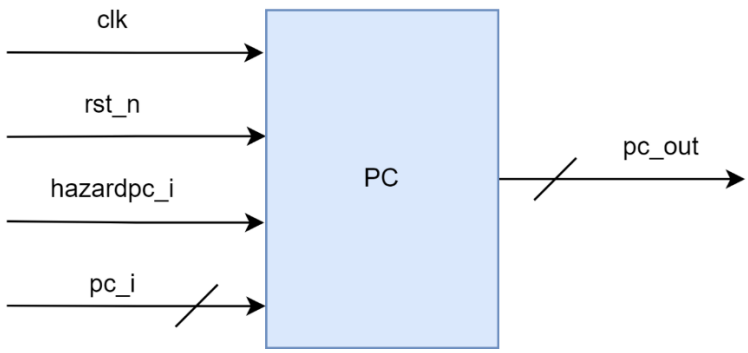
Các giá trị đầu vào imm_sel được định nghia như Bảng 2.6.

Bảng 2.6. Định giá các giá trị cảu immediate select

Tên	Giá trị
ImmSelI	3'b000
ImmSelS	3'b001
ImmSelB	3'b010
ImmSelJ	3'b011
ImmSelU	3'b100
ImmSelR	3'b111

2.2.4 PC

Khối PC đưa ra giá trị PC phù hợp ở từng thời điểm để đưa vào khối instruction memory (Hình 2.7).



Hình 2.7. Sơ đồ khối của khối PC

Ở các trường hợp xuất hiện hazard, giá trị PC sẽ được giữ nguyên như ở chu kỳ trước. Giá trị PC đưa vào là giá trị PC + 4 hoặc giá trị nhảy PC + imm.

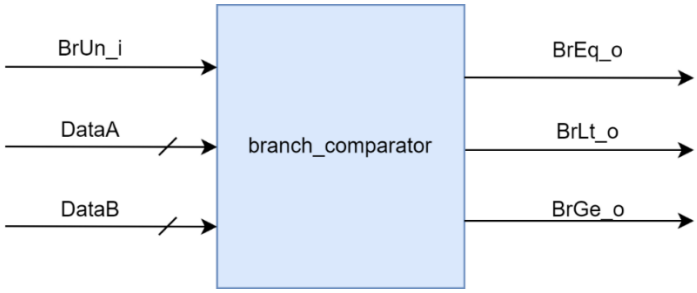
Bảng 2.7 chỉ rõ chức năng, độ rộng của các tín hiệu có trong khối PC.

Bảng 2.7. Chân vào ra của khối PC

Name	Width	Input/Output	Description
clk	1	Input	Tín hiệu xung đồng hồ
rst_n	1	Input	Tín hiệu reset tích cực mức thấp
hazardpc_i	1	Input	Tín hiệu chỉ ra có hazard
pc_i	32	Output	Giá trị PC tiếp theo
pc_out	32	Output	Giá trị PC đưa vào khối instruction memory

2.2.5 Branch Comparator

Khối Branch comparator nhận 2 dữ liệu đầu vào DataA và DataB, đồng thời so sánh 2 giá trị đó. Ở đây chúng ta có thể chọn so sánh có dấu hay không có dấu bằng tín hiệu BrUn_i (Hình 2.8).



Hình 2.8. Sơ đồ khối của khối branch comparator

Bảng 2.8 chỉ rõ chức năng, độ rộng của các tín hiệu có trong khối branch comparator. Dưới đây là mức hành vi của khối:

```
assign BrEq_o = (DataA == DataB) ? 1'b1 : 1'b0;
    always @(*) begin
  if (BrUn_i) begin
      BrLt_o = (DataA <  DataB) ? 1'b1 : 1'b0;
  end
```

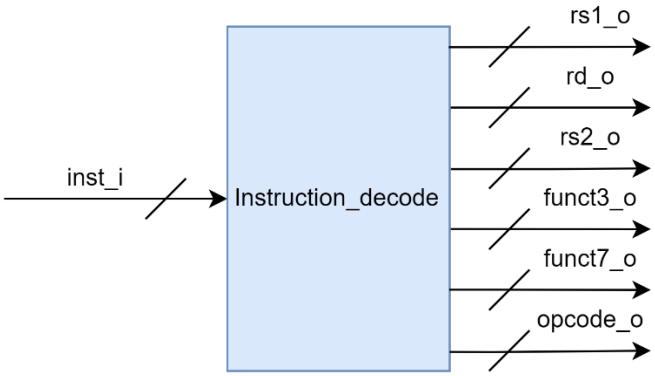
```
    else begin
        BrLt_o = ($signed(DataA) <  $signed(DataB)) ? 1'b1 : 1'b0;
    end
  end
  always @(*) begin
    if (BrUn_i) begin
        BrGe_o = (DataA >=  DataB) ? 1'b1 : 1'b0;
    end
    else begin
        BrGe_o = ($signed(DataA) >=  $signed(DataB)) ? 1'b1 : 1'b0;
    end
  end
```

Bảng 2.8. Chân vào ra của khối branch comparator

Name	Width	Input/Output	Description
BrUn_i	1	Input	Tín hiệu chọn so sánh có dấu hay không có dấu. - 1: có dấu - 0: không dấu
DataA	32	Input	Dữ liệu đầu vào A
DataB	32	Input	Dữ liệu đầu vào B
BrEq_o	1	Output	Đầu ra biểu thị 2 đầu vào bằng nhau
BrLt_o	1	Output	Đầu ra biểu thị A < B
BrGe_o	1	Output	Đầu ra biểu thị A >= B

2.2.6 Intruction decode

Intruction decode (Hình 2.9) có chức năng giải mã lệnh đưa vào. Từ đó, tách ra các trường tương ứng với các loại lệnh. Các giá trị đầu ra (opcode, rd, rs1, rs2, function3, function7) sẽ được đưa vào khối control để tính toán các tín hiệu điều khiển hệ thống và làm giá trị tính toán cho các stage tiếp theo.



Hình 2.9. Sơ đồ khối của khối Intruction decode

Các giá trị đầu ra ở các loại lệnh sẽ khác nhau, như Hình 2.4.

Bảng 2.9. Chân vào ra của khối Intruction decode

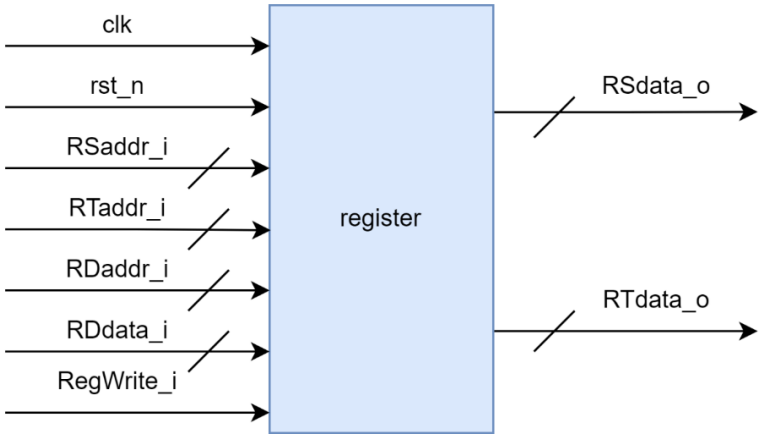
Name	Width	Input/ Output	Description
inst_i	32	Input	Mã lệnh đưa vào để giải mã
rd_o	5	output	Giá trị rd
rs1_out	5	output	Giá trị rs1
rs2_out	5	output	Giá trị rs2
funct3_o	3	output	Giá trị function 3
funct7_o	7	output	Giá trị function 7
opcode_o	7	output	Giá trị opcode

Tín hiệu vào ra của khối được mô tả trong Bảng 2.9.

2.2.7 Register

Khối register là nơi lưu trữ các giá trị của các thanh ghi. Đối với kiến trúc RISC-V, chúng ta sẽ triển khai 32 thanh ghi ($x0, x1, \dots, x31$) có độ rộng là 32 bít với thanh ghi $x0$ sẽ mặc định là giá trị 0 và không bao giờ thay đổi.

Để tránh structural hazard, trong 1 chu kỳ, ở sườn dương của clk sẽ là quá trình đọc giá trị từ thanh ghi có địa (rs_1, rs_2), còn ở sườn âm sẽ là quá trình ghi vào địa chỉ rd.



Hình 2.10. Sơ đồ khối của khối register

Hình 2.10 mô tả sơ đồ tổng quát của khối register.

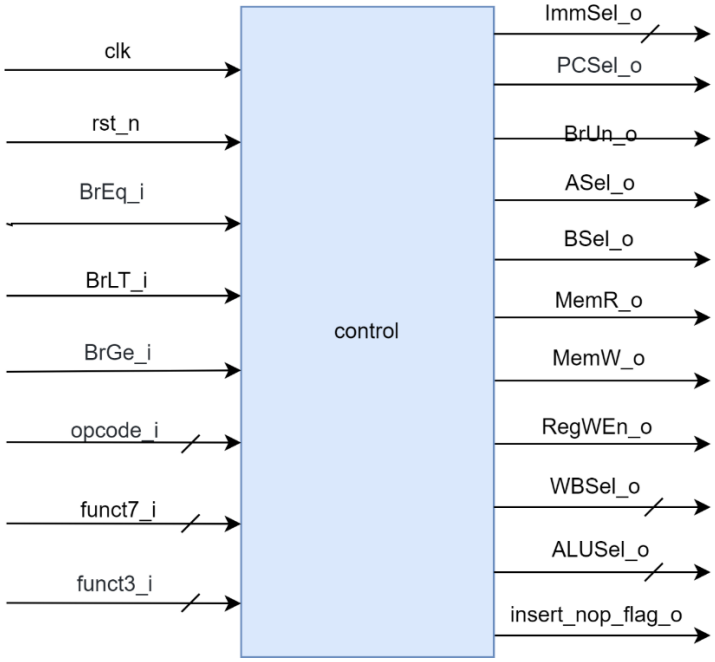
Bảng 2.10. Chân vào ra của khối register

Name	Width	Input/Output	Description
clk	1	Input	Tín hiệu xung đồng hồ
rst_n	1	Input	Tín hiệu reset tích cực mức thấp
RSaddr_i	5	Input	Địa chỉ đọc thanh ghi rs1
RTaddr_i	5	Input	Địa chỉ đọc thanh ghi rs2
RDaddr_i	5	Input	Địa chỉ của thanh ghi rd
RDdata_i	32	Input	Giá trị ghi vào thanh ghi có địa chỉ rd
RegWrite_i	1	Input	Tín hiệu cho phép ghi vào register
RSdata_o	32	Output	Giá trị của thanh ghi có địa chỉ rs1
RTdata_o	32	Output	Giá trị của thanh ghi có địa chỉ rs2

Tín hiệu vào ra của khối được mô tả trong Bảng 2.10.

2.2.8 Control

Khối control (Hình 2.11) có chức năng đưa ra các tín hiệu điều khiển ứng với các lệnh. Các tín hiệu đầu ra được mô tả như bảng chân lý (Bảng 2.12).



Hình 2.11. Sơ đồ khối của khối control

Bảng 2.11. Chân vào ra của khối control

Name	Width	Input/Output	Description
clk	1	Input	Tín hiệu xung đồng hồ
rst_n	1	Input	Tín hiệu reset tích cực mức thấp
BrEq_i	1	Input	Đầu vào biểu thị 2 giá trị bằng nhau
BrLT_i	1	Input	Đầu vào biểu thị A<B
BrGe_i	1	Input	Đầu vào biểu thị A >= B
opcode_i	7	Input	Giá trị opcode
funct7_i	7	Input	Giá trị function 7
funct3_i	3	Input	Giá trị function 3

ImmSel_o	3	Output	Tín hiệu xác định loại lệnh
PCSel_o	1	Output	Giá trị chọn PC. - 1: PC = PC + JUMP - 0 : PC = PC + 4
BrUn_o	1	Output	So sánh số có dấu: - 1: có dấu - 0: không dấu
ASel_o	1	Output	A select
BSel_o	1	Output	A select
MemR_o	1	Output	Tín hiệu cho phép đọc vào data memory
MemW_o	1	Output	Tín hiệu cho phép ghi vào data memory
RegWEn_o	1	Output	Tín hiệu cho phép ghi vào register
WBSel_o	2	Output	Tín hiệu chọn đường Write back
ALUSel_o	4	Output	Tín hiệu chọn chức năng cho khối ALU
insert_nop_flag_o	1	Output	Tín hiệu cho biết cần chèn lệnh NOP, dùng cho lệnh JAL

Bảng 2.11 mô tả chức năng của chân vào ra của khối.

Bảng 2.12. Các tín hiệu điều khiển ứng với các lệnh

	PCSel_o	ImmSel_o	BrUn_o	ASel_o	BSel_o	ALUSel_o	MemR_o	MemW_o	RegWen_o	WBSel_o
add	0	ImmSelR	x	x	0	ALUadd	0	0	1	01
sub	0	ImmSelR	x	x	0	ALUsub	0	0	1	01
sll	0	ImmSelR	x	x	0	ALUsll	0	0	1	01
slt	0	ImmSelR	x	x	0	ALUslt	0	0	1	01
sltu	0	ImmSelR	x	x	0	ALUsltu	0	0	1	01
xor	0	ImmSelR	x	x	0	ALUxor	0	0	1	01
srl	0	ImmSelR	x	x	0	ALUsrl	0	0	1	01
sra	0	ImmSelR	x	x	0	ALUsra	0	0	1	01
or	0	ImmSelR	x	x	0	ALUor	0	0	1	01
and	0	ImmSelR	x	x	0	ALUand	0	0	1	01
addi	0	ImmSelI	x	x	1	ALUadd	0	0	1	01
xori	0	ImmSelI	x	x	1	ALUxor	0	0	1	01

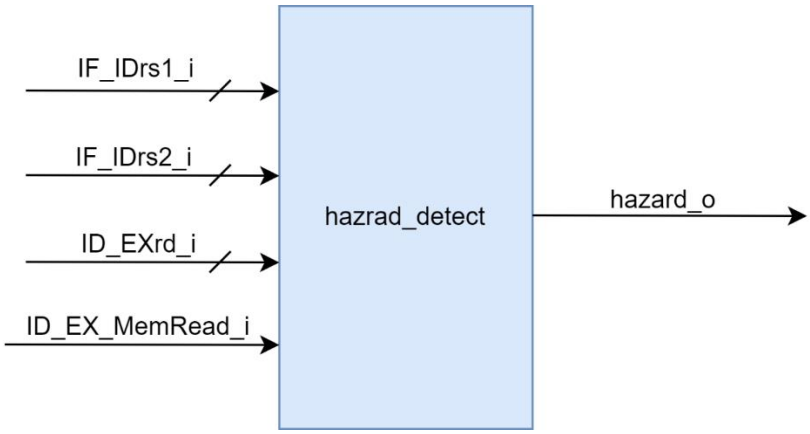
ori	0	ImmSelI	x	x	1	ALUor	0	0	1	01
andi	0	ImmSelI	x	x	1	ALUand	0	0	1	01
slli	0	ImmSelI	x	x	1	ALUsll	0	0	1	01
lw	0	ImmSelI	x	x	1	ALUadd	1	0	1	00
S	0	ImmSelS	x	x	1	ALUadd	0	1	0	xx
beq	(BrEq_i) ? 1 : 0	ImmSelB	x	x	1	ALUadd	0	0	0	xx
bne	(BrLT_i) ? 0 : 1	ImmSelB	x	x	1	ALUadd	0	0	0	xx
blt	(BrLT_i) ? 1 : 0	ImmSelB	0	x	1	ALUadd	0	0	0	xx
bge	(BrGe_i) ? 1 : 0	ImmSelB	0	x	1	ALUadd	0	0	0	xx
bltu	(BrLT_i) ? 1 : 0	ImmSelB	1	x	1	ALUadd	0	0	0	xx
bgeu	(BrGe_i) ? 1 : 0	ImmSelB	1	x	1	ALUadd	0	0	0	xx

jal	0	ImmSelJ	x	x	1	ALUadd	0	0	1	10
lui	0	ImmSelU	1	x	1	ALUadd	0	0	1	01
default	0	1111	0	x	1	ALUnop	0	0	0	01

2.2.9 Hazard Detection

Khối Hazard detection (Hình 2.12) có nhiệm vụ xác định có hazard mà không thể sử dụng kĩ thuật forwarding để xử lí mà cần stall pipeline 1 chu kì clk. Khối Hazard detection unit xác định trường hợp cần stall như sau:

```
if (ID_EX_mem_read) and (ID_EX_rd == IF_ID_rs1 or ID_EX_rd == IF_ID_rs2)and (ID_EX_rd !=0) then stall the pipline.
```



Hình 2.12. Sơ đồ khối của khối hazard dectection.

Tín hiệu hazard sẽ được gửi đi các khối để điều khiển stall 1 chu kỳ.

Bảng 2.13 mô tả chức năng của tín hiệu vào ra của khối

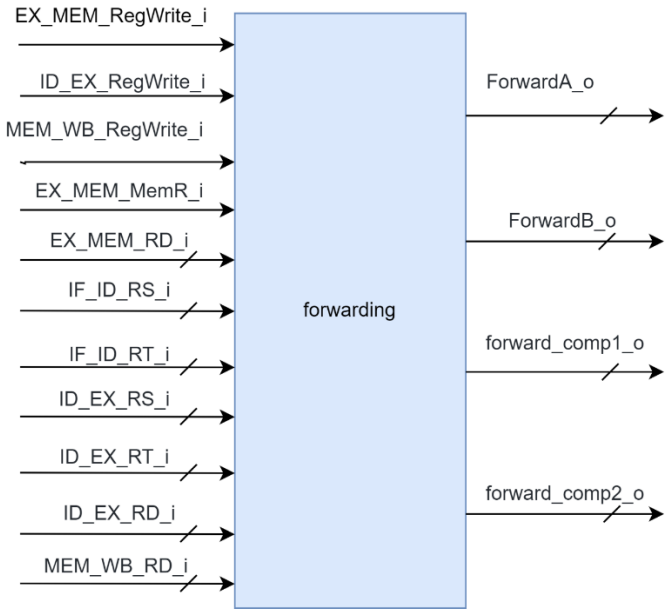
Bảng 2.13. Chân vào ra của khối Hazard detection

Name	Width	Input/Output	Description
IF_IDrs1_i	5	Input	Giá trị rs1 được lưu ở thanh ghi IF/ID
IF_IDrs2	5	Input	Giá trị rs2 được lưu ở thanh ghi IF/ID
ID_EXrd_i	5	Input	Giá trị rsd được lưu ở thanh ghi ID/EX

ID_EX_MemRead_i	1	Input	Giá trị cho phép đọc dữ liệu từ data memory
hazard_o	1	Output	Tín hiệu cho biết phát hiện hazard

2.2.10 Forwarding

Khối Forwarding sẽ nhận các giá trị địa chỉ rs1, rs2, rd và các tín hiệu điều khiển từ các thanh ghi pipeline nhằm xác định các trường hợp các lệnh liên tiếp nhau có xảy ra hazard (data hazard, mem hazard, control hazard) để đưa ra các tín hiệu điều khiển cho các bộ mux nhằm forward các dữ liệu bị hazard tương ứng để có được kết quả tính toán chính xác nhất mà không cần phải stall pipeline quá nhiều chu kì.



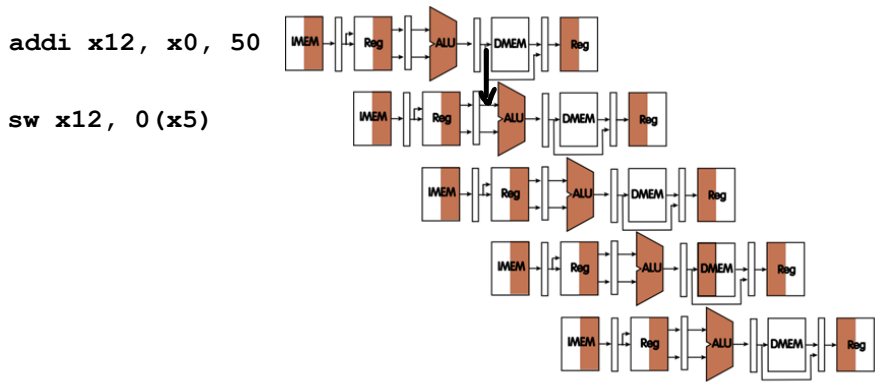
Hình 2.13. Sơ đồ khối của khối forwarding

Khối Forwarding unit sẽ xác định các trường hợp cần forward như sau:

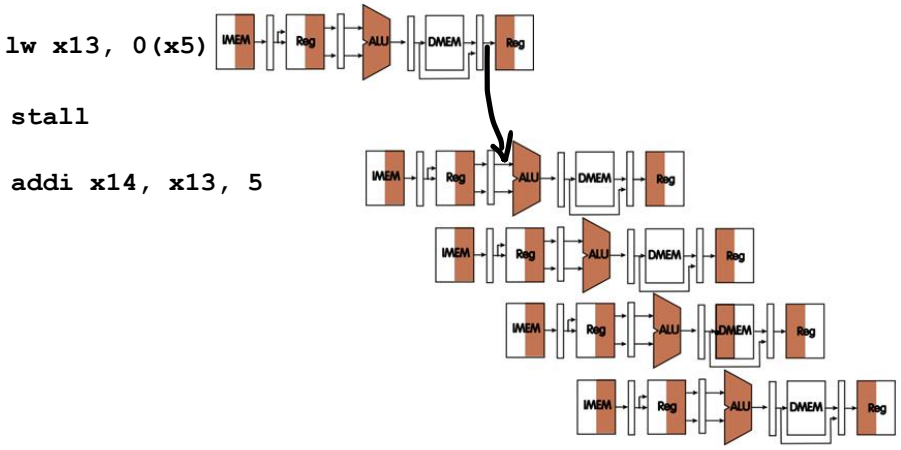
```
if(EX_MEM_RegWrite_i && (EX_MEM_RD_i != 5'b00000) && (EX_MEM_RD_i == ID_EX_RS_i)) Forward = 2'b10; //EX hazard

else if(MEM_WB_RegWrite_i && (MEM_WB_RD_i != 5'b00000) && MEM_WB_RD_i == ID_EX_RS_i)  Forward = 2'b01; //MEM hazard

else Forward = 2'b00;
```



Hình 2.14. Mô phỏng quá trình EX haxzard

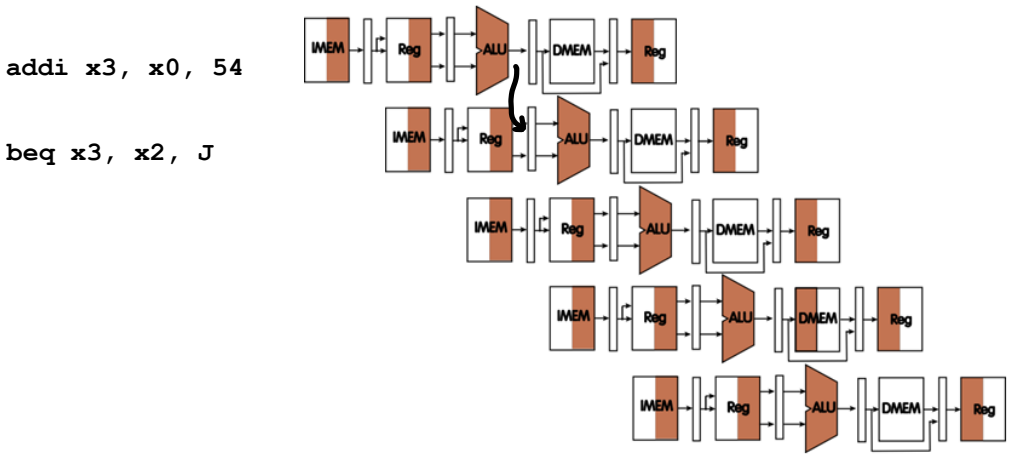


Hình 2.15. Mô phỏng quá trình MEM hazard

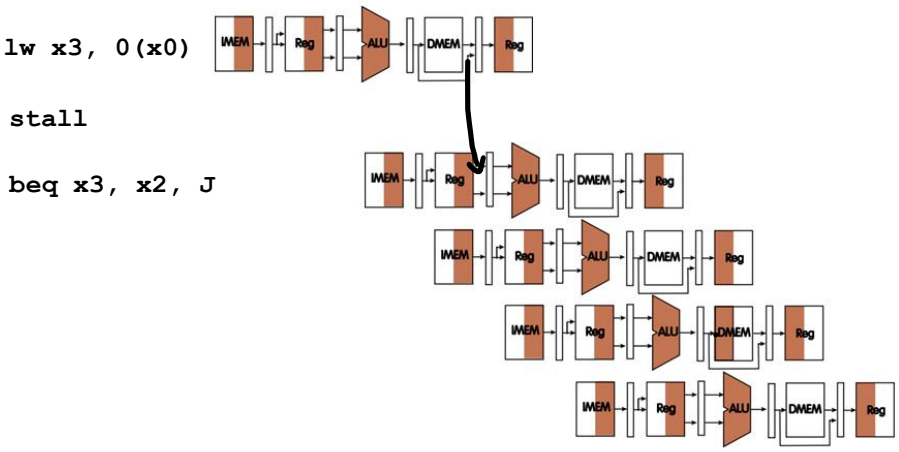
Quá trình xử lý hazard dùng forward như Hình 2.14 Hình 2.15.

Vì thiết kế đề xuất cải tiến tiết kiệm 1 chu kỳ stall cho lệnh loại B, nên bộ so sánh sẽ được đặt ở ở ID/IF nên cần 1 forward data về sau bộ so sánh. Xác định như sau:

```
if ((ID_EX_RD_i != 0) && ID_EX_RegWrite_i && (IF_ID_RS_i == ID_EX_RD_i))
      forward_comp1 = 2'b01;
else if (EX_MEM_MemR_i && (EX_MEM_RD_i != 0) && EX_MEM_RegWrite_i &&
(EX_MEM_RD_i == IF_ID_RS_i)) forward_comp1_o = 2'b10;
  else forward_comp = 2'b00;
```



Hình 2.16 Quá trình forward đối với trường hợp hazard cho lệnh loại B



Hình 2.17. Quá trình forward đối với trường hợp hazard cho lệnh loại B

Quá trình xử lý hazard dùng forward như Hình 2.16 Hình 2.17.

Bảng 2.14 mô tả các đường dữ liệu forward ứng với các giá trị select tương ứng.

Bảng 2.14. Mô tả chọn đường dữ liệu forward

Mux control	Nguồn	Mô tả
forward = 00	ID/EX	Đường input thứ nhất của ALU đến từ tệp thanh ghi.
forward = 10	EX/MEM	Đường input thứ nhất của ALU được foward từ kết quả tính toán của ALU cho lệnh ngay trước đó.

forward = 01	MEM/WB	Đường input thứ nhất của ALU được forward từ kết quả đọc data từ memory hoặc kết quả tính toán của ALU cho lệnh trước đó cách lệnh đang thực hiện 1 lệnh.
forward_comp = 00	Register	Đường input thứ hai của branch compare đến từ tệp thanh ghi.
forward_comp = 10	mem or EX/MEM	Đường input thứ hai của branch compare đến từ kết quả đọc data từ memory hoặc kết quả ALU của lệnh trước đó cách lệnh đang thực hiện 1 lệnh.
forward_comp = 01	ALU	Đường input thứ hai của branch compare đến từ kết quả tính toán ALU của lệnh trước đó

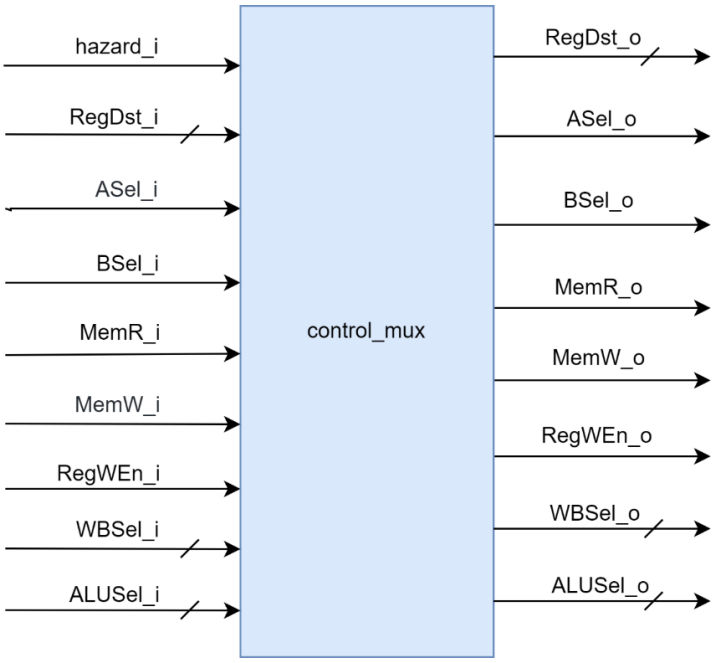
Bảng 2.15. Chân vào ra của khối forwarding

Name	Width	Input/Output	Description
EX_MEM_RegWrite_i	1	Input	Giá trị cho phép ghi vào register lưu ở EX/MEM
ID_EX_RegWrite_i	1	Input	Giá trị cho phép ghi vào register lưu ở ID/EX
MEM_WB_RegWrite_i	1	Input	Giá trị cho phép ghi vào register lưu ở MEM/WB
EX_MEM_MemR_i	1	Input	Giá trị cho phép đọc data memory lưu ở EX/MEM
EX_MEM_RD_i	5	Input	Giá trị rd lưu ở EX/MEM
IF_ID_RS_i	5	Input	Giá trị rs1 lưu ở IF/ID

IF_ID_RT_i	5	Input	Giá trị rs2 lưu ở IF/ID
ID_EX_RS_i	5	Input	Giá trị rs1 lưu ở ID/EX
ID_EX_RT_i	5	Input	Giá trị rs2 lưu ở ID/EX
ID_EX_RD_i	5	Input	Giá trị rd lưu ở ID/EX
MEM_WB_RD_i	5	Input	Giá trị rd lưu ở MEM/WB
ForwardA_o	2	Output	Giá trị chọn đường forward của 1
ForwardB_o	2	Output	Giá trị chọn đường forward của 2
forward_comp1_o	2	Output	Giá trị chọn đường forward của khối comp 1
forward_comp2_o	2	Output	Giá trị chọn đường forward của khối comp 2

2.2.11 Control Mux

Khi phát hiện có hazard, khối control mux (Hình 2.18) sẽ xóa hết các tín hiệu các tín hiệu điều khiển về 0 và đẩy sang thanh ghi ID/EX.



Hình 2.18. Sơ đồ khối của khối control mux

Bảng 2.16 mô tả chức năng của từng chân vào ra của khối.

Bảng 2.16. Chân vào ra của khối control mux

Name	Width	Input/Output	Description
Hazard_i	1	Input	Tín hiệu chỉ ra có hazard
RegDst_i	5	Input	Giá trị rd ở thanh ghi IF/ID
ASel_i	1	Input	A select
BSel_i	1	Input	B select
MemR_i	1	Input	Tín hiệu cho phép đọc từ memory
MemW_i	1	Input	Tín hiệu cho phép ghi từ memory
RegWEn_i	1	Input	Tín hiệu cho phép ghi vào register
WBSel_i	2	Input	Tín hiệu chọn đường Write Back
ALUSel_i	4	Input	Lựa chọn toán tử
RegDst_o	5	Output	Giá trị rd đẩy sang thanh ghi ID/EX
ASel_o	1	Output	A select đẩy sang thanh ghi ID/EX
BSel_o	1	Output	B select đẩy sang thanh ghi ID/EX
MemR_o	1	Output	Tín hiệu cho phép đọc từ memory sang thanh ghi ID/EX
MemW_o	1	Output	Tín hiệu cho phép ghi từ memory sang thanh ghi ID/EX

RegWEn_o	1	Output	Tín hiệu cho phép ghi vào register đẩy sang thanh ghi ID/EX
WBSel_o	2	Output	Tín hiệu chọn đường Write Back đẩy sang thanh ghi ID/EX
ALUSel_o	4	Output	Lựa chọn toán tử đẩy sang thanh ghi ID/EX

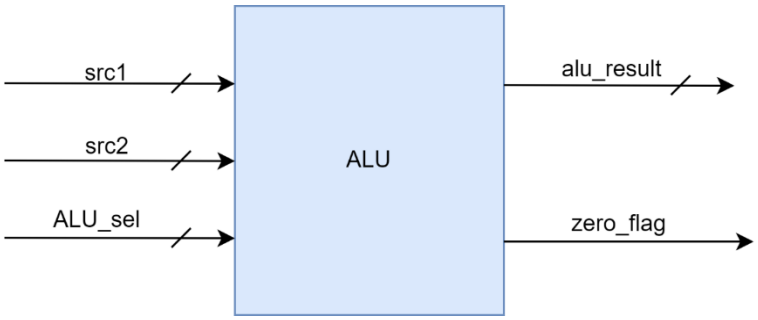
2.2.12 ALU control

Khối ALU (Arithmetic logic unit) thực hiện các chức năng tính toán số học bao gồm (add, sub, sll, slt, sltu, xor, srl, sra, or, and).

Tín hiệu ALU_Sel sẽ chọn chức năng thực hiện cho ALU, các chức năng được định nghĩa với giá trị như Bảng 2.17.

Bảng 2.17. Giá trị ứng với từng chức năng của ALU

ALUadd	4'b0000
ALUsub	4'b0001
ALUsll	4'b0010
ALUslt	4'b0011
ALUsltu	4'b0100
ALUxor	4'b0101
ALUsrl	4'b0110
ALUsra	4'b0111
ALUor	4'b1000
ALUand	4'b1001
ALUnop	4'b1111



Hình 2.19. Sơ đồ khối của khối ALU

Bảng 2.18 mô tả các tín hiệu vào ra của khối.

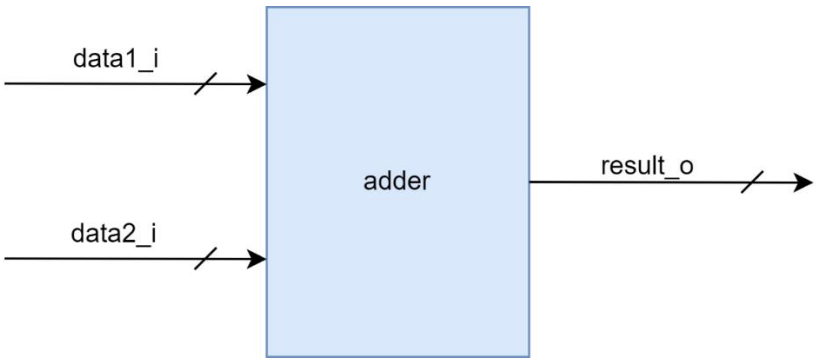
Bảng 2.18. Chân tín hiệu vào ra của khối ALU

Name	Width	Input/Output	Description
src1	32	Input	Dữ liệu nguồn 1
src2	32	Input	Dữ liệu nguồn 2
ALU_Sel	4	Itput	Lựa chọn chức năng
alu_result	32	Output	Kết quả đầu ra
zero_flag	1	Output	Cờ zero

2.2.13 Adder

Khối adder thực hiện chức năng của bộ full adder 32 bit.

```
assign result_o = data1_i + data2_i;
```



Hình 2.20. Sơ đồ khối của adder

Bảng 2.19. Chân vào ra của khối adder

Name	Width	Input/Output	Description
data1_i	32	Input	Số hạng thứ 1
data2_i	32	Input	Số hạng thứ 2
result_o	32	Output	Kết quả sau phép cộng

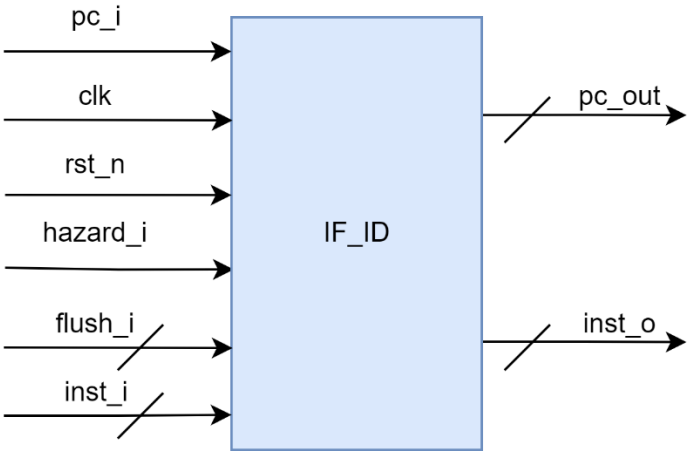
Bảng 2.19, Hình 2.20 mô tả tín hiệu vào ra và sơ đồ khối của khối adder

2.2.14 IF/ID

Khối IF/ID là 1 thanh ghi phục vụ cho pipeline (Hình 2.21).

Khi xuất hiện có hazard giá trị PC và mã lệnh sẽ được giữ nguyên và khi có tín hiệu flush mã lệnh sẽ gán bằng 0 (lệnh NOP). Khối sẽ thực hiện như sau:

```
if(hazard_i) begin
     pc_o <= pc_o;
     inst_o <= inst_o;
   end else
   if(flush_i) begin
     pc_o <= pc_i;
     inst_o <= 32'b0;
   end else begin
     pc_o <= pc_i;
     inst_o <= inst_i;
   end
```



Hình 2.21. Sơ đồ khối của khối IF/ID

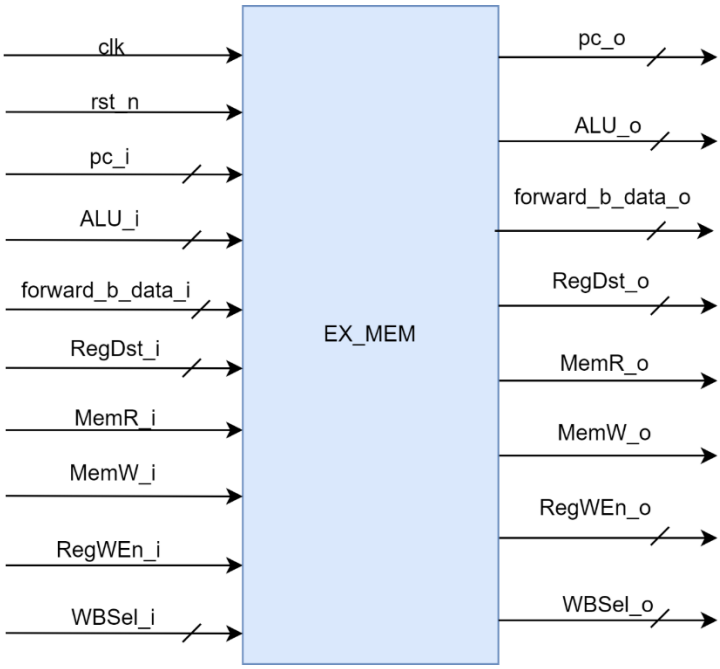
Bảng 2.20 mô tả các tín hiệu của khối IF/ID.

Bảng 2.20. Chân vào ra của khối IF/ID

Name	Width	Input/Output	Description
clk	1	Input	Tín hiệu xung đồng hồ
pc_i	32	Input	Giá trị PC ở tầng IF
rst_n	1	Input	Tín hiệu reset tích cực mức thấp
hazard_i	1	Input	Tín hiệu chỉ ra có hazard
flush_i	1	Input	Tín hiệu thông báo chèn lệnh NOP
inst_i	32	Input	Mã lệnh lưu từ tầng IF
pc_o	32	Output	Giá trị PC đẩy sang tầng ID
inst_o	32	Output	Mã lệnh đẩy sang tầng ID

2.2.15 EX_MEM

Khối EX/MEM (Hình 2.22) là thanh ghi trung gian giữa 2 tầng EX và MEM.



Hình 2.22. Sơ đồ khối của khối EX/MEM

Bảng 2.21 mô tả các tín hiệu của khối EX/MEM.

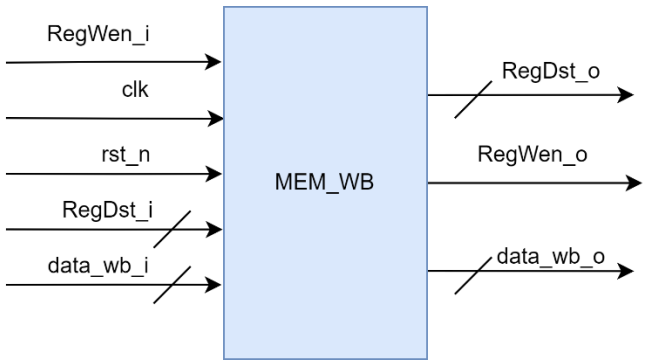
Bảng 2.21. Chân vào ra của khối EX/MEM

Name	Width	Input/Output	Description
clk	1	Input	Tín hiệu xung đồng hồ
rst_n	1	Input	Tín hiệu reset tích cực mức thấp
pc_i	32	Input	Giá trị PC ở tầng EX
ALU_i	32	Input	Giá trị được tính toán ở ALU
forward_b_data_i	32	Input	Giá trị sau bộ mux A ở tầng EX
RegDst_i	5	Input	Giá trị rd ở EX
MemR_i	1	Input	Tín hiệu cho phép đọc từ memory từ tầng EX
MemW_i	1	Input	Tín hiệu cho phép ghi từ memory từ tầng EX

RegWEn_i	1	Input	Tín hiệu cho phép ghi vào register từ tầng EX
WBSel_i	2	Input	Tín hiệu chọn đường Write Back từ tầng EX
pc_o	32	Output	Giá trị PC đẩy sang tầng MEM
ALU_o	32	Output	Giá trị kết quả ALU đẩy sang tầng MEM
forward_b_data_o	32	Output	Giá trị sau bộ mux A đẩy sang tầng MEM
RegDst_o	5	Output	Giá trị rd đẩy sang tầng MEM
MemR_o	1	Output	Tín hiệu cho phép đọc từ memory đẩy sang tầng MEM
MemW_o	1	Output	Tín hiệu cho phép ghi từ memory sang tầng MEM
RegWEn_o	1	Output	Tín hiệu cho phép ghi vào register đẩy sang tầng MEM
WBSel_o	2	Output	Tín hiệu chọn đường Write Back đẩy sang tầng MEM

2.2.16 MEM_WB

Khối MEM/WB (Hình 2.23) là thanh ghi lưu lại các giá trị dùng cho việc write back.



Hình 2.23. Sơ đồ khối của khối MEM/WB

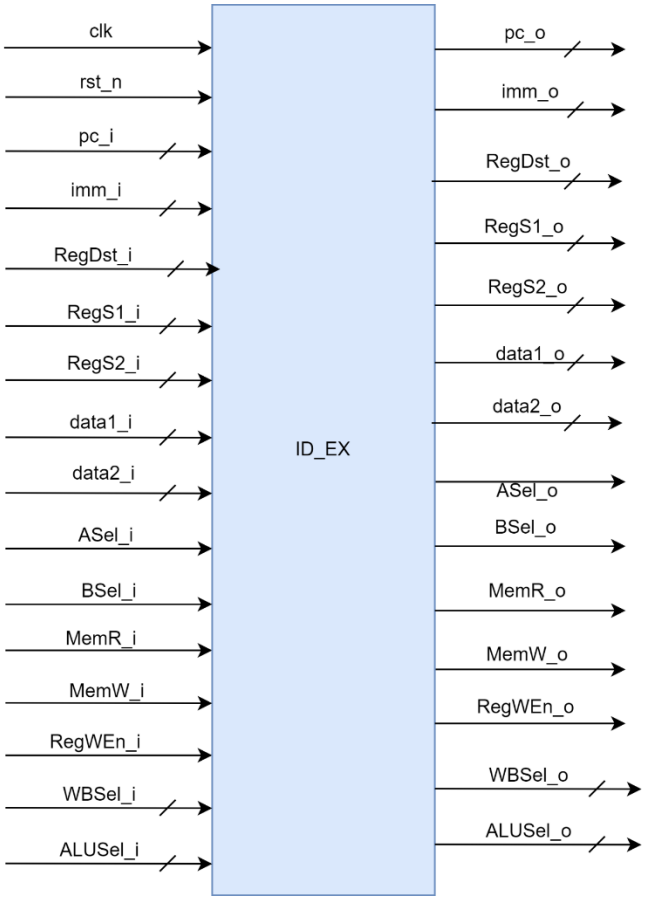
Bảng 2.22 mô tả chức năng của các tín hiệu có trong khối.

Bảng 2.22. Chân vào ra của khối MEM/WB

Name	Width	Input/ Output	Description
RegWen_i	1	Input	Tín hiệu cho phép ghi vào register từ tầng MEM
clk	1	Input	Tín hiệu xung đồng hồ
rst_n	1	Input	Tín hiệu reset tích cực mức thấp
RegDst_i	5	Input	Giá trị rd ở tầng MEM
data_wb_i	32	Input	Giá trị write back xử lý ở tầng MEM
RegDst_o	5	output	Giá trị rd đẩy sang tầng WB
RegWen_o	1	output	Tín hiệu cho phép ghi vào register đẩy sang tầng WB
data_wb_o	32	output	Giá trị write back

2.2.17 ID/EX

Khối EX/MEM (Hình 2.24) là thanh ghi trung gian giữa 2 tầng ID và EX phục vụ cho pipeline.



Hình 2.24. Sơ đồ khối của khối ID/EX

Bảng 2.23 mô tả chức năng của các tín hiệu có trong khối.

Bảng 2.23. Chân vào ra của khối ID/EX

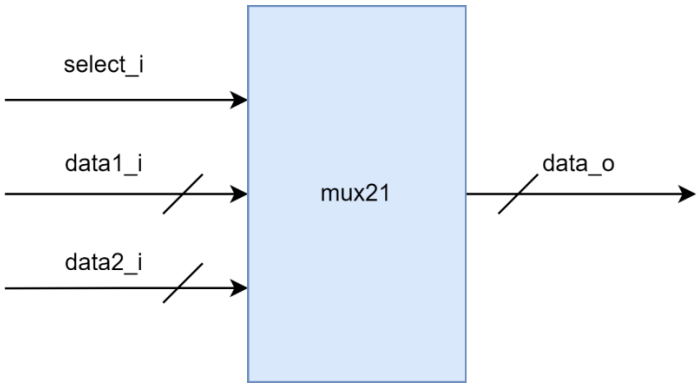
Name	Width	Input/ Output	Description
clk	1	Input	Tín hiệu xung đồng hồ
rst_n	1	Input	Tín hiệu reset tích cực ở mức thấp
pc_i	32	Input	Giá trị PC ở tầng ID
imm_i	32	Input	Giá trị immediate đã mở rộng dấu
RegDst_i	5	Input	Giá trị rd ở tầng ID
RegS1_i	5	Input	Giá trị rs1
RegS2_i	5	Input	Giá trị rs2

data1_i	32	Input	Giá trị đầu vào 1 ở tầng ID
data2_i	32	Input	Giá trị đầu vào 2 ở tầng ID
ASel_i	1	Input	A select
BSel_i	1	Input	B select
MemR_i	1	Input	Tín hiệu cho phép đọc từ memory từ tầng ID
MemW_i	1	Input	Tín hiệu cho phép ghi từ memory từ tầng ID
RegWEn_i	1	Input	Tín hiệu cho phép ghi vào register từ tầng ID
WBSel_i	2	Input	Tín hiệu chọn đường write back từ tầng ID
ALUSel_i	4	Input	Tín hiệu lựa chọn chức năng của ALU
pc_o	32	Output	Giá trị PC đẩy sang tầng EX
imm_o	32	Output	Giá trị immediate đã mở rộng dấu đẩy sang tầng EX
RegDst_o	5	Output	Giá trị rd đẩy sang tầng EX
RegS1_o	5	Output	Giá trị rs1 sang tầng EX
RegS2_o	5	Output	Giá trị rs2 sang tầng EX
data1_o	32	Output	Giá trị đầu vào 1 sang tầng EX
data2_o	32	Output	Giá trị đầu vào 1 sang tầng EX
ASel_o	1	Output	A select đẩy sang tầng EX
BSel_o	1	Output	B select đẩy sang tầng EX
MemR_o	1	Output	Tín hiệu cho phép đọc từ memory sang tầng EX
MemW_o	1	Output	Tín hiệu cho phép ghi từ memory sang tầng EX
RegWEn_o	1	Output	Tín hiệu cho phép ghi vào register đẩy sang tầng EX

WBSel_o	2	Output	Tín hiệu chọn đường Write Back đẩy sang tầng EX
ALUSel_o	4	Output	Tín hiệu lựa chọn chức năng của ALU đẩy sang tầng EX

2.2.18 MUX21

Bộ Mux này (Hình 2.25) đơn giản là gán dữ liệu ra theo một trong số các tín hiệu vào dựa vào tín hiệu select_i. Trong Verilog ta dùng toán tử ? : để rẽ nhánh các điều kiện của sel và xác định đầu ra, code của bộ Mux ở dưới hình Mux tương ứng.



Hình 2.25. Sơ đồ khối của mux21

Mô tả như sau:

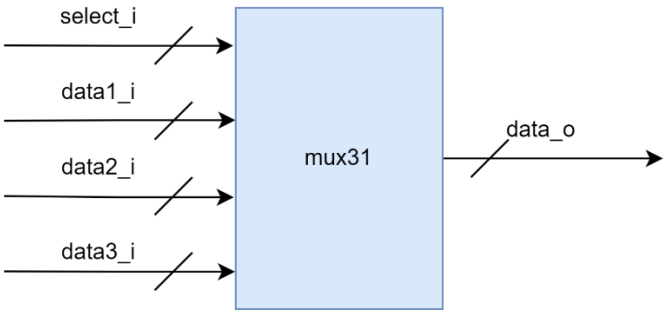
```
assign data_o = (select_i) ? data2_i : data1_i;
```

Bảng 2.24. Chân vào ra của khối mux21

Name	Width	Input/Output	Description
select_i	1	Input	Tín hiệu lựa chọn
data1_i	32	Input	Giá trị đầu vào 1
data2_i	32	Input	Giá trị đầu vào 2
data_o	32	Output	Giá trị đầu ra

2.2.19 MUX31

Bộ Mux này tương tự bộ mux21 chỉ khác là có 3 đầu vào.



Hình 2.26. Sơ đồ khối của khối mux31

Khối thực hiện như sau:

```
always @(*) begin
     case(select_i)
           2'b00: data_o = data1_i;
           2'b01: data_o = data2_i;
           2'b10: data_o = data3_i;
           default : data_o = data1_i;
     endcase
   end
```

Bảng 2.25. Chân vào ra của khối mux31

Name	Width	Input/Output	Description
select_i	1	Input	Tín hiệu lựa chọn
data1_i	32	Input	Giá trị đầu vào 1
data2_i	32	Input	Giá trị đầu vào 2
data3_i	32	Input	Giá trị đầu vào 3
data_o	32	Output	Giá trị đầu ra

CHƯƠNG 3. KIỂM THỬ (VERIFICATION)

Chương này trình bày các kết quả mô phỏng cho từng khối và toàn bộ thiết kế được triển khai bằng ngôn ngữ SystemVerilog trên phần mềm Questasim.

3.1 Test case

Các kịch bản kiểm tra:

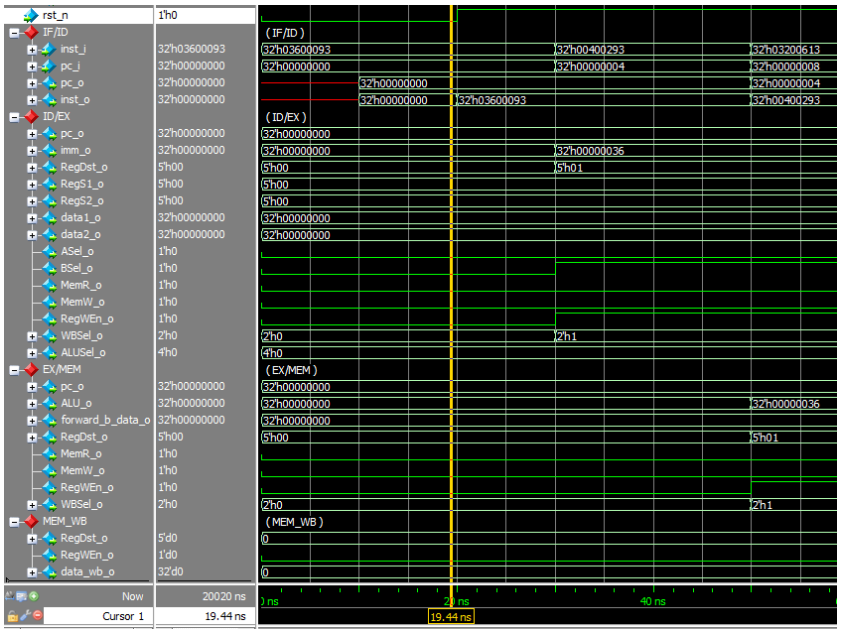
- Kiểm tra quá trình reset, đảm bảo các đầu ra được thiết lập về đúng giá trị
- Kiểm tra các trường hợp instruction không có hazard
- Kiểm tra các trường hợp instruction có data hazard
- Kiểm tra các trường hợp instruction có structural hazard
- Kiểm tra các trường hợp instruction có control hazard
- Kiểm tra lệnh JAL

3.2 Tạo đầu vào vào kiểm tra các test case

Sử dụng mã nguồn chuyển đổi từ assembly sang mã máy dạng kiến trúc RISC-V [2]. Từ đó, nạp mã máy vào instruction memory để hệ thống vận hành.

3.2.1 Quá trình reset

Hình 3.1 cho thấy quá trình reset đảm bảo các đầu ra thiết lập về đúng giá trị



Hình 3.1. Kết quả mô phỏng reset của hệ thống

3.2.2 Các trường hợp instruction không có hazard

Thực hiện nạp code sau vào hệ thống:

addi x1, x0, 54	0x03600093
addi x2, x0, 64	0x04000113
addi x5, x0, -10	0xFF600293
addi x6, x0, -30	0xFE200313
sub x11, x2, x1	0x401105B3
sw x2, 0(x7)	0x0023A023
blt x5, x6, J	0x0062C663
addi x22, x0, 1	0x00100B13
addi x23, x0, 1	0x00100B93
J: addi x22, x0, 9	0x00900B13
lw x23, 0(x7)	0x0003AB83

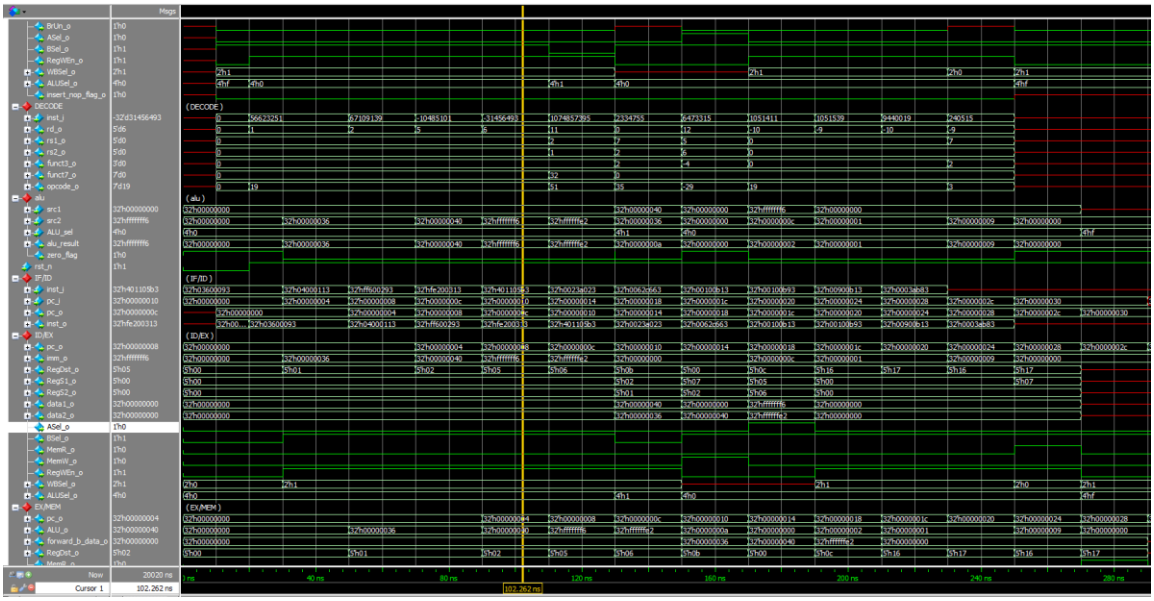
Kết quả sau mô phỏng:

```
# @ 300 ns => REGISTER
#    [ 0]: 0
#    [ 1]: 54
#    [ 2]: 64
#    [ 3]: 0
#    [ 4]: 0
#    [ 5]: -10
#    [ 6]: -30
#    [ 7]: 0
#    [ 8]: 0
#    [ 9]: 0
#    [10]: 0
#    [11]: 10
#    [12]: 0
#    [13]: 0
#    [14]: 0
#    [15]: 0
#    [16]: 0
#    [17]: 0
#    [18]: 0
#    [19]: 0
#    [20]: 0
#    [21]: 0
#    [22]: 9
#    [23]: 64
#    [24]: 0
#    [25]: 0
#    [26]: 0
#    [27]: 0
#    [28]: 0
#    [29]: 0
#    [30]: 0
#    [31]: 0
```

Hình 3.2. Giá trị của của các thanh ghi sau quá trình mô phỏng

```
# @ 190 ns => DATA MEMORY
#    [  3] = 'h00, [  2] = 'h00, [  1] = 'h00, [  0] = 'h40 => data = 64
#    [  7] = 'h00, [  6] = 'h00, [  5] = 'h00, [  4] = 'h00 => data = 0
#    [ 11] = 'h00, [ 10] = 'h00, [  9] = 'h00, [  8] = 'h00 => data = 0
#    [ 15] = 'h00, [ 14] = 'h00, [ 13] = 'h00, [ 12] = 'h00 => data = 0
#    [ 19] = 'h00, [ 18] = 'h00, [ 17] = 'h00, [ 16] = 'h00 => data = 0
#    [ 23] = 'h00, [ 22] = 'h00, [ 21] = 'h00, [ 20] = 'h00 => data = 0
#    [ 27] = 'h00, [ 26] = 'h00, [ 25] = 'h00, [ 24] = 'h00 => data = 0
#    [ 31] = 'h00, [ 30] = 'h00, [ 29] = 'h00, [ 28] = 'h00 => data = 0
#    [ 35] = 'h00, [ 34] = 'h00, [ 33] = 'h00, [ 32] = 'h00 => data = 0
#    [ 39] = 'h00, [ 38] = 'h00, [ 37] = 'h00, [ 36] = 'h00 => data = 0
#    [ 43] = 'h00, [ 42] = 'h00, [ 41] = 'h00, [ 40] = 'h00 => data = 0
#    [ 47] = 'h00, [ 46] = 'h00, [ 45] = 'h00, [ 44] = 'h00 => data = 0
#    [ 51] = 'h00, [ 50] = 'h00, [ 49] = 'h00, [ 48] = 'h00 => data = 0
#    [ 55] = 'h00, [ 54] = 'h00, [ 53] = 'h00, [ 52] = 'h00 => data = 0
#    [ 59] = 'h00, [ 58] = 'h00, [ 57] = 'h00, [ 56] = 'h00 => data = 0
#    [ 63] = 'h00, [ 62] = 'h00, [ 61] = 'h00, [ 60] = 'h00 => data = 0
#    [ 67] = 'h00, [ 66] = 'h00, [ 65] = 'h00, [ 64] = 'h00 => data = 0
#    [ 71] = 'h00, [ 70] = 'h00, [ 69] = 'h00, [ 68] = 'h00 => data = 0
#    [ 75] = 'h00, [ 74] = 'h00, [ 73] = 'h00, [ 72] = 'h00 => data = 0
#    [ 79] = 'h00, [ 78] = 'h00, [ 77] = 'h00, [ 76] = 'h00 => data = 0
#    [ 83] = 'h00, [ 82] = 'h00, [ 81] = 'h00, [ 80] = 'h00 => data = 0
#    [ 87] = 'h00, [ 86] = 'h00, [ 85] = 'h00, [ 84] = 'h00 => data = 0
#    [ 91] = 'h00, [ 90] = 'h00, [ 89] = 'h00, [ 88] = 'h00 => data = 0
#    [ 95] = 'h00, [ 94] = 'h00, [ 93] = 'h00, [ 92] = 'h00 => data = 0
#    [ 99] = 'h00, [ 98] = 'h00, [ 97] = 'h00, [ 96] = 'h00 => data = 0
#    [103] = 'h00, [102] = 'h00, [101] = 'h00, [100] = 'h00 => data = 0
#    [107] = 'h00, [106] = 'h00, [105] = 'h00, [104] = 'h00 => data = 0
#    [111] = 'h00, [110] = 'h00, [109] = 'h00, [108] = 'h00 => data = 0
#    [115] = 'h00, [114] = 'h00, [113] = 'h00, [112] = 'h00 => data = 0
#    [119] = 'h00, [118] = 'h00, [117] = 'h00, [116] = 'h00 => data = 0
```

Hình 3.3. Giá trị của các ô nhớ trong data memory sau quá trình mô phỏng



Hình 3.4. Wave của quá trình mô phỏng

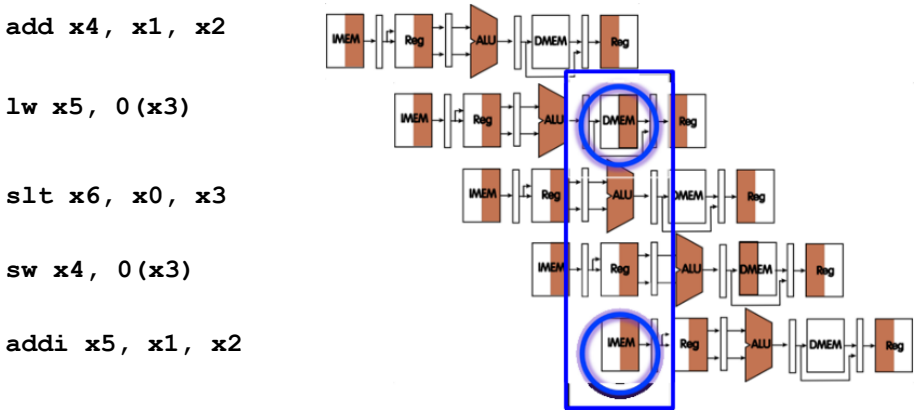
→ Kết luận: Từ kết quả ở Hình 3.2 Hình 3.3 Hình 3.4, cho thấy hệ thống hoạt động đúng với chức năng.

3.2.3 Trường hợp instruction có structural hazard

Đồng thời ghi và đọc từ memory → tách thành 2 khối memory

Đồng thời đọc ghi từ register → đọc ở sườn dương, ghi ở sườn âm

addi x1, x0, 54	03600093
addi x2, x0, 64	04000113
sw x2, 0(x3)	0021A023
add x4, x1, x2	00208233
lw x5, 0(x3)	0001A283
slt x6, x0, x3	00302333
sw x4, 0(x3)	0041A023
add x5, x1, x2	002082B3

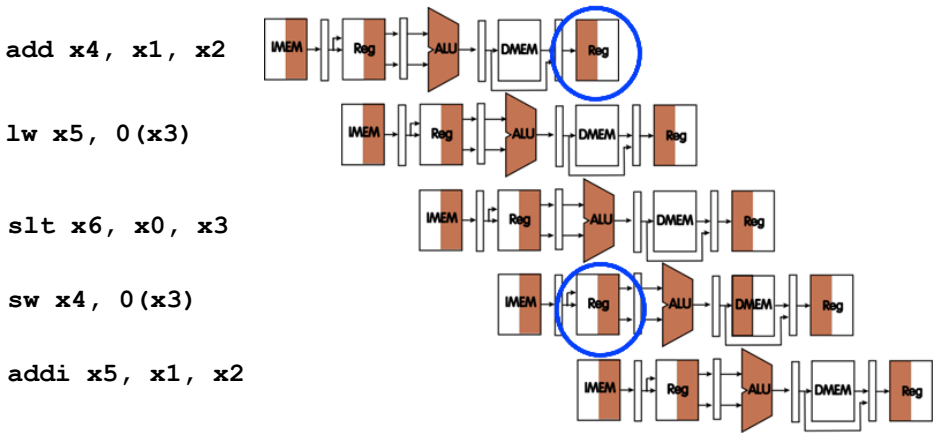


Hình 3.5. Mô tả cùng ghi vào memory

```
# @ 180 ns => REGISTER
#    [ 0]: 0
#    [ 1]: 54
#    [ 2]: 64
#    [ 3]: 0
#    [ 4]: 118
#    [ 5]: 64
#    [ 6]: 0
#    [ 7]: 0
#    [ 8]: 0
#    [ 9]: 0
#    [10]: 0
#    [11]: 0
#    [12]: 0
#    [13]: 0
#    [14]: 0
#    [15]: 0
#    [16]: 0
#    [17]: 0
#    [18]: 0
#    [19]: 0
#    [20]: 0
#    [21]: 0
#    [22]: 0
#    [23]: 0
#    [24]: 0
#    [25]: 0
#    [26]: 0
#    [27]: 0
#    [28]: 0
#    [29]: 0
#    [30]: 0
#    [31]: 0
#
```

```
# @ 240 ns => REGISTER
#    [ 0]: 0
#    [ 1]: 54
#    [ 2]: 64
#    [ 3]: 0
#    [ 4]: 118
#    [ 5]: 118
#    [ 6]: 0
#    [ 7]: 0
#    [ 8]: 0
#    [ 9]: 0
#    [10]: 0
#    [11]: 0
#    [12]: 0
#    [13]: 0
#    [14]: 0
#    [15]: 0
#    [16]: 0
#    [17]: 0
#    [18]: 0
#    [19]: 0
#    [20]: 0
#    [21]: 0
#    [22]: 0
#    [23]: 0
#    [24]: 0
#    [25]: 0
#    [26]: 0
#    [27]: 0
#    [28]: 0
#    [29]: 0
#    [30]: 0
#    [31]: 0
```

Hình 3.6. Kết quả của các thanh ghi ở 2 thời điểm



Hình 3.7. Mô tả cùng đọc ghi vào register

```
# @ 240 ns => REGISTER
#    [ 0]: 0
#    [ 1]: 54
#    [ 2]: 64
#    [ 3]: 0
#    [ 4]: 118
#    [ 5]: 118
#    [ 6]: 0
#    [ 7]: 0
#    [ 8]: 0
#    [ 9]: 0
#    [10]: 0
#    [11]: 0
#    [12]: 0
#    [13]: 0
#    [14]: 0
#    [15]: 0
#    [16]: 0
#    [17]: 0
#    [18]: 0
#    [19]: 0
#    [20]: 0
#    [21]: 0
#    [22]: 0
#    [23]: 0
#    [24]: 0
#    [25]: 0
#    [26]: 0
#    [27]: 0
#    [28]: 0
#    [29]: 0
#    [30]: 0
#    [31]: 0
```

```
# @ 210 ns => DATA MEMORY
#    [  3] = 'h00, [  2] = 'h00, [  1] = 'h00, [  0] = 'h76 => data = 118
#    [  7] = 'h00, [  6] = 'h00, [  5] = 'h00, [  4] = 'h00 => data = 0
#    [ 11] = 'h00, [ 10] = 'h00, [  9] = 'h00, [  8] = 'h00 => data = 0
#    [ 15] = 'h00, [ 14] = 'h00, [ 13] = 'h00, [ 12] = 'h00 => data = 0
#    [ 19] = 'h00, [ 18] = 'h00, [ 17] = 'h00, [ 16] = 'h00 => data = 0
#    [ 23] = 'h00, [ 22] = 'h00, [ 21] = 'h00, [ 20] = 'h00 => data = 0
#    [ 27] = 'h00, [ 26] = 'h00, [ 25] = 'h00, [ 24] = 'h00 => data = 0
#    [ 31] = 'h00, [ 30] = 'h00, [ 29] = 'h00, [ 28] = 'h00 => data = 0
#    [ 35] = 'h00, [ 34] = 'h00, [ 33] = 'h00, [ 32] = 'h00 => data = 0
#    [ 39] = 'h00, [ 38] = 'h00, [ 37] = 'h00, [ 36] = 'h00 => data = 0
#    [ 43] = 'h00, [ 42] = 'h00, [ 41] = 'h00, [ 40] = 'h00 => data = 0
#    [ 47] = 'h00, [ 46] = 'h00, [ 45] = 'h00, [ 44] = 'h00 => data = 0
#    [ 51] = 'h00, [ 50] = 'h00, [ 49] = 'h00, [ 48] = 'h00 => data = 0
#    [ 55] = 'h00, [ 54] = 'h00, [ 53] = 'h00, [ 52] = 'h00 => data = 0
#    [ 59] = 'h00, [ 58] = 'h00, [ 57] = 'h00, [ 56] = 'h00 => data = 0
#    [ 63] = 'h00, [ 62] = 'h00, [ 61] = 'h00, [ 60] = 'h00 => data = 0
#    [ 67] = 'h00, [ 66] = 'h00, [ 65] = 'h00, [ 64] = 'h00 => data = 0
#    [ 71] = 'h00, [ 70] = 'h00, [ 69] = 'h00, [ 68] = 'h00 => data = 0
#    [ 75] = 'h00, [ 74] = 'h00, [ 73] = 'h00, [ 72] = 'h00 => data = 0
#    [ 79] = 'h00, [ 78] = 'h00, [ 77] = 'h00, [ 76] = 'h00 => data = 0
#    [ 83] = 'h00, [ 82] = 'h00, [ 81] = 'h00, [ 80] = 'h00 => data = 0
#    [ 87] = 'h00, [ 86] = 'h00, [ 85] = 'h00, [ 84] = 'h00 => data = 0
#    [ 91] = 'h00, [ 90] = 'h00, [ 89] = 'h00, [ 88] = 'h00 => data = 0
#    [ 95] = 'h00, [ 94] = 'h00, [ 93] = 'h00, [ 92] = 'h00 => data = 0
#    [ 99] = 'h00, [ 98] = 'h00, [ 97] = 'h00, [ 96] = 'h00 => data = 0
#    [103] = 'h00, [102] = 'h00, [101] = 'h00, [100] = 'h00 => data = 0
#    [107] = 'h00, [106] = 'h00, [105] = 'h00, [104] = 'h00 => data = 0
#    [111] = 'h00, [110] = 'h00, [109] = 'h00, [108] = 'h00 => data = 0
#    [115] = 'h00, [114] = 'h00, [113] = 'h00, [112] = 'h00 => data = 0
#    [119] = 'h00, [118] = 'h00, [117] = 'h00, [116] = 'h00 => data = 0
#
```

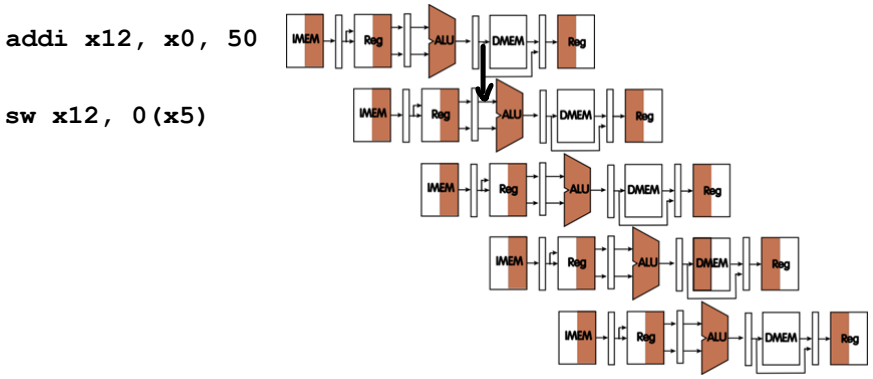
Hình 3.8. Kết quả của thanh ghi và data memory ở 2 thời điểm

→ Kết luận: Hình 3.6, Hình 3.8 cho thấy, hệ thống hoạt động đúng.

3.2.4 Trường hợp instruction có data hazard

EX Haxzard:

`addi x5, x0, 4`	`00400293`
`addi x12, x0, 50`	`03200613`
`sw  x12, 0(x5)`	`00C2A023`
`addi x12, x0, -30`	`FE200613`
`sw  x12, 4(x5)`	`00C2A223`



Hình 3.9. Mô phỏng quá trình EX haxzard

```
# @ 120 ns => REGISTER
#    [ 0]: 0
#    [ 1]: 0
#    [ 2]: 0
#    [ 3]: 0
#    [ 4]: 0
#    [ 5]: 4
#    [ 6]: 0
#    [ 7]: 0
#    [ 8]: 0
#    [ 9]: 0
#    [10]: 0
#    [11]: 0
#    [12]: 50
#    [13]: 0
#    [14]: 0
#    [15]: 0
#    [16]: 0
#    [17]: 0
#    [18]: 0
#    [19]: 0
#    [20]: 0
#    [21]: 0
#    [22]: 0
#    [23]: 0
#    [24]: 0
#    [25]: 0
#    [26]: 0
#    [27]: 0
#    [28]: 0
#    [29]: 0
#    [30]: 0
#    [31]: 0
```

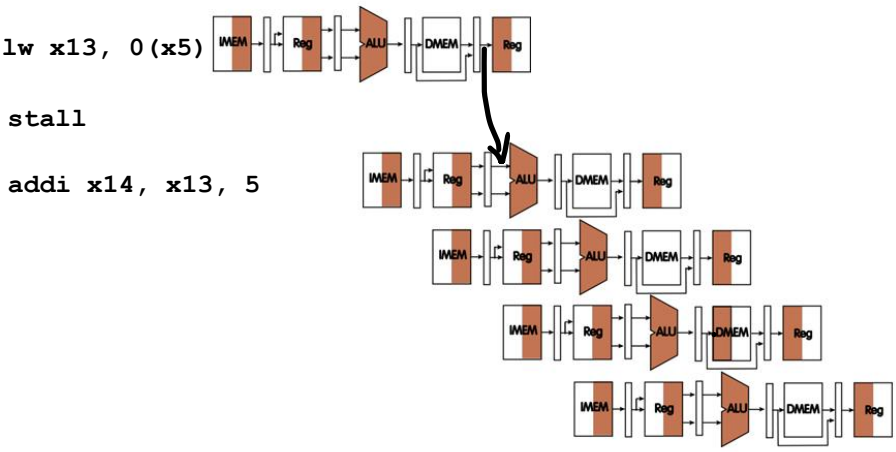
```
# @ 160 ns => REGISTER
#    [ 0]: 0
#    [ 1]: 0
#    [ 2]: 0
#    [ 3]: 0
#    [ 4]: 0
#    [ 5]: 4
#    [ 6]: 0
#    [ 7]: 0
#    [ 8]: 0
#    [ 9]: 0
#    [10]: 0
#    [11]: 0
#    [12]: -30
#    [13]: 0
#    [14]: 0
#    [15]: 0
#    [16]: 0
#    [17]: 0
#    [18]: 0
#    [19]: 0
#    [20]: 0
#    [21]: 0
#    [22]: 0
#    [23]: 0
#    [24]: 0
#    [25]: 0
#    [26]: 0
#    [27]: 0
#    [28]: 0
#    [29]: 0
#    [30]: 0
#    [31]: 0
```

```
# @ 170 ns => DATA MEMORY
#    [  3] = 'h00, [  2] = 'h00, [  1] = 'h00, [  0] = 'h00 => data = 0
#    [  7] = 'h00, [  6] = 'h00, [  5] = 'h00, [  4] = 'h32 => data = 50
#    [ 11] = 'hff, [ 10] = 'hff, [  9] = 'hff, [  8] = 'he2 => data = -30
#    [ 15] = 'h00, [ 14] = 'h00, [ 13] = 'h00, [ 12] = 'h00 => data = 0
#    [ 19] = 'h00, [ 18] = 'h00, [ 17] = 'h00, [ 16] = 'h00 => data = 0
#    [ 23] = 'h00, [ 22] = 'h00, [ 21] = 'h00, [ 20] = 'h00 => data = 0
#    [ 27] = 'h00, [ 26] = 'h00, [ 25] = 'h00, [ 24] = 'h00 => data = 0
#    [ 31] = 'h00, [ 30] = 'h00, [ 29] = 'h00, [ 28] = 'h00 => data = 0
#    [ 35] = 'h00, [ 34] = 'h00, [ 33] = 'h00, [ 32] = 'h00 => data = 0
#    [ 39] = 'h00, [ 38] = 'h00, [ 37] = 'h00, [ 36] = 'h00 => data = 0
#    [ 43] = 'h00, [ 42] = 'h00, [ 41] = 'h00, [ 40] = 'h00 => data = 0
#    [ 47] = 'h00, [ 46] = 'h00, [ 45] = 'h00, [ 44] = 'h00 => data = 0
#    [ 51] = 'h00, [ 50] = 'h00, [ 49] = 'h00, [ 48] = 'h00 => data = 0
#    [ 55] = 'h00, [ 54] = 'h00, [ 53] = 'h00, [ 52] = 'h00 => data = 0
#    [ 59] = 'h00, [ 58] = 'h00, [ 57] = 'h00, [ 56] = 'h00 => data = 0
#    [ 63] = 'h00, [ 62] = 'h00, [ 61] = 'h00, [ 60] = 'h00 => data = 0
#    [ 67] = 'h00, [ 66] = 'h00, [ 65] = 'h00, [ 64] = 'h00 => data = 0
#    [ 71] = 'h00, [ 70] = 'h00, [ 69] = 'h00, [ 68] = 'h00 => data = 0
#    [ 75] = 'h00, [ 74] = 'h00, [ 73] = 'h00, [ 72] = 'h00 => data = 0
#    [ 79] = 'h00, [ 78] = 'h00, [ 77] = 'h00, [ 76] = 'h00 => data = 0
#    [ 83] = 'h00, [ 82] = 'h00, [ 81] = 'h00, [ 80] = 'h00 => data = 0
#    [ 87] = 'h00, [ 86] = 'h00, [ 85] = 'h00, [ 84] = 'h00 => data = 0
#    [ 91] = 'h00, [ 90] = 'h00, [ 89] = 'h00, [ 88] = 'h00 => data = 0
#    [ 95] = 'h00, [ 94] = 'h00, [ 93] = 'h00, [ 92] = 'h00 => data = 0
#    [ 99] = 'h00, [ 98] = 'h00, [ 97] = 'h00, [ 96] = 'h00 => data = 0
#    [103] = 'h00, [102] = 'h00, [101] = 'h00, [100] = 'h00 => data = 0
#    [107] = 'h00, [106] = 'h00, [105] = 'h00, [104] = 'h00 => data = 0
#    [111] = 'h00, [110] = 'h00, [109] = 'h00, [108] = 'h00 => data = 0
#    [115] = 'h00, [114] = 'h00, [113] = 'h00, [112] = 'h00 => data = 0
#    [119] = 'h00, [118] = 'h00, [117] = 'h00, [116] = 'h00 => data = 0
```

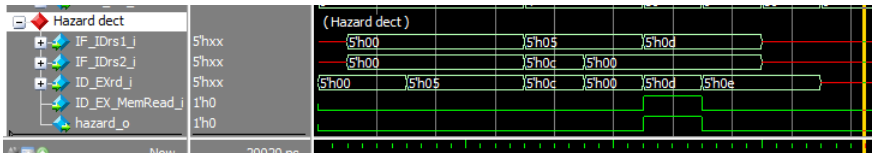
Hình 3.10. Kết quả của các thanh ghi và data memory ở 3 thời điểm

MEM Hazard:

addi x5, x0, 4	00400293
addi x12, x0, 50	03200613
sw x12, 0(x5)	00C2A023
lw x13, 0(x5)	0002A683
addi x14, x13, 5	00560713



Hình 3.11. Mô phỏng quá trình MEM hazard



Hình 3.12. Phát hiện có hazard

```
# @ 200 ns => REGISTER
#    [ 0]: 0
#    [ 1]: 0
#    [ 2]: 0
#    [ 3]: 0
#    [ 4]: 0
#    [ 5]: 4
#    [ 6]: 0
#    [ 7]: 0
#    [ 8]: 0
#    [ 9]: 0
#    [10]: 0
#    [11]: 0
#    [12]: 50
#    [13]: 50
#    [14]: 55
#    [15]: 0
#    [16]: 0
#    [17]: 0
#    [18]: 0
#    [19]: 0
#    [20]: 0
#    [21]: 0
#    [22]: 0
#    [23]: 0
#    [24]: 0
#    [25]: 0
#    [26]: 0
#    [27]: 0
#    [28]: 0
#    [29]: 0
#    [30]: 0
#    [31]: 0
```

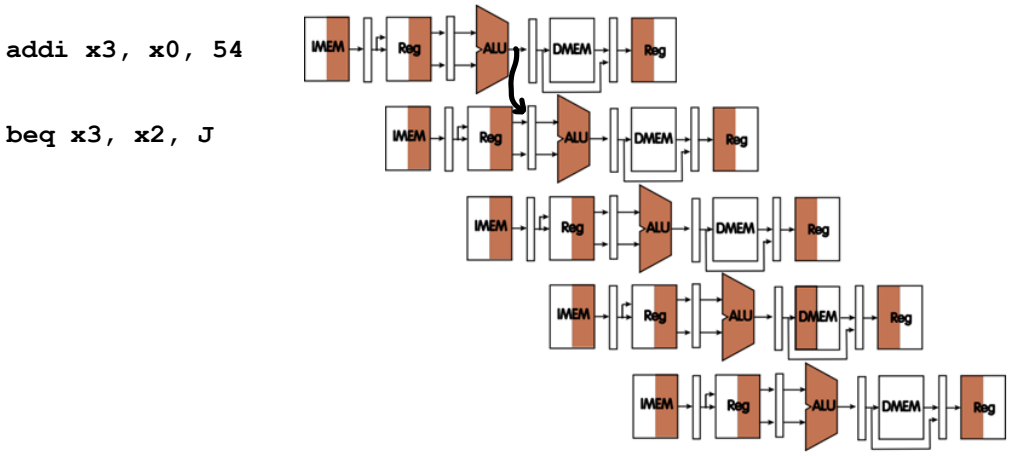
Hình 3.13. Kết quả của MEM hazard

Kết luận: Hình 3.12, Hình 3.13 cho thấy, hệ thống hoạt động đúng khi xuất hiện data hazard.

3.2.5 Trường hợp instruction có control hazard

Trước lệnh loại B là lệnh số học:

`addi x2, x0, 54`	`03600113`
`addi x3, x0, 54`	`03600193`
`beq  x3, x2, J`	`00218463`
`addi  x4, x0, 3`	`00300213`
`J: addi  x4, x0, 4`	`00400213`



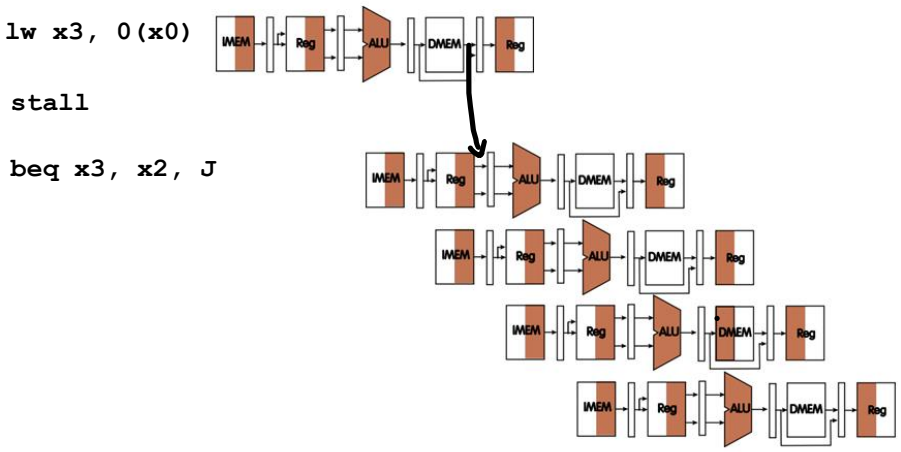
Hình 3.14 Quá trình forward đối với trường hợp hazard cho lệnh loại B

```
# @ 180 ns => REGISTER
#    [ 0]: 0
#    [ 1]: 0
#    [ 2]: 54
#    [ 3]: 54
#    [ 4]: 4
#    [ 5]: 0
#    [ 6]: 0
#    [ 7]: 0
#    [ 8]: 0
#    [ 9]: 0
#    [10]: 0
#    [11]: 0
#    [12]: 0
#    [13]: 0
#    [14]: 0
#    [15]: 0
#    [16]: 0
#    [17]: 0
#    [18]: 0
#    [19]: 0
#    [20]: 0
#    [21]: 0
#    [22]: 0
#    [23]: 0
#    [24]: 0
#    [25]: 0
#    [26]: 0
#    [27]: 0
#    [28]: 0
#    [29]: 0
#    [30]: 0
#    [31]: 0
```

Hình 3.15. Kết quả sau quá trình mô phỏng

Trước lệnh loại B là lệnh load:

addi x2, x0, 54	03600113
sw x2, 0(x0)	00202023
lw x3, 0(x0)	00002183
beq x3, x2, J	00218463
addi x4, x0, 3	00300213
J: addi x4, x0, 4	00400213



Hình 3.16. Quá trình forward đối với trường hợp hazard cho lệnh loại B

```
# @ 220 ns => REGISTER
#    [ 0]: 0
#    [ 1]: 0
#    [ 2]: 54
#    [ 3]: 54
#    [ 4]: 4
#    [ 5]: 0
#    [ 6]: 0
#    [ 7]: 0
#    [ 8]: 0
#    [ 9]: 0
#    [10]: 0
#    [11]: 0
#    [12]: 0
#    [13]: 0
#    [14]: 0
#    [15]: 0
#    [16]: 0
#    [17]: 0
#    [18]: 0
#    [19]: 0
#    [20]: 0
#    [21]: 0
#    [22]: 0
#    [23]: 0
#    [24]: 0
#    [25]: 0
#    [26]: 0
#    [27]: 0
#    [28]: 0
#    [29]: 0
#    [30]: 0
#    [31]: 0
```

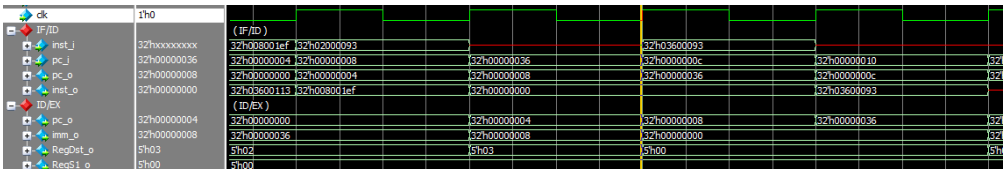
Hình 3.17. Kết quả của quá trình mô phỏng

Kết luận: Hình 3.15, Hình 3.17 cho thấy hệ thống hoạt động đúng chức năng

3.2.6 Kiểm tra lệnh JAL

Đối với lệnh JAL, hệ thống sẽ chèn 2 lệnh NOP giữa 2 lệnh hazard (Hình 3.18).

addi x2, x0, 54	03600113
jal x3, A	008001EF
addi x1, x0, 32	02000093
A:addi x1,x0, 54	03600093



Hình 3.18. Mô tả chèn 2 lệnh NOP trên questasim

```
# @ 180 ns => REGISTER
#    [ 0]: 0
#    [ 1]: 54
#    [ 2]: 54
#    [ 3]: 8
#    [ 4]: 0
#    [ 5]: 0
#    [ 6]: 0
#    [ 7]: 0
#    [ 8]: 0
#    [ 9]: 0
#    [10]: 0
#    [11]: 0
#    [12]: 0
#    [13]: 0
#    [14]: 0
#    [15]: 0
#    [16]: 0
#    [17]: 0
#    [18]: 0
#    [19]: 0
#    [20]: 0
#    [21]: 0
#    [22]: 0
#    [23]: 0
#    [24]: 0
#    [25]: 0
#    [26]: 0
#    [27]: 0
#    [28]: 0
#    [29]: 0
#    [30]: 0
#    [31]: 0
```

Hình 3.19. Kết quả sau mô phỏng

Kết luận: Hình 3.19 cho thấy hệ thống hoạt động đúng với lệnh JAL.

CHƯƠNG 4. MÔ PHỎNG CHẠY THUẬT TOÁN SẮP XẾP NỔI BỌT

Chương này trình bày, mô phỏng thuật toán sắp xếp nổi bọt chạy trên hệ thống đã thiết kế ở CHƯƠNG 2.

4.1 Triển khai thuật toán nổi bọt bằng assembly và chuyển sang ngôn ngữ máy

Thuật toán sắp xếp nổi bọt viết bằng assembly:

```
# bubble sort
addi x5, x0, 4                          #x3 base address
addi x12, x0, 50
sw   x12, 0(x5)
addi x12, x0, -30
sw   x12, 4(x5)
addi x12, x0, 70
sw   x12, 8(x5)
addi x12, x0, 1
sw   x12, 12(x5)
addi x12, x0, 3
sw   x12, 16(x5)
addi x12, x0, 56
sw   x12, 20(x5)
addi x12, x0, 12
sw   x12, 24(x5)
addi x12, x0, -85
sw   x12, 28(x5)
addi x12, x0, -1
sw   x12, 32(x5)
addi x12, x0, 17
sw   x12, 36(x5)
addi x12, x0, 18
sw   x12, 40(x5)
```

```
addi x12, x0, -47
sw   x12, 44(x5)
addi x12, x0, 78
sw   x12, 48(x5)

addi x1, x0, 0

addi x2, x0, 11

outer_loop_start:
  add x10, x0, x5
  lw x7, 0(x10)
  sub x11, x2, x1 # Inner loop count: outer loop count - sorted elements
inner_loop_start:
  lw x4, 4(x10)
  blt x7,x4, noswap
  sw x4, 0(x10)
  sw x7, 4(x10)
noswap:

  slt x6, x7, x4

  beq x6, x0, skip

  add x7, x4, x0
skip:

  addi x10, x10, 4  #Increment the address to next element

  addi x11, x11, -1

  bne x11, x0, inner_loop_start

inner_loop_end:

addi x1, x1, 1
```

```
blt x1, x2, outer_loop_start
outer_loop_end:
```

Dùng [2] để chuyển mã nguồn trên sang mã máy:

```
0x00400293
0x03200613
0x00C2A023
0xFE200613
0x00C2A223
0x04600613
0x00C2A423
0x00100613
0x00C2A623
0x00300613
0x00C2A823
0x03800613
0x00C2AA23
0x00C00613
0x00C2AC23
0xFAB00613
0x00C2AE23
0xFFF00613
0x02C2A023
0x01100613
0x02C2A223
0x01200613
0x02C2A423
0xFD100613
0x02C2A623
0x04E00613
0x02C2A823
0x00000093
0x00B00113
0x00500533
0x00052383
```

```
0x401105B3
0x00452203
0x0043C663
0x00452023
0x00752223
0x0043A333
0x00030463
0x000203B3
0x00450513
0xFFF58593
0xFC059EE3
0x00108093
0xFC20C4E3
```

4.2 Nạp mã lệnh vào hệ thống và mô phỏng

Ở thời điểm bắt đầu, tiến hành nạp mã lệnh vào instrution memory (Hình 4.1).

```
# Instruction Memory:
# [          0]: 'h00400293
# [          1]: 'h03200613
# [          2]: 'h00c2a023
# [          3]: 'hfe200613
# [          4]: 'h00c2a223
# [          5]: 'h04600613
# [          6]: 'h00c2a423
# [          7]: 'h00100613
# [          8]: 'h00c2a623
# [          9]: 'h00300613
# [         10]: 'h00c2a823
# [         11]: 'h03800613
# [         12]: 'h00c2aa23
# [         13]: 'h00c00613
# [         14]: 'h00c2ac23
# [         15]: 'hfab00613
# [         16]: 'h00c2ae23
# [         17]: 'hfff00613
# [         18]: 'h02c2a023
# [         19]: 'h01100613
# [         20]: 'h02c2a223
# [         21]: 'h01200613
# [         22]: 'h02c2a423
# [         23]: 'hfd100613
# [         24]: 'h02c2a623
# [         25]: 'h04e00613
# [         26]: 'h02c2a823
# [         27]: 'h00000093
# [         28]: 'h00b00113
# [         29]: 'h00500533
# [         30]: 'h00052383
# [         31]: 'h401105b3
# [         32]: 'h00452203
# [         33]: 'h0043c663
# [         34]: 'h00452023
# [         35]: 'h00752223
# [         36]: 'h0043a333
# [         37]: 'h00030463
# [         38]: 'h000203b3
# [         39]: 'h00450513
# [         40]: 'hfff58593
# [         41]: 'hfc059ee3
# [         42]: 'h00108093
# [         43]: 'hfc20c4e3
# [         44]: 'hxxxxxxxx
# [         45]: 'hxxxxxxxx
# [         46]: 'hxxxxxxxx
# [         47]: 'hxxxxxxxx
# [         48]: 'hxxxxxxxx
# [         49]: 'hxxxxxxxx
#
```

Hình 4.1. Khởi tạo instruction memory

Tiến hành đọc giá trị trong data memory khi có giá trị thay đổi.

```
# @ 610 ns => DATA MEMORY
#    [  3] = 'h00, [  2] = 'h00, [  1] = 'h00, [  0] = 'h00 => data = 0
#    [  7] = 'h00, [  6] = 'h00, [  5] = 'h00, [  4] = 'h32 => data = 50
#    [ 11] = 'hff, [ 10] = 'hff, [  9] = 'hff, [  8] = 'he2 => data = -30
#    [ 15] = 'h00, [ 14] = 'h00, [ 13] = 'h00, [ 12] = 'h46 => data = 70
#    [ 19] = 'h00, [ 18] = 'h00, [ 17] = 'h00, [ 16] = 'h01 => data = 1
#    [ 23] = 'h00, [ 22] = 'h00, [ 21] = 'h00, [ 20] = 'h03 => data = 3
#    [ 27] = 'h00, [ 26] = 'h00, [ 25] = 'h00, [ 24] = 'h38 => data = 56
#    [ 31] = 'h00, [ 30] = 'h00, [ 29] = 'h00, [ 28] = 'h0c => data = 12
#    [ 35] = 'hff, [ 34] = 'hff, [ 33] = 'hff, [ 32] = 'hab => data = -85
#    [ 39] = 'hff, [ 38] = 'hff, [ 37] = 'hff, [ 36] = 'hff => data = -1
#    [ 43] = 'h00, [ 42] = 'h00, [ 41] = 'h00, [ 40] = 'h11 => data = 17
#    [ 47] = 'h00, [ 46] = 'h00, [ 45] = 'h00, [ 44] = 'h12 => data = 18
#    [ 51] = 'hff, [ 50] = 'hff, [ 49] = 'hff, [ 48] = 'hd1 => data = -47
#    [ 55] = 'h00, [ 54] = 'h00, [ 53] = 'h00, [ 52] = 'h4e => data = 78
#    [ 59] = 'h00, [ 58] = 'h00, [ 57] = 'h00, [ 56] = 'h00 => data = 0
#    [ 63] = 'h00, [ 62] = 'h00, [ 61] = 'h00, [ 60] = 'h00 => data = 0
#    [ 67] = 'h00, [ 66] = 'h00, [ 65] = 'h00, [ 64] = 'h00 => data = 0
#    [ 71] = 'h00, [ 70] = 'h00, [ 69] = 'h00, [ 68] = 'h00 => data = 0
#    [ 75] = 'h00, [ 74] = 'h00, [ 73] = 'h00, [ 72] = 'h00 => data = 0
#    [ 79] = 'h00, [ 78] = 'h00, [ 77] = 'h00, [ 76] = 'h00 => data = 0
#    [ 83] = 'h00, [ 82] = 'h00, [ 81] = 'h00, [ 80] = 'h00 => data = 0
#    [ 87] = 'h00, [ 86] = 'h00, [ 85] = 'h00, [ 84] = 'h00 => data = 0
#    [ 91] = 'h00, [ 90] = 'h00, [ 89] = 'h00, [ 88] = 'h00 => data = 0
#    [ 95] = 'h00, [ 94] = 'h00, [ 93] = 'h00, [ 92] = 'h00 => data = 0
#    [ 99] = 'h00, [ 98] = 'h00, [ 97] = 'h00, [ 96] = 'h00 => data = 0
#    [103] = 'h00, [102] = 'h00, [101] = 'h00, [100] = 'h00 => data = 0
#    [107] = 'h00, [106] = 'h00, [105] = 'h00, [104] = 'h00 => data = 0
#    [111] = 'h00, [110] = 'h00, [109] = 'h00, [108] = 'h00 => data = 0
#    [115] = 'h00, [114] = 'h00, [113] = 'h00, [112] = 'h00 => data = 0
#    [119] = 'h00, [118] = 'h00, [117] = 'h00, [116] = 'h00 => data = 0
```

Hình 4.2. Dữ liệu trong data memory ở thời điểm thực hiện xong các lệnh load dữ liệu khởi tạo

```
# @ 18510 ns => DATA MEMORY
#    [  3] = 'h00, [  2] = 'h00, [  1] = 'h00, [  0] = 'h00 => data = 0
#    [  7] = 'hff, [  6] = 'hff, [  5] = 'hff, [  4] = 'hab => data = -85
#    [ 11] = 'hff, [ 10] = 'hff, [  9] = 'hff, [  8] = 'hd1 => data = -47
#    [ 15] = 'hff, [ 14] = 'hff, [ 13] = 'hff, [ 12] = 'he2 => data = -30
#    [ 19] = 'hff, [ 18] = 'hff, [ 17] = 'hff, [ 16] = 'hff => data = -1
#    [ 23] = 'h00, [ 22] = 'h00, [ 21] = 'h00, [ 20] = 'h01 => data = 1
#    [ 27] = 'h00, [ 26] = 'h00, [ 25] = 'h00, [ 24] = 'h03 => data = 3
#    [ 31] = 'h00, [ 30] = 'h00, [ 29] = 'h00, [ 28] = 'h0c => data = 12
#    [ 35] = 'h00, [ 34] = 'h00, [ 33] = 'h00, [ 32] = 'h11 => data = 17
#    [ 39] = 'h00, [ 38] = 'h00, [ 37] = 'h00, [ 36] = 'h12 => data = 18
#    [ 43] = 'h00, [ 42] = 'h00, [ 41] = 'h00, [ 40] = 'h32 => data = 50
#    [ 47] = 'h00, [ 46] = 'h00, [ 45] = 'h00, [ 44] = 'h38 => data = 56
#    [ 51] = 'h00, [ 50] = 'h00, [ 49] = 'h00, [ 48] = 'h46 => data = 70
#    [ 55] = 'h00, [ 54] = 'h00, [ 53] = 'h00, [ 52] = 'h4e => data = 78
#    [ 59] = 'h00, [ 58] = 'h00, [ 57] = 'h00, [ 56] = 'h00 => data = 0
#    [ 63] = 'h00, [ 62] = 'h00, [ 61] = 'h00, [ 60] = 'h00 => data = 0
#    [ 67] = 'h00, [ 66] = 'h00, [ 65] = 'h00, [ 64] = 'h00 => data = 0
#    [ 71] = 'h00, [ 70] = 'h00, [ 69] = 'h00, [ 68] = 'h00 => data = 0
#    [ 75] = 'h00, [ 74] = 'h00, [ 73] = 'h00, [ 72] = 'h00 => data = 0
#    [ 79] = 'h00, [ 78] = 'h00, [ 77] = 'h00, [ 76] = 'h00 => data = 0
#    [ 83] = 'h00, [ 82] = 'h00, [ 81] = 'h00, [ 80] = 'h00 => data = 0
#    [ 87] = 'h00, [ 86] = 'h00, [ 85] = 'h00, [ 84] = 'h00 => data = 0
#    [ 91] = 'h00, [ 90] = 'h00, [ 89] = 'h00, [ 88] = 'h00 => data = 0
#    [ 95] = 'h00, [ 94] = 'h00, [ 93] = 'h00, [ 92] = 'h00 => data = 0
#    [ 99] = 'h00, [ 98] = 'h00, [ 97] = 'h00, [ 96] = 'h00 => data = 0
#    [103] = 'h00, [102] = 'h00, [101] = 'h00, [100] = 'h00 => data = 0
#    [107] = 'h00, [106] = 'h00, [105] = 'h00, [104] = 'h00 => data = 0
#    [111] = 'h00, [110] = 'h00, [109] = 'h00, [108] = 'h00 => data = 0
#    [115] = 'h00, [114] = 'h00, [113] = 'h00, [112] = 'h00 => data = 0
#    [119] = 'h00, [118] = 'h00, [117] = 'h00, [116] = 'h00 => data = 0
```

Hình 4.3. Dữ liệu trong data memory ở thời điểm thực hiện xong sắp xếp

Hình 4.2, Hình 4.3 mô tả dữ liệu trong data memory lúc chưa sắp xếp và sắp xếp đã xong.

→ Tiến hành xem log ở [3], nhận thấy đã thực hiện đúng thuật toán sắp xếp nổi bọt.

CHƯƠNG 5. KẾT LUẬN

Báo cáo này đã trình bày và triển khai kiến trúc của một RISCV32I 5-stage processor áp dụng kĩ thuật pipeline nhằm tăng tốc độ xử lí, bên cạnh đó, thiết kế đã xữ lí được toàn bộ các hazard có thể xảy ra trong quá trình xử lí thực hiện lệnh của CPU bao gồm data hazard, mem hazard và control hazard. Thiết kế được tiến hành triển khai bằng ngôn ngữ mô tả phần cứng SystemVerilog và mô phỏng kiểm thử trên phần mềm Questasim. Thực hiện chạy thuật toán sắp xếp nổi bọt, cho ra kết quả hoạt động đúng với yêu cầu đặt ra. Kiến trúc có thể thực hiện 5 kiểu lệnh assembly trong tập lệnh của RISCV gồm: R-type, I-type, S-type, B-type, U-type tuy nhiên chưa thể thực hiện được tất cả các lệnh có trong tập lệnh của RISCV (xấp xỉ 40 lệnh). Trong tương lai nhóm sẽ tiến hành hoàn thiện và triển khai kiến trúc của một RISCV32I 5-stage processor hoàn chỉnh với đầy đủ các chức năng, thực hiện được đầy đủ các lệnh trong kiến trúc tập lệnh của RISCV một cách tối ưu nhất.

TÀI LIỆU THAM KHẢO

[1] P. David A. and H. John L., Computer Organization and Design RISC-V Edition: The Hardware Software Interface, Morgan Kaufmann Publishers Inc., 2017.

[2] A. Kritagya, "Assembly To Machine Code Translation," github, 2019. [Online]. Available: https://github.com/Kritagya-Agarwal/Assembly-To-Machine-Code-RISC-V.

[3] nguyenviethi, "RISC-V processor - 32 bit, 5-stage pipeline," github, 8 2022. [Online]. Available: https://github.com/nguyenvietthi/KTMT_20212/tree/main/risc_v_pipeline.